Strape SDK

A gesture-based peer-to-peer transaction system

H. Lycklama à Nijeholt & Joris Oudejans

Technichal University of Delft

TU Delft

Strape SDK

A gesture-based peer-to-peer transaction system

by

H. Lycklama à Nijeholt & J. Oudejans

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be presented and defended publicly on Tuesday July 4, 2017 at 09:00 AM.

Project duration:	April 15, 2017 – July 4, 2017	
Thesis committee:	Prof. dr. ir. R.L. Lagendijk	TU Delft, supervisor
	Dr. O. W. Visser	TU Delft
	Ir. C. Liezenberg	OK

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

This document wraps up the project that we had the pleasure of working on for 10 weeks, from the beginning of May until the end of June, 2017. The Bachelor End Project (BEP) took place in association with OK. After three years of studying at the TU Delft, from linear algebra to algorithms and from software development to fundamental logic, this project serves as the final test to complete our studies and obtain the title of Bachelor of Science.

Although the project is the final part of our bachelor's degree, it is special for us in that it encompasses our 'side-project'. On the day we received our high school diplomas we founded the company VLINDERSTORM with the goal of developing an app that would provide a *gesture-based peer-to-peer transaction system* called STRAPE. The development of this app evolved with the courses taken at the TU, as is visible in the many revisions of Strape. Taking the original Strape idea and designing it from the ground up once again with the guidance of the TU and the knowledge gained from our studies has been a comprehensive final combination of our two main interests of the past three years.

We feel incredibly privileged to get the opportunity to work on something we love and to experience entrepreneurship first-hand. We would like to thank the following people for their help and participation throughout the duration of this project and our studies at the TU Delft. Firstly, we want to thank our TU Delft supervisor, Inald Lagendijk, for his clear communication, structured meetings and sharp feedback throughout the project. Secondly, the OK product owner, Chiel Liezenberg, for his creativity, guidance and comprehensive insight in the product that enabled us to grasp the core functionality as well. Finally, as special thanks, we would like to thank Zekeriya Erkin for guiding us through the Honours Programme, showing us the academic world outside our normal studies and sponsoring many cups of coffee.

H. Lycklama à Nijeholt & J. Oudejans
Delft, July 2017

Contents

Abstract

Strape enables users to initiate and authorize peer-to-peer transactions without alphanumerical input by using gestures. OK, a smart wallet app, combines authentication, marketing and payment services in a single app. By choice, peer-to-peer payments were excluded from the initial design of the OK app, because no compelling user experience for this type of payment was available, yet. With Strape there is. To incorporate the Strape functionality in OK and other host applications, the Strape functionality needs to be made available as an SDK.

The goal of the project is therefore twofold: An SDK must be developed that provides the gesture-based functionality and adheres to a number of integration use cases for various possible hosting applications. First, it should be possible to initiate transactions by using gestures, and authorize the transactions by tapping on a discovered user and swiping it towards them. Secondly, the SDK should be structured in such a way that it is possible for a host application to fully integrate the SDK.

The system is implemented using a set of configurable classes with the business logic and a data provider that handles all data mutation and persistence. Furthermore, it holds views that can be extended by a host application for further customization. Discovery is implemented by using Bluetooth Low Energy, a protocol that enables constant advertisement and scanning of users in proximity. To ensure testability and maintainability, a Model-View-Presenter architecture is used.

The SDK is implemented in OK at various points in the app, namely as a favorite between payment accounts, IDs and tickets, as a shortcut in the main overview and as an overlay with transaction objects. OK mostly uses the standard SDK configuration variables.

The project was managed using Scrum with 1-week sprints. Version control was handled by Git and SemVer and the project was continuously integrated by using Bitrise. Although a connection to the Strape Virtual Account System was planned, time constraints and prioritization of the core Strape functionality resulted in leaving it out of the final product.

The result satisfies the project goal in that an SDK is realized comprising the core Strape functionality with the specified integration requirements. Furthermore, a first version of the SDK is integrated in the OK app, satisfying all UI integration requirements. The project has showed us the importance of the necessary overhead with software development projects such as this one. We recommend developing the requirements we deemed out of scope for this project to complete the entire peer-to-peer transaction proposition.

1 Introduction

Mobile peer-to-peer payments are increasingly occurring in society[1]. Whether it be for splitting a check, exchanging goods or paying for small services, these small transactions are part of our everyday lives. In the U.S. alone, the market for peer-to-peer payments is estimated to reach a volume of $86 billion by 2018[1]. Various companies such as Facebook[2], Snapchat[3] and PayPal[4] are pushing mobile peer-to-peer payment functionality in their apps. The way these payments are usually facilitated is by the banking or credit card infrastructure[10]. This system is the result of centuries of legal and operational development with the aim to mainly facilitate high-volume or customer-to-business (B2C) transactions. Initiating a transaction requires identification with some form of authentication and then entering the transaction data. Transaction authorization again requires some form of authentication. This secure and tedious process is to be expected for high-level asset management and customer-to-business transactions, but the system is a blunt instrument for low-volume, less sensitive transactions. For these low-risk transactions, one would expect to have an easy and instant experience, like handing over some cash.

Strape was developed to take away all the inconvenience associated with banking transactions. It aims to provide consumers with a user experience for transaction initiation and authorization that is as simple as exchanging cash, using ubiquitous mobile technologies. To achieve this, the complete process of initiation and authorization is based on gestures, rather than on data entry. Users can initiate a transaction by shaking their phone or swiping up on the screen to create a 'transaction object', e.g. a coin. Users can then pinch the object, i.e. move their fingers towards or away from each other, to set the base amount and rotate it to set the decimal amount. Other users are either discovered when in proximity or can be selected from a contact list when on a remote location. For authorization, the user taps another user and swipes ('strapes') the object towards them. Figure 1.1a illustrates the Strape app with a coin object that is pinched and rotated to display €1,25 and another user that is discovered by proximity.

The app was launched in the student community in Delft in august 2016 and received a positive response on the ease of transaction initiation and authorization. The effort did, however, also show that it is not easy to change consumer behavior. The challenge is to have them keep a separate application to hold their 'pocket money', i.e. the money they keep in their digital wallet, from applications that hold the rest of their assets. Therefore, Strape has been searching for means to deploy their patent-pending gesture-based transaction functionality in other places than a standalone app. Thus, it makes sense to package the functionality into a Software Development Kit (SDK), to be integrated into other applications.

OK is a smartwallet that combines authentication, marketing and payment services in a single app and is scheduled for November 2017 to be launched with a big media campaign in The Netherlands. With the comprehensive functionality OK is a candidate to be the next big thing in the payment industry in The Netherlands. Figure 1.1b illustrates the OK app, where the 'Birdseye View' is visible with some payment accounts and coupons. In the original OK launch plan, peer-to-peer payments functionality was excluded, because no compelling user experience for this type of payment was available, yet. With Strape there is. Therefore,

[1] Report available at http://uk.businessinsider.com/growth-in-peer-to-peer-payment-apps-report-2015-4
[2] Facebook Messenger allows users to send money to friends using their credit card
[3] Snapcash, a peer-to-peer payment service, has been launched by Snapchat in 2014.
[4] PayPal has allowed peer-to-peer payments using gestures with Bump.

(a) Strape, a peer-to-peer payment app

(b) OK, an all-in-one smart wallet app

Figure 1.1: Strape and OK

an agreement was made between Strape and OK to develop an SDK that encompasses the core Strape functionality and have OK integrate the SDK in their app. This collaboration completes the proposition of OK to include peer-to-peer payments and enables Strape to reach a larger customer audience.

Although the first host application, i.e. the application integrating the SDK, will be OK, the SDK is developed with the purpose to be easy to integrate globally in any Android or iOS app. Therefore, the project is divided in two main parts: (1) the core SDK development and (2) the integration of the SDK in OK. The core SDK development is done in a separate environment and is tested with a simple example app. The OK integration is developed in a separate Git branch of the OK code. This way, the Strape SDK can still be integrated in any other app that wishes to make use of the functionality.

In Chapter 2, the project goal is described and the Requirements are described in Chapter 3. Chapter 4 sets out the specifics of the SDK implementation and Chapter 5 those of the integration in OK. In Chapter 6, the development process is described including the used tools and decisions made. Chapter 7 assesses the result against the described requirements and Chapter 8 provides a reflection of the project and recommendations for future work on the SDK.

Appendix A provides an executive summary of the project with information and highlights. Appendix B holds the original project description. Appendix C describes the plan we made at the start of the project and Appendix D is the report of the first two weeks of the project where we only did exploratory research to prepare answering the research questions. Appendix E explains terms and acronyms used in the report. In Appendix, F the feedback is provided that the Software Improvement Group[5] gave at two separate during the project. Appendix G provides a comprehensive system UML of the SDK. Finally, Appendix H and I, the decision log and sprint retrospectives, respectively, serve as documentation for the overall project process.

[5]SIG is a consultancy company that focuses on software quality: https://www.sig.eu/

2

Project goal

The goal of the project is to create an SDK comprising the peer-to-peer gesture-based transaction initiation and authorization technology and make the technology available for integration in other applications. The technology on its own is too limited as a proposition, but many other applications require functionality for peer-to-peer transactions. Therefore, we distinguish two main challenges of the project. The first one is developing the core SDK functionality to initiate and authorize transactions. The second one is integrating the SDK into OK, the first host application.

2.1. Constraints and prioritization

Regarding the scope of the project, we must take two things into consideration. For one, the project has a time constraint. The project has a duration of 10 weeks in total, 2 for doing just research, 7 for developing the product and 1 for wrapping up. The second consideration is the level of technical challenge certain features pose. As this project is a showcase for the skills acquired and development made during our studies, the challenging features are the most interesting for research purposes. Therefore, we might prefer including those and excluding other, less technically challenging features.

Therefore, we must prioritize which research questions we want to answer and which features we want to implement. For example, displaying user information such as a profile or a list of previous transactions is trivial and therefore deemed out of scope. Functionality to up-/offload funds to the user's wallet does not fit the time frame for the project, so it is also deemed out of scope. As a third example, contact management, i.e. keeping a list of contacts including 'best friends' to select for a remote transaction, is deemed out of scope because it does not pose a real technical challenge nor fits the time frame. The functionality that is in scope is what we define as the core Strape functionality, being the initiation and authorization of transactions. Regarding discovery of other users, we will focus on discovering users that are in proximity. This method is the most challenging, as we will elaborate on in Section 4.2, and is a requirement to make the transaction authorization seamless. The core value Strape offers is the ease of use and speed at which transactions can be initiated and authorized, and this is optimal when users are in proximity of one another, as there is no need to manually select other users.

Furthermore, we discussed a lot of how we should approach the project with the client OK. The company is redesigning their app and they wanted us to prioritize that the look and feel of Strape within OK fits the current UI seamlessly.

2.2. Research question

Considering the prioritization, the main goal of the project is to answer the following question:

> **How can a mobile SDK for Strape functionality be developed that (1) comprises the gesture-based transaction functionality and (2) meets integration requirements that enable as many foreseeable use cases as possible?**

To answer (1), one must first answer:

1. How can the Strape user interface be implemented that allows for transaction initiation and authorization by using gestures only?

2. How can the app discover other users in proximity to be displayed in the user interface?

To answer (2), one must first answer:

1. What kind of host applications should the SDK target? That is, what categories of customers would be interested in integrating the Strape SDK in their app?

2. What are the various integration use cases for these host applications, i.e. how would they want to use the SDK?

3. How can the SDK be integrated in OK so that it fits the UI seamlessly?

4. And which software architecture suits these use cases?

Part (1) of the main question focuses on separating the core Strape functionality from the original Strape app and encapsulating it in a standalone SDK. This challenge consists of two main parts, as illustrated by the subquestions. Firstly, gestures must be translated to interact with the user interface. This means that the input events from the specific platforms, i.e. Android and iOS, need to be translated into transaction initiation, adjustment and finally authorization using nothing but gestures. Secondly, other users should be identifiable via some discovery method, to find possible recipients for transactions.

Part (2) is about making sure the SDK can be integrated in any host application that might be interested in doing so and integrating it in OK. Therefore, the possible customers must be determined and the various integration use cases these customers might want to implement. Furthermore, the specific integration in OK needs to be thought out regarding the UI. With these things in mind, a software architecture must be chosen that makes all these use cases possible.

3

Requirements & Scope

Considering the project goal and constraints, we define the following requirements the SDK should satisfy regarding the user experience for the end user, the integration possibilities for host applications and the software quality.

The project's scope is heavily dependent on the degree of integration with the host application, because it determines the functional possibilities for the end user. We considered three possible integration levels between the SDK and the host app, which we each explain briefly here. A more detailed explanation can be found in the research report, attached as Appendix D.

Overlay A predefined user-interface provided by the SDK is used to interact with the SDK functionality. This is the simplest form of SDK integration, because no user-interface components must be in place, but the extent to which customization of the SDK is possible is low.

Embedded The host application specifies a container in which the SDK functionality is embedded. The embedded version would also be similar to the overlay one regarding the possibility to customize, but there is more freedom as to the selection of the container.

Fully Integrated The host application is provided with elements from the SDK, e.g. the transaction objects, discovery logic and other views. The extent to which the SDK can be customized here is high as this option enables the host application to specify every small detail about the integration of the SDK.

We decided to design the SDK to be fully integratable into a host application, as per the third integration level. The main motivation for this choice was that this level allows for the most integration use cases, because the SDK can be deeply integrated into the host application and is the most open for extension and customization. Furthermore, this level of integration automatically enables the other two. Although this version is also the most complex to develop, the first host application to integrate the app, i.e. OK, already requires an extensive integration that cannot be realized by the other levels.

Considering the decision to make the SDK fully integratable, we identify three types of requirements: End user requirements, integration requirements and software engineering requirements. The SDK will be used by end-users, while being embedded in some host application. The SDK provides functionality to the end-users, but needs to integrate seamlessly with the host application in terms of user interaction and data management. Therefore, there are both functional requirements from the end-user side and requirements for the integration on the host application side. Moreover, we define a third type of requirements that are related to software engineering to ensure the project's software is maintainable, testable and extendable.

3.1. End user requirements

The main purpose of the SDK is to provide an easy-to-use interaction system for end users. Therefore, the SDK must satisfy requirements regarding user experience.

3.1.1. Transactions

The core user interaction is the gesture-based transaction functionality. Therefore, the properties described in Table 3.1 must be satisfied in the SDK regarding the initiation and authorization of transactions. The transaction amount adjustments are illustrated by Figure 3.1

(a) The transaction object is initialized by swiping up or shaking the phone.

(b) The transaction object's base value is adjusted by pinching.

(c) The transaction object's decimal value is adjusted by rotating.

Figure 3.1: The adjustment of amounts for transaction objects.

Considering the constraints as described in Section 2.1, Requirements 2, 6, 7 and 8 are deemed out of scope. These requirements need account information to be stored and edited, features that take significant time to implement but do not pose a technical challenge. Furthermore, the end user requirements all come with usability constraints. That is, they must not be less easy to use than the original Strape app, as that is the functional performance assumed by the client.

3.1.2. Discovery

For the user to be able to transact to other users, they should first be able to select them. In the case of this SDK, to send transactions, the recipient user must be visible on the screen in so the user can authorize, as described in Requirement 5. Requirements regarding the discovery of users are depicted in Table 3.2.

As elaborated on in Section 4.2, the immediate vicinity use case poses the most of a technical challenge, so we chose to prioritize Requirement 9 over Requirement 10. Furthermore, enabling the user to edit the position or remove other users from their screen is not vital functionality, but the implementation costs are significant. Therefore, we chose to deem those out of scope as well.

3.1.3. User interface design

To provide the user with all the information they need, the interface that mainly serves as a container for the transaction objects and discovered users needs to satisfy certain requirements. Although the host application may choose to leave certain parts out, the elements described in Table 3.3 are included in the SDK and we highly recommend showing them somewhere in the user interface, because they display essential information to the user. For the constraints described in Section 2.1, displaying previous transactions is deemed out of scope for the project, as displaying them is trivial but requires significant implementation costs in both the user interface and data provider.

3.2. Integration requirements

For a company that wishes to integrate the SDK, building a comprehensive integration between the SDK and the host application is important. The integration between the host app and the SDK should result in a seamless user experience. Furthermore, integration on the data layer ensures that the data used in the SDK is consistent with the rest of the app.

Table 3.1: End user requirements: The striked out numbers mean the requirement is out of scope.

#	**Requirement**
1.	The user can initiate a transaction, i.e. create a transaction object, by either shaking their phone or swiping up on it, as illustrated by Figure 3.1a .
~~2.~~	By choosing the display currency, the user can specify what currency the transaction is in.
3.	The user can set the base amount of the transaction by pinching the transaction object, as illustrated by Figure 3.1b. The higher the value, the faster the amount increases (so exponentially increasing).
4.	The user can set the decimal amount of the transaction by rotating the coin, as illustrated by Figure 3.1c.
5.	The user can authorize (send) the transaction by tapping on the desired beneficiary and swiping the transaction object towards them.
~~6.~~	The user attach a text to a sent transaction.
~~7.~~	The user can attach an image to a sent transaction.
~~8.~~	The recipient user can respond to the transaction with an emoji.

Table 3.2: Discovery requirements: The striked out numbers mean the requirement is out of scope.

#	**Requirement**
9.	The user can discover users that are in their immediate vicinity.
~~10.~~	The user can select users that are not in their immediate vicinity.
~~11.~~	The user can remove other users from their screen.
~~12.~~	The user can edit the position of other users on their screen.

First, we analyze the categories of potential host applications. Next, we analyze integration use cases these host applications might have. Finally, we derive the integration requirements from these use cases.

3.2.1. Customers

We define six categories of customers, potential companies that would want to use the SDK in their app. Each category has a use case for the SDK with distinct functionality and, therefore, integration requirements. For each category, a customer in the category has a relevant use for one or more of its services for the SDK in its app. Six categories of customers that are relevant to the SDK are shown in table 3.4, along with the application of the SDK.

3.2.2. Use cases

Each category has a specific use case where the SDK can play a central role.

A. Bank transactions The SDK can be used to initiate and receive transactions in the banking system. In this case, the bank app provides the virtual account system. The bank displays the interface of the SDK to the user and receives call-backs from the SDK when events, such as making a transaction, have occurred. The bank provides the SDK with the necessary data, such as account balance and profile information.

B. TPP transactions Third-Party service Providers (TPP) typically provide transaction functionality in a mobile application. This is similar functionality to use case 1. However, the difference is the location of the

Table 3.3: User interface design requirements: The striked out numbers mean the requirement is out of scope.

#	Requirement
13.	The wallet balance w, i.e. the total user balance b minus the sum of the amounts of all created transaction objects t: $w = b - \sum_{i=0}^{n} t_i$
14.	The user information, i.e. name and/or avatar of the current active user.
15.	In case of different currencies, such as in the Strape app, the current display currency, e.g. Euro or Bitcoin.
16.	Depending on the current state of the user interface, e.g. there are/are not transaction objects created or the user does/does not have funds, display a tip for the user on what would be a logical next step.
~~17.~~	A list of all previous transactions including transactions to up-/offload funds.

data in the virtual account system. Furthermore, TPPs might want to provide additional functionality and points of interaction with the SDK as features.

C. Mobile wallets Financial service providers can provide the ability to transact peer-to-peer or customer to business, such as online of offline checkout payments. Both functions are supported in the functional requirements of the SDK. In this scenario, several different payment objects could be displayed in the SDK. Furthermore, in the customer to business payment scenario, the SDK would need to have additional visual representation of the customer to business payment.

D. Remittance transactions Remittance services provide functionality to send funds abroad. Remittance services might want to provide their own data, and transactions will translate into specialized remittance transactions. On top of this, remittance apps may want to integrate additional discovery methods, e.g. via email address or post address to discover transaction recipients.

E. Property-management services Apps that do not handle financial transactions, but transactions that present rental agreements or ownership transfer. Instead of coins, the visual representation of these transactions should also be different. For example, a transaction can be represented as the object that is being transferred.

F. Event apps Event organizers often provide an application for events. Commercial events usually use their own tokens to exchange as currency. These tokens could have a digital representation in the event app, using the SDK. This way, tokens can easily be transferred to the point of sale or peers. Furthermore, tokens can be purchased in-app, instead of having to wait in line.

3.2.3. Requirements

By identifying integration similarities in the use cases from section 3.2.2, we define several integration requirements in table 3.6. The integration requirements are the minimum requirements that must be satisfied for a host application to be able to integrate with the SDK.

3.3. OK UI requirements

Apart from the host application-generic requirements, discussions with OK resulted in some specific requirements for the integration. These are depicted in Table 3.5.

Most of the requirements that resulted from discussions with OK fall under the host application-generic requirements of Table 3.6, but the way the SDK is integrated in the UI is specific for a host application.

3.4. Software engineering requirements

In addition to functional and integration requirements, there are several requirements with regards to software quality. The main goals for this requirement are to ensure maintainability, testability and code quality

Table 3.4: Categorization of potential customers

#	Category	Relevant service	Use case	Example
A	Banks	Banks provide financial services such as bank accounts that can be managed through their mobile applications	Provide way to initiate and manage IBAN transactions between bank accounts	ING, Rabobank, ABN Amro
B	Third-party service providers[i]	Third-party service provider mobile applications to initiate and manage transactions	Provide way to initiate and manage IBAN transactions in an external virtual account system	MyUros, an example TPP
C	Mobile wallets	Apps to manage money	Provide way to manage inter-wallet transactions	PayPal, peer-to-peer payment apps such as WieBetaaltWat[ii]
D	Remittance services	Apps to transfer money between countries	Provide way to initiate money transfers	Western Union
E	Property-managing services	Apps to manage ownership or rental of objects	Transfer object ownership or rent objects	Ticket-reselling apps (TicketSwap), digital marketplaces (Marktplaats[iii]), peer-to-peer rental services (Peerby[iv])
F	Event apps	Manage event credits	Manage digital credits for events	Share festival 'tokens'

of the project.

3.5. Prioritization

As described in Section 2.1, we must prioritize certain requirements. We have defined the core Strape functionality as the gesture-based initiation and authorization of transactions with the discovery of other users in proximity. Therefore, the core SDK development focuses on making this functionality available for integration, as illustrated by Table 3.1 and 3.2.

Just below the core functionality in priority order is the ability for host applications to integrate the SDK and after that comes the actual integration in OK.

Table 3.5: OK UI integration requirements

#	Requirement
18.	The Strape interface should be visible in between the favorite payment accounts, IDs and tickets.
19.	A shortcut should be implemented in the 'Birdseye view' that leads the user to the favorite page.
20.	When a transaction is received and the user is not in the Strape UI, an overlay should be displayed with the incoming transaction object and the sending user.

Table 3.6: Integration requirements: The striked out numbers mean the requirement is out of scope.

#	Relevant use case	Requirement
21.	A, B, C, D	The host app can display a simple SDK interface implementation with transaction objects, an account view and other users.
22.	B, C, E, F	The host app can partially or completely extend interface views, such as transaction objects, other users and the account view.
23.	B, C, D	The host app can customize parameters of the gesture-based translation algorithms.
24.	A, B, C, D, E, F	The host app can override configuration properties for SDK settings that influence the business logic.
25.	A, B, C, D, E, F	The host app can provide a virtual account system that is asynchronously connected and can be loosely coupled to the rest of the system.
26.	B, C, F	The host app can receive call-backs for SDK events based on actions happening inside the SDK.
~~27~~.	C, D, E, F	The host app can provide actions to deposit or withdraw funds in the SDK.
28.	A, B, C, D, E, F	The host app can provide additional discovery methods to identify relevant recipients to the end-user.
29.	A, B, C, D, E, F	The host app can display an SDK overlay in any location in the app, based on incoming SDK events.
~~30~~.	A, B, C, D, E, F	The host app can connect to the SDK URI scheme.
31.	A, B, C, D, E, F	The host app can connect securely to the SDK's data layer
32.	A, B, C, D, E, F	The SDK does not create any additional security vulnerabilities in the app it is contained in.

Table 3.7: Software engineering requirements

#	**Requirement**
33.	The system must be continuously tested on a continuous integration server with passing builds on all branches
34.	The system must have a test suite to test the business logic with a code coverage of at least 75%
35.	The system must have regular code reviews by at least one peer at each pull request.
36.	The system's commit log must correspond with issues from the issue tracker to keep track of development history

4

Implementation

We have created an implementation to satisfy the requirements. In this chapter, we give a detailed description of the system and the underlying decisions, while referring to the requirements in Chapter 3.

4.1. System overview (ecosystem)

To meet the integration level requirements as described in Section 3.2, the system is set up as illustrated in Figure 4.1. The host application holds the SDK as a dependency and provides it with a data provider and configuration. The data provider provides, as the name suggests, all the data the SDK depends on. User and transaction data for example, but also Bluetooth identifiers, push notifications, location-based content, etc. The data provider is also the component listening for action requests from the SDK regarding data mutation, such as authorizing a transaction or up-/offloading funds. The configuration holds all configurable variables the SDK depends on, such as curvature of transaction object adjustment, maximum number of discovered users on the screen, maximum transaction amount, etc. The SDK includes a default view implementation of the functionality, which can be displayed by the host application as a single overlay. Because the SDK provides full integration possibilities, single parts of the default implementation can be reused. The default view implementation includes views that are like the views used in the original Strape application. In this implementation, the account view is represented by a cloud, the transaction objects are represented by coins and the outlet objects by a user avatar and name. The views of the default implementation can be extended by the host application. For example, the transaction object, which is a coin, can be extended by the host application to have different colors, shapes or background images.

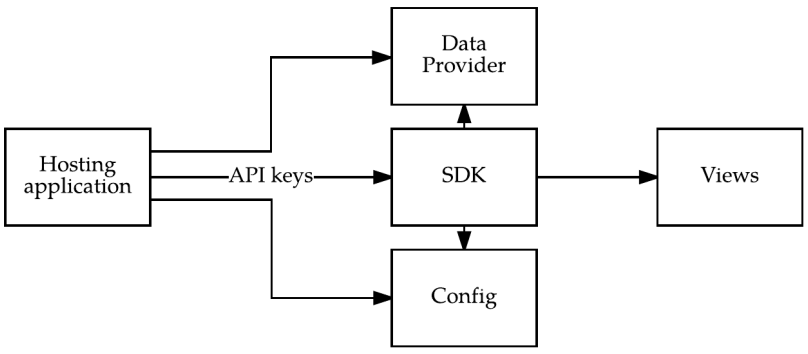


Figure 4.1: System overview

The separation of the SDK from the data provider and configuration variables satisfies the requirements as described in Section 3.2, while only exposing the parts that need to be configurable.

4.1.1. Data provider

The data provider component is responsible for all data exchange that concerns the SDK. The data provider has important implementation considerations with regards to security and data consistency between the SDK and host app, corresponding with requirements 25, 26, 27, 31 and 32. Therefore, we decided to keep the data provider as simple as possible, by providing a single data provider interface. This interface must be implemented by the host app. Using an interface makes the dependency of the SDK on the data provider very clear and concise. Furthermore, the host app is free to choose how to implement the data provider and where the data is stored. The notion that the data provider can be implemented freely in an asynchronous way is in line with requirement 25. For example, a data provider could be connected to a Virtual Account System on some server on the Internet. Or it could serve as an adapter between the host application and the SDK, where all data is essentially provided by the host application. The most important functions the data provider are depicted in Table 4.1. The data provider has three methods of providing data: It listens to *Get* requests to answer with the appropriate data and to *Post* requests, which carry some data provided by the SDK, do something with it and optionally answer with appropriate data. Furthermore, the SDK listens to push events from the data provider. Examples of push events are receiving a transaction or a balance change.

Table 4.1: Most important data provider functions

#	Method	Model	Description
1.	Get	Account	Requesting account data of current user or other users.
2.	Get	Transaction	Requesting a new transaction to be authorized later.
3.	Post	Transaction	Authorizing a transaction.
4.	Push	Transaction	Receiving a new transaction
5.	Post	Fund	Up-/offloading funds to the wallet

4.1.2. Configuration

To enable the host application to configure the SDK to its specific needs, the configuration component exposes several variables that are used by the SDK core logic. The most important configuration variables are depicted in Table 4.2. The following overview gives a more detailed description of the categories in the table:

Transaction object interaction The host application can specify the default amount, currency and size the transaction object initializes with. The trade-off here is that a larger starting size is more user-friendly in its interaction, but larger objects fill the screen faster and can make the interface less clear. Furthermore, as the host application can specify where the account view is positioned on the screen, it can also specify at which side of the view a transaction object is to be removed, or 'added back' to the wallet. Finally, the host application can either enable or disable shaking the phone to create a coin.

Transaction object adjustment This describes the constants used in the scaling algorithm as described in Section 4.6. These essentially define the curve the scaling function follows. Increasing the exponent results in faster increase at higher amounts and increasing the factor makes the amount increase faster at all values.

Other user appearance The host application can configure the sides at which other users can appear, the total other users that can be visible in the interface and the margin for each side. This can be useful for when certain parts of the interface are covered by other elements, usually at least an account view, but the host application can add custom elements as well.

Bluetooth Low Energy A more detailed overview of the discovery implementation and the reason we chose for Bluetooth Low Energy (BLE) is given in Section 4.2. As BLE is an open protocol by default, the SDK must be configured using unique identifying parameters. If all implementations of the SDK would have the same identifiers, the BLE discovery would interfere with other versions of the SDK. Therefore, it is necessary for the host application to specify a unique identifier to enable BLE discovery.

Table 4.2: Most important configuration variables

#	Type	Description
1.	Transaction object interaction	Default amount, currency and size, side of account view for removal, shake enabled.
2.	Transaction object adjustment	Exponent, factor, minimum and maximum amount.
3.	Other user appearance	Enabled sides, maximum, side-specific margin.
4.	Bluetooth Low Energy	Enabled, Service UUID, scan timeout.

4.2. Discovery

An important functionality of the SDK is the discovery of other users. Because the SDK tries to make transaction initiation and authorization as easy as possible, it must be easy for the user to choose a recipient, without having to type a recipient identifier, such as a phone number or email address. This functionality corresponds with requirements 9 and 10. The SDK can employ methods to display potentially relevant recipients to the user, using technology that is available on the phone. We refer to the process of finding relevant recipient users as discovery.

The SDK can rely on several discovery methods to find relevant users. We can distinguish between proximity discovery, finding users that are physically close by, and remote discovery, finding users that are far away. In the project, we focused on proximity discovery, as defined in the scope of the discovery requirements in Section 3.1.2.

The SDK needs to discover other users that are close by using technology that is reliable and kept as simple as possible. There are three methods that can be used to achieve proximity discovery and that are available in most Android and iOS smartphones: Wi-Fi[7], Bluetooth[9] and server-side GPS-coordinate matching. Firstly, Wi-Fi is not always available, because phones must be connected to the same Wi-Fi network to see each other, and sometimes the Wi-Fi network blocks peer discovery. Secondly, server-side GPS-coordinate requires detailed location data from the user, which requires explicit consent from the user. Furthermore, an external server and database is needed to do the location coordinate matching. Thirdly, Bluetooth Low Energy is always available, straightforward to implement, but requires the user to turn on Bluetooth functionality in their smartphone.

Considering the pros and cons of each method, we decided to use Bluetooth, as it is the most practical and available technology. It is always available and requires the least user permissions.

The Bluetooth Low Energy standard is a protocol that allows device discovery in an energy-efficient way. A BLE device can advertise one or more services, identified by a Universally Unique Identifier (UUID), a 128-bit number that guarantees uniqueness across space and time[24]. Each service can have one or more characteristics, which are also identified by UUIDs and store a value than can be read or written by other devices. The SDK creates a service with several characteristics for the discovery functionality. At the same time the SDK scans for the SDK service on nearby BLE devices. Scanning the environment for other users is battery-intensive, so a timeout must be set for every time the SDK starts scanning. The maximum of this timeout is 10000 milliseconds (10 seconds) as to not drain the battery.

4.3. Software Architecture

The general architecture of the code should be suitable for the system as described in Section 4.1, while keeping the system maintainable and allowing testing of vital components. Because the SDK has extensive integration requirements, another important factor to consider was the extendibility of the architecture, i.e. components should be loosely coupled so they can be replaced with other implementations. We decided to base the structure of the code on the Model-View-Presenter (MVP) design pattern. The decision process and motivation is described briefly. A more detailed explanation is given in the research report and can be found in Appendix D.

4.3.1. Model-View-Presenter

The Model-View-Presenter design pattern (MVP) has been introduced in the nineties at Taligent[30]. It is based on the Model-View-Controller (MVC) architectural pattern, which is widely used in a multitude of different ways and on different platforms. However, this pattern has shown several limitations in terms of software testing. This is because traditionally in an MVC architecture, the view is tightly coupled with the model. That is, it interacts directly with the model. Furthermore, the MVC pattern can create problems for complex user interfaces, because the logic is not separated from the view.

The MVP architectural pattern attempts to improve the testability of the system by moving all application business logic to a different component: the presenter. In the MVP architecture, the components are defined as follows:

The model is responsible for managing all application data.

The view displays information from the presenter and connects user input events to the presenter. The view contains as little application logic as possible.

The presenter is responsible for connecting the model to the view and performing all business logic, including data formatting and user input processing.

The MVP pattern is a suitable architecture for our system that has a complex user interface, because it improves the testability, maintainability and extendibility of the system in comparison to the MVC pattern. The reason for this is that it isolates the business logic in the presenter, meaning each component has their own function. The view component is kept as 'thin' as possible, meaning that it contains as little application logic as possible. The presenter component interacts with the model and the view using interfaces. This makes the presenter easier to test, because mock[26] classes can be made for both the model and the view. Furthermore, the presenter and views can be easily extended because both have a clear role in the system.

4.3.2. Mobile user-interfaces

Both iOS and Android contain similar components to build a mobile user-interface. These components include Views, for both platforms, and UIViewControllers[5] and Activities[14], for iOS and Android respectively. This structure can be compared to the MVC architecture, because the UIViewControllers and Activities are generally used to represent user-interface screens with multiple Views, all controlled by the controller component.

Complex user-interfaces like the Strape interface require a more structured approach. Instead of handling all the logic of one 'screen' in one controller, we must split the screen up in smaller components. These components can each be managed by their own presenter. The user-interface is split up in several sub views, each with their own presenter. To implement this idea in both mobile platforms, this means that the traditional controllers, be it UIViewControllers or Activities, are contained in the *View* component of the MVP model. In addition to this, a separate class is created for the *Presenter* functionality.

4.3.3. Structure

The SDK structure is subdivided into several components that are based on the Model-View-Presenter design pattern. Firstly, the abstract structure of the SDK that identifies the relationships between presenter, view and model components. Secondly, a default implementation of these interfaces to implement the main SDK functionality. Thirdly, several utility classes to manage settings, animations and events. The structure is shown in Figure 4.2. Furthermore, a UML overview can be found in Appendix G.

The first layer of classes that define the core functionality of the SDK are entirely defined as abstract interfaces. This way, the SDK is extendable and testable by design. The PLAYGROUNDPRESENTER is used to display the main PLAYGROUNDVIEW that handles all transaction objects, other users and the account. On the PLAYGROUNDVIEW several sub views are displayed. These are the TRANSACTIONOBJECTVIEW, for displaying transaction objects such as coins, OUTLETVIEW to display other users, and one ACCOUNTVIEW to display account information. All views are managed by their own presenter, TRANSACTIONOBJECTPRESENTER, OUTLETPRESENTER and ACCOUNTPRESENTER, respectively. Furthermore, the views are implemented using the native view system of both mobile platforms. Furthermore, the TRANSACTIONOBJECTPRESENTER uses a TRANSACTIONOBJECTADJUSTER to handle the different types of transaction object adjustments that occur when an object is interacted with by a user. The TRANSACTIONOBJECTADJUSTER is dependent on a specific SCALINGALGORITHM implementation, which by default is the EXPONENTIALALGORITHM implementation.

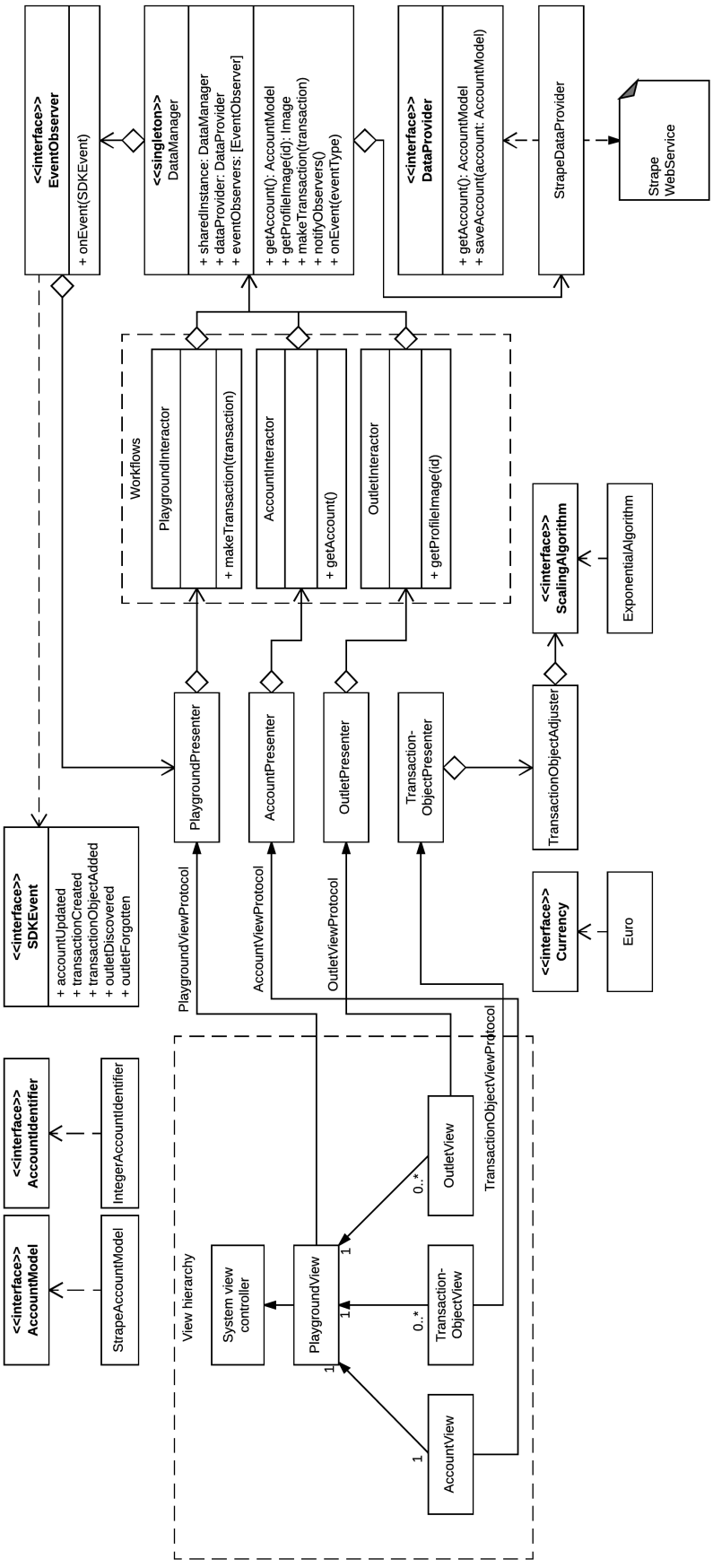


Figure 4.2: Overview of the structure of the SDK implementation

When considering Figure 4.2, it is apparent that for every sub view two references exist. The first reference is from the parent view, which contains the view in the operating systems view hierarchy. The second reference is from the presenter that is coupled with the view. Like the systems view hierarchy, the SDK contains a presenter hierarchy.

Implementation
The second layer of core classes implement the abstract structure as described in Section D. These classes provide a standard functional implementation of the SDK. This implementation is that of the "cloud UI", using CLOUDPLAYGROUNDPRESENTER/VIEW and CLOUDACCOUNTPRESENTER/VIEW, with coin objects, using COINTRANSACTIONOBJECTPRESENTER and COINVIEW.

4.4. Testing

The Model-View-Presenter architecture makes components testable, because it separates the business logic from the view code. Parallel to the implementation process, a test suite was developed that immediately verifies the functioning of newly added code. The guidelines of this system were based on the Test-driven development (TDD) process[8]. In TDD, a test case that verifies the implementation is written before the implementation itself. The idea is that the test helps to work out the specification of the functionality, which improves the implementations quality. Furthermore, it is trivial to see that the implementation is successful. While we did not see TDD as the only correct form of development, we adhered to it by writing automated tests for new functionality within the same development cycle. This way, new functionality could easily be verified. The SDK's test suite consists of unit and integration tests for the presenter and utility classes, in which views are represented using mock classes.

4.4.1. Testing BLE

One case of testing we want to feature is testing Bluetooth Low Energy on Android, as it poses some significant challenges. Although most tests implemented are Unit tests[11], BLE management on Android requires native classes that cannot be mocked[26]. Therefore, the Bluetooth manager is tested by using instrumented tests[16]. This kind of testing enables the use of native Android instruments in the process, e.g. the native Bluetooth Low Energy scanner and adapter. Furthermore, using Bluetooth on Android requires permissions to be given by the user since API 23, Android 6.0 Marshmallow[17]. The instrumentation provides a method to assume the permission for the tests.

Another challenge of testing the BLE functionality is that all the native Android classes concerning it are *final*, i.e. they cannot be inherited from by other classes. Mocking relies on extending these classes, so these one's cannot be mocked. A solution for this problem is to use the DECORATOR design pattern[27]. This effectively wraps the final classes in non-final one's and passes through the methods used by the Bluetooth manager. The non-final classes can be mocked, so this results in the Bluetooth manager being testable.

4.5. Security

The SDK provides functionality that is connected to sensitive data and important actions, for example authorizing payment transactions. It is thus important that the SDK is designed with security principles in mind. A system can never be completely secure, but one should take security into consideration when designing the system to minimize the potential threats and their impact. We have identified several potential vulnerability scenarios in the SDK that could be exploited by an adversary.

1. An adversary gains access to data stored by the SDK.
2. An adversary interacts with the SDK that results in unauthorized actions.
3. An adversary interferes with the availability of the SDK, also known as Denial-of-Service attack.

We have designed the data provider to mitigate scenarios 1 and 2. An important factor for data security is the origin of the data used inside the SDK. A reason to design the data provider as a single programmatic interface, is that the complexity is low. Using a local interface inside the app, it is up to the host application to manage the data loading, for example locally or over the network. The data is exposed through this local interface to the SDK, which means the SDK has no external connection to the data source. Furthermore, all communication with the data provider goes through the same interface. This way, data inside the SDK can only be compromised when the complete applications code and data are exposed.

The SDK is embedded in the host application, which is embedded in an application sandbox of the mobile operating system[3][18]. The sandbox ensures the isolation of all app data and code, which means the SDK data and code is inaccessible to the rest of the operating system. Thus, scenario 3 is also unlikely, because all SDK code execution is done in this sandbox and only becomes available when the app as whole is unavailable. Nevertheless, it can be possible to access data and code of other apps when an application gains root access rights on the device. However, this is only possible when the device is *rooted* or *jailbroken*, which usually requires manual action by the device's owner. The risk of this scenario is limited, because when a device is rooted the security of all apps is compromised and the owner is often aware of the security consequences.

The design of local data transfer via the data provider interface keeps all data inside the application, which is protected using a sandbox by the operating system. Therefore, unauthorized access, unauthorized interaction or Denial-of-Service attacks are not additionally likely due to the SDK. This satisfies Requirement 32, which states that the SDK should not create any additional vulnerabilities for the host application it is integrated in.

4.6. Algorithms

For translating the user movements into the amount values several algorithms for specific stages of the translation process are implemented. The gestures for the amount adjustment are illustrated by Figure 3.1. The structure of the implementation of these algorithms is illustrated by Figure 4.3. All gestures concern specific transaction objects, so one can adjust the number of individual objects separately.

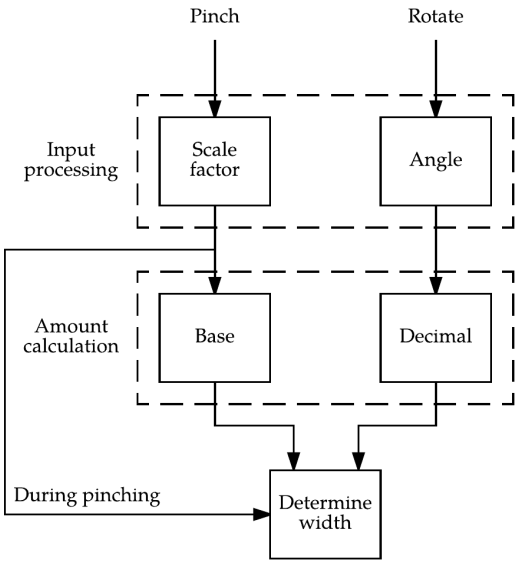


Figure 4.3: Transaction object adjustments

The system, whether it is Android or iOS, calls either a pinch or a touch event which is then passed on to the listeners registered by the SDK. These events are processed to values that can be used to calculate the base and decimal amounts. The combination of the calculated base and decimal values finally results in the width of the transaction object. During pinching the determination of the width results directly from the scale factor to make sure the width is adjusted in a linear fashion instead of in steps every base change. On releasing, i.e. the event a user stops touching the screen, the width is snapped to represent an integer base value. The following sections describe these algorithms in more detail.

4.6.1. Equations

The variables used in the equations that concern the transaction object are illustrated by Figure 4.4.

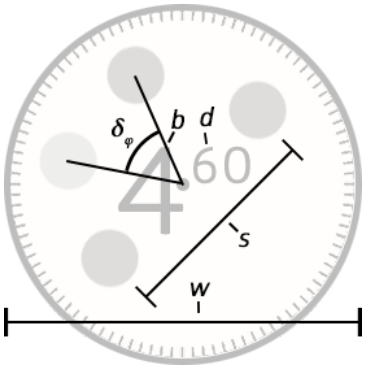


Figure 4.4: The transaction object variables used in the equations: Relative angle δ_ϕ, base b, decimal d, span s and width w.

Scale factor
When scaling, i.e. when the most right and bottom touch points in Figure 4.4 are moving towards or away from one another, the scale factor f is calculated. This factor is later used to determine the base value of the transaction object and depends upon the span s, the distance between the two touch points:

$$f_n = f_{n-1} + \frac{s_n - s_{n-1}}{s_0} \tag{4.1}$$

So, the scale factor is derived by adding the difference between the current span and the previous span divided by the starting span to the previous value of the scale factor. Equation 4.1 always grows relative to the touch event starting position. Therefore, it keeps a constant increase. More on this function can be found in the research report attached in Appendix D.

Angle
When tracking rotation movements, we want to calculate the relative angle between the previous touch event and the current one δ_ϕ. The current angle is calculated by taking the arc tangent of the x and y coordinates of the touch event relative to the center of the object:

$$\phi_n = \arctan\frac{y_n}{x_n} - \arctan\frac{y_c}{x_c} \tag{4.2}$$

For retrieving the difference between the current angle and the previous angle, in most cases we can just do $\delta_\phi = \phi_n - \phi_{n-1}$. But when the phase wraps 2π, we need to adjust it to represent the appropriate relative value. So if the previous angle $\phi_{n-1} > \frac{\pi}{2}$ and the current angle $\phi_n < -\frac{\pi}{2}$ or the other way around, $\delta_\phi = \pm|2\pi - (|\phi_n| + |\phi_{n-1}|)|$, positive or negative, respectively. This procedure always results in the appropriate relative value between the previous angle and the current one.

Base
To calculate the base value resulting from the scale factor, parameters are used that can be set by the host application, as described in Section 5.3. These parameters are:

- Multiplier $m > 0$, default 1, to adjust the overall calculated values to a certain range. For a host application with higher transaction values on average, m can be higher than for applications with low transaction values.
- Exponent e, default 4, to adjust how much faster the amount increases for higher amounts. Configuring e to be higher makes it easier to specify the amount to high values, but makes it more difficult to precisely specify smaller amounts.

Given these variables and the scale factor f as derived in Section 4.6.1, the base value of the transaction object amount b is calculated as follows:

$$b = \lfloor m \cdot f^e \rfloor \tag{4.3}$$

To illustrate this with an example, let the parameters have their default values. The base values for various scale factors is then given in Table 4.3.

Table 4.3: Base calculation

Scale Factor	Base
1	1
1.4	3
1.7	8
2	16
2.3	27

Decimal
The decimal number of the amount is based on the angle as calculated in Section 4.6.1. The equation to derive the change in decimal δ_d from the change in angle δ_ϕ given the number decimals in one base unit o is as follows:

$$\delta_d = \lfloor \frac{\delta_\phi \cdot o}{2\pi} \rfloor \tag{4.4}$$

Determining the width
With the amount value, the width of the transaction object can be derived. The idea is that this width is a continuous function of this amount value, which is a combination of the base and decimal values. To calculate the width w, the equation is similar to the inverse function of the base value calculation as depicted in Equation 4.3. Given the configuration parameters as described in Section 4.6.1 and the amount a, the width is calculated as follows:

$$w = \sqrt[e]{\frac{a}{m}} \tag{4.5}$$

4.6.2. Algorithms

Using the derived equations for each step in the process, algorithms are set out that result in the desired behavior for resizing and rotating each transaction object.

The algorithm called when a resizing touch event is received is depicted in Algorithm 1. This algorithm depends on the following input variables at the nth event:

- f_{n-1}, the scale factor calculated in the previous touch event.
- s_0, s_{n-1}, s_n, the start, previous and current span of the scale gesture.
- Multiplier m, exponent e, minimum amount a_{min}, maximum amount a_{max}, the configuration parameters for the scale gesture.
- d, the current decimal value.
- u, a boolean that is true iff the event is the user releasing, i.e. stopping the scale gesture.

It returns the width the transaction object should present and the amount value to be displayed.

Algorithm 1 Resizing the transaction object

$f \leftarrow f_{n-1} + \frac{s_n - s_{n-1}}{s_0}$
$a \leftarrow \max(a_{min}, \min(a_{max}, \lfloor m \cdot f^e \rfloor + d))$
if u = true **then**
    $w \leftarrow \sqrt[e]{\frac{a}{m}}$ ▷ If the gesture ends, snap to the width of the integer base
else
    $w \leftarrow f$ ▷ Otherwise, scale the object linearly with the gesture
end if
return (w, a)

The algorithm for calculating the decimal value and the appropriate width is depicted by Algorithm 2 and depends on the following input variables:

- $(x_c, y_c), (x_n, y_n)$, the center coordinates of the transaction object and the current coordinates of the drag gesture.
- $\phi n - 1$, the previous angle.
- Global variable r, an accumulator for the current rotation.
- Multiplier m, exponent e, minimum amount a_{min}, maximum amount a_{max}, the configuration parameters for the scale gesture.
- o, the number of decimals in one base unit.
- b, the number of units fit in one base unit.
- a_{n-1}, the previous amount.

As with Algorithm 1, it returns the width the transaction object should present and the amount value to be displayed.

Algorithm 2 Rotating the transaction object

$\phi_n \leftarrow \arctan\frac{y_n}{x_n} - \arctan\frac{y_c}{x_c}$
$\delta_\phi \leftarrow 0$
if $\phi_n < -\frac{\pi}{2} \wedge \phi_{n-1} > \frac{\pi}{2}$ **then**
  $q \leftarrow |2\pi - (|\phi_n| + |\phi_{n-1}|)|$
  if $\phi_n < -\frac{\pi}{2}$ **then**
    $\delta_\phi \leftarrow |q|$
  else
    $\delta_\phi \leftarrow -|q|$
  end if
else
  $\delta_\phi \leftarrow \phi_n - \phi_{n-1}$
end if
$r \leftarrow r + \delta_\phi$ ▷ Global variable that accumulates the current rotation.
$t \leftarrow \frac{2\pi}{o}$ ▷ Threshold rotation to change the amount.
$s \leftarrow \lfloor \frac{r}{t} \rfloor$ ▷ The number of steps to change the amount by.
$v \leftarrow \frac{b}{o}$ ▷ The amount a single step represents.
$a \leftarrow a_{n-1}$
if $|s| > 0$ **then**
  $i \leftarrow s \cdot v$ ▷ Proposed increase in amount
  $r \leftarrow r - s \cdot t$ ▷ Decrease accumulator by the rotation handled in this iteration of the algorithm.
  $a \leftarrow \min(a_{max}, a_{n-1} + i)$ ▷ Set amount to either its max value or add the increase.
end if
if $a < a_{min}$ **then**
  $a \leftarrow a_{min}$ ▷ Set amount to the minimum
  $r \leftarrow 0$ ▷ Reset the accumulator
end if
$w \leftarrow \sqrt[e]{\frac{a}{m}}$
return (w, a)

4.7. Platform differences

The SDK has been implemented for two mobile platforms: iOS and Android. While the requirements are the same for both platforms, there are implementation differences. Most differences are not significant and come down to details. However, we have identified two components that differ between the platforms.

One difference is the way the system returns the scale factor for pinch gesture events. For Android, the scale factor is relative to the previous event call-back, whereas on iOS, the scale factor is relative to the very first pinch event call-back. This means the growth rate of the Android scale factor will keep increasing when

the user is pinching, whereas on iOS this growth rate remains constant. To remedy this fact, we decided to translate the Android scale factor to be like the iOS scale factor, meaning relative to the first pinch event call-back. This way, the same scale factor algorithms can be used on both platforms. More explanation and illustrations on this difference can be found in the research report in Appendix D.

Secondly, both systems have a significantly different view system. The user interface of the SDK is displayed to the user using the system view hierarchy. This hierarchy is part of a view system that is different for both platforms. The Android view system was designed to support a wide range of different devices with different screen sizes[21]. In contrast to Android, the iOS view system originally relied on absolute placing of views using a coordinate system, because only the screen size of the original iPhone was supported. Over time, the iOS view system[6] has been extended to support different screen sizes, but it still has a different approach than to the Android view system. Because of this difference in user interface architecture, the implementations of the SDK were different for iOS and Android. However, due to the choice of using the Model-View-Presenter (MVP) architecture pattern, this difference was isolated to only the view components of the SDK, as the MVP pattern defines a clear separation between components.

Regarding software architecture, both platforms allowed us to use the same design patterns and roughly the same structure. The Android implementation was programmed using Java[22] and the iOS version using Swift[23]. Java is significantly older than Swift, but both languages are object-oriented and have similar features. Apart from the differences in gesture event input and the view system, both platforms therefore allowed for a similar implementation of the SDK.

5

OK Integration

As the first host application, OK serves as the baptism of fire for the SDK. The integration is implemented in the recommended way, i.e. following the SDK documentation. The SDK is initialized on start-up and starts scanning and advertising immediately. An instance of the playground view, which holds the transaction objects, other users and the user account information, is initiated as a screen amidst favorite payment accounts and IDs.

First, considerations raised by the OK product owner need to be taken into account. Secondly, the user interface is defined so that the SDK fits neatly in the current OK application and finally the specific configuration of the SDK by OK is described.

5.1. Considerations

When considering Table 3.4, we assign OK to category C, Mobile wallets, as that is how OK wants to use the Strape SDK. All requirements depicted in Table 3.6 are therefore relevant for OK.

As described in Section 5.2.1, the playground view is included in a pager. A pager can recycle instantiated views and removes them from memory when that memory is necessary for some other data. This means that the playground view should be prepared to be re-instantiated at random times. Therefore, the state of the playground should be persistent, e.g. the transaction objects, visible outlets and their positions on the screen.

A feature particular for OK is the integrated scanning functionality. One of the main components of OK is the QR-code scanner, which serves as an interface between the app and counterparties. As the Strape SDK uses QR-code scanning as a fall-back method to discover other users, an integration of the Strape URI scheme should be implemented in the OK scanner. This is an extension of Requirement 27, which is out of scope for the project. Therefore, the integrated scanner functionality is also out of scope, but the feature is subject to further development.

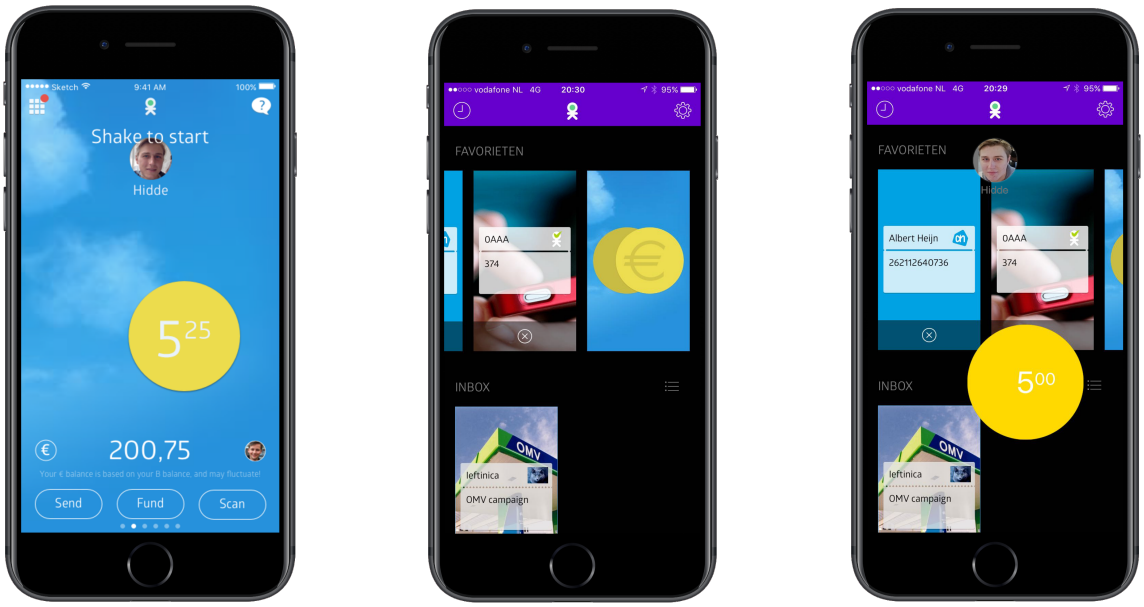
5.2. User interface

The SDK is integrated in the user interface at three locations in the app, following the OK UI integration requirements as depicted in Table 3.5:

1. In the favorites details screen, as a page between favorite payment accounts and IDs, illustrated in Figure 5.1a.

2. In the 'Birdseye view', the overview of all content, a tile is added that serves as a shortcut to the page described above, illustrated in Figure 5.1b.

3. When the user is in the Birdseye View and receives a transaction, a transaction object is laid over the current content, illustrated in Figure 5.1c.



(a) Playground view as a favorite

(b) Shortcut in the Birdseye view

(c) Overlay of a received transaction

Figure 5.1: UI integration of the SDK in OK

5.2.1. Favorites page

To make sure the user can access Strape within one touch when OK is opened, the main view, i.e. the playground view that holds all transaction objects etc., is included as a favorite. The favorites are displayed in a pager that is opened on start-up of the OK app. For the pager is cyclic, Strape is added as the last item in this adapter to only require one drag to the right to access it. Apart from the various UI elements, the main view visible is the playground view, which takes up the entire screen, as illustrated by Figure 5.1a. All other elements are laid on top of it.

The following descriptions of the UI elements all concern the display of information and action flows that are not in scope for the project, but that will be implemented by further development. For the project, only the UI elements themselves are implemented and not the underlying features. The user balance is displayed in the bottom center along with the current display currency on the left and the user avatar on the right. If the current display currency is not equal to the backing currency, a small text is displayed that informs the user that the displayed balance can fluctuate as the exchange rate does. In the case of OK, the backing currency is in Bitcoin and the display currency is in Euro's.

The top two buttons are native in OK. The grid icon on the top left one returns the user to the Birdseye view, while the top right one opens an overlay where the user can contact support. The user can choose a display currency by clicking the button with the currency symbol in it, here '€'. For changing profile information, e.g. name and avatar, the user can click their avatar. The 'send'-button opens a list of contacts to be selected by the user that will be displayed on the screen on selection. The 'fund'-bottom leads the user to a screen to either up- or offload funds to their wallet. Finally, the 'scan'-button opens the OK scanner which scans for other users' QR-codes.

5.2.2. Birdseye view

The Birdseye view consists of various rows of tiles, e.g. rows of payment accounts, coupons or tickets. The tiles are abstracts of the underlying information. Coupons, for example, show the campaign name and the amount of discount associated with them. When clicking on a tile, the interface 'zooms in' to display one tile in full-screen with all the details of the model. The other items in the row can then be reached by dragging left or right in a pager, as also described in Section 5.2.1. Furthermore, the top row shows the items the user defined as favorite. Following the position of the SDK interface between the favorites, a tile displaying two coins is included in the favorites row in the Birdseye view as well, as illustrated in Figure 5.1b. Clicking this tile brings the user to the full-screen playground view as illustrated in Figure 5.1a.

5.2.3. Overlay

As the app is always advertising the user, it is possible that they are discovered by other users when the playground view is not visible. If, in that case, a transaction is received, there is no playground view present to display the incoming transaction object. Therefore, in the OK app, an overlay is shown that displays the user that sent the transaction and the corresponding transaction object, as illustrated by Figure 5.1c. If clicked anywhere that is neither the other user nor the transaction object, the overlay disappears. The same happens when the transaction object is swiped downwards out of the screen, a gesture in line with how a transaction object would be added to the account in the playground view.

5.3. Configuration

As described in Section 4.1, apart from extending the views and specifying the data provider, the host application can configure the SDK by providing several configuration variables. The values of these variables are depicted in Table 5.1. As one can see, every variable is configured to use the default SDK value, except for the side-specific margin for other users. This is because, as illustrated by Figure 5.1a, there is a bar at the top of the screen, which would cover the other users that appear there. With the extra top margin the other user appears just below the bar.

Although the configuration variables are almost all configured to their default values, our expectation is that OK will want to tune them when they receive their first user feedback regarding the interaction. For example, once it is apparent what range of transaction amounts users will specify the most, Variables 6, 7 and 8 should be tuned in a way it is the easiest for a user to specify an amount in that range.

Table 5.1: OK configuration

#	**Variable**	**Value**
1.	Default transaction object amount	100 (default)
2.	Default display currency	Euro (default)
3.	Default transaction object size	150 pixels (default)
4.	Side of account view for removal	Bottom (default)
5.	Shake to create transaction object enabled	Yes (default)
6.	Exponent for scaling algorithm	4 (default)
7.	Factor for scaling algorithm	1 (default)
8.	Transaction object minimum amount	1 (default)
9.	Transaction object maximum amount	∞ (default)
10.	Enabled sides for other users	Left, top, right (default)
11.	Maximum number of other users on screen	9 (default)
12.	Side-specific margin for other users	Top: 50 pixels
13.	BLE enabled	Yes (default)
14.	BLE Service UUID	Default
15.	BLE scan timeout	10000 ms (default)

6

Process

The project goal of creating an SDK that can integrate with many kinds of host applications can be implemented in multiple ways. To translate the project's requirements into a concrete implementation, implementation decisions had to be made during the project. To keep track of these decisions and their impact, we used a structured project approach. Furthermore, because this is a software development project, we took extra care to pick the right tools for code management and analysis to keep track of the project's progress and the code's quality. Although the process was the same for both platforms, it is notable that Joris implemented the SDK on Android and Hidde on iOS.

In this chapter, we start by describing the project management approach. Next, we give an overview of the software development conventions and tools. Finally, we look at the planning deviations during the project.

6.1. Project management

The project management is defined by a set of principles, conventions and processes. A Scrum-based approach was used to manage the project. The project was split up in weekly sprints, with deliverables for every sprint. This way, an installable product was already available at the end of the first sprint. Figure 6.1 shows the SDK in the example app after the first sprint.

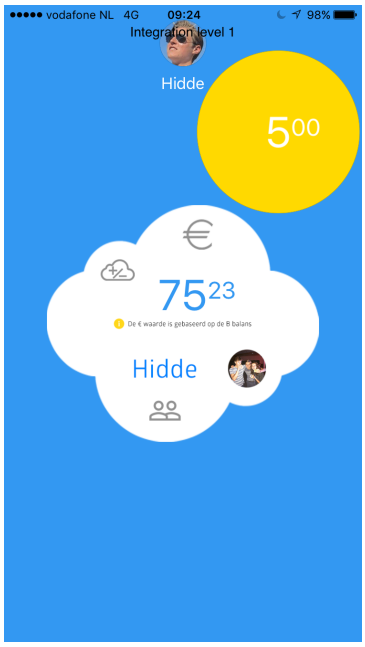


Figure 6.1: Result of sprint 1: SDK embedded in the example application.

Documentation of the project was done using Confluence, an online content collaboration tool. In Con-

fluence, all implementation documentation, planning, meeting notes, decisions, retrospectives and guidelines were recorded. This way, all the content of the project was in one place and accessible to all the project team members, including the client and university supervisor. Furthermore, content in Confluence could be linked to issues in JIRA, which improved the connection between the documentation and the software itself.

6.1.1. Sprint cycle

The JIRA[1] issue tracker was used to manage and document the Scrum process. The tool provides a collaborative system to manage a software project. Every sprint was executed according to the following cycle.

1. **Collection** Issues are added to the backlog.
2. **Sprint grooming** Go through the issues and provide a more detailed description, assign an epic, a release version and story points.
3. **Sprint planning** Define the next sprint with all relevant issues.
4. **Sprint execution** Start the sprint and work on the issues.
5. **Sprint retrospective** At the end of the sprint, look at the finished and unfinished issues and determine the status of the progress. Create a retrospective with an overview of the results, what went well and what went wrong. Move all unfinished issues to the next sprint.

6.1.2. Versioning

For every sprint, an incremental version was used, following the Semantic Versioning[2] system. Semantic versioning is a set of rules and requirements to convey meaning through version numbers about the underlying code and what has been modified from one version to the next. In the SDK, a version is of the format *MAJOR.MINOR.PATCH*, where

- *MAJOR* version is incremented in principle with every functional non-backwards-compatible change. Over the course of this project, this is unlikely to happen. However, we have defined the *MAJOR* version to be 0 throughout the development phase of the project and the end product version to have *MAJOR* version 1.
- *MINOR* version is incremented with every new functionality that is backwards-compatible. Typically, new functionality that introduced in normal sprints will increment the *MINOR* version.
- *PATCH* version is incremented with every backwards-compatible bug-fix.

Taking the versioning system into consideration, we have defined the versions as shown in table 6.1. Each version indicates the functional improvements from the next iteration. At the end of the project, a major release is made, as marked by version *1.0.0*.

6.2. Software development tools

The usage of suitable software development tools and guidelines can greatly improve a project's code quality, maintainability and efficiency. Therefore, we have spent considerable time on setting up an appropriate workflow for development.

6.2.1. Repository guidelines

Git[3] was used as a version control system. Git is an open source tool for tracking changes to files with multiple people and is often used for software development. In a git repository, file versions are tracked using *commits*, the tracking points in the history of the repository. Commits track the state of one or more files in the repository, together with a commit message describing the changes. Furthermore, branches can be used to work on features without breaking the code. Git provides the necessary functionality of branching and merging code versions that fits with our collaborative workflow. We have adhered to the following repository guidelines for development:

[1] JIRA is a tool to track the development of software projects and is developed by Atlassian. `https://www.atlassian.com/software/jira`

[2] https://semver.org

[3] Git is a distributed version control system. `https://git-scm.com/`

Table 6.1: Release versions

Version	Description
0.1.0	Set up project architecture
0.2.0	Reference view implementation with coins, outlets and cloud
0.3.0	Business logic implementation, as well as discovery via Bluetooth Low Energy
0.4.0	Transaction handling and data provider system
0.5.0	Integration with OK application, customized OK views for SDK
0.6.0	New features as a result of OK integration, app-wide SDK overlay for events and architecture improvements
1.0.0	Final product demo

1. Every feature that is related with a scrum story is developed on a separate, so-called feature-branch.
2. Feature-branches are merged into the development branch via a Pull-Request (PR), which must be reviewed and approved by at least one person that is not the creator of the PR. Before the PR is merged, the test suite that is run by continuous integration must pass for the PR, as described in section 6.2.2.
3. For every release, the development branch is merged into the master branch. The master branch is then tagged with the appropriate version number, according to section 6.1.2.
4. A commit message includes the relevant issue identifiers from the issue tracker.

Separate repositories for iOS and Android were used, because the two versions of the SDK are designed for different platforms and are different projects. Bitbucket was used to host the repository on the internet. This serves as a backup and a central management point to integrate with other online services, such as continuous integration.

6.2.2. Continuous integration, delivery and deployment

Continuous integration is a key component of any professional software engineering project. It ensures that the project's code is always in a working state and notifies when any breaking changes are made to the code. For Android and iOS, continuous integration use to be a bit of a challenge, because setting up a mobile testing environment as a service is not as straightforward. We decided to go with Bitrise[4] as continuous integration server, because it allows users to define extensive workflows for their testing. Bitrise was used for continuous integration, but also allowed other functions, such as automatic app generation and delivery, based on certain business rules. We defined two separate workflows, one for continuous integration and one for deployment.

Continuous integration

The continuous integration workflow was used to make sure no breaking changes were included in commits. This workflow was invoked on every git push, on any branch in the repository. The workflow executed the following actions:

1. Git checkout the new version in the repository
2. Generate documentation[5]
3. Compile source code with debug configuration and execute tests

The continuous integration step makes sure the code is always compiling and the tests are passing. If not, the team is immediately notified of any errors. Furthermore, the documentation of the project is always up-to-date with the code. An example of the documentation is shown in figure 6.2.

[4] Bitrise is a continuous integration and delivery service for mobile platforms. `https://www.bitrise.io/`

[5] Documentation was generated using Gradle for Android (`https://gradle.org/`) and Jazzy for Swift (`https://github.com/realm/jazzy`

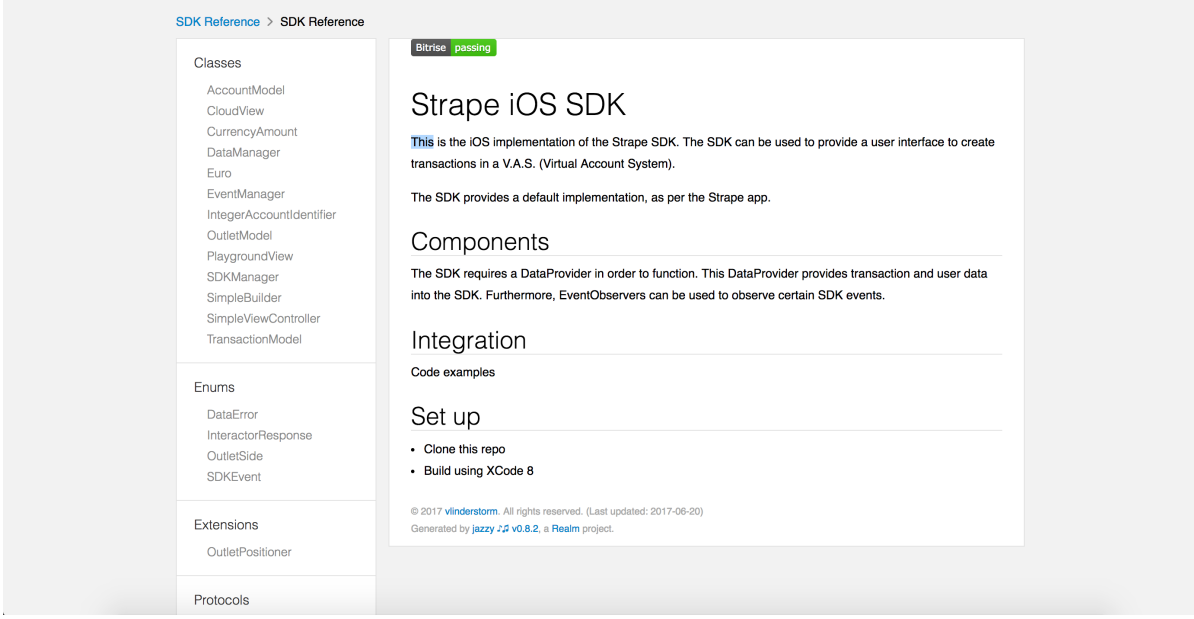


Figure 6.2: Strape SDK documentation

Continuous deployment

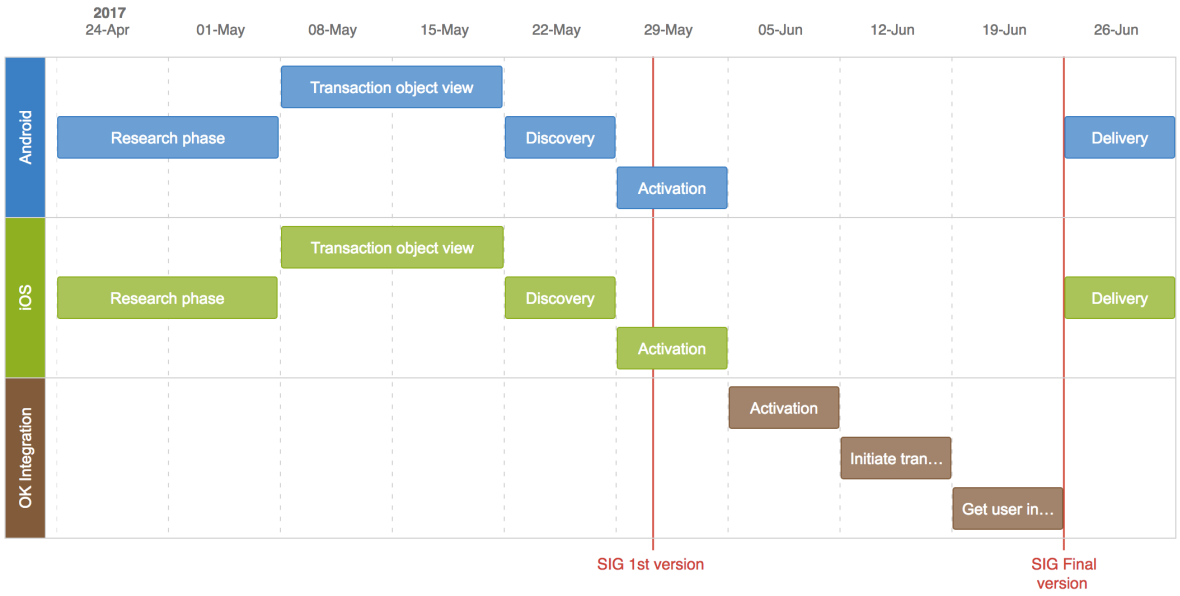
The deployment workflow was used to deploy a release version of the code, and was invoked when any commit with a Git tag was pushed to the repository, as explained in section 6.1.2.

1. Git checkout the new version in the repository
2. Generate documentation using Jazzy
3. Compile source code with production configuration and execute tests
4. Generate archive file
5. Send email with new app version.

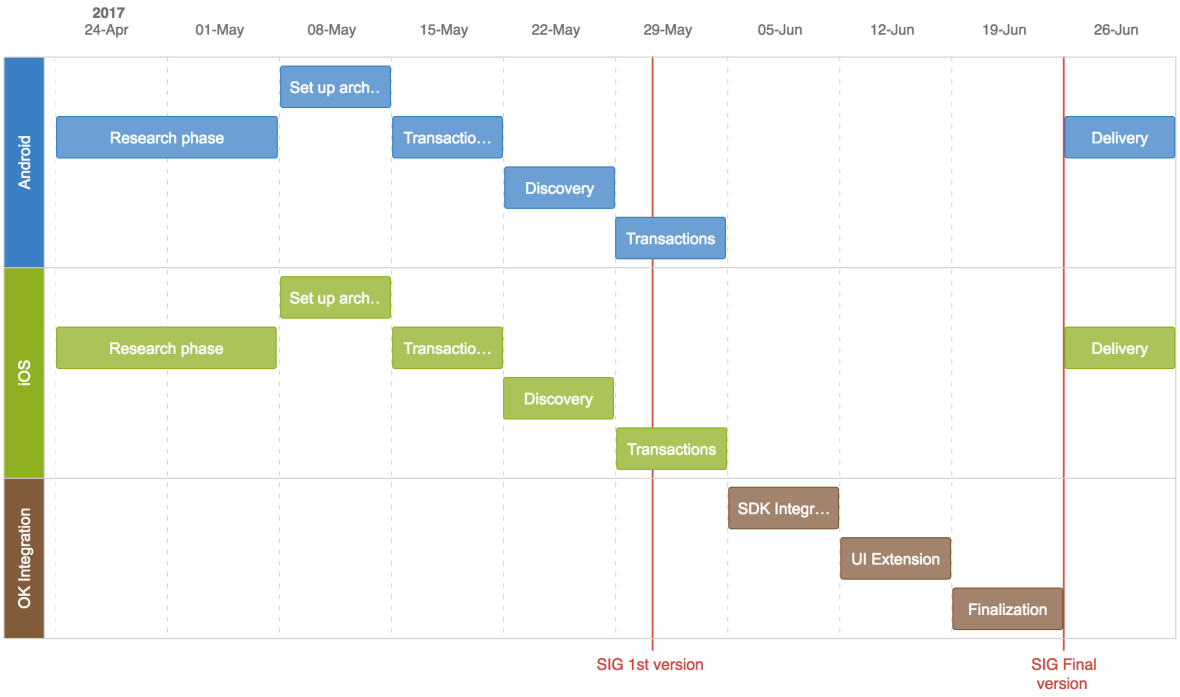
Using the deployment workflow, new versions can easily be deployed by tagging a commit. If the tagged version passes all tests, a new package is generated and automatically distributed by email.

6.3. Planning deviations

During the project, several changes were made to the planning. We created a roadmap for the planning and updated it according to the changes. Figure 6.3 shows the roadmap created at the start of the project and the revised roadmap at the end. When comparing both roadmaps, two changes stand out. Firstly, a minor change in the planning at the start of the project. The TRANSACTION OBJECT VIEW block was split up in two parts, SET UP ARCHITECTURE and TRANSACTION OBJECTS. This did not change the activities for those two weeks, because the splitting was only done to give more detail to the planning.



(a) Roadmap at the start of the project (24/04/2017)



(b) Roadmap at the end of the project (26/06/2017)

Figure 6.3: Project roadmaps. Each column represents a week in the project. Six sprints were done for the weeks of May 8th and June 12th. The dates above the columns represent the end date of the sprint.

Apart from the minor change, there was one deviation from the planning that had significant impact on the final product. For sprint 4, 5 and 6, we had planned to integrate the SDK into OK and to connect the integration to the online Strape Virtual Account System (VAS). However, the OK integration took longer than expected in sprint 4, due to the need to develop a custom user interface for OK. On top of this, we discovered some necessary changes to the SDK that were needed for a deep integration. For example, it became apparent that an app-wide overlay was required to display any SDK notifications to the user in any place in the host app. Furthermore, a connection to the Strape VAS required some changes on the server side of the VAS, which would also require significant time, about a half sprint. Therefore, we decided to put the focus on the OK integration and make use of a simplified Virtual Account System. This way the integration with OK

could be completed and the app could be presented as a demo-able product. The final SDK implementation has not been changed, because the data provider still exists. However, the impact of this decision is on the underlying processing of the transactions. Instead of settling the transactions on the Strape VAS, transactions are handled in the host application locally.

7

Product evaluation

After 7 weeks of developing the SDK and integrating it in OK, we can evaluate the result and assess the features against the original requirements.

7.1. Result

The end product of this project consists of three parts: There is the SDK with the core gesture-based transaction functionality, a demo app to showcase the SDK and a first version of an SDK integration in OK. Transactions can be initiated and authorized using gestures only and users can discover each other and transfer funds using Bluetooth Low Energy. This functionality is also present in the integration with OK.

The SDK consists of a library for both Android and iOS that mainly contains the code that handles:

- The presentation of the various elements.
- The logic for transaction object adjustment.
- The Bluetooth Low Energy management.
- The interaction with a data provider.
- A default implementation of the SDK with the original Strape user interface design.

The code that handles the core SDK logic has an 80% test coverage and is hosted in private repositories on Bitbucket[1]. With the right access keys, a hosting application can use a standard platform-specific package manager to include the libraries as a dependency. The continuous availability of the code is guaranteed by Bitrise, which makes sure the code always builds and the tests succeed. At the time of the submission of this report, version 1.0 of the libraries is deployed and ready to integrate by hosting applications.

The demo app is an application shell around the default Strape implementation of the SDK. It is deployed via Bitrise every time a new version tag is pushed to the Bitbucket repository to a group of testers including the TU Delft supervisor and OK product manager. Part of the demo app is also a data provider that holds some mock data for the SDK to show in the interface, such as accounts with avatars and names.

The OK integration is extensively described in Chapter 5. In summary, the SDK is fully integrated in the OK app in a separate Git branch from the develop branch of the OK Android and iOS app. The integration consists of the following main parts:

- The SDK advertises the user whenever the OK app is opened and scans for other users.
- A playground view that is integrated in the pager on the OK start-up screen between the favorite payment cards and IDs.
- A shortcut in the OK 'Birdseye View' that guides the user on click to the playground view in the favorites.
- An overlay in the app that appears when the user receives a transaction while not in a playground view. This overlay appears only when not in screens where the user is performing some action, e.g. adding an ID or making a payment.

[1] Bitbucket is an on-line Git repository host: https://bitbucket.com

7.2. Requirements assessment

Now the product is developed and the result is visible, we can assess it against the requirements described in the original action plan attached as Appendix C and in Chapter 3. The action plan requirements are the precursors of the detailed requirements of Chapter 3.

The requirements in the action plan have been prioritized using the MoSCoW-method[29], but one of the must-haves has not been realized. After handing in the action plan, we analyzed the requirements further and concluded that it was not relevant in the current proof of concept stage with only one client. We first thought we could not deploy the SDK without such a system, hence the categorization as must-have. But once we had the development environment set up, we realized that we needed to include the SDK locally to be able to streamline the development process.

Secondly, we did not complete the only should-have. Connecting the SDK to the Strape Virtual Account System could not be implemented because of time constraints, as described in Section 6.3. During the initial planning, we saw this as a possible fall back option if time became a limiting factor, so this deviation was accounted for in the action plan already.

Except for the above deviations, the requirements in the action plan correspond with the more detailed requirements of Chapter 3. Regarding the overall integration scope, we can conclude that the app can indeed be fully integrated, as is done extensively already in the integration in OK. Furthermore, the requirements prescribed a focus on the core Strape functionality, i.e. initiating and authorizing transaction using gestures only, and deemed display of detailed information out of scope for the project. These end user requirements, comprising the gestures and user discovery, are indeed successfully implemented. The requirements regarding display of information are also satisfied, with the capability to up-/offload funds deemed out of scope from the start.

For the app to be integrated in OK to the extent that it now is, all scoped requirements must have been satisfied, and they are. Mainly the developed software architecture aided in the accomplishment of these goals, providing several endpoints for the host application to connect to and either listen or send data. This architecture evolved with the implementation of the integration in OK and the result is there.

Following the software engineering requirements every step of the way has made sure there was always a stable version of the code available for testing. Furthermore, code quality was assured by the peer reviews and the code was kept maintainable by the test suites. Finally, a consistent issue tracker corresponding to the commit log made sure we could constantly keep an overview of the work done and planned.

Concluding, nearly all the scoped requirements were satisfied. Planning deviations were scarce but handled accordingly and we can confidently say we have successfully completed the product.

8

Conclusions

The goal of this project was to answer the following question:

> **How can a mobile SDK for Strape functionality be developed that (1) comprises the gesture-based transaction functionality and (2) meets integration requirements that enable as many foreseeable use cases as possible?**

By answering the sub questions, we have shown how to develop and integrate an SDK for the Strape functionality. Firstly, we have defined the requirements for a user interface that allows transaction initiation and authorization using gestures. Secondly, we identified and implemented Bluetooth Low Energy functionality for discovering users in proximity. Furthermore, to determine the integration requirements, we identified categories and use cases of host applications that would have interest in using the SDK's functionality. OK provided the host app for testing, in which we addressed the requirements with an actual integration, including a customized user interface extension for OK. Lastly, we implemented the Model-View-Presenter architecture to allow the integration use cases. This has resulted in a working prototype of the SDK in a real-world application, containing the core Strape functionality.

8.1. Reflection

While working at the OK office, we have gained some valuable insights into the development and management process of a tech start-up. OK has an office in Amsterdam where we spent four days a week during the project. About 10 people work at the OK Amsterdam office, but the only one present with a software engineering background is the CTO. During the project, we were part of a distributed development team of 10 developers. They are situated in Iasi, Romania and work on the several OK systems: back-end platform, front-end webapp and mobile applications.

8.1.1. Romania

In week 3 of the project we visited the development team in Iasi for five days. The goal was to meet the developers and discuss the integration of the SDK into the OK mobile applications. During the project, we became part of OK's bi-weekly scrum sprints to do planning, grooming and coding. We stayed in close contact with the developers via weekly meetings on Tuesday and Friday morning. Being part of a larger, full-time development team taught us several things about developing in a big team.

For example, that communication is key in projects with multiple people, and becomes more important when the group gets larger. With multiple people, it becomes harder to keep track of what everyone is doing, and everyone needs to have a clear and consistent view on what is happening in the project. Furthermore:

Complying with the scrum process saves time Documenting issues and progress in an issue tracker is an essential part of working in a large team and is helpful to anyone, not just the project manager. Each team member can easily check the current progress, what other people are working on and what needs to be done. Furthermore, transferring work between individuals becomes easier, as all knowledge about the work is documented.

Say what you do, do what you say People need to know what others are currently doing. Furthermore, everyone must be approachable by the rest of the team for questions or feedback. Therefore, it is crucial that everyone shares what they are working on. Parallel to this, people must be reliable and follow up on their commitments, which is one of the key prerequisites of scrum.

Large teams have even larger overhead While communication is important in teams, it can cause great overhead, which was new for us. In every 10 workday Sprint, at least 2 days were spent on scrum meetings between team members.

Communication tools are important What was especially new for us, is that sometimes the sender and receiver of information may still not be on the same page after talking about something. It is often useful for parties to get back to what was talked about. Therefore, written communication using issue trackers and chatrooms is an important means of communication in a group, because it records the communication.

Going on a trip to Romania during the bachelor's thesis broadened our knowledge on developing for a company in a distributed development team and improved the experience of doing the project.

8.1.2. Project management

The project supervision from both the TU Delft and OK was very fruitful. Both the supervisor (TU Delft) and the product owner (OK) were always quick to respond to any questions and took the time to sit down with us. The decision to plan regular meetings a with the TU Delft supervisor was, in retrospect, a very good one. The regular meetings created a heartbeat in the project, because it created a weekly assessment of the state of the project and forced us to deliver. Additionally, the regular interval meetings matched well with the system of weekly sprints.

The supervisor and product owner each took a different role in the project. The product owner focused more substantively on the product, whereas the supervisor focused on the process.

Supervisor

The weekly meetings with our supervisor were efficient. Before every meeting, we would create an agenda for the meeting. This allowed us to prepare the meeting beforehand and to use the meeting's time as efficiently as possible. It was important to plan the meetings long in advance, as the supervisor was not always available. However, we did get to sit down with him regularly to discuss the current and future state of the project. While the supervisor did not provide much input about the product, he helped us significantly in managing the project. The supervisor asked sharp questions about various parts of the project, which forced us to think critically about the project's planning, goals and requirements.

We are very satisfied with the role the supervisor played in the project. We can work autonomously but are proactive in asking for help and addressing problems. The supervision style of being present in regular intervals and providing critical, general, feedback, instead of being around all the time and focusing on the details, matched our preference.

Product owner

We had regular meetings with the product owner to look at the project requirements and progress. Together with the product owner defined the broad requirements of the SDK. Moreover, he gave us a lot of valuable insight into the integration with OK. Sitting down with the product owner made us better understand the essential requirements of the product, as well as the requirements for the end user.

A combination of factors made the project a success. First, the supervisor helped us with project management and thinking critically about the product and its requirements. Secondly, the product owner improved our understanding of the product, its use cases and its integration possibilities. Thirdly, we used the knowledge of software engineering, algorithms and architecture design that we gained by following the bachelor Computer Science and by previous development experience.

8.2. Recommendations

We have shown that it is possible to create an SDK for gesture-based transaction initiation and authorization with this project. However, we have limited the project to a specific scope, focusing on core functionality and integration possibilities. We have several recommendations for future work in the project, to complete the entire peer-to-peer transaction proposition for the SDK.

8.2.1. Licensing system
The licensing system was a component that we deemed important, but not relevant in the project's proof of concept stage with a single client. A licensing system is an essential component of any commercial SDK, because it defines the way revenue is generated. However, developing a suitable licensing system can be a complex challenge, because the logic in a licensing system is the result of extensive financial and legal consideration. To develop such a system, one must embody the financial and legal requirements into the SDK's software architecture. Because a licensing system is a key component in any SDK, we recommend that it is developed in future work.

8.2.2. Virtual Account System
As explained in Section 7.2, an implementation of the data provider was not given that connects to a Virtual Account System (VAS) over the network. We recommend creating a data provider instance that connects to a VAS that manages real funds. For example, creating a data provider for the Strape VAS would enable the SDK to initiate and authorize Bitcoin transactions. Similarly, it is possible to create a data provider to connect the SDK to a bank account, for example as a third-party service provider.

Implementing a data provider that connects to a real VAS will introduce new use cases and challenges with regards to data management and might result in new requirements for the data layer of the SDK.

8.2.3. Secondary interaction
Apart from the core SDK functionality implemented in the project, the user experience can be greatly enhanced by implementing other features that display information such as previous transactions. Implemented in the original Strape app is functionality to add a message or photo to a transaction or to respond with an emoji, features that are also specified in Chapter 3 but were deemed out of scope for the project.

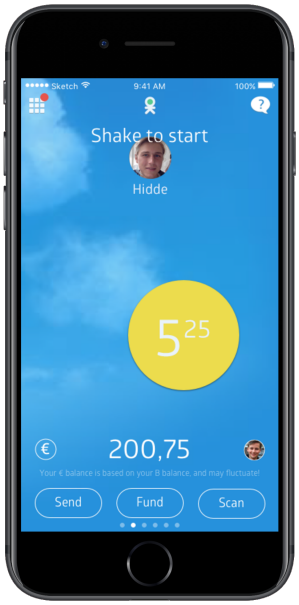
8.2.4. Remote transactions
While initiating and authorizing transactions to users in proximity may be the core Strape functionality, one can imagine users wanting to send transactions when they are not nearby each other. Therefore, we recommend adding the feature to select a recipient from a contact list.

8.3. Ethical implications
Big banks have an enormous monopoly on the financial system, because they control consumers' financial accounts and personal data, as well as the payment systems that are connected to it. The European Union has introduced the second Payment Service Directive (PSD2) to allow third party service providers to get access to the bank accounts and user data, after consent from the user. This will shift the role the banks have in payments greatly, as will allow third parties to compete with banks in financial services.

The direct ethical implications of a gesture-based payment system are hard to identify. However, technology that makes it easier to initiate and authorize transactions that can be integrated into any iOS or Android application may improve the viability of some payment apps. Therefore, we feel that the technology can play a role in competing with the monopoly position of the banks in payments.

A Infosheet

Strape SDK, A gesture-based peer-to-peer transaction system.
Project in association with **OK**
Final presentation on **04-07-2017**



The technology currently available for peer-to-peer payments targets high-volume business-to-customer transactions and thus requires an extensive process for initiation and authorization. Strape provides functionality to ease this process by initiating and authorizing transactions using gestures only. OK is a smartwallet app that combines authentication, identification and marketing content in one solution. The challenge of the project was to create an SDK comprising the peer-to-peer gesture-based transaction initiation and authorization technology and make the technology available for integration in other applications. During the research phase, several approaches were considered with regards to integration depth, system architecture and algorithm design, which meant the system could be designed with the result in mind. The project was divided into six one-week and one final two-week sprint. During the project, one of the challenges that turned out to be harder than expected was the actual dependency integration of the SDK in the OK project. The SDK was tested using automated testing tools, and deployment was tested by integrating the SDK into the OK app.

The integration into the OK app was successful, as shown here to the right. The SDK will be used by OK in the near future. However, as the scope of the project was constrained by time, not all SDK features were incorporated in this project planning. Therefore, the team recommends the expansion of SDK functionality and integration in the future, to have the system facilitate actual financial transactions.

Team

Joris and Hidde have been working on websites and mobile apps together since they were 14 years old. For the past three years, they have taken on several software projects in Fintech and Blockchain in their company Vlinderstorm[1]. Within Vlinderstorm, they developed the first version of the Strape functionality.

Joris Oudejans
Android lead
I am interested in developing products with a focus on user experience and all the business development surrounding it.
E: `joris.oudejans@gmail.com`

Hidde Lycklama à Nijeholt
iOS lead
I am interested in developing systems that have a lasting impact by solving real-world problems.
E: `mail.hidde@gmail.com`

Client: Chiel Liezenberg — Founder and director of OK
TU Supervisor: Inald Lagendijk — Head of Intelligent Systems Department, TU Delft

The final report for this project can be found at:
`http://repository.tudelft.nl`

[1] https://vlinderstorm.com

B

Project description

Met OK kan transacties worden gedaan van de klant naar de verkoper. Wat nog mist is de mogelijkheid om onderling te kunnen afrekenen. Het doel is om een peer-to-peer (P2P) transactie component te ontwikkelen voor de OK app. Deze functionaliteit zorgt ervoor dat gebruikers van OK ook onderling betalingen kunnen doen. Hierdoor kunnen gebruikers euro's, maar ook bijvoorbeeld loyalty items zoals tickets en coupons, uitwisselen. De component zal bestaan uit een SDK voor zowel native iOS als native Android. Hoewel het prototype zal worden getest in OK, kan deze SDK in een willekeurige andere applicatie ook geintegreerd worden.

High-level functionele requirements zijn als volgt:

- Intuitieve user-interface en user-experience (à la Strape, https://getstrape.com/)
- Mogelijkheid om tegoeden onderling te versturen zoals euro's. Hiervoor is er de mogelijkheid om een integratie te doen met een euro wallet, zoals een van Intersolve.
- Mogelijkheid om andere waarde objecten te versturen, zoals coupons en tickets.
- Licensing model (met license keys) en analytics vanaf de SDK

Onderdeel van het project is ook om een test-integratie te doen in de OK app. Hiervoor moet worden gecoördineerd met het ontwikkelteam van de OK app om de integratie succesvol te laten verlopen.

Company description

Met OK Betalen kun je snel, makkelijk en veilig mobiel afrekenen.

OK is een innovatieve, alles-in-één betaaloplossing op je mobiele telefoon, van de ontwikkelaars van iDEAL. De OK-app maakt van je smartphone je nieuwe portemonnee met daarin eindelijk al je bankpassen, paspoort, rijbewijs, lidmaatschapskaarten, verzekeringspassen, klantenkaarten, spaaracties, kortingen, tickets, loten en kassabonnen bij elkaar.

Met de OK kun je overal 24/7 afrekenen met je mobiele telefoon: online, in de winkel, of aan de deur. OK geeft je eenvoudig inzicht en controle over ál je transacties en spaaracties; bovendien worden kortingen direct verrekend. Het enige wat je nodig hebt is de OK-app met je zelfgekozen, 5-cijferige OK-code.

Met OK gaat elke transactie slimmer, sneller, makkelijker en helemaal veilig: je hebt zelf volledige controle over alle betaal- en persoonsgegevens die zijn gekoppeld aan je mobiel.

C

Action plan

Project assignment

This document describes the project plan of the Bachelor End Project of Hidde & Joris which aims to create SDKs from the functionality as implemented in Strape for iOS and Android. Furthermore, these SDKs will be integrated into the OK Betalen app.

Project environment

OK Betalen is a company that develops an all-in-one payment app to be released in The Netherlands. The proposition is an app that combines payment, authentication and loyalty into one solution for all kinds of merchants. The company was founded in 2011 and has grown to a company with more than 20 employees.

Although the app encompasses almost everything regarding payments, a missing component is a peer-to-peer payment system. Hidde & Joris have founded and developed an app called Strape, which provides such a peer-to-peer payment system by using gestures and a gamified interface for transactions. OK Betalen commissioned this BEP for integrating the Strape functionality into its own app and thereby broaden their overall proposition.

Project goal

The main goal of this project is to develop SDKs for Android and iOS from the Strape functionality and integrate these in the respective OK Betalen apps.

Aiming for this goal, the problem statement follows: “How can a peer-to-peer transaction SDK be developed that can be implemented in a third party app?” Considerations in answering this question are for example completely decoupling the SDK code from any code in the hosting app while preserving capabilities for mutually sending event messages,

Requirements

The goal of the project is to realise the following specifications.

Non-functional
All non-functional requirements should be realised.

- SDK for iOS and Android with:
  - Integration guide
  - Continuous Integration using Bitrise
  - Licence requirement
  - Integrated analytics
  - Test coverage > 50
- Test integration of the SDKs into OK Betalen mobile apps.

Functional
We specify the functional requirements according to the MOSCOW-method.
Must-have:

- Licensing system
- Creating, adjusting and sending of a transactional object.
- Discovering other users with Bluetooth Low Energy.
- Extendability in the SDKs to extend/enable features.

Should-have:

- A connection to the Strape Virtual Accounting System (Bitcoin)

Could-have:

-

Won't have:

- A connection to a fiat currency wallet

Plan B
If we do not manage to meet the stated requirements within the time period of the project, the system will not be in the OK Betalen app, but in the Strape app. The focus is on the core SDK functionality as described in the Must-have's in the section above. We will fall back to this plan if we are behind on more than one entire sprint. Furthermore, an optional feature is discovery by Bluetooth Low Energy (BLE). We can choose to not implement BLE for discovery of peers. This would also save time, because BLE functionality can be complex to implement.

Project set-up

The project set-up will be Agile SCRUM-based. Sprints will consist of one week time and the JIRA SCRUM-tool will be used for feature-, issue- and bug-tracking. Not all tools and approaches are definite and we still have to do more research into this topic. The research report will have a definite report on the project approach. A semi-weekly heartbeat meeting with Inald is held to discuss issues and progress in the project. There will be weekly contact and discussions with the client. The goal is to be present at the office of OK Betalen in Amsterdam, on average, two working days per week. Joris will work on the Android SDK, hidde on the iOS one.

Contacts

The contact information for the project members is depicted in Table C.1.

Table C.1: Contact information

Joris Oudejans: Team member	Joris.Oudejans@gmail.com, +316 395 787 00
Hidde Lycklama: Team member	Mail.Hidde@gmail.com, +316 207 899 55
Inald Lagendijk: TU Delft supervisor	R.L.Lagendijk@TUDelft.nl, +31 15 27 83731
Chiel Liezenberg: OK Betalen supervisor	Chiel@okit.com, +31 206 580 651

Tools

Next to JIRA to manage the SCRUM process, a multitude of other tools will be used, which are depicted in Table C.2.

In Confluence, decisions will be recorded and the overall roadmap will be kept up to date. Furthermore, sprints will be documented from start to finish to keep track of the progress.

Table C.2: Tools that will used in the project

Type	**Android**	**iOS**
Issue tracking	JIRA	JIRA
Project tracking	Confluence	Confluence
Version control	Git	Git
Communication	Hipchat	Hipchat
Automated testing	FireBase, Espresso, UI Automator	To be researched...
Code obfuscation	Proguard	To be researched...
Continuous integration	Bitrise	CircleCI (used Bitrise eventually)

Semantic versioning

At the end of every sprint a new version will be released. This versioning will be done according to semantic versioning. This means every version will be of the form x.y.z, where after every sprint release, y will be incremented by one. At the end of sprint 8, version 1.0.0 will be released.

Planning

Time period: April 24 - July 3

The planning is made up of 8 sprints, starting after the research phase is completed. Every sprint deadline is on sunday, as shown in the brackets in Figure C.1. In the second week of May, Hidde & Joris will go to Romania to meet the team and work at their office.

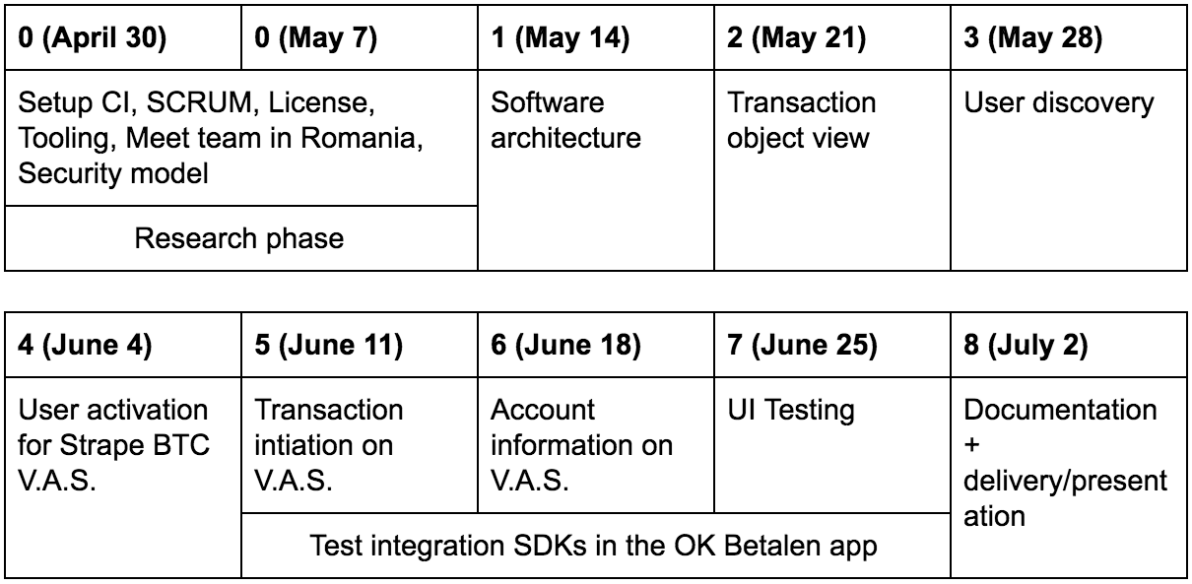


Figure C.1: Roadmap

Furthermore intermediary meetings are scheduled with Inald on:

- May 9
- May 16
- May 30

- June 14
- June 20

Quality assurance

For the developed SDKs will be integrated in an app that is in the production stage, assuring the quality in certain aspects is of utmost importance. For assuring code quality, on two occasions source code will be sent for evaluation to the SIG (Software Improvement Group). The deadlines for this are 01/06 (75

D

Research report

Introduction

More and more mobile payment applications are brought into the market and all mobile transaction statistics are growing forcefully[25]. In the Netherlands, a new player is entering the market called OK BETALEN, an all-in-one smart-wallet, encompassing payments, identification, authentication and loyalty. The one thing that is still missing from this proposition is the ability for users to make transactions to each other. This is where STRAPE comes into the picture. STRAPE is a peer-to-peer payment app developed by VLINDERSTORM that uses gestures to initialize and authenticate transactions between users. The goal of this project is incorporate the STRAPE functionality in the OK BETALEN app and broadening their proposition to include peer-to-peer transactions.

Before integrating the gesture-based transactions into the OK BETALEN app, the current STRAPE functionality must be encapsulated in an SDK for both Android and iOS.This document describes the various aspects that need to be taken into account when developing an SDK that will be used in a intensively utilized app in a production environment and the decisions we made considering them. The scope of the research could have been wider and incorporated for example the discovery of other users and UI algorithms as the flinging of a coin, but we focused on aspects that required more extensive research and left out the easier ones. First, we discuss the SDKs functional features and integration possibilities. Next, we review a possible software architecture of the SDK using presenter components. Finally, we look at the considerations for algorithms regarding transaction object adjustment.

Framework

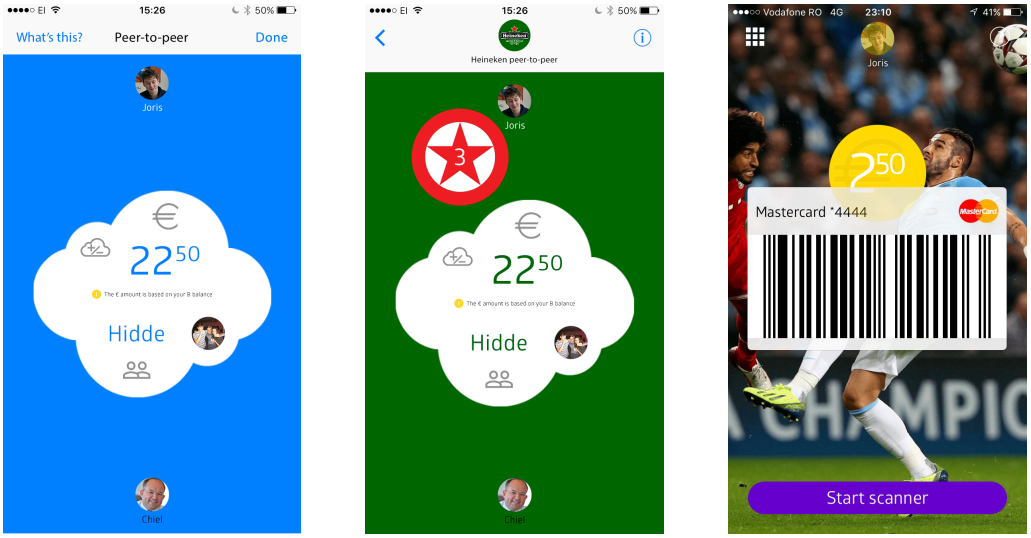
An important aspect of developing an SDK is choosing the scope of functionalities it comprises and how to achieve these in the framework of the hosting application.

UI integration level

The most important decision about the scope in this specific SDK is the level of integration possible from the SDK. There are three options to consider:

Overlay A pre-defined user-interface provided by the SDK is used to interact with the SDK functionality. The hosting application can open this user interface responding to some event. For example, by providing a button that opens the fixed user-interface. This is the simplest form of SDK integration, because no user-interface components have to be in place. There is a simple entry point into the SDK functionality, by opening the user-interface. From that point, the SDK handles the rest of the user interaction. On top of an easy integration, this ensures that the security of the data is higher, because the SDK controls where and how data is accessed. However, the user experience for this approach is limited, because the component will always be separated from the host application. Apart from some styling, configurability of this component is low.

Embedded The host application specifies a container in which the SDK functionality is embedded. This is similar to an overlay in that there is still one component that is handled by the SDK. The integration ease



(a) Overlay (b) Embedded (c) Fully Integrated

Figure D.1: UI Integration levels

and security are therefore similar to that of the overlay, albeit that an embedded component is more easily accessible by the host application and is thus more complicated and vulnerable. The embedded version would also be similar to the overlay one in configurability, except that there is more freedom as to the selection of the container.

Fully Integrated The host application is provided with elements from the SDK, e.g. the transaction objects, discovery logic and other views. It can use these components wherever it wants. Although the user experience depends on the implementation of the hosting application, one can assume that a better overall user experience is provided than when choosing the overlay or embedded option. The integration for the host application is tedious, though, as every single component needs to implemented separately. Furthermore, the security is less than with the others as this also needs to considered for each component separately. The configurability is very good, as this option enables the host application to specify every small detail about the integration of the SDK.

A matrix comparing these three options is given in Table D.1. User experience is the extent to which the end user has a good experience with the specific integration option. Integration ease describes how easy it is for a hosting application to integrate an SDK with the specific integration option. Security is how protected the SDK component is from breaking by actions from the hosting application. Configurability is the extent to which SDK components are configurable by the hosting application.

	Overlay	Embedded	Fully Integrated
User experience	-	+	++
Integration ease	+	+/-	–
Security	+	+/-	-
Configurability	-	+	++

Table D.1: SDK Integration

Different host applications want different levels of UI integration depending on which of the properties of Table D.1 they value the most. Host applications might even want different levels of UI integration at different stages. They might prefer to start with the easiest level to integrate, i.e. the overlay, and see how well the functionality fits within their proposition and how the users react. Then they can expand to use a more integrated approach if previous results were positive. Because of this, we define three levels of integration hosting applications can implement.

Level 1. Basic integration. This is the SDK set up with the default *cloud* and *coin* components, as defined in Section D. This set up can be customized using the SDKSETTINGS class.

Level 2. Partial integration. In this set up, the SDK can use the default set up, but allows for custom TRANSACTIONOBJECT components. This can be used for apps that would like some custom functionality for transactions, but do adhere to the same basic concepts of TRANSACTIONOBJECTS, ACCOUNTS and OUTLETS.

Level 3. Full extendability. In this case, the SDK functionality can be fully customized new business logic can be added. Moreover, SDK components can be used independently from each other, even outside of their regular environment.

By implementing the above integration levels, we believe the SDK will be open for use in most foreseeable use-cases ranging from just an overlay to a fully integrated set-up.

Account integration level

Next to the UI layer, it is necessary to integrate the data layer. The SDK contains functionality to keep track of transactions per user. It is necessary to create a connection between the SDK identities and the identities of the host application.

There are several ways to implement this connection, ranging from lightweight to data-intensive. Furthermore, connections can be tightly or loosely coupled. As we want to make it as easy as possible for a host application, we have designed a component that can be used to implement a lightweight, loosely coupled integration between a virtual account system provider and the SDK. More detail on this component is given in Section D.

Licensing system

In order to ensure authorized use of the SDK, we can use a licensing system. In such a system, license keys are issued to users of the SDK. These license keys are tied to a specific application. This is done via the application identifier, a unique identifier for an application that is used by iOS and Android.

We separate between online and offline licensing systems. The distinction is defined by the way the system verifies the license keys.

Offline

Offline licensing systems does not connect to a server to check for the validity of the license. In this case, public-key cryptography is used to verify the license key. The SDK contains a trusted public key from the issuer identity. The license key is a signed message by the issuer, containing the application identifier and start and end dates for validity.

$$\text{LICENSEKEY\_SIGNATURE} = \sigma_{Pk}(\text{APPID} \,||\, \text{VALID\_START} \,||\, \text{VALID\_END})$$

The SDK verifies that the digital signature of the license key is signed by the identity of the trusted public key[28]. It also verifies that the application identifier defined in the key is the same as the actual application identifier, and that the system date is within the license key period.

Because no connection to the internet is made, this system can be deployed for apps that require no internet connection. Furthermore, the system is relatively lightweight in terms of processing.

Online

An online licensing system verifies the license key with a server that is owned by the issuer. In this case, the license key does not contain a cryptographic signature, but can be a Universally Unique IDentifier (UUID)[24]. An example is shown The license key is verified by the server each time the SDK is used in the host application.

Listing D.1: Example of a UUID

```
8cf6c790-35c2-4ec2-bd02-b5d8be5044d2
```

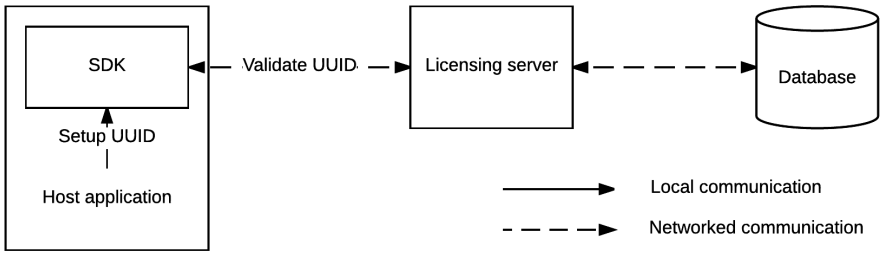


Figure D.2: Networked licensing model

While this system with network communication is more complex to set up, it has several benefits. Firstly, online server verification allows for flexible prolonging and revoking licenses, whereas offline license keys are tied to the pre-defined period they are issued for. Secondly, connecting to a server allows Vlinderstorm to collect usage statistics about the SDK in apps. This way, more elaborate license pricing models can be applied, such as taking the amount of installs into account.

The SDK will most likely be used in application that do some kind of transaction processing between several parties. Therefore, it is unlikely that the SDK will be used in apps that are completely offline, or very lightweight. Because of this, we have decided to create an online licensing system with a server that does verification. In the host application, the SDK will be set up using a license key in the form of a UUID. The SDK will verify the validity of the key with the licensing server over the internet. Once this is done, all features of the SDK can be used. An example is shown in Figure D.2.

Architecture

The software architecture of the SDK is a big factor in the quality of the final product. It ensures that the codebase is maintainable and understandable. Furthermore, it improves the efficiency of extending the SDK with new improvements and functionality. By defining a robust architecture model, it is easier to make reliable assumptions about security and performance of the SDK.

The pattern that is key to the SDK's architecture is the Model-View-Presenter (MVP) architecture. We introduce this architecture by describing the research process that lead to the result of using MVP. Afterwards we describe the full structure of the SDK, based on this MVP model.

Model-View-Presenter

The Model-View-Presenter design pattern (MVP) has been introduced in the nineties at Taligent[30]. It is based on the Model-View-Controller (MVC) architectural pattern, that is used to implement user-interfaces[13]. The MVC pattern defines three separate components.

The model is responsible for all the applications data and business logic.

The view displays and formats the necessary information in a user-interface. The view also processes user input.

The controller connects both the model and the view, by accepting input from the model and events from the view.

The MVC pattern is widely used in a multitude of different ways and on different platforms. However, this pattern has shown several limitations in terms of software testing. This is because traditionally in an MVC architecture, the view is tightly coupled with the model. That is, it interacts directly with the model. Furthermore, the view often contains business logic for presenting the data and processing user input. This logic is hard to test, because performing a test on the view's business logic requires rendering the whole view component, which is often complex and unnecessary.

The MVP architectural pattern attempts to improve the testability of the system by moving all application business logic to a different component: the presenter. In the MVP architecture, the components are defined as follows:

The model is responsible for managing all application data.

The view displays information from the presenter and connects user input events to the presenter. The view contains as little application logic as possible.

The presenter is responsible for connecting the model to the view and performing all business logic, including data formatting and user input processing.

While there is a lot of ongoing discussion on the differences between a controller and a presenter class[2], we define a key distinction between the two. A controller can display multiple types of views, whereas a presenter implementation is specific for a certain view. This means that a controller is less tightly coupled with the view and model, which requires more business logic in the view. The MVP pattern improves the testability of the business logic in comparison to the MVC pattern, because all the business logic is contained in the presenter component. The view component is kept as 'thin' as possible, meaning that it contains application logic as little as possible. The presenter component interacts with the model and the view using interfaces. This makes the presenter easier to test, because mock classes can be made for both the model and the view.

Mobile platform implementation
Both iOS and Android contain similar components to build a mobile user-interface. These components include Views, for both platforms, and UIViewControllers[5] and Activities[14], for iOS and Android respectively. This structure can be compared to the MVC architecture, because the UIViewControllers and Activities are generally used to represent user-interface screens with multiple Views, all controlled by the controller component. As mobile user-interfaces have developed to become more complex and flexible, it was necessary to develop a more maintainable system to manage this. This meant the possibility to divide a user-interface into independent, smaller components. An example of such an approach is React[12]. Both iOS and Android have adapted to this requirement. In iOS, UIViewControllers can now be added to other UIViewControllers, to promote dividing a user-interface into separate components, thereby giving UIViewControllers a more presenter-like role. For Android, Google introduced Fragments[15], independent, controller-like components that are used to implement parts of user-interfaces.

We can see that complex user-interfaces require a more structured approach. Instead of handling all the logic of one 'screen' in one controller, we must split the screen up in smaller components. These components can each be managed by their own presenter. In the case of the SDK, we have decided to go with this presenter idea. The user-interface is split up in several subviews, each with their own presenter. To implement this idea in both mobile platforms, this means that the traditional controllers, be it UIViewControllers or Activities, are contained in the *View* component of the MVP model. In addition to this, a separate class is created for the *Presenter* functionality.

Structure

The structure for the SDK is based on the idea of Model-View-Presenter. The SDK structure can be subdivided into several components. Firstly, the abstract structure for the main business logic. Secondly, a concrete, default implementation of these interfaces to actually implement the main SDK functionality. Thirdly, a builder component to instantiate the presenters. And finally, several utility classes to manage settings, licensing and connectivity. The structure is shown as an UML diagram in Figure D.3. Furthermore, the builder and utility classes are shown in Figure D.4.

Abstract implementation
The first layer of classes that define the core functionality of the SDK are entirely defined as abstract interfaces. This way, the SDK is extendable and testable by design. The PLAYGROUNDPRESENTER is used to display the main PLAYGROUNDVIEW with data from the ACCOUNTMODEL. On the PLAYGROUNDVIEW several subviews are displayed. These are the TRANSACTIONOBJECTVIEW, for displaying transaction objects such as coins, OUTLETVIEW to display other users, and one ACCOUNTVIEW to display account information. All views are managed by their own presenter, TRANSACTIONOBJECTPRESENTER, OUTLETPRESENTER and ACCOUNTPRESENTER, respectively. In this view structure, data is displayed from the ACCOUNTMODEL and TRANSACTIONOBJECTMODEL. Furthermore, the TRANSACTIONOBJECTPRESENTER uses a TRANSACTIONOBJECTADJUSTER to handle the different types of view adjustments that occur when an object is interacted with by a user. The TRANSACTIONOBJECTADJUSTER is dependent on a specific CURRENCY, because every currency can have a different impact on the display of the view.

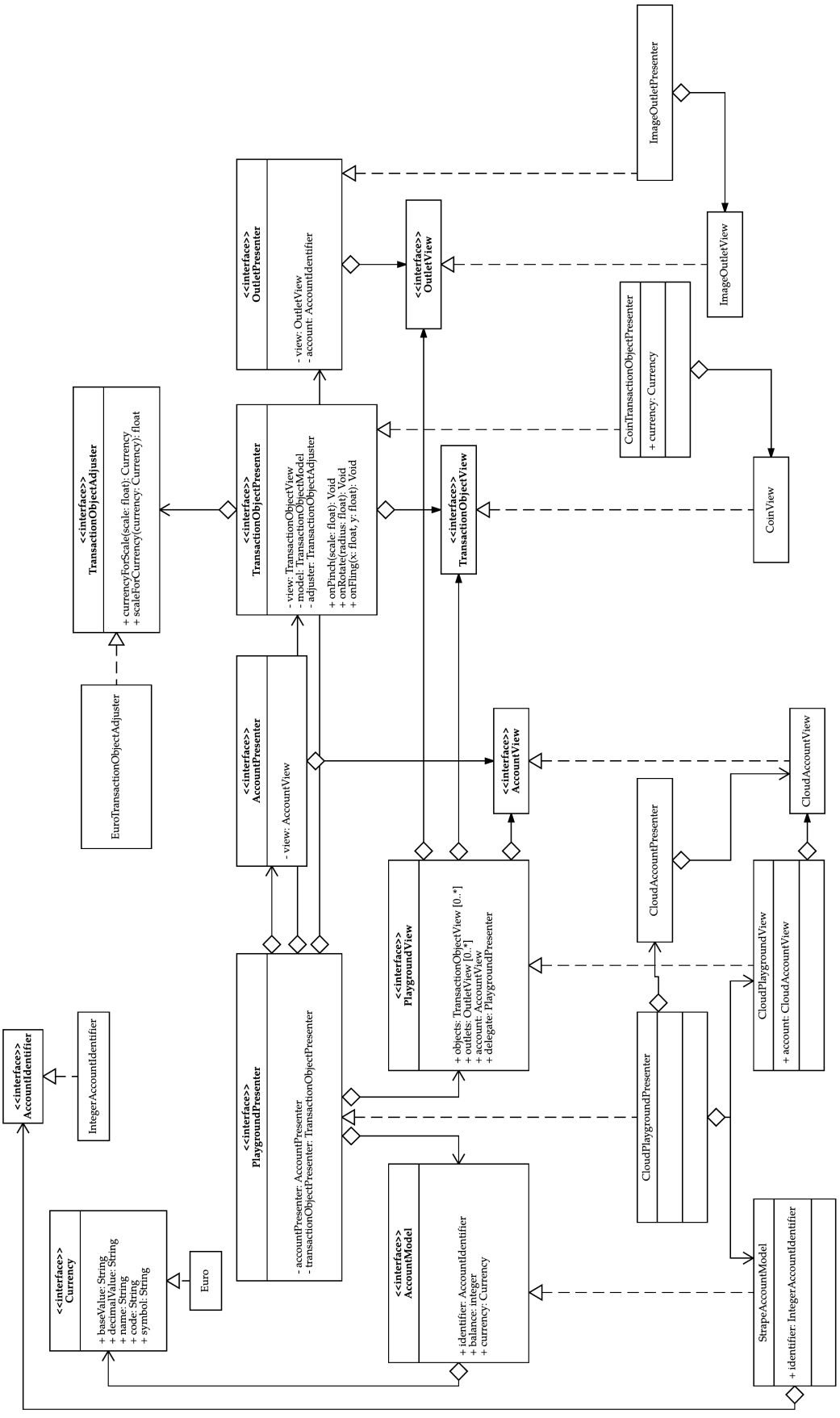


Figure D.3: UML diagram of SDK architecture for main functionality

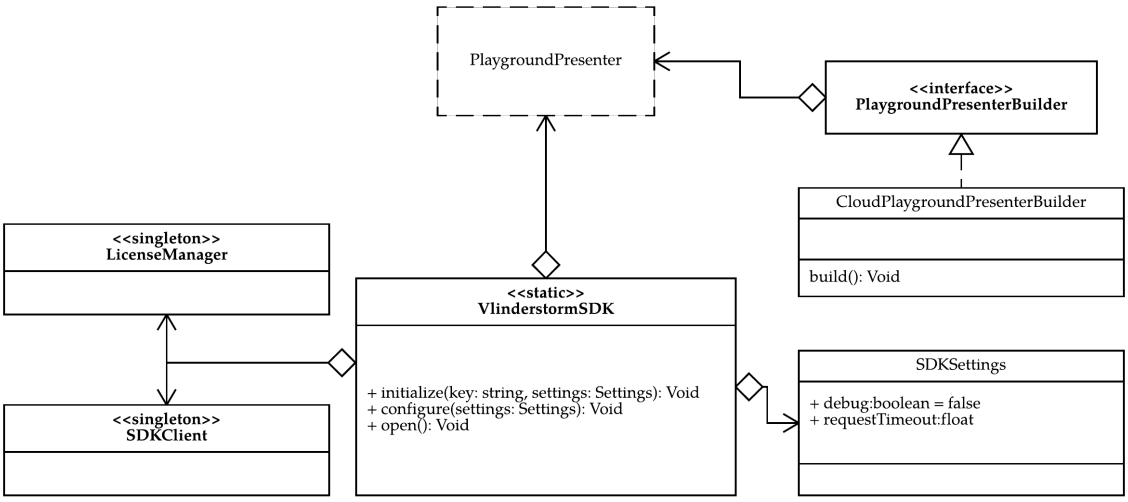


Figure D.4: UML diagram of SDK architecture for builder and utility classes. The PLAYGROUNDPRESENTER class references the same class defined in Figure D.3

When looking at Figure D.3, it is apparent that for every subview two references exist. The first reference is from the parent view, which contains the view in the operating systems view hierarchy. The second reference is from the presenter that is coupled with the view. Similar to the systems view hierarchy, the SDK contains a presenter hierarchy.

Implementation
The second layer of core classes implement the abstract structure as described in Section D. These classes provide a standard functional implementation of the SDK. This implementation is that of the "cloud UI", using CLOUDPLAYGROUNDPRESENTER/VIEW and CLOUDACCOUNTPRESENTER/VIEW, with coin objects, using COINTRANSCTIONOBJECTPRESENTER and COINVIEW. Moreover, outlets are presented as profile names and images, by the IMAGEOUTLETPRESENTER/VIEW.

Builder
A tradeoff when using a decoupled structure is the fact that it can be a challenge to connect all the presenters together with their views and data model. Therefore, the SDK has a builder component that is responsible for setting up the connection between classes. This way it is simpler to construct the SDK even with custom components.

Utilities
Apart from the core functionality and builder components, the SDK contains several utility classes for common functionality and simplification. The included utilities are as follows.

- Networking component to facilitate network communication.
- Settings can be used to define customizable settings for components.
- Static intializers are used to provide an easy, standardized way to set up the SDK in a host application.
- The LicenseManager is used to ensure the use of the SDK is authorized.

The classes provide common functionality and helpers to ensure consistent functioning of the SDK.

Providers

Host applications should be able to interact with the SDK and retrieve meaningful information from it. For example, when a transaction has been made, or when a transaction is received. In the SDK, this can be done using a provider. A provider is a component that listens for SDK events. Furthermore, a provider supplies the

Table D.2: SDK events

Event	Description
Transaction Sent	A transaction has been created by the user
Transaction Received	A transaction has been received
Outlet discovered	The device has discovered an outlet. An outlet can be discovered in several ways, including Bluetooth Low Energy or QR scanning
Account updated	The account data has been updated by a provider. This includes SDK default data such as the account identifier and the balance

SDK with account data. Account data can be arbitrary, as long as it contains an ACCOUNTIDENTIFIER, which the SDK uses to uniquely identify accounts. The different types of SDK events are described in Table D.2.

A host application can connect multiple SDK providers. For example, the OK application will use a provider to listen for events, but will also rely on a custom Strape provider to manage the Virtual Account System.

Transaction object adjustment

The main goal of the SDK is to provide the gesture-based functionality. The user can resize by pinching the coin and thereby adjusting the base value displayed on the object. For example, when we have a coin that displays a euro value €2.50 on it, pinching out (i.e. the zooming in gesture) would increase the value to €3.50, then €4.50 etc. The other gesture is a dragging one in a rotating way around the center of the transaction object to adjust the decimal value. Rotating would therefore change the original €2.50 value to €2.51, then €2.52, etc. To achieve this behavior, several equation need to be used and these equations need to be combined into algorithms for the resizing and the rotating. In the following section the equations for the necessary steps in the algorithm are set out and in the next section the equations are used to derive the two algorithms.

Equations

For translating the user movements into the amount values several algorithms for specific stages of the translation process are implemented. These algorithms are structured in Figure D.5. The system, whether it is

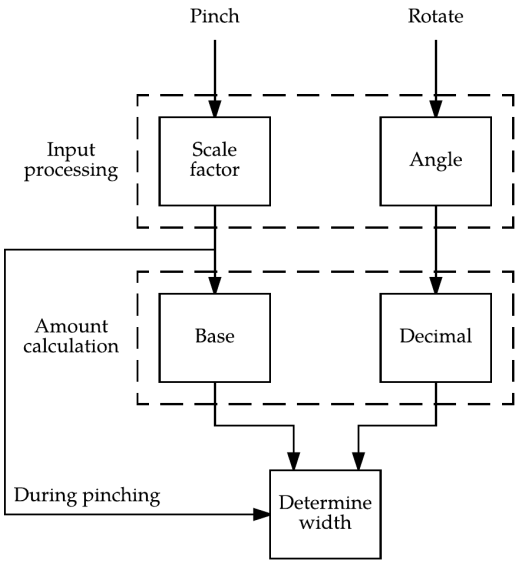
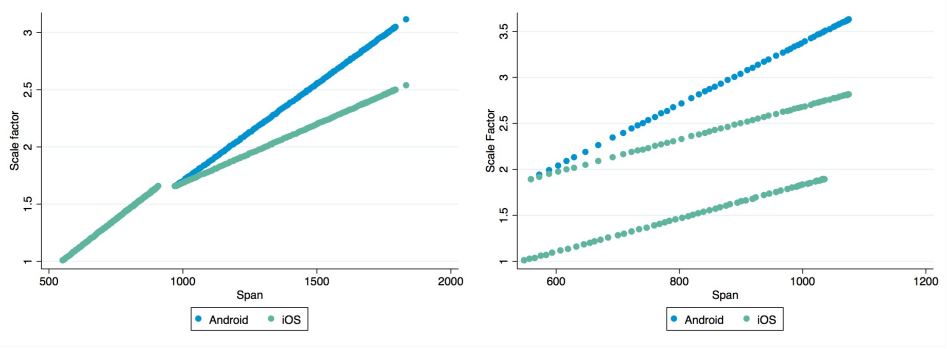


Figure D.5: Transaction object adjustments



(a) Keeping touch position

(b) Resetting touch position

Figure D.6: Scale factor increasing

Android or iOS, calls either a pinch or a touch event which is then passed on to the listeners registered by the SDK. These events are processed to values that can be used to calculate the base and decimal amounts. The combination of the calculated base and decimal values finally results in the width of the transaction object. During pinching the determination of the width results directly from the scale factor to make sure the width is adjusted in a linear fashion instead of in steps every base change. On releasing, i.e. the event a user stops touching the screen, the width is snapped to represent an integer base value. The following sections describe these algorithms in more detail.

Scale factor

On Android, the OnScaleGestureListener can be implemented to receive pinch events[20]. Similarly, in iOS UIKits UIPinchGestureRecognizer[4] can be used to receive pinch events. There are multiple ways to calculate the scale factor to be able to determine the base value. The recommended way on Android to calculate the scale factor f given a span s, i.e. the average distance between the touch events, is in the following way[19]:

$$f_n = f_{n-1} \cdot \frac{s_n}{s_{n-1}} \tag{D.1}$$

Thus, the current scale factor is derived by multiplying the previous scale factor by the ratio of the current span minus the previous span. On iOS, the scale factor is calculated in the following way:

$$\begin{aligned} f_n &= f_{n-1} + \frac{s_n}{s_0} - \frac{s_{n-1}}{s_0} \\ &= f_{n-1} + \frac{s_n - s_{n-1}}{s_0} \end{aligned} \tag{D.2}$$

In this case, the scale factor is derived by adding the difference between the current span and the previous span divided by the starting span to the previous value of the scale factor. The main difference between these two is that Equation D.1 grows faster for larger f_{n-1}, while Equation D.2 always grows relative to the touch event starting position. Therefore, Equation D.1 will keep increasing the scale factor faster when a user keeps pinching from the center of the view, while Equation D.2 keeps a constant increase, as illustrated by Figure D.6a. Corollary to this fact is that Equation D.1 also keeps a linear increase when starting span increases, while Equation D.2 flattens out on higher spans, as illustrated by Figure D.6b. We decided to use Equation D.2 for determining the scale factor as we can mimic the iOS calculation on Android, but not the other way around, and our goal is to give the user the same experience on Android and iOS.

Angle

The angle is calculated by taking the arc tangent of the x and y coordinates of the touch event relative to the center of the view. Thus, the angle difference at a random point (x_n, y_n) relative to the view center after dragging is calculated by the following equation:

$$a_n = (2\pi + \arctan\frac{y_n}{x_n} - \arctan\frac{y_0}{x_0} \mod 2\pi) - \frac{\pi}{2} \tag{D.3}$$

Base
To calculate the base value resulting from the scale factor, parameters are used that can be set by the host application. These parameters are:

- Multiplier $m > 0$, default 1.
- Exponent e, default 2.
- Constant $c > 0$, default 0.

Given these variables and the scale factor f as derived in Section D, the base value b is calculated as follows:

$$b = m \cdot f^e + c \tag{D.4}$$

To illustrate this with an example, take the default values for the parameters. The base values for various scale factors is then given in Table D.3.

Scale Factor	Base
1	1
2	4
3	9
4	16

Table D.3: Base calculation

Decimal
The decimal number is based on the angle as calculated in Section D. The equation to derive the decimal d from the angle a given the amount of decimals in one base unit o is as follows:

$$d_n = d_0 + \lfloor \frac{a + \frac{\pi}{2}}{2\pi \cdot o} \rfloor \mod o \tag{D.5}$$

Determining the width
With the calculated base and decimal values, the width of the view can be derived. The idea is that the radius is a continuous function of these two values. To accomplish this, the radius is the addition of two components, namely one based on the base value and one based on the decimal value. Given the parameters m, e, c from Section D, the base value b, the decimal value d and the amount of decimals in one base unit m, the width w is calculated in the following way:

$$w = \log e \frac{(b + \frac{d}{m}) + c}{m} \tag{D.6}$$

Algorithms

Using the derived equations for each step in the process, algorithms are set out that result in the desired behaviour for resizing and rotating the transaction object. The algorithm called when a resizing touch event is received is depicted in Algorithm 3. This algorithm depends on the following input variables at the nth event:

- f_{n-1}, the scale factor calculated in the previous touch event.
- s_0, s_{n-1}, s_n, the start, previous and current span of the scale gesture.
- Mutiplier m, exponent e and constant c, the config parameters for the scale gesture.
- d, the current decimal value.
- a, the maximum base value.
- u, a boolean that is true iff the event is the user releasing, i.e. stopping the scale gesture.

Algorithm 3 Resizing the transaction object

$f \leftarrow f_{n-1} + \frac{s_n - s_{n-1}}{s_0}$
$b \leftarrow \text{MIN}(a, \text{MAX}(0, \lfloor m \cdot f^e + c \rfloor))$
if u = true **then**
    $w \leftarrow \log e \frac{(b + \frac{d}{m}) + c}{m}$ ▷ If the gesture ends, snap to the width of the integer base
else
    $w \leftarrow f + \log e \frac{\frac{d}{m} + c}{m}$ ▷ Otherwise, scale the object linearly with the gesture
end if
return (w, b)

It returns the width the transaction object should present and the base value to be displayed.

The algorithm for calculating the decimal value and the appropriate width is depicted by Algorithm 4 and depends on the following input variables:

- $(x_0, y_0), (x_n, y_n)$, the start and current coordinates of the drag gesture.
- d_0, d_{n-1}, the start and previous decimal values.
- Mutiplier m, exponent e and constant c, the config parameters for the scale gesture.
- o, the amount of decimal in one base unit.
- b, the current base value.

As with Algorithm 3, it returns the width the transaction object should present and the base value to be displayed, and here also the decimal value is returned.

Algorithm 4 Rotating the transaction object

$a \leftarrow (2\pi + \arctan \frac{y_n}{x_n} - \arctan \frac{y_0}{x_0} \mod 2\pi) - \frac{\pi}{2}$
$d \leftarrow d_0 + \lfloor \frac{a + \frac{\pi}{2}}{2\pi \cdot o} \rfloor$
if $d_{n-1} < \frac{1}{3}o \wedge d \geq \frac{2}{3}o$ **then** ▷ Decrease base value if decimal value change indicates it
    $b \leftarrow b - 1$
else if $d_{n-1} \geq \frac{2}{3}o \wedge d < \frac{1}{3}o$ **then** ▷ Same with the other way around
    $b \leftarrow b + 1$
end if
$w \leftarrow \log e \frac{(b + \frac{d}{m}) + c}{m}$
return (w, d, b)

Conclusion

As described in this report, the gesture-based transaction SDK provides a peer-to-peer functionality. The host application can choose the level of UI integration depending on the user experience, integration ease, security and configurability they desire. Based on these levels, the SDK provides a layered structure of components the host application can use for integration. Apps will use an online licensing model based on UUIDs because that enables us to keep track of detailed usage statistics and implement more elaborate pricing models in the future. For the SDK to have independent, loosely coupled components and be easily and most extensively testable, the Model-View-Presenter design pattern will be used. All components are based on interfaces so that every component can easily be extended by the host application. Transaction objects are adjusted by resizing and rotating, resulting in respectively the base and decimal values, which in turn result in the width of the object. Equations for these steps are described and resulting from these the algorithms that are called upon receiving touch events.

During the implementation of the SDK, it is probable that the specifics will be adjusted according to insights gained in the process. However, the decisions described in this report are the basis the SDK will be built upon, and changes to them will have to be carefully considered.

E

Acronyms & Glossary

Acronyms

PSD2	Second Payment Service Directive
SDK	Software Development Kit
TPP	Third party service provider
UI	User interface
VAS	Virtual Account System

Glossary

End user	User of a host application that is therefore the one to use the the Strape SDK functionality in practice.
Host application	The application the SDK is integrated into.
OK	All-in-one smart wallet app.
Payment Service Directive	Directive created by the European Union that forces banks to share user data with third party service providers. It enables bank customers, both consumers and businesses, to use third party service providers to manage their finances.
Pinching	Moving two fingers on a touch screen either apart from each other for zooming in or towards each other for zooming out.
Strape	App that makes use of the Strape SDK and provides several other social features.
Strape SDK	SDK that comprises gesture-based transaction functionality.
Third party service provider	Under the second Payment Service Directive (PSD2), a third party service provider is a party that offer payments services, but are not the sender or beneficiary of the transactions and do not hold the bank accounts of the users.
Virtual Account System	A system that manages and persists virtual accounts.

F

Software Improvement Group feedback

The projects source code is analyzed twice during the project. The first iteration is to get feedback on potential software engineering-related issues in the code and to improve the code accordingly. The second iteration is used to see if the changes in the meantime have dealt with the highlighted issues. The feedback is depicted in the sections below.

The issues raised by the first round of feedback were code duplication and unit complexity. To mitigate the first, we analyzed the code and moved all duplicated code to separate methods. The second one was handled by dividing the relevant code into parts or moving the complex methods to separate classes. This also helped dividing the responsibility into more classes, adhering more to the Single Responsibility Principle. We also made sure to extend the test suite as the code-base grew.

First iteration (01/06/2017)

De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Duplication en Unit Complexity.

Voor Duplication wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren.

In jullie project zit er bijvoorbeeld duplicatie tussen TransactionObjectView.shoot en TransactionObjectView.shootTo. Het is beter om het gemeenschappelijke deel in een aparte methode te plaatsen en die vervolgens aan te roepen. In PlaygroundPresenter is iets soortgelijks aan de hand, daar zit functionaliteit in die sterk op elkaar lijkt. Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt.

Onnodige complexiteit ontstaat vaak doordat een methode te veel verantwoordelijkheid heeft. Voorbeelden hiervan zijn PlaygroundPresenter.positionOutlet en PlaygroundView.onFling. Door verantwoordelijkheden te scheiden wordt niet alleen de complexiteit lager, maar wordt het ook makkelijker om de logica te begrijpen en te testen.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Over het algemeen scoort de code dus gemiddeld, hopelijk lukt het om dit niveau nog wat te laten stijgen tijdens de rest van de ontwikkelfase.

Second iteration (25/6/2017)

The second round of SIG feedback is not in time for the final submission. However, it will be included into the repository upload version.

G UML specification

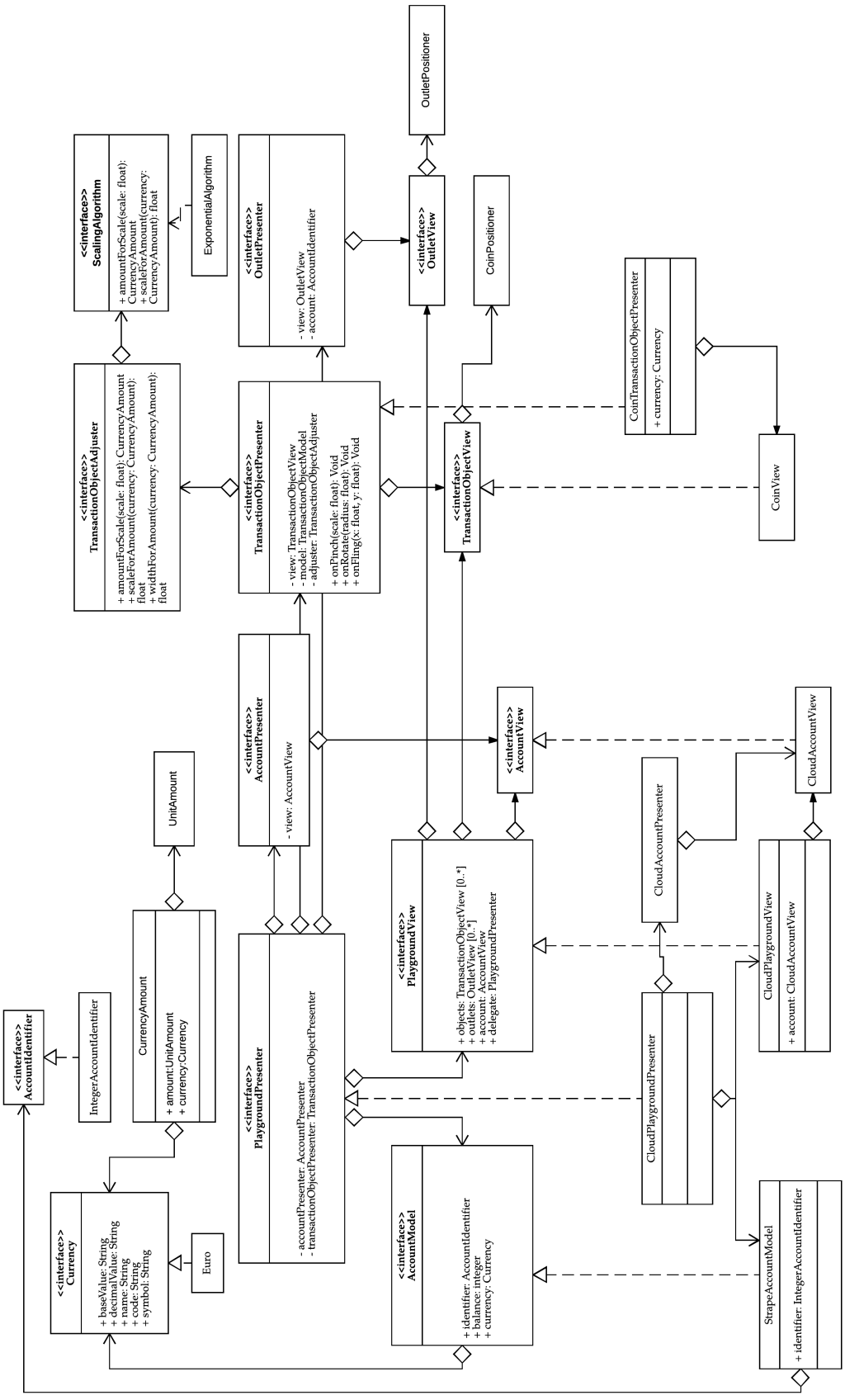
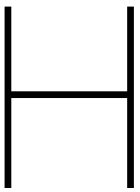


Figure G.1: UML diagram of SDK architecture for main functionality



Decision Log

The decision log is a list to keep track of implementation decisions during development, and served as a first documentation step of important decisions.

Android Minimum API Version

Status	DECIDED
Stakeholders	Hidde Lycklama, Joris Oudejans
Outcome	API level 15
Due date	07 May 2017
Owner	Joris Oudejans

Background

Although API 21 (Android 5.0 Lollipop) brings some significant improvements that would make the development more easy, but only 40% of users has this version. Furthermore, there are support libraries that make it possible to have API 21 functionality in earlier versions.

Event pushed to the presenter

Status	DECIDED
Stakeholders	Hidde Lycklama, Joris Oudejans
Outcome	Use separate observer component
Due date	15 May 2017
Owner	Hidde Lycklama

Background

Whether interactor listener, or separate observer component

Framework architecture - MVP restructuring

Status	DECIDED
Stakeholders	Joris Oudejans, Hidde Lycklama
Outcome	Stay with old structure, as this turned out to be more maintainable.
Due date	12 May 2017
Owner	Hidde Lycklama

Background

Getting back to research decision because during implementation we ran into structural and logical inconsistencies.

Instead of every Presenter <-> Protocol being implemented by a separate view, the Protocol is object-agnostic and can be implemented by anything. From this follows that all defined protocols are handled by a 'system view component'. A way to make the view system extendable is separate from the MVP and not in the scope of MVP. This way, the presenters are way less tightly with the views.

To consider:
- The old structure was already partly implemented, because we are mid-sprint.
- The new structure is more logical and maintainable

Implement listeners for interactor

Status	**DECIDED**
Stakeholders	Joris Oudejans Hidde Lycklama
Outcome	Create listener interfaces and implement by presenter
Due date	13 May 2017
Owner	Hidde Lycklama

Background

Presenters call function on interactors. It is not common to have the interactors know about the presenter

In rotation algorithm: store decimal and base value or just overall amount?

Status	**DECIDED**
Stakeholders	Hidde Lycklama Joris Oudejans
Outcome	Go with iOS implementation version: Store the overall amount and derive the base and decimal values accordingly
Due date	14 May 2017
Owner	Joris Oudejans

Background

We have to figure out which method is more accurate and results in cleaner code.

Multiple data providers (multi-wallet / currency) implementation

Status	**DECIDED**
Stakeholders	Joris Oudejans Hidde Lycklama
Outcome	Allowing for multiple data providers and currencis is out of scope for this project
Due date	12 May 2017
Owner	Hidde Lycklama

Background

Allow for multiple data providers and wallet providers in one SDK instance. For example, to allow Strape and Intersolve at the same time.

Software architecture - Use interactors/worksflows per use-case

to model action flows

Status	DECIDED
Stakeholders	Hidde Lycklama Joris Oudejans
Outcome	Use architecture patterns in line with MVP model
Due date	12 May 2017
Owner	Hidde Lycklama

Background

For modularity, use per use-case interactors for presenters actions. Interactors communicate with the Data layer

The data layer is an aggregator for providers

Status	DECIDED
Stakeholders	Hidde Lycklama Joris Oudejans
Outcome	Create the data provider as a single programmatic interface.
Due date	12 May 2017
Owner	Hidde Lycklama

Background

Data layer is a component that implements data functions.

It can use one DataProvider (StrapeAccountModel) and multiple DataListeners (Strape / OK).

Communication is often async, so push based from DataProvider to Data layer to interactor. This means push notifications can be implemented on the same level as async calls This also means BLE Discovery can use the same system.

Sprint retrospectives

Here, the sprint retrospectives are attached. These were created in Confluence after every sprint to assess how the previous sprint went and what are the points of action for the next one.

2017-05-15 Retrospective

Date	15 May 2017
Participants	Hidde Lycklama Joris Oudejans

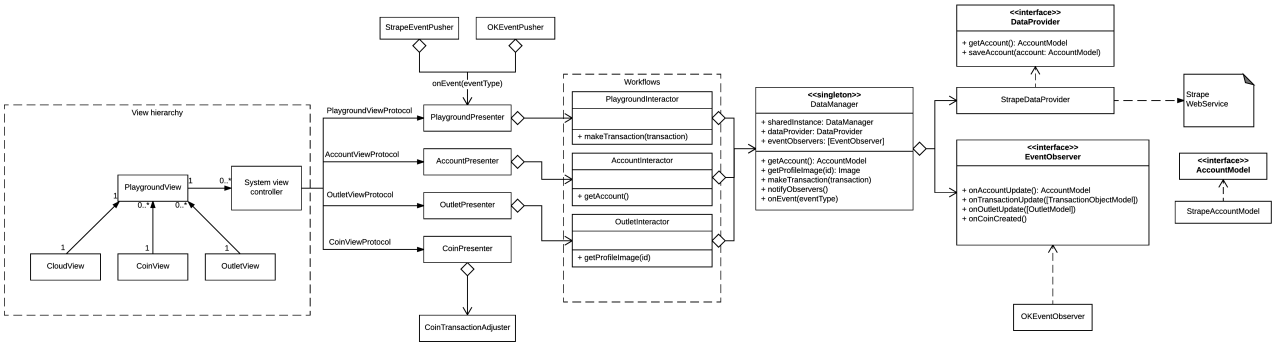


Figure 1: Updated software structure

Retrospective

This sprint we started with implementing the SDK structure. However, while implementing, we came across several subtle issues with the structure that came out of the research. Therefore, we came back to some implementation details from the research phase, which we revised. This resulted in some extra implementation work, but did not pose any problems. Meanwhile, we took care to keep both platforms in sync architecture-wise, by comparing implementations using PR's and by documenting details in Confluence.

What did we do well?

- Constant (daily, several times) sync about Android / iOS implementation details and differences
- Tooling set up, CI and deployment from the start
- PR peer-reviews
- Documentation in Confluence

What should we have done better?

- Plan the end of the sprint on a Monday, not a Sunday

2017-05-22 Retrospective

Date	22 May 2017
Participants	Hidde Lycklama Joris Oudejans

Retrospective

This Sprint we worked on transaction object implementation. This includes implementing the CoinView-based default functionality. However, this is a big feature with a lot of sub-features. On top of this, because this was the first implementation of the SDK framework, the development required extra time due to coordinating with the structures and algorithms. Therefore, we did not finish one feature: The Coin amount mutation via rotation. This has been moved to the next Sprint.

However, because the development was very feature-oriented, this missing feature is the only one missing and all features do work as planned.

What did we do well?

- A lot of working together in the office, syncing about software architecture and algorithm implementation
- Incremental, feature-based stories
- Test-driven development, writing tests for new components. Thinking about how we can improve testability for components

What should we have done better?

- The Sprint was very full - allocate less work for next Sprint and more possibilities for fallback.

2017-05-29 Retrospective

Date	29 May 2017
Participants	Hidde Lycklama Joris Oudejans

Retrospective

This sprint we focused on the last part of the functional requirements of the SDK. We created support for Outlets, account views and coins. Furthermore, we added peer-to-peer discovery via Bluetooth Low Energy. Although the sprint was quite full, we did manage to finish everything.

What did we do well?

- Fix issues from last Sprint + keep up with the current workload
- Continuous integration, continuous deployment and continuous documentation

What should we have done better?

- Allocate less work in a sprint

2017-06-05 Retrospective

Date	05 Jun 2017
Participants	Hidde Lycklama Joris Oudejans

Retrospective

This sprint focused on core functionality details, to make the SDK ready for integration. Furthermore, we created a plan for integration, as seen in 2017-06-02 Meeting notes: OK Integration

What did we do well?

- Sync about requirements per-platform
- Write extra tests for functionality

What should we have done better?

- Sprint planning was done mid-sprint, on Thursday

2017-06-12 Retrospective

Date	12 Jun 2017
Participants	Hidde Lycklama Joris Oudejans

Retrospective

This Sprint we worked on integrating the SDK into the OK app, visually and logically. The SDK is a standalone framework in the OK app, exactly like the integration with the ExampleSDK app. Both apps live side-by-side and rely on the exact same framework code. Furthermore, we have created a separate component in both OK apps (Android & iOS), to extend the SDK with OK-specific functionality, such as a different *cloud* and *playground* implementation. Moreover, we have integrated the SDK component into a logical place in the OK app. Namely in the Home view, on top of the Birdseye View.

What did we do well?

- Prioritize tasks that needed pair programming at the office. For example, for the Bluetooth Low Energy implementation it was necessary to debug two implementations together on separate systems.

What should we have done better?

- Planning with VAS integration may have to be changed due to this weeks results.

2017-06-19 Retrospective

Date	19 Jun 2017
Participants	Hidde Lycklama Joris Oudejans

We focused on finalizing the product to have a demo ready for the presentation. While we can see that there are so much more possibilities with regards to features, we feel that this is a complete, standalone product demo.

Retrospective

What did we do well?

- Deliver demo-able product

What should we have done better?

- Planning changes throughout the sprint

Bibliography

[1] Accenture. 2016 north america consumer digital payments survey, 2016. URL https://www.accenture.com/t20161013T024052__w__/us-en/_acnmedia/PDF-34/Accenture-2016-North-America-Consumer-Digital-Payments-Survey.pdf.

[2] Anonymous. What are MVP and MVC and what is the difference? URL http://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference.

[3] Apple. iOS Security Guide - App Security, . URL https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[4] Apple. UIPinchGestureRecognizer, . URL https://developer.apple.com/reference/uikit/uipinchgesturerecognizer.

[5] Apple. UIViewController, . URL https://developer.apple.com/reference/uikit/uiviewcontroller.

[6] Apple. View and Window Architecture, . URL https://developer.apple.com/library/content/documentation/WindowsViews/Conceptual/ViewPG_iPhoneOS/WindowsandViews/WindowsandViews.html.

[7] Vangie Beal. Wi-Fi (wireless networking). URL http://www.webopedia.com/TERM/W/Wi_Fi.html.

[8] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[9] Bluetooth.com. what is Bluetooth? - How it works. URL https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works.

[10] Capgemini. Top 10 trends in payments in 2016, 2016. URL https://www.capgemini.com/resource-file-access/resource/pdf/payments_trends_2016.pdf. Trend 03.

[11] Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP*, volume 2374, pages 231–255. Springer, 2002.

[12] Facebook. React: A javascript library for building user interfaces. https://facebook.github.io/react/, 2017.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of, 1994.

[14] Google. Android API: Activity, . URL https://developer.android.com/reference/android/app/Activity.html.

[15] Google. Android API: Fragment, . URL https://developer.android.com/reference/android/app/Fragment.html.

[16] Google. Building Instrumented Unit Tests, . URL https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html.

[17] Google. Requesting Permissions at Run Time, . URL https://developer.android.com/training/permissions/requesting.html.

[18] Google. Security Tips, . URL https://developer.android.com/training/articles/security-tips.html.

[19] Google. Android Training: Dragging and Scaling, . URL https://developer.android.com/training/gestures/scale.html.

[20] Google. Android API: OnScaleGestureListener, . URL https://developer.android.com/reference/android/view/ScaleGestureDetector.OnScaleGestureListener.html.

[21] Google. UI Overview, . URL https://developer.android.com/guide/topics/ui/overview.html.

[22] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java language specification*. Pearson Education, 2014.

[23] Apple Inc. *The Swift Programming Language (Swift 4 beta)*. Apple Inc., 2017.

[24] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005. ISSN 2070-1721. URL https://www.rfc-editor.org/rfc/rfc4122.txt.

[25] Carolyn Lowry. What's in your mobile wallet? an analysis of trends in mobile payments and regulation. *Fed. Comm. LJ*, 68:353–353, 2016.

[26] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2000.

[27] James E McDonough. Decorator design pattern. In *Object-Oriented Design with ABAP*, pages 207–224. Springer, 2017.

[28] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[29] International Institute of Business Analysis and K. Brennan. *A Guide to the Business Analysis Body of Knowledger*. International Institute of Business Analysis, 2009. ISBN 9780981129211. URL https://books.google.nl/books?id=u_AjngEACAAJ.

[30] Mike Potel. MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, 1996.