FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

SmartMaut

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Natalie Kollarits
Matrikelnummer 0926244

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. eva Kühn

Wien, 23.01.2015

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

SmartMaut

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering/Internet Computing

by

Natalie Kollarits
Registration Number 0926244

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. eva Kühn

Vienna, 23.01.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Natalie Kollarits
Am Nussfeld 48, 7343 Neutal

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

Hereby, I want to thank my advisor, eva Kühn, for the great collaboration. It helped a lot that we had regular meetings in which the current status of the work could be discussed. Furthermore, she was always available if some serious questions turned up.

Also Alexander Fürdös needs to be thanked for providing the excellent idea of starting the ”SmartMaut” project, which resulted in the diploma thesis at hand.

Of course, my friends and colleagues, Stefanie, Stephan, Matthias, Georg, Benjamin, Manuel and David need to be mentioned, too. We did some lectures together and they added much fun to my studies.

Last, but not least, I need to thank my family. Especially my parents and boyfriend, who supported me throughout all my years of study and built me up whenever I experienced hard times.

Abstract

Toll systems are very important in our society. They aim at various tasks, from refinancing the construction and maintenance of road infrastructure over the reduction of congestions to the limitation of car emissions, in order to help saving our environment. Current toll charging systems, however, have some serious drawbacks and interoperability among European toll charging systems still needs to be solved satisfactorily. On the other hand, smart devices, like smart phones and tablets are owned by an increasing number of people. These devices can host mobile applications and could therefore act as an On Board Unit (OBU). This is why, in the presented thesis, a mobile application, which provides the features of current OBUs is developed for the Android platform. It implements the functionality of a location based toll charging system, with the help of virtual points, which can be used to generate bills. In order to see if this is not only possible technically, but if such an application would still remain usable, usability tests have been performed and evaluated, too. Such a system would be usable throughout Europe, easy to deploy, and easy and comfortable to use.

Kurzfassung

Mautsysteme sind sehr wichtig für unsere Gesellschaft. Sie bezwecken die Erfüllung verschiedenster Aufgaben, von der Refinanzierung der Konstruktion und Erhaltung von Straßeninfrastruktur, über die Reduktion von Staus, bis hin zum Senken von Schadstoffemmissionen durch Fahrzeuge, um zum Schutz unserer Umwelt beizutragen. Derzeitige Mautsysteme haben allerdings einige schwerwiegende Nachteile und die Zusammenarbeit europäischer Systeme ist noch immer nicht zufriedenstellend gelöst. Andererseits besitzen immer mehr Menschen intelligente Geräte, wie Smart Phones und Tablets. Diese Geräte können mobile Applikationen verwenden und eignen sich daher als ”On Board Unit” (OBU). Deshalb wird für die vorliegende Diplomarbeit eine mobile Applikation für die Android Plattform entwickelt, die die Funktionalität einer derzeitigen OBU zur Verfügung stellt. Sie implementiert die Funktionalität eines leistungsbasierten Mautsystems, welches mit Hilfe virtueller Punkte, so genannter M-Points, Rechnungen erstellen könnte. Um festzustellen, ob dies nicht nur technisch umsetzbar ist, sondern die Applikation auch verwendbar bleibt, wurden auch Usability Tests durchgeführt und ausgewertet. Ein solches System wäre europaweit einsetzbar, einfach zur Verfügung zu stellen und einfach verwendbar.

Contents

CHAPTER 1

Introduction

Toll systems are very important in our society. They can be used to accomplish various tasks. These tasks range from refinancing the construction and maintenance of road infrastructure over the reduction of congestions to the limitation of car emissions, in order to help saving our environment. Of course, the focus on these tasks depends on the country that implements the corresponding system.

As there are various use-cases for toll systems, the toll business is growing steadily and new systems emerge continuously (where the current discussion about launching a toll charge for passenger cars in Germany is only one example for this development), while existing systems are extended and technically modernised.

Additionally, toll charging systems do not only affect the people living in the country that implements them, but also each visitor from abroad, who wants to use the toll charged infrastructure. Therefore, great care has to be taken when such systems are designed or implemented. This can also be seen very clearly in the current discussions in Germany, where toll charges for passenger cars shall be implemented, while German drivers shall not need to pay more for their cars and driving activities than they do today and foreign drivers must not be discriminated against, according to EU regulations. So, the fact that one toll charging system affects multiple countries increases the complexity of its design and elevates it to a very hot political issue for all potential stakeholders.

This leads us to the major drawbacks of current approaches, which are as follows:

- Current toll charging systems usually are expert systems that cause very high investments for infrastructure and also for the continuous operation for the system provider.
- Due to the first drawback, there are only a few providers all over the world.
- In most cases, the acquisition of the right to use toll charged roads (vignette, OBU etc.) is complicated for customers.
- Although desirable, interoperability among European toll charging systems still needs to be solved satisfactorily.

These drawbacks are also responsible for the fact that the necessary continuous extension and adaptation of existing systems, to meet the altered necessities of the market, are very complex and expensive.
On the other hand, new technologies evolve constantly. As they are decentralised, they bear the potential for cost savings. Smart devices, like smart phones or tablets, are an example for such a new technology. Additionally, more and more people of the European population own at least one smart device. However, smart devices and data management technologies are barely used by the conservative toll industry.

To overcome these issues, this thesis will evaluate if the functionality of current toll charging systems can be implemented in an application for smart devices that will enable ad-hoc deployment for any interested country, worldwide. As most potential users possess at least one smart device, and such a solution needs hardly any additional infrastructure, it will be cheaper for providers and much more convenient for users. In addition to being flexible (e.g. allow the change of payment models), the proposed application shall also ensure that users remain as anonymous as possible, in order to protect their privacy. Of course, usability is a very important factor for the success of such an application, too, which is why this thesis will not only focus on implementing the needed functionality in an efficient way, but it will also try to figure out, how users are best led through all these features, such that using our application feels as natural as possible. These design decisions will be validated by usability tests.

Naturally, known regulations will be taken into account for the design and implementation of the afore-mentioned application. So, this approach will not only support currently existing toll charging systems, but it will also be able to deal with the requirements of possible new systems.

Please note, that the concrete application that was implemented for this thesis is a prototype for a revolutionary toll charging approach, based on the user's own smart phone. This means that the resulting application serves as a proof of concept and is no full-fledged product. To avoid confusion, tasks that are out of scope of this work will be listed below:

- Although it is focused on designing the user interface in a way that will help the user to understand how and where desired tasks can be done, the graphical design of the user interface will not receive any attention. Of course, the interface will follow the basic guidelines, as given by Android [1], but default colours will be used and elements will be positioned as dictated by the used layout, or as assumed to be logical, based on the design of other applications.

- The implementation of the back-end will be handled in two other diploma theses, which are currently in progress and is therefore out of scope of this work. The back-end functionality that is needed to illustrate the basic idea of our system will be simulated by a local database.

- As the back-end, used for this proof of concept, is a local database, it was decided to declare the encryption of the communication between the application and its back-end to

[1]Android API Guides On User Interfaces. http://developer.android.com/guide/topics/ui/index.html Accessed December 2014

be out of scope of this work, as the usage of such mechanisms is well known and would not add any additional value to the results.

- Due to the fact, that the application resulting from this thesis is a proof of concept, it will only be developed for the Android platform, as a corresponding device is available to the student. Implementations for other platforms will be subject to future iterations of this application.
- Finally, the proof of concept resulting from this thesis will only implement the features needed for toll collection in a so-called location based system. This is, because implementing a platform that enables users to buy a vignette would basically result in implementing a webshop. However, it is already well known that webshops can be used with smart devices. Also, most mobile Apps provide the possibility of in-app purchases, through which further features can be bought (e.g. ”runtastic” [2] is basically free to use, but provides the possibility to buy features, like recording used paths).

So, to summarise this, the presented thesis will mainly try to answer,

- if it is possible to implement the functionality of current toll charging systems (especially location based ones) as an application for smart devices,
- if all necessary features can be implemented, while the application remains usable and the user can stay anonymous and
- if such an application could replace an existing toll charging system, where expensive infrastructure is used; thus, reducing the complexity and costs for implementation, maintenance and continuous operation of toll charging systems

1.1 Methodological Approach

To achieve the results, the following steps will be taken:

1. literature research: find out, how current systems work, which technical solutions they use and which functionality an OBU provides
2. design the user interface and the architecture for the prototype application ([1], [10])
3. implement the needed functionality: (this proof-of-concept will only be developed for the Android platform, as a corresponding device is available to the student) possibilities/data offered by the device (like GPS) will be used ([4, 5, 15])
4. usability tests for the functional interface: (where it has to be noted that the graphical design is out of scope of this work, as already mentioned in the previous section) ten users will be asked to use the resulting application and to give feedback about their experiences. ([2, 3, 6, 7])

[2]runtastic - mobile sports App. https://www.runtastic.com/de Accessed December 2014

1.2 Structure of the Work

Chapter 2 describes the state of the art of toll charging systems, mobile applications for paying toll charges and other related work. Chapter 3 describes some basic thoughts about the ”SmartMaut” application, along with usage scenarios and corner cases which were detected and discussed. The actual implementation of the prototype will be described in chapter 4, followed by a report about the usability tests in chapter 5. In chapter 6, some ideas for future work are collected. Finally, chapter 7 concludes this thesis.

CHAPTER 2

State of the Art and Related Work

2.1 Existing Toll Charging Systems

Current toll charging systems are different for each country, as there is no unified European system, yet. Some countries might not have toll charging systems at all, or just for certain cities or buildings (e.g. expensive bridges or tunnels). Furthermore, in most systems lorries and cars are treated differently.

In cases, where toll charging systems are installed for city centres, this aims at reducing congestions and/or keeping emissions low by reducing (unnecessary) traffic. Aside from regulatory purposes, toll charging systems are installed in order to refinance the construction and maintenance of expensive infrastructure.

So, currently, different countries run toll charging systems for different road types (like motorways, clearways and urban motorways), for special infrastructure (like bridges or tunnels) or for certain areas (like city centres). The mode of payment may differ for each of these toll charged infrastructure types and also for cars and lorries.

To make this a little clearer, the toll charging systems that are installed in Austria will serve as an example:

Basically, toll charges have to be payed for motorways, clearways and urban motorways. For being allowed to use these types of roads with a car, a vignette, which needs to be sticked onto the car's windscreen, has to be bought. This way, a specific car obtains the permission to use these roads. Registration number and driver may change, as they are not linked to the car. The main advantage of vignettes is that the anonymity of users remains untouched. Furthermore, vignettes allow to use the infrastructure for a specified amount of time, regardless of how often the infrastructure is used during that time [1]. Therefore, this is an example for a time based system.

[1]ÖAMTC – Vignette 2014. http://www.oeamtc.at/?id=2500,1373867 Accessed September 2014

Lorries, on the other hand, are charged regarding to a location based system. In Austria, this means, that they need to have an OBU, which communicates with gantries, that are installed alongside the roads [2]. So, payment has to be made for every driven kilometre. Moreover, the emission class of the used vehicle affects the toll charges, according to a mandatory action of the EU (see [3],[4]). Of course, also the number of axles is used to calculate the amount payable [5]. Typically, payment is made via credit cards or similar payment options that do not directly affect the driver.

For special toll charged roads, like, for example, the "Brenner-Autobahn" [5], however, toll plazas are installed, where physical tickets have to be bought against cash payment. Of course, this implies waiting times. Additionally, building toll plazas is very expensive and requires quite some space. An alternative to cash payment has been created with the introduction of the video toll card, which can be purchased in advance via SMS, using an application for smart phones or online. This prepaid ticket enables drivers to pass the toll plazas on an extra lane, where the registration number will be photographed to check the validity of the ticket (see [5]). So, here we basically have a location based system, too, but in this case, it also applies to cars.

A detailed report about the toll charging systems implemented in Austria, Hungary and Sweden can be found in the project report [8]. The following will give a quick overview of the used models:

country	car	lorry
Austria	time based ([1])	location based ([5],[6])
Hungary ([7],[8])	time based ([9])	location based
Sweden ([5])	city/special-toll	city/special-toll

Table 2.1: Classification of toll charging systems for Austria, Hungary and Sweden

Looking at the table above, mentioning the Hungarian toll charging system in addition to the Austrian one might seem to be redundant, but it is really not. There are some differences, regarding the mode of payment. The first striking difference is, that Hungarian vignettes are e-vignettes. They can either be bought via the Internet or in shops like the tobacconist's or at

[2]Austria GO-Box. https://www.go−maut.at/portal/faces/pages/common/portal.xhtml Accessed December 2014

[3]EU Wegekostenrichtlinie. http://eur−lex.europa.eu/legal−content/DE/TXT/PDF/ ?uri=uriserv:OJ.L_.2006.157.01.0008.01.DEU Accessed December 2014

[4] Gebrüder Weiss: Mautgebühren Österreich. http://roadpricing.gw−world.com/de/ MautgebuehrenAT.aspx Accessed December 2014

[5]ARBÖ − "Mautgebühren in Europa 2013". http://www.arboe.at/uploads/media/ maut_europa_2013_01.pdf Accessed September 2014

[7]Nationale Mauterhebung geschlossene Dienstleistungs Aktiengesellschaft − FAQ toll Hungary. https://www.hu−go.hu/media/events/thumbnails/5440/ 1397566734FAQ140415.pdf Accessed September 2014

[8]Nationale Mauterhebung geschlossene Dienstleistungs Aktiengesellschaft − HU−GO. https://www.hu−go.hu/articles/index/3194 Accessed September 2014

[9]ÖAMTC − "Ungarn Maut & Vignette". http://www.oeamtc.at/portal/ungarn-maut-vignette+2500+1055201 Accessed September 2014

a filling station. However, buying the ticket via the Internet, results in an additional fee of 100 HUF (= ca. 0,30 EUR). Moreover, the Hungarian vignette is bound to the registration number of the car, not to the car itself, like the Austrian vignette. (see [7],[8], [9])

Furthermore, in Austria, it is obligatory for lorries to use OBUs. In Hungary, carriers can choose between using an OBU, which is recommended when trips have to be taken via Hungarian roads, regularly, or buying a prepaid ticket, which is recommended if only very few trips are taken via Hungarian roads. For prepaid tickets, the planned route and vehicle data have to be provided, in order to buy the ticket. Both possibilities correspond to a location based system. (see [7],[8])

Even if a carrier decides to use an OBU, in Hungary, there are two possibilities (see [7],[8]):

1. the "plug and play" OBU, which can be bought at filling stations and has to be configured by the customer. Shops provide equipment to configure the most important things, but some information has to be entered at home. As soon, as this device is plugged into the cigarette lighter, it starts operating.

2. a built in OBU, which is incorporated into the lorry itself and needs to be configured and installed by trained personnel.

In Sweden, there is no toll obligation. Only city-tolls and special tolls for buildings like bridges are charged and only Swedish vehicles have to pay charges.

So, to sum this up, toll systems may either be time or location based and one country might implement a mixture thereof, where time based systems can be used with prepaid tickets and location based systems require OBUs, gantries (for microwave communication, like in Austria, see [10]), cameras, toll plazas (for manual toll collection, like in Italy) and satellites (like in Germany). Providing and operating different systems is also expensive.

2.2 On Board Units

So far, OBUs have been mentioned several times as a device that is needed for automatic toll collection for location based systems. As this device is omnipresent, and we want to implement its functionality in this thesis and some extra features in a future project, it was decided to collect as much information about OBUs and their current functionalities as possible.

Unfortunately, this amount of information turns out to be very little, as there seems to be no detailed description of the technical aspects of an OBU. Short “step-by-step – guides”, on the other hand, can be found on the website of each provider of OBUs, but they only show how to use the device to pay toll charges correctly and do not say if there are any additional features or how these devices work exactly.

Nevertheless, some papers and patents which describe systems that might be considered to be related to our idea or give a very basic overview of the capabilities of the used OBUs, at least, were found. The most interesting facts will be summarised in the following paragraphs.

[10]ASFINAG – overview "Zehn Jahre Go-Maut". http://www.asfinag.at/maut/maut-fuer-lkw-und-bus Accessed September 2014

Although no paper proposing a basic OBU was found, in [12] the information we were looking for turned up in one sentence: “The main purpose of vehicle on-board computing and communication systems is to collect and process information on the current positions of moving vehicles in real time.” So this ensures us, that a basic OBU will really only collect positioning data in order to account correct toll charges.

While looking for this essential piece of information, however, other ideas for designing or enriching OBUs, such that they will be as suitable and convenient as possible were found. In [12], for example, it was proposed to implement a communication from the OBU to its “home-base” (e.g. the office of a carrier), such that it can be checked where one’s vehicles are and it becomes possible to redirect them, if necessary, or simply check their position or speed. The location information can also be used to check if a vehicle with a very valuable or dangerous load is still on its planned route, or if it is heading elsewhere. Additionally, there is an emergency button on the OBU, which, when pressed, will send an emergency call, containing its current position, to the control centre (from where these calls have to be redirected if this should be necessary). Another feature the authors of [12] included is the capability of the OBU to read any desired map (which includes existing printed street maps). This data, together with the information about where vehicles of the fleet currently are, allows the display of vehicles on the map. Via a click on the vehicle in question, one can see detailed information about it. Furthermore, it is possible to view a vehicle’s route, displayed as a line of dots. Position is measured continuously, which allows to determine the vehicle’s speed out of the distance between two positions.

The OBU of this system has a module for receiving GPS information, may process data and is also capable of mobile communication. Power is drawn from the vehicle that uses the OBU. It also has a rechargeable battery, which allows the detection of failures of external power supply that last too long. In such a case, the control centre is alerted.

The standard mode of operation detects the current position and sends this information to the control centre at regular intervals. It can be configured, for which events the control centre should be contacted. For emergency situations, it is also possible that the OBU immobilises the vehicle. Additionally, OBUs have got an I/O interface, which allows them to interact with in-car systems. This interaction allows to determine if the engine requires service or maintenance. Furthermore, the OBU enables drivers and control centres to exchange written messages or to initiate voice telephone connections.

OBUs can even compose and send messages autonomously e.g. to report safety violations or emergency calls.

The authors claim that this system even allows to create maps of atmospheric pollution in real-time. Furthermore, their OBUs can do the following: ”For safety-sensitive applications, board units can switch to back-up data services (as, e.g. satellite links) whenever the primary communication links fail.”

So, to sum this up, [12] proposes a classical OBU (i.e. one that needs to be built into the vehicle) which provides several additional features, which are related to our idea of providing a mobile OBU, capable of basic tasks for toll payment and offering added value services to users.

In [11] an OBU that seems to be an extension of the one in [12] is proposed. This unit provides predefined interfaces which can be used to extend its functionality by additional services.

The main feature in this paper, however, is that the OBU synchronises its clock via the timing information received from the GPS-satellites, together with the positioning data.

[1] describes not only some OBU, but a whole system, which shows another possibility for using the data known by the OBU. This paper illustrates an "automatic vehicle access control system which is capable of controlling and reducing the traffic inside the historical centre" (of Rome). The system is called "Iride" and works with OBUs, which use vehicle-specific details that are stored on a smart card that has to be provided. Generally, this system aims at checking if a specific vehicle is allowed to enter a specific road and not on collecting toll charges. In order to being able to check access rights, microwave emitters are needed to communicate with the OBU and cameras are needed to check a vehicle's registration number. However, the authors of [1] already thought of the extension that would allow to use "Iride" to charge time-based fares.

The following sections will deal with information found in patents. (Generally, this information is very vague.) [9] describes that in a closed toll charging system, an OBU communicates with the signalling devices at the entrance/exit of the toll road, in order to effect toll payment, while a vehicle without an OBU has to pay manually (where the actual amount is determined via a pass that is handed to the driver at the entrance and is collected at the exit). This first part is basically a description of a closed toll charging system. The OBU is vaguely described later on as something with a memory, holding data regarding the vehicle which has been stored in advance. This data includes the vehicles dimensions, the number of axles, its weight and the like. The vehicles registration number and its class, as well as the unique ID of the OBU itself, are also stored.

According to this reference, for their system to work, a car with an OBU has to be detected by six different vehicle detectors. Some of the detectors are needed to make sure that the vehicle is really in the communication area of the system and other detectors are needed for exchanging data for toll collection with the OBU. Vehicles which pay a reduced charge, are identified via the according information stored in the OBU and the registration number that is recognised by cameras.

[13] describes an OBU that does not seem to be intended to be used for the collection of toll charges, but it is capable of communicating with other OBUs, such that the driver can be alerted when the preceding vehicle brakes. Thus, it rather implements an added-value-feature, which aims at enhancing safety, than fulfilling the basic toll collection use case.

The OBU described in [14] is similar to the one in [13] in that it also provides peer to peer communication among OBUs. Basically, this communication is needed to determine the congestion condition according to the vehicles that are determined within a certain range, which is displayed to the driver via the OBU.

2.3 Existing Mobile Applications

The mobile application that will result from this thesis aims at collecting toll charges in a location based manner. This will be implemented using the GPS location data, provided by the sensors of the used device. Clearly, it is very interesting if such or similar applications already exist.

Existing mobile phone based toll (support) systems have already been assessed in the project report [8]. Therefore, the findings about these systems will only be summarised, in the following.

Generally, it seems like there are not too much mobile phone based toll (support) systems for usage in the EU. Additionally, the few systems that are being provided are rather simple and not suitable for the usage as an easy-to-use European system, which can be configured to be used by each interested (European) country easy and fast.

Paybox

Paybox offers the possibility to pay toll charges via one's mobile phone, in Austria (see [11]).

In order to use this service, one has to register with a combination of mobile phone number and the car's registration number. However, this registration does not need to be a separate step. It may also be part of the first order.

Tickets are bought via sending an SMS, containing the keyword "MAUT" and the name of the toll charged road that shall be used. - Every mobile phone is supported by this system and it is not reduced to smart phones. With such a ticket, the special lanes that are reserved for video toll cards may be used, at the checkpoints, where special toll charged roads are entered. Thus, this system only supports special toll charged roads, for which it aims to reduce waiting times. Still, a vignette has to be bought for all other toll charged roads. Additionally, one has to consider that the mobile phone number will be linked to the registration number of one's car.

According to [12], the balance of such a transaction is deducted from the users bank account.

A1 Handy Maut

This system is also available for Austria and uses "paybox" as a means of payment for non-A1 customers, while customers of A1 have the possibility to pay via an automatic debit transfer system or with their next payment of their A1 invoice (see [13]).

"A1 Handy Maut" also provides the possibility to buy tickets for special toll charged roads via one's mobile phone. It works just like "paybox".

Unfortunately, it was not possible to find out, whether "A1 Handy Maut" only provides additional means of payment that are not available in "paybox" or if it is to be considered as an individual service.

SMS-Maut (DE)

As the name already suggests, this system was made to ease payment of toll charges in Germany (see [14]). Currently, there are no toll charges for cars, so this system targets carriers. In addition to buying tickets, it is also possible to e.g. redeem or just view tickets that have already been bought. In order to keep the system fully automated, offering this variety of actions requires the usage of predefined keywords.

Of course, one has to register before toll payments can be made, using this system. It has to be noted, that toll charges will be deducted from the balance on one's user account only.

[11] paybox − HANDY−Maut. http://www.paybox.at/6218/Privat/Services/HANDY-Maut Accessed January 2015

[12] paybox − how to pay. http://www.paybox.at/6192/Privat/Bezahlen/Wie-bezahlen Accessed January 2015

[13] A1 Handy Maut. http://www.a1.net/apps/privat/a1commerce2/maut/pin.seam Accessed January 2015

[14] SMS−Maut (DE). http://www.sms-maut.de/de/description.aspx Accessed January 2015

Clearly, deductions from bank accounts or credit card payments can only be used to refill one's user account.

”SMS-Maut” has also got a bonus system: bonus credits can be obtained for bringing new customers and these bonus credits will never forfeit. However, such bonus credits cannot be used for the payment of toll charges.

As this system targets carriers, it is possible to register several mobile phones and vehicles, which may be used in any combination. The last combination is remembered by the servers, such that repeated usage of the same combination will result in easier interactions with ”SMS-Maut”.

Additionally, the possibility for payment via WAP or one's own program, which uses the available webservices, is provided. So, this system provides much more features and is thus, much more complex than the above mentioned ones, while it can still be used with every mobile phone.

PayUrToll

Detailed information, regarding this system, can be found in [15]. ”PayUrToll” aims at reducing the time spent at toll plazas and at reducing the need for pernonnel, which needs to collect toll charges at toll plazas. It is claimed that the system works globally, but contrary to the systems described above, where any mobile phone could be used, one needs a smart phone or a computer for being able to use ”PayUrToll”. For people, who cannot access this service via the mobile application (due to the lack of a smart phone), it is also available via a website.

Again, a registration is required. Details about one's car and a preferred means of payment have to be provided.

For the actual toll payment, ”PayUrToll” communicates with existing electronic toll systems. So it seems, like this application can only be used when the required road-side infrastructure already exists.

This application is especially interesting for us, as it is designed for the use with smart phones. Additionally, it does not only provide the possibility to pay toll charges in a convenient way, but it also aims at being usable globally and at enabling the payment of all traffic-related costs, which would also include costs for parking or public transport. These features are among the ones which have been considered for future implements of our own application.

WAZE

A detailed analysis of WAZE can be found in the project report [8]. This application does not provide the possibility to pay any traffic-related charges, but is a free navigation system. What makes it interesting for us is that it is very popular among users of mobile application. It also provides interesting additional features. This is why it was decided to analyse it, to find out, what people like so much about it, as such findings would also help our application to gain popularity.

Lots of information about WAZE can be found in the Internet. Most of it is available via the provided user guide [16] and the WAZE Wiki [17]. These are also the main sources for our analysis.

[15]PayUrToll. http://payurtoll.com/ Accessed January 2015

[16]WAZE Userguide. https://wiki.waze.com/wiki/Anleitung_f%C3%BCr_Benutzer Accessed January 2015

[17]WAZE Wiki. https://wiki.waze.com/wiki/ Accessed January 2015

In addition to the Internet-research, we also tested the application ourselves.

Short Description of WAZE

As already mentioned above, WAZE is a navigation application. In addition to this main functionality, it also provides several features, like personal avatars which may express the drivers' mood and internal rank (see Figure 2.1 and Figure 2.2). Furthermore, one can communicate with fellow drivers and import facebook-friends or appointments with specified locations (such that WAZE will be able to navigate to these locations). If one wants to pick up someone else, who also uses WAZE, a request for that person's current location can be sent, such that WAZE can navigate to that person. If another person, for example the one who will be picked up, shall be informed about the driver's progress, the application enables us to share the drive. This means that the other person will receive a map where the driver's car (which is represented by the driver's avatar) can be seen. Additionally, WAZE is able to show the nearest point of interest, like a filling station, café or restaurant.

Apart from these features, symbols, which will be displayed to inform drivers about certain events along their route, like construction sites, are provided (see Figure 2.3).

WAZE lives from user contribution. This means, that users are able to warn others about construction sites or other obstacles along one's route. Additionally, roads can be recorded, configured and corrected by users. Usage of the application and contribution of knowledge or work is rewarded by points. These points will, in turn, enable users to gain a higher rank.

The ranking system shows, how much one uses WAZE, compared to other "wazers". One starts as a baby and may evolve over time. There is also a ranking system for contributions to the forum and maps. To gain these ranks for forums and maps, however, one requires approval by WAZE administrators. The higher the rank, the more permissions are granted.

Although, WAZE provides various possibilities for communicating and interacting with other users, one can also become invisible, if this is wanted: "At any time, you can switch to invisible mode. Going invisible hides you completely for the duration of your drive. You won't be seen on the map, you'll appear to friends as 'offline', and any reports you make during your drive will be seen by others as if sent by 'anonymous'. Note that in invisible mode you still get points as usual. [...] Please note that you'll remain invisible only for the duration of your current session. Next time you start WAZE you'll automatically return to being visible." [18]

Finally, spoken orders are possible with this application and the user account can easily be deleted at any time.

Discussion

From the description above, it can be seen that WAZE offers many useful and nice features that aim at improving navigation and that enable communication among users. Aside from an interesting design, however, it is important that an application does not request permissions which are not absolutely necessary for its operation, in order to be accepted by users who still care about their privacy. As WAZE offers numerous features, which are all related to navigation and

[18]WAZE Userguide section "Unsichtbar". https://wiki.waze.com/wiki/Unsichtbar Accessed January 2015

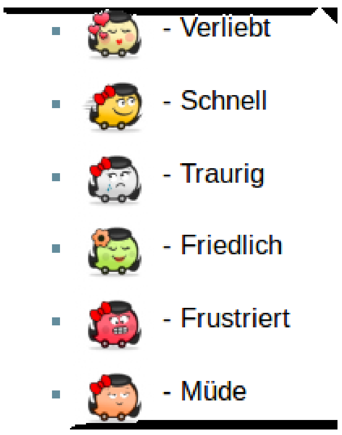


Figure 2.1: Moods that can be chosen for one's avatar in WAZE. From top to bottom they are: "In Love", "Fast", "Sad", "Friendly", "Frustrated" and "Tired"

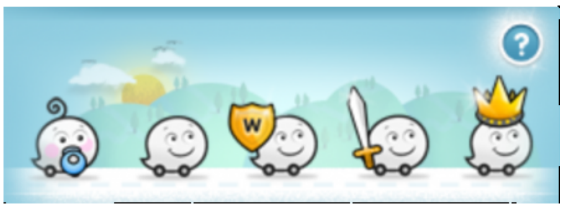


Figure 2.2: Ranks that can be achieved in WAZE

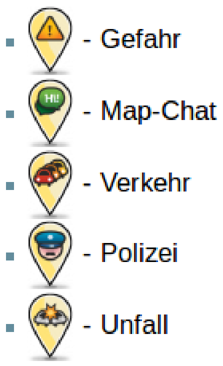
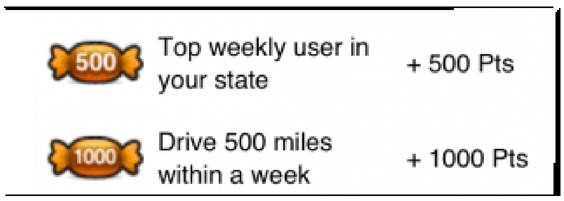


Figure 2.3: Symbols that will mark important events on the map in WAZE. From top to bottom they stand for: "Danger", "Map-Chat" (this is a live chat with fellow drivers), "Traffic", "Police" and "Accident"



500	Top weekly user in your state	+ 500 Pts
1000	Drive 500 miles within a week	+ 1000 Pts

Figure 2.4: Examples for possible achievements in WAZE

communication, it also requires permission to access hardware controls, network communication and the like. The only requested permission, which could not be related to any of the offered features is ”personal information”. That one made us believe that WAZE might try to spy on us, but of course, there is no proof for that.

In general, the possibility to easily contribute to the system appears to be very tempting. It seems like this motivates to use the application and to spend one’s spare time to help to improve the system. Additionally, WAZE generates a game-like atmosphere by granting so called achievements (see Figure 2.4). Achievements will be rewarded with points and most of them are relatively easy to get, like driving a certain amount of kilometres with the application turned on. Furthermore, they provide ”road goodies” on special days, like holidays. These are virtual sweets that appear alongside the road and can be collected by driving over them. Each road goody will earn you points.

Most additional features from WAZE, like rerouting in the event of traffic jams, however, require an active connection to the Internet. Navigation also works with GPS only, but only as long as no new parts of the map need to be downloaded. So, in order to provide its full service, WAZE would reload the map regularly, while driving, if an Internet connection is available. However, ”this might cause high traffic and also high costs”, as was already noted in [8].

Analysing WAZE also led to new ideas on which features might be offered in an own application. The list of these ideas can be seen in [8]. What unites them is the idea of rewarding users for actions taken with the application turned on, or for actions that could improve the application. Furthermore, it could be seen that most users will want to communicate with others and that they might also want to share ranks or other information, related to our application, on social platforms like facebook or twitter.

Finally, it is believed that the possibility to report obstacles and similar things that might be dangerous or annoying satisfies people and might even contribute to calming them down, as they can immediately warn others and the authorities might take steps to improve the situation in the near future.

Using Geo-Fences in Mobile Applications

In [19], the results of a workshop are described. In this workshop students built a mobile application, using Geo-fences and tested it outside. Geo-fences are available in the Android-API. They

[19]Thinking About Design With GEO−Fences. https://civic.mit.edu/blog/erhardt/thinking-about-design-with-geo-fences Accessed January 2015

can be defined using an application and may also be hard coded. In general, they describe a circle-shaped area. Upon entering, exiting or remaining in this area, certain things may happen (see [20]). As this concept is used for our application, reports about experiences with using it are very welcome.

In [19], the following definitions are given: "

- Geo-location: identifying the real-world location of a user with GPS, Wi-Fi, and other sensors
- Geo-fencing: taking an action when a user enters a geographic area
- Geo-awareness: customizing and localizing the user experience based on rough approximation of user location, often used in browsers

"

As these three terms are very similar, and as it is very important to keep them apart, it was decided to cite them here, to avoid confusion. In the following, the findings of [19] will be summarised.

A general problem with Geo-fencing is the accuracy of the location that can be obtained for a certain user, which can be influenced by various factors, like device settings or the user's whereabouts. This affects design decisions, like the size of the radius for a given Geo-fence. Areas with small radii might not be detected if the user location is very inaccurate. Also the user's speed of movement, together with the update interval for the user location will influence whether a certain Geo-fence will be detected or not. Frequent location updates, however will drain the battery of the used device.

Of course, also the amount of data and the frequency of sending data to or from the application have to be considered, to prevent users from having to pay a huge amount for data transfer, as is also pointed out in [19].

2.4 Designing and Implementing Mobile Applications

When a mobile application shall be implemented, this leads to the need of making further decisions. First of all, it has to be agreed upon a development platform. In [4] and [5], the most popular runtime environments have been compared to each other. However, these studies are from the years 2010 and 2011. Back then, development of mobile applications had just started, as smart phones gradually gained popularity. Thus, the results might be different, today. Furthermore, none of them considers Apple's iOS platform. Nonetheless, one can get an overview over possible platforms and their various pros and cons.

Android, which is the chosen platform for this thesis, as the device that is available for testing is Android based, was relatively young, when the aforementioned studies were made. Therefore, the developer community was not as big as for other platforms, but it was growing

[20]Android Developers Training – Creating and Monitoring Geofences. http://developer.android.com/training/location/geofencing.html Accessed January 2015

steadily. Moreover, [4] thought, that the Android community would continue to grow, which obviously was a correct assumption.

[5] found that Android was developer friendly. It was also perceived to be very interesting, due to its open platform, which means that the software can be adapted to any special needs. Additionally, the provided API was constantly evolving, already. Both studies described the platforms they investigated. However, [5] provided a very compact description of the main characteristics of the Android system. As it is still true for the version of Android that was used for the diploma thesis at hand, this description is cited in the following: "Android is an open source mobile operating system based on the Linux kernel. The Android platform does not only provide the mobile operating system itself including the development environment, it also provides a custom built virtual machine (Dalvik Virtual Machine) for the applications to run on as well as acting as the middleware between code and operating system."

The platform that seemed to be promising a few years ago is very successful, indeed. According to [21] and [22], Android had a market share of nearly 85% in 2014.

As important as choosing one or more target platforms might be, developing mobile applications comes with further considerations. As [15] states, mobile applications may be "native", which means that they are running on the mobile device as a whole, or they may be web applications. The latter have a small client, based on the device, while the actual execution is done on a remote server. Based on the requirements and goals of an application, one of these forms has to be chosen.

[15] also points out that mobile applications come with some additional requirements with regard to software engineering. Only some of them are picked out, here. For example, it has to be considered that an interaction among different applications is possible in general. Furthermore, mobile devices have got a variety of sensors built in. These may be used by one's App. Closely related to using device sensors is considering power consumption. Users will absolutely not be happy, if an application makes extensive use of different sensors, such that their device's battery will be drained. Thus, one has to think of whether the usage of any specific sensor is really necessary and when it would be best to use it, if so, in order to save battery. Last, but not least, user interfaces are much more important in mobile applications than in traditional desktop applications. Developers cannot make any assumptions as to where their program might be used and the screen size is limited. Thus, the interface has to be very simple, assuring that users will find the most important features at one glance. Additionally, there are different user interface guidelines for each platform. These guidelines seek to assure that "native applications for a device will share a common 'look and feel'." Moreover, users expect that a standard set of gestures can be used among all applications, running on a certain platform.

However, as [15] states correctly, "the Developer's Guide for Android includes a Best Practices section that addresses application compatibility, user interface guidelines, and designing for performance and responsiveness, among other things". The API Guide for Android can be

[21]Report: Android reached record 85% smartphone market share in Q2 2014, Xiaomi now fifth-largest vendor. http://thenextweb.com/google/2014/07/31/android-reached-record-85-smartphone-market-share-q2-2014-report/ Accessed January 2015

[22]Smartphone OS Market Share, Q3 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp Accessed January 2015

found at [23].

It can be seen that some challenges arise when developing mobile applications. However, at least when Android is chosen, this is no problem, due to the API Guides and the huge developer community. This makes it easy to find guidance and help relatively fast.

[23] Android API Guide. http://developer.android.com/guide/index.html Accessed December 2014

CHAPTER 3

Steps towards the App

This chapter defines the functionality of the SmartMaut (SMA) application. Although, the prototype will only implements the functionality needed for toll payment, this chapter also shows basic thoughts regarding additional features, to show our future perspective for the SMA application. Also, this chapter gives the original idea of how our application works. This means that some parts might have changed after implementing them and seeing their usefulness in tests. Such changes will be described and explained in the last subsection of this chapter. For this, three different perspectives will be used: First, scenarios that will explain the usage of the application from the users point of view will be given. This will be followed by an outline of the functionality of our application. Finally, it will be discussed how an implementation might work, based on the functionality that was outlined in the previous section.

Please note, that possible features that could be implemented for the SMA application are already listed and assessed in [8]. Therefore, they will not be mentioned in this work, again.

3.1 Important Terms

Before we dive into specific scenarios and their stepwise achievement, some terms, that were thought to be important with regard to our application shall be enumerated.

First of all, points that are very important for us, with regard to the application and the data it needs, will be given:

- We do not need to know who the driver or legitimate owner of the car is. We only need to know that someone drove on a toll charged road with a specific registration number and car.
- The term M-Point is used to refer to a geographic location, which marks a section for which toll charges have to be paid. These points need to be defined by the road authority and they need to be made known to the application, such that it can work properly. M-Point is a german abbreviation for ”toll-point”.

Next, possible roles of people who interact with our system are described:

- Responsible uploader: has to be present in the car and is responsible for the upload of the coordinates, such that the trip can be billed correctly. There is no need for this person to be the driver.
- Payment authority: registered for a vehicle/registration number as the person who will pay the full amount of the toll charges, as long as there are no co-payers. This person does not need to be in the car during a trip, but each car/registration number needs someone registered as payment authority. (Of course, there can only be co-payers if a payment authority is registered.)
- Co-payer: person that is driving together with someone else and wants to pay some share of the toll charges for the trip in question. A co-payer always needs to be linked to a payment authority.
- Legitimate owner: the person to which the vehicle belongs and that will be punished for any trespassing that might occur.
- Driving package: assignment of vehicles to co-drivers/co-payers (important for discounts/sharing); also the approval certificate is needed; for the prototype implementation, it simply means the selection of a vehicle and means of payment that should be used for a certain drive.

In the following, issues, that might not be that obvious, but shall be possible with our application nonetheless, will be collected.

1. It shall be possible, to declare oneself as payment authority, without joining the trip.
2. It shall be possible, to give toll vouchers as a present.
3. The one who is declared as payment authority does not need to be the legitimate owner of a car. (E.g. A lends her/his car to B, but does surely not want to pay B's toll charges)
4. It shall be possible to see and modify all data that is stored at the server for one's user account. This way, a user can make sure that all data is correct and in case of the loss of one's mobile phone, it will be easy to bind the new device to the existing data. (This is also true, if someone wants to register an additional device some time after the creation of the user account.)
5. The sharing of toll charges shall be possible, if there are several co-drivers in one car. At the time someone registers him-/herself as a co-payer, this person will be asked to give a maximum share as well as a maximum amount to be paid. The actual amount payable will be calculated at the server, aliquot.
6. Additional to the possibility of sharing toll charges, the responsible uploader and/or the payment authority of a given trip will benefit from each co-driver who was declared for a specific trip. Such a benefit could be that (environment) points are granted, that a possible

rank is effected positively or that some charged feature of the application may be used at a cheaper price or usage of these features might even be granted for free for a given period of time. Please note that these price reductions do not affect toll charges. They only concern charged features of the SMA application.

7. Someone who is registered as a co-payer for one or several cars will only be charged, if his/her application reports approximately the same coordinates and timestamps as the application of the responsible uploader of the vehicle in question.

8. It will be possible to store preferences for vehicles for which one is registered as co-driver or co-payer. If several cars one is registered for are driving in a convoy and the vehicle containing this person cannot exactly be determined (which means that no specific vehicle was selected for the current drive), a vehicle will be selected as beneficiary by the application, according to the given preferences.

9. Via the displayed application symbol, a user will be able to know, if the application is turned on and in which state (responsible uploader, payment authority, pedestrian etc. see [8]) it is running.

10. The user is responsible for switching on the smart device, needed for reporting, as well as the application. Furthermore, the user is responsible for providing valid data about vehicles and means of payment.

3.2 Usage Scenarios

Although the SMA application aims at providing a location based toll collection service, using GPS, existing services should also be supported. Therefore, two initialisation scenarios will be given - one for a location based system and one for a time based system. All other scenarios will only regard location based systems, unless the time based system is explicitly mentioned. These additional scenarios will make the mechanisms of a possible additional features for the application clearer.
Generally, the scenarios will give each step that is necessary for being able to effect toll payment via our application.

Scenario 1 - Initialisation (Location Based)

Initialisation

In short, the following steps are needed to initialise our application for a location based toll collection system:

1. vehicle/smart device needs to exist (otherwise the usage of our application is impossible or does not make sense)

2. install SmartMaut application (SMApp)

3. register the car in SMApp
    a) registration number
    b) vehicle type (this includes every information that is needed for determining the amount of the toll charges, like weight, CO2 emissions etc. for lorries, it needs to be specified whether a trailer is used or not)
4. register the smart device (on which the App is installed) at the SMA server
    a) e.g. register the user via a ”Bürgercard”-ID
    b) if someone wants to use several devices, each of them needs to be registered with the same ID
    c) one or more means of payment has to be defined

Having all these data declared, it is possible, to combine cars and means of payment arbitrarily via their IDs.

Before Starting A Trip

Before starting a trip, the following steps need to be completed (if this configuration was made before and is still valid, nothing needs to be done, as this configuration will be remembered by the App and the SMA server until a change occurs):

1. declare the current means of payment at the SMA server
2. declare the driving package in SMApp
    a) a registration number always needs a payment authority (who generally pays for 100% of the toll charges, unless a co-payer takes some part of it)
    b) NB: a co-payer may only be assigned to one registration number at one specific point in time
    c) OPTIONAL:
        - declare co-driver
        - declare co-payer
            - will be asked to give a maximum percentage and a maximum absolute amount for the contribution (where a selection that might have been made previously will be suggested)

OPTIONAL: it is possible to declare oneself as payment authority (before or after a planned trip), although there is no intention of joining the ride – this can be done via a handshake between the applications of the one who wants to be the payment authority and the one who will be the responsible uploader. This way, the server will know that someone will pay for the recorded trip and e.g. parents are enabled to pay for the trips taken by their children without the need of always being with them or swapping mobile phones. However, this case will be shown in detail in the corresponding following subsection.

For usability reasons, an application will of course remember these settings until they are changed, as already stated.

Scenario 2 - Time Based System

Initialisation

1. vehicle/smart device needs to be available
2. install SMApp
3. register car in SMApp
   a) registration number
   b) vehicle type (this includes every information that is needed for determining the amount of the toll charges, like weight, CO2 emissions, whether a trailer is used or not etc.) - for countries, where this information is not needed, the user will, of course, not be asked for it
4. register the smart device (on which the App is installed) at the SMA server
   a) e.g. register the user via a "Bürgercard"-ID
   b) if someone wants to use several devices, each of them needs to be registered with the same ID
   c) one or more means of payment has to be defined

Buying the Ticket

First of all, the means of payment, to be used for the purchase, has to be declared at the SMA server. This selection will be remembered until a change occurs.

Tickets for "Normal" Toll Charged Roads

1. select the needed time of validity for the ticket out of a given list

Tickets for Special Toll Charged Roads

1. declare the special toll charged road(s) that should be used
2. declare the date when the usage of the special toll charged road(s) is intended (where the current day will be suggested)

The purchase of either ticket is concluded by sending the request. The ticket is valid as soon as a confirmation of the server is received. In the case that neither a confirmation nor an error is sent by the server, the data stored at the server can be checked. If there is a confirmation, everything is fine and the trip may be started. If there is no confirmation, the data declared for the purchase and the means of payment need to be checked. If everything is correct(ed), the request needs to be reissued.

Scenario 3 - Driving in a Convoy

Obligatory Settings

1. declare the payment authority - This includes the selection of a means of payment. If there has been a previous selection, this choice will be suggested and does not have to be changed if it is still OK.

2. declare the responsible uploader

Note: payment authority and responsible uploader may be the same person, but this is not mandatory.

Optional Settings

1. declare a co-driver

2. declare a co-payer - If a co-payer is declared, this person will be asked to declare a maximum percentage and a maximum absolute amount for the contribution to the payment of toll charges. If a previous selection exists, it will be suggested.

Please note that a co-driver or co-payer can only be assigned to one of the cars in the convoy. If a co-driver is registered for several cars of the convoy and does not declare a specific one for this trip, the car occurring first in the co-driver's favourite list will be chosen as beneficiary for the given drive. On the other hand, co-payers need to explicitly declare or at least accept a beneficiary for each drive to which they want to contribute. This rule seeks to prevent co-payers from unintentional contributions.

Scenario 4 - Anonymous Payment Authority

For this scenario, it is assumed that a father lends his car to his child and is also willing to pay the toll charges for the trip, but does not want to actually join the trip.

Obligatory Settings

1. one of those who will join the whole trip has to be declared as the responsible uploader

2. the father declares himself as payment authority by binding his app (and his declared means of payment) to the car and the ID of the responsible uploader. This binding can be performed before or after the trip, but it has to be performed before the report is sent to the SMA server.

Optional Settings

1. declare a co-driver

2. declare a co-payer - If a co-payer is declared, this person will be asked to declare a maximum percentage and a maximum absolute amount for the contribution to the payment of toll charges. If a previous selection exists, it will be suggested.

Scenario 5 - Voucher

For this scenario, it is assumed that grandparents want to buy a toll-voucher as a present for their grandchild.

Like for each of the given scenarios, a smart device that has our application installed is needed. For this specific scenario, however, it needs to be distinguished between the location based service and the time based service, again.

Location Based Service

1. declare the user ID of the beneficiary

2. declare the value of the voucher

3. optionally declare the date (and maybe the time) when the voucher should be added to the beneficiary's balance, if this shall not happen immediately

Time Based Service

1. declare the user ID of the beneficiary

2. declare the wanted time of validity for the toll ticket through selecting it from a provided list

3. declare the date (and maybe the time) when the voucher should be received by the beneficiary

In either case, the beneficiary will be informed about the voucher by his or her application upon receipt of the voucher.

3.3 Discussed Corner Cases

This section enumerates corner cases, which were discussed when thinking of undesirable things that might happen when using our application. Where we could think of one, the solution will be given, too.

General Corner Cases

1. People with several (maybe borrowed) smart devices could create several identities, such that they will always have co-drivers.

    - **Solution 1**: (only possible when drivers are stopped) surveillance has to handle this case. Generally, users are required to register each of their devices with the same ID (e.g. ”Bürgercard” - if only a ”Bürgercard”-ID would be acceptable, this would be a very hard constraint for users, as not everyone owns a ”Bürgercard”)
    - **Solution 2**: surveillance is still required to handle this case, but there is no need to stop every single vehicle. Instead, the personnel that performs the surveillance will be aided by data telling them that a vehicle that claims to contain X persons is approaching. It can then be checked if a corresponding vehicle can be seen. If the personnel determines that some vehicle provides wrong data, the vehicle in question can either be stopped or the registration number will be memorised to charge fines afterwards. Alternatively, this solution could be automated by using cameras, which might require less personnel.

2. A co-driver without a (smart) mobile phone cannot contribute to any discounts.(e.g. phone is too old, no phone (or other smart device) at all, co-driver is a baby, pets)

3. Someone (= the payment authority) pays the toll charges for someone else, but is not in the car, when the trip is taken.

    - **Solution**: the payment authority can declare that toll payment shall be effected for a trip that was (or will be) reported by some specific ID for a given vehicle and registration number, including the date of the trip. This way, it is possible, that e.g. parents pay for their children’s trips without actually joining them.

4. There are only co-payers, but no one is registered as payment authority.

    - **Solution**: This is prevented by definition, according to which there cannot be any co-payer without a payment authority.

5. Phantom-car: Several people are driving together in one car. They all have their applications bound to their own cars and each application is active. In such a case, payment would be made for each of their cars, although only one is actually using the toll charged roads.

    - **Solution**: It should be possible to see that not all of the cars can be at the same coordinate at the same point of time, by using algorithms to sort out reports which are too close to each other, with regard to the sent locations and timestamps. However, if each application reported for another vehicle, one of the co-drivers has to declare its own car as the one that really drove, because if they were not checked during their trip, there would not be any other way to determine the car that really needs to pay. If no one declares the own car as the one that has to pay, each of them will be charged, as the system cannot determine the car that actually drove, without extra surveillance infrastructure, like videocameras being installed. It is considered that such extra equipment will not be wanted by the road authority, as the installation of the equipment and the processing of the data is expensive.

Corner Cases with Regard to the Scenarios Described Above

Scenario 3 - Driving in a convoy

1. If a co-driver for car A is actually sitting in car B, but A has been chosen as beneficiary by the server, enforcement might for example think, that 6 people were sitting in A (instead of 5). This might lead to a fine for the legitimate owner of A.

    - **Solution:** The server may only allow to choose a favourite car as beneficiary, if it does not already have 5 known occupants (if one or more of the occupants do not have the App or do not wish to contribute, it is OK, because enforcement will see that no more than 5 people are sitting in the car/registered as contributors for the car in question). If declaration is not made against the server, but against the application of the responsible uploader or payment authority, the corresponding application has to perform this check. The application of the potential co-driver has to be informed of the success or failure of the adding - in case of a failure, the co-driver can select the car in which he or she is actually driving as beneficiary.

2. If data, including declarations of co-payers and co-drivers, is collected in the application of the responsible uploader and if it is only uploaded e.g. in the evening, it is very hard for the user to correct illegal assignments, as the co-payers and co-drivers might not be able to correct their declarations in time.

Scenario 4 - Anonymous Payment Authority

1. Generally, a co-payer has to declare him-/herself against the ID of the payment authority. Friends of the child might not know the ID of the father.

    - **Solution:** Declarations of co-payers can also be made against the ID of the responsible uploader. As the whole report will be bound to the ID of the payment authority, co-payers will be encoded as well.

Corner Cases With Regard To City Toll

1. A pedestrian, who is registered as payment authority (and maybe also as responsible uploader) for some vehicle and whose application is running passes a toll point (which is called M-Point).

    - **Solution:** This problem can only be solved via a configuration made by the user. It would be necessary to tell the application that one currently wants to use the pedestrian mode or to switch the application off entirely. Also the reporting of additional

GPS-coordinates might prevent this problem, in case the user forgets to make these configurations.

2. A pedestrian who walks besides a car for which she/he is registered as a co-driver might contribute to discounts.

3. A pedestrian passes an M-Point at the same time as a vehicle for which the pedestrian is registered as a co-driver. If the driver is checked at some point farther down the road, this might lead to a fine, as a co-driver was reported, while in reality there is none.

   - **Solution:** Such situations can be prevented when additional GPS-coordinates are reported (not just the M-Points).

A general solution for the problems with pedestrians might be to filter them out via their mobility patterns, as these are different for driving vehicles and pedestrians.

3.4 Outline of the Basic Functionality of the Application

This section will explain how the SMA application is working in general. In Figure 3.1 it can be seen that our application consists of two parts.

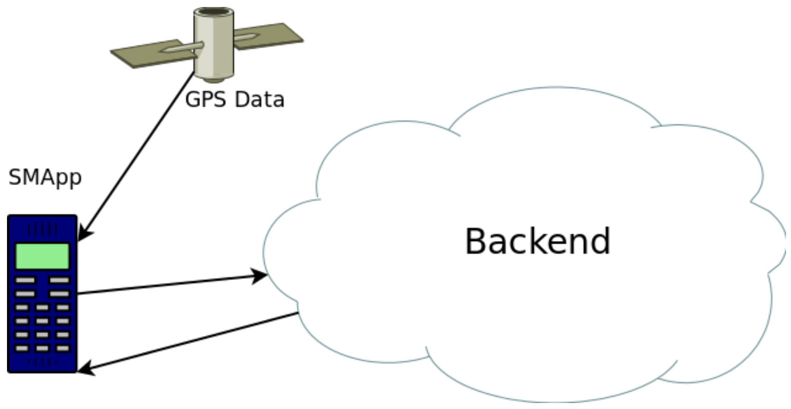


Figure 3.1: Basic structure of the overall SmartMaut application system

On the one hand, there is the user application, running on the user's smart device, and on the other hand, there is an external backend. The user application provides the interface to the end users. It requests data from the users and collects location data, using the sensors of the device on which it is installed. This location data, together with the user provided data about the used vehicle and the chosen means of payment are cumulated to so called reports. These reports are sent to the backend, such that the user can be charged correctly.

In order for being able to deliver correct reports, M-Points need to be declared. These points are set virtually and are used by the application, to determine whether toll charges have to be payed or not. Thus, passing an M-Point triggers sending a report to the server. No toll has to be payed in areas where no M-Points are set. All needed information about M-Points is obtained from the backend. This makes it easy to maintain and update the information which is provided by the road authority.

The backend consists of some middleware layer, a layer that provides the necessary business logic and a layer for data storage.

Data provided upon registration is sent to the backend, where it will be stored. This enables the usage of different devices. Each of them is able to access the data that has been stored for the user that is currently logged in. Of course, the application must keep a local copy of data like M-Points or registered vehicles. This saves time, network bandwidth, battery and maybe money (depending on the user's contract with its mobile service provider).

Billing data can be requested from the server, for the user to check.

It is important to note that most of the logic is located in the backend to keep the client application as lightweight as possible. On the other hand, the part that runs on the user's smart device must know and do enough to minimise the amount of data which needs to be transferred between the client application and the backend.

To demonstrate, how an advanced feature might work with SMApp, let us take a closer look at scenario 5 from above. In that scenario, a toll voucher is bought.

In order to sell toll vouchers, the application needs to be able to provide the user with a price list and, when a time based system is used, with correct times of validity for tickets. Of course, this information has to be obtained from the backend. As changes of this information will, however, be rather infrequent (maybe once a year), it can be obtained from the backend when the application is updated anyway. From this time on, this information can be stored locally.

As soon as the most recent pricing data is available, it can be shown to the user, who, in turn, can select the desired product. Additionally, one of the means of payment that have been registered for this user needs to be selected. This information is sent to the backend.

In response to the purchase request, the client application will receive an acknowledgement or an error message from the backend. This response is displayed to the user, in a suitable way. When the request was acknowledged, everything is fine and the case is done. In case of an error, however, SMApp suggests possible actions to the user. The application leads the user through all necessary steps, until the purchase succeeds, eventually.

3.5 What the Proposed Prototype Implementation will be able to do

The application resulting from this thesis will be a proof of concept, which shows that it is possible to create an understandable and usable application that runs on user-provided smart devices and implements the functionality of current toll charging systems. Additionally, this application aims at being usable across country borders. These goals are very ambitious. Therefore, it was decided, that this first version of SMApp shall focus on the location based system approach, rather than implementing a web-shop-like user experience for time based systems. This is, because the location based system approach is the more complicated one, which makes it more interesting with regard to usability. Also, it is already known that good web-shops can be realised for smart devices.

For the sake of simplicity, it was also decided, that features, like declaring co-drivers or co-payers, as well as declaring a responsible uploader or a payment authority will be left out. In this very first version, a user can only pay her/his own toll charges, will not have any benefits, within the system, from giving others a ride and has to report each drive him-/herself.

Furthermore, as opposed to the initialisation scenarios which have been given before, user registration will be required, as our discussions showed that registering devices is much more complicated and insecure.

To sum this up, only the basic use case of toll payment will be implemented. This includes a backend system, that is mocked by a database. The ”real” backend system, that will be optimised and much more intelligent than a simple database is being developed in other diploma theses.

The following list shows all features, the final application ended up with:

1. Register a new user with a unique ID. The usage of the ”Bürgercard”-ID is proposed for this task.
2. Log in an existing user.
3. Make the application remember a user.
4. Register one or more vehicles for the user that is logged in. It is possible to store incomplete data and complete the registration later.
5. Register one or more means of payment for the user that is logged in. It is possible to store incomplete data and complete the registration later.
6. View a list of all vehicles and means of payment, registered for the logged in user.
7. View the bills for a given period of time. This period depends on legal regulations, as personalised data may only be stored for a certain amount of time (see [1]).
8. Edit the stored user data.
9. Edit or complete the stored vehicles.
10. Edit or complete the stored means of payment.
11. Delete stored vehicles or means of payment.
12. Delete the user account. This will disable the account. Upon registration with the same unique ID as was used for the disabled account, the user-specific data will become available again.
13. Take a drive.

[1](Austrian) Federal Act Concerning the Protection of Personal Data – DSG 2000. http://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer= 10001597 Accessed January 2015

a) Select the vehicle and means of payment that will be used for the drive.
b) Collect location data and send reports, periodically. SMApp will do this automatically, as soon as this feature is started and will stop when the feature is stopped.

CHAPTER 4

Implementation

For the development of the SMApp prototype, a SCRUM-like [1] procedure was used. The application was split into small tasks. As meetings were held with the advisor every two weeks, this was the duration of a sprint. So, each sprint resulted in new functionality that had been implemented and could be presented in the meeting. During development, each feature was extensively tested. Issues that were revealed were fixed immediately.

In addition to a suitable development procedure, a good architecture is needed. Of course, the application needs to be easily maintainable, changeable, configurable and extendable. These requirements strongly suggest a modular system, as can for example be seen in [10].

The resulting application consists of three parts. One of them contains code for creating and interacting with the two databases that are used. Please note that the backend is simulated with one of these databases. Another one contains reusable code which is needed by several different classes. The last part consists of classes that are needed to provide the functionality of the application, which might be considered as business logic. They display user interfaces and react to user actions.

All of these parts can be seen in Figure 4.1. Together, ”SMApp Startup” and ”Main Menu” form the business logic part.

Android allows to define user interfaces and menus in XML files. This is very convenient for separating user interface design from the actual logic of the application. Additionally, it aids the modular system design. Layouts can be defined for different device sizes and screen orientations. Moreover, different resolutions of devices can be supported by providing different graphic files or pictures.

An XML file, called ”strings.xml” is used to define all strings that are used in the application. This greatly eases maintenance. Furthermore, these files enable to support internationalisation, easily. All that has to be done is to create a folder called ”values_XX”, where ”XX” stands for the language code. Within this newly created folder, a ”strings.xml”, containing strings in the specified language has to be placed. Android uses the locale defined for the device to select

[1]SCRUM. https://www.scrum.org/ Accessed January 2015

the appropriate language, automatically. If the language that corresponds to the locale is not supported, the default strings are used. Default values will also be used if the corresponding strings are not included in the ”strings.xml” file for some other language. These default values have to be defined in the ”strings.xml” contained by the folder ”values”. If no matching language and no default strings can be found, the application will not work. For detailed information about localizing Apps, please refer to [2].

SMApp provides German and English strings, where the English ones are default.

Permissions need to be granted and requested via the file ”AndroidManifest.xml”. Our application needs the permission to obtain fine grained location data for accurate positioning of the user device. This is crucial for billing users correctly. In the manifest file, it needs to be declared, that this permission is needed. When a user decides to install SMApp, a request for this permission is shown. If the user continues the installation, the permission is considered to be granted. SMApp does not do this, but an application might also share certain components with other applications. The permission to invoke SMApp or parts of it from other applications would be granted in the manifest file. For more information please refer to [3].

The modules, used for this application can be seen in Figure 4.1. ”Login” and ”Register user” are pooled as ”SMApp Startup”, because when the application is started, a user initially has the possibility to either login or register her-/himself. This holds at least as long as the App was not told to remember the user. In either way, the user will eventually end up in the main menu. There, the modules ”Register” (for registering one’s vehicles and means of payment) ”Drive” (for actually driving around and paying toll charges) ”Info” (to check all given information and recent bills) and ”Edit” (for changing, completing or deleting given data) are collected. ”Common” contains reusable functionality, while ”Database” provides interfaces for interacting with the cache database and with the database which is used to simulate the backend. These modules are depicted on their own, as they are used by every other module.

Each of the modules will be covered in detail in the remainder of this chapter. The corresponding class diagrams depict one module at a time and can be seen in the appendix. Also the classes which implement the modules are described in detail in the appendix. Concrete references to the appendix are given in the sections that describe the modules. As a thorough description of the Android system would go beyond the scope of this thesis, the interested reader is encouraged to refer to the resources provided by Android ([4]), in order to gain deeper insight in how single components of the Android system are working exactly.

To understand the implementation of SMApp, however, it suffices to know that an *Activity* is a component, which provides a graphical user interface, with which users may interact. This means that it is able to display data to the user and also to read in data provided by users (see section ”App Components”/”Activities” in [4]). Furthermore, *Activities* are needed for special navigation concepts, like tabs.

A *Fragment* must be embedded in an *Activity*. It may provide its own user interface, has its own lifecycle and provides its own events (see section ”App Components”/”Activities”/ ”Frag-

[2]Android Localisation Checklist. http://developer.android.com/distribute/tools/localization-checklist.html Accessed January 2015

[3]Android App Manifest. http://developer.android.com/guide/topics/manifest/manifest-intro.html Accessed January 2015

[4]Android API Guide. http://developer.android.com/guide/index.html Accessed December 2014

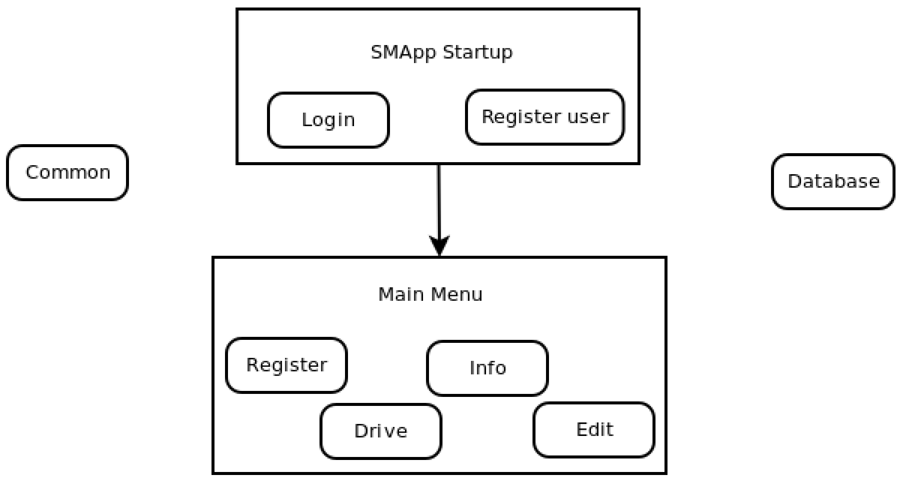


Figure 4.1: System architecture showing the used modules.

ments" in [4]). So, a *Fragment* can be seen as a modular building block. In SMApp they are used to display or request information that

- regards different topics, like vehicles or means of payment
- may change for different countries that may use the App

So, to sum this up, *Activities* are used for providing special navigation constructs and entry points into the App. *Fragments*, on the other hand, are used for providing dynamic information. If a *Fragment* will be displayed or not is decided at runtime, depending on device settings and user decisions.

Activities and *Fragments* are the major building blocks of SMApp. They provide most of the abilities that are needed for our purposes.

When user interaction is involved, an application might need to respond to clicks on provided buttons. This can be achieved by defining onClicked-listeners during layout definition (in the .xml file) or on *Activity* startup (in the program code). For onClicked-listeners it is important to know, that the view that is handed to the method is not the whole layout for this *Activity*, but only the object that has been clicked. If any other elements would be needed, it would be necessary to get hold of a reference to the whole layout. However, this is mainly important for *Fragments*, as *Activities* can access their layouts directly.

Android applications will generally run in a single main thread, also called the "UI-thread". However, long-running operations should run in the background, in order to prevent the user interface from "freezing". Also, no user wants to see the "Application Not Responding" message, which might also occur when the UI-thread is too busy. There are several possibilities to run expensive operations in the background. For this implementation, the *AsyncTask* is used. The possibilites it provides are completely sufficient for the purposes of SMApp. Thus, by choosing this concept, unnecessary overhead, which would result from every other possibility could be avoided. "This class allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers." (see [5]) The *AsyncTask* has to be started explicitly and runs only once, unless it is restarted. It has to be subclassed. The method *doInBackground(Params..)* has to be overridden. Code that is written in this method will asynchronously be performed in the background. *onPostExecute(Result)* is a method that may be overridden, optionally. It is called when *doInBackground(Params..)* returns and the parameter *onPostExecute(Result)* receives is the one which is returned by *doInBackground(Params..)*. *onPostExecute(Result)* is useful to display data on the user interface. Please note, that only *doInBackground(Params..)* will run in the background. *onPostExecute(Result)* is executed in the UI-thread, again (see Figure 4.2).

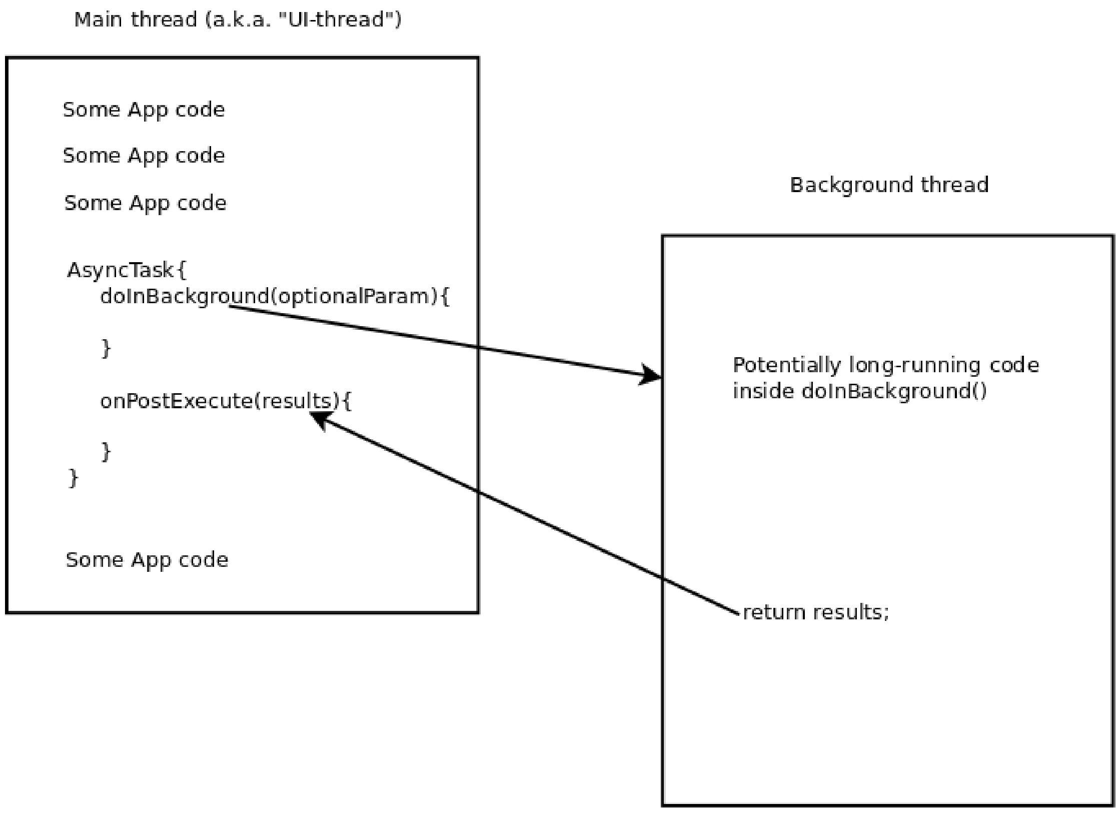


Figure 4.2: Running expensive operations in the background.

As suggested by Android, *AsyncTasks* are implemented as inner classes of the component that uses them. For SMApp, these classes are used for opening/closing connections to the databases and for any operations that require database operations to be performed.

[5]Android Reference AsyncTask. http://developer.android.com/reference/android/os/ AsyncTask.html Accessed January 2015

4.1 Login

The purpose of this module obviously is to provide the possibility for the user to log in. On the one hand, it provides security, as people who might get hold of the device, but do not know the needed login data will not be able to cause any harm. On the other hand, it enables the usage of a single device for multiple users. Each of the users can simply use the application with his/her own account. This is especially interesting for corporate customers.

The class diagram for this module can be seen in appendix A, in A.1.

The "Login" activity is the first thing that is shown when the App is started. However, *StartupActivity.java* is the first *Activity* that is called by Android. It has no layout and only checks if the user has previously told the application to remember its credentials. If this is the case, "Login" is skipped and the user is immediately forwarded to the main menu. When no such instructions were given, *LoginUserActivity.java* is called. From here on, the application woks as follows:

If a user has got login credentials, already, they need to be given. After clicking the "Login" button, the entered data is checked against the backend. Depending on the result, the user is either forwarded to the main menu or an error message appears. If no login credentials are available, yet, the user may press the button "Register". This causes a switch to the activity for registering a user, which will be described in the next section. So, the login credentials are only checked against the backend when the user enters them and not when the App remembered them. This is insufficient for security in a product, but for the purpose of this prototype, it suffices.

When the button "Login" is clicked, the corresponding method reads in the provided username and password, makes sure that none of them are empty, hashes the password and calls the *AsyncTask*, which checks the provided user credentials against the backend. Therefore, the provided credentials, a reference to the corresponding Data Access Object (DAO) and the context of the activity have to be provided. Via the context of the application, the *AsyncTask* may access the elements of the graphical user interface of the *Activity*.

If the provided credentials turn out to be wrong, an error message is displayed. If they are correct, the unique user ID is obtained from the backend and then kept in memory, in order to have it available for the use in future tasks. Finally, the user is forwarded to the main menu. All tasks, involving queries to the backend are executed in the *doInBackground* method of an *AsyncTask* to ensure that they are asynchronously performed in the background. Displaying an error message and forwarding the user to the main menu, are done in *onPostExecute*. This pattern is followed in all *AsyncTasks*, used for this application. The main menu consists of the modules "Register", "Drive", "Info" and "Edit" and will be described in an own section.

Figure 4.3 shows a preview for the layout of "Login". Such previews are provided for all layout files by Android Studio [6], the Integrated Development Environment (IDE) that was used for development.

[6]Android Studio. http://developer.android.com/sdk/index.html Accessed January 2015

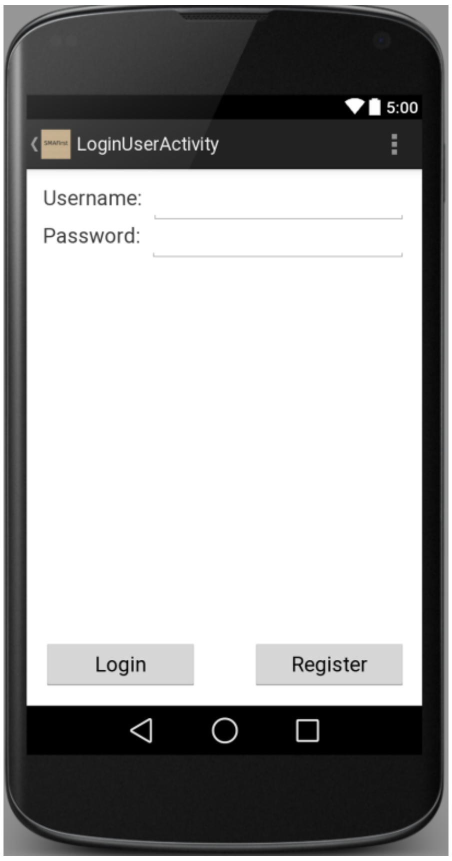


Figure 4.3: Preview of the layout for ”Login”, provided by Android Studio.

4.2 Register User

This module can only be reached by the user, when the corresponding button in the ”Login” activity is pressed. The preview of its layout can be seen in Figure 4.4 It demands all data that is needed for identifying and managing a user within the system. Besides username, password and email address, this includes a unique user ID. Currently, no specific form of such an ID is enforced. The application only ensures that the selected ID is indeed unique within the provided system. However, the layout already suggests the usage of the ”Bürgerkarte”-ID. This is, because such an ID is definitely unique and it already becomes increasingly present for security purposes.

Figure 4.4 shows that user registration also provides two buttons: ”Clear” and ”Add”. Their purpose is to remove all content from the text fields, such that they are empty again and to perform the actual user registration, respectively. The class diagram for this module can be seen in appendix A, in A.2.

Clicking the ”Add” button calls an *AsyncTasks* and provides it with the application context and a reference to the backend. The method *doInBackground* first gets all input that was provided by the user and checks if any of it is empty. If so, an error message is shown from *onPostExecute*.

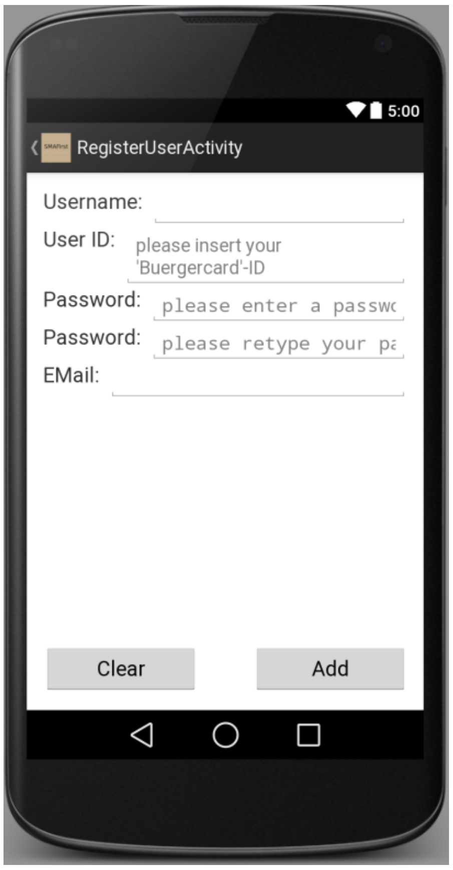


Figure 4.4: Preview of the layout for user registration, provided by Android Studio.

If all demanded information has been given, it is checked if the given unique user ID already exits and if the user is known to be an active one, first of all. Should this be the case, it will cause *onPostExecute* to display an error message, as multiple registrations for one user are not allowed. Otherwise, *doInBackground* continues with checking the given password's length. Less than eight symbols cause an error message in *onPostExecute*. Else, it is checked if the given password matches the retyped one. If this test succeeds, the password is hashed and it is attempted to register the user at the backend. However, if the unique user ID is already known and the user is currently inactive (which means that the account has been deleted some time ago), the account will instead be reactivated, as it already exists. (Reactivating accounts is another use case, where the ”Bürgerkarte”-ID proves to be useful.)

Upon the successful execution of these activities, *onPostExecute* causes a transition to the main menu. If any of the above operations failed or produced an undesired result, a corresponding error string reaches *onPostExecute*, such that an error message can be displayed.

4.3 Main Menu

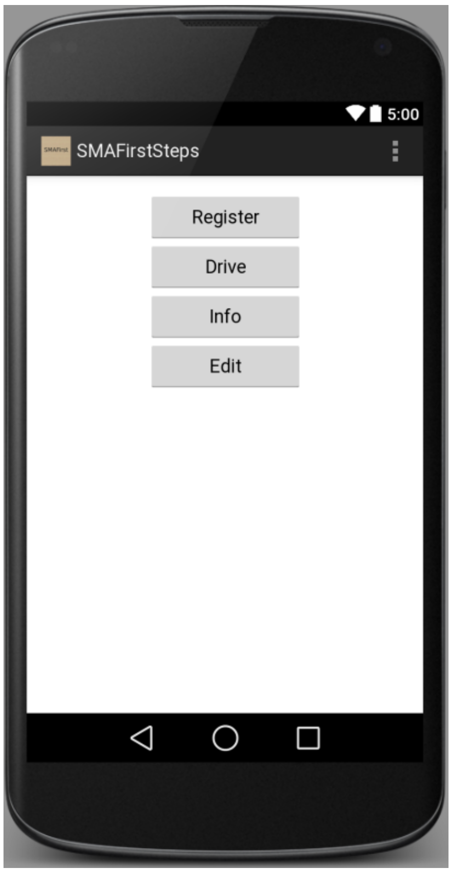


Figure 4.5: Preview of the layout for the main menu, provided by Android Studio.

The main menu provides a layout, which gives users the possibility to select either one of the modules ”Register”, ”Drive”, ”Info” or ”Edit”. Thus it only forwards the user to the chosen module if one of the buttons is clicked. Moreover, from here on, each *Activity* provides a *Settings* menu via the menu button. This might be a physical button, on some devices; for devices which do not offer a hardware button, the menu can be accessed via the symbol consisting of three dots, in the *ActionBar*. The *ActionBar* is the bar displayed on top of the screen in the generated previews; for more information please refer to [7]. - This symbol can be seen in the upper right corner of the displays of the generated previews, for example, in Figure 4.5. In case of the SMApp prototype, the *Settings* menu only contains the possibility to tell the application to remember one’s login credentials.

The preview for the layout of the main menu can be seen in Figure 4.5. Following, all reachable modules are described in their own subsection.

[7]Android API Guide On Action Bars. http://developer.android.com/guide/topics/ui/ actionbar.html Accessed January 2015

Register

For the registration of vehicles and means of payment, this module has to be used. Of course, it is possible to register several vehicles and means of payment. The data may be provided completely, but it is also possible to store incomplete information. However, at least a nickname has to be given. Furthermore, vehicles and means of payment do not have to be bound together.

Information stored through the ”Register” *Activity*, can be changed, completed or even deleted via ”Edit”, which will be handled in one of the following subsections.

The class diagram for this module can be seen in appendix A, in A.3. *RegisterActivity* is needed for being able to letting the user select vehicle registration or registration of means of payment via tabs. These tabs are shown in the *ActionBar* of the *Activity*. Upon selection of one of them, the corresponding *Fragments* will be loaded.

In Figure A.3, it can be seen that both possibilities of registration are implemented using two *Fragments*. The reason for this design decision is that, initially, the possibility to register multiple vehicles or means of payment, at once, was implemented. To achieve this, the first *Fragment* i.e. *RegisterVehicleFragment* or *RegisterMoPFragment*, which is also the one which is called, when the corresponding tab is selected, only provides a container for adding the details *Fragment* and all necessary buttons. The details *Fragment* shows the fields for entering the required information and is thus responsible for collecting and saving this data. Obviously, this design bears two great advantages. First, it is possible to add as much details fragments as one wants, while the buttons will only be provided once. This is, because buttons are only needed to occur once and multiple copies would only clutter the screen and confuse the user. The second advantage is that the details *Fragment* could be replaced seamlessly, to suffice the requirements of different countries.

Mentioning that the feature to register multiple vehicles or means of payment, at once was implemented initially, means that it is not available any more. The following will explain, how the feature had worked and why it was decided to take it out. In addition to the buttons ”Clear” and ”Add”, which are available now, a plus button would have been displayed. Through clicking ”+”, an additional details *Fragment* would have been displayed, beneath the last visible one. Additionally, the details *Fragments* would have been separated by horizontal lines, such that it would be easy to see where one block of data ends and the next one begins. If ”+” was pressed accidentally, pressing the device’s ”Back” button would have removed the details *Fragment* that was added last. (The design of the ”Back” button may vary between devices, but as far as we could see that, it is always some sort of arrow pointing left.) Pressing ”Add” would have caused each details *Fragment* to collect its information and store it in the backend.

However useful this feature might or might not be, in one of our meetings it was found that it could be a problem that it was only possible to remove the last details *Fragment*. This means that if one had opened three details *Fragments*, in this initial implementation, and decided that the second entry was irrelevant, there was no way to delete it, before saving the others, without deleting the third *Fragment* first. Furthermore, each potential solution would have made the user interface more complex and due to the variety of possibilities, users might have been confused.

Another feature that was cut out, but has not been mentioned so far is that the register *Activity* showed incomplete entries, when it was started. This aimed at reminding users about their incomplete datasets and providing them with the possibility to complete them right away. Nevertheless, such datasets may also be completed in ”Edit”. Thus, the same action would have been available, using two different *Activities*, which might have confused users.

Now that we know about the basic functioning of ”Register” and about all the features that are not supported anymore, let us have a look at what it does. The implementation of this feature is logically identical for vehicles and means of payment. The only real difference is that one implementation interacts with the user interface fields that are specific for vehicles and the other implementation interacts with the user interface fields which are specific for means of payment. At this point, it may be mentioned that with regards to means of payment, only credit cards are simulated in SMApp. This is, because one means of payment suffices to show the general idea of our prototype and the implementation of several options would not bring any further information. However, the user may choose between four different providers of credit cards.

The following description of the functionality of ”Register” does not distinguish between registering vehicles or means of payment, as the overall procedure is the same. Only the user interface differs, which causes different data to be fetched and processed.

Upon initialisation of the *Fragments*, a method which adds the corresponding details *Fragment* to the layout is called. For an implementation of the possibility to register multiple vehicles or means of payment at once, this method may be called several times, to add further details *Fragments* to the visible user interface.

Clicking the ”Clear” button causes the corresponding details *Fragment* to erase all values that might have already been entered into any of the fields of the corresponding user interface form.

When the ”Add” button is clicked, the corresponding *Fragment* informs the *Activity* by which it is contained about this event. This is necessary, since *Fragments* cannot directly communicate with each other. The *Activity* informs the correct details *Fragment* about the event, such that it calls an *AsyncTask* which retrieves all information provided by the user from the user interface, in *doInBackground*, first of all. It is checked if a nickname for the vehicle or means of payment to be registered was given. If so, the given nickname is checked for uniqueness among all vehicles or means of payment which are registered for the user, respectively. Upon passing this evaluation, it is determined if any of the fields has been left empty. With this information, the dataset to be stored can be marked as either complete or incomplete. Finally, the vehicle or means of payment is stored in the backend. A *String* value informing about the success or failure of the storage is handed to *onPostExecute*. Here, a short message informing the user about the successful operation is shown in case of a success. Furthermore, *clearFields* will be called, such that the user can register another vehicle or means of payment. Upon failure, an according error message is displayed.

When the user presses the ”Back” button, the *Fragment* is informed about this event. With only one details *Fragment* in use, it simply invalidates the corresponding reference. If multiple details *Fragments* would be visible, it would remove the details *Fragment* that was added last from the list of references, to avoid errors due to giving orders to components that are no longer there.

RegisterVehicleDetailsFragment and *RegisterMoPDetailsFragment* start with filling their *Spinner* elements in *onCreateView*. A *Spinner* is a drop down list that can be shown in the user interface. The values to be shown are, of course, provided by the application that uses this element, which is why it has to be initialized upon creation of the user interface. Such values aim at helping the user to provide correct data. Examples are vehicle weights or a number of axles.

When the user selects an item from the *Spinner*, a callback-method which stores the selection in memory for usage later on is called.

Drive

For better understandability of the following descriptions, the following is important to know: *Geofences* are used as a virtual representation of M-Points, in SMApp. A *Geofence* is a circular geographical area, which can be monitored. The monitoring of *Geofences* allows to recognise, when a user enters this specific area and has therefore to pay toll charges. In order to achieve this, a *Geofence* needs to be created for each M-Point. After their creation, they need to be ”spanned”, which means that they are actually placed, virtually. *Geofences* can only be monitored, when they are spanned.

When the application is stopped, *Geofences* have to be removed, in order to save resources and to prevent any errors when the application is restarted. This is, because it cannot be determined if *Geofences* have been spanned, after the application is restarted. However, this leads to the fact that a future implementation of SMApp will need an intelligent algorithm for the creation and positioning of *Geofences*. The application might become very slow, if all possible M-Points are created as *Geofences* and spanned at once. It is preferable to use an algorithm which allows to create and span *Geofences* for a specific part of a map when the borders of the area for which M-Points are already in place are about to be reached. This ensures fast operation, no matter if the application is used in a small test area or globally.

The class diagram of the ”Drive” module can be seen in appendix A, in A.4. In the following, the most important aspects of the functioning of this module are described. Details to the classes that are used to implement the functionality can be found in appendix B, in section B.1.

The ”Drive” module has to be used during a drive, in order to pay toll charges, hence, its name. Here, the user is asked to select the vehicle and means of payment to use for the drive. Only entries, for which complete information has been provided are shown and those marked as favourite are suggested. (If no complete entries can be found, the user is asked to provide a complete registration for at least one vehicle and means of payment.) When the user has finished the selection, pressing the ”OK” button tells the application to start collecting location data and to send reports to the backend, periodically. It is attempted to send a report, each time an M-Point is passed. For this to be possible, GPS needs to be turned on, on the device running SMApp, and all M-Points need to be in place. If SMApp detects that GPS is turned off, the user is asked to turn it on. As soon as the user chooses to do that, she/he is forwarded to the device settings.

Geofences are only created and spanned when the user presses ”OK”. This aims at saving resources, in case the user selected ”Drive” accidentally.

When ”OK” is pressed, the chosen vehicle and means of payment are read in and the corresponding details are obtained from the backend and stored in memory for usage for the reports. As soon as this is done and the *Geofences* are spanned, location updates are requested and received. Upon receiving a location update, it is checked if this location is an M-Point and the location is stored into the cache database. If an M-Point was passed, this is communicated to the user via a small pop-up window that disappears automatically after some time. Furthermore, it is attempted to send a report, containing all cached locations, together with the user ID and the details about the used vehicle and means of payment, to the backend. Should this attempt fail, sending the report is retried once. If this fails again, it will be tried to report all cached locations upon passing the next M-Point. When the report could be sent, all reported locations are removed from the cache. In case that the location is no M-Point, it is cached and nothing else happens.

As all location information which is stored in the cache database is sent in the report, also locations which are not part of toll charged roads will be transmitted. This behaviour is needed, such that planned additional features, like a logbook can be implemented.

Info

Like ”Register”, the ”Info” module uses tabs to navigate to the data that should be presented to the user. It provides a ”Data” section, where all registered vehicles and means of payment can be viewed, and a ”Billing” section. The latter only contains example data, at the moment, but in a future implementation of SMApp, it will show the most recent bills for a certain period of time. A class diagram of this module can be seen in appendix A, in A.5.

The ”Data” section is the one that is displayed to the user when ”Info” is opened. To display all details about registered vehicles and means of payment on the user interface, a *LayoutInflater* is used. A *LayoutInflater* ”instantiates a layout XML file into its corresponding View objects” [8]. This enables to access each element, which is defined in the user interface of the *Fragment* programmatically.

A layout has to be inflated for each vehicle or means of payment. The information that is shown this way is simple text. This means that the user can neither click on it to start any actions, nor can the data be edited directly. It simply provides an overview over all data that is known to the backend about the logged in user.

For debugging purposes, a further button may be activated for the main menu. It is called ”Debug” and displays reports that have been stored for the user that is currently logged in. As they are only used for debugging purposes, the classes, used for the implementation of the ”Debug” feature are not described in this document, but the implementation works exactly as for the ”Data” section of ”Info”.

When the ”Billing” tab is selected in ”Info”, at the moment, some static example text is displayed. In a future implementation of SMApp, however, a mechanism, similar to the one described for displaying details about vehicles and means of payment, can be used for showing recent bills.

Edit

This module allows to change, complete or even delete registered vehicles or means of payment. Also the data that has been given upon user registration may be changed and pressing ”Delete”

[8]Android Reference LayoutInflater. http://developer.android.com/reference/android/view/ LayoutInflater.html Accessed January 2015

inactivates the user account. A class diagram of this module can be seen in appendix A, in A.6.

The offered functionality is made available, using three tabs. One for user data, one for vehicles and one for means of payment. When the user clicks ”Edit” in the main menu, the tab ”User” is selected by default. As user data will only rarely be checked, any change requires the user password for security reasons. The user *Fragment* handles everything which regards data about the actual user. It provides the buttons ”Edit” and ”Delete”. Via an *AsyncTask*, the user data for the logged in user is fetched from the backend. *onPostExecute* receives this data as a parameter, uses it to fill the fields in the layout and remembers the hash of the current password.

When the ”Edit” button is pressed, the *AsyncTask ”UserEditor”* is called. It receives a reference to the *Fragment's* view and to the backend DAO, as well as the hash of the old password. The password is called old, because the user might have changed it. *doInBackground* first obtains all data from the layout. None of the fields must be empty. If this is fulfilled, it is checked if the old password has been entered correctly. If this is the case and the password was not changed, the user data is updated in the backend. If the password changed, it is checked if it is long enough and user data is only updated when this test is passed. *onPostExecute* receives a String containing the result of the update information. If all tests passed and the data could successfully be updated in the backend, a success message is shown to the user. If something went wrong, however, the String parameter contains sufficient information for displaying detailed error messages.

Should the user press ”Delete”, the *AsyncTask ”UserDeleter”* is called. Its *init* method receives the same parameters as the one of *UserEditor*. *doInBackground* checks if the correct password has been given. If this is the case, and the App was told to remember the user, previously, this data is deleted. Finally, the backend is told to inactivate the user account. *onPostExecute* receives a result String. If any error occurred, a corresponding error message is displayed. In case of a success, *StartupActivity*, which was described at the beginning of this chapter, is called. The user experience might be improved if pressing the ”Delete” button would first cause a dialogue to be displayed, which tells the user that proceeding with this action will inactivate the user account and that the account can be reactivated by registering with the same unique ID again.

Editing or deleting vehicles or means of payment basically works the same way as it was described for user data. Three *AsyncTasks* are used. One for fetching the details for the chosen vehicle or means of payment from the backend and displaying them to the user, one for editing these details and one for deleting the whole dataset. However, no password is required to perform these operations. Furthermore, if the user wants to change the favourite vehicle or means of payment, this has to be achieved by selecting some other vehicle or means of payment as favourite. The previously selected entry is deselected automatically. The application does not permit to deselect any entry manually, to avoid situations in which no favourites exist. This is because it was defined that a favourite should always exist, such that the application can save the user's time by suggesting the most likely selection. The favourite status is also checked when a vehicle or means of payment should be deleted. If there are other vehicles or means of payment available, and the current favourite would be deleted, the user is asked to select another favourite, first.

However, these are rather minor differences. What is really important is that Editing vehicles and means of payment needs two *Fragments* to provide the needed functionality, as can be seen in appendix A, in A.6. This is, because only one user will be logged in at a time, on one device, but this user might have registered several vehicles and means of payment. Therefore, a *Spinner* is provided, such that the user may select the vehicle or means of payment to be edited. The last selection is cached until the application is restarted. Furthermore, a details *Fragment* updates the corresponding *Spinner*, when an entry was deleted. So a *Fragment* is used to fill the *Spinner* with the nicknames of the vehicles or means of payment which are stored for the logged in user, while a details *Fragment* is used to display the stored details of the selected item.

When the last available vehicle or means of payment is deleted, the corresponding details *Fragment* will also be removed, as there are no details to be displayed anymore.

4.4 Common

This module contains reusable code that is needed by most of the classes, described above. A class diagram, showing the contained classes can be seen in appendix A, in A.7. The single classes are not connected, as they do not use each other.

As this module does not contain any specific functionality, but aids the implementation of the other modules, a detailed description of its classes can be found in appendix B, in section B.2.

4.5 Database

This module contains classes which are needed to interact with the cache database and with the backend, which is currently mocked using a database. The database in use is *SQLite* [9], as it is available on all Android devices and as Android, therefore, provides very convenient libraries for accessing it. Class diagrams for the cache database and the backend database can be seen in appendix A, in A.8 and appendix A, in A.9, respectively.

A so called ”contract” has to be used for each database, in order to define its tables. *Sma - CacheDbContract* and *SmaDbContract* both contain a String constant, which specifies the name of the database file and an Integer constant, declaring the current version number of the database. For each table, the contract needs to contain a static abstract class, implementing the interface *BaseColumns*. These abstract classes contain String constants for the table name and the name of each of the table's columns. The column ”_ID”, which is inherited from *BaseColumns*, does not have to be declared. This column is needed, such that SQLite can assign a unique ID to each entry. The constant names are very useful for creating tables and for inserting, updating or deleting certain entries.

When a connection to a database is requested - which, in case of SMApp, happens in the corresponding DAO classes - the database will be created if it does not exist, already, or updated if the version number changed since the last usage. For being able to do that, a class which extends *SQLiteOpenHelper* is needed. In SMApp, these classes are *CacheOpenHelper* for the cache database and *DbAccessHelper* for the backend database. In these classes, Strings for

[9]SQLite. http://www.sqlite.org/ Accessed January 2015

creating the necessary tables are created, first. Such a String for creating the table that will contain user data for the backend database might look as follows:

```
private static final String CREATE_TABLE_USERS =
   "CREATE TABLE "
   + SmaDbContract.UserTable.TABLE_NAME +" ( "
   + SmaDbContract.UserTable._ID +" INTEGER PRIMARY KEY
     AUTOINCREMENT, "
   + SmaDbContract.UserTable.COLUMN_UNIQUE_ID
   +" TEXT NOT NULL, "
   + SmaDbContract.UserTable.COLUMN_USER_NAME
   + " TEXT NOT NULL, "
   + SmaDbContract.UserTable.COLUMN_PASSWORD
   + " TEXT NOT NULL, "
   + SmaDbContract.UserTable.COLUMN_EMAIL
   + " TEXT NOT NULL, "
   + SmaDbContract.UserTable.COLUMN_IS_ACTIVE
   + " INTEGER NOT NULL ) ";
```

In this example, it can be seen that the constants that have been defined in *SmaDbContract* are used. In *onCreate*, the Strings will be executed as SQL statements, to really create the tables for the database. *onUpgrade* will be called when the version number of the database changed. It will cause the existing tables to be dropped and recreated, according to the specifications of the new schema. So far, the necessary steps were the same for both databases.

The cache database not only caches location updates until they are reported, it also contains vehicle classes, possible vehicle weights, suggestions for numbers of axles, supported credit card providers and months and years for possible expiration dates of credit cards. This information is used to fill *Spinner* elements, in order to help users to declare the details about their vehicles and means of payment correctly. *Spinner* elements need to be filled relatively often and it is assumed that the needed information would be provided in the backend, when the application was really used. However, this information will not change regularly. Therefore, it can be obtained from the backend, once and then stored in the cache database. Currently, possible values are defined directly in *CacheInitiator*, but this class may be used to obtain them from the backend in future versions of SMApp. When *open* is called in *CacheDAO*, it calls each method, provided in *CacheInitiator*, in order to fill the tables of the cache database with values that can be used for *Spinners*.

The methods *open* and *close* exist for both databases. They are needed, such that classes which need to access the database can open the required database for read and write access and that they can close the database when they are done.

Let us look at *CacheDAO* first. It provides methods for inserting the data needed to fill *Spinner* elements, one method for storing M-Points and one for storing locations. M-Points will also be provided by the backend, in a future version of SMApp.

Location updates are stored in this database, until a report has successfully been sent to the backend and the deletion of location data is requested. For inserting data into any database table, a special object is needed - *ContentValues*. Each value has to be assigned to a specific column, via this object. It can then be given to the insert statement for actually writing the values into the database.

As no fine grained access is needed for data stored in the cache database, each "get" method, which exists for each of the tables, will return all data these tables contain. For example, *getAll -VehicleClasses* returns all vehicle classes, one could possibly select in SMApp, while *getAll-CreditCardProviders* returns all credit card providers that are supported by SMApp and therefore stored in the corresponding table etc.

updateLocation enables to update location data. It is needed if a location, which is actually an M-Point is already cached. This method enables to declare the cached location as M-Point, afterwards, as described in appendix B, in section B.1. For deleting the location data that has already been sent as a report, *deleteReportedLocations* is used. It takes the ID of the last location that was reported as parameter and deletes every location with a lower or equal ID value, as these have already been reported.

Finally, *locationExists* can be used to find out if a given location already exists in the cache database.

DAO, which provides methods, necessary for interacting with the backend database, contains much more methods. But, of course, it works the same way as *CacheDAO*. First, it enables the application to store users, vehicles, credit cards and reports. For this prototype implementation of SMApp, storing a report is equal to sending a report to the backend. However, the methods for retrieving data allow specific selections. For example, it is possible, to either retrieve the unique user ID by providing the correct username and password or to get all user data by providing the correct unique user ID. Vehicles may be obtained via their database ID, via the unique ID of the user for which they are registered or via their nickname, combined with their user's unique ID. Furthermore, it is possible to get only vehicles that have been registered completely, i.e. no data is missing, or to retrieve only the favourite vehicle. The same methods are available for means of payment - which in the case of this prototype are only credit cards. Of course, there is an update method for the data of each table.

As already mentioned in the subsection "Edit" in "MainMenu", a user account can be inactivated, using *inactivateUser* or reactivated using *reactivateUser*. Methods for deleting vehicles and means of payment are also provided.

The additional methods are more interesting. Using *userExists*, it can be checked, if a user with the given unique ID is already registered. If this is the case and if *isUserActive* returns false for the same unique user ID, the account can be reactivated. Otherwise an error message will be displayed by the application. The reactivation of user accounts is explained in the section "Register User", above. If the unique user ID is not registered, yet, the user will be registered by storing him/her into the database. *loginCredentialsCorrect* checks if the entered login credentials match any of the registered users.

vehicleNameExists and *mopNameExists* are used in "Register", to check if the user might have already used the given nickname for another vehicle or means of payment. The methods that allow to check whether any vehicles or means of payment remain or are selected as favourite are useful to change screens correctly and to determine if the "Edit" or "Delete" action, requested by the user is permitted at that particular time. The methods that check whether a favourite already exists are also important when the user registers something as favourite. When this is done, SMApp automatically deselects any other vehicle or means of payment that was marked as favourite, previously, by using *deselectFavouriteVehicle* or *deselectFavouriteMoP*, respectively.

SMApp will automatically mark the first vehicle and means of payment that is entered as favourite, if this is not done by the user. For this, *doesFavouriteVehicleExistAtRegistration* and *doesFavouriteMoPExistAtRegistration* are helpful.

getVehicleString, getMoPString and *getLocationString* are used for processing report data. They convert a given *Cursor* into a String.

CHAPTER 5

Usability Tests

To find out whether SMApp can be used by the targeted user group it was decided to execute usability tests with ten test users. Our targeted user group are people who use toll charged roads and who are not older than 60 years. This maximum age was chosen, as no one who is older than 60 and owns or uses a smart device was available to perform the usability tests.

All tests were executed using a Samsung Galaxy Tab 3 tablet, where our application was already installed. On average, one test took about an hour.

5.1 Approach

According to [3], research on usability testing concludes that ”four or five participants will reveal approximately 80% of the usability problems in any product. Similarly, using ten participants will usually result in 90% of all usability problems detected.” They therefore recruited ten participants for their own tests. This is also why it was settled for ten test users for testing our mobile toll charging application.

To perform the tests in a structured way, a questionnaire was used. For the preparation of this questionnaire, especially [3] and [6] served as a source of inspiration. Of course, they do not mention actual questions they asked, but they write about the type of questions they have used in their own tests. The resulting questionnaire can be seen in appendix C.

Regarding the test environment, the papers that were used as a guide for this thesis disagreed with each other. [7], for example, found that participants that were allowed to execute their tests in a peaceful environment, which would correspond to the traditional method of usability testing, found much more usability problems than people who had to move around while testing or who were confronted with different sorts of distractions during the tests. The approach where people have to move around or are confronted with different sorts of distractions during the tests, tries to bring the testing environment as close to the real environment, in which the application will be used, as possible. However, the additional problems that were found in the peaceful environment have mainly been cosmetic ones.

[3], on the other hand, concluded that field tests revealed much more usability problems than tests that were executed in the laboratory. Additionally, the problems their participants discovered in the field tended to be critical ones.

[6], however, found that there is no significant difference between field testing and laboratory tests. In their study the average amount and severity of the problems that were found in either testing environment was equal. They therefore conclude that there is no additional gain of found usability problems by using one methodology over the other. The only thing that was mentioned was that field testing takes much more time than laboratory testing (about twice as much), because the preparations are more complex and you have to consider travel times and unplanned interruptions.

So, there is no real consensus on the most effective testing environment in the literature. However, our possibilities were limited, anyway, as our tests had to be executed in an environment, where the participants were unwatched by others, but where they could still hear the other people moving around, talking and joking with each other. This is due to the architecture of the house where the usability tests took place. To show the whole functionality of the ”Drive” activity, participants had to go on a short walk.

Upon execution of the tests, each participant had to complete a predefined set of tasks. These tasks were chosen in a way which would ensure that each participant would see and use each feature of the application. Before testing SMApp, the users had to interact with a preinstalled memo application to give them the opportunity to familiarise with the device and the interaction patterns that are used in Android. This idea was taken from [6], where participants also had to solve ”warm-up” tasks before performing the real test. During this initial ”get-to-know” between the user and the device, it was made sure that the participants saw how the keyboard could be hidden or shown when that was desired, how the previous activity could be reached and how the application menu could be obtained.

After the participants felt comfortable enough, our ”SmartMaut” application was presented. In this application, the following tasks had to be completed:

1. explain what can be done on the initial screen and what still needs to be done
2. register oneself as a new user (and remember the chosen password!)
3. register vehicles and means of payment - at least one complete and one incomplete entry
4. explore ”Info”
5. check out ”Edit”
    a) tell what possibilities are provided
    b) change user data
    c) find out how to switch between the different vehicles and means of payment which have been stored previously
    d) change/complete data for vehicles and means of payment

6. start ”Drive”, complete all necessary steps and then take a little walk to see what will happen

In addition to performing these tasks, the test users were asked to think aloud while performing them. This was helpful to see where problems or insecurities arise. Moreover, this method got people to try to explain what causes their problems, which is very important information for optimising the App. This ”think aloud” approach was also used in [3], [6] and [7].

To evaluate the collected feedback, all detected issues and comments were collectively written down. Next, it was counted, how often they were mentioned. Of course, also all other information that was obtained from the questionnaires has been structured. Finally, the detected issues were categorised, according to [3]. For better understandability, their definition of the categories is cited below: ”

Critical problems

- Prevented test subject from completing tasks and/or
- Recurred across all test subjects

Serious problems

- Increased test subjects’ time to complete task of test subjects severely and/or
- Recurred frequently across test subjects and
- Test subjects still managed to complete task eventually

Cosmetic problems

- Increased test subjects’ time to complete task slightly and/or
- Recurred infrequently across test subjects and
- Test subjects could complete task easily

”

To enhance testability of the current prototype, however, a second application has been written. It simply provides a single button. When this button is pressed, the current location is determined and stored into the cache database, using the method *insertMPoint*. So, currently, M-Points are defined manually in some area where test users were able to walk around with the App. In order for this second application to be able to write into the cache database, an ID needs to be defined in its ”AndroidManifest.xml”. SMApp needs to grant access to its cache database for this specific application ID via its own manifest file.

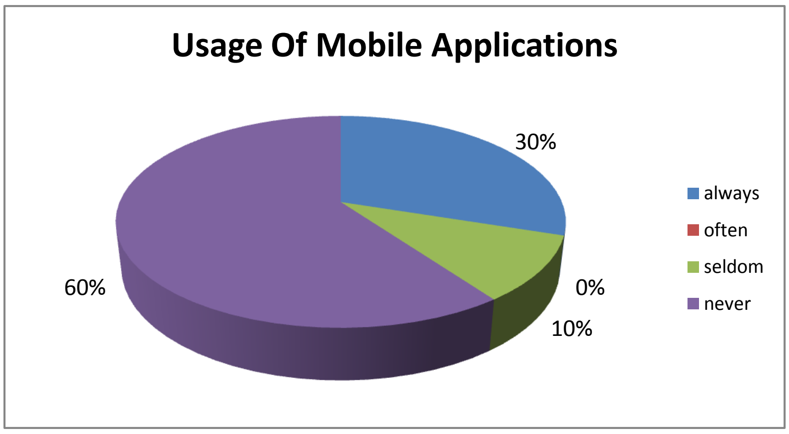


Figure 5.1: Frequency of usage of mobile applications by participants.

5.2 Results

This section summarises the feedback that was obtained from our test users. Before we talk about the found issues, here is a quick overview about the demographics of the participants:

An equal number of men and women has been chosen. 60% of them are aged 45-55 and 20% are aged 20-35 and 35-45, respectively. 90% of the participants uses smart devices all the time, as they possess a smart phone. However, only 30% of the participants uses mobile applications regularly. 60% do not use mobile applications at all and one test user uses them from time to time (see Figure 5.1).

The technical affinity of the participants ranges from high to low, where every step is included. 20% were classified to have a low technical affinity, 40% were classified as ”medium low”, 10% are ”medium high” and 30% have high technical affinity (see Figure 5.2). Furthermore, 60% of the participants claimed to be familiar with the Android operating system, as it is installed on their phones.

When it comes to the usage of toll charged roads, 20% of the test users claim to use them every day, 50% use them every week, another 20% uses them every month and 10% use them only some times a year (see Figure 5.3).

To get on to the specific feedback, a total number of 65 issues and comments was collected. 38 of them have only been mentioned by one person. 25 of the collected issues were categorised as problems. Thus, the remaining 40 comments have been identified as wishes or ideas on how the application could be improved, in order to become a useful product. This is because they did

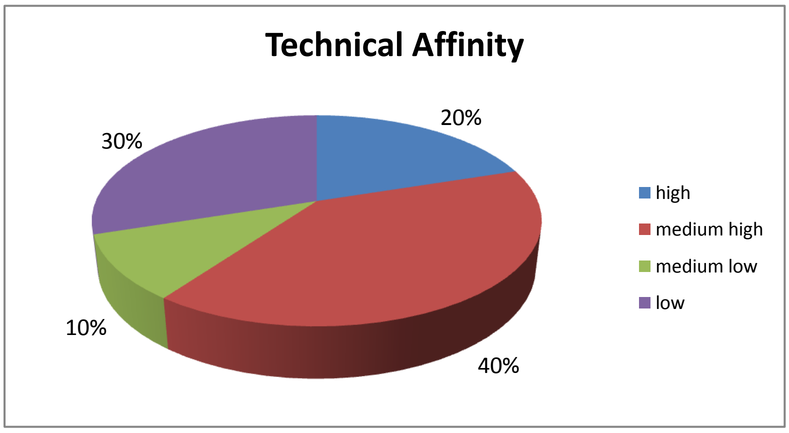


Figure 5.2: Technical affinity of participants.

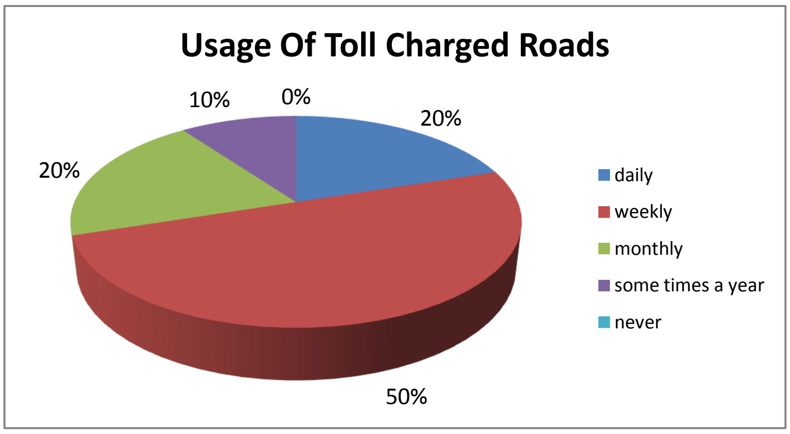


Figure 5.3: Frequency of usage of toll charged roads by participants.

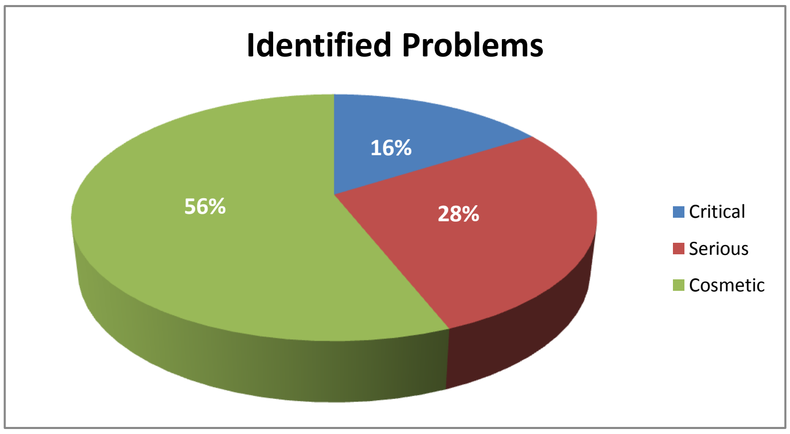


Figure 5.4: Share of problem categories.

not comply to any of the definitions for problems, given in the previous section.
Out of these 25 problems, 4 have been classified as critical ones, 7 as serious ones and 14 are categorised as cosmetic problems. These absolute numbers correspond to 16% critical problems, 28% serious problems and 56% cosmetic problems (see Figure 5.4).

Table 5.1 shows all the detected problems together with their classification. The problems are ordered according to the App features they belong to. The fact, that ”vehicle name” is associated with the brand of the vehicle might not seem to be a problem at all. However, all test users did that and additionally, ”vehicle name” is meant to be any nickname a user has for his/her vehicle, such that it can be found in lists later on. This means that the UI field does not communicate its purpose properly. In fact, this issue is also related to the problem that people did not know what ”payment name” is for.

The remainder of this section will be structured in such a way, that there will be a subsection for each main menu entry and also for the whole application. These subsections will describe the problems and comments, related to their specific topic. It will be dealt with the detected problems, before the suggestions for improvement are handled.

Register

The most important issue that occurred with regard to this feature is the fact, that 60% of the participants found the name ”Register” very confusing. This is, because they had just registered themselves in another activity and did not understand why this option was offered again. Out of curiosity, they would have clicked the button anyway, but to avoid confusion, they suggested to rename the button.

Not surprisingly, when asked about their expectations, all participants expected to get the possibility to register a user, when clicking the ”Register” button. Only one test user expected that also a means of payment could be registered, as it should be possible to pay toll charges with this application. A single other test user expected the possibility to register a vehicle, in addition to registering a user.

Thus, the issue regarding the button name was categorised to be a serious problem. It did not prevent people from getting their tasks done, but it confused most of them on first sight and it recurred throughout all tests.

Not knowing where to find the information required for vehicle registration was also a problem that was faced by 60% of our participants, which is why it is rated to be serious. First, some of the test users thought that they would not be able to complete the information. Only after some time, they noticed that the required information could be found in their vehicle documents.

This problem would be best solved by providing some kind of help text, either in the form of an additional hint, as static text on the top of the form or within a help menu which can be accessed via an icon that is always visible.

Having trouble with ”payment name” and associating ”vehicle name” with the brand of the vehicle, both result in the need to explain the purpose of the fields. These problems have been categorised differently, because none of them prevented any test user from completing a task, but everyone thought that ”vehicle name” had to be associated with the brand and maybe also with the model of the vehicle, which is a wrong assumption. Additionally, this assumption could cause problems, if someone wanted to register a whole car pool, which could easily contain several vehicles from the same brand and model (e.g. driving schools usually use a pool of identical cars).

20% of our participants thought, that a means of payment has to be registered for or bound to each vehicle. This problem is said to be serious, although it affected so few users, because those who thought so, were a bit confused during registering their vehicles and means of payment. While one of them simply accepted that the assumption might be wrong and continued exploring the application until it became clear that a direct binding was not necessary, the other one kept searching through the register activity for quite some time. As there was nothing to be found, the search continued in the edit activity, to see if it would allow to establish the binding.

Thus, this issue cost the participants quite some time and it also caused negative feelings regarding SMApp. This clearly needs to be avoided.

A whole 50% of test users did not notice that the vehicle’s weight has to be given in tons. This lead to some confusion when they first saw the selectable values. To avoid that people overlook this important information, the unit should be written as a whole word, or it should be attached to each suggested value.

Only one participant wanted to be warned, if ”Register” would be left without saving data.

Still, it was categorised as a cosmetic problem instead of a suggestion for improvement, because this one user really did not like that all data was gone, just because the activity was explored.

When registering a credit card as means of payment, 20% of the test users had troubles to select the expiration date. It was not clear for them, that month and year had to be selected (in this order), instead of a specific date. To overcome this problem, they wanted it to be declared that month and year have to be selected in this specific order.

The wish that the information about the success of the saving operation should be displayed longer was also stated by 20% of the participants. Displaying this information in a bigger box or in the centre of the window was also suggested. The ones who noted this had not been sure if the data they had provided had really been saved or not.

Now, that all problems regarding the register activity have been explained, the suggestions for improving SMApp will be described. As most of these suggestions were only made by one participant, the number of occurrences will only be mentioned when this amount is exceeded.

- A short notice about having registered oneself successfully would have been appreciated, before entering the main menu of the App.
- To have more control about what happens in the application and also to gain more assistance, it was suggested that when a registration is saved, the given data should only disappear when ”register additional” is selected. This would be a new button which would appear after a successful save operation. If no additional data is wished to be registered, the information given before shall remain visible, until the activity is closed.
- 20% of the participants wanted to be warned, if the data they wanted to save was incomplete.
- Even the wish for being forced to complete the data at once was brought forward. This results from the fear to forget to enter the missing data, while all needed resources, like vehicle documents, are available.
- It should be possible to save vehicle and means of payment in a single step.
- When incomplete information would be saved, the application should ask if this is really wanted and it should also suggest further steps that could be taken. Such a step could, for example, be to enter ”Edit” to complete the information.
- A dialogue should pop up and inform about the success of the save operation. Additionally, this dialogue should ask the user what should be done next and it should also redirect the user to the chosen activity.
- The user should be told that the given information is legally binding, to prevent people from providing wrong data.

- Finally, it was also suggested to rename the button ”Clear” to ”Delete”, to stress that the fields will be emptied and the data that was entered into these fields, before, cannot be restored any more.

A remark which did not turn up during the usability tests, but was discussed in a meeting with my advisor, is that it might be better to preselect the checkbox for setting a vehicle or means of payment as one’s favourite for the first vehicle and means of payment that is registered. This would inform the user about the fact, that this particular entry will be considered to be one’s favourite by default, until another entry is chosen as favourite by the user.

Drive

When asked about their expectations, regarding this activity, users correctly assumed, that it had to be used while really driving somewhere. However, 60% thought, that the desired route had to be declared, here. Some also supposed that it could be used for navigation, to declare the desired vignette to use, to obtain a forecast for the toll charges that would arise or to be able to choose an alternative route which would be suggested because it is cheaper and to see the entered route, whether toll charges have to be payed when using it and what price that would be.

The detected issues for this activity are much less than for the register activity. Additionally, all of these issues have been classified as cosmetic problems:

One participant did not understand the information that is displayed during the actual drive and would have appreciated an explanation. This explanation would prevent worries about something going wrong.

50% of the participants would have preferred seeing a map when driving around. It would allow to check one’s surroundings, when driving in an unfamiliar area. This was considered to be a problem, as half of the test users brought this wish forward. It was also suggested to provide the possibility for the user to choose whether a map shall be displayed or not.

During the actual drive, the application displays the obtained location data to the user. Also found M-Points are displayed. One participant noted this information is too much and that the used font size is too small. It caused the user to concentrate on the application for a while. This would clearly be bad when the application was really used while driving.

When asked about the information displayed during the actual drive, 30% of the participants said that they would find it distracting while driving and 20% said that they wanted only the passed M-Points to be displayed. 40% would prefer sound information about passing an M-Point over the displayed text. They said that this would be more convenient, while driving, as it does not require to keep an eye on the application. Even a suggestion for the concrete sound was made - it should say that an M-Point was just passed. One participant suggested to use sound information as well as text, but with the option to turn these signals on or off oneself. All other participants did absolutely not want to have sound information. Another 40% would have preferred to have no information about passing M-Points at all. In this case, a menu option is required, for the configuration of individual preferences.

Further suggestions are:

- The ”Drive” activity should display the kilometres that have been driven so far and also the current costs for the drive.
- The amount due should be displayed after each drive.
- 30% of the test users thought that navigation would be a useful extension for the App.
- 20% wanted to be alerted by SMApp, when a toll charged road was approached. This alert should be early enough, such that another route might be chosen. However, this might only be possible, when the application is navigating the user to some destination. Otherwise the alert might still come too late, when the nearest roads do not lead to the desired destination.

Info

In this activity, there was only one thing, users found strange. 30% of them suggested to change the display of whether a certain vehicle or means of payment is set as favourite. Currently, for each vehicle and means of payment, there is a line that says ”Favourite: True/False”, where the corresponding boolean value is given. This, however is not the expression a typical user would choose. Therefore, they suggested to either use ”Favourite: Yes/No” or to just display ”Favourite” in case the corresponding vehicle or means of payment is set as favourite. When an entry is no favourite, nothing about favourites should be displayed for it.

This issue was categorised as a cosmetic problem, as it was perceived to be strange, but it occurred very seldom and it did not prevent anyone from achieving the tasks and no one was slowed down.

When participants were asked about their expectations regarding the ”Info” activity, 70% expected to get some information about the application itself, like an imprint or some help. Other answers were as follows:

- See areas where radars, accidents or congestions are.
- Get information about the ”Drive” activity, prices, possible alternative routes (to avoid congestion or to save money).
- See fastest or shortest routes.
- Tell the application whether to use routes that lead through foreign countries or not.
- Maybe view the map in greater detail and extend the given information. (In this case, the user would contribute in some way.)

Comments about the part of the ”Info” activity that displays the stored data are given in the following. Each of them was only mentioned by one person.

- For each vehicle, it should be displayed, how much toll charges it has cost so far. Maybe the amount of charges that still need to be payed should also be given. Likewise, it should be displayed, how much has been payed with each of the registered means of payment, already. - However, if this was chosen to be implemented, first, it would be necessary to check if the display of such information would be allowed by legal regulations.
- When a detail is clicked, the application should forward the user into the ”Edit” activity. There, the corresponding entry should be displayed, such that the chosen information can be changed right away.
- The display of the stored data was found to be unnecessary, as all of this information can also be found in the ”Edit” activity.

Of course, suggestions were also made for the billing information. Where no other figure is given, the suggestion was only made by one participant.

- 70% of the participants wished that the billing information would contain the driven route (at least the starting and end point), for being able to check the bill.
- All information that has been entered for the vehicle and means of payment, used for a drive, should be displayed with the bill.
- A vehicle’s number of axles and weight is thought to be an unnecessary information.
- It should be possible to print the bill, in order to have an evidence.
- Bills should be sorted according to vehicle names. Additionally, an overall sum should be displayed.
- Each bill should contain tariff information. This means that the used price per km or the vehicles classification should be displayed, in order to check the bill.
- 20% of the test users want to have the possibility to define a certain period for billing, like every week, each month, only once a year etc.

Edit

Expectations regarding the functionality of this activity came very close to its actual implementation. Each participant said that data which has been entered, including user information and vehicle details can be changed or corrected, here. Seldom, other possibilities were expected as well. These other expectations are configuring the App (show points of interest yes/no etc.), selecting or editing the tariff model to be used and editing a route that had been defined previously.

Two problems have been detected for this activity. The first one occurred among 30% of the participants and slowed down their progress, which is why it was categorised as serious problem. People would have preferred a list of vehicle names to the currently implemented drop-down selection. This list should lead to the corresponding details via clicking on a listed name for the selection of an entry to edit. A similar list would be needed for the means of payment.

The second problem occurred only once and did hardly slow down the participant. However, the error message that informs the user about the fact that the (old) password has to be entered before any user data can be changed, was misunderstood. This problem is considered to be cosmetic, as it will not prevent users from getting changes done, but it might confuse them for a while, which is bad user experience.

Unfortunately, the participant could not name the confusing part of the error message. However, it is assumed that it might be the wording ”(old) password”. This intends to say, that the old password has to be given, when the password is changed. If the password remains the same, the user simply needs to enter it into the given field, in order to perform the changes to the other user data. This was already described in the chapter about the actual implementation of SMApp.

Only few wishes for changes have been stated for this activity. Additionally, each of them was only mentioned by one participant. The suggestions are as follows:

- When the user password is changed, it should be required to enter it twice.
- ”Edit” should be named ”Maintain Account”, so that it will immediately be clear what can be done with this activity.
- ”Edit” should be renamed to ”Edit Registration” or something similar. So, basically, this is the same wish as the one before. It just suggests the usage of another description.

SmartMaut Application

This subsection contains problems and comments that could not be assigned to any of the previous subsections and therefore regard the application as a whole.

Interestingly, no participant knew the ”Menu” button, no matter if they had used the Android system before or not. Some participants eventually tried to use it, when they were asked to tell the application that they wanted to stay logged in and could not find this option in any other activity. However, most participants forgot, that the ”Menu” button was pointed out during their interaction with the memo application.

As a result, 40% of the test users expected the setting to stay logged in in the ”Edit” activity and another 20% expected it to be somewhere in the (user) registration activity or in ”Edit”. Very few of them even considered trying the ”Menu” button and most were unable to complete this task at all. This is why the problem of not finding the application settings is considered to be a critical one.

It was pointed out that this option is often shown upon registration. Also the buttons in the main menu that lead to different activities seem to suggest, that everything which can be done with the application is achievable through using them. Thus, users do not even think of looking anywhere else. It was also said that it might help if the menu popped up in the middle of the screen, such that it will be noticed by users, who sometimes did not see it, even if they tried the ”Menu” button. By default, the menu appears in the lower right corner of the display, on devices that provide a hardware button for the menu. Software buttons are usually provided in the *ActionBar*, which is typically located at the top of the screen. These menus appear in the upper right corner of the display. For detailed information about *ActionBars* please refer to [1].

What is very interesting about this particular issue is the fact that placing the application settings into the ”Menu” and placing actions that are used very seldom, like telling the App that it is wished to stay logged in, into the application settings is strongly recommended by the Android design guidelines (for example, see [2]). Obviously, most users do not use the menu in any application, as they were surprised to see that applications they are already using also provide this feature.

One participant had no idea what ”Set as favourite” would do exactly, while another one expected that several favourites would be possible, similar to bookmarks in a browser. These misconceptions were rated as cosmetic problems. Explaining the intended meaning of ”Set as favourite” in the context of SMApp, using a hint or a section in some general help text, might help to prevent confusion among users.

20% of the participants assumed that the application would automatically jump to the different menus for inserting missing information. So, they expected something like the ”Wizard” that is often used for initially installing and configuring computer programs. This was rated as cosmetic problem, because these users found their way through the application within a reasonable amount of time and without forgetting to enter any information. However, the participants remained to be a little insecure, until they saw that all relevant information had been given and that the application worked as intended.

Another cosmetic problem that was pointed out by only one participant is the fact that numeric fields for which no value was given by the user contain ”0”. It was suggested to leave these fields empty, such that the user will know that something is still missing.

An issue that prevented no one from getting their tasks done and which did also not slow down anybody is that the login activity should provide the possibility to reverse one’s password, if it has been forgotten. It was pointed out by 20% of the participants and is considered to be a cosmetic problem. Still, such a feature is considered to be natural and should not have been left out in the first place. Of course, it will greatly enhance usability, if problems like forgetting one’s password can easily be solved.

20% of the participants were constantly looking for some ”Help” menu that would explain when and where to do what for achieving one’s goals in SMApp. As they never really stopped looking for ”Help”, which slowed them down and made them very insecure because they could not find it, this is considered to be a serious problem.

[1] Android API Guide On Action Bars. http://developer.android.com/guide/topics/ui/ actionbar.html Accessed January 2015

[2] Android API Guide On Designing Menus. http://developer.android.com/guide/topics/ui/ menus.html Accessed January 2015

Finally, it seems like leading ”0”s are optimised away by the App, for the security code of credit cards. As this was no problem during the tests but is undesirable, nonetheless, this issue was classified as a cosmetic problem.

Further comments that are not considered to be problems are listed in the following. Again, each piece of feedback that is given without a number of participants who mentioned it, was given by only one test user.

- For 20% of the participants, text boxes are too close to each other. This results in a difficulty to click them.

- Having the possibility to see driven routes in some menu would be interesting for supervisory purposes.

- It was wished to be forwarded to the next possible activity, automatically.

- A favourite should be the first entry in each list, no matter how that list is sorted in general.

- All buttons could contain more or better text to better explain themselves.

- The App should notice that the user is driving and should thus activate ”Drive” automatically. - This would prevent users from forgetting to activate the ”Drive” activity, as long as they are willing to leave SMApp running in the background, all the time. However, the combination of vehicle and means of payment that are used for the current drive, still need to be given correctly, in order to prevent fines.

- Linking the application to an external navigation system in a wireless manner might be nice, as navigation systems are often built into cars, nowadays.

To finish this section, it needs to be said that one of the participants simply did not read any hints at first, even if the information given there would have been needed. However, when the search for more information began, hints were read eventually. The hints are displayed in the input fields in grey colour, until the user starts entering information into these fields.

20% of the participants would have preferred the App to remember their password as default. All other test users did clearly favour the log in for security reasons.

Another 20% of the test users did not know the exact difference between ”Login” and ”Register”. They expected the possibility to register themselves in the login activity. However, when they saw the registration form, they admitted that their previous assumption did not make sense, as registration is always decoupled from logging into an application.

Finally, one participant could not really see the difference between ”Edit” and ”Info”, at the beginning. This was because the data that has been registered for a user can be seen in both activities. ”Info”, however is meant to provide a quick overview and also contains billing data, while ”Edit” allows to change, correct or delete previously provided data.

5.3 Conclusion About Usability Tests

As can be seen in the sections above, the participants really took some time to inspect the ”Smart-Maut” application. They revealed usability issues and also made some very good suggestions about how the system could be improved to gain users.

Most of the issues that were mentioned might be solved by providing a ”Help” menu and by using a nice-looking user interface. Also, additional hints and longer or more accurate description texts on buttons would be helpful.

Overall, the feedback was principally positive. When people were asked to summarise their impression they said that navigating through the menus was logical and easy. Additionally, it was said that it is very good that each activity only contains the most important information.

Each user liked the idea of SMApp and said that it would make life easier and that using it would prevent stress.

All participants would use the application, especially when it would also provide the possibility to buy tickets for a time based system. This is because most of them have some fears regarding a location based system:

The location based system is expected to be much more expensive than the time based system. Additionally, they fear that they could forget their phone or that the battery could die during the drive. If the application would only support the location based system, the decision of whether to use it or not would depend on the price for each system. Of course, this statement assumes that users are allowed to choose between a location based system and a time based system.

Generally, those who have to use toll charged roads very often prefer the time based system or would make their decision dependent on the prices. Those who do not use toll charged roads that much, clearly prefer the location based system.

80% of the participants considered the location based system to be fairer. The rest thought that both systems would be equal or that the answer might depend on the price. However, there was one very interesting thought: if one had to commute to get to work, a location based system might not be well suited.

Clearly, if a location based system would be considered to be used by a road authority, they would have to work on reducing their user’s fears, first.

Using a foreign device to perform the tests was said to have not been a problem. Only one participant would have preferred working with the own device.

During the tests, the technical affinity of each participant could be seen very clearly. The higher the technical affinity of the user was, the easier the usage of the application seemed to be. Younger users who use mobile applications all the time figured out how to use SMApp very fast, as they recognised interaction patterns from other applications.

Finally, it was very interesting to see that most people thought that SMApp had to be related to a navigation system, because it is a mobile application which has something to do with cars and it would be used while driving.

Detected Problem	Critical	Serious	Cosmetic
keyboard hides necessary buttons on login-screen (and sometimes in other features)	x		
”vehicle name” is associated with the brand of the vehicle	x		
”Register” button in App is confusing, as user is already registered, renaming it might help		x	
for vehicle registration, a hint on where to find the required information is wished		x	
”payment name” needs to be explained		x	
thought, that a means of payment has to be registered for/bound to each vehicle		x	
did not notice that vehicle weight is required to be given in tons		x	
when ”Register” would be left without saving data, warn user			x
for selection of expiration date for credit card, ”month”/”year” should be written next to the spinners			x
information about successful save should be displayed longer, maybe bigger			x
in ”Info”, favourite: true/false should be favourite: yes/no			x
”remember me” was expected to be in ”Edit”	x		
”remember me” was expected to be in ”Register (user)”, or ”Edit”	x		
not clear, what ”set as favourite” would do exactly			x
expected several favourites to be possible (like bookmarks in a browser)			x
assumed that App would automatically jump to menus for inserting missing information			x
if no value has been set by the user and it is therefore ”0”, it might be better to leave the field empty, instead of displaying ”0”, such that the user will know that something is still missing			x
login should provide the possibility to reverse one’s password, if it has been forgotten			x
would like to have ”Help”, where App is explained (i.e. when and where to do what)		x	
leading ”0”s in the security code of credit cards seem to be optimised away by the App			x
”Drive”: displayed information should be explained			x
”Drive”: a map would be nice			x
”Drive”: displayed information is too much/too small			x
”Edit”: error message for giving the (old) password before user data can be changed was misunderstood			x
”Edit”: prefers list of vehicle names that leads to the details via clicking on the name		x	

Table 5.1: Classification of detected usability problems

CHAPTER 6

Future Work

Currently, two other master theses, which are work in progress, focus on the development of an intelligent and scalable backend system. Such an intelligent backend system would not only need to support all functionality, which is already provided by the database, used for this prototype, but it would also need to calculate toll charges and bill users. Furthermore, the road authority needs to be able to provide and update pricing information, M-Points and values which can be suggested in *Spinners*, in order to aid users in registering their vehicles and means of payment. Moreover, the communication channels need to be secure. Such a backend will also need its own user interface to provide a convenient way to specify and update data for the road authority.

In addition to developing an intelligent backend, the mobile application itself needs to be optimised. Some of the optimisations that need to be made are already summarised in chapter 5. Moreover, an appealing user interface needs to be designed and implemented. The redesign of the user interface might even help to reduce or remove some of the difficulties which were experienced by the test users. Another useful feature which would definitely enhance user experience would be to show a warning, when a user attempts to delete his/her account. Such a message could explain that the account will be inactivated and that it can be reactivated when the user registers with the same unique ID again.

As already mentioned in chapter 4, the application needs a suitable algorithm for spanning M-Points on demand.

In order to display billing data to the user, legal regulations concerning the storage of such data need to be studied and taken into account.

Of course, SMApp has to be provided for several platforms, in order to reach the whole user group. As it is currently developed for Android only, implementations for other platforms need to be provided.

Finally, it needs to be noted that the current application provides only basic features for paying toll charges in a location based system. Advanced features like declaring co-drivers, sharing toll charges, paying toll charges for someone else or buying toll vouchers and giving them as a present remain to be implemented. Furthermore, the project report [8] contains a list of possible additional features, which might increase safety or harvest a larger user group by creating additional incentives for using SMApp.

CHAPTER 7

Conclusion

In our society, toll charged roads are widely used, which leads to the fact that toll charging systems are very important. As current toll charging systems are expert systems, there are only a few providers worldwide. Furthermore, installation and maintenance of these systems is very expensive and they are not interoperable.

On the other hand, more and more people own at least one smart device, which they take with them, wherever they go. Thus, it seems natural to provide a mobile application for toll payment. This would be very convenient for users, as they do not need to stop during their trips or to buy toll tickets some days before they want to start their trip. The road authority also profits by the usage of a mobile application, as no special infrastructure is needed to perform the task and no OBUs have to be distributed. Furthermore, a mobile application can be adjusted to the needs of every country that wants to use it. Thus, the same application can be used in different countries, which, again, contributes to customer satisfaction.

When thinking about how such a toll charging application could be realised, it becomes clear that countries can generally choose between two systems. In a time based system, toll charges are paid for a specific amount of time, no matter how often the toll charged infrastructure is used during that time. In such systems, one usually has to buy a vignette and stick it onto the car's windscreen. In a location based system, toll charges have to be paid for each kilometre that is driven on a toll charged road. For this system, toll plazas are used. In some countries, OBUs are provided for lorries to avoid congestion.

A mobile application can support a time based system by providing a webshop-like functionality. To support a location based system, the functionality of current OBUs has to be implemented.

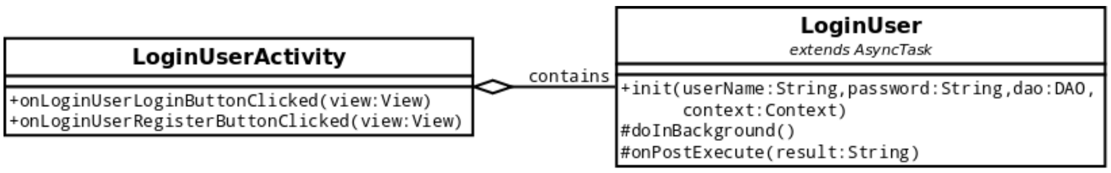
This thesis investigated if it is possible to develop a mobile application, which is suitable for location based toll payment and still remains usable. Therefore, a prototype, which aimed at providing the most important functionality and guiding the user through all necessary steps, was developed.

Usability tests have been performed with ten participants. Although, it was tried to make the application as self-explaining as possible, the test users had troubles some times and they made very good suggestions for improvements. All issues have been categorised from severe to cosmetic. Overall, people liked the notion of having a mobile application for toll payment very much. The proposed functionality was well understood and accepted by the test users. Main ctiticism concerned the graphical layout of the menus, however this was not the objective of this thesis. All in all, the approach taken for SMApp (SmartMaut application) seems to be good and can be used as a basis for future implementations, but for a full-fledged product improvements have to be made.

APPENDIX A

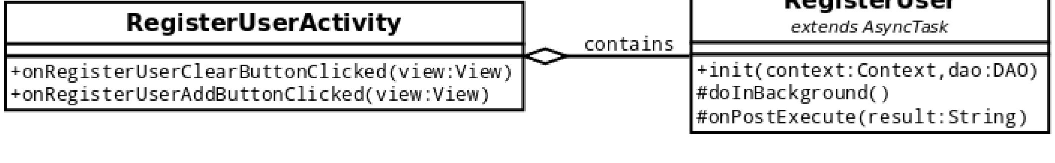
Class Diagrams

This appendix shows the class diagrams of the modules which form SMApp.



LoginUserActivity
+onLoginUserLoginButtonClicked(view:View)
+onLoginUserRegisterButtonClicked(view:View)
contains
LoginUser
extends AsyncTask
+init(userName:String,password:String,dao:DAO,
context:Context)
#doInBackground()
#onPostExecute(result:String)

Figure A.1: Class diagram showing the module "Login".



RegisterUserActivity
+onRegisterUserClearButtonClicked(view:View)
+onRegisterUserAddButtonClicked(view:View)
contains
RegisterUser
extends AsyncTask
+init(context:Context,dao:DAO)
#doInBackground()
#onPostExecute(result:String)

Figure A.2: Class diagram showing the module for user registration.

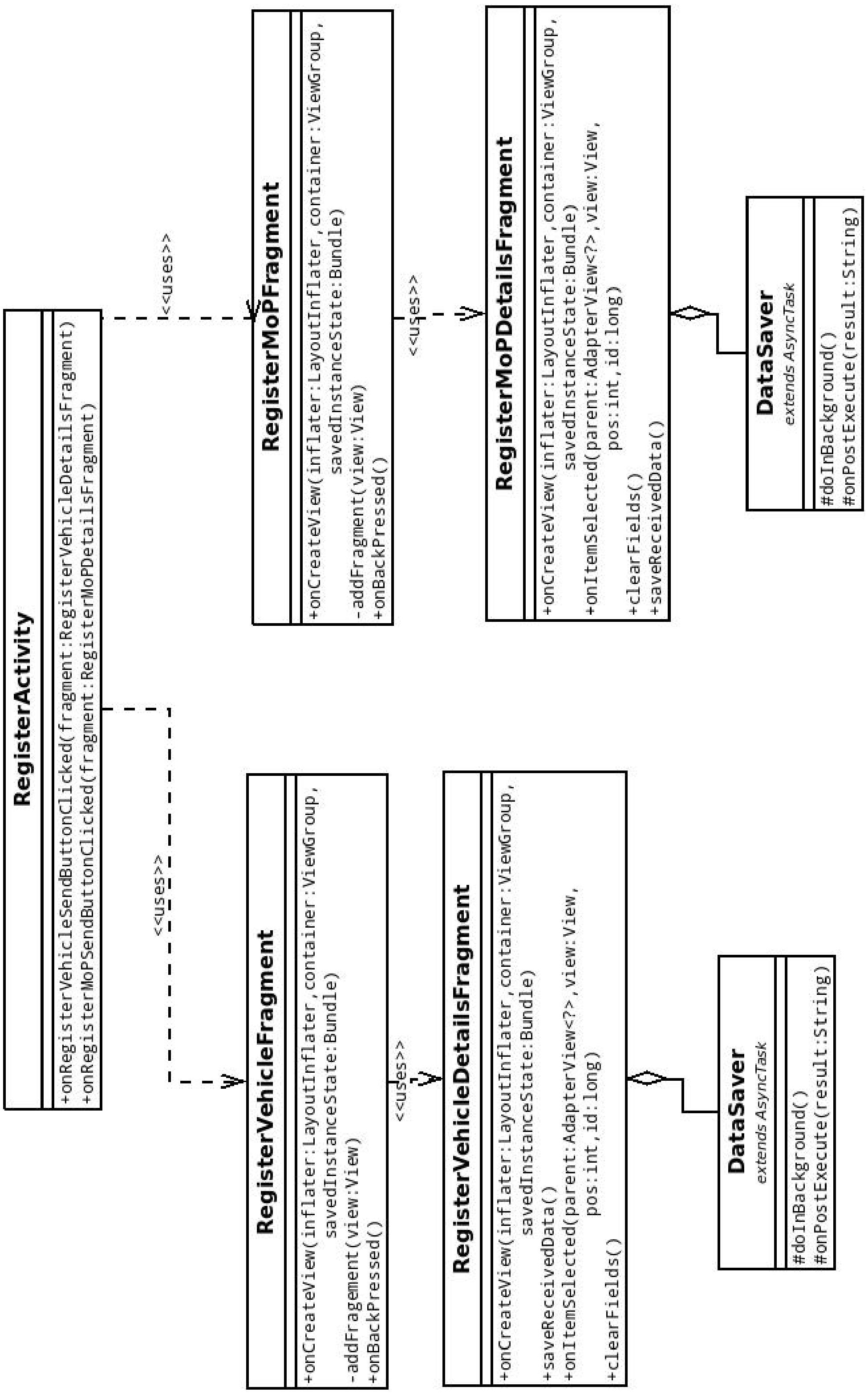

RegisterActivity
+onRegisterVehicleSendButtonClicked(fragment:RegisterVehicleDetailsFragment)
+onRegisterMoPSendButtonClicked(fragment:RegisterMoPDetailsFragment)
<<uses>>
<<uses>>
RegisterMoPFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
-addFragment(view:View)
+onBackPressed()
<<uses>>
RegisterMoPDetailsFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
+clearFields()
+saveReceivedData()
DataSaver
extends AsyncTask
#doInBackground()
#onPostExecute(result:String)
RegisterVehicleFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
-addFragement(view:View)
+onBackPressed()
<<uses>>
RegisterVehicleDetailsFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+saveReceivedData()
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
+clearFields()
DataSaver
extends AsyncTask
#doInBackground()
#onPostExecute(result:String)

Figure A.3: Class diagram showing the module for registering vehicles and means of payment.

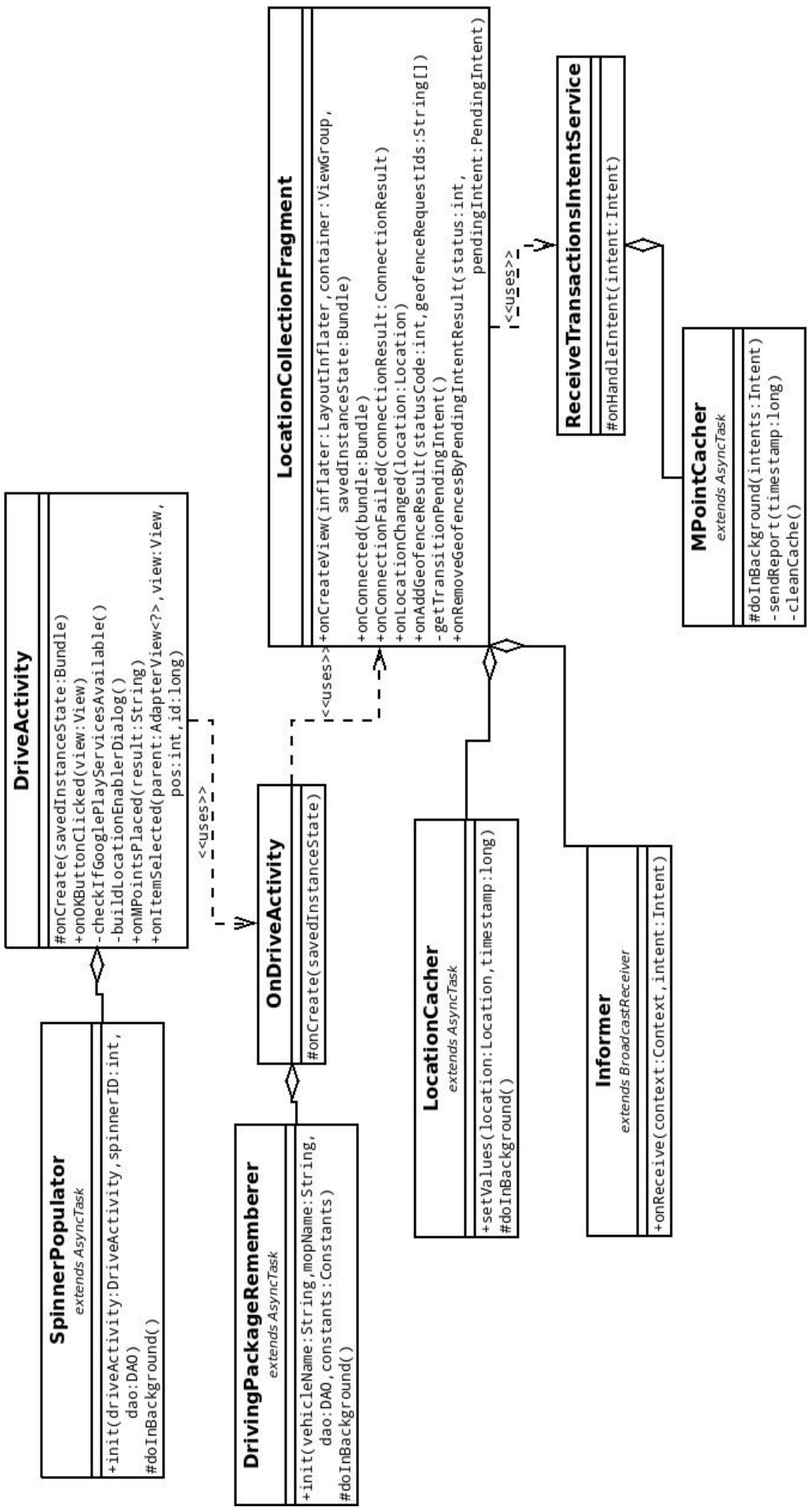

DriveActivity
#onCreate(savedInstanceState:Bundle)
+onOKButtonClicked(view:View)
-checkIfGooglePlayServicesAvailable()
-buildLocationEnablerDialog()
+onMPointsPlaced(result:String)
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
SpinnerPopulator
extends AsyncTask
+init(driveActivity:DriveActivity,spinnerID:int, dao:DAO)
#doInBackground()
<<uses>>
OnDriveActivity
#onCreate(savedInstanceState)
DrivingPackageRememberer
extends AsyncTask
+init(vehicleName:String,mopName:String, dao:DAO,constants:Constants)
#doInBackground()
<<uses>>
LocationCollectionFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+onConnected(bundle:Bundle)
+onConnectionFailed(connectionResult:ConnectionResult)
+onLocationChanged(location:Location)
+onAddGeofenceResult(statusCode:int,geofenceRequestIds:String[])
-getTransitionPendingIntent()
+onRemoveGeofencesByPendingIntentResult(status:int, pendingIntent:PendingIntent)
LocationCacher
extends AsyncTask
+setValues(location:Location,timestamp:long)
#doInBackground()
Informer
extends BroadcastReceiver
+onReceive(context:Context,intent:Intent)
<<uses>>
ReceiveTransactionsIntentService
#onHandleIntent(intent:Intent)
MPointCacher
extends AsyncTask
#doInBackground(intents:Intent)
-sendReport(timestamp:long)
-cleanCache()

Figure A.4: Class diagram showing the module ”Drive”.

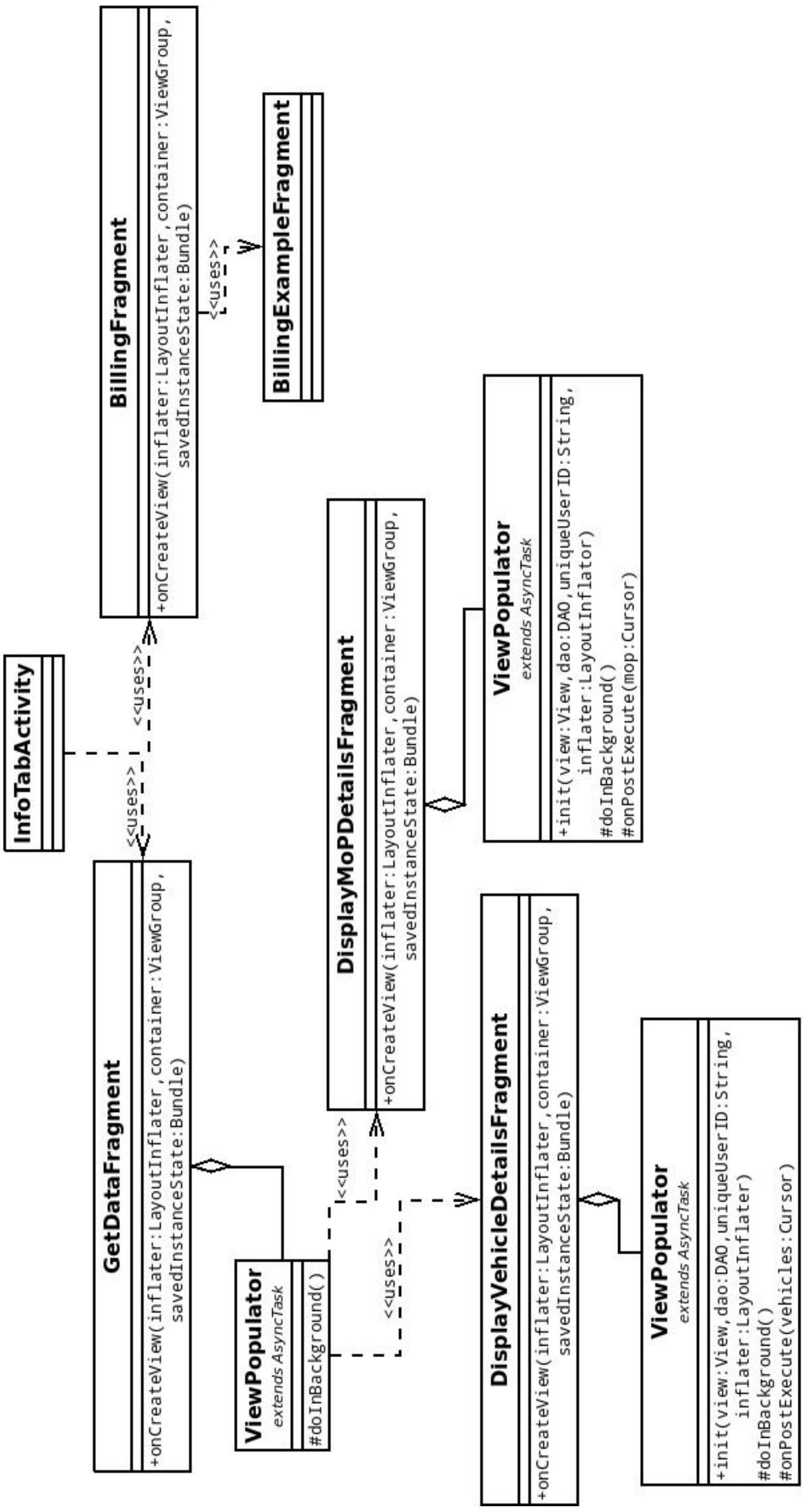
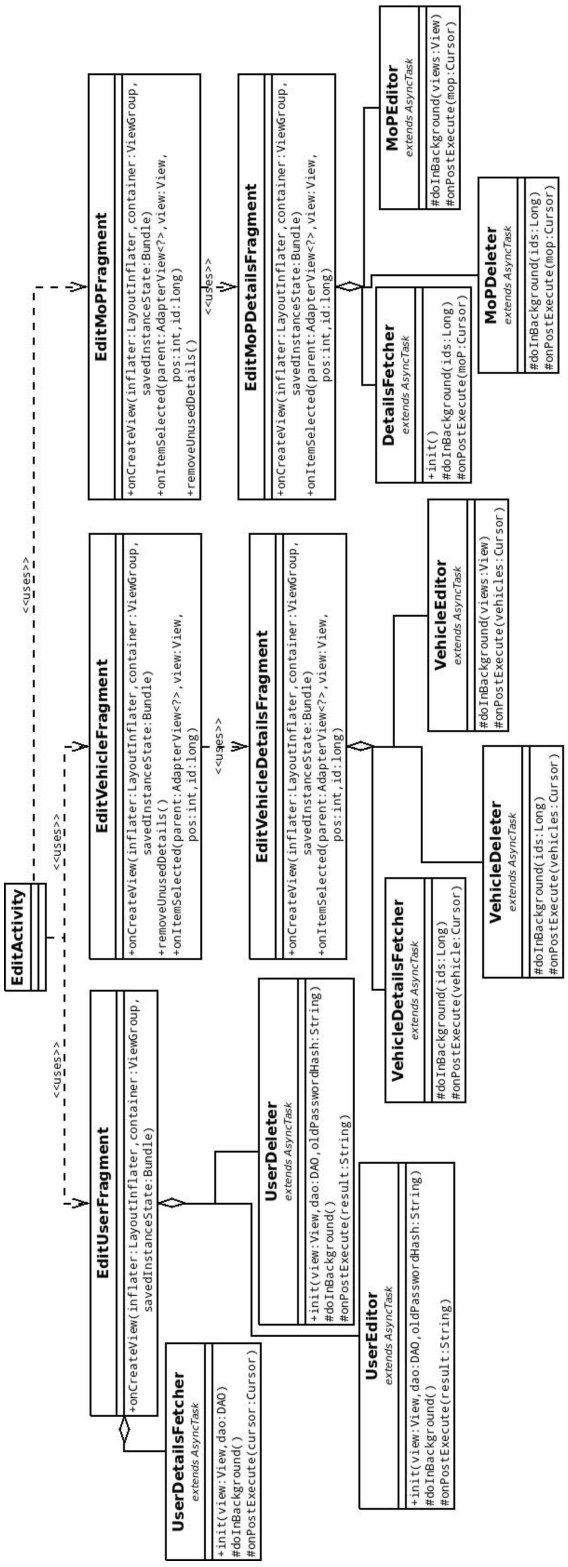
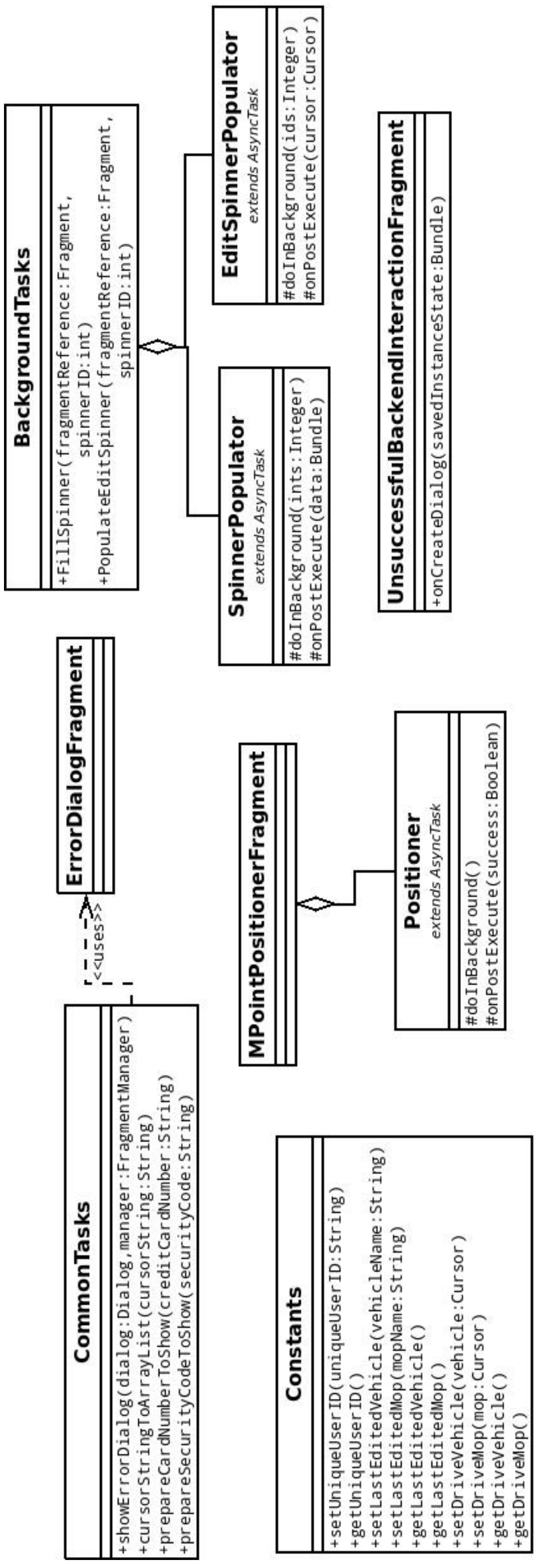

BillingFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
<<uses>>
BillingExampleFragment
InfoTabActivity
<<uses>>
<<uses>>
GetDataFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
ViewPopulator
extends AsyncTask
#doInBackground()
<<uses>>
<<uses>>
DisplayMoPDetailsFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
ViewPopulator
extends AsyncTask
+init(view:View,dao:DAO,uniqueUserID:String, inflater:LayoutInflator)
#doInBackground()
#onPostExecute(mop:Cursor)
DisplayVehicleDetailsFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
ViewPopulator
extends AsyncTask
+init(view:View,dao:DAO,uniqueUserID:String, inflater:LayoutInflater)
#doInBackground()
#onPostExecute(vehicles:Cursor)

Figure A.5: Class diagram showing the module "Info".



EditActivity
<<uses>>
EditUserFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
UserDetailsFetcher
extends AsyncTask
+init(view:View,dao:DAO)
#doInBackground()
#onPostExecute(cursor:Cursor)
UserDeleter
extends AsyncTask
+init(view:View,dao:DAO,oldPasswordHash:String)
#doInBackground()
#onPostExecute(result:String)
UserEditor
extends AsyncTask
+init(view:View,dao:DAO,oldPasswordHash:String)
#doInBackground()
#onPostExecute(result:String)
EditVehicleFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+removeUnusedDetails()
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
EditVehicleDetailsFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
VehicleDetailsFetcher
extends AsyncTask
#doInBackground(ids:Long)
#onPostExecute(vehicle:Cursor)
VehicleDeleter
extends AsyncTask
#doInBackground(ids:Long)
#onPostExecute(vehicles:Cursor)
VehicleEditor
extends AsyncTask
#doInBackground(views:View)
#onPostExecute(vehicles:Cursor)
EditMoPFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
+removeUnusedDetails()
EditMoPDetailsFragment
+onCreateView(inflater:LayoutInflater,container:ViewGroup, savedInstanceState:Bundle)
+onItemSelected(parent:AdapterView<?>,view:View, pos:int,id:long)
DetailsFetcher
extends AsyncTask
+init()
#doInBackground(ids:Long)
#onPostExecute(moP:Cursor)
MoPDeleter
extends AsyncTask
#doInBackground(ids:Long)
#onPostExecute(mop:Cursor)
MoPEditor
extends AsyncTask
#doInBackground(views:View)
#onPostExecute(mop:Cursor)

Figure A.6: Class diagram showing the module ”Edit”.



BackgroundTasks
+FillSpinner(fragmentReference:Fragment, spinnerID:int)
+PopulateEditSpinner(fragmentReference:Fragment, spinnerID:int)
EditSpinnerPopulator
extends AsyncTask
#doInBackground(ids:Integer)
#onPostExecute(cursor:Cursor)
SpinnerPopulator
extends AsyncTask
#doInBackground(ints:Integer)
#onPostExecute(data:Bundle)
UnsuccessfulBackendInteractionFragment
+onCreateDialog(savedInstanceState:Bundle)
ErrorDialogFragment
<<uses>>
MPointPositionerFragment
Positioner
extends AsyncTask
#doInBackground()
#onPostExecute(success:Boolean)
CommonTasks
+showErrorDialog(dialog:Dialog,manager:FragmentManager)
+cursorStringToArrayList(cursorString:String)
+prepareCardNumberToShow(creditCardNumber:String)
+prepareSecurityCodeToShow(securityCode:String)
Constants
+setUniqueUserID(uniqueUserID:String)
+getUniqueUserID()
+setLastEditedVehicle(vehicleName:String)
+setLastEditedMop(mopName:String)
+getLastEditedVehicle()
+getLastEditedMop()
+setDriveVehicle(vehicle:Cursor)
+setDriveMop(mop:Cursor)
+getDriveVehicle()
+getDriveMop()

Figure A.7: Class diagram showing the module "Common".

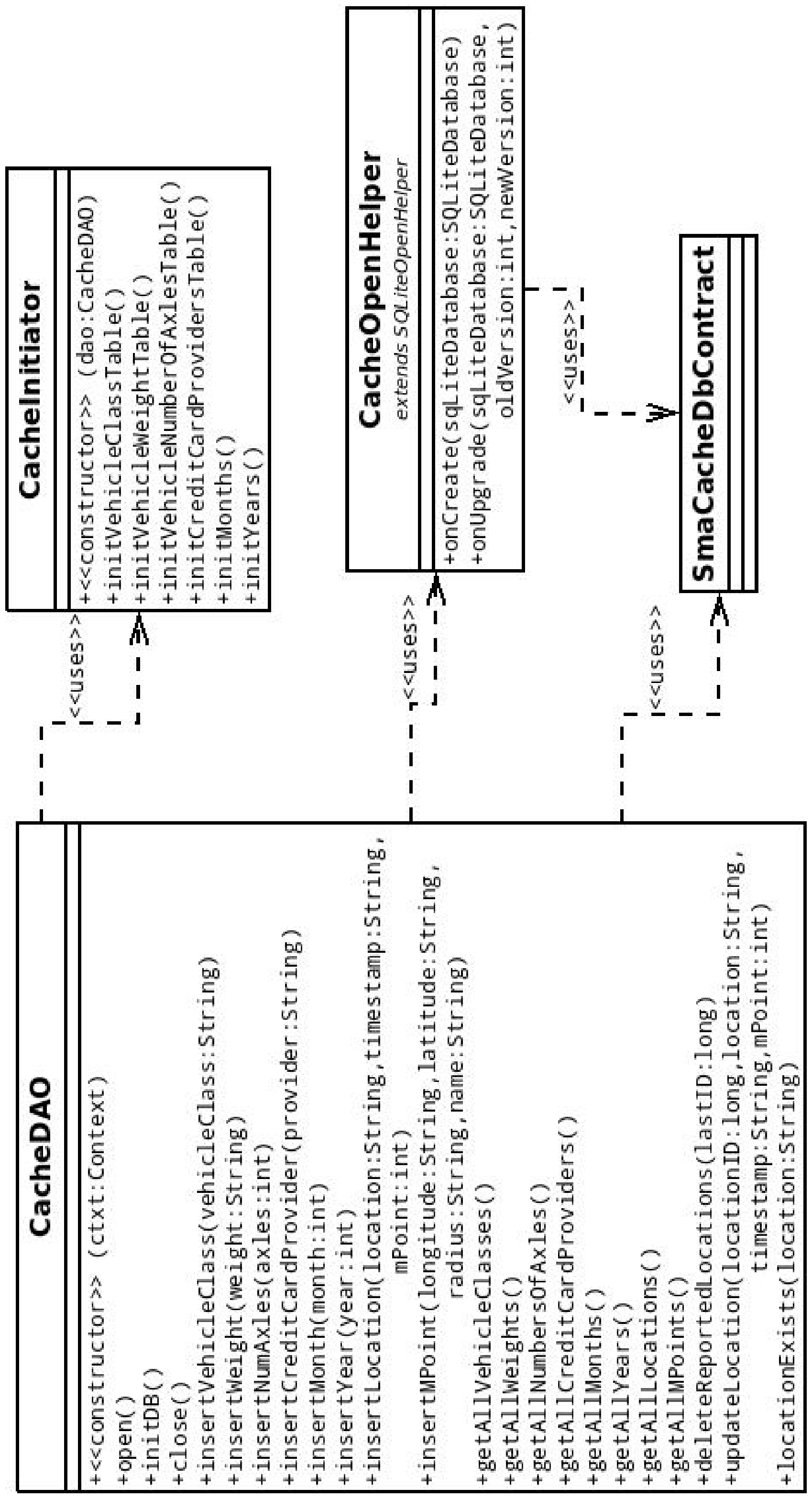
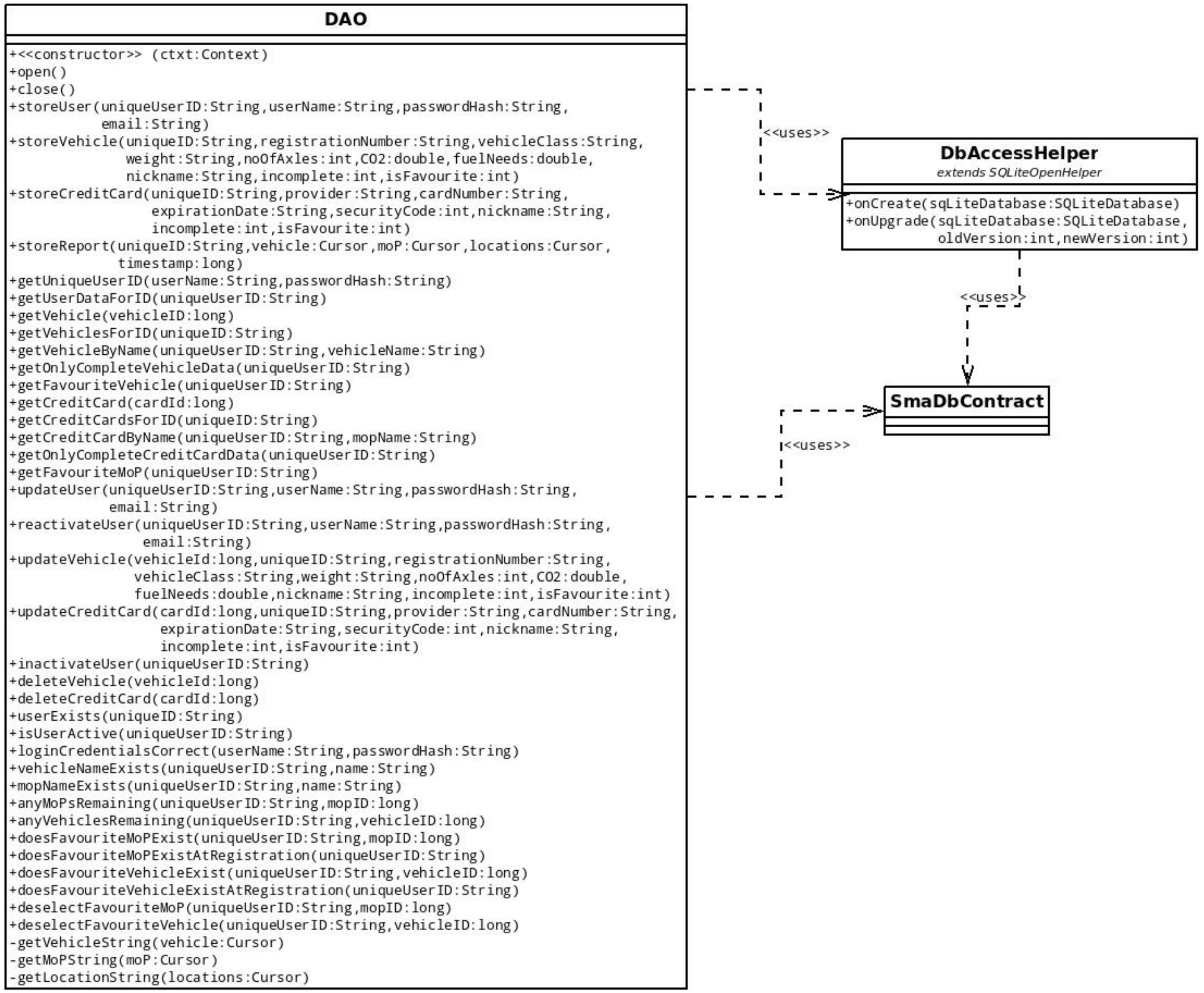

CacheDAO
+<<constructor>> (ctxt:Context)
+open()
+initDB()
+close()
+insertVehicleClass(vehicleClass:String)
+insertWeight(weight:String)
+insertNumAxles(axles:int)
+insertCreditCardProvider(provider:String)
+insertMonth(month:int)
+insertYear(year:int)
+insertLocation(location:String,timestamp:String, mPoint:int)
+insertMPoint(longitude:String,latitude:String, radius:String,name:String)
+getAllVehicleClasses()
+getAllWeights()
+getAllNumbersOfAxles()
+getAllCreditCardProviders()
+getAllMonths()
+getAllYears()
+getAllLocations()
+getAllMPoints()
+deleteReportedLocations(lastID:long)
+updateLocation(locationID:long,location:String, timestamp:String,mPoint:int)
+locationExists(location:String)
<<uses>>
<<uses>>
<<uses>>
<<uses>>
CacheInitiator
+<<constructor>> (dao:CacheDAO)
+initVehicleClassTable()
+initVehicleWeightTable()
+initVehicleNumberOfAxlesTable()
+initCreditCardProvidersTable()
+initMonths()
+initYears()
CacheOpenHelper
extends SQLiteOpenHelper
+onCreate(sqLiteDatabase:SQLiteDatabase)
+onUpgrade(sqLiteDatabase:SQLiteDatabase, oldVersion:int,newVersion:int)
SmaCacheDbContract

Figure A.8: Class diagram showing the cache database.



```
+<<constructor>> (ctxt:Context)
+open()
+close()
+storeUser(uniqueUserID:String,userName:String,passwordHash:String,
           email:String)
+storeVehicle(uniqueID:String,registrationNumber:String,vehicleClass:String,
              weight:String,noOfAxles:int,CO2:double,fuelNeeds:double,
              nickname:String,incomplete:int,isFavourite:int)
+storeCreditCard(uniqueID:String,provider:String,cardNumber:String,
                 expirationDate:String,securityCode:int,nickname:String,
                 incomplete:int,isFavourite:int)
+storeReport(uniqueID:String,vehicle:Cursor,moP:Cursor,locations:Cursor,
             timestamp:long)
+getUniqueUserID(userName:String,passwordHash:String)
+getUserDataForID(uniqueUserID:String)
+getVehicle(vehicleID:long)
+getVehiclesForID(uniqueID:String)
+getVehicleByName(uniqueUserID:String,vehicleName:String)
+getOnlyCompleteVehicleData(uniqueUserID:String)
+getFavouriteVehicle(uniqueUserID:String)
+getCreditCard(cardId:long)
+getCreditCardsForID(uniqueID:String)
+getCreditCardByName(uniqueUserID:String,mopName:String)
+getOnlyCompleteCreditCardData(uniqueUserID:String)
+getFavouriteMoP(uniqueUserID:String)
+updateUser(uniqueUserID:String,userName:String,passwordHash:String,
            email:String)
+reactivateUser(uniqueUserID:String,userName:String,passwordHash:String,
                email:String)
+updateVehicle(vehicleId:long,uniqueID:String,registrationNumber:String,
               vehicleClass:String,weight:String,noOfAxles:int,CO2:double,
               fuelNeeds:double,nickname:String,incomplete:int,isFavourite:int)
+updateCreditCard(cardId:long,uniqueID:String,provider:String,cardNumber:String,
                  expirationDate:String,securityCode:int,nickname:String,
                  incomplete:int,isFavourite:int)
+inactivateUser(uniqueUserID:String)
+deleteVehicle(vehicleId:long)
+deleteCreditCard(cardId:long)
+userExists(uniqueID:String)
+isUserActive(uniqueUserID:String)
+loginCredentialsCorrect(userName:String,passwordHash:String)
+vehicleNameExists(uniqueUserID:String,name:String)
+mopNameExists(uniqueUserID:String,name:String)
+anyMoPsRemaining(uniqueUserID:String,mopID:long)
+anyVehiclesRemaining(uniqueUserID:String,vehicleID:long)
+doesFavouriteMoPExist(uniqueUserID:String,mopID:long)
+doesFavouriteMoPExistAtRegistration(uniqueUserID:String)
+doesFavouriteVehicleExist(uniqueUserID:String,vehicleID:long)
+doesFavouriteVehicleExistAtRegistration(uniqueUserID:String)
+deselectFavouriteMoP(uniqueUserID:String,mopID:long)
+deselectFavouriteVehicle(uniqueUserID:String,vehicleID:long)
-getVehicleString(vehicle:Cursor)
-getMoPString(moP:Cursor)
-getLocationString(locations:Cursor)
```

```
+onCreate(sqLiteDatabase:SQLiteDatabase)
+onUpgrade(sqLiteDatabase:SQLiteDatabase,
           oldVersion:int,newVersion:int)
```

Figure A.9: Class diagram showing the database used as backend.

APPENDIX B

Class Files

This appendix provides a detailed description of some classes that were used for the implementation of the modules, where this is perceived to be interesting.

Only methods which provide functionality that is specific for the use case of toll payment are described. This means that automatically generated methods where nothing has been added or changed are left out, as they are the same for each *Activity* or *Fragment*.

The following sections contain the relevant descriptions of the class files of the corresponding module.

B.1 Drive

DriveActivity.java

When the user enters ”Drive”, *DriveActivity* is shown first. In its *onCreate* method, it initializes the *Spinner* elements for selecting the vehicle and means of payment to be used for the drive. The *Spinners* will only suggest vehicles and means of payment for which the registration is complete. If only incomplete or no data can be found for either of them, an error dialogue is presented to the user. This dialogue tells the user that at least one completely registered vehicle and means of payment is needed for correctly paying toll charges. The user is asked to either register something or to complete data for which registration was already made in an incomplete manner. Following that, *DriveActivity* checks if *GooglePlayServices*, which provides the APIs, needed for accessing GPS data and creating and monitoring M-Points, is available. If not, the user has to perform an update of *GooglePlay*, the deployment platform for Android, in order to obtain them. If, however, the *GooglePlayServices* are available, SMApp checks if GPS is enabled. This is done in the method *checkIfGooglePlayServicesAvailable*. When it is found that GPS is disabled, *buildLocationEnablerDialog* is called. This method displays an error dialogue to the user, which informs about the fact that GPS is turned off and that the drive cannot be reported and should thus not be started without changing this. On clicking ”Enable” in this dialogue, the user is forwarded to the device settings, where GPS can be enabled.

Of course, this *Activity* also implements *onItemSelected*, in order to know the user's selection of vehicle and means of payment to be used for this drive. As soon as the user presses "OK", *MPointPositionerFragment* is called. This is part of the "Common" package, because it might be used by other modules in a future version of SMApp, and will therefore be described in section B.2. To summarise its purpose shortly, it can be said that it is responsible for the creation of *Geofences* for all needed M-Points.

When *MPointPositionerFragment* is finished, *onMPointsPlaced* is called in *DriveActivity*, via the *OnFragmentInteractionListener* of the *Fragment*. This causes *OnDriveActivity* to be started. It is informed about the users selection of the vehicle and means of payment to use.

The only thing that remains to be described for *DriveActivity* is the inner class *Spinner-Populator*. It is an *AsyncTask* and fills the provided *Spinner* elements with data. In order to do that, the *init* method needs to receive a reference to *DriveActivity*, the ID of the *Spinner* element - for distinguishing between the vehicle *Spinner* and the *Spinner* for the means of payment - and a reference to the backend DAO. Furthermore, an instance of *Constants.java* will be created, to obtain the unique user ID of the current user. In *doInBackground*, complete vehicles or means of payment are derived from the backend, depending on the ID of the *Spinner* to populate. Only nicknames are used as values for the list. As soon as the *Spinner* is filled with these values, it is checked if any of these complete vehicles or means of payment have been set as favourite. If so, the *Spinner* is configured to display the favourite entry as first suggestion. If any problems arise during this activity, an according error message is displayed.

OnDriveActivity.java

In *onCreate*, this class retrieves the nicknames of the chosen vehicle and means of payment from the arguments it received and displays this information, such that the user can always see the selection. After that, the inner class *DrivingPackageRememberer* is called. It is an *AsyncTask*, which receives the nicknames of the chosen vehicle and means of payment, as well as a reference to the backend DAO and to *Constants.java* as parameters for its *init* method. In *doInBackground*, the complete information about the chosen driving package is retrieved from the backend and stored in *Constants.java*, such that it can be used for reports without asking the backend for the same data all the time.

When a more intelligent backend is used, it would, of course, suffice to include only the nicknames in the reports, in order to keep the amount of data that has to be sent as low as possible.

When *DrivingPackageRememberer* is finished, *LocationCollectionFragment* is started.

LocationCollectionFragment.java

This class is responsible for requesting and receiving location updates. Furthermore, it starts the *IntentService* that monitors the Geofences and will be described in the next subsection.

In *onCreateView, LocationCollectionFragment* prepares for the location updates, by creating a *LocationClient* and a *LocationRequest*. The *LocationClient* is needed to connect to the needed API services. *LocationRequest* specifies the desired accuracy of location information and the update interval for location data. Please note that choosing the time for this update interval correctly is crucial for being able to monitor *Geofences*. Currently, the interval is fitted for slow movement, like walking. For the application to be usable in cars, the update interval needs to be reconfigured. Of course, it can be implemented that this reconfiguration is made by the application automatically, as soon as it gets known how fast the user can move (e.g. by manually switching the mode from walking to driving).

As soon as the *LocationClient* is connected to the *Location Services*, the callback method *onConnected* is invoked. From this point on, the current location can be requested or periodic updates can be started. So, this method first checks if *Geofences* have already been spanned. This is because the *LocationClient* might connect and disconnect several times to/from the *Location Services*, during the lifecycle of the application. In such a case, the location updates have to be requested again, but the *Geofences* are still in place. Therefore, they do not need to be spanned again. So, if no *Geofences* are available, a request to span the *Geofences* is made to the *LocationClient*. This request contains an instance of *ReceiveTransitionsIntentService*, which is an *IntentService*, as *PendingIntent*. The *PendingIntent* needs to be given to the *Location Services*, such that it can trigger the *IntentService* in this application when a *Geofence* transition occurs. *getTransitionPendingIntent* creates the *PendingIntent*. After *onAddGeofenceResults* receives the information that all *Geofences* have been placed successfully, it requests location updates. If M-Points are available, already, it suffices to request location updates immediately.

onConnectionFailed is another callback method that is called by the *Location Services* if the attempt to connect to them fails. This method tries to resolve the error with predefined methods, which are provided by *GooglePlayServices*. If this is not possible, an error message is displayed to the user.

onLocationChanged is called by the *Location Services* with an location object as parameter, as soon as an update for the current location becomes available. The current system time is requested. Together with the location object, that contains all data about the current location, the update time is handed to *LocationCacher*. This is an *AsyncTask* which stores the location information together with the timestamp in the cache database.

When *LocationCollectionFragment* is stopped, it removes all *Geofences*. The result of this operation is received by the callback method *onRemoveGeofencesByPendingIntentResult*, which is called by *GooglePlayServices*. If the removal was successful, the *PendingIntent* is stopped to avoid errors when the *Fragment* is restarted. The removal of the *Geofences* is necessary, to save resources and to prevent any errors when the application is restarted.

The inner class *Informer* remains to be described. It extends the *BroadcastReceiver* and is therefore used for receiving broadcast events and informing the user about them. This particular receiver gets messages sent from *ReceiveTransactionsIntentService*, as soon as an M-Point is passed. The user can thus be informed about the passed M-Point.

ReceiveTransactionsIntentService.java

The heart of this class is *onHandleIntent*. This method is called by the *Location Services* when the user's device passes an M-Point. First, the received intent is checked for errors. If everything is alright, the transition type is retrieved. It tells if the device entered or exited the *Geofence* or if it stayed within. SMApp only cares for entering transitions. For each triggering M-Point, a broadcast event is sent. Furthermore, the inner class *MPointCacher*, which extends *AsyncTask* is executed with the triggering intent as parameter.

In *doInBackground* the *MPointCacher* will obtain a timestamp and check if the reported location already exists in the cache database. This might happen, as *LocationCollectionFragment* might have received a location update and stored it into the cache database. To avoid double entries, the M-Point is only inserted into the cache database, if it does not exist already. If it does exist, the entry is updated. It gets a new timestamp and it is marked as M-Point.

The next step is calling *sendReport*. It obtains all location data that is currently stored in the cache database, remembers the ID of the last location to be sent and fetches the data about vehicle and means of payment and the unique user ID from *Constants.java*. This data, together with the timestamp, forms a report. *sendReport* tries to send this report to the backend. If sending this operation failed, it will be retried once.

If sending the report was successful, *cleanCache* is called. It deletes all locations that were sent with the latest report from the cache database.

Due to the fact that the backend is currently simulated by a local database, sending the reports will always succeed. Thus, the potential overflow of the cache is not handled, yet. Future implementations will, however, have to deal with this situation. For example, locations which are no M-Points and which are among the oldest data in the cache could be deleted. That way, the relevant information for toll payment is perceived, although part of the cached data has to be deleted.

B.2 Common

BackgroundTasks.java

Registering and editing vehicles and means of payment requires multiple *Spinners* to be filled. In order to ensure a good performance of the application, these filling operations need to run in the background, because they need to fetch data from the backend and from the cache database. However, these operations are very similar to each other, which makes the corresponding code reusable. There are two kinds of *Spinners* that need to be filled: Some *Spinners* aim at helping users with providing predefined details by, for example, suggesting values for vehicle weights or supported credit card providers. Others are needed for showing the nicknames of vehicles and means of payment in the corresponding tabs in ”Edit”, such that the user can select the entry for which details should be fetched. So, this second type of *Spinner* shows data that has been provided by the user.

Therefore, *BackgroundTasks* provides two methods - *fillSpinner*, which populates *Spinners* for selecting details and *populateEditSpinner*, which fills a *Spinner* with the nicknames of registered vehicles or means of payment. Both methods require a reference to the calling *Fragment*, such that the actual *Spinners* can be accessed and the ID of the *Spinner* to fill. Furthermore, both methods invoke an *AsyncTask* to do the job.

SpinnerPopulator is called by *fillSpinner* with the given *Spinner* ID. In *doInBackground*, it determines the *Spinner* to be filled by the ID and fetches the corresponding data from the cache database. This data is put into a *Bundle*, together with the *Spinner* ID and the number of Data that was obtained. *Bundles* allow to return multiple data with only one object, without having to code an own object for this purpose. *onPostExecute* receives this *Bundle* and uses the contained data to actually fill the *Spinner*, which is indicated by the ID. If a problem occurs, a corresponding error message is displayed to the user.

When *populateEditSpinner* is called, it invokes *EditSpinnerPopulator*. In *doInBackground*, the vehicles or means of payment, registered for the logged in user are fetched from the backend, depending on the ID of the *Spinner* to be filled. This data is returned, which means that *onPost -Execute* can use it to fill the given *Spinner* with the nicknames of the corresponding entries. If any name is cached, because its details were edited last in a previous usage of ”Edit”, this entry is preselected. Of course, an error message is shown if something goes wrong.

CommonTasks.java

This class provides methods which are needed to communicate information to the user. *show-ErrorDialog* is called by *DriveActivity*, which is described in the subsection ”Drive”, in ”Main-Menu”, when it determines that *googlePlayServices* is not available. It calls *ErrorDialog - Fragment* to display this information to the user.

cursorStringToArrayList transforms a String containing *Cursor*-Data into an ArrayList, containing String Arrays. This method is needed for displaying debug data about reports, as each report will be one entry in the ArrayList, while each location that was reported will be held in one cell of the String Array. So, this method eases the processing of data that has been reported and now needs to be displayed, for debugging. Such a method might also be interesting for classes that are needed for the functionality provided for end users. Therefore, this method is located within *CommonTasks* and is not embedded into the classes needed for displaying the debug data. A *Cursor* will be returned by queries against databases, containing the results [1].

prepareCardNumberToShow is used by ”Info” and ”Edit” to transform all but the last four digits of the credit card number into ”X”s, before displaying it, to provide a higher degree of security. Similarly, *prepareSecurityCodeToShow* will transform the whole security code into ”X”s. This enables the user to see whether all necessary information has been provided for a credit card, while no one else will be able to obtain this data by exploring the application.

Constants.java

This class mainly contains String constants, which are used throughout SMApp. Most of these constants are used for specifying errors, in order to display correct and specific error messages. Others are used for storing or retrieving user preferences, or for defining update intervals for the location updates in ”Drive”.

Constants also contains data that is simply kept in memory while the App is running, to avoid unnecessary queries to the backend or to enhance user experience. Therefore, getter and setter methods are provided for remembering the unique ID of the user that is currently logged in - when SMApp is configured to remember the login credentials of a user, the unique user ID is stored with them and will be loaded into *Constants* upon startup. Getter and setter methods are also available for remembering the vehicle or means of payment that was last edited, in order

[1] Android Reference Cursor. http://developer.android.com/reference/android/database/ Cursor.html Accessed January 2015

to display it, when ”Edit” is reopened, and there are also methods for remembering the vehicle and means of payment that have been chosen for the current drive, such that this information can be added to reports, before they are sent, without repeatedly querying the backend for the same information.

MPointPositionerFragment.java

In its *onCreate* method, *MPointPositionerFragment* calls the inner class *Positioner*, which is an *AsyncTask*. In *doInBackground*, this class retrieves all M-Points from the cache database and uses this data for the creation of so-called *”Geofences”*. A *Geofence* is a circular geographical area, which can be monitored. When a user crosses its boundary, an alert will be sent (see [2]). All created *Geofences* will be collected in a list, which is stored in *Constants.java*. If all *Geofences* could be created successfully, the *OnFragmentInteractionListener* is called to inform *DriveActivity*. This means, that *onMPointsPlaced* is called in *DriveActivity*. In case of a failure, an error message is displayed.

UnsuccessfulBackendInteractionFragment.java

This is a *DialogFragment*, which is used to generate small windows, containing error messages and an ”OK” button for dismissing them. This class is used every time, when the application needs to display an error message to the user. When *UnsuccessfulBackendInteractionFragment* is instantiated, it receives one of the String constants, defined in *Constants*, for being able to display the according error message. The name of this class stems from the fact that most errors will occur during an operation that wants to communicate with the backend in some way.

[2] Android location API – Geofence. http://developer.android.com/reference/com/google/ android/gms/location/Geofence.html Accessed January 2015

APPENDIX C

Questionnaire for the Usability Tests

The questionnaire for the usability tests looks as follows:

- First of all, demographic data about the participants is collected. This also includes their technical affinity (ranging from ”high” to ”low” in four steps), their frequency of using smart devices and mobile apps (both ranging from ”always” to ”never” in four steps), if they are familiar with the Android system and their frequency of using toll charged roads.
- To ensure a maximum in usability in future iterations of our application, the test users were asked, what they expected from the four buttons, ”Register”, ”Drive”, ”Info” and ”Edit”, in the main menu, before clicking on any of them.
- Similarly, they were asked about their feelings of what SMApp really provides for the aforementioned four main menu buttons. Did it make sense that the activities were implemented that way? The participants were also asked to mention everything that they would want to be changed in these activities. This methodology was especially inspired by [6].
- Another set of questions was dedicated to the perception of the information given while the ”Drive” activity is active.
- It was perceived to be interesting to know if users would prefer a location based system or a time based system and which of these two they think to be fairer. So, this was also asked.
- Of course, the participants were asked to reflect about their experience of working with the application, tell us the tasks they found particularly difficult, if any and show us what the biggest problem of the App was, in their opinion.
- We were curious as to know if anyone had a problem with using a foreign device, which is also a tablet instead of a smart phone.

- Finally, the participants were asked if they would use the application if it was offered as an alternative to the traditional vignette. For this last question about whether an application would be preferred to the traditional vignette, it was assumed that the application would allow the user to choose between the time based system and the location based system.

Glossary

App short for application. 3, 16, 22, 23, 27, 34, 35, 37, 45, 53, 56, 58, 60, 61, 63, 64, 67, 85, 87

co-payer person that is driving together with someone else and wants to pay some share of the toll charges for the trip in question. 20–22, 24–27, 30

DAO Data Access Object. 37

driving package assignment of vehicles to co-drivers/co-payers (important for discounts/sharing); also the approval certificate is needed; for the prototype implementation, it simply means the selection of a vehicle and means of payment that should be used for a certain drive. 22, 82

IDE Integrated Development Environment. 37

legitimate owner the person to which the vehicle belongs and that will be punished for any trespassing that might occur. 19, 20, 27

location based system a toll charging system, where users have to pay for each km that has been driven on the toll charged infrastructure. 3, 6, 7, 29, 71

lorry a vehicle with a maximum permissible weight of more than 3.5t. 6, 7

M-Point short for ”toll point” (german: ”Maut-Punkt”); this is a (virtual) point along the road, which is used to determine the amount of the toll charges that have to be paid. vii, 19, 27–29, 43, 44, 47, 48, 53, 59, 69, 81–84, 86

OBU On Board Unit. v, vii, 1, 3, 6–9, 71

payment authority registered for a vehicle/registration number as the person who will pay the full amount of the toll charges, as long as there are no co-payers. This person does not need to be in the car during a trip, but each car/registration number needs someone registered as payment authority. (Of course, there can only be co-payers if a payment authority is registered.). 20–22, 24, 26, 27, 30

responsible uploader has to be present in the car and is responsible for the upload of the coordinates, such that the trip can be billed correctly. There is no need for this person to be the driver, if co-drivers are present during the whole trip.. 20–22, 24, 27, 30

road authority legal entity who is responsible for maintaining the higher order road infrastructure (like motorways or clearways) and who is therefore allowed to collect toll charges. 19, 27, 29, 65, 69, 71

SMA SmartMaut. 19, 21–24, 28

SMApp SmartMaut application. 21–23, 29, 31, 33–36, 40, 42–44, 46–49, 51–53, 57, 58, 60, 62–65, 69, 70, 72, 73, 81–83, 85, 87

surveillance check users of toll charged roads and the vehicles they use, for being able to show that someone is guilty of a toll offence. 26

time based system a toll charging system, where the user obtains the permission to use the toll charged infrastructure for a specified amount of time, upon payment. 5, 7, 29, 71

wazer a user of the mobile application WAZE. 12

Bibliography

[1] E Antonucci, F Garzia, and GM Veca. The automatic vehicles access control system of the historical centre of rome. *Sustainable City II*, pages 853–861, 2002.

[2] Nis Bornoe and Jan Stage. Usability engineering in the wild: How do practitioners integrate usability engineering in software development? In Stefan Sauer, Cristian Bogdan, Peter Forbrig, Regina Bernhaupt, and Marco Winckler, editors, *Human-Centered Software Engineering*, volume 8742 of *Lecture Notes in Computer Science*, pages 199–216. Springer Berlin Heidelberg, 2014.

[3] Henry Been-Lirn Duh, Gerald C. B. Tan, and Vivian Hsueh-hua Chen. Usability evaluation for mobile device: A comparison of laboratory and field tests. In *Proceedings of the 8th Conference on Human-computer Interaction with Mobile Devices and Services*, MobileHCI '06, pages 181–186, New York, NY, USA, 2006. ACM.

[4] Damianos Gavalas and D. Economou. Development platforms for mobile applications: Status and trends. *Software, IEEE*, 28(1):77–86, Jan 2011.

[5] Tor-Morten Grønli, Jarle Hansen, and Gheorghita Ghinea. Android vs windows mobile vs java me: A comparative study of mobile development environments. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pages 45:1–45:8, New York, NY, USA, 2010. ACM.

[6] Anne Kaikkonen, Aki Kekäläinen, Mihael Cankar, Titti Kallio, and Anu Kankainen. Usability testing of mobile applications: a comparison between laboratory and field testing. *Journal of Usability Studies*, 1(1):4–16, 2005.

[7] Jesper Kjeldskov and Jan Stage. New techniques for usability evaluation of mobile systems. *International Journal of Human-Computer Studies*, 60(5–6):599 – 620, 2004. {HCI} Issues in Mobile Computing.

[8] Natalie Kollarits. Smartmaut – project report, 2014. TU Vienna, E185/1.

[9] K. Naito. Toll collection system, on board unit and toll collection method, October 4 2001. US Patent App. 09/813,131.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[11] Bernhard Sterzbach. Gps-based clock synchronization in a mobile, distributed real-time system. *Real-Time Systems, Kluwer Academic Publishers*, 12(1):63–75, 1997.

[12] Bernhard Sterzbach and Wolfgang A. Halang. A mobile vehicle on-board computing and communication system. *Computers & Graphics*, 20(5):659 – 667, 1996. Mobile Computing.

[13] S. Tengler and J. Auflick. Vehicle on-board unit, June 30 2009. US Patent 7,554,435.

[14] S. Tengler and R. Heft. Vehicle on-board unit, October 7 2008. US Patent 7,433,773.

[15] Anthony I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.