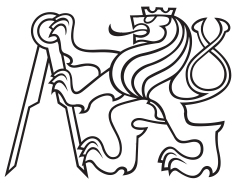
Master's Thesis



Czech Technical University in Prague

F8

Faculty of Information Technology
Department of Software Engineering

Tablet infotainment system

Bc. Michael Bláha

January 2016
Supervisor: Ing. Jan Šedivý, CSc.

CZECH TECHNICAL UNIVERSITY IN PRAGUE

FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	A tablet infotainment system
Student:	Bc. Michael Bláha
Supervisor:	Ing. Jan Šedivý, CSc.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2016/17

Instructions

The in-car infotainment systems are reflecting the current trend to digital lifestyle. People are taking their mobile devices to cars and use them while driving. The current applications are not designed for in-car use, they draw too much of driver's attention and they are negatively impacting the road safety. Design a new Android tablet infotainment application minimizing the additional cognitive load. Proceed following the next steps:

1. Review existing Android applications for in-car use.
2. Review and analyse User Interface development methods for in-car infotainment applications.
3. Analyze the in-car OBD API and exported data.
4. Design an application system architecture for accessing the OBD data and resources.
5. Design a tablet User Interface for in-car use.
6. Design and implement in-car application offering the OBD data for Android tablet platform.
7. Perform UI and application testing and evaluate results (unit testing).

References

Will be provided by the supervisor.

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

prof. Ing. Pavel Tvrdík, CSc.
Dean

Acknowledgement / Declaration

I would like to express my gratitude to my master thesis supervisor, Ing. Jan Šedivý, CSc., for his support, time and valuable advice.

I am also deeply grateful to the Department of Vehicle Technology, Faculty of Transportation Sciences, CTU for all the help and patience with testing and everything related.

I would also like to thank my college, Bc. Lukáš Hrubý for cooperation and advice.

This work was partially supported by CZECH TECHNICAL UNIVERSITY MEDIA LABORATORY and I am grateful for that as well.

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

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

Prague, 12th January 2016

.....

Abstrakt / Abstract

Cílem této práce je navrhnout grafické uživatelské rozhraní pro použití tabletu v automobilu, které minimalizuje kognitivní zátěž a přitom poskytne požadovanou funkcionalitu. Dalším cílem je s použitím tohoto GUI vyvinout tabletovou aplikaci, která umožní zobrazení informací obdržených z automobilu pomocí OBD. Posledním cílem je otestovat tabletovou aplikace ve vhodném prostředí.

Klíčová slova: Android, Java, OBD, GUI, auto, LCT, A/B testování, testování použitelnosti

The goal of this thesis is to design a graphical user interface for in-car tablet usage, which will minimize the cognitive load and still offer the required functionality. Next goal is to develop a tablet application using this GUI and displaying the information obtained from a car via OBD. Final goal is to test the tablet application in a suitable environment.

Keywords: Android, Java, OBD, GUI, car, LCT, A/B testing, usability testing

Contents /

Tables / Figures

Chapter 1
Introduction

1.1 Motivation

In the modern era of portable electronic devices people use these devices daily. Unfortunately, even when it is inappropriate – for example in cars during driving. Such usage can easily cause safety hazards and often they actually do[1]). This situation calls desperately for a proper solution. As the usage of these devices is forbidden while driving and yet drivers still use them, a prohibition is not the solution. And when people do not adapt, the environment has to. While we cannot change the way of transportation, we can change the way of controlling these devices. Therefore an application will be made, trying to solve this issue and helping the driver do all the tasks he wants to do, but in a safe way without endangering the driver himself and everybody else who might get hurt in a possible accident.

1.2 Project

The goal of this project is to create an application that will enable users to use their android tablet safely while driving. In addition to focusing on usability and minimizing the cognitive load, this application should offer rich variety of use cases in a simple but attractive design. For the purpose of this project, this application will be referred to as "CarDashboard".

First step in the project is reviewing existing applications, as it is essential for better insight and inspiration. Designing a proper user interface will follow. To design a user interface for a car environment is not a simple task, because it is the main mean of interaction with the driver, it's quality is the most important factor in driver's distraction when controlling the application. After designing the GUI, an application will be developed implementing the given GUI. It will enable the driver to communicate with car (currently using read-only operations) via OBD. After the development process, the application will be properly tested using a car simulator. The tests will be thoroughly evaluated using various evaluation methods.

[1]) `http://www.dailymail.co.uk/news/article-2591148/One-four-car-accidents-caused-cell-phone-use-driving-five-cent-blamed-texting.html` [1]

1.3 Assignment analysis

1.3.1 Assignment tasks

1.3.1.1 Review existing Android applications for in-car use

One of the key approaches in research project is reviewing the existing progress in the given field. Reviewing existing applications helps to understand the topic, see the bigger picture, learn from mistakes of others and last but not least, to get a general idea about competition.

1.3.1.2 Review and analyze User Interface development methods for in-car infotainment applications

Considering the car environment, the user interface must deal with a lot more problems than usual. This task will review existing User Interface development rules and apply them to the car environment, then analyze them and choose a proper method for car-UI design process.

1.3.1.3 Analyze the in-car OBD API and exported data

On-Board Diagnostics API is a standard API provided by modern cars for gathering various information, such as speed or engine temperature. This task focuses on understanding and gathering data from the OBD API.

1.3.1.4 Design an application system architecture for accessing the OBD data and resources

Having the data from OBD and preparing an application for displaying them, designing a proper architecture is required for everything to work well. The application has to gather data and display them properly without unnecessary delay.

1.3.1.5 Design a tablet User Interface for in-car use

After reviewing existing applications and UI development methods, the next goal is to create new User Interface for in-car use, while considering the constraints this environment puts on it.

1.3.1.6 Design and implement in-car application offering the OBD data for Android tablet platform

With everything prepared and thought through, the application will be developed based on result from all the tasks accomplished so far. In this case, the Android platform will be used, as explained later in the text.

1.3.1.7 Perform UI and application testing and evaluate results

For best results the application must and will be tested. Both code and UI must be tested properly, using various testing approaches, such as unit tests or usability testing with real users in a car simulator.

Chapter 2
Analysis

This chapter is about the process of analyzing resources, researches, tools and project related areas. It contains research about existing related applications, possible application platforms and a description of the chosen platform in a more detailed way. It also presents a basic insight into GUI design process as well as general GUI principles and GUI specifics for the car environment. Then the application idea is described from the point of view of use-cases and tasks followed by the review of tools used for development. Finally the OBD is described with it's API and the data it provides.

2.1 Existing applications

An important step in developing a new application is checking related applications (if they exist) for valuable information to learn from. Based on applications listed in an article[1]), multiple applications are reviewed and analyzed, listing their advantages and disadvantages.

2.1.1 Torque

Torque[2]) can actually show almost anything that OBD (described in section 2.9 on page 19) provides. It is currently the most downloaded application from all the listed applications.

Starting with an empty screen, a lot of settings are required before using this application since there is no default mode. Adding new views is intuitive, but the add menu lacks hierarchy and everything is just a list of various options. There is no cancel button when popping the menu dialog. Several kinds of displays are supported, but it is hard to tell by their names. Responsiveness it not smooth at all and launching the application in a horizontal mode is confusing, as everything behaves like if it was in a vertical mode.

[1]) `http://www.makeuseof.com/tag/5-best-dashboard-car-mode-apps-android-compared/`
[2]) available from `https://play.google.com/store/apps/details?id=org.prowl.torquefree&hl=en`

Figure 2.1. Screenshot from Torque

2.1.1.1 Advantages

- High amount of data from OBD available,
- various layout settings and themes,
- HUD mode.

2.1.1.2 Disadvantages

- One-level confusing menu without hierarchy,
- limited size options for displays (3 types),
- lacks default mode with predefined displays,
- hard to place displays, the grid does not work well,
- slow and unresponsive.

2.1.2 CarHome Ultra

CarHome Ultra[1]) appears to be just a simple application offering speed, compass, weather and external application launcher. New version also displays a location (an address) and a phone version is able to reply to text messages. It also supports text-to-speech feature (on touch).

The application starts with a pop-up tutorial for it's elementary functionality, telling the user about a speedometer, a compass, a weather forecast and a customizable dashboard for launching external applications. In default it offers Google Maps, Google Navigation and a voice search. Adding another external application shortcut is done by tapping the tab. There are also some basic settings, which offer brightness mode (day, night, auto), theme and safety options.

[1]) available from `https://play.google.com/store/apps/details?id=spinninghead.carhome&hl=en`

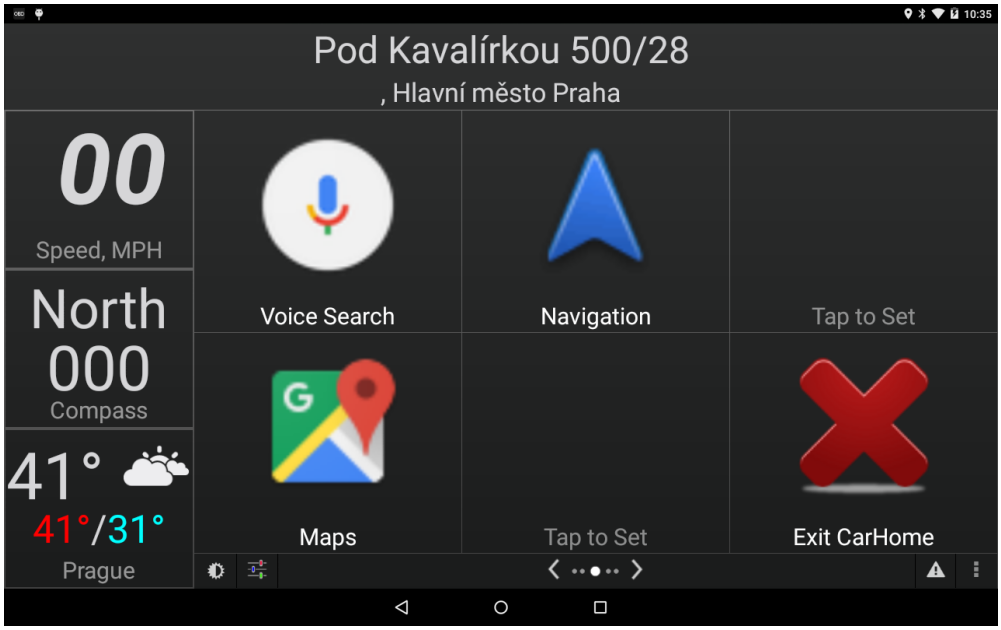


Figure 2.2. Screenshot from CarHome Ultra

2.1.2.1 Advantages

- Simple UI, easy to understand,
- responsive, fluent,
- possible to change units (mile/km, etc.),
- lot of themes available,
- adjustable update rates,
- a lot of different settings.

2.1.2.2 Disadvantages

- Small buttons on small screens (fixed amount of (six) buttons),
- even smaller setting buttons,
- limited functionality,
- tapping weather makes the app speak for every single tap, no matter if it already speaks (it can speak for hours after a lot of taps).

2.1.3 Car Dashdroid

Car Dashdroid[1]) is another similar application providing basic information and functionality. It also provides settings for Bluetooth communication, brightness, screen rotation, full-screen, day/night mode and application settings, where other options can be set, such as home address, theme, units, etc.

After a long on-load time of the application, a main window appears. It has three screens which change by swiping right or left. The left screen contains a dial keyboard, the right screen contains customizable cards (for external application shortcuts or built-in tools) and the main screen consists of weather, speed and shortcuts to contacts, music, navigation and voice commands.

[1]) available from `https://play.google.com/store/apps/details?id=com.nezdroid.cardashdroid&hl=en`

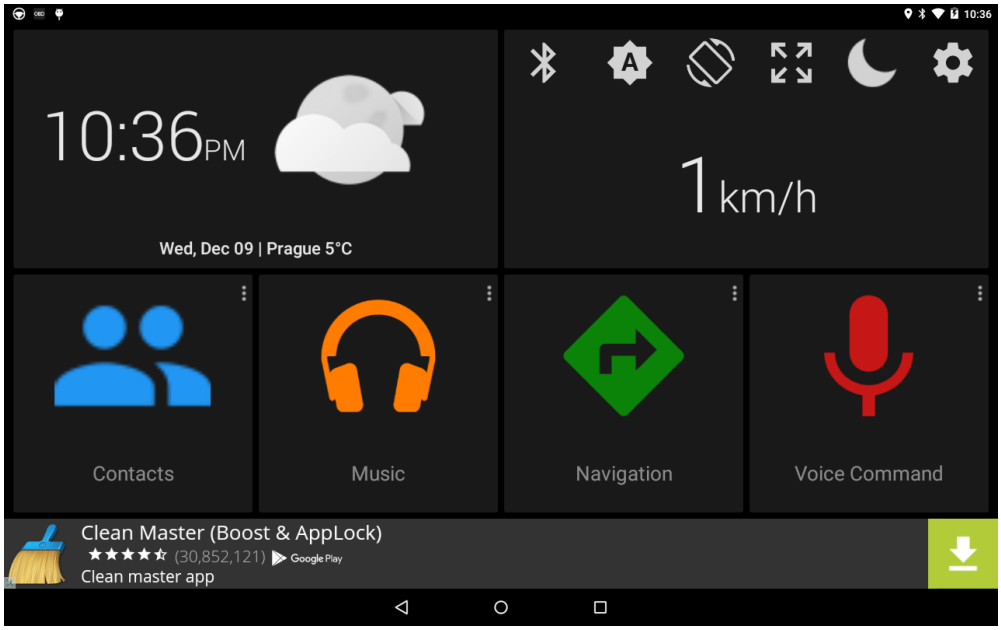


Figure 2.3. Screenshot from Car Dashdroid

2.1.3.1 Advantages

- Simple UI, easy to understand,
- responsive, fluent,
- possible to change units (mile/km, etc.),
- able to read incoming SMS using TTS.

2.1.3.2 Disadvantages

- Very limited functionality,
- not optimized for a tablet,
- distractive commercial ads in a free version.

2.1.4 Ultimate Car Dock

While the design is very similar to CarHome Ultra, Ultimate Car Dock[1]) offers fewer displays on a single screen. There are five screens, each one consists of six cards. Every card can change into shortcut or a build-in application. The Ultimate Car Dock has only few built-in applications: music player, voice command, speed, weather, messages and calls. It also supports shortcuts to other applications.

[1]) available from `https://play.google.com/store/apps/details?id=com.appsontoast.ultimatecar-dock&hl=en`

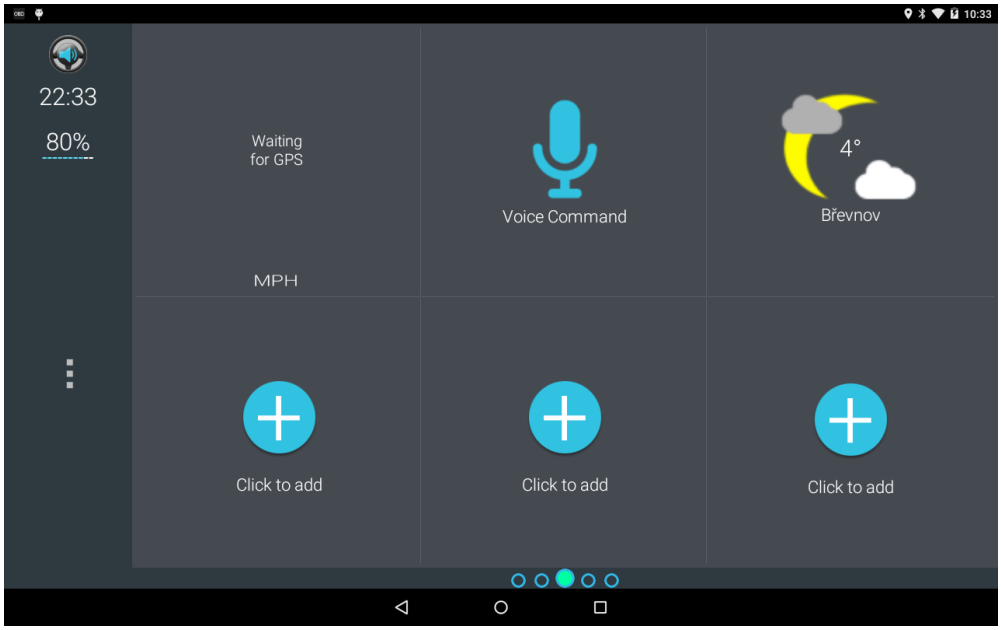


Figure 2.4. Screenshot from Ultimate Car Dock

2.1.4.1 Advantages

- Simple UI, easy to understand,
- responsive, fluent,
- possible to change units (mile/km, etc.),
- able to read various incoming notifications using TTS (Gmail, WhatsApp, etc.),
- predefined SMS responses (selectable when a message comes),
- direct calls and messages (shortcut to call/message a certain person).

2.1.4.2 Disadvantages

- Limited functionality,
- not optimized for a tablet,
- small text font.

2.1.5 Conclusion

Except by Torque, which focuses mainly (and only) on OBD, all the applications are very similar to each other. They have similar design and functionality – mostly weather, speed provided by GPS, a voice command feature and shortcuts for external applications.

2.1.5.1 Suggestions

- OBD support,
- shortcuts to other applications,
- adjustable cards,
- built-in cards (weather, speed, voice command, etc.),
- simple grid-based UI,
- possibility to change displayed units,
- responsive and fluent,
- day and night theme,

- predefined message and call responses,
- TTS for incoming notifications.

2.1.5.2 Possible issues to avoid

- Slow responsiveness,
- limited functionality,
- small and hardly visible font,
- distractive ads.

2.1.6 Android Auto

Recently, Google Inc. presented new application model for information delivery while driving [2]. It is called Android Auto and it provides a standardized user interface and user interaction model for Android devices. Focusing on minimizing the driver distraction, it presents a few options to interact with a user. It supports three application types:

- System overview,
- audio applications,
- messaging applications.

2.1.6.1 System overview

System overview is supposed to be a home screen for an Android Auto application. It presents both current and past notifications. The amount of notifications is limited based on screen size. Every notification consists of an intent icon, a text and an image, while following certain sizing rules. Every such notification can be expanded on the spot or another sub-application can be launched.

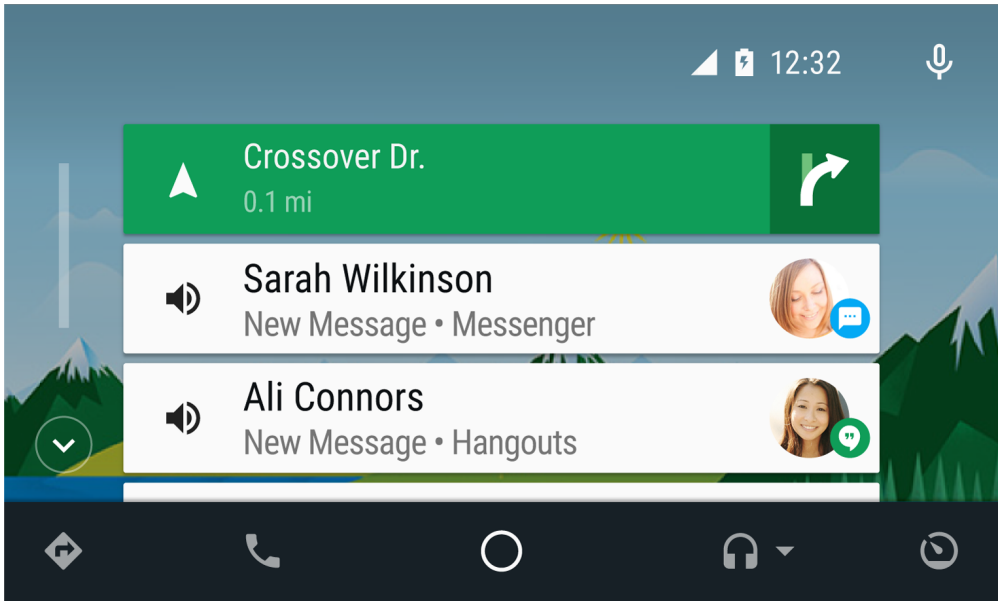


Figure 2.5. Android Auto Home screen

2.1.6.2 Audio applications

Audio applications in Android Auto have a simple template structure. It consists of a main consumption view, a drawer and a queue screen. The main consumption view

displays a few control elements and a cover background. The drawer is a simple list and provides access to favorite and popular content. Finally the queue screen displays a list of pending content (for example songs in a queue).

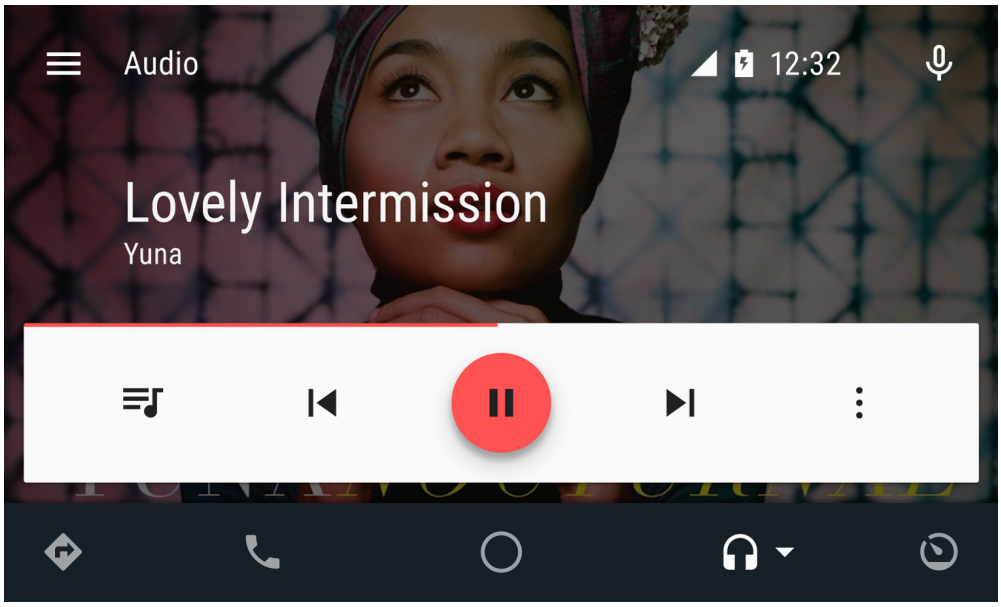


Figure 2.6. Android Auto audio application

2.1.6.3 Messaging applications

Focusing on minimizing the cognitive load, messaging concept in Android Auto prefers voice control to looking and typing. It allows reading the message out-loud and responding with a set of predefined voice commands as well as dictating a whole message using a built-in speech recognition.

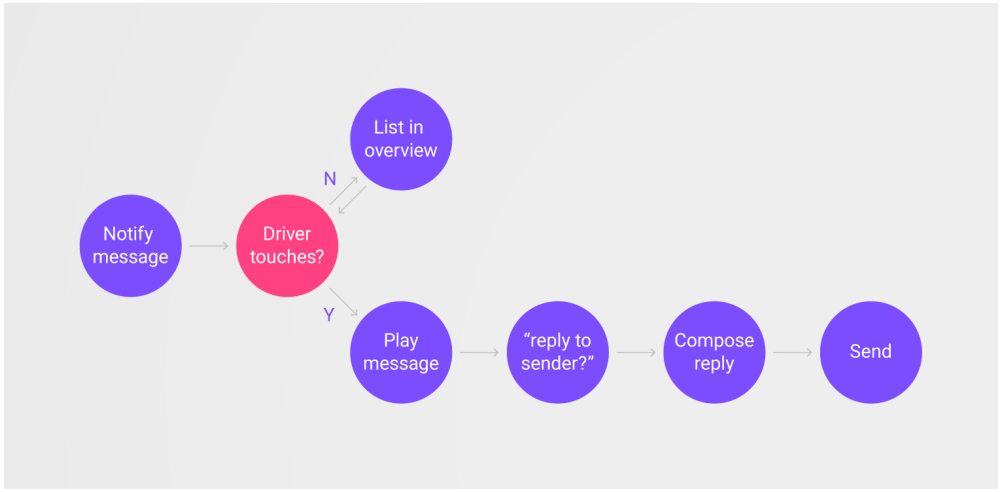


Figure 2.7. Android Auto conversational flow

2.1.6.4 Conclusion

It seems to be a good sign that even Google Inc. is interested in this area and performs such a research. Every Android application can be designed for Android Auto and use it's simplified user interface, allowing the developer to focus on other issues than in-car user interaction. However, the functionality is currently very limited. Hopefully there will be further progress soon.

2.2 Platforms

The chosen platform heavily influences market share an application can reach. Therefore, only platforms with a solid market share are considered. Another criteria is a simplicity of development, which influences the time and effort put into an application before it can be released. This is especially important for finding out the sale potential of an application quickly.

Following the first rule mentioned above and based on tablet sales in past years [3], the only viable options for an application are platforms Android, iOS and Windows, since other platforms did not score high results.

2.2.1 Android

In 2013, the Android platform had 61.9 % market share [3], making it the most used platform in the world. Targeting the Android platform creates a large base of potential customers.

The development language for Android is Java, commonly known object-oriented programming language with a solid developer base. Therefore it is easy to find developers as well as answers to variety of programming related issues, making the development easier.

2.2.2 iOS

With 36 % market share in 2013 [3], iOS is the second most popular tablet platform. Considering a typical iOS user who is willing to pay for quality, iOS could be a good choice for an application in context of potential customers.

However, the development language called Swift is somewhat new in the world, which brings a lot of possible difficulties. Searching for answers while developing in this technology might prove to be too troublesome.

2.2.3 Windows

With only 2.1 % market share in 2013 [3], the Windows platform does not seem to be a valid choice for given criteria. Having thirty times lower customer base than Android, it goes into the nice-to-have section when it comes to multi-platform applications.

2.2.4 Conclusion

Fulfilling the requirements for customer base as well as simplicity of development, the Android platform seems to be the best choice available at the time of writing this thesis. As such, it will be analyzed more thoroughly later in this chapter (2.3).

2.3 Android platform

There are some platform aspects to be considered when developing for Android-based tablet device. First is a problem which is present with most of the tablet devices today – the device performance. While the hardware is continuously evolving, one must consider older devices as well as growing requirements for graphical presentation. The second possible issue is the Android architecture, which influences the inner communication throughout an application. [4]

2.3.1 Performance

Nearly with every new version of Android, new presentation effects are prepared for developers to use. While it is not mandatory, it is still advised to hold the platform standards as the market demands it. An application must have a good look and feel in order to attract attention. This must be considered when creating an application, because the environment demands fluent responses.

2.3.2 Architecture

The main building block in Android application is an Activity. The Activity is an independent component of an application, a hybrid between a controller and a view in MVC architecture. It contains a single screen (which contains a single layout), it has it's own independent data. An application usually consists of multiple loosely coupled Activities. Those Activities are held in an Activity stack, where they are preserved to be used later without need to create them all over again. However, if the system needs memory, it clears the stack from the bottom (least recently used Activities).

Serving for communication between Activities there are so called Intents. An Intent is a main concept of communication between two components. A component can be for example an Activity or a Service. Intent can contain simple data such as primitive or serialized data.

Presentation is handled using XML layout descriptors, which contain information about View objects and their parameters. This feature allows to separate the actual code from a layout creation, which could help the front-end designers create a GUI without having to understand the Java language or the Android API.

XML is not used just for layouts. Most of the resources are defined using XML descriptors. There are strings, values, dimensions and even certain graphical objects defined using XML. These resources are accessible from code using a static class `R`, which is created during build time by most build system automatically.

As a relatively new concept, a new element called Fragment was created. It is similar to the Activity, however it is not a mandatory component. It can be used as a controller for a certain functionality area. It's advantage is that a developer can create separate Fragments with separate area of concern and display one or many of these based on the screen size. The typical use-case example can be a list of items and a detail of a selected item. On small screens two Activities, containing a single Fragment each, will be needed, while on larger screens one Activity can contain both Fragments.

2.3.3 Material design

Material design is a visual language created by Google Inc. It is inspired by a real material, it's behavior in motion, the effects of light and dark, the rules of physics. It also describes colors, which should please the eye and create meaning and focus. The usage of this language is described in Material design guidelines [5].

Every material has certain properties. Every element is considered to be a real object with it's depth and it's position in the 3D space. This causes a lighting to create shadows based on an elevation, to distinguish between layers, to show distance of elements.

Material design guidelines also describe animations considering the mass and weight of animated objects, responses to interaction, also the physical laws of motion in acceleration and deceleration, in jumping up and falling down.

Apart from these general descriptions of material and motion, it also states certain rules and exact measurements. For example, lists should be scrollable vertically and fluently. Buttons, icons, fonts and all the other elements should be of certain sizes.

Views should choose from a certain set of layouts. Layouts for lists, cards, buttons and so on are all specified in the guidelines.

2.4 GUI

2.4.1 Basic principles

As the main tool of communication between an application and it's user, user interface must follow one basic rule – the user goes first. UI is about the user, he must have a good feeling when using the application. He must understand what to do and how to do it. Therefore there are four rules that a proper UI must obey [6]:

- **Clear** – it must be obvious what and where the user can control,
- **effective** – minimizing required user interactions for a certain (requested) thing to happen,
- **foolproof** - avoiding errors before they happen,
- **pleasant** - no stress when working with the UI, pleasant colors, a contrast, a good readability.

Those rules might seem too shallow. That is why there are certain subgoals which are more specific, helping to achieve the main four goals. Those subgoals are the following seven:

- **Minimality** – removing everything that can be removed without losing a requested information value,
- **responsiveness** – giving the user a proper feedback so that he knows something is happening,
- **forgiveness** – letting the user make mistakes, allowing him to fix them (for example undo button or prompt message),
- **familiarity** – using familiar, commonly used metaphors, icons, procedures,
- **consistency** – using a consistent visual and interaction language,
- **integration** – using platform specific elements and rules
- **simplicity** - allowing the user to quickly learn how to use the UI

2.4.2 UI in a car environment

When developing a user interface for a car, certain responsibility is added. The need of safety while using the UI becomes a main priority. Because of that, some aspects are more important than others [7]. The most important aspects are described later in this section.

2.4.2.1 Minimality

For minimizing the cognitive load, there must be as little information as possible at a certain time. A user must see what he wants to see on first sight without seeking the answer for too long. When minimizing the information displayed, there is no confusion, which minimizes the glance time.

2.4.2.2 Consistence

Supporting usability and shortness of learning curve, consistence allows a user to remember one procedure and apply it successfully in different sections of UI. It allows user to learn things just once.

2.4.2.3 Readability

Good readability is one of the conditions for an application to be pleasant to use. In case of a car environment, however, the readability of information is not just pleasant but also critical. Allowing the user to see the information he needs to see in the shortest time possible is fatal when it comes to driving. Therefore the text font has to be large enough for every driver to recognize it.

2.4.2.4 Controls

When it comes to controlling an application in an environment such as car, it is required to consider certain aspects that are not present in other environments. The moving car prevents user from being precise when it comes to touch. Therefore controls must be large enough to be reliably reachable.

2.4.2.5 Colors

While in other environments a user can usually control a device brightness, it is not as easy task while driving. Furthermore, blinding the user with too much light might be fatal. Therefore proper colors must be used. For example, dominance of white color might be visible well in the daylight, but might blind the user at the night time. Also, proper color contrast must be considered for good a visibility and readability.

2.4.2.6 Responsiveness

Responsiveness is an important factor when it comes to pleasure of using an application, but when it comes to using it in a car, it becomes extremely important for safety as well. When an application is responsive, it's user does not have to check the screen for progress so often or worse, wait for the progress looking at it continuously.

2.4.3 Development process

The GUI development process is a part of a bigger process – the User Interface development process. As the decision has already been made to create a graphical user interface, development methods for other types of user interface will not be described.

The basic procedure of creating a UI design consists of multiple steps [8]. Fulfilling requirements for each step properly should guarantee a proper outcome. The UI design steps are as follows:

- Assignment and understanding,
- research,
- behaviour specification,
- basic vision (mockup),
- detailed design of the looks,
- implementation,
- usability testing,
- evaluation,
- final implementation.

The process can also be divided into fewer phases, from which each contains multiple tasks. The list mentioned above is divided into these phases, so that these phases are certain sets of steps that can be iterated over and over for the best result possible. These phases are the lo-fi phase, the hi-fi phase and the final phase.

2.4.3.1 Lo-fi phase

- Basic product statement,
- needs assessment,
- use case brainstorming,
- task list definition,
- task analysis,
- prototyping,
- evaluation,
  - cognitive walk-through,
  - collaborative critiquing,
  - heuristic evaluation,
- re-design.

The product statement should state what the product is, what it does and who is it for. This ensures that the developer knows what is he actually trying to achieve and why. Also, it briefly describes a target user group.

The needs assessment is a systematic process for determining and addressing the needs. It is not necessary to perform unless the goal or the user group is unknown. It also involves a user research.

The use-case brainstorming is used for finding the use-cases of the application. In other words, the outcome should be a set of use-cases, of things user can do with the application. It also gives an idea about functionality, not just the UI.

Also created using the brainstorming method, the task list is defined. Is is based on the use cases created earlier. A task is a procedure that a user has to do with the application when achieving a single goal. After defining the tasks they are also analyzed.

After the analysis is completed, a prototype can be created. Prototypes are the early drafts of the GUI, they serve as something to work on, a physical representation of the current GUI design direction. They are usually done with a paper and pencil or a professional prototyping software, but they lack functionality. Prototypes in this phase can also be called mock-ups, wire-frames or lo-fi prototypes.

The prototype is then evaluated using several evaluation processes. A cognitive walk through, a collaborative critiquing and a heuristic evaluation should be done. The cognitive walk through is an attempt of an expert to act as a user and walk through the application. The collaborative critiquing is a session where a group of people tries to find problems. And the heuristic evaluation is about fulfilling the heuristic rules and should be taken into consideration during the whole design process.

2.4.3.2 Hi-fi phase

The hi-fi phase assumes the completion of the lo-fi phase and takes the prototype further into reality. The hi-fi prototype adds functionality. It is an illusion of the final visual and interaction design. It also already runs on the target platform and follows it's look&feel. While it should mostly work like the final application, the actual application logic does not have to be implemented yet. Also, only the main parts of the application UI are prototyped.

Also in the hi-fi phase, an iterative evaluation process is present. The prototype is implemented, tested, evaluated and then optionally redesigned over and over again. Usually the final design is used in the application itself, which, however, does not have to be the best way.

The evaluation in this phase is already done with users, but also testing without users is present to check the direction correctness (heuristic testing etc.). Usability testing is performed and depends on the application itself. As mentioned in [9], five users should be enough to test an application, as an additional tester does not add as much precision.

2.5 Business requirements

Business requirements describe the application from the business view. They do not describe exact details, neither they describe easily measurable requirements. It is a set of *what* should be achieved with the developed software. Simple business requirements follow:

- Minimizing the cognitive load,
- simple and consistent user interface,
- fast and responsive,
- usability before attractivity,
- performance before delightful details,
- rich, extensible functionality:
  - display information about car,
  - display common information (time, battery, weather, ...),
  - etc.

2.6 Use-cases

Use-cases describe objectives users want to achieve with a system. They describe not only the UI, but also the functionality. They are usually named with a verb and optionally a noun. The name should be descriptive enough in order to give a proper idea about the specific goal.

The use-case list is based on the analysis so far. It is inspired by the research on existing applications in section 2.1 and the general idea of the application mentioned in section 1.1.

- Display information from the car,
- display device information (time, battery, etc.),
- display icons for the information to be easily recognizable,
- allow customization of displayed information,
- provide safe controls,
- provide easy access to other applications,
- support different themes (light, dark).

2.7 Task list

The tasks are based on the use-cases, they are more exact subtasks of the use-case scenarios. A single use-case scenario can be done by performing one or multiple tasks from the task list. They describe the system from the user perspective.

- Display a single piece of information,

- add a display:
  - select a position for a new display,
  - select a desired information or
  - select a desired action:
    - select a simple action or
    - select an external application,
- remove a display,
- invoke an external application:
  - add external application display,
  - invoke an external application on touch,
- change a theme,
- create a group of displays,
- move to another group of displays,
- move back from another group of displays,
- go back.

2.8 Development and support tools

A fluent development process cannot be done without proper development and support tools. These tools provide additional safety of code (prevention from loss of code), additional protection layer against bugs (automatic tests), help with implementation (code-completion, linking etc.) and more.

2.8.1 Development environment

Even though using text editor and command line is an option, for speed of development only Integrated Development Environments are considered. In the time of writing this text, there were two possibilities for Android development – Eclipse and Android Studio.

2.8.1.1 Eclipse

Based on research by Oliver White [10], the Eclipse IDE[1]) is the most often used Java IDE. That is probably the reason why Google Inc. suggested this IDE for Android development in early phase. However, Eclipse has lost Android development support in late 2014 [11].

2.8.1.2 Android Studio

Released in 2014, Android Studio[2]) became the main platform for Android development. It is based on IntelliJ IDEA IDE and supported by Google Inc. For that reason, it is an obvious choice for new applications to be developed in Android Studio.

2.8.2 Version control system

Versioning is very important part of a software development process. Being able to go back to working version or to develop new features while the main version is still working, is priceless. Currently there are three main VCS worth considering (based on an article[3])).

[1]) `https://eclipse.org/`
[2]) available at `http://developer.android.com/tools/studio/index.html`
[3]) `http://www.sitepoint.com/version-control-software-2014-what-options/`

2.8.2.1 Subversion

Subversion[1]) has a single repository where all the data are stored. This simplifies the backup of a whole project, because all the data are located in one place. This, however, creates possible threat of data loss when the central repository gets corrupted without backup.

Because of the central repository, Subversion allows read and write access controls for every single location and have them enforced across the entire project, which can come in handy when developing in a large community, but it is usually not required when developing in a small team.

2.8.2.2 Mercurial

Mercurial[2]) is a distributed source control management tool, it focuses on performance and scalability. It also gives a high priority to keeping history as it is fairly difficult to alter historic data inputs. Also several GUI tools exist for the Mercurial version control system.

2.8.2.3 Git

Git[3]) is a widely used version control system. It uses a concept of distributed repositories. Those repositories contain immutable objects identified by the hash of their content. This makes the history very safe, as there is no way of changing a commit. However a commit can be replaced with another commit and the development story can be altered, making it well-arranged for the potential needs of future analysis. It also supports branching and staging.

2.8.2.4 Conclusion

As the author has a long experience with Git and it is also one of the most widely used VCS, it will be used for version controlling. It is continuously being improved and enhanced, it supports various additional functionality (for example by using hooks) and is overall widely supported, therefore Git seems to be the best choice.

2.8.3 Test driven development

Being one of the main development approaches in the last decade, test driven development helps to develop an application quickly and fluently. The main idea of TDD is to create automatic tests before the actual application code. While this enforces a developer to think twice when creating tests, which makes him think about what he actually wants to achieve, it also helps against random errors in code. Having the application tested with every build also supports continuous integration, which is described later in this text.

The most usual tool in Java is the JUnit framework[4]). It allows writing simple repeatable tests and is an instance of the xUnit architecture. Also API for testing from the Android support libraries will be used, as the application environment is specific and for proper testing, an access to certain resources and objects is necessary.

2.8.4 Continuous integration

"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading

[1]) `https://subversion.apache.org/`
[2]) `https://www.mercurial-scm.org/`
[3]) `https://git-scm.com/`
[4]) `http://junit.org/`

to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." [12]

Continuous integration supports rapid application development while giving the much needed feedback, so that a developer can see and adjust the direction, which the application development takes.

For simple CI integration in practice, there are several online services. For this particular application development, Travis[1]) system was chosen. While offering usual CI functionality, it also integrates easily with GitHub[2]) and Gradle[3]) build system.

2.8.5 Test evaluation

As described in a section about usability testing (5.3), testing with users on a car simulator will be performed. There is an eye-tracking system on the simulator as well, therefore there will be huge amount of data gathered from both the simulator and the eye-tracker. A proper software must be used to analyze these data and to present them. Also it has to be purchasable cheaply or for free. A few options will be described here: Wolfram Mathematica, Matlab and R.

2.8.5.1 Wolfram Mathematica

Wolfram Mathematica[4]) is a professional software for technical computing. It covers all areas of technical computing from mathematics, physics and so on. It is based on Wolfram Language, which has strong algorithmic power as well as wide range of capabilities. In Wolfram Mathematica pretty much every technical computation can be done in the most reasonable time. It has one of the most advanced help systems available with hundreds of thousands examples.

However, all of this does not come for free. The Wolfram Mathematica software is very costly and for purpose of this thesis, a minimum functionality would be used. As the results of the testing will probably be presented later in a research paper, a professional license would have to be used.

2.8.5.2 Matlab

Matlab[5]) is another software known for it's technical computing capabilities. It is capable of numeric computation, data analysis and visualisation, programming and algorithm development and even application development and deployment. It is a high-level language and an interactive environment on it's own.

The disadvantage is the price of Matlab. Even a student license is very costly and a professional license would be needed in the future for the research paper release. It might be an option in the future, however it is not a viable option now.

2.8.5.3 R

R[6]) is a free software environment focused on statistical computing. It does not have as wide area of use as the software mentioned above, but it is still capable of analyzing large sets of data, which means it might just be enough for the test results to be analyzed properly. Should it not be enough, a different approach has to be taken.

[1]) available at `https://travis-ci.org/`
[2]) `https://github.com/`
[3]) `http://gradle.org/`
[4]) `https://www.wolfram.com/mathematica/`
[5]) `http://www.mathworks.com/products/matlab/`
[6]) `https://www.r-project.org/`

Even though it is completely free, the use and functionality it not limited. Also it fits the cheep purchase requirement making it a viable choice. Therefore, R will be used for test results analysis and evaluation.

2.9 On-Board Diagnostics

On-Board Diagnostics stands for a self-diagnostic equipment requirements for automotive vehicles. The modern implementations offer standardized communication port to provide real-time data as well as diagnostic trouble codes.

Currently there are two versions of OBD. The first one (OBD I) provides only diagnostic trouble codes. The second one (OBD II) adds real-time vehicle data. The third version (OBD III) is currently being developed. It should support so called "remote OBD", which would broadcast the data to other vehicles, which could prevent collisions by warning the drivers when something bad happens. [13]

2.9.1 Connection

To connect to the OBD II, an OBD-II Blue-tooth Adapter (often also referred to as Dongle) has to be connected to the car. This dongle then enables creating a bluetooth connection and via that connection it provides a communication channel. Some dongles also support Wi-Fi.

2.9.2 API

The OBD communication protocol supports certain modes and allows reading information on certain PIDs. The mode is used to set a mode of an OBD adapter. As stated in the latest OBD-II standard SAE J1979[1]) there are 10 modes available. Based on the current mode the adapter behaves differently. The modes are:

- Show current data,
- show freeze frame data,
- show stored Diagnostic Trouble Codes,
- clear Diagnostic Trouble Codes and stored values,
- test results, oxygen sensor monitoring,
- test results, other component/system monitoring,
- show pending Diagnostic Trouble Codes,
- control operation of on-board component/system,
- request vehicle information,
- permanent Diagnostic Trouble Codes.

Vehicle manufacturers are not required to implement all the modes as well as they are not required to implement all the PIDs. However, there is a special request available to receive the list of supported PIDs. This is done via sequence of bits stating 1 for supported and 0 for not supported PIDs.

For every mode, there are different PIDs available. The actual vehicle data (such as speed, fuel, engine load, temperatures, etc.) are available via modes 1 and 2. The Diagnostic Trouble Codes are available via modes 3, 7 and 9. For information retrieval, a hexadecimal number is sent to the adapter containing the PID number according to the data requested. The full table of PID codes is available at the OBD-II wikipedia page[2]).

[1]) `http://standards.sae.org/j1979_201408/`
[2]) `https://en.wikipedia.org/wiki/OBD-II_PIDs`

As the OBD is widely used in software, libraries used for accessing the data are available. Such library can save a lot of work required to implement the communication protocol, to solve all the safety issues as well as to cover different vehicles. For such task, an OBD-II Java API library[1]) will be used [14].

2.9.3 Data

The OBD-II covers nearly all the driving data one can imagine a vehicle knows. It goes from the usual data, such as vehicle speed or engine RPM, to less usual such as multiple oxygen sensors. As mentioned earlier, the full table of the data provided is available at the OBD-II wikipedia page. All of these data should be available (via the application) to the driver if he demands it and if the car supports it

[1]) available from `https://github.com/pires/obd-java-api`

Chapter 3
Design

This chapter is about the design process. The first section is about the application architecture, it's requirements and the platform conditions. It is followed by a thorough description of the GUI design process divided into four phases as parts of the iterative process.

3.1 Application architecture

Designing a proper application architecture is one of the main and most challenging tasks in the development process. Changing the architecture in the future proves to be one of the most expensive changes as for man-hours [15]. Application architecture influences a data flow, a communication between components and overall application performance, as well as an extensibility and a possibility to change or add features in the future. While the Android application architecture enforces certain components and platform features to be used, there is still a space for diversity.

3.1.1 Platform limitations

As mentioned in Android platform analysis in section 2.3 on page 10, the typical Android application consists of multiple Activities, which communicate with each other using Intents. While this approach supports the loosely coupled concept, it makes certain inter-cooperations rather difficult. Sharing an object between activities usually means serializing the object or saving it to the database, which leads to deserializing or loading from the database later. When striving for excellent performance, this can emerge into a serious problem. As the Android platform does not allow database IO operations on the main presentation thread, it requires background thread with callbacks to the main one and a screen revalidation when such callbacks occur. It is critical to avoid such delays as much as possible when comes to car environment where fast reactions are required.

3.1.2 Extensibility

With the current rapid application development there is a need to be able to adjust an application based on market requirements. While creating a new application with every new feature is a possibility, it is certainly better to be able to add new features to the old application so that it actually never becomes old. Extensibility is one of the main requirements for many reasons. When it comes to the application developed in this thesis, new features are planned to be added based on a user feedback. Therefore the architecture must be prepared to be easily extensible.

The main approach to achieve a proper extensibility should be to write a clean code, which can prove to be a good idea when considering nearly every part of an implementation process. Also the modularity concept is very useful when it comes to extensibility and it will be discussed in section 3.1.3.

3.1.3 Modularity

3.1.3.1 Note for Android platform limitations

The first considered approach was to create requirements on modules, such as manifest file as a descriptor and an implementation file with source codes and resources, so that the modules could be loaded dynamically and the extensions could be customizable. Then a user-base could develop modules on their own and add them freely into the application once they meet the requirements.

However, the Android concept with XML layouts does not allow their inflating during runtime. Because it is a performance-expensive operation, it pre-processes a XML file when building the application, as quoted below.

"For performance reasons, view inflation relies heavily on pre-processing of XML files that is done at build time. Therefore, it is not currently possible to use LayoutInflater with an XmlPullParser over a plain XML file at runtime." [1])

3.1.3.2 Overview

Modularity concept allows application to contain certain modules, each offering a certain functionality based on some predefined requirements. The modularity will be supported via extending predefined classes and implementing required functionality (such as action on update). This limits the modularity, however it is not suitable to do it differently at the moment given the restrains mentioned above.

3.1.4 Adaptability

Because the space on the screen is limited and also unknown in advance (multiple devices have varying screen sizes) and every user might want to see a different kind of information, he must be able to modify the layout, to choose the information he wants to see. The application must be adaptable to user's needs and requirements, so that he can control the application fluently and spend as little time as possible seeking the requested information.

For that reason there will be module containers which can contain multiple modules. A user then selects the module for each container and selects a single container to be displayed at a time. This allows to build custom module sets for greater adaptability.

3.1.5 Architecture

A multilayer architecture will be used for the application. With the Android architecture requirements, a modified MVC architecture will be used, where an Activity works as a controller and partially as a view. The activity will contain a set of modules, which are single-purpose elements based on the predefined classes (as mentioned in section 3.1.3). Those modules will communicate via interface, which will be implemented by the activity. Also an event-driven approach will be used in communication, especially with timed events.

[1]) from documentation available at `http://developer.android.com/reference/android/view/LayoutInflater.html`

3.2 GUI

Given the car environment, designing a proper graphical user interface is crucial for an in-car application. Not only it has to look good, it also has to consider safety issues such as minimizing the cognitive load and required glance time to control the application or to read displayed information. To achieve that the GUI should follow the principles mentioned in section 2.4.

3.2.1 Phase one

In the early phase of the design process, the main idea was to display a single piece of information at a time. Given that, a certain concept was created with a single application panel per screen, which would be a scrollable list. Swiping left or right would change the focus to another application panel. Part of the previous and following application panel would be seen as shown in figure 3.1.

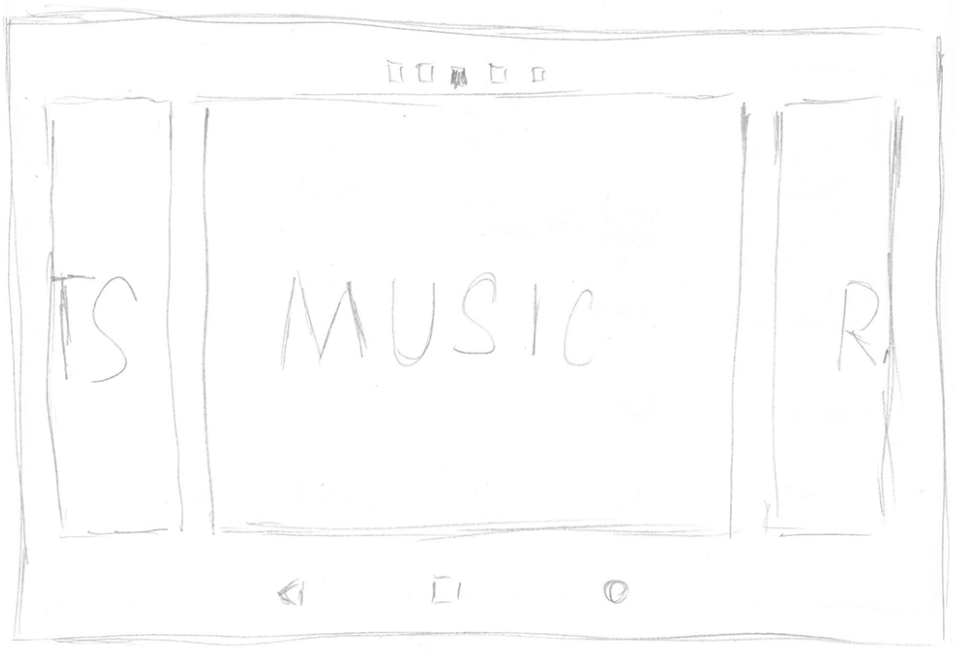


Figure 3.1. GUI draft #1 with multiple panels

This concept was recreated into a similar concept with a difference in sizes of a previous and a following application panel. Those panels would be moved into the background which would make them smaller, as shown in figure 3.2, however, more of these panels could be visible letting the user know more about the actual structure. Also, it presents combination of a name and an icon for easier recognizability.

Figure 3.2. GUI draft #2 with the next and previous panels pushed into the background

For both drafts the following applies. The swipe action would invoke text-to-speech action telling user the name of a selected panel. This could lower the need to look at the application screen while driving. Also, all the panels would have different colors making them recognizable on the first sight. The touch on an application panel would invoke the related application. This could be a music player, a map, etc. Examples of a music player sub-application are shown in figures D.3 and D.4 in the appendix B on page 63.

3.2.1.1 Advantages

- **Readability** – given a single panel per screen with only a name and an icon in it, the font can be large enough to be properly readable.
- **Colors** – colors can distinguish separate applications panels making them easily recognizable once the user learns the colors for each application.

3.2.1.2 Disadvantages

- **Consistence** – however is the main screen consistent, the invoked sub-applications are not. The concept does not force them to be, neither it gives a clue about how they should look.
- **Limited** – the main screen has a limited functionality (near to none) while the layout of sub-applications would have to be created independently every time a new feature is implemented. This also limits easy extensibility as creating a proper GUI is not a simple task with the given constraints.

3.2.2 Phase two

The next step was to fix the problems mentioned above. Being inspired by the reviewed applications (2.1) one attempt ended with the concept shown in figure 3.3. It presents

a vertical list of applications displayed in a column on the right side of the screen instead of a horizontal list over the whole screen. The main area contains the usual car data such as speed, rpm and consumption.

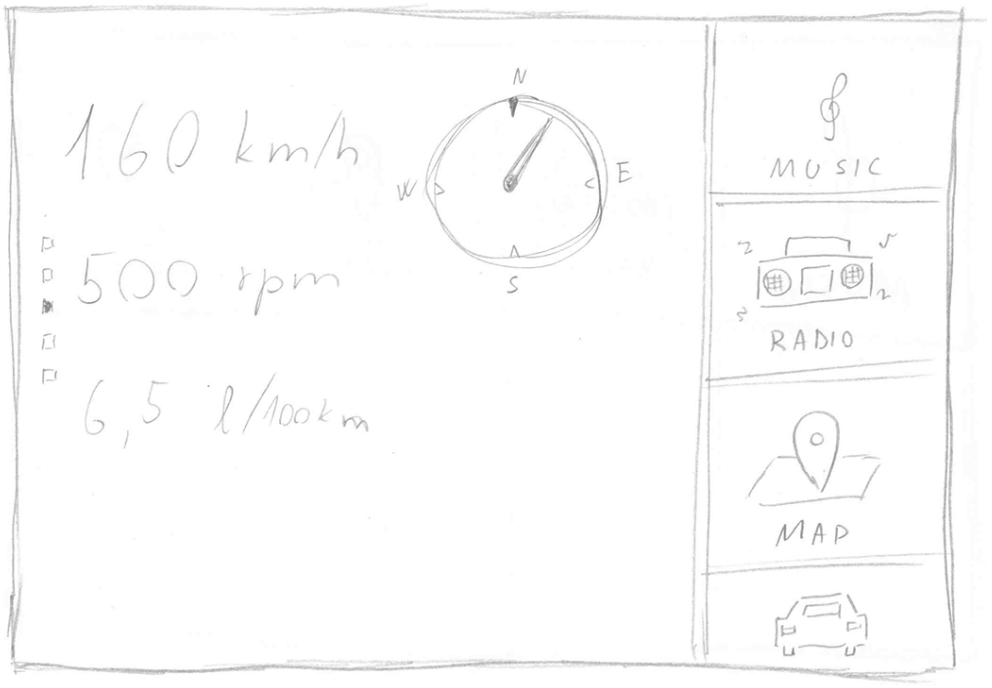


Figure 3.3. GUI draft #3 with the main section in the center and the menu in the right panel

The second image 3.4 shows possibility of inserting a sub-application screen between an application list and car data, for example a navigation. Also, it presents the concept of micro-controls in an application list. It would allow a single control button to be displayed on an application panel such as pausing a song or muting the music player.

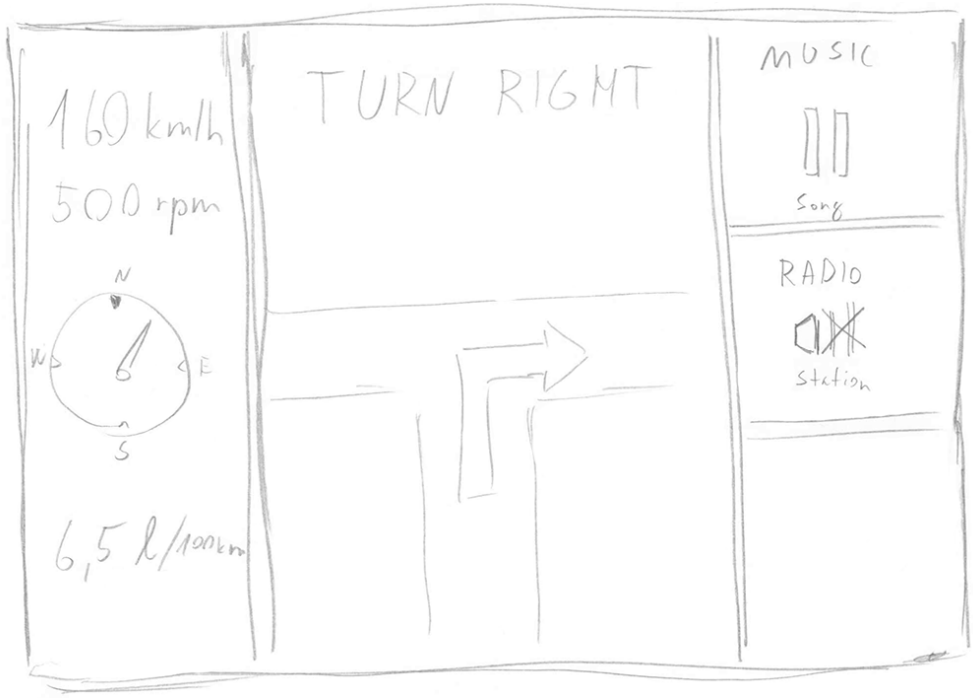


Figure 3.4. GUI draft#4 with additional application screen in the center, pushing the car data to the left panel

3.2.2.1 Advantages

- **Controls** – the concept shows improvement in consistent functionality for displayed application panels, which eases the control.

3.2.2.2 Disadvantages

- **Minimality** – the amount of data grows and it appears to be too much. There are different kinds of data displayed at the same time.
- **Consistency** – the vertical application list is consistent, however, the central panel is still suffering from the lack of consistency, as every sub-application can have a different layout.

3.2.3 Phase three

The next step towards a consistency and a space usage was to create a grid. This grid would be adjustable based on a screen size, displaying the proper amount of application panels for a given device. As shown in figure 3.5, it is just an extension of a previously shown vertical list making it vertical and horizontal – two dimensional.

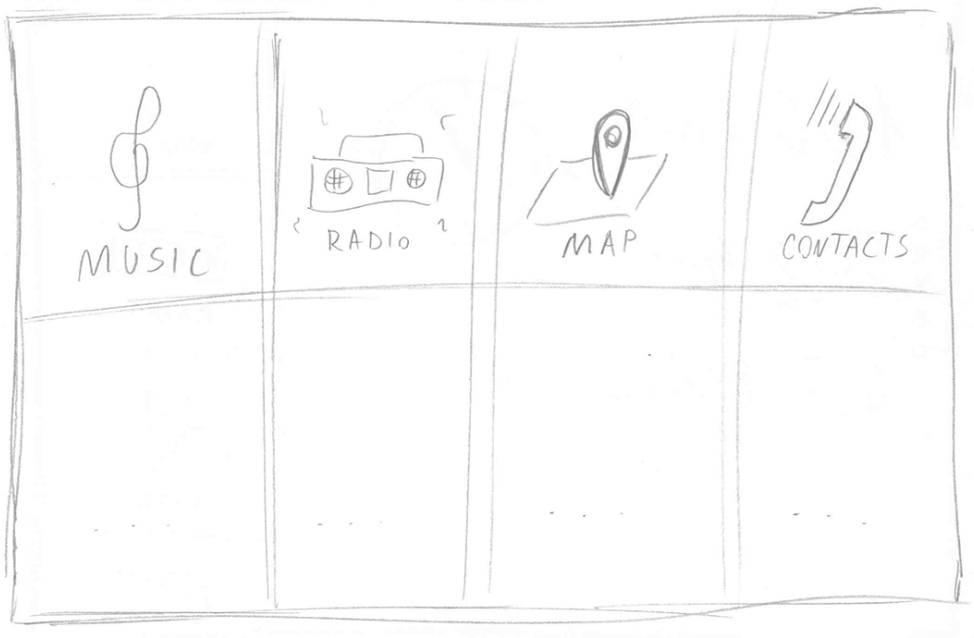


Figure 3.5. GUI draft#5 with a grid of panels

Adding functionality to this grid, a new card concept emerged. It consists of cards, which provide additional information as well as control elements (as shown in figure 3.6). They would be active demo-versions of the full applications, which would then be invoked by a touch to the upper area of the card, as shown in figure D.6 in the appendix B.

Figure 3.6. GUI draft#6 with a grid of panels, which display information and offer basic functionality

3.2.3.1 Advantages

- **Accessible functionality** – the concept shows basic functionality available without a need of invoking the full application. This allows a user to remain in the main screen in most cases.

3.2.3.2 Disadvantages

- **Controls** – in order to fit in the card area, the controls might prove to be too small, which makes it difficult to touch them
- **Readability** – in order to fit in the card area, the text might have to be too small, which makes it difficult to be read

3.2.4 The final design

Figure 3.7. GUI draft#7 with a grid of simple panels

Because every single one of the previously mentioned designs had at least one critical disadvantage, a new approach had to be taken. Because of consistence, every element must be specified. But considering the need for simplicity, there must be very limited amount of these elements.

Given the requirements for both consistence and simplicity as well as extensible functionality, the elements are divided into two groups: these, that display information and these, that control the application. The simplest way appeared to be the following: one element serves as a display element, which displays one and only one kind of information, and the second element serves as a control button, which allows user to perform a single action. Every kind of functionality appears to be achievable by these elements or by sets of these elements.

Also, for improved adaptability a hierarchy model was considered, which makes it possible to create independent sets of functionality using a hierarchical model, which supports the consistence and simplicity by repeating the same pattern in distinct areas.

As shown in figure 3.8, the display panel consists of a name, an icon, a value and a unit. The control button is more simple, it consists of a name and an icon. An icon serves as a checkpoint for eyes to seek out the requested information quickly.

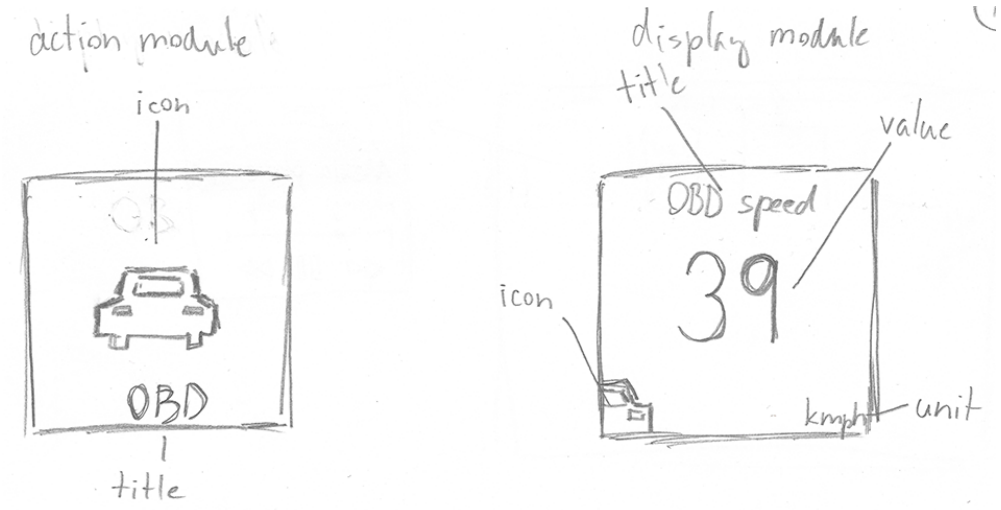


Figure 3.8. GUI draft#8 presenting the action panel (left) and the display panel (right)

The idea is to have several screens containing several application panels (where amount of panels is based on screen size) with changing the screen by swiping left or right. As mentioned in section 2.3.3 on page 11, Android suggests using vertically scrollable lists when presenting large sets of data. This can be suitable in most cases, however, in a car a user can easily swipe too heavily and scroll elsewhere and getting back to original place can put an unnecessary load on the driver's attention. Therefore there have to be separate pages, where one swipe changes a page by one.

As for the colors, a proper contrast has to be present for a good readability. As mentioned in section 2.4.2 on page 12, two modes should be present. While light mode offers good readability even in a direct sunlight, it blinds the driver during the night time as it emits too much light. Therefore it is a good idea to implement a dark mode as well for a night time usage. For the highest contrast possible, white on black or black on white are the best options.

3.2.4.1 Advantages

- **Minimality** – only a single value is displayed per each panel,
- **familiarity** – using familiar (platform specific) icons should ease the information seeking process,
- **consistency** – consistent hierarchical model with only two types of elements,
- **integration** – using platform specific icons and specifics is a part of realization process,
- **simplicity** – again, there are only two kinds of elements, which is simple enough,
- **readability** – because of the good contrast the text will be easily recognizable and readable.

Chapter 4
Implementation

This chapter is about the whole implementation process – the process of implementing the application with it's logic and GUI. It starts with a section about preparation, which provides an insight into the preparation of implementation environment and tools. It is then followed by the description of tablet specific part and then the actual core of the application. At last the final GUI is described.

4.1 Preparation

While this thesis focuses on creating a tablet application, it's functionality could be shared amongst other Android platforms just by reflecting the differences. A possibility to extend application to a mobile or any other platform creates a need to divide a single project into two – a core with shared functionality and a tablet project which focuses on tablet GUI and other specifics.

4.1.1 Environment

As mentioned in analysis section 2.8.1 on page 16, Android Studio IDE is used for code development. As it does not allow importing other project as a library, it requires a shared project to be registered as a module. This module has to be placed in a project sub-folder and it is an Android Studio project of it's own. This is done by adding a new module and selecting a library module.

4.1.2 Versioning

Because of the workaround with a shared library in Android Studio, a special approach has to be taken. As Git supports submodules, a shared library has to be registered as one. This can be achieved by calling command `git submodule add LIBRARY_REPOSITORY_URL` in the root project folder and then adding the library sub-folder into the `.gitignore` file.

4.2 Tablet specific

While the Android API is shared across all Android platforms, it is the device size that is usually different. The GUI has to adapt based on the platform and therefore it's implementation differs. This section describes the tablet-related implementation using a shared library described in section 4.3.

4.2.1 ModulePagerActivity

As mentioned in analysis section 2.3.2 on page 11, an Activity is the basic component of every Android application. In this case, it is the only launch point of the application, it implements so called `IModuleContext` interface described in section 4.3 on page 32,

which controls the interaction with modules (3.1.3). Presentation of screen content is delegated to `ModulePageFragment` (4.2.2).

While usually multiple activities are present in a single application, thanks to consistent hierarchical model a single activity class can be reused for multiple instances with different data. Which means that there is a single activity invoked per one requested set of modules.

4.2.1.1 Improvements

There are some improvements implemented for performance, battery consumption and hardware overloading prevention reasons, one of which is reacting to Activity state by disabling inactive modules. Because of the independent module concept, modules display data on their own and they do not know when the data are requested. Therefore, the activity uses a list of these modules to deactivate them when entering a paused or stopped state.

Also, sometimes it is required to restart the entire application, for example when some global changes need to be performed. As the Android architecture saves latest activities in a stack as mentioned in section 2.3.2 on page 11, it is necessary to clear this stack first, so that the OS does not backtrack to an old Activity. This is done by keeping control over existing activities and ending them one by one. This can also be used, when forcing the application to exit, as there is no proper option to end an application on Android platform from the application developer's view.

4.2.2 ModulePageFragment

As mentioned in section 2.3.2 on page 11 a Fragment can take over part of Activity's functionality. In this case a `ModulePageFragment` handles the presentation of module set using a `ModuleFragmentAdapter` (4.2.3) for obtaining the data and a custom layout `GridLayout` (4.2.4) for displaying them in a grid.

4.2.3 ModuleFragmentAdapter

`ModuleFragmentAdapter` follows the adapter concept, where extension of Android API class `Adapter` is used to cover the access to list of data. This adapter gets a single `ParentModule` (described in section 4.3.1 on page 32) and retrieves it submodules on demand.

4.2.4 GridLayout

Because the Android concept does not expect functionality required by this application, a library class `android.widget.GridLayout`[1]) is not suitable for this situation. As mentioned in section 3.2.4 on page 28, Android suggests the lists to be scrollable vertically. This is a functionality fully supported by `android.widget.GridLayout` but unsuitable for the given use-case. It also makes it difficult to use this library class for the use-case described in section 3.2.4.

For reasons mentioned above a custom layout had to be created. Based on given measurements (of a module tile, a space) it computes amount of modules displayed per page and also their positions. Given the computed positions it then lays out all the provided modules.

[1]) more information available at `http://developer.android.com/reference/android/widget/GridLayout.html`

4.3 Core

The core contains all the functionality, it handles data, logic and also a standardized part of presentation, which consists of predefined single module views. Everything will be described in following text.

4.3.1 Modules

As mentioned in section 3.1.3), there are so called modules, which handle the interaction between the application and it's user offering a single action or information. Together they can create multiple connected sets of functionality with consistent interface.

4.3.1.1 IModule

`IModule` is an interface which covers the basic module functionality. It has to be implemented by every single module in order to achieve proper polymorphism. Using this approach, a tablet implementation can display a set of modules without knowing which module does what.

4.3.1.2 AbstractSimpleModule

`AbstractSimpleModule` is an abstract class which implements most of the `IModule`'s functionality. It handles creating a unique Id for every module, which will be described later in section 4.3.3. It also handles common module events and overrides simple methods to ease the implementation of a new module, which does not need these methods. Every other module extends this class.

4.3.1.3 AbstractParentModule

For consistent hierarchical model, there has to be a module containing other module. This module extends the `AbstractParentModule` class, which covers the module container functionality. Every instance of `ModulePagerActivity` (4.2.1) contains such container and displays it's content as a list of modules.

4.3.1.4 AbstractDisplayModule

Displaying information is one of the most important goals of the developed application. `AbstractDisplayModule` is the base class to be extended by modules displaying information. It handles updating the displayed value on request. Also it supports text-to-speech, as the value is said out loud on touch.

4.3.1.5 AbstractTimedUpdateDisplayModule

`AbstractTimedUpdateDisplayModule` serves as an extension to `AbstractDisplayModule` handling automatic timed updates. It uses advanced generics to offer multiple update modes for extending modules. Such mode states the frequency of calling the `getUpdatedValue` method, which is to be implemented by subclasses. An optimization is implemented for this process, as `getUpdatedValue` can invoke a long-lasting process. The last value is saved for further use by the `updateValue` method, while the `getUpdateValue` method merely updates this last value when it is done.

4.3.1.6 AbstractShortcutModule

As mentioned in section 2.3.2, there is an Intent as a mean of communication. This Intent is able to invoke an Activity, a Service and many other things. It can also invoke Activity of a different application installed on the device, which launches the application. The `AbstractShortcutModule` handles invoking a custom Intent.

4.3.1.7 Other modules

There are several implementations of the modules mentioned above. A few will be shortly described in the following list:

- **SimpleShortcutModule** A mere implementation of `AbstractShortcutModule` class (4.3.1)
- **SimpleParentModule** A mere implementation of `AbstractParentModule` class (4.3.1)
- **AppShortcutModule** An extension to `AbstractShortcutModule` which limits to Intents invoking other installed application, therefore the CarDashboard can server as an application launcher optimized for in-car usage
- **EmtpyModule** An empty module meant to be swapped for a different one, occupying an empty space
- **BackModule** A module handling the back button, which can be pressed to get back to upper parent module (go up in the hierarchy)
- **LightButtonModule** A module created for IoT support, offering a way to turn a given light on or off
- **ObdRpmModule** A module communicating with the OBD and displaying information about current RPM of the vehicle
- **ObdSpeedModule** A module communicating with the OBD and displaying information about current speed of the vehicle

4.3.1.8 IModuleContext

An interface to be implemented by the Activity which should display the modules. It provides functionality to go up or down in module hierarchy, to toggle quick menu for a certain module or to gain access to resources.

4.3.1.9 Quick menu

A quick menu serves as a quick options menu for a simple module. Every module can invoke such quick menu. Usually it contains *cancel*, *edit* and *delete* options. It might contain other options specified by the given module.

4.3.2 Application

This section describes the application logic. While most of the logic is hidden in modules themselves, the communication across application components must be handled elsewhere.

4.3.2.1 UpdateApplication

`android.app.Application` is the main class of Android architecture. There is a single instance of this class per application. For that reason, this class is extended and enhanced with creating and starting timers for timed updates. An instance of this extended class is to be used by the tablet implementation instead of the original `android.app.Application`.

4.3.2.2 FastEventBus

The concept of event bus is to have publishers and subscribers. It is most suitable for timed events, which serve as a signal for modules to update themselves. However, it is not limited just for time updates. Most of the communication can be handled using the event bus. `FastEventBus` offers such functionality while remaining as simple as possible for better performance.

4.3.3 Data

4.3.3.1 Resources

As mentioned in section 2.3.2, resources usually consist of XML files accessible as static properties of an automatically generated class. A new class was created in this project to wrap the access to resources for selected types of data. `StringResource` and `IconResource` classes wrap the access to single sources of resource, meaning that for example a `StringResource` can load the string from resource or from runtime memory. Accessing a resource this way separates the resource user from the data access layer, making the code simpler.

4.3.3.2 Storage

As the application is adjustable by users, the settings need to be preserved. The main data area to be saved is the user-customized hierarchy of modules. Given the hierarchy model of these data and the simplicity of content (module type, name, additional data), a JSON format is used for data persistence. A single JSON file is created containing all the required data for customized user interface. The advantage of the JSON format is the ability to easily persist these settings on a server, given the support of JSON format from web communication protocols.

To save and load these data there is a class `ModuleDAO`. This class separates the access code from the rest of application, making it easy to change the saving format, the type of data saved or even the location of data. It also enables data to be saved in a background thread, so that the saving process does not block the application.

4.3.3.3 Runtime data

For performance reasons to avoid unnecessary loading and object creating, current modules are preserved in a runtime data container called `ModuleSupplier`. This container wraps the access to modules based on their Id, as mentioned in section 4.3.1 on page 32. It also contains a default set of modules when no preserved data are available.

Another advantage of this class it possibility to adapt. Should the modules take too much space in memory, it is easy to switch to data loading model, where modules would be loaded into memory on demand and deleted when they are not currently in use. Doing this change would not affect the rest of the application.

4.3.3.4 Object creation tools

One of the challenging tasks was to save and then load all types of modules. Since there can be custom modules, it is not an easy task without requesting the developer to create DAO for every module. To simplify the module implementation process as much as possible, a set of creation tools was made. Once the module fits in one of these tools, it can be loaded and created without further effort. However, once a new module type is created, for example a type which requires additional data to be saved, a new tool has to be implemented into existing tools.

There are two types of these tools. The tools to create an object based on a loaded module data and the tools to create an object based on selected module when adding a new one into the structure. Both of these tools are based on Java reflection API. For both of these tools a map exists in `ModuleCreationToolsMap` class based on the module class. Every module class has to be registered with a `ModuleCreator` and a `ModuleLoader` before being able to be loaded or added.

The `ModuleLoader` serves to create a new object from previously persisted session. It is an enum of many enum items, each of which implements a method to load from

JSONObject and to save into a JSONObject. Support methods are provided for saving and loading common data, so that only the specifics have to be implemented.

The ModuleCreator serves to create a new object when swapping the EmptyModule (4.3.1). On the first sight it is simpler than ModuleLoader, because most modules can be created just by reflection (creating a new module based on class with default data in it). However, several modules require custom data, such as SimpleShortcutModule, AppShortcutModule, GmapsShortcutModule, etc. Those usually use custom Fragments handling the user data input with callbacks method back to the creator. Those Fragments will be discussed later in section 4.3.4 on page 35.

4.3.4 Fragments

Following are DialogFragments, which is a Fragment in a Dialog window. The advantage of DialogFragment is the ability to adapt. On larger screens it is a Dialog window as a pop-up, on smaller screen it is a full-screen window.

4.3.4.1 ModuleListDialogFragment

This Fragment is used when adding a new module into the structure. It contains a list of available modules in a structure defined by the developer. This structure uses description objects for all the modules. When a module is selected, based on it's class a ModuleCreator is obtained from the ModuleCreationToolsMap (4.3.3). Using this creator a new module object is created and inserted into given position using a callback method to IModuleContext (4.3.1).

4.3.4.2 ApplicationListDialogFragment

When a shortcut to an external application is selected as a new module, an ApplicationListDialogFragment is invoked by the ModuleCreator. Adapter of this Fragment loads all the available applications installed on the device and provides their data to the ApplicationListDialogFragment. The icons and names are displayed in a list for user to select. Selecting a module invokes a callback method, which calls the related method in ModuleCreator.

4.3.4.3 CustomShortcutDialogFragment

When a custom Intent is to be created as a module, a CustomShortcutDialogFragment is invoked. This Fragment offers input for title and Intent content and as usual, invokes a callback method once the data are provided.

4.3.4.4 GmapsShortcutDialogFragment

As an extension to previous, this Fragment allows creating custom Intents particularly for Google Maps. Using Google Maps API[1]), it allows creating shortcut modules, which can invoke the following:

- Display a location on a map,
- launch navigation to a certain location from the current location,
- search current location for a given string, for example hospital, pharmacy or gas station.

4.3.4.5 RenameDialogFragment

RenameDialogFragment has been added to allow user to change the title of a selected module. Because the title is saved as a StringResource (4.3.3), it is source-independent

[1]) https://developers.google.com/maps/

on the outside. Therefore, a `String` can be used instead of a XML resource. The user can then customize his user interface a bit more.

4.3.5 OBD

While modules are as independent as possible, there are cases where a shared functionality is required. Communicating with the OBD protocol is relatively expensive and it would be inefficient to handle it separately. Therefore the communication is centered into a single sub-package, which handles the data retrieval using requests and saving the responses for later use. Every module displaying the OBD data can then ask this package for information and receive it as quickly as possible.

Handling the OBD communication is done by background service which uses the OBD-II Java API library [14] to send requests to the OBD and to receive results. It sends requests based on tasks from a queue, where modules push their requests. This ensures that only currently needed information will be requested, minimizing the load.

4.3.6 Utility classes

There are several utility classes – stateless classes providing certain sets of functionality. As a utility class, every one of them is filled with static methods that help with frequently used operations that do not require to change the outer state.

4.3.6.1 ModuleUtils

A `ModuleUtils` class implements several methods offering a functional approach for lists of modules. Providing a Single Abstract Interface for an action on an `IModule` given as a parameter, it performs this action for each (even recursive) submodule of a given parent module. Also a particular module class or super class can be provided, so that only the related modules are affected.

The simpler method called `forEach` merely iterates over all submodules, performs action on each one of them and if the submodule happens to be also a parent module, it calls itself recursively on this parent submodule as well.

The more complicated method called `forEachDeepCopy` not only iterates recursively over all submodules, but also creates a deep copy of all the parent modules, so that changing their structure does not affect the original. This is helpful when adjusting modules before saving them.

4.3.6.2 ModuleViewUtils

An utility class providing methods to edit ModuleViews described later in section 4.3.7 on page 37. It enables filling them with the data provided by given `IModule`, preparing listeners and quick menus. This covers the access to certain View fields, separating the view layer from the rest.

4.3.6.3 ModuleViewFactory

A `ModuleViewFactory` class enables creating new `ModuleViews` (4.3.7). It offers creation of a simple `ModuleView` or a `ModuleView` in a certain holder `View`. This holder can then wrap a module and adjust it's size based on the platform it is displayed on.

4.3.6.4 TextToSpeechUtils

This class provides simplified text-to-speech functionality. It handles all the settings and preparations and the calling object merely has to provide a string to be read out loud. This class is especially useful given the environment and it is often used with several modules. For example, all the implementations of `AbstractDisplayModule`

(4.3.1) use the text-to-speech functionality when touched, saying the related value. Once the driver memorizes the position of a module, he can easily push it without even looking at it and still receive the information about value.

4.3.7 Views

As the in-car GUI is not the usual type of GUI, it requires several implementations of custom `Views`. Some provide functionality that is not provided by the Android API, some minimize the programming effort when working with modules as well as cover the low-level implementation.

4.3.7.1 AutoResizeTextView

To be able to create custom modules easily, it is necessary to create automatically adjustable elements. Such element is an `AutoResizeTextView`, which automatically resizes the text based on the space provided. This enabled the information to be as large as possible, while still being able to display several types of data (even longer strings). This class is used to help displaying an `AbstractDisplayModule` (4.3.1).

4.3.7.2 ModuleView

`ModuleView` is a main element of presentation layer for a module. It handles accessing the inner data, such as title or icon, as well as access to the related module object. A `ModuleView` is an extension to the `android.widget.RelativeLayout` and uses a XML descriptor, from which it is inflated.

4.3.7.3 ModuleActiveView

`ModuleActiveView` is an extension to `ModuleView`, it uses a different XML layout descriptor and adds a value and unit data display. It is optimized for frequent data update by saving the pointer to the `View` containing the actual value. This avoid the unnecessary load when seeking an element inside a layout.

4.3.7.4 Other views

There are many other views similar to the ones described above or just simple views used for displaying custom lists. Those views wrap access to the inner data presentation elements to separate the layers properly.

4.4 GUI

Implementation of GUI is based on the final design described in section 3.2.4 on page 28. While following the designed concept, also platform specific rules, as mentioned in section 2.3.3 on page 2.3.3, were applied where possible. Following the Material Design was a secondary goal, since the safety of the driver is the most important goal. Therefore compromises had to be made and they will be discussed later in this section. The final result is shown on figure 4.1.

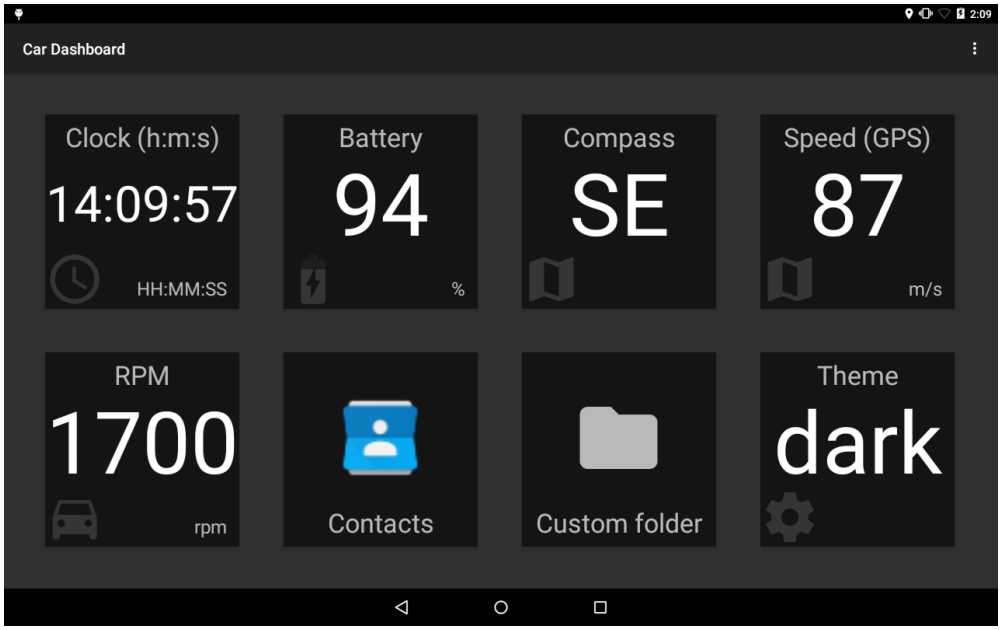


Figure 4.1. Final GUI implementation

4.4.1 Common elements

4.4.1.1 Colors

As described in section 3.2.4, two color modes are present – so called themes. One consists of white font, gray secondary icons and black background, while the other consists of black font, gray secondary icons and white background.

4.4.1.2 Sizes

While Material Design (2.3.3) suggests certain measures, they are not suitable for a car environment, as the control and presentation elements would be too small. Therefore sizes are adjusted and much larger.

4.4.1.3 Effects

Trying to follow Material Design principles (2.3.3), several graphical effects are present in order to increase the overall attractivity. All the modules support proper elevation with shadowing even with increasing the elevation when touch the button. Also, a ripple effect is present when the module is touch, a stronger ripple effect appears on a longer touch. This gives the user proper visual feedback making the application more pleasant.

4.4.1.4 Quick menu

As mentioned in section 4.3.1, a quick menu is a limited set of options for every module. It gets invoked by a long touch on a module. It separates the rectangular module into four rectangular pieces, each containing a button (as shown on figure 4.2).

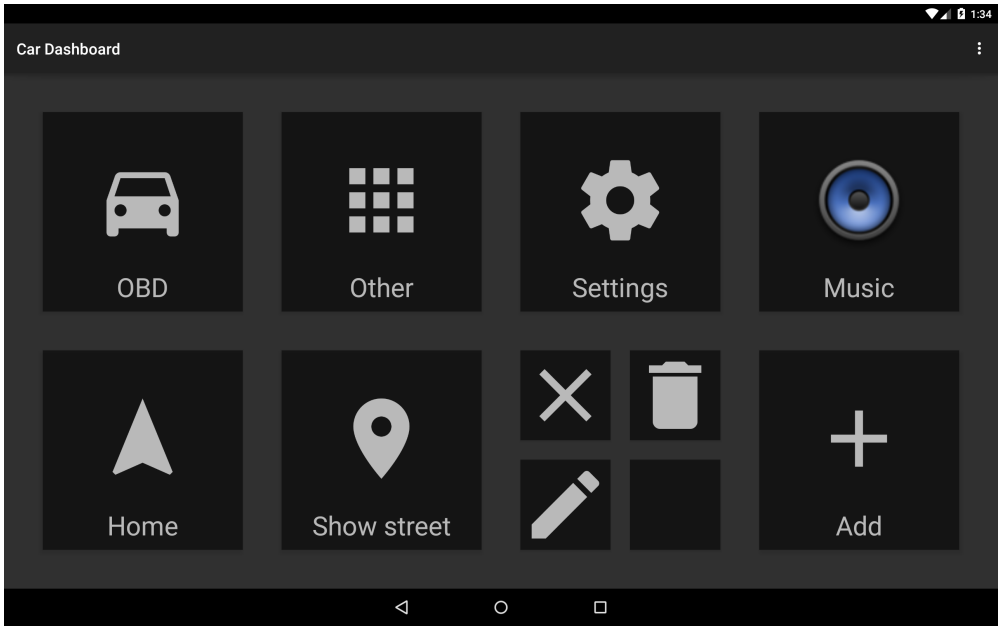


Figure 4.2. Quick menu (2nd row, 3rd column) in the final GUI

4.4.1.5 Icons

Given the platform guidelines (2.3.3), it is easier to find proper icons for various actions. Material Design icons are frequently updated and more icons are added on demand. Should an icon be missing currently, it has a high chance of being created later. It also helps to use familiar icons, so that the user does not have to learn more images and their meanings.

4.4.2 Multiple designs

As mentioned in section 3.2.4, there are two types of modules – an action module and a display module. Both of these modules have separate implementation, while sharing common elements like colors, standard icons, fonts and effects, as mentioned earlier (4.4.1).

4.4.2.1 Action module

An action module is a rectangular element consisting of a large centered icon and a title in the bottom on background of the opposite color to the font color. The icon ensures recognizability, while the title specifies the module identity. The result can be visible on the figure 4.2 (except for the quick menu it is full of action modules).

Such action module is meant to be pressed, not to display information. On press it performs some action, which may or may not give a feedback, based on it's purpose. However, it always gives a visual feedback (4.4.1).

4.4.2.2 Display module

A display module is also a rectangular element, however it consists of a large centered value text, a title on top, a value in the right bottom corner and a small gray icon in the left bottom corner. All of it is on background of the opposite color to the font color. The result can be seen on the figure 4.1 (all modules in the first row are display modules as well as the first and the last module in the second row).

Such display module serves as a source of information for the driver. It can display various data in a well readable form, while preserving an attractive design. It is able to display several types of data from numbers to short strings.

Chapter 5
Testing

This chapter is about testing and all that is related. Starting with a section about testing the code, it describes the use of test driven development via unit tests. Then the heuristic testing is performed as a main mean of testing without users and the results are reviewed. Finally, the testing with users is thoroughly described, as it consists of the usage of a real-world car simulator, it's preparations and an elaborate evaluation of all the gathered data including the eye-tracking system and a simulator log. During the testing with users, A/B testing and Change Lane Test are performed.

5.1 Code

As mentioned in section 2.8.3, the application development aimed to follow the principles of test-driven development approach. This means that unit tests cover part of the application functionality. They do not cover everything, as automating certain functionality (such as GUI) might actually be more time-consuming than manual testing.

5.1.1 Unit testing

At first some difficulties have to be mentioned. As Android application is not usually being developed in an Android environment, the problem of accessing actual Android API emerges. This means that writing proper unit tests gets complicated and it is often easier (when comes to simple applications) to test it manually. This goes against the test-driven development approach, however, amount of time was limited and automated tests for features dependent on Android API will hopefully be implemented later.

Apart from that, unit tests were created for Android API independent classes and methods, for example utility classes. JUnit[1]) framework was used, as it eases the test implementation process, which consists of creating a test class in a package according to the tested class or method, then creating test methods with a `@Test` annotation. Such methods get invoked during the test phase of a build process. They contain assertions, which check input for it's validity (usually expected value and actual value are provided to the assertion). Every such assert command is then evaluated and the failed ones are presented to the developer.

This approach helped discovering a lot of hidden bugs. Even a small change, which seemingly does not influence the tested component, can actually cause errors, which then get caught by the unit test (in the best case scenario). With minimal effort this can save a lot of hard work looking for a flawed area of code.

5.2 Heuristic testing

Heuristic testing is based on following certain set of rules. It is based on somebody's experience. It is a speed process of checking the user interface for common issues. How-

[1]) `http://junit.org`

ever, it does not interpret user's activity. The following evaluation will use the heuristic created by Jakob Nielsen, which consists of ten rules (following later in the text).

For purpose of presentation, in further text there will be signs plus (+) and minus (−) used for interpreting positive (plus) and negative (minus) evaluations. Also, priorities will be stated next to the rule violations in parentheses and they will be limited to words low, medium and high. Also additional notes are present (for future use).

5.2.1 Evaluation

5.2.1.1 Visibility of system state

+ No long-lasting operations present, every long-lasting operations happen in the background without the user knowing,
+ issues might appear with server synchronization, which is not implemented yet.

5.2.1.2 Match between system and reality

+ Icons match their real world models,
  + car informative modules have a car icon,
  + clock module has a clock icon,
  + etc.

5.2.1.3 Minimal responsibility and stress

− Missing confirmation prompt when removing a module (irreversible operation) (high),
− missing confirmation prompt when editing a module (irreversible operation) (high),
− missing proper edit option for shortcut modules (irreversible module addition operation) (high).

5.2.1.4 Match with platform and common standards

+ Material Design present where possible,
− Material Design not present where not suitable:
  − icon size (too large) (small),
  − list controls (does not scroll fluently, but scrolls page by page) (small),
  − navigation is not done by a navigation drawer, but rather a file-system like style (small),
  − measures do not match the standards (too large) (small),
  − platform back button does not work immediately after changing the theme (medium).

5.2.1.5 Error prevention

+ Mandatory fields are properly highlighted,
+ keyboard for text fields is limited based on the given field type,
− Intent module is not properly tested before adding (medium).

5.2.1.6 Look and see

+ User interface is simple and consistent,
– availability of quick menu is not visible, user must memorize it (medium),
– position in a tree structure is missing, user does not see which layer is he in (high).

5.2.1.7 Flexibility and effectivity

+ Basic settings are very simple,
– there is no advanced mode for advanced users (medium),
– there are no macros (small),
– there are no key shortcuts (small).

5.2.1.8 Minimality

+ Only the most important information are shown,
+ the concept is minimalistic,
+ only a single information displayed per module.

5.2.1.9 Meaningful error lines

– Only the platform default error line is present (high),
– the error line does not say what happened wrong (high),
– the error line does not say how to prevent the error from happening (high).

5.2.1.10 Help and documentation

– The documentation is very limited (high),
– the inner help is missing (high),
– there is no context hint for input fields (medium).

5.2.2 Conclusion

Overall results are relatively positive. Several compromises which break the platform standards had to be made for the sake of safety during usage, but the priority list is clear.

However, there are several missing supportive elements, such as prompts, error reports and hints. All of these are on the to-do list for later implementation and hopefully it will be fixed before the release of the application.

5.3 Testing with users

The importance of proper testing is critical as the driver cannot be distracted from driving. Any significant flaw in the application design might prove to be fatal and it certainly is not the intention. Therefore thorough testing must be performed in order to achieve the desired level of reliability.

One of the most commonly used approaches is the usability testing, where usually several testers try to perform certain actions with the application. This is done in a development environment and watched by the developers.

The UI is meant to be used in unusual conditions, therefore testing in development environment cannot cover usability tests well enough. Considering that, the UI evaluation follows commonly recognized rules about Car UI testing - the LCT (Lane Change Test). Given the issue of performing possibly dangerous tasks in live traffic, the usability tests are performed in safe but realistic environment - in the car simulator with real-drive scenarios.

Also, to see how the application is doing in context of competition, thorough A/B testing is performed. This testing uses the advantages of a car simulator as well. The competition to be compared with is the application Torque (2.1.1), which is currently one of the most downloaded OBD-supporting applications on Android market[1]).

As a related application has been developed simultaneously for a mobile platform [16], testing was performed for both of these applications together. Therefore, preparations were made just once as well as some parts of actual testing, such as introduction and questionnaires. However, both platforms were tested by each user so that the results are relevant.

5.3.1 Simulator

While providing safety, the car simulator has other advantages as well. It is equipped with set of cameras that track eye movement. It is then easy to find out, where and for how long is the user focusing his sight, which is really important for evaluating the cognitive load. It is also easy to try out different scenarios with a single click, not requiring to drive around looking for a proper place.

Figure 5.1. Simulator interior

The simulator is located in Albertov in Prague in the building of the Faculty of Transportation Sciences[2]), CTU. It is build from the interior (5.1) of the Škoda Octavia car and surrounded by three screens. There are three computers as the hardware background. One serves for computing the physical mode, other one serves for visualizing and the last one handles the control and communication interface.

The data from the simulator are broad-casted over serial line and they are limited to speed and revolutions per minute. However, they can supposedly be extended to wider area of information.

[1]) `https://play.google.com/store/search?q=OBD&c=apps&hl=en`
[2]) `https://www.fd.cvut.cz/english/`

The control software is called “CarDynamics” (shown on 5.2), it shows all kinds of data about the current state of the vehicle, such as speed, rounds per minute, steering wheel position, gear position, acceleration, position in a world model and more. It contains several world scenarios – highways, countryside or even small cities.

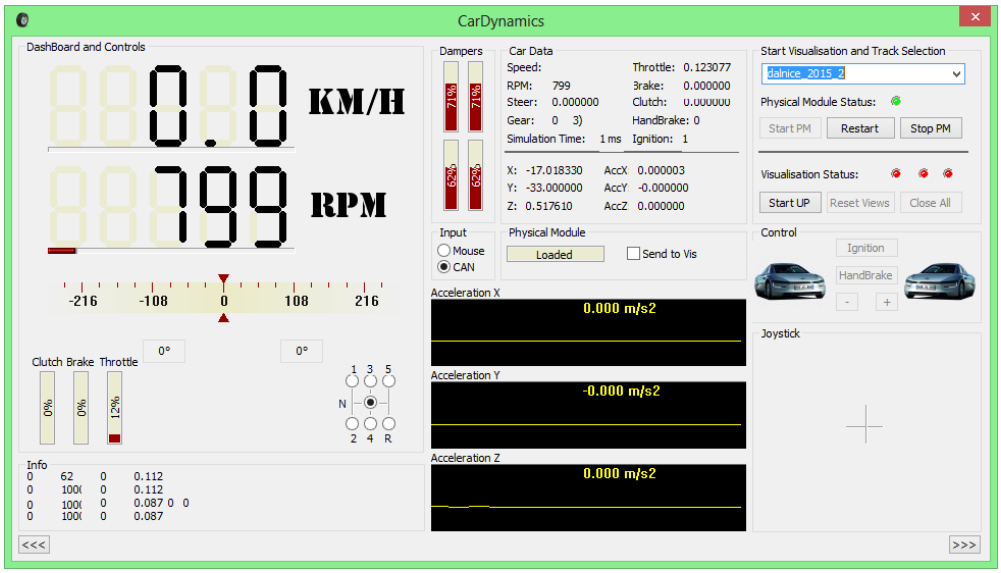


Figure 5.2. Screenshot from CarDynamics

As mentioned above, there is also an eye-tracking system. This enables the tester to precisely determine when and how often a user looks at the application, on the road or elsewhere. The system consists of two EyeTracker cameras and a complex software called Smart Eye Pro (see image 5.3).

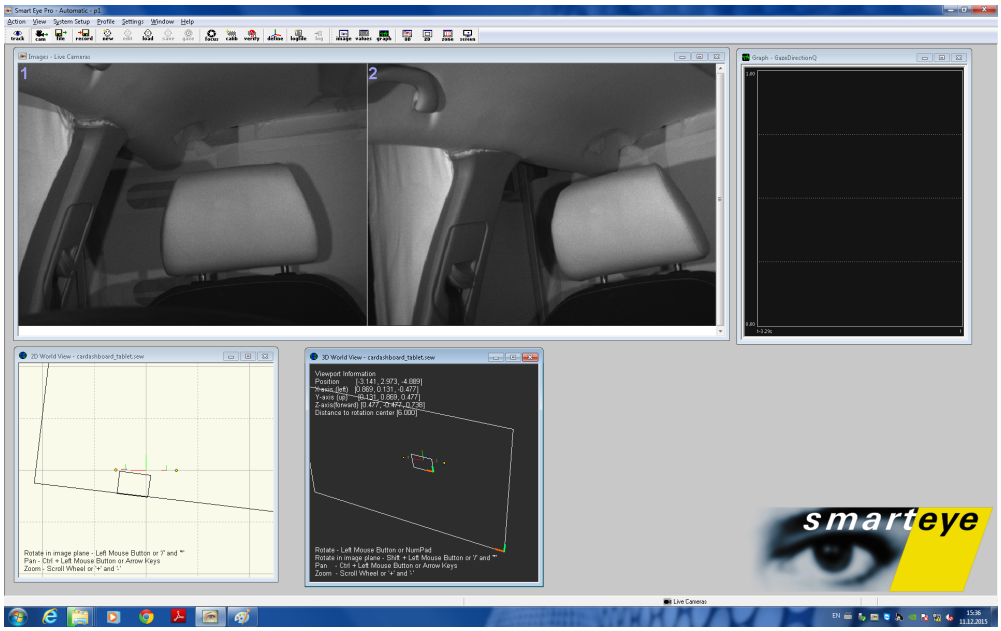


Figure 5.3. Screenshot from Smart Eye Pro

5.3.2 Preparations

Before even starting to test with users, several preparations had to be made. This included implementing a software, that can emulate an OBD module to transfer data

from the simulator to the application. Also cameras had to be prepared, because the software is not flawless and proper eye tracking prove to be an issue. Then, finally, scenarios could have been made for all the tested situations.

5.3.2.1 Software

As the simulator broadcasts data through serial link, it is impossible to easily catch the data in tablet and still have a realistic scenario. Also an OBD module is expected to be used with the application, therefore it is most suitable to emulate it. A simple software has been developed in Java language for such task. It listens to the serial link on a certain port, translates these data into inner Java primitive types, as they are in the C structure format. Then a bluetooth connection is initialized and an OBD module's protocol is emulated in order to communicate with the actual tablet application. This had to be optimized for Torque as well in order to have even conditions.

5.3.2.2 Cameras

After testing the eye tracking cameras, a new issue emerged. Those cameras were unable to track the intended position of tablet, as it was too low. Moving the tablet up did not help, therefore positions of the cameras had to be adjusted. The cameras were moved, so that they can track the eye when looking at the tablet. However, they were no longer able to track eyes when following the road. The test results had to be limited to looking at the tablet or not looking at the tablet.

5.3.2.3 Scenarios

As shown on tablet 5.1, the testing schedule has been made containing multiple supportive tasks as well as two testing scenarios. While the rest of schedule is shared across platforms, the A/B and LCT testing scenarios are unique for each platform, therefore more time is required.

Start	End	Duration [min]	Content
00:00:00	00:05:00	5	Introduction
00:05:00	00:10:00	5	Pre-test questionnaire
00:10:00	00:25:00	15	Instructions and EyeTracker setup
00:25:00	00:40:00	15	Warm-up driving
00:40:00	00:55:00	15	A/B testing
00:55:00	01:05:00	10	LCT testing
01:05:00	01:10:00	5	Post-test questionnaire
01:10:00	01:15:00	5	Debriefing

Table 5.1. Single user testing schedule

The A/B testing scenario is simple. At first the user drives the route without any application, so that he gets to know it. Then he drives the same route with application A (Torque), while being frequently asked to read out loud the speed and RPM. The speed is to be read from the first second with interval of 20 seconds, the RPM is to be read from the tenth second with interval of 20 seconds. The RPM and speed displays are located on different screens, so that the user has to scroll from one to another. The same scenario goes for the application B. A and B applications switch places for every tester (B first, A second or the other way around). The metrics then are the glance times for both applications.

The Lane Change Test (LCT) scenario is similar, however only the CarDashboard application is being tested. The driver is supposed to drive on a highway until a speed limit sign appears telling him to drive at 60 km/h. After a while (approximately 3

minutes) an object appears from nowhere 35 meters in front of the driver in his lane. He is then supposed to turn to the left lane and avoid the object. During the whole journey the user is asked for RPM to be read from the tablet application, so that he has to control the application while driving. The metrics is avoiding the object and following the predefined path while using the application. It measures how much the application distracts the driver and how much does it influence the reaction time.

5.3.2.4 World models

As there are multiple world models available, it is necessary to choose the proper ones for the given scenarios. Therefore several world models were tested and the best ones were chosen. For the AB testing, a countryside with villages on the route was chosen. It is the most realistic model available, containing even traffic and so on. For the LCT testing a highway world model without traffic is the most suitable. However it had to be edited, so that an object could appear at given location when driver crosses a certain radius. This object would appear seemingly randomly, but there were two fixed locations in an environment, when every part of the road looks the same, not giving the driver anything to memorize. This ensures that the driver is not prepared when the object appears.

5.3.2.5 Questionnaires

A Screener questionnaire was created in order to select participants. This questionnaire ensured that only relatively active drivers would participate. Also only smart-phone users were considered, as other drivers are not likely to use such a device (smart-phone, tablet) while driving. It is based on three questions:

- Are you a smart-phone user?
- Are you a driver?
- How many times per week do you drive?

A pre-test questionnaire was created in order to get some information about the participant. The information provided would be anonymous and would only serve for statistics. It consists of six questions:

- How old are you?
- What is your sex?
- How many kilometers you drive by a car per year?
- Which operating system is your smart-phone running?
- Have you ever used smart-phone while driving?
- How often are you using your smart-phone while driving?

This questionnaire also focuses on the market demand – trying to find out, if the drivers tend to use their devices while driving. Again, it is assumed that by answering the smart-phone questions, answers to similar tablet questions would be strongly related. Usually when a person has a smart-phone with a certain operating system and also has a tablet, the operating system on the tablet is the same. This simplifies the questionnaire and does not overload the participant, as the participation was voluntary and without payment.

A post-test questionnaire serves to find out, how satisfied are the participants with the application and the testing process itself. It consists of following five questions, some of which are to be rated from 1 to 5, where 1 is the best:

- What is your impression of using the device while driving? (1-5)

- Were the goals clear for you? (Yes/No)
- How acceptable was the way of solutions for given tasks? (1-5)
- Was the amount of displayed information appropriate? (1-5)
- Would you use the application in everyday driving? (Yes/No)

5.3.3 Process

After selecting 5 participants (based on the screener questionnaire) and scheduling the testing times, they were invited to come to the Faculty of Transportation Sciences in Albertov, where the simulator was located. There they filled the pre-test questionnaire and begin the actual testing based on a schedule mentioned in section 5.3.2.

After eye-tracking setup was done and once the participant felt comfortable with the simulator (after driving for a while in the warm-up phase), the scenarios took their place. The eye-tracking and simulator logging was turned on as the participant accomplished all the given tasks.

A post-questionnaire was filled and the testing was over. This was done for all 5 participants. Approximately 3 gigabytes of plain text data were collected from logging, both eye-tracking and simulator logging.

5.3.4 Questionnaire evaluation

5.3.4.1 Pre-test questionnaire

Both men and women participated in the testing, in age range from 22 to 45 years with the mean of 28 and standard deviation of 9.62. The kilometers per year driven by the participants were in range from 4000 to 30000 with the mean of 14800 and the standard deviation of 10849. As for the smart-phone operating systems, all the most commonly used were present. With Windows Phone and iOS both appearing once, the Android OS was present with 3 participants. All of the participants confessed to having used the smart-phone while driving at least once, with 3 of them confessing to use it often. The other 2 confessed to using the smart-phone only occasionally.

5.3.4.2 Post-test questionnaire

While the goals appeared to be clear for all the participants, not all were entirely happy with the application. The overall satisfaction with the application was in range from 1 to 3 out of 5 with the mean of 1.8 and the standard deviation of 0.8366 and the acceptability of the ways of solutions for given tasks was also in range from 1 to 3, also with the mean of 1.8, however with the standard deviation of 0.7582. The participants expressed satisfaction with the amount of information displayed, as the grades were in range from 1 to 2 with the mean of 1.2 and the standard deviation of 0.4472. At last, 4 out of 5 participants would use the application in everyday driving.

5.3.5 A/B testing evaluation

The A/B testing is a testing of Torque versus CarDashboard in the same condition. The tester drives while frequently being asked to read information of the screen (as described in section about scenarios 5.3.2). The eye-tracking system tracks the eye movement and logs every 16.6 milliseconds if the driver is looking at the tablet or not at the given moment. This ensures certain precision and reliability of results.

Logs have been examined and several outcomes measured. As first the comparison of glance frequency will be shown. Then some statistical data are measured, such as confidence intervals and such. An overall comparison is created. Then duration of

glance time is examined for both application and compared. The maximal glance time and the average glance time are compared as well as mean and standard deviation.

5.3.5.1 Glance frequency

Glance distribution in log files for Torque is visible on image 5.4. The X axis contains the individual log entries while the Y axis adds 1 per each user looking at the application at the given moment. The log entries are limited from 5000 to 10000 and added together, as shown on the R script below (the initialization script is shown on F.8).

```
addedData <- c()
for(i in 5000:10000){
  counter = 0
  for(j in 1:5){
    counter = counter + data[[j]]$Tablet[[i]]
  }
  addedData <- c(addedData, counter)
}
barplot(addedData)
axis(1)
```

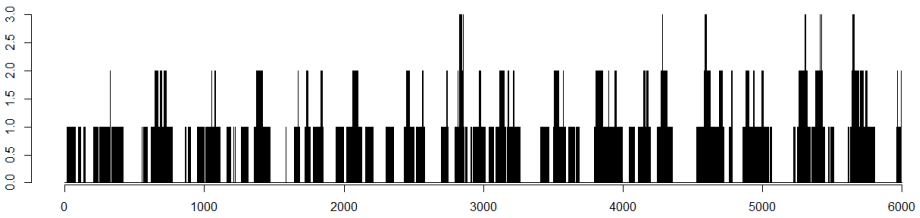


Figure 5.4. Glances for Torque

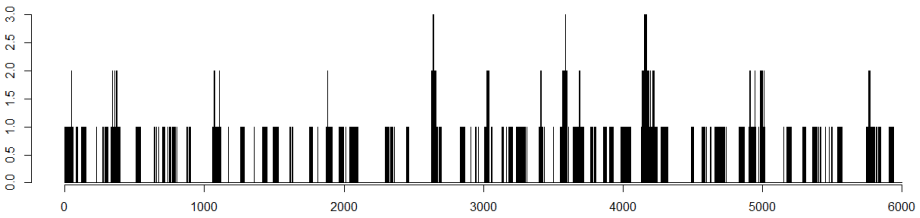


Figure 5.5. Glances for CarDashboard

Those are just illustrative plots to see the portion of time looking at the application (the black area of the plot) and not looking at the application (the white area of the plot). The glance distribution in log files for CarDashboard is visible below the Torque image for better comparison (5.5).

While the individual percentage ratio of time spent looking at the Torque is in range from 13 to 18 percent with the mean of 15.55 and the standard deviation of 2.19, the same for CarDashboard is only in range from 6 to 8 percent with the mean of 7.45 and the standard deviation of 1.32. The average percentage ratio of time spent looking at the Torque is then 15.55 %, while for CarDashboard it is only 7.40 %. The R script

for the average time ratio is shown below, the R scripts for individual (per participant) data are shown in F.10.

```
glanceLength = 0
glanceSum = 0
for(j in 1:5){
  glanceLength = glanceLength + length(data[[j]]$Tablet)
  glanceSum = glanceSum + sum(data[[j]]$Tablet)
}
print(100 * glanceSum / glanceLength)
```

5.3.5.2 Glance time

As another important metric, the maximal glance time (the longest continuous time interval spent looking at the application) measured for Torque was in range from 817 to 1616 milliseconds with the mean of 1263.3 and a standard deviation of 296.11. The same was measured for CarDashboard and the results for the maximal glance time are in range from 450 to 1217 milliseconds with the mean of 716.7 and a standard deviation of 298.38. This means that the average maximal glance time for CarDashboard is nearly half the average maximal glance time for Torque.

As mentioned by Daniel McGehee [17], drivers tend to try obtaining the required information for 1.5 to 2 seconds, then they give it up for the moment and try later. The 1.5-2 seconds glance time is also considered to be the safety limit for a single task requiring a visual focus. With the CarDashboard exhibiting maximal glance times lower than 1.2 second the limit of 1.5 second is not even reach, which is a success.

Furthering the glance time analysis, all the log entries were combined into one (as shown in R script in the addition F.9). Based on Student's t-test, there is 1 % chance that the driver will be looking continuously at the Torque application for more than 157.67 milliseconds. For CarDashboard there is 1 % chance of looking at the application for more than only 109 milliseconds. Both of the results from the t-test performed in R are shown in the additions (F.11 and F.12), the result for CarDashboard is shown below for illustration.

```
t.test(glanceList, alternative="less", conf.level=0.99)

        One Sample t-test

data:  glanceList
t = 24.935, df = 983, p-value = 1
alternative hypothesis: true mean is less than 0
99 percent confidence interval:
     -Inf 108.5484
sample estimates:
mean of x
 99.27168
```

With the average glance time being 143.926 milliseconds for Torque, the CarDashboard appears to perform much better with it's average glance time of 99.27 milliseconds. The glance times for Torque are shown on figure 5.6, the glance times for CarDashboard are shown below the Torque figure on figure 5.7.

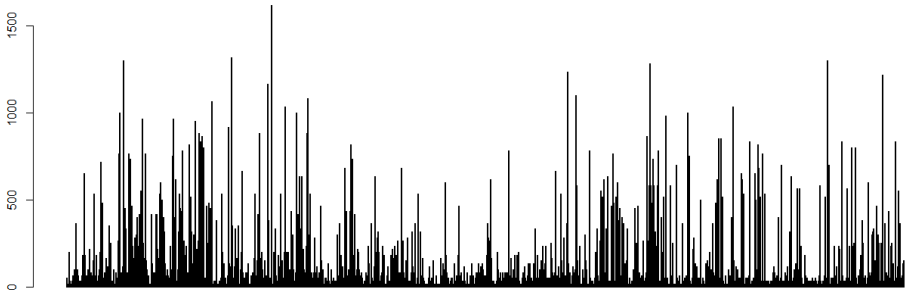


Figure 5.6. Torque glance times

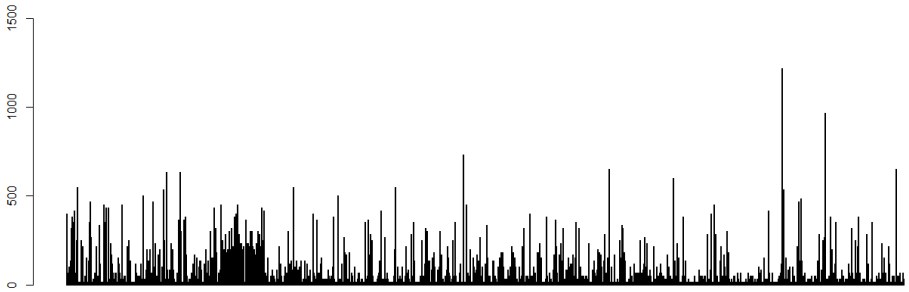


Figure 5.7. CarDashboard glance times

From further analysis it appears, that the distribution of glance times is exponential. Should this assumption be correct, the probability of looking at the application for longer time would decrease exponentially. The comparison plot used for determining the distribution is shown on image 5.8, it is a result of a R script shown below.

```
xMean = mean(glanceList)
xSd = sd(glanceList)
a = xMean - sqrt(0.25)
b = a~+ xSd * sqrt(11.75)
hist(glanceList, prob = 1)
xWidth=max(glanceList) - min(glanceList)
xGrid=seq(min(glanceList) - 0.2 * xWidth,
                  max(glanceList) + 0.2 * xWidth, length = 30)
lines (xGrid,dnorm(xGrid, mean = xMean, sd = xSd),
         col = 'red', lw = 2, lty = 2)
lines (xGrid,dunif(xGrid, min = a, max = b),
         col = 'blue', lw = 2, lty = 2)
lines (xGrid,dexp(xGrid, rate = 1 / xMean),
         col = 'green', lw = 2, lty = 2)
```

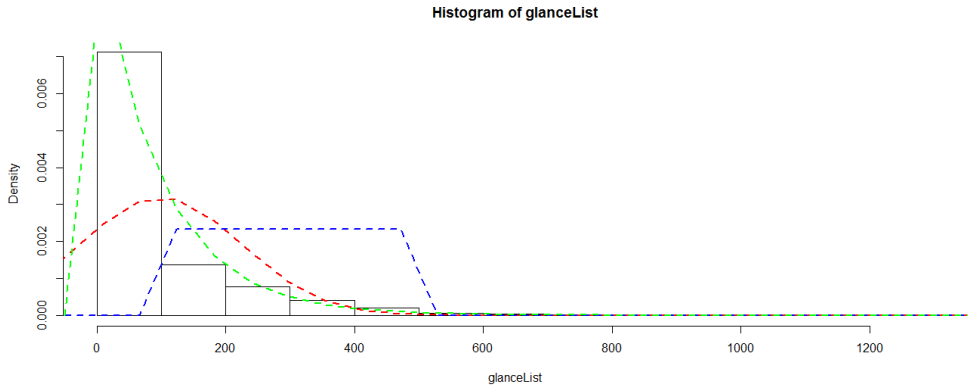


Figure 5.8. Glance times distribution

5.3.6 Lane Change Test evaluation

As described in the scenario (5.3.2), every participant drove for approximately 3 minutes in the speed of 60 kilometers per hour, when the object appeared from nowhere 35 meters in front of the car. The object was a yellow cube with 2.2 meters long side and a red arrow pointing to the left lane. This suggested the driver to go around the object by changing the lane to the left one.

5.3.6.1 Object avoidance

The most important metric in the Lane Change Test is avoiding the object while working with the application. Every participant managed to avoid the object, from which we can assume the success rate around 100 % (the success rate of the testing is 100 %, however it can be safely stated that nothing is perfect and real 100 % in every situation possible is not achievable).

5.3.6.2 Reaction time

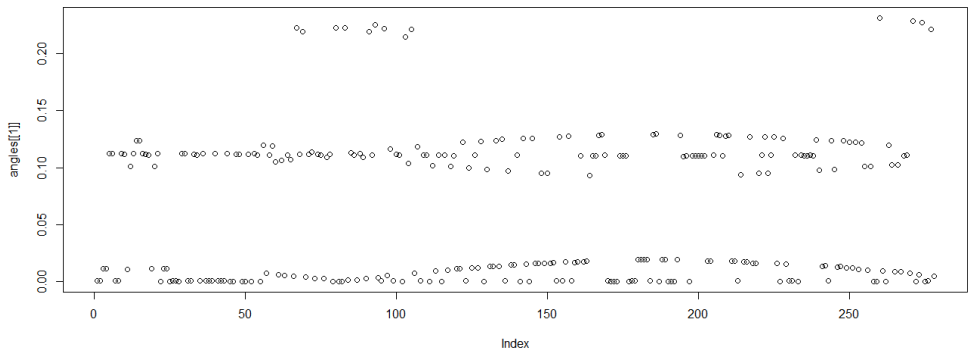


Figure 5.9. Path angles of the first participant (0.1 meters point distance)

The second metric is the reaction time. The reaction is determined from the path of the vehicle. The beginning of the turning of the car is considered as the start of the actual reaction. The turning is determined from significantly increasing the angle between subsequent vectors in the path. This is done by creating pairs of subsequent vectors from trinities of subsequent points. Experimentally, it was discovered (for the measured data, see example at figure 5.9 based on the script F.13) that an angle of 0.2 degrees

is significant enough to determine the turn. The exact results are visible in table 5.2, the visual comparison is visible on the related figure 5.10.

Color	Turn coordination X	Turn coordination Y
black	-5265.919	2294.193
red	-5267.260	2293.928
blue	-5266.978	2293.958
green	-5262.808	2293.958
yellow	-5263.845	2293.725

Table 5.2. Results of path angle comparison

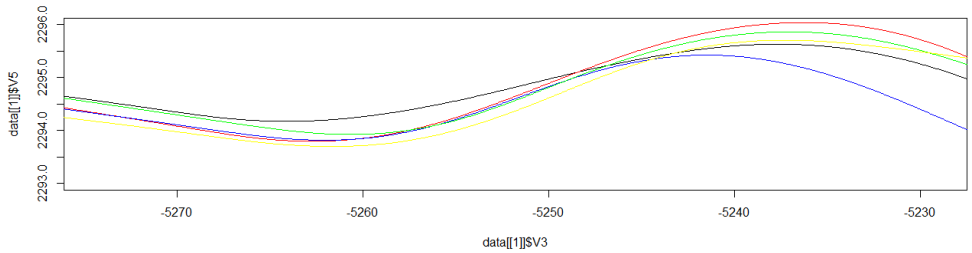


Figure 5.10. Turn paths

Given the results, a reaction time can be measured. As described by Marc Green [18], reaction time consists of three parts, which are highly situation-dependable and unexact:

- **Mental processing time** – the amount of time it takes for brain to perceive a signal and to decide upon a response,
- **movement time** – the amount of time it takes to perform the response (using muscles),
- **device response time** – the amount of time it takes for a vehicle to react to input.

For the purpose of evaluating, the reaction time will be assumed to be a single value without dividing it into three parts. A note has to be made, that the simulator reaction delay is much lower than real vehicle delay, therefore a device response time it minimized. When driving at 60 kilometers per hour, it is approximately the speed of 16,667 meters per second. The times of reaction were measured based on the speed and the traveled distance from the moment of object appearance. The measured times and the distances (from the object) of turns are shown in table 5.3, an illustration is shown in figure 5.11.

Participant	Turn distance [m]	Turn time [ms]
P1	8.29	497.4
P2	6.97	418.4
P3	7.25	435.1
P4	11.41	684.6
P5	10.39	623.6

Table 5.3. Times and distances of turn from the object

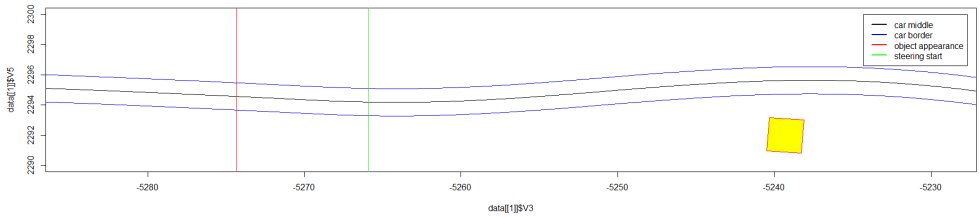


Figure 5.11. Car path

According to statistics gathered so far on a human benchmark page[1]), where participants push the button when it turns green (knowing it will do so) five times over, the average human reaction time for highly expected signal is 215 milliseconds. While the participants in the change lane test did expect something to happen and that they will have to go around something, they drove for three minutes focusing on keeping the speed as well as controlling the tested application in a homogeneous environment of highway in the forest, which heavily lowers their alertness. However, the results are still under 0.7 second with the mean of 531 milliseconds. As described in a master's thesis by Pamela M. D'Addario [19], a similar test was performed focusing on alerted obstacle crash avoidance. Alerted means that the participants knew what is the test about. The measured mean time was 0.78 seconds. Another similar test was performed also with average results around 0.7 seconds. This means that the application controlling in the change lane test did not influence the participant's awareness. The results are even slightly better than the ones from the tests mentioned above, but it can be caused by the difference of measured signal – the accelerator pedal vs. steering wheel movement, as the steering wheel movement is usually performed faster.

5.4 Summary

Thanks to automatic testing, the amount of bugs that got into the release version was limited. The unit tests prevented many bugs from flowing through. However, certain bugs and imperfections were present in the automatic tests as well, allowing dysfunctional code go past them. The test driven development just adds another layer to the application development protection layer, but it is not perfect.

After the development of the graphical user interface was done, the heuristic testing came into place. Many shortcomings have been found, however, not all were fixed because of the thesis deadline. They will surely be fixed as soon as possible, as the application will be released into the real market and it has to be nearly perfect by then. The heuristic testing however gives a great insight into certain areas of the user interface.

Then even with a few difficulties, the application was tested in the most suitable environment available – the realistic car simulator. Thanks to the eye-tracking and the simulator logs huge amount of data has been gathered. The evaluation of these data was done using the R software.

The A/B testing against the potential competitor – Torque was performed. The tests have shown that the CarDashboard performs better in the real life scenarios based on obtaining information from a car.

The Lane Change Test was done in order to find out the influence of reaction time when using the application. None or minor influence (in terms of units of milliseconds)

[1]) available at `http://www.humanbenchmark.com/tests/reactiontime`

has been found, from which one can assume that the application usage is as safe as planned.

Overall the testing was successful and it helped developing a good and safe application for in-car environment. The testing participants were usually happy with the way the application is done and most of them claimed that they would use the application daily.

Chapter 6
Conclusion

In this chapter, first the assignment completion will be reviewed (fulfillment will be described for every single point of assignment). Information about the project life cycle, it's present and future will follow. Finally the thesis will be reviewed followed by a personal opinion and experience.

6.1 Assignment completion

6.1.1 Completing the assignment tasks

6.1.1.1 Review existing Android applications for in-car use

The reviewed applications are described in the section 2.1. There are following applications: Torque, CarHome Ultra, Car Dashdroid, Ultimate Car Dock. Also Android Auto was briefly described, as it is the current direction of Google in the automotive area.

6.1.1.2 Review and analyze User Interface development methods for in-car infotainment applications

A GUI is analyzed in section 2.4. This analysis contains requirements for GUI (2.4.1) as well as differences for the in-car usage (2.4.2). It also describes the development process of user interface in section 2.4.3.

6.1.1.3 Analyze the in-car OBD API and exported data

The OBD is analyzed in section 2.9. The API is then described in section 2.9.2 and the data are described in 2.9.3. The OBD is also mentioned in the realization chapter in section 4.3.5.

6.1.1.4 Design an application system architecture for accessing the OBD data and resources

The application architecture is described in section 3.1, where the limitations of the platform architecture (3.1.1), the extensibility (3.1.2), the modularity (3.1.3) and the adaptability (3.1.4) are described.

6.1.1.5 Design a tablet User Interface for in-car use

The process of designing the tablet User Interface for in-car use is described in the design section 3.2. It presents four phases of GUI creation process and the emergence of the final GUI design.

6.1.1.6 Design and implement in-car application offering the OBD data for Android tablet platform

The application development phase is described in chapter 4 in sections 4.2 and 4.3. The OBD access itself is described in section 4.3.5 while the data are provided using modules described in section 4.3.1.

6.1.1.7 Perform UI and application testing and evaluate results

The testing is described in chapter 5. Both application (5.1) and UI testing (5.2 and 5.3) are present. The testing with users was performed on a realistic car simulator and it is thoroughly described in section 5.3.

6.2 Project life cycle

6.2.1 Present

Currently the application is being prepared for release. As mentioned in the chapter about testing (5), some adjustments have to be made in order to fix all the issues. After this is done, the application will be releases to the Google Play Store[1]).

6.2.2 Future

In the future, the application will be enhanced with additional functionality, statistics and so on. It is planned to implement a logging mechanism, which will log interesting data on server. Those data can later be evaluated and presented to the user (for example his driving style can be evaluated, it can also serve as a path tracker and so on).

Also, the application can serve as a device for Internet of Things as a controller and/or a viewer. It can display simple information (such as home temperature) or even turn on lights, open the garage, play music, etc. Some of the functionality is already being tested, the application contains a few IoT modules (temperature display and light switch).

6.3 Summary

The goal of this thesis has been achieved, however a lot of work is still left to be done. The CarDashboard application offers a simple graphical user interface and easy extendability, making it truly versatile. The car does not have to be the only environment the application can be used in. The next step can be smart homes or even factories, where precision and simplicity are necessary.

I am really glad I had a chance to work with a wide area of technologies during this project. I had a chance to perform a proper testing with access to the car simulator, which is far better than what I could've imagined at the beginning. I used all kinds of statistics when evaluating results and it was very interesting to see it from different points of view.

I've learned a lot more about developing for Android. However imperfect it was, it is a great experience to have. Also trying the GUI development process in practice is really interesting. During that I've learned some wonderful techniques from start to end.

And at last but not least, I've improved my research abilities, as I've reviewed several research papers and soon I might attempt to create one or two (in cooperation with Lukáš Hrubý) regarding this topic. Overall, this has been a great experience.

[1]) `https://play.google.com/store?hl=en`

References

[1] GORMAN, Ryan. One in four car accidents caused by cell phone use while driving.. but only five per cent blamed on texting. *Http://www.dailymail.co.uk* [online]. Associated Newspapers Ltd, 2014, 2014-03-27 [cit. 2016-01-03]. Available from: http://www.dailymail.co.uk/news/article-2591148/One-four-car-accidents-caused-cell-phone-use-driving-five-cent-blamed-texting.html

[2] *Designing for Android Auto* [online]. Google Inc. [cit. 2016-01-03]. Available from: https://www.google.com/design/spec-auto/designing-for-android-auto/designing-for-cars.html

[3] LUNDEN, Ingrid. Gartner: 195M Tablets Sold In 2013, Android Grabs Top Spot From iPad With 62% Share. *Http://techcrunch.com* [online]. AOL Inc., 2014, 2014-03-03 [cit. 2016-01-03]. Available from: http://techcrunch.com/2014/03/03/gartner-195m-tablets-sold-in-2013-android-grabs-top-spot-from-ipad-with-62-share/

[4] *Android Developers* [online]. Google Inc. [cit. 2016-01-03]. Available from: http://developer.android.com/index.html

[5] *Material design* [online]. Google Inc., 2015 [cit. 2016-01-09]. Available from: https://www.google.com/design/spec/material-design/

[6] ŽIKOVSKÝ, Pavel. *Návrh uživatelských rozhraní: Úvod* [online]. 2015, 10-16 [cit. 2016-01-03]. Available from: https://edux.fit.cvut.cz/courses/MI-NUR/_media/lectures/x01-uvod.pdf

[7] HEATON, Andrew. Designing for In-Dash Automotive. A UX Primer. *REVINITY : ANDREW HEATON* [online]. 2013-07-02 [cit. 2016-01-05]. Available from: http://revinity.com/?p=128

[8] ŽIKOVSKÝ, Pavel. *Návrh uživatelských rozhraní: Návrh UI, prototypy* [online]. 2015, 1-71 [cit. 2016-01-05]. Available from: https://edux.fit.cvut.cz/courses/MI-NUR/_media/lectures/x02-navh_a_prototyping.pdf

[9] NIELSEN, Jakob. How Many Test Users in a Usability Study? *Nielsen Norman Group* [online]. 2012-06-04 [cit. 2016-01-07]. Available from: https://www.nngroup.com/articles/how-many-test-users/

[10] WHITE, Oliver. Java Tools and Technologies Landscape for 2014. *ZeroTurnaround* [online]. ZeroTurnaround, 2014-05-21 [cit. 2016-01-07]. Available from: http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/

[11] DUCKETT, Chris. Google releases Android Studio, kills off Eclipse ADT plugin. *ZDNet* [online]. CBS Interactive, 2014-12-09 [cit. 2016-01-07]. Available from: http://www.zdnet.com/article/google-releases-android-studio-kills-off-eclipse-adt-plugin/

[12] FOWLER, Martin. Continuous Integration. *Martin Fowler* [online]. 2006-05-01 [cit. 2016-01-08]. Available from: http://www.martinfowler.com/articles/continuousIntegration.html

[13] *Global OBD Vehicle Communication Software Manual* [online]. Snap-on Inc., 2013-08-01 [cit. 2016-01-08]. Available from: https://www1.snapon.com/Files/Diagnostics/UserManuals/GlobalOBDVehicleCommunicationSoftwareManual_EAZ0025B43B.pdf

[14] PIRES, Paulo. *OBD-II Java API*. 2015-12-18 [cit. 2016-01-07] Available from: https://github.com/pires/obd-java-api.

[15] MCCONNELL, Steve. *Code complete*. 2nd ed. Redmond, Wash.: Microsoft Press, 2004, xxxvii, 914 p. ISBN 079-0145196705.

[16] HRUBÝ, Lukáš. *Car infotainment application on a smartphone*. Prague, 2016. Master's Thesis. Faculty of Electrical Engineering, CTU in Prague.

[17] MCGEHEE, Daniel V. *Visual and cognitive distraction metrics in the age of the smart phone: A basic review* [online]. Association for the Advancement of Automotive Medicine, 2014, 2014-03-01 [cit. 2016-01-08].

[18] GREEN, Marc. *Driver Reaction Time* [online]. Visual Expert, 2013 [cit. 2016-01-08]. Available from: http://www.visualexpert.com/Resources/reactiontime.html

[19] D'ADDARIO, Pamela Maria. *Perception - Response Time to Emergency Roadway Hazards and the Effect of Cognitive Distraction*. Toronto, 2014. Master's Thesis. University of Toronto.

Appendix A
CD content

```
|   readme.txt.........................description of the CD content
+---apk..................folder containing the application .apk file
+---src..................................................sources folder
|   +---CarDashboard.....................implementation source codes
|   \---MP_Blaha_Michael_2016.....................tex source codes
\---text.....................................thesis text folder
        MP_Blaha_Michael_2016.pdf.................thesis in PDF format
```

Appendix B
User's guide

B.1 Installation guide

B.1.1 Prerequisites

- application .apk installation file,
- Android tablet device,
- OBD-II Blue-tooth adapter (optional).

B.1.2 Installation process

1. Copy the .apk file inside the Android device,
2. execute the .apk file from the Android device file system,
3. complete the installation process,
4. insert OBD-II Blue-tooth adapter into the car (optional),
5. pair device with the adapter via Blue-tooth (optional),
6. launch the application (optional),
7. go to settings (optional),
8. connect the application to the adapter (optional).

B.2 User guide

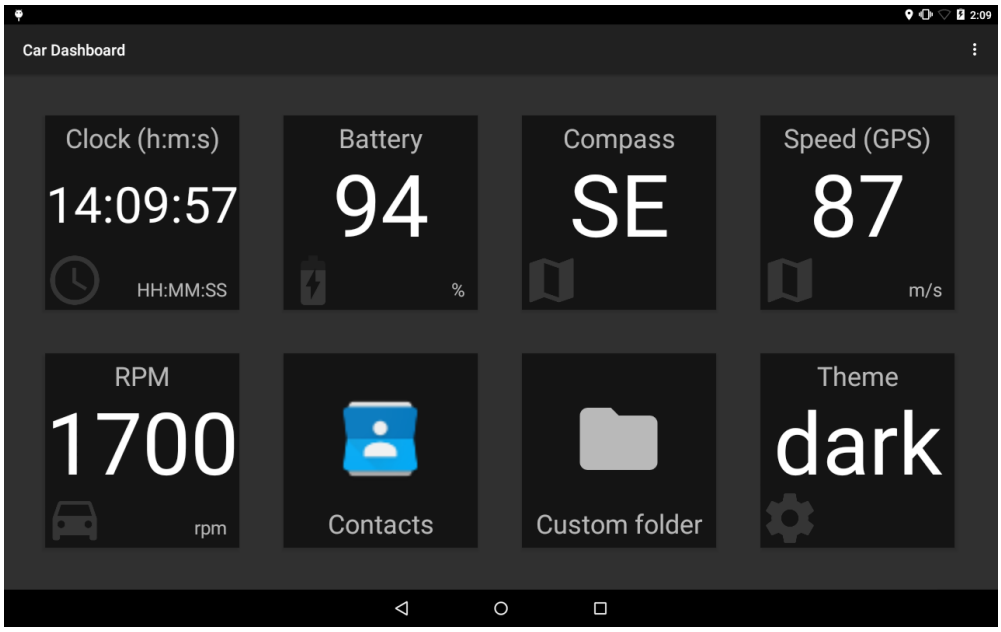


Figure B.1. Application GUI

As visible in figure B.1, there are several rectangular modules visible. Interaction with a module is done using a single touch. For modules displaying some value, the touch causes the application to say the value out loud. For other modules it performs an action depending on the module type:

- Folder module invokes a sub-folder (displays other modules),
- shortcut module invokes a different application or an intent,
- add module invokes a module selection screen for addition,
- back module goes back to the parent from sub-folder,
- other modules, such as light button module, can invoke certain action (based on their purpose).

Navigation in application is done using a swipe to the left or right. This changes the screen to the right or to the left screen. Navigation to sub-folders is done via touching them, as mentioned earlier.

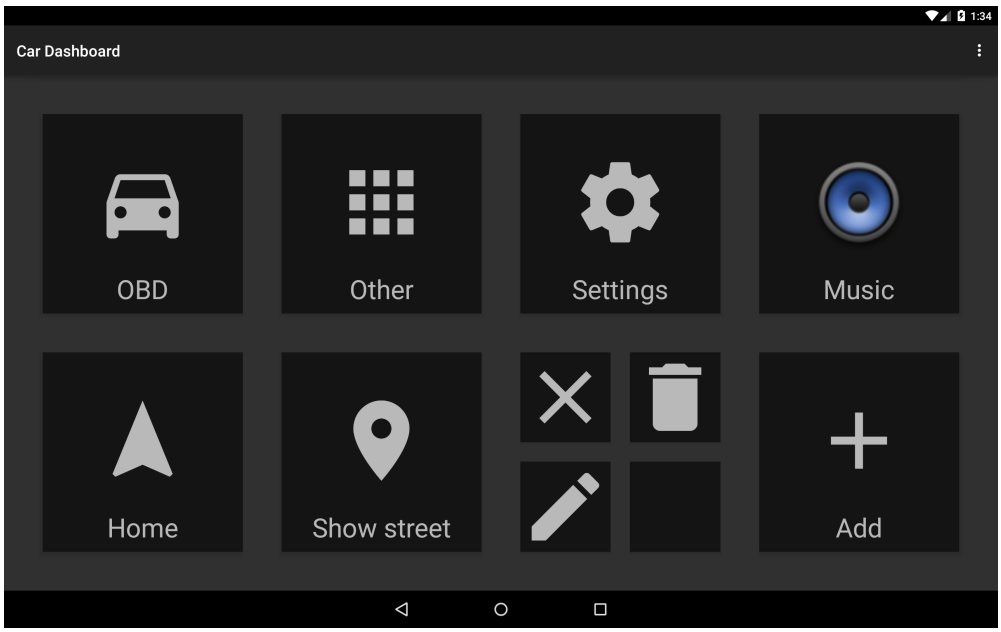


Figure B.2. Application GUI with Quick menu (2nd row, 3rd column)

There is also a feature called "Quick menu" for almost every module (with exception of back modules and add modules). The Quick menu is invoked by a long touch. As shown in figure B.2, the touched module divides itself into four separate buttons containing basic features - cancel, delete and edit.

Adding a module is done via the add module (module with a plus icon). This invokes a list of available modules (divided into few sections). Some modules can require additional settings before being added. Such module is an intent module or an application shortcut module.

Appendix C
Glossary

- API — Application Program Interface
- CI — Continuous Integration
- CTU — Czech Technical University
- DAO — Data Access Object
- GUI — Graphical User Interface
- HUD — Head-Up Display
- IDE — Integrated Development Environment
- IoT — Internet of Things
- JSON — JavaScript Object Notation
- LCT — Lane Change Test
- MVC — Model-View-Controller
- OBD — On-Board Diagnostics
- RPM — Revolutions Per Minute
- SMS — Short Message Service
- TDD — Test Driven Development
- TTS — Text To Speech
- UI — User Interface
- VCS — Version Control System
- XML — eXtended Markup Language

Appendix D
Images

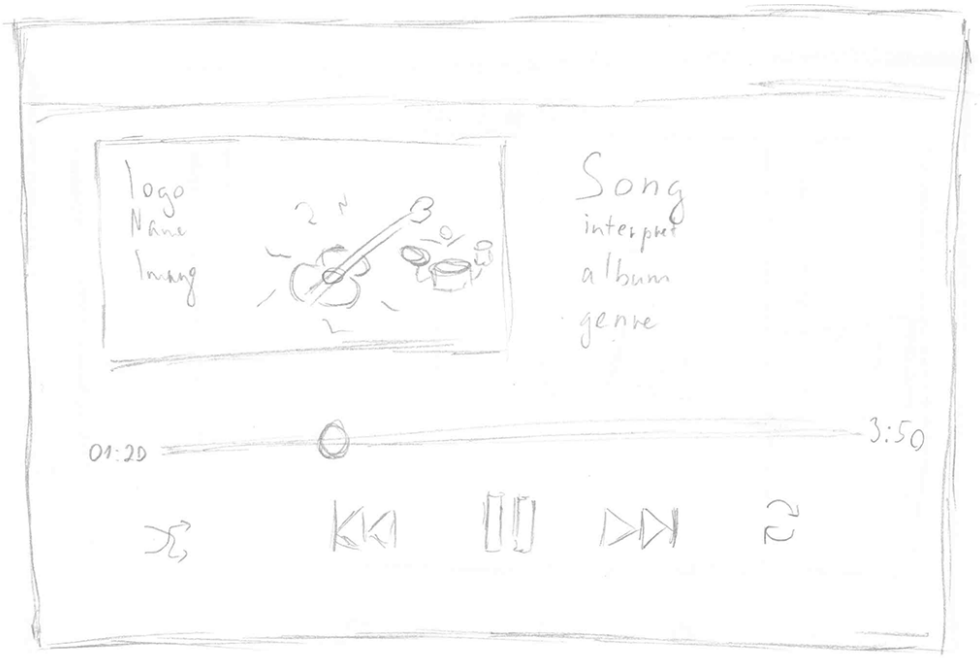


Figure D.3. Music player GUI draft

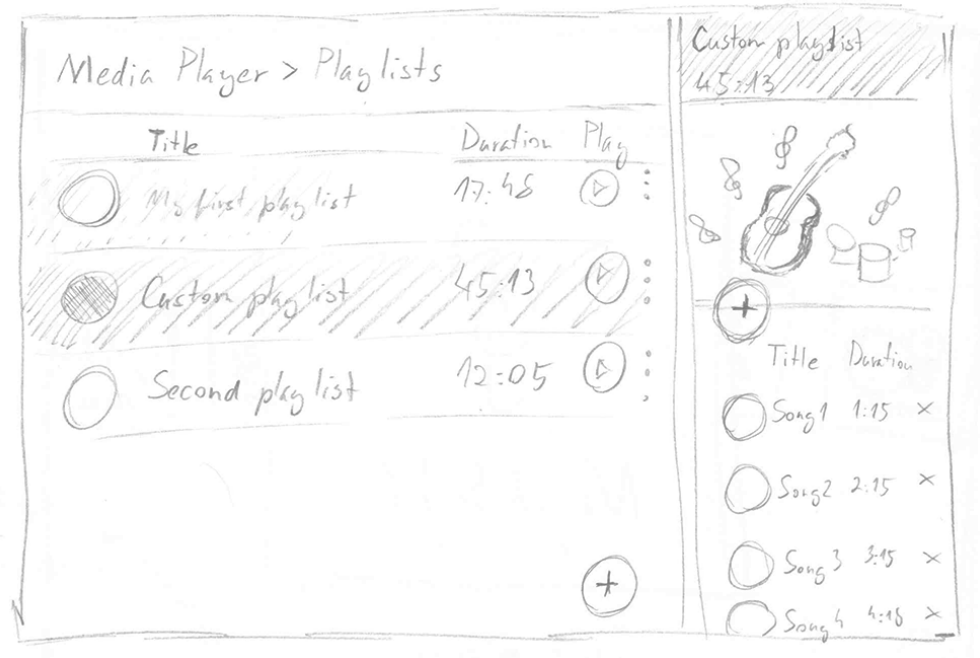


Figure D.4. Music playlist GUI draft

Figure D.5. Implementation of the draft image 3.1

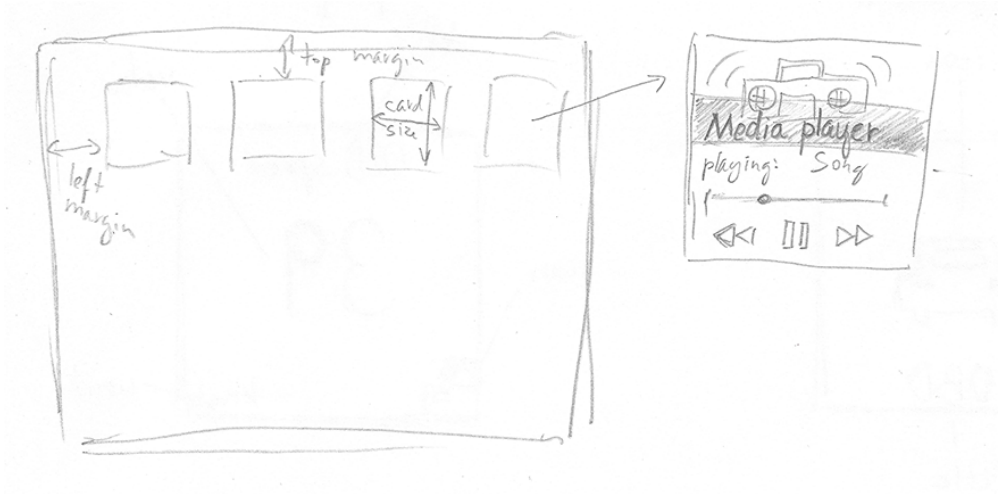


Figure D.6. The grid with a music player panel and measurements

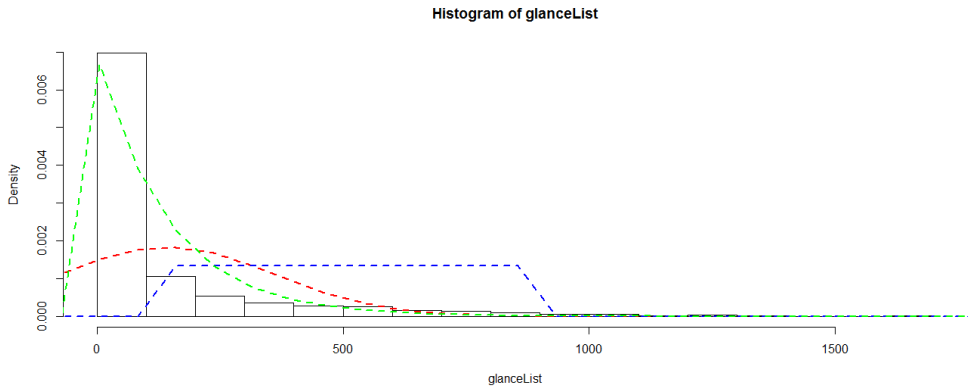


Figure D.7. Glance times distribution for Torque

Appendix E
Tables

	Torque		CarDashboard	
Participant	**Glance ratio**	**Max glance time**	**Glance ratio**	**Max glance time**
P1	18.63329 %	1300 ms	8.291183 %	633 ms
P2	15.71007 %	1617 ms	5.478639 %	450 ms
P3	13.1708 %	817 ms	7.768305 %	550 ms
P4	13.76879 %	1283 ms	8.866204 %	733 ms
P5	16.47285 %	1300 ms	6.895412 %	1216 ms
Mean	15.55116 %	1263 ms	7.459949 %	717 ms
S. deviation	2.191875 %	286.1138 ms	1.32362 %	298.3752 ms

Table E.1. A/B testing

Appendix F
Scripts

```
data <- list()
data[[1]] <- fread("../usability tests/p1_t_ab_torque.log",
                   select = c("ClosestWorldIntersection.objectName"))
data[[2]] <- fread("../usability tests/p2_t_ab_torque.log",
                   select = c("ClosestWorldIntersection.objectName"))
data[[3]] <- fread("../usability tests/p3_t_ab_torque.log",
                   select = c("ClosestWorldIntersection.objectName"))
data[[4]] <- fread("../usability tests/p4_t_ab_torque.log",
                   select = c("ClosestWorldIntersection.objectName"))
data[[5]] <- fread("../usability tests/p5_t_ab_torque.log",
                   select = c("ClosestWorldIntersection.objectName"))
for(i in 1:5){
  data[[i]][data[[i]]!="tablet"] <- 0
  data[[i]][data[[i]]=="tablet"] <- 1
  names(data[[i]])[names(data[[i]])=="ClosestWorldIntersection.objectName"]
      <- "Tablet"
  data[[i]]$Tablet <- as.numeric(data[[i]]$Tablet)
}
```

Figure F.8. Initial script for loading and preparing the data

```
glanceList <- c()
counter = 0
for(j in 1:5){
  for(i in 1:length(data[[j]]$Tablet)){
    val = data[[j]]$Tablet[[i]]
    if(val == 0){
      if(counter > 0){
       # multiplied by 1000/60 because of the~log entry frequency
       #  -> 1000/60 milliseconds
        glanceList <- c(glanceList, counter * 1000 / 60)
        counter = 0
      }
    } else {
      counter = counter + 1
    }
  }
}
if(counter > 0){
  glanceList <- c(glanceList, counter * 1000 / 60)
}
```

Figure F.9. Creating list of glance times

```
eval <- list(c(),c())
```

```
for(j in 1:5){
  ratio = sum(data[[j]]$Tablet) / length(data[[j]]$Tablet)
  max = 0
  counter = 0
  for(i in 1:length(data[[j]]$Tablet)){
    val = data[[j]]$Tablet[[i]]
    if(val == 0){
      if(max < counter){
        max = counter
      }
      counter = 0
    } else {
      counter = counter + 1
    }
  }
  eval[[1]] <- c(ratio * 100, eval[[1]])
  eval[[2]] <- c(max * 1000 / 60, eval[[2]])
  print(ratio * 100)
  print(max * 1000 / 60)
}
mean(eval[[1]])
sd(eval[[1]])
mean(eval[[2]])
sd(eval[[2]])
```

Figure F.10. Evaluating glance ratio and max glance time for all the participants

```
> t.test(glanceList, alternative="less", conf.level=0.99)

        One Sample t-test

data:  glanceList
t = 24.39, df = 1360, p-value = 1
alternative hypothesis: true mean is less than 0
99 percent confidence interval:
     -Inf 157.6701
sample estimates:
mean of x
  143.926

> t.test(glanceList, conf.level=0.99)

        One Sample t-test

data:  glanceList
t = 24.39, df = 1360, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
99 percent confidence interval:
 128.7046 159.1475
sample estimates:
mean of x
  143.926
```

Figure F.11. Confidence intervals for Torque glance times

```
> t.test(glanceList, conf.level = 0.99, alternative="less")

        One Sample t-test

data:  glanceList
t = 24.935, df = 983, p-value = 1
alternative hypothesis: true mean is less than 0
99 percent confidence interval:
     -Inf 108.5484
sample estimates:
mean of x
 99.27168

> t.test(glanceList, conf.level = 0.99)

        One Sample t-test

data:  glanceList
t = 24.935, df = 983, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
99 percent confidence interval:
  88.99696 109.54640
sample estimates:
mean of x
 99.27168
```

Figure F.12. Confidence intervals for CarDashboard glance times

```
startX <- c()
startY <- c()

angles <- list(c(),c(),c(),c(),c())
xpoints <- list(c(),c(),c(),c(),c())
ypoints <- list(c(),c(),c(),c(),c())

for(column in 1:5){
  printedA = 0
  printedB = 0
  xlimA = 0
  xlimB = 0
  for(i in 1:length(data[[column]]$V3)){
    if(printedA == 0){
      if(data[[column]]$V3[[i]] >= (-5239.3-35)){
        print(i)
        xlimA = i
        printedA = 1
      }
    } else {
      if(printedB == 0){
        if(data[[column]]$V3[[i]] >= (-5239.3)){
          print(i)
          xlimB = i
          printedB = 1
        }
```

```
      } else {
        break
      }
    }
  }

  space = 0.1
  xCoord = 0
  x1 = 0
  x2 = 0
  x3 = 0
  y1 = 0
  y2 = 0
  y3 = 0
  for(i in xlimA:xlimB){
    if(xCoord == 0){
      x1 = data[[column]]$V3[[i]]
      y1 = data[[column]]$V5[[i]]
      xCoord = x1
      xpoints[[column]] <- c(xpoints[[column]], x1)
      ypoints[[column]] <- c(ypoints[[column]], y1)
    } else {
      xCoord2 = data[[column]]$V3[[i]]
      if(abs(xCoord - xCoord2) < space){
      } else {
        if(x2 != 0){
          x3 = data[[column]]$V3[[i]]
          y3 = data[[column]]$V5[[i]]
          xCoord = x3
          xpoints[[column]] <- c(xpoints[[column]], x3)
          ypoints[[column]] <- c(ypoints[[column]], y3)
          ux = x2 - x1
          uy = y2 - y1
          vx = x3 - x2
          vy = y3 - y2
          top = ux * vx + uy * vy
          bot = sqrt(ux * ux + uy * uy) * sqrt(vx * vx + vy * vy)
          angles[[column]] <- c(angles[[column]], acos(top/bot)*180/pi)
          x1 = x2
          y1 = y2
          x2 = x3
          y2 = y3
        } else {
          x2 = data[[column]]$V3[[i]]
          y2 = data[[column]]$V5[[i]]
          xCoord = x2
          xpoints[[column]] <- c(xpoints[[column]], x2)
          ypoints[[column]] <- c(ypoints[[column]], y2)
        }
      }
    }
  }
```

```
  angles[[column]][is.nan(angles[[column]])] <- 0

  for(i in 1:length(angles[[column]])){
    if(angles[[column]][[i]] > 0.2){
      print(xpoints[[column]][[i+1]])
      print(ypoints[[column]][[i+1]])
      print(i)
      startX <- c(startX, xpoints[[column]][[i+1]])
      startY <- c(startY, ypoints[[column]][[i+1]])
      break
    }
  }
}
plot(angles[[1]])
```

Figure F.13. Script for calculating the path angles