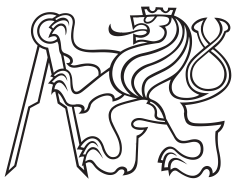
Master's Thesis



Czech
Technical
University
in Prague

F8

Faculty of Information Technology
Katedra softwarového inženýrství

Tablet infotainment system

Bc. Michael Bláha

January 2016

Supervisor: Ing Jan Šedivý, CSc.

/ Declaration

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Abstrakt / Abstract

Tento dokument je pouze pro potřeby testování.

This document is for testing purpose only.

Contents /

Chapter 1
Introduction TODO

sources:

- https://docs.google.com/document/d/1pGtlS5uY4PdKfHjf83dFGrajyVcea0tvzAnwXf61Cs8/edit
- generally progress: https://docs.google.com/document/d/1CEWym7MphsCO0v3CXe_bTHOgFBgquBbPVb edit

1.1 Project TODO

See above

1.1.1 Motivation TODO

See above

1.2 Assignment analysis

1.2.1 Assignment tasks

1.2.1.1 Review existing Android applications for in-car use

One of the key approches in research project is reviewing the existing progress in the given field. Reviewing existing applications helps understanding the topic, seeing the bigger picture, learning from mistakes of others and last but not least, getting general idea about competition.

1.2.1.2 Review and analyse User Interface development methods for in-car infotainment applications

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

1.2.1.3 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 from speed to engine temperature. This task focuses on understanding and gathering data from the OBD API.

1.2.1.4 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 proper architecture is required for everything to work well. The application has to gather data, while displaying them properly without unneccesary (TODO FIX!) delay.

1.2.1.5 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.

1.2.1.6 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.

1.2.1.7 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 UI testing with reaul users in a car simulator.

Chapter 2
Analysis TODO

sources:

- application analysis https://docs.google.com/document/d/1QyOiMzVOikcDhPY3P5MsRL_80cCGzjoGf
  UL0/edit
- priority list https://docs.google.com/document/d/1juKYgUUDSI5CmfzjR4BsYSPHVYCGqrWuejgbhqzw
  edit

2.1 Existing applications

2.1.1 Applications TODO

sources:

- https://docs.google.com/document/d/1p_pSGTUHEojOyP7ICCDNVV7RW1vn8iN_KECipC4Y9tY/
  edit
- http://www.makeuseof.com/tag/5-best-dashboard-car-mode-apps-android-
  compared/

2.1.2 Torque

Starting with an empty screen, lot of settings are required before using this application, since there is no default mode. Adding new views is easy and intuitive, but still very confusing. The add menu lacks hierarchy and everything is just sorted array of various options. There is no cancel button when popping the menu dialog.

This application can actually show almost anything OBD provides. It supports differents types of display, but it is hard to tell by their names. Responsiveness it not smooth at all and launching the application in horizontal mode confuses it, everything behaves like if it was in vertical mode.

Figure 2.1. Screenshot from Torque

2.1.2.1 Advantages

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

2.1.2.2 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 laggy.

2.1.3 CarHome Ultra

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

It appears to be just a simple application offering speed, compass, weather and external application launcher. The new version also displays location (address) and a phone version is able to respond to text messages. It also supports text to speech (on touch).

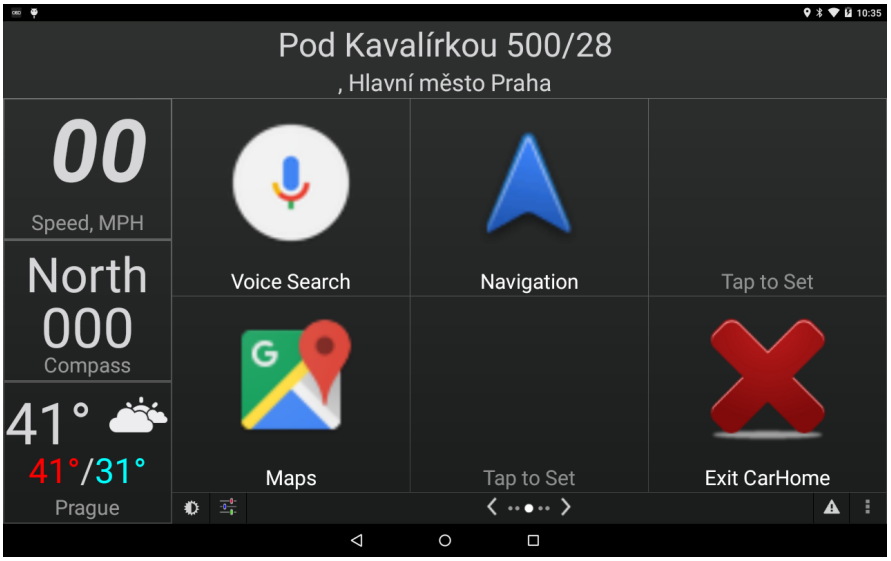


Figure 2.2. Screenshot from CarHome Ultra

2.1.3.1 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.

2.1.3.2 Disadvantages

- Small buttons on small screens (fixed 6 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 few taps).

2.1.4 Car Dashdroid

After a long loading the main window appears. It has three screens, which change by swiping right or left. The left screen contains 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 command.

It also provides settings for bluetooth, brightness, screen rotation, fullscreen, day/night mode and application settings, where other options can be set, such as home adress, theme, units.

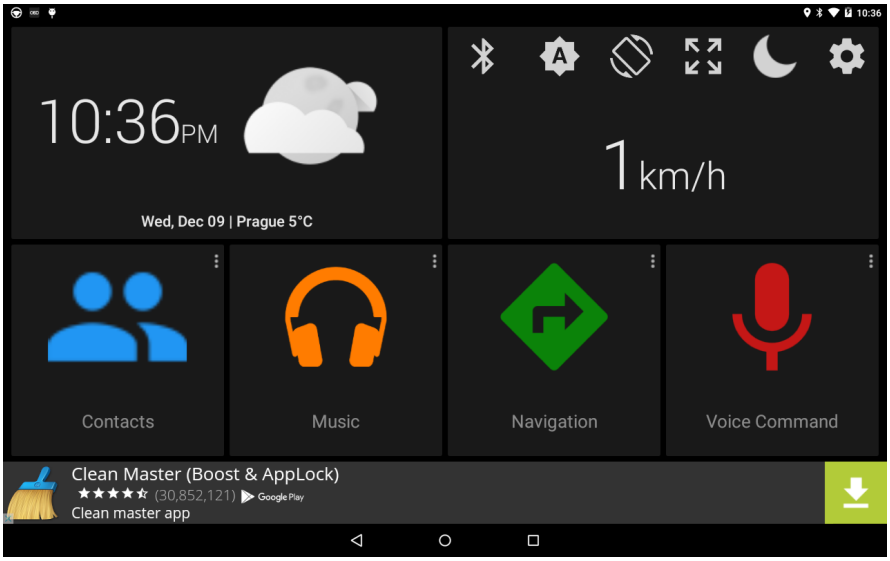


Figure 2.3. Screenshot from Car Dashdroid

2.1.4.1 Advantages

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

2.1.4.2 Disadvantages

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

2.1.5 Ultimate Car Dock

While the design is very similar to CarHome Ultra, this application 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 external applications.

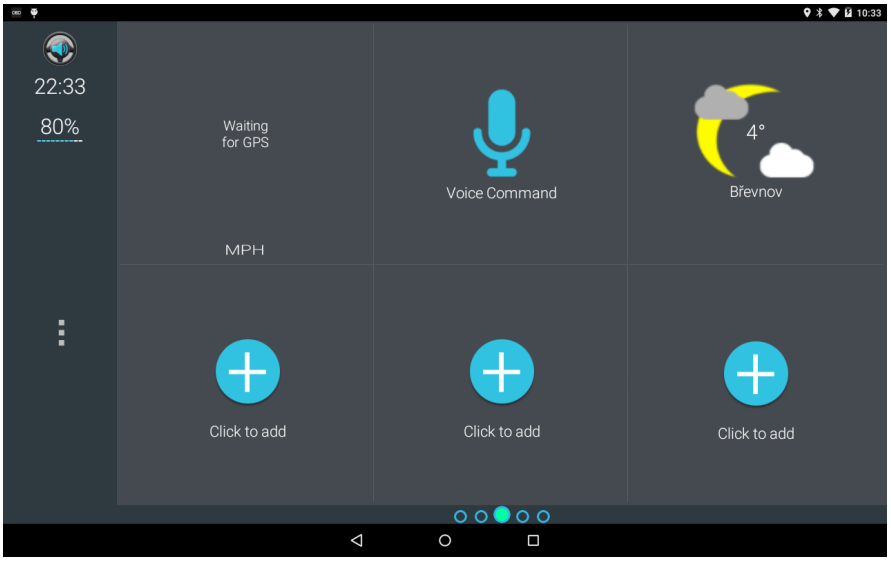


Figure 2.4. Screenshot from Ultimate Car Dock

2.1.5.1 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).

2.1.5.2 Disadvantages

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

2.1.6 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, voice command and shortcuts for external applications.

2.1.6.1 Suggestions

- OBD data,
- shortcuts to other applications,
- adjustable cards,
- built-in cards (weather, speed, voice command, etc.),
- simple grid UI,
- possibility to change displayed units,
- responsive and fluent,
- day and night theme,
- predefined message and call responses,
- TTS for incoming notifications.

2.1.6.2 Possible issues to avoid

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

2.1.7 Android Auto

sources:

- `http://developer.android.com/design/auto/index.html`
- `https://www.google.com/design/spec-auto/designing-for-android-auto/designing-for-cars.html`

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

- System overview
- Audio applications
- Messaging applications

2.1.7.1 System overview

System overview is supposed to be a home screen for 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, text and image, while following certain sizing rules. Every such notification can be expanded on the spot or another subapplication can be launched.

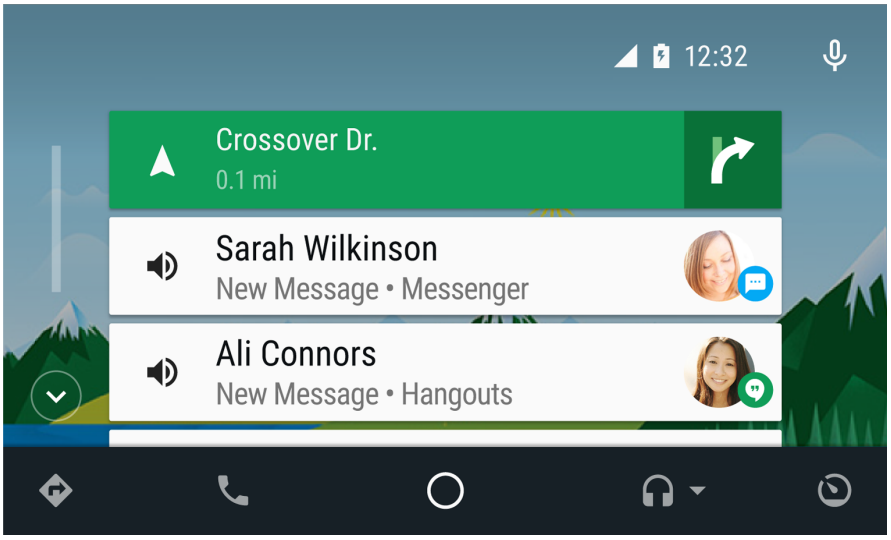


Figure 2.5. Android Auto Home screen

2.1.7.2 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.

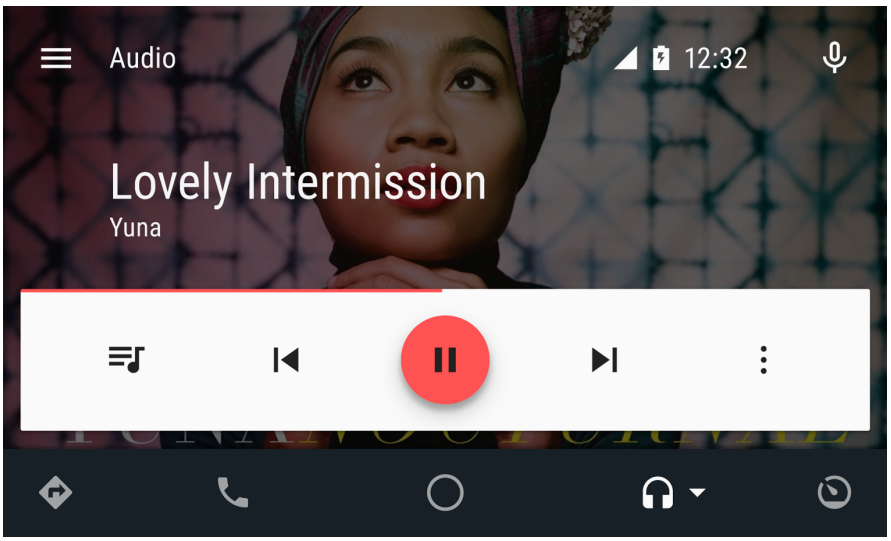


Figure 2.6. Android Auto audio application

2.1.7.3 Messaging applications

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

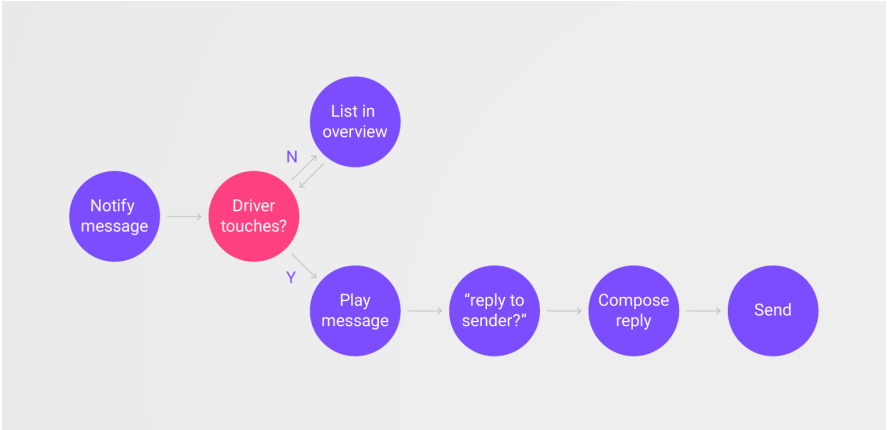


Figure 2.7. Android Auto conversational flow

2.1.7.4 Conclusion

It seems to be a good sign that even Google 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 as soon as possible.

2.2 Use-cases

Based on above. Also other typical analysis stuff.

2.3 Platforms

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

Following the first rule mentioned above and based on tablet sales in past years (sources: `http://techcrunch.com/2014/03/03/gartner-195m-tablets-sold-in-2013-android-grabs-top-spot-from-ipad-with-62-share/`), the only viable options for an application are platforms Android, iOS and Windows.

2.3.1 Android

In 2013, the Android platform had 61.9 % market share, making it the most used platform in the world. Targeting the Android platform would create large base of potential customers.

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

2.3.2 iOS

With 36 % market share in 2013, 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 too troublesome.

2.3.3 Windows

With only 2.1 % market share in 2014, the Windows platform does not seem to be a valid choise 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.

2.3.4 Conclusion

Fulfiling 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. As such, it will be analyzed more thoroughly later in this chapter.

2.4 Android platform TODO

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

2.4.1 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 adviced to hold the platform standards as the market demands it. The application must have a good look and feel in order to attract attention. This must be considered when building the application, because the environment demands fluent responses.

2.4.2 Architecture

The main building block in Android application is an activity. The activity is an idependent component of application, a hybrid between controller and 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).

For communication between activities there are so called Intents. An intent is the 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 allows to separate the actual code from the layout creation, which could help the GUI designed 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. Those resources are accessable from code using static class `R`, which is created during build time by most build system automatically.

As a relatively new concept, so called Fragment was created. It is similar to activity, however it is not a mandatory component. It can be used as a controller for 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 those 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.

2.4.3 Material design

Describe scrolling lists

2.5 GUI TODO

2.5.1 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 a proper UI must obey:

- **Clear** – it must be obvious what and where the user can control,

- **Effective** – minimizing the required user interactions for certain (requested) thing to happen,
- **Foolproof** - avoiding the errors before they happen,
- **Pleasant** - no stress when working with the UI, pleasant colors, contrast, 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 the 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 consistent visual and interaction language,
- **Integration** – using platform specific elements and rules
- **Simplicity** - allowing user to quickly learn how to use the UI

sources:

- MI-NUR `https://edux.fit.cvut.cz/courses/MI-NUR/lectures/start`
- Designing for indash automotive `http://revinity.com/?p=128`
- UX design stackexchange `http://ux.stackexchange.com/questions/51968/what-ux-guidelines-should-one-keep-in-mind-when-designing-the-gui-for-a-automobi` 0

2.5.2 UI in a car environment

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

2.5.2.1 Minimality

For minimizing the cognitive load, the must be as little information as possible at a certain time. The 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, minimizing the glance time.

2.5.2.2 Consistence

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

2.5.2.3 Readability

Good readability is one of the condition for application to be pleasant to use. In case of car environment, however, the readability of information is not just pleasant, but 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 font has to be large enough for every driver to see.

2.5.2.4 Controls

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

2.5.2.5 Colors

While in other environment the user can usually control the 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 visibility and readability.

2.5.2.6 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 the application is responsive, the user does not have to check the screen for progress so often or worse, wait for the progress looking at it continuously.

2.6 Development and support tools TODO

2.6.1 Development environment

While 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 AndroidStudio.

2.6.1.1 Eclipse

Based on research by (`http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/`), the Eclipse IDE 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 (`http://www.zdnet.com/article/google-releases-android-studio-kills-off-eclipse-adt-plugin/`).

2.6.1.2 Android Studio

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

2.6.2 Version control system

In a software development process, version is very important. 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 (`http://www.sitepoint.com/version-control-software-2014-what-options/`).

2.6.2.1 Subversion

Based on CVS, Subversion 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 destroyed without backup.

Because of the central repository, Subversion allows read and write access controls for a 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.

2.6.2.2 Mercurial TODO

2.6.2.3 Git TODO

2.6.3 Test driven development

Being one of the main development aproaches in the last decade, test driven development helps to develop the application quickly and fluently. The main idea of TDD is to create automatic tests before the actual application code. While this enforces the 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. (TODO JUnit?)

2.6.4 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 to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.“ (Martin Fowler, `http://www.martinfowler.com/articles/continuousIntegration.html`)

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

For simple CI practice integration, there are several online services. For this particular application development, Travis system was chosen (TODO link `https://travis-ci.org/`). While offering usual CI functionality, it also integrates easily with GitHub and Gradle build system.

2.7 On-Board Diagnostics TODO

On-Board Diagnostics states 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. The full specifications are explained in ISO 15031 (TODO fix). Some of them will be discussed later in the text.

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 suddenly.

2.7.1 API TODO

2.7.2 Data TODO

Chapter 3
Design TODO

3.1 Application architecture TODO

Designing proper application architecture is one of the main and most challanging tasks in development process. Changing the architecture in to future proves to be one of the most expensive changes as for manhours (TODO link Code Complete?). Application architecture influences data flow, communication amongst components and overall application performance, as well as extensibility and possibility to change or add features in the future. While Android application architecture enforces certain components and platform features to be used, the is still space for diversity.

3.1.1 Platform limitations

As mentioned in Android platform analysis in section ?? on page ??, the typical Android application consits of multiple Activities, which communicate with each other using Intents. While this approach supports the loosely coupled concept, it makes certain inter-cooperation rather difficult. Sharing an object between activities usually means serializing the object or saving it to the database, which means deserializing or loading from database later. When striving for excellent performance, this can emerge into a real 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 screen revalidation when such callback occurs. It is critical to avoid such delays as much as possible, when comes to car environment where rapid reactions are required.

3.1.2 Extensibility

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

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

3.1.3 Modularity TODO

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.“ (TODO andoid layoutinflater documentation and quotes)

3.1.3.2 Overview

Modularity concept allows application to contain certain modules, each offering certain functionality based on some predefined requirements.

3.1.4 Adaptability

Because the space on the screen is limited and also unknown (multiple devices have varying screen sizes) and every user might want to see 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. 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 adaptibility.

3.1.5 Android Auto preparation TODO

3.2 GUI TODO

Given the car environment, designing proper graphics 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 chapter (TODO link).

3.2.1 Basic elements TODO

Basic idea

3.2.2 Drafts TODO

3.2.3 The early phase TODO

In the early phase of the design process, the main idea was to display single piece of information as 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 on image (TODO link and image #1).

This concept was recreated into a similar concept with difference in size of previous and following application panels. Those panels would be moved into the background

which would make them smaller, as shown on image (TODO link and image #2), however, more of those 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.

For both drafts the following would apply. The swipe action would invoke text-to-speech action telling user the name of 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 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 subapplication are shown on images (TODO #3 #4) in the additional (TODO PRILOHA?). An early implementation is shown in the additional as well (TODO link).

3.2.3.1 Advantages

- **Readability** – given a single panel per screen with only a name and 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.

3.2.3.2 Disadvantages

- **Consistency** – however is the main screen consistent, the invoked subapplications 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 subapplications would have to be created independently everytime a new feature is implemented. This also limits easy extensibility, as creating a proper GUI is not a simple task with the given constraints.

3.2.4 The second phase TODO

Next step was to fix the problems mentioned above. Being inspired by the reviewed applications (TODO link to section) one attempt ended with concept shown on image (TODO link #5). It presents a vertical list of applications displayed in a column on the right side of the screen instead of horizontal list over whole screen. The main area contains the usual car data such as speed, rpm and consumption.

The second image (TODO link #6) shows possibility of inserting a subapplication screen between application list and car data, for example a navigation. Also, it presents the concept of micro-controls in application list. It would allow a single control button to be displayed on an application panel such as pausing the song or silencing the radio.

3.2.4.1 Advantages

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

3.2.4.2 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 lack of consistency, as every sub-application can have a different layout.

3.2.5 The grid

The next step towards consistency and space usage was to create a grid. This grid would be adjustable based on screen size, displaying the proper amount of application panels for given device. As shown on image (TODO link #7), it is just an extension of preveously shown vertical list making it vertical and horizontal – two dimensional.

Adding functionality to this grid, a new card concept emerged. It would consist of cards, which would provide additional information as well as control elements (as shown on image (TODO link #8)). They would be active demoversion of the full application, which would be invoked by touch to the upper area of the card.

3.2.5.1 Advantages

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

3.2.5.2 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 read

3.2.6 The final design

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

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

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

As shown on image (TODO link #11), 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.

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 2.4.3 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 good readability. As mentioned in section 2.5.2 on page 12, two modes should be present. While light mode offers good readability even in a direct sunlight, it blinds the driver during night time, as it emits too much light. Therefore it is a good idea to implement a dark mode as well for night time usage. For the best contrast possible, white on black or black on white are the best options.

3.2.6.1 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.

Describe the process, phases, analyse and compare advantages, disadvantages, thoughts

Chapter 4
Realization TODO

4.1 Preparation TODO

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 2.6.1 on page 13, 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 subfolder 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 subfolder into the `.gitignore` file.

4.1.3 Testing TODO

4.1.4 Scripting TODO

4.2 Tablet specific TODO

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 2.4.2 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 21, 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 acitivity 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 those modules to deactive 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 2.4.2 on page 11, it is nessessary 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 2.4.2 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 22) 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` is not suitable for this situation. As mentioned in section 3.2.6 on page 18, 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 3.2.6.

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 layouts all the provided modules.

4.3 Core TODO

The core contains all the functionality, it handles data, logic and also a standardized part of presentation, which consists of predefined single module veiws. 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 polymorfism. 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 those 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.

4.3.1.5 AbstractTimedUpdateDisplayModule

`AbstractTimedUpdateDisplayModule` serves as an extension to `AbstractDisplayModule` handling automatic timed updates. It uses advanced genericity 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 , 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 (TODO name) developed application 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 2.4.2, 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 those data and the simplicity of content (module type, name, additional data), a JSON format is used for data persistance. 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 those settings on a server, given the support of JSON format in web communication procotols.

To save and load those 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 ?? on page ??. 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 those 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 those 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 those tools are based on Java reflection API. For both of those 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 25.

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 particulary for Gmaps. It uses Google Maps API described in (TODO `https://developers.google.com/maps/`here). It allows creating the following:

- Displaying a location on map,
- launching a navigation to a certain location from the current location,
- searching the current location for 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 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 TODO

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 subpackage, 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 recieve it as quickly as possible.

Handling the OBD communication is done by background service which uses the (TODO OBD lib) library to send requests to the OBD and to recieve results. It sends requests based on tasks from a queue, where modules push their requests. This ensures that only currently needed information will be requeste, 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 27. 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 outloud. 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 recieve 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 TODO change code

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 TODO

Implementation of GUI is based in the final designed described in section 3.2.6 on page 18. While following the designed concept, also platform specific rules, as mentioned in section 2.4.3 on page 2.4.3, 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.

GUI implementation based on the design! Implementing modules, color, responsive effects

4.4.1 Common elements TODO

4.4.1.1 Colors

As described in section 3.2.6, 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 (2.4.3) 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 (2.4.3), several graphics effects are present in order to increase the overall atractivity. 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. (TODO IMAGES)

4.4.1.5 Icons

Given the platform guidelines (2.4.3), 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.

Hierarchical model, effects, submenus

4.4.2 Multiple designs TODO

As mentioned in section 3.2.6, there are two types of modules – an action module and a display module. Both of those 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.

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 backgrdoun of the opposite color to the font color.

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.

Limited set of module types

Chapter 5
Testing TODO

Brag about TDD, CI and Simulator!

5.1 Code TODO

Describe testing code, common testing (look&see, etc.)

5.1.1 Unit testing TODO

Unit testing on android, mention Test driven development, continuous integration, automatic tests, consider giving an example

5.1.2 Integration testing TODO

Instrumentation? Describe TDD, CI, automation

5.1.3 System testing TODO

Server testing, consider removing

5.1.4 Qualification testing TODO

Testing with users - consider section on its own - testing application as a whole thing

5.2 Heuristic testing TODO

Introduction, description

5.2.1 Evaluation TODO

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),
- missing edit button for shortcut modules (irreversible module addition operation).

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),
  - list controls (does not scroll fluently, but scrolls page by page),
  - navigation is not done by a navigation drawer, but rather a file-system like style,
  - measures do not match the standards (too large),
  - platform back button does not work immidietaly after changing the theme.

sources:

- `https://docs.google.com/document/d/1LAPqmYqe5LBE6vqWpi-rRYjHY1-zVPCzkFP2Gvh5i-Q/edit`

5.2.2 Conclusion TODO

Did not have time to fix

5.3 Testing with users TODO

5.3.1 Usability testing TODO

Testing the application as a regular application. Is it understandable? Is it easy to control, to see data, to understand, to comprehend, to learn?

5.3.2 Simulator TODO

Describe the car simulator in Albertov. DO NOT FORGET TO THANK THE DEPARTMENT OF DRIVING SMTHING, CVUT FD

5.3.3 Preparations TODO

5.3.3.1 Testing scenario

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 EyeT
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	CLT testing
01:05:00	01:10:00	5	Post-test questionnai
01:10:00	01:15:00	5	Debriefing

Selecting the world models and preparing them for testing, installing EyeTracker cameras, installing WebCamera, preparing data gathering, designing scenarios

5.3.4 Process TODO

The testing itself, describing participants

5.3.5 Evaluation TODO

Evaluating results

5.4 Summary TODO

Chapter 6
Conclusion TODO

6.1 Assignment completion TODO

6.2 Project life cycle TODO

6.2.1 Present TODO

6.2.2 Future TODO

6.3 Summary TODO