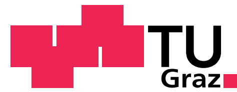

Magdalena Hackenberger, BSc

Development of a Modular Construction Kit Enabling End Users to Generate Android Apps for Real-Time Visualization and Manipulation of Vehicle Bus Data

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, July 2019

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Abstract

Automotive electronic systems have become an essential aspect in the design and construction of cars. The vast number of electrical control units (ECUs) in a modern car enable an abundance of features. However, they also cause an urgent need for good runtime testing and demonstration techniques. The AKKA group offers a system called *Display App*. The combination of an Android application and an ECU connected with the onboard vehicle bus provides high quality runtime visualization and the possibility of modifying ECU values.

Up until now, a new application needs to be programmed and released by AKKA for every car model. Furthermore, even minor adjustments in the user interface or functionality need to be done by AKKA in a separate release cycle. The winter test that are performed by vehicle manufacturers yearly, are limited to a short timespan of about two to three weeks. Performing unplanned adjustments quickly to the *Display Apps* during that time was not achievable until now.

The goal of this thesis is to provide a solution for this problem, enabling clients to build and modify simple *Display Apps* quickly and independently. This thesis describes the conceptionalization, design and implementation of a modular construction kit for the creation of *Display Apps*. Firstly, the state of the art of visual coding tools is analyzed. Secondly, the technical background of *Display Apps* is explained, including all necessary hardware and protocols. Thirdly, the development and implementation of the construction kit is described. The resulting system consists of two parts. The first part is a C#/.NET application named *AppBuilder* in which *Display Apps* can be designed from a set of preconfigured graphical user interface (GUI) elements without writing any code. These GUI elements can be linked to vehicle bus data. The second part is an Android application named *AppLoader* which constructs a *Display App* from a configuration file.

Kurzfassung

Elektronische Systeme wurden in den letzten Jahren beim Design und der Konstruktion von Fahrzeugen zum unverzichtbaren Werkzeug der Automobilindustrie. Die große Anzahl an ECUs (electronic control units) in einem modernen Auto ermöglicht eine Vielzahl an neuen Funktionalitäten, macht aber auch gutes Runtime Testing und gute Vorführfunktionen notwendig. *Display App* ist ein Produkt des Unternehmens AKKA Group. Bestehend aus einer Android Anwendung kombiniert mit einer an den Fahrzeugbus angebundenen ECU ermöglicht es hochqualitative Echtzeitvisualisierung sowie Modifikation der ECU Parameter.

In der Vergangenheit musste für jedes Automodell von AKKA eine neue Anwendung programmiert werden und selbst kleine Änderungen im User Interface oder der Funktionalität machten ein neues Release seitens AKKA notwendig. Die von Fahrzeugherstellern im Winter durchgeführten Tests sind zeitlich auf zwei bis drei Wochen beschränkt, bislang waren zeitgerechte Änderungen an den *Display Apps* in dieser Zeit kaum realisierbar.

Ziel dieser Masterarbeit ist es, eine Lösung für dieses Problem zu entwickeln. Kunden sollen einfache *Display Apps* schnell und selbstständig entwickeln können. Im Zuge der Arbeit wird Konzeptionierung, Design und Implementierung eines modularen Bausatzsystems für die Entwicklung von *Display Apps* beschrieben. Als erstes wird der Stand der Technik im Bereich von visuellen Coding Tools analysiert. Anschließend wird der technische Hintergrund der *Display Apps* inklusive der involvierten Hardware und Protokolle erklärt. Danach wird das Konzept und die Entwicklung des Bauchsatzsystems beschrieben. Das resultierende System besteht aus zwei Teilen. Der erste Teil ist ein C#/.NET Programm namens *AppBuilder*, mit dem *Display Apps* aus vorkonfigurierten graphischen Steuerelementen ohne Programmieren entworfen werden können. Die erstellten Steuerelemente können

mit Daten des Fahrzeugbusses verbunden werden. Der zweite Teil ist eine Android App namens *AppLoader*, die die entworfene *Display App* anhand einer Konfigurationsdatei anzeigt.

Contents

Abstract v

Kurzfassung vii

1 Introduction 1
- 1.1 Motivation 2
- 1.2 Aim and Objectives 3
- 1.3 Structure of the Thesis 3

2 Related Work 5
- 2.1 MIT App Inventor 5
  - 2.1.1 Design Editor 7
  - 2.1.2 Blocks Editor 8
- 2.2 AppyBuilder 9
  - 2.2.1 Design Editor 10
  - 2.2.2 Blocks Editor 10

3 Technologies, Frameworks and Development Methods 13
- 3.1 Test-Driven Development 13
- 3.2 Model View ViewModel (MVVM) Pattern 15
  - 3.2.1 Comparison with the Model View Controller (MVC) Pattern 16
  - 3.2.2 Windows Presentation Foundation (WPF) 18
  - 3.2.3 Prism 20
- 3.3 Android Application 22
  - 3.3.1 Kotlin 22
  - 3.3.2 Model View Presenter (MVP) Pattern 23
- 3.4 JSON 23

3.5 Communication between Car and Android Device 24
3.5.1 Technological Background 25
3.5.2 Generation of Firmware and A2L 27
3.5.3 Data Flow 28

4 AppBuilder (.NET Application) 31
4.1 Design 31
4.1.1 Use Cases 31
4.1.2 Graphical User Interface (GUI) 33
4.2 Implementation 35
4.2.1 Creation of a new project 36
4.2.2 Opening a Project 38
4.2.3 EditorMain 39
4.2.4 Controls 46
4.2.5 TextControl, BatteryControl 47
4.2.6 Radiobutton Field 48
4.2.7 Icon Button 48
4.2.8 Progress Bar 49
4.2.9 Adding and Replacing DBC and A2L Files 50
4.2.10 Export of a Project 51
4.3 Testing 52
4.4 Usability Inspection 57
4.4.1 Setup 57
4.4.2 Test Result 58
4.4.3 Refactoring 64

5 AppLoader (Android Application) 75
5.1 Design 76
5.1.1 Requirements 76
5.1.2 User Interface 78
5.2 Implementation 79

6 Summary and Future Work 85
6.1 Summary 85
6.2 Future Work 85

Bibliography 89

List of Figures

1.1 Functionality of *Display Apps* 2

2.1 *MIT App Inventor* Design Editor 8
2.2 *MIT App Inventor* Blocks Editor 9
2.3 *AppyBuilder* Design Editor 11
2.4 *AppyBuilder* Blocks Editor 11

3.1 Red-Green-Refactor approach 14
3.2 Model View ViewModel (MVVM) pattern 17
3.3 Model View Controller (MVC) pattern 17
3.4 The Model View Presenter (MVP) pattern 24
3.5 Process of creating a GIGABOX Beo firmware and an A2L file 27
3.6 Communication between car and tablet 29

4.1 First draft of the graphical user interface 34
4.2 Second and final draft of the graphical user interface 35
4.3 `CreateNewProject` View : Creation of a new project 36
4.4 `OpenProject` View : Opening an existing project 38
4.5 `EditorMain` View 39
4.6 Item class diagram 41
4.7 Ribbon 42
4.8 Visualization 43
4.9 Item Overview 46
4.10 Properties 47
4.11 Signal Selection 48
4.12 Configuration of a Radiobutton Field 49
4.13 Configuration of an Icon Button 50
4.14 Configuration of a Progress Bar 51
4.15 Project Options 51

4.16 Export of a Project 52
4.17 Identifying uncovered code parts with AcoCover 53
4.18 Overview over code coverage with the tool AxoCover 56
4.19 Old starting page 64
4.20 New starting page 65
4.21 Old project creation interface 66
4.22 New project creation interface 66
4.23 Old interface to open a project 67
4.24 New interface to open a project 67
4.25 Old placement of buttons on the ribbon 68
4.26 Refactoring of item class hierarchy 69
4.27 Old Property Grid 70
4.28 Old interface for signal configuration of text controls and battery controls 71
4.29 Old interface for configuration of radiobutton fields 72
4.30 New interface for configuration of radiobutton fields 73
4.31 Old way to confirm that the project is saved 73
4.32 New way to confirm that the project is saved 74
4.33 Old A2L File Replacement Warning 74
4.34 New A2L File Replacement Warning 74

5.1 First draft of the graphical user interface 75
5.2 First and final draft of the *AppLoader*'s graphical user interface 79
5.3 List of available projects 82
5.4 The loading screen is shown while the GUI is configured . . . 83
5.5 The configured application 84

1 Introduction

As of today, sophisticated electronic systems are a vital aspect of the design and construction of modern cars. The vast number of electrical control units (ECUs) enable an abundance of features but also poses major challenges.

Firstly, engineers must be able to monitor vehicle parameters in real-time in order to ensure correct behaviour during the testing stage. It must be possible to check how values change over time and how they react to changed parameters. This is only possible by accessing the vehicle bus. Reading raw values from a notebook computer in the car can be very inconvenient and confusing.

Secondly, certain non-visible functions and features can be difficult to showcase in a regular driving setting. Presenting results to superiors and customers can be challenging.

Lastly, the human machine interface (HMI) of a car limits possibilities during testing and presentation. Engineers need to test and assess functions and modes which are not configured in the HMI of the car. For example, it is not possible for the driver to deactivate driver-assistance systems like electronic stability control (ESC) and anti-lock braking system (ABS) in most modern cars due to safety reasons. Another example are different drive modes. In a typical car with automatic transmission, the driver can choose from three drive modes: eco, normal and sport. Test drivers need more drive mode configurations.

High quality visualization of vehicle data is the key to mastering these challenges. The AKKA group offers *Display Apps*, a family of applications for Android tablets that are specifically designed for visualization and manipulation of vehicle data. *Display Apps* are used in a setup with a specialized microcontroller named *GIGABOX Beo*, that is connected to the

vehicle's bus system (see section 3.5.1) as well as to a WLAN router. This setup is shown in Figure 1.1. Over WLAN, *Display Apps* can read and write data from and to the vehicle bus wirelessly and in real-time. Therefore, *Display Apps* offer detailed visualizations of vehicle data and processes like speedometers, progress bars, batteries, images and many more. Furthermore, they enable the user to control and parametrize vehicle functions.

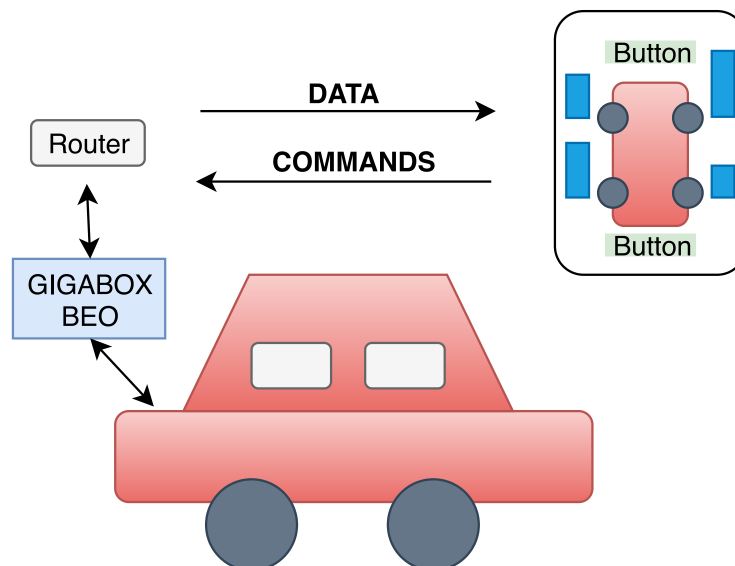

Figure 1.1: Functionality of *Display Apps*.

1.1 Motivation

Currently, every *Display App* has to be developed and adjusted individually for every vehicle. All changes in the application are carried out by AKKA Austria in a new release of the respective *Display App*. This workflow concerns bigger changes in functionality as well as slight changes of the user interface (UI).

While complex *Display Apps* with highly sophisticated functionality will still run through this development and release process in the future, an

alternative development process should be implemented for simpler applications. Clients should have the option to create simple *Display Apps* independently.

1.2 Aim and Objectives

To meet this requirement, a prototype for a modular construction kit consisting of two programs should be designed and implemented:

- UI Editor (C#/*.NET*) which exports a configuration file
- Android Application which imports the configuration file and builds the *Display App*

This prototype construction kit should enable the user to create a *Display App* without writing any code. To have good usability, the UI must be intuitive and easy to use for windows users. The thus created *Display Apps* should be capable of connecting to the vehicle and reading/writing data from/to the vehicle bus system based on the AKKA communication library. The user should be able to create intelligent controls by linking user interface elements to bus signals. The prototype should offer a small number of controls like a textbox and a button. It should be easily expandable for future further development.

1.3 Structure of the Thesis

This master thesis describes the background, the design and the development process of the *AppBuilder* prototype.

Chapter 2 introduces the reader to existing modular software construction kits and analyses their functionality, domain and potential similarities to the *AppBuilder*.

In chapter 3, all used technologies, programming languages, methods and frameworks are described. This includes background information, capabilities as well as the technological background of vehicle busses and automotive network protocols.

The development of the AppBuilder is discussed in chapter 4. It starts by describing the design process and the considerations which led to the finished design. The way from the first mock-up to the finished program is documented. The first and most important use cases are listed to show the planned development process.

In the subsection Implementation, the structure, architecture and implementation of the *AppBuilder* is described including screenshots of all features. Furthermore, it examines the conduction of a usability inspection as well as the test result and the changes which were subsequently made to the program. Also, the realization of test-driven development is discussed.

Chapter 5 describes the design process, requirements and implementation of the *AppLoader*.

Chapter 6 summarizes the main points of the thesis. Finally, an outline of future work and further development of the *AppBuilder* is given.

2 Related Work

Essentially, this project is a visual coding tool which enables users to create an Android application without writing actual code. The general idea is not a new one and there are already several tools on the market to achieve this goal. Müller et al. compared the most popular mobile app construction tools and summarized the key data as can be seen in Table 2.1. Müller et al. concluded that all of the compared tools enable users to create apps easily in a different way. There is no "best tool" but the preferred tool depends on requirements and personal preference (Müller et al., 2018-11). Both *Thunkable* and the *AppyBuilder* are originally based on the *MIT App Inventor*, which is an open source project (Cervantes, 2016) (AppyBuilder, 2019).

Although none of these tools meet the code requirements to build an *DisplayApp* due to the very specific domain, analysing them is a good starting point before entering the design stage of the *AppBuilder*.

In this chapter, two existing visual app construction system will be analysed. The *MIT App Inventor* was chosen because it is the first of its kind and many other tools are based on it. Also, it is a Browser/Desktop application, targets Android, is free and was developed in an academic context. The *AppyBuilder* was chosen because it is free, it is a Browser/Desktop application and it targets Android.

2.1 MIT App Inventor

The *MIT App Inventor* is an open source visual coding tool. While the first pilot version of the *MIT App Inventor* was released by Google in 2009,

Feature/App	MIT App Inventor	Thunkable	AppyBuilder	Sketchware	Pocket Code
Website	appinventor .mit.edu	thunkable .com	Appybuilder .com	Sketchware .io	Catrob.at/pc
Environment	Browser (Desktop)	Browser (Desktop)	Browser (Desktop)	Mobile	Mobile
App platforum	Android	Android, iOS	Android	Android	Android
Organization	academic	commercial	unknown	commercial	academic
Costs	free	most features free	free	most features free	free
Free Open Source	yes	no	no	no	yes
APK export	yes	yes	yes	yes	yes
Active Internet connection required	yes	yes	yes	no	no
Integrated sharing platforum	yes	no (thread in forum)	no (thread in forum	yes	yes

Table 2.1: Comparison of the most important visual coding tools on the market (Müller et al., 2018-11)

it has been developed and maintained by the Massachusetts Institute of Technology (MIT) since 2011. (Wolber, Abelson, and Friedman, 2015).

According to the website, the main motivation behind the development of the *MIT App Inventor* is to democratize software development and educate children in the field of computing. As of 2019, there are over 400,000 people actively using the *MIT App Inventor* per month. Users from 195 different countries have built nearly 22 million Android apps. (MITAppInventor, 2019)

The web application enables user to create an Android application combining drag and drop of visual elements with code blocks. It consists of two main parts: the **Design Editor** and the **Blocks Editor**.

2.1.1 Design Editor

The Design Editor (see Figure 2.1) allows the user to design the layout of the app by drag and drop. Available elements are listed in the ”palette” window in different categories. All added components are listed in the “Components” window. The selected component can be configured in the ”Properties” Window. The Design Editor is easy to use and its graphical user interface closely resembles the first draft vision of the *AppBuilder* user interface.

The extensive variety of preconfigured elements allows the user to easily build a highly functional, intelligent Android application. In addition to basic elements like buttons and text fields, there are many categories filled with sophisticated elements.

This is a list of categories in the Design Editor with three exemplary elements of each:

- *User Interface*, including the elements *Button*, *Checkbox* and *Spinner*.
- *Layout*, including *HorizontalAlignment*, *TableArrangement* and *VerticalScrollArrangement*.
- *Media*, including the elements *Camera*, *ImagePicker* and *SpeechRecognizer*.

- *Drawing and Animation* including the elements *Ball*, *Canvas* and *ImageSprite*.
- *Maps*, including the elements *Map*, *Marker* and *Circle*.
- *Sensors*, including the elements *AccelerometerSensor*, *BarcodeScanner* and *LocationSensor*.
- *Social*, including the elements *ContactFinder*, *Sharing* and *Twitter*.
- *Storage*, including the elements *File*, *TinyDB* and *TinyWebDB*.
- *Connectivity*, including the elements *BluetoothClient*, *BluetoothServer* and *Web*.
- *LEGO® MINDSTORMS®*, including the elements *NxtDrive*, *Ev3Motors* and *Ev3UI*.
- *Experimental*, including the elements *CloudDB* and *FirebaseDB*.
- *Extension* which enables the user to import extentions into the project.

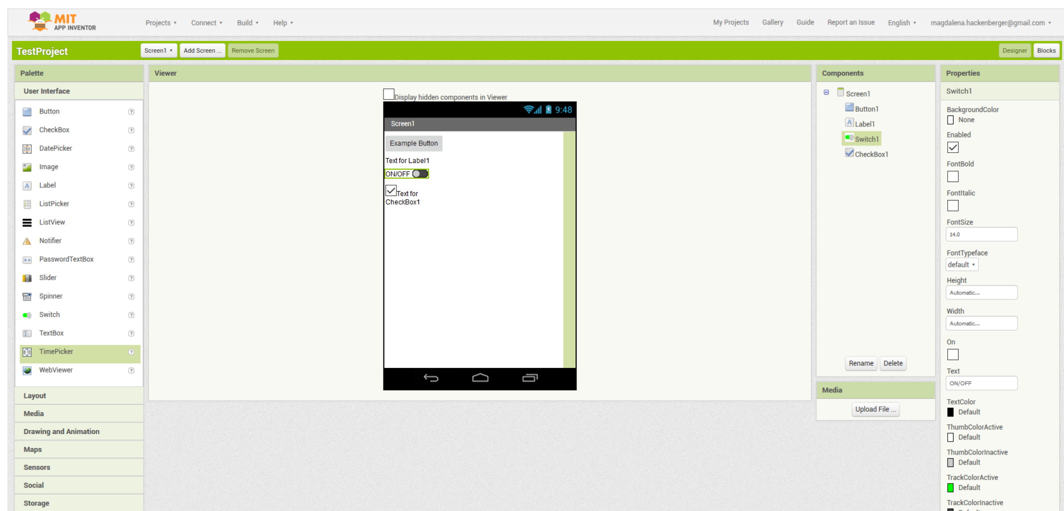

Figure 2.1: *MIT App Inventor* Design Editor (Screenshot created using MITAppInventor, 2019).

2.1.2 Blocks Editor

The components which were selected and placed in the Design Editor can be programmed in the Blocks Editor. Every component has a set of eligible blocks. Components can be parametrized, linked together and embedded

in a program logic by using control structures like loops or mathematical operations (see Figure 2.2).

Separate elements in DisplayApps are not connected to each other and the logic of elements does not exceed reading, writing and converting data from GIGABOX Beo. Therefore, the prototype won't include a feature analogous to the *MIT App Inventor*'s Blocks Editor.

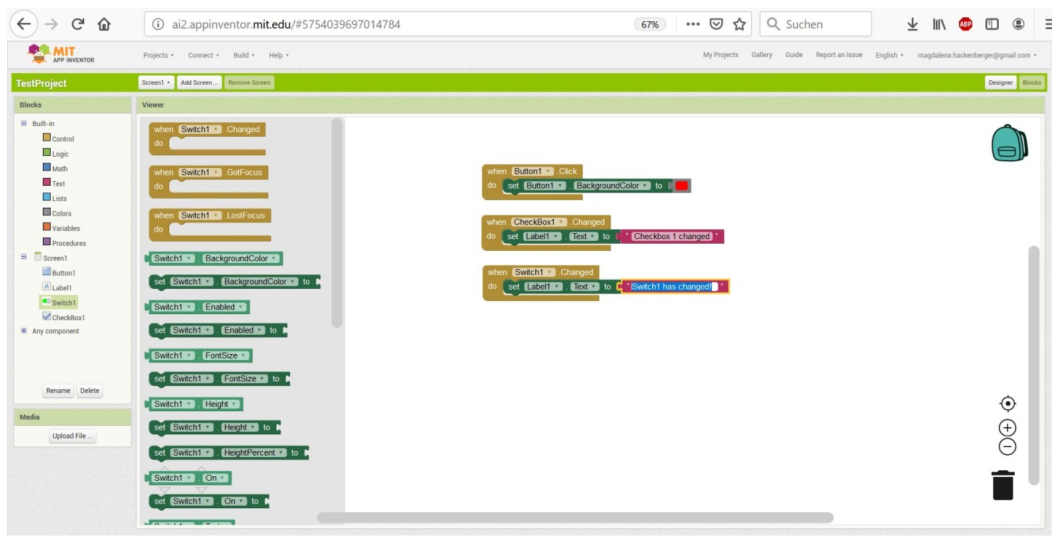

Figure 2.2: *MIT App Inventor* Blocks Editor (Screenshot created using MITAppInventor, 2019).

2.2 AppyBuilder

The *AppyBuilder* is a commercial visual coding tool which is based on the open-source tool *MIT App Inventor* (AppyBuilder, 2019). It combines all functionality of the *MIT App Inventor* projects with a big number of additional features built on top of it. Projects that were built with the *MIT App Inventor* can be imported to be further developed in the *AppyBuilder*.

Like the *MIT App Inventor*, the *AppyBuilder* consists of two main parts: the **Design Editor** and the **Blocks Editor**. Both parts will be described in the following subsections, especially in comparison to the *MIT App Inventor*.

2.2.1 Design Editor

As visible in Figure 2.3, the graphical user interfaces of the *AppyBuilder*'s Design Editor is almost identical to the graphical user interfaces of the *MIT App Inventor*'s Design Editor (see Figure 2.1). The *AppyBuilder*'s Design Editor has the same structure with a slightly different design, making them look nearly identical.

After briefly analysing the palette of possible elements on the left, it is evident that the *AppyBuilder* has significantly more features. While the *MIT App Inventor* has 12 categories of elements, the *AppyBuilder* has 16 categories - the same 12 categories and four additional ones: *Monetize*, *Advanced*, *Effects* and *Visualization*. The category *Monetize* features a variety of advertisement elements and InApp payment which adds the possibility to monetize the built application. The category *Advanced* includes features to add push notifications and SQL Lite .The category *Effects* includes additional navigation elements like SideBar and SnackBar. Elements in the category *Visualization* enable the user to create charts.

The difference in features is just as obvious within the categories: The *MIT App Inventor* has 15 possible elements in the category "User Interface". This category has 26 elements in the *AppyBuilder*, including chronometer, gallery viewer, grid view and many more. This pattern repeats itself through most categories.

2.2.2 Blocks Editor

As visible in Figure 2.4, the *AppyBuilder*'s Blocks Editor is almost identical to *MIT App Inventor*'s Blocks Editor (see Figure 2.2) as well. They have the same structure and functionality with minor differences in design. The only obvious distinction is that there is an increase in the number of elements analogous to the Design Editor.

Figure 2.3: *AppyBuilder* Design Editor (Screenshot created using AppyBuilder, 2019).

Figure 2.4: *AppyBuilder* Blocks Editor (Screenshot created using AppyBuilder, 2019).

3 Technologies, Frameworks and Development Methods

Several technologies, frameworks and methods are used for the development of the *AppBuilder*. This chapter explains the most important technologies and methods that are the basis of the two applications.

3.1 Test-Driven Development

Test-driven development (TDD) is a method in which programmers create tests before creating the code to be tested. It is most commonly used in agile software development, especially in the extreme programming (XP) method.

Kent Beck is the founder of the popular TDD method Red-Green-Refactor, where red and green correspond to fail and pass respectively (Beck, 2002). In this method, programming happens in small, repeated micro-iterations which should each only take a few minutes. It consists of the following phases (see Figure 3.1):

- **Red:** Writing a unit test, testing the functionality which is about to be implemented. This test must fail as the functionality is not programmed yet.
- **Green:** Change your program code by adding the new functionality with as little effort as possible. All tests should pass after this step.
- **Refactor:** Clean up the code and refactor it to meet coding conventions. Remove duplicate code, restructure hierarchies if necessary. New functionality must not be added in this step. After every modification

of code, all tests are re-run. At the end of this phase, the code should be clean and well-structured.

These three steps are repeated until the code contains all desired functionalities and all errors are eliminated. When these goals are reached, the coding unit is viewed as finished (Beck, 2002).

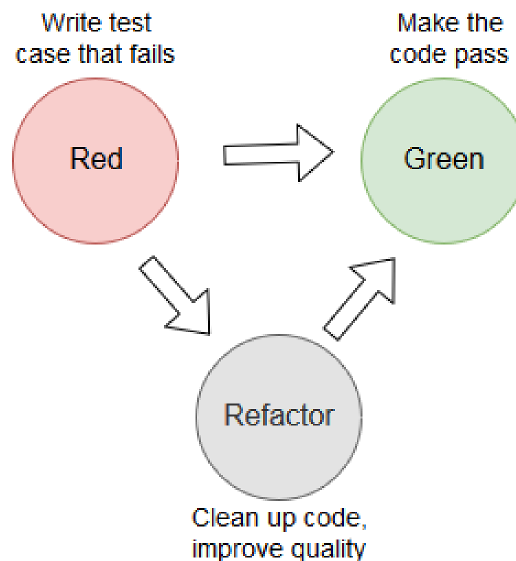

Figure 3.1: The Red-Green-Refactor approach.

Test-driven development has multiple advantages over traditional programming: Firstly, the programmer has to thoroughly think about all aspects of a method before they even start programming it. Secondly, every method of the program is fully covered by tests which can be re-run periodically. This secures that changes in code like new features or refactorings don't break existing features (Crispin, 2006).

A study about TDD in industry found that the quality of code was higher when it was developed with the TDD method, passing 18 percent more functional test than the control group which developed with a waterfall-like approach. Also, 79 percent of programmers thought that the TDD approach led to simpler design. and 71 percent perceived the method as effective. The TDD group needed 18 percent more time (George and Williams, 2003).

An article from 2018 compared and analysed five existing studies about the results of test-driven development on code quality and productivity. Four out of five studies found that TDD improved the quality of code, while the fifth study found improvement or no difference. Three of the five studies reported inconclusive results about the productivity with TDD, while one study found a degradation of productivity and one study found degradation or no difference. In summary, the article indicates that TDD does improve code quality while it mostly doesn't affect productivity and sometimes degrades it. (Karac and Turhan, 2018)

3.2 Model View ViewModel (MVVM) Pattern

The editor application is developed for the .NET platform, written in C#. *Windows Presentation Foundation (WPF)* is used for the graphical user interface. The application's underlying architectural design pattern is MVVM, implemented by the *Prism* framework. Firstly, the MVVM pattern will be explained including its motivation, its advantages and how the three core parts interact. Secondly, in subsection 3.2.1 the MVVM pattern will be compared with the widely known Model View Controller (MVC) pattern to highlight their differences. Subsequently, the *WPF* library and its important role in MVVM will be described in subsection 3.2.2. Finally, the *Prism* framework and its role implementing the MVVM pattern will be explained in subsection 3.2.3.

In classical C# *.NET* applications, the business logic (Model) and the user interface (View) are usually tightly coupled. This can lead to considerable additional expenses in maintenance when such applications are modified or when they grow in scope or size. Also, designers need a programmer in order to be able to implement changes in the graphical UI. Finally, unit testing is a big challenge when business logic and UI are this closely intertwined (Microsoft, 2019b).

By using the MVVM pattern, these problems are avoided. Firstly, it separates concerns: Model and View are decoupled. Thus, UI modifications can be easily done without a software engineer. Additionally, separating the

UI from the business logic boosts testability and maintainability of an application (Microsoft, 2019b).

The three components of the MVVM pattern are the Model, View and ViewModel. The relationship between the three components is shown in Figure 3.2. The role of each component will be briefly described in the next paragraphs.

The **Model** is responsible for the application's business logic. It represents all data that is shown to and manipulated by the user. Furthermore, it validates data entered by the user and notifies View and ViewModel about changes in data. The data source can be anything: for example a relational database or the local file system (Kühnel, 2013a).

The **View** is responsible for the graphical user interface (GUI), including the structure, the look and the layout of all displayed elements. It is connected to the ViewModel via Data Binding (see subsection 3.2.2). This loosely coupled connection makes it possible to keep code behind[1] to a minimum and to switch Views without altering the ViewModel. Controls can be bound to properties of the ViewModel to present data and manipulate data per user input. The classic event based notifications are replaced by a command mechanism to keep View and ViewModel decoupled (Kühnel, 2013a).

The **ViewModel** is the central part of the pattern and serves as a link between Model and View. It is responsible for the UI logic, i.e. it is the Model of the View. The ViewModel exchanges information with the Model to acquire and manipulate data. It doesn't have any information about the View but it implements an observable interface and provides properties to which the View can bind (Kühnel, 2013a).

3.2.1 Comparison with the Model View Controller (MVC) Pattern

The main intent of the MVC pattern is, like MVVM, the separation of the application's concerns. The three components of the MVVM pattern are

[1]"Code-behind is a term used to describe the code that is joined with markup-defined objects, when a XAML page is markup-compiled." (Microsoft, 2019a).

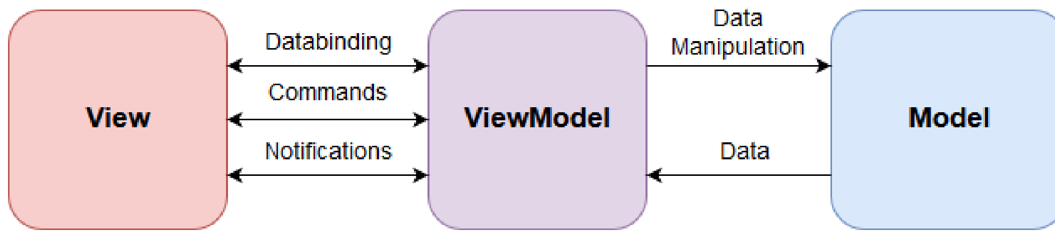

Figure 3.2: The Model View ViewModel (MVVM) pattern.

Model, View and Controller. The relationship between the three components is shown in Figure 3.3. Like in the MVVM pattern, the Model represents the application's business logic and the View represents the UI. The Controller manipulates the data in the Model and updates the View upon data changes.

The main difference in contrast to the MVVM pattern is that in MVVM, the View does not get updated data directly from the Model. All dataflow passes through the ViewModel. Also, the ViewModel has no information about the View. Exchange of data and user input exclusively happens by data binding: The ViewModel has public properties to which the View can bind (Syromiatnikov and Weyns, 2014).

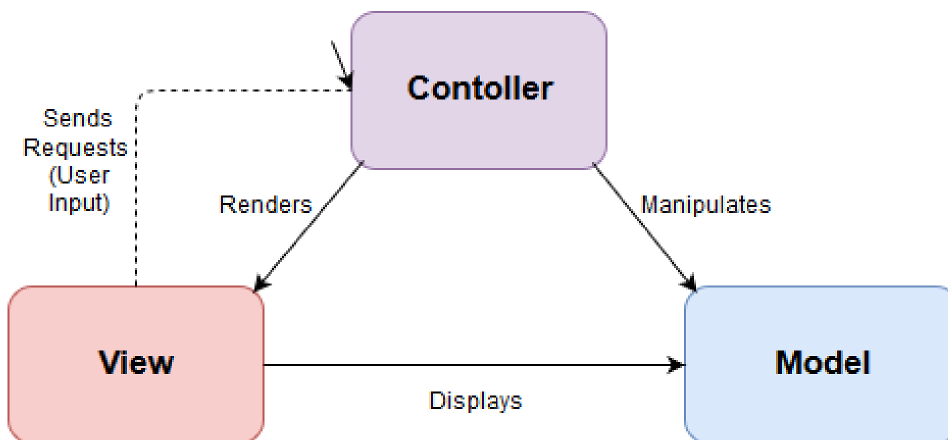

Figure 3.3: The Model View Controller (MVC) pattern.

3.2.2 Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) is a graphic user interface (GUI) framework and class library of the Microsoft *.NET* framework. It is a more modern alternative to *WinForms*, the previous standard GUI framework of *.NET*. *WPF* was first released in 2006 with *.NET* version 3.0 and is deployed with Windows since *Windows Vista*. With *WPF* it is possible to create user interfaces in a purely declarative way. The markup language used for this purpose is called eXtensible Application Markup language (XAML), which is based on *XML*. New features include integration of *DirectX* for hardware acceleration, mutimedia component support via *Windows Media*, animations and data binding.

Data binding is one of the most important new features of *WPF*. With this concept, developers are able to follow the principle of separation of concerns and to implement the MVVM pattern (section 3.2). Data binding is the automated transmission of data between objects, typically between a data object and a user control. In *WPF*, these objects are called *dependency properties*: Their value depends on the other object. The following paragraphs and code snippets explain and demonstrate *WPF* data binding with dependency properties.

There are two types of data binding: one-way data binding and two-way data binding. In one-way data binding, data is bound from the data source to the target property.

In an example, code 3.1 is the XAML file (View) the data context of the Window in our example is set to the ViewModel 3.2. The property "Text" in the `TextBox` is a dependency property which defines the displayed text of the `TextBox`. It is bound to the property "ProjectName" in the ViewModel (3.2) via one-way binding. The value of the property `ProjectName` is set to "New Project". When this application is run, the `TextBox` displays the text "New Project" as it is bound to a data source with this value. When the text is changed to "My Project" by the user in the UI, the data source remains unchanged as the binding only goes in one direction. The value of the property `ProjectName` is still "New Project".

```
<Window x:Class="ExampleProject.MainWindow"
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:VM="clr-namespace:ExampleProject.ViewModels">

  <Window.DataContext>
    <VM:CreateProjectViewModel />
  </Window.DataContext>

  <TextBlock Text="Project Name:"/>
  <TextBox Width="280" Text="{Binding ProjectName, Mode = OneWay}"/>
</Window>
```

Listing 3.1: One-way binding: Label and Textbox in the XAML file.

```
namespace ExampleProject.ViewModels
{
  public class CreateNewProjectViewModel
  {
    private string _projectName = "New Project";

    public string ProjectName
    {
      get => _projectName;
      set => SetProperty(ref _projectName, value);
    }
  }
}
```

Listing 3.2: ProjectName property in the ViewModel.

In two-way data binding, data that is edited by the user in the UI is also changed in the data source. A slightly different XAML example (3.3) is set to the same data context, ViewModel (3.2). The dependency property "Text" of the `TextBox` is again bound to the property "ProjectName" in the ViewModel (3.2), this time via two-way binding. Again, the `TextBox` would display the value of the data source: "New Project". When it is changed to "My Project" by the user, the data source is modified to this new value as well.

```
<Window x:Class="ExampleProject.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:VM="clr-namespace:ExampleProject.ViewModels">

  <Window.DataContext>
```

```
    <VM:CreateProjectViewModel />
  </Window.DataContext>

  <TextBlock Text="Project Name:"/>
  <TextBox Width="280" Text="{Binding ProjectName, Mode = TwoWay}"/>
</Window>
```

Listing 3.3: Two-way binding: Label and Textbox in the XAML file.

(Kühnel, 2013b)

3.2.3 Prism

Prism is the framework and library which is used as the foundation of the *AppBuilder* application. It provides an implementation of MVVM, navigation between Views, dependency injection and command handling. The following paragraphs give an overview over the most important features of *Prism* used in this project.

ViewModelLocator

The *Prism* ViewModelLocator sets the data context of a View automatically to a ViewModel according to their assembly and names. By using the ViewModelLocator, the data context doesn't have to be set explicitly like in the XAML snippets before (3.1, 3.3). The convention for automatic Viewmodel-resolving/loading is as following: The View and the ViewModel need to be in the same assembly and in the .View and .ViewModel namespaces respectively. Also, the ViewModel needs to have the name as the View, with the suffix "ViewModel" (Prism, 2019b).

In the XAML snippet 3.4, the data context of the window is set to the ViewModel from the section above(3.2) via ViewModelLocator. As one can see, the assembly, folder and naming conventions are met and the XAML is significantly smaller. This *Prism* feature makes it more natural and convenient to implement the MVVM pattern.

```
<Window x:Class="ExampleProject.Views.CreateProject"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:VM="clr-namespace:ExampleProject.ViewModels"
     prism:ViewModelLocator.AutoWireViewModel="True">

<TextBlock Text="Project Name:"/>
<TextBox Width="280" Text="{Binding ProjectName, Mode = TwoWay}"/>
</Window>
```

Listing 3.4: Setting the DataContext with the Prism ViewModelLocator.

Commanding

Additionally to displaying and editing information by data binding as described in chapter subsection 3.2.2, binding can also be used to let the user trigger actions in the ViewModel. Instead of using events like in *WinForms*, *WPF* has introduced commands. The difference to events is that commands are one-to-one connections between a UI control and a Command Object in the ViewModel rather than being sent out to various listeners. Command objects are objects that implement the *WPF* `ICommand` interface. Commands are not handled in code behind like events but implemented directly in the ViewModel.

While the `ICommand` interface was introduced by *WPF*, *Prism* provides a very convenient implementation called `DelegateCommand` (Prism, 2019a). XAML code 3.5 shows how a UI control is bound to a `DelegateCommand`. Code 3.6 shows the Command Object property and the code it encapsulates in the connected ViewModel.

```
<Window x:Class="ExampleProject.Views.CreateProject"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     xmlns:VM="clr-namespace:ExampleProject.ViewModels"
     prism:ViewModelLocator.AutoWireViewModel="True">

 <TextBlock Text="Project Name:"/>
 <TextBox Width="280" Text="{Binding ProjectName, Mode = TwoWay}"
         Command="{Binding CreateCommand}"/>
</Window>
```

Listing 3.5: Binding a button to a Prism `DelegateCommand`.

```
namespace ExampleProject.ViewModels
{
  public class CreateNewProjectViewModel
  {
    private string _projectName = "New Project";

    public CreateNewProjectViewModel()
    {
      CreateCommand = new DelegateCommand(Create);
    }

    public string ProjectName { get => _projectName;
                                set => SetProperty(ref _projectName, value); }

    public DelegateCommand CreateCommand { get; private set; }

    private void Create(object parameter)
    {
      //do something
    }
  }
}
```

Listing 3.6: Prism DelegateCommand and Prism Regions in the ViewModel.

3.3 Android Application

The Android Application is programmed in the language Kotlin and uses the Model View Presenter (MVP) pattern.

3.3.1 Kotlin

Kotlin is a modern statically typed, object-oriented programming language which is translated to bytecode and runs on the *Java Virtual Machine (JVM)*. It is mainly developed by the Czech company JetBrains and was named after an island near St. Petersburg, one of the locations of the company.

Kotlin has been developed since 2011 but was not officially released until February 2016 (Lardinois, 2019).

Because the code is translated to bytecode, Kotlin has full interoperability with Java code and depends on the Java Class Library. Although it is a young language, it is already highly popular. Android Studio has fully supported Kotlin since version 3.0 and since 17.05.2019, Kotlin has been Google's preferred language for the development of Android Apps (Lardinois, 2019).

3.3.2 Model View Presenter (MVP) Pattern

Testability is a priority in the development of the AppBuilder. Unit testing is very difficult when business logic and UI are tightly coupled. To avoid this, the implementation of the Android application follows the MVP pattern.

The main intent of the MVP pattern is the same as in MVVM (section 3.2) and MVC (subsection 3.2.1): The separation of an application's concerns. MVP consists of the tree components Model, View and Presenter. The relationship of these components is shown in Figure 3.4. Like in MVC and MVVM, the Model represents the application's business logic and the view represents the UI. The presenter is the link between Model and View. It manipulates the data in the Model and updates the View upon data changes.

MVP is very similar to MVVM: Model and View are decoupled as the View is updated from the presenter, not from the Model. The main difference between the presenter in MVP and the ViewModel in MVVM is that there is no data binding in MVP. View and Presenter communicate directly.

3.4 JSON

JavaScript Object Notation (JSON) is used to serialize and deserialize projects that were created with the AppBuilder. All relevant project data is serialized

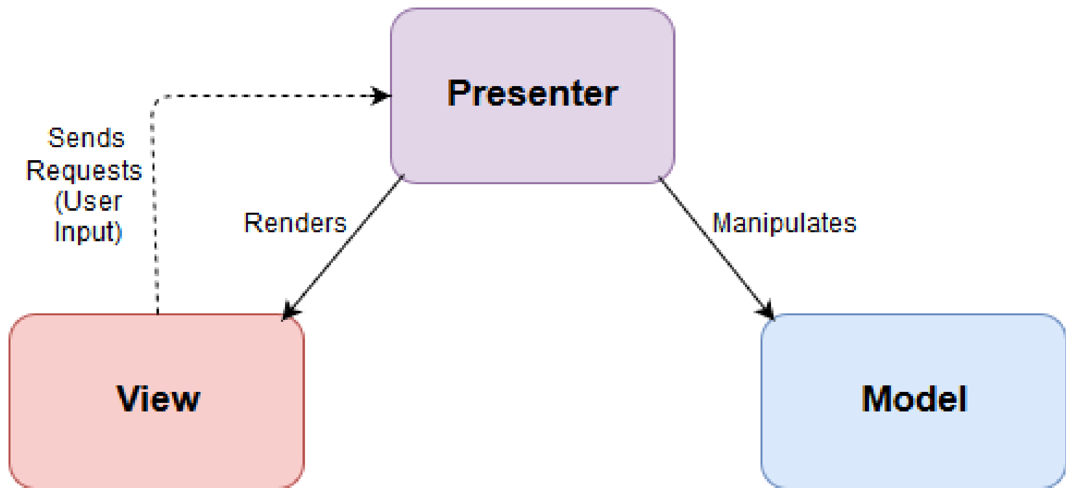

Figure 3.4: The Model View Presenter (MVP) pattern.

to a *JSON* file, copied to the Android device and deserialized by the Android application.

JSON is a human-readable compact data format for data exchange (serialization) between applications. It is most commonly used to exchange structured data between client and server in mobile and web applications. Although every valid JSON document is also a valid JavaScript, JSON is programming-language-independent as parsers exist in all popular programming languages. This quality makes it the ideal data exchange format for the *AppBuilder*, as there are parsing libraries both in both C# and Kotlin/Java.

3.5 Communication between Car and Android Device

As mentioned in the introduction, AKKA Display Apps are specifically designed for visualization and manipulation of vehicle data. This chapter describes all involved hardware, protocols and file formats. Subsequently, the development, setup and operation of a Display App is explained in detail.

3.5.1 Technological Background

GIGABOX Beo

GIGABOX Beo is an electronic control unit (ECU) which is specifically designed to enable bi-directional data exchange. It is equipped to exchange data via various automotive bus systems including CAN, FlexRay and LIN. In this thesis, only the CAN bus is used for data exchange (AKKADigital, 2013).

GIGABOX Beo comes with extensive basic software and an API interface. Complex application modules can be implemented with either C or *MATLAB/Simulink*. The application modules which are used in this thesis are typically implemented with *MATLAB/Simulink* (AKKADigital, 2013).

CAN Bus

CAN (Controller Area network) is a serial bus standard. It was developed by the company Bosch in 1983 and presented in cooperation with Intel three years later. It is widely used as a vehicle bus in the automotive industry. The intent of CAN was to add functionality and to reduce the amount of cables in cars in order to minimize weight and costs. (CiA, 2019) CAN is ISO-11898 certified. It defines the physical layer (layer 1) and the data link layer (layer 2) of the ISO/OSI model. (ISO, 2015)

Its design allows ECUs to communicate in real-time with the multi-master principle, i.e. all ECUs have master functions and are equal in this regard. The priority of messages are based on an identifier. (Bosch, 1991)

DBC File Format

DBC is a file format by the company Vector which describes the communication structure of a CAN network. It has become the de facto standard descriptive file format for CAN (Vector, 2019). The DBC file contains information about all participants as well as all messages and signals of a CAN

bus. This information can be used to analyse the network and simulate ECUs which are not physically available (Vector, 2007).

Universal Measurement and Calibration Protocol (XCP)

XCP is a master-slave communication protocol which was first standardized by the Association for Standardization of Automation and Measuring Systems (ASAM) in 2003. As a two-layer protocol, it separates the protocol and transport layer from each other. The "X" represents the variable transport layer which can be implemented with a number of different protocols including CAN, Ethernet and FlexRay among others (ASAM, 2017).

XCP is mainly used in the development of automotive electronics development, to test electronic control units (ECUs) and calibrate parameters. It enables read and write access to internal variables of ECUs at runtime (ASAM, 2017).

The normal process of XCP communication is that a request is sent by the master (*Display App*) and an answer is sent by the slave (*GIGABOX Beo*). Since a request must be sent separately for every variable repeatedly, this leads to performance problems when there is a high number of requested variables. To solve this problem, there is a measuring method called data acquisition (DAQ). The master (*Display App*) sends the slave (*GIGABOX Beo*) a list of variables. Subsequently, these variables are automatically sent to the master in a defined frequency. As they are sent in bulk (big frames), high numbers of variables can be inquired in real-time (Andreas Patzer, 2016).

A2L File Format

A2L files are descriptive files which contain information about the generated ECU firmware. The most important information for the Display Apps is a list of all publicly available variables of the firmware. These variables are categorised into measurement and characteristics, read-only and mutable values respectively. Every individual definition consists of its variable category, name, memory address and data type. This enables the Display App to access *GIGABOX Beo*'s internal variables by their symbolic names.

The snippet below shows a small part of an A2L file, describing a measurement named `u_EXAMPLE_MEASUREMENT_CH_A_sig` and a characteristic named `y_EXAMPLE_MEASUREMENT_1_sig`.

```
/begin MEASUREMENT u_EXAMPLE_MEASUREMENT_CH_A_sig ""
  UBYTE NO_COMPU_METHOD 0 0 0 255
  ECU_ADDRESS 0xFEDE58EF
  ECU_ADDRESS_EXTENSION 0x0
  FORMAT "%.15"
  /begin IF_DATA CANAPE_EXT
    100
    LINK_MAP "u_EXAMPLE_MEASUREMENT_CH_A_sig" 0xFEDE58EF 0x0 0 0x0 1 0x87 0x0
    DISPLAY 0 0 255
  /end IF_DATA
  SYMBOL_LINK "u_EXAMPLE_MEASUREMENT_CH_A_sig" 0
/end MEASUREMENT

/begin CHARACTERISTIC y_EXAMPLE_MEASUREMENT_1_sig ""
  VALUE 0xFEDE4EAA __UBYTE_S 0 NO_COMPU_METHOD 0 255
  ECU_ADDRESS_EXTENSION 0x0
  EXTENDED_LIMITS 0 255
  FORMAT "%.15"
  /begin IF_DATA CANAPE_EXT
    100
    LINK_MAP "y_EXAMPLE_MEASUREMENT_1_sig" 0xFEDE4EAA 0x0 0 0x0 1 0x87 0x0
    DISPLAY 0 0 255
  /end IF_DATA
SYMBOL_LINK "y_EXAMPLE_MEASUREMENT_1_sig" 0
/end CHARACTERISTIC
```

Listing 3.7: A small part of an A2L file describing a measurement and a characteristic.

3.5.2 Generation of Firmware and A2L

Figure 3.5 shows the process of creating a firmware for *GIGBAOX Beo* and generating an A2L file.

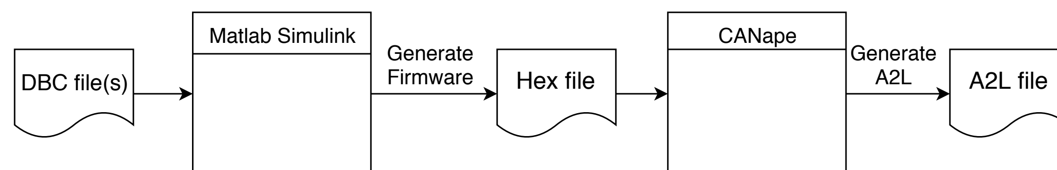

Figure 3.5: The process of creating a GIGABOX Beo firmware and an A2L file.

One or more DBC files containing information about the CAN bus(ses) of the car are loaded into Matlab/Simulink. A firmware for *GIGABOX Beo* is modelled with blocks that were specifically designed by AKKA. The firmware is generated based on this model. The firmware is loaded into the Vector program *CANape* where an A2L file is generated. With information of this A2L, the Android application knows which variables are on which memory addresses.

3.5.3 Data Flow

Figure 3.6 shows the data and command flow between the car and the Android tablet in detail. *GIGABOX Beo* is connected to the onboard CAN of the car and to a router. *GIGABOX Beo* stores firmware-defined CAN data in its memory. Since the A2L file maps memory addresses to symbolic names (measurements and characteristics), the Android application knows the relevant memory addresses

The router and the Android device are connected via wireless LAN and communicate with the protocol XCP. The Android device requests data (measurements) from the *GIGABOX Beo* with XCP, using the DAQ measurement method (see section 3.5.1). The Android device sends commands (characteristics) with XCP to *GIGABOX Beo*.

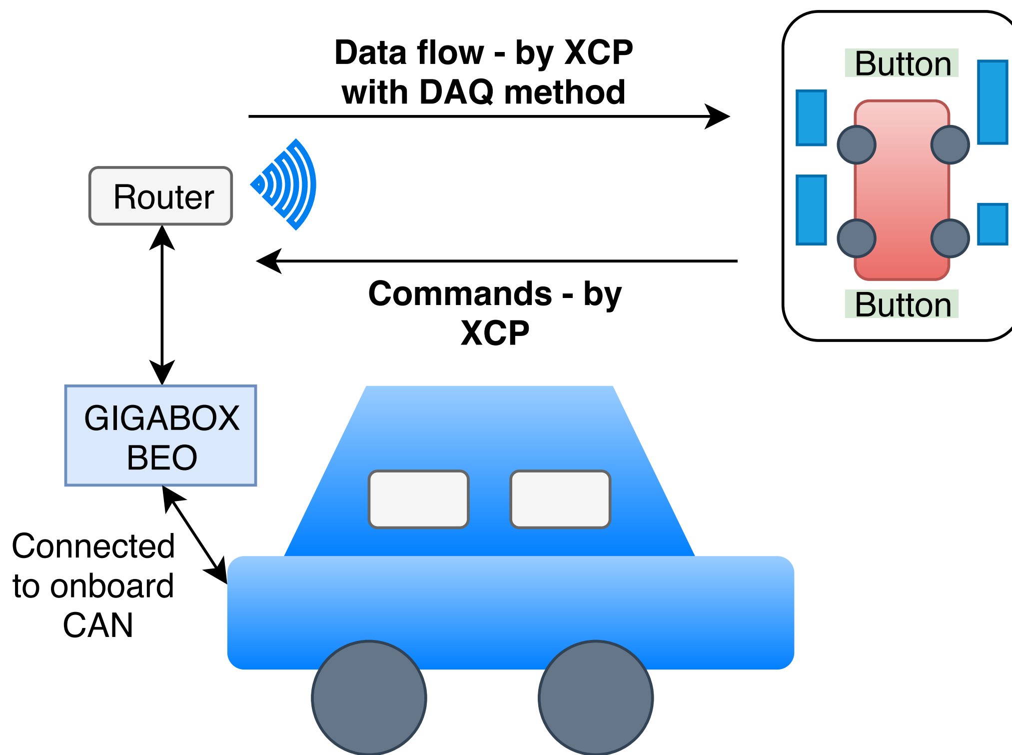

Figure 3.6: Communication between car and tablet.

4 AppBuilder (.NET Application)

This chapter describes the development of the AppBuilder which enables users to design, configure and modify *DisplayApps* without programming.

4.1 Design

One of the first design decisions to be made was the order in which the two parts of the project would be designed and implemented. It was decided that the *AppBuilder* (*.NET*) and the *AppLoader* (Android) would be designed and implemented in parallel in order to avert the risk of compatibility issues at a later stage development.

4.1.1 Use Cases

The following use cases were specified as requirements for the *AppBuilder* together with AKKA Austria. Each use case consist of a description and a list of acceptance criteria.

Create and save project

A new project can be created by the user. A name and several dimensions of the target platform (the tablet) are enterd by the user. Theses parameters are:

- Height in pixel
- Width in pixel

Acceptance criteria are:

- Blank canvas with the dimensions entered for the project is shown.
- A project file is created containing the parameters entered on project creation.
- The project file is in a human readable and editable format.

Add image

An image file can be selected from the hard disk and placed on the canvas. The image can be moved and resized by. When the project is saved, the information entered by the user as well as the correct filename is saved to the project file.

Acceptance criteria are:

- An image can be placed onto the canvas of the graphical UI.
- The image can be moved and sized (x, y coordinates, width, height).
- The image is displayed on the canvas with the correct dimensions entered by the user.
- The image can be an abstract representation.
- The image meta data is saved to the project. file

Add text field

An new text field can be places on the canvas. The user can enters the memory address of the variable which should be displayed in this text field.

Acceptance criteria are:

- A text field can be placed onto the canvas.
- The text field is displayed as an abstract representation with the correct dimensions on the canvas.
- The text field meta data is saved to the project file entered by the user.

Add button

A button can be placed on the canvas by the user. Subsequently, a text must be entered, which is then displayed as the button label. Furthermore, he must enter the memory address and a fixed value which is written to this address when the button is pressed.

Acceptance criteria are:

- A button can be placed onto the canvas
- A label text can be assigned to the button
- The button can be an abstract representation with the correct dimensions
- The correct meta information for the button is written to the project file

Reconfigure component

The user wants to adapt the parameters of a components which has already been placed on the canvas. The possibility to select the correct component must be available. After selecting the component, the user may enter new parameters according to the selected component. Acceptance criteria are:

- A component on the canvas can be selected by the user.
- The parameters for this component get displayed and can be edited.
- The changes can be saved.

4.1.2 Graphical User Interface (GUI)

Figure 4.1 shows the first mock-up of the GUI. In the original design, controls would be dragged and dropped into the visualization screen. To link signals to items, they would be dragged and dropped on the controls. All visual configurations of items would be done directly in the visualization.

For several reasons, the planned GUI was altered before implementation. Firstly, the GUI was too simple for more complex tasks and controls. Signals need to be categorized into measurements and characteristics as not all

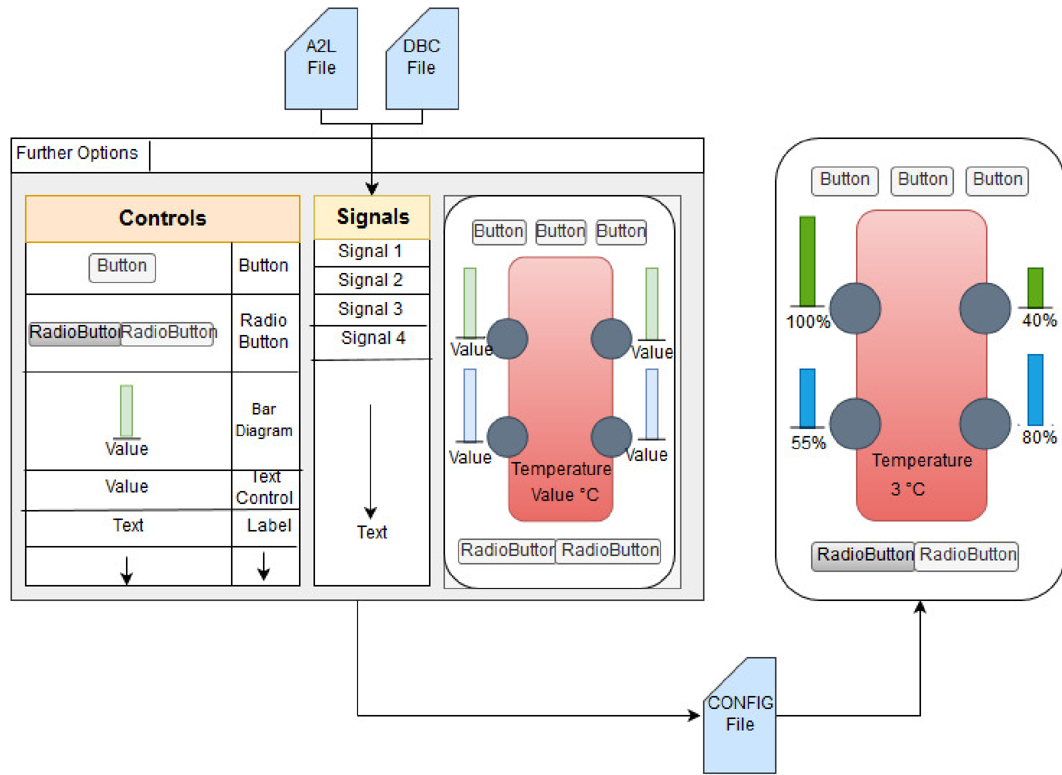

Figure 4.1: First draft of the graphical user interface.

controls can be linked to both. Secondly, an easily accessible interface for the exact configuration of size and position of items is necessary to assure a precise alignment of items. Finally, the interface provided no information about the currently selected item and its signal configuration.

Figure 4.2 shows the second and final draft of the UI. In this GUI, a ribbon serves as the central control element. The ribbon was chosen because of its wide spread use in current versions of Microsoft Windows which makes it intuitive for the potential users. Creation of items, management of files and all other central project options are located on the ribbon. Items can be selected in the visualization or in the object overview panel which lists all items per type or, if available, per name. Subsequently, most properties of selected item can be seen and configured in the Properties panel. For example in Figure 4.2, the selected item is a text control which displays

the torque of the wheel on the front left of the car. Properties like name, position, size, font size, font colour et cetera can be configured directly in the property panel. More complex configurations are carried out in a separate dialog window which is opened by pressing the "Edit Control" button on the ribbon.

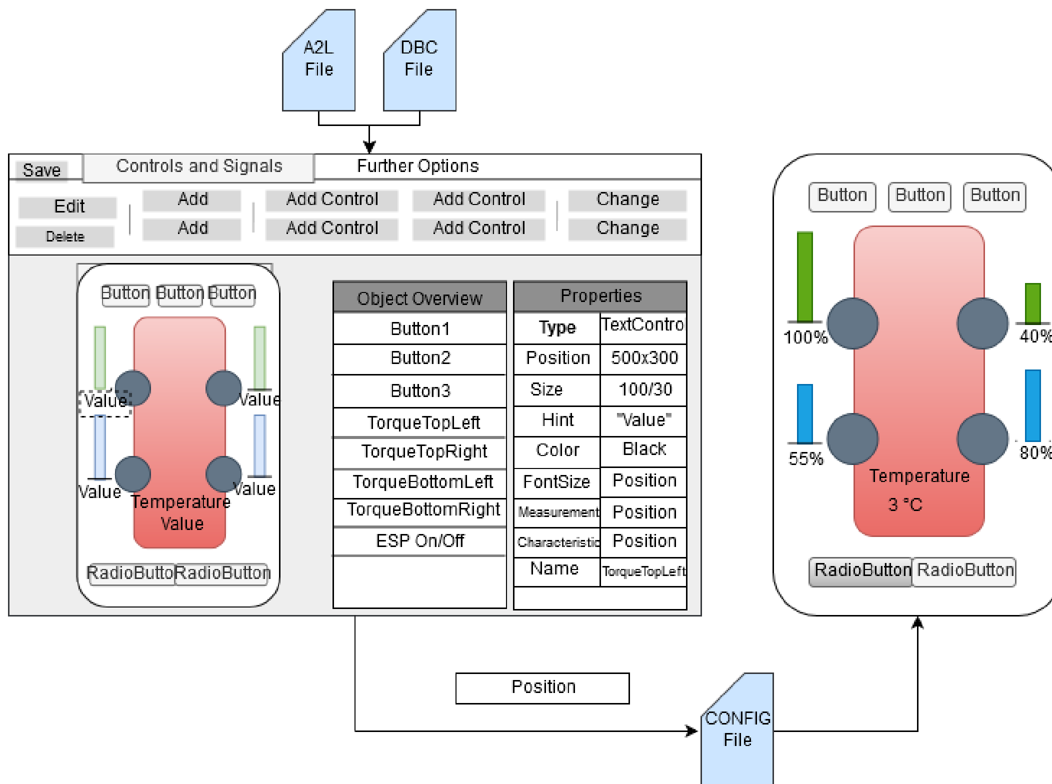

Figure 4.2: The second and final draft of the graphical user interface.

4.2 Implementation

This chapter shows and explains the use of the *AppBuilder* with screenshots, featuring the most important functionality.

In this chapter we provide a walkthrough for the finished *AppBuilder* prototype. It describes and shows in pictures how to use the *AppBuilder* to easily construct an Android application for vehicle data visualization.

4.2.1 Creation of a new project

To create a new project, the button "Neue App erstellen" ("create new App") on the start screen is pressed by the user. A modal window holding the view `CreateNewProject` is opened (see Figure 4.3). The name and screen dimensions of the target device need to be entered by the user. It is possible to change the screen dimensions at a later stage of the project. However, this is not recommended as the layout elements may be displaced. An *A2L file* is needed to be able to assign signals to controls, as all signals are parsed from the provided *A2L file* (see section 3.5.1). It can be added at a later stage of the project though, if the layout design comes first. DBC files are fully optional for Display Apps (see section 3.5.1).

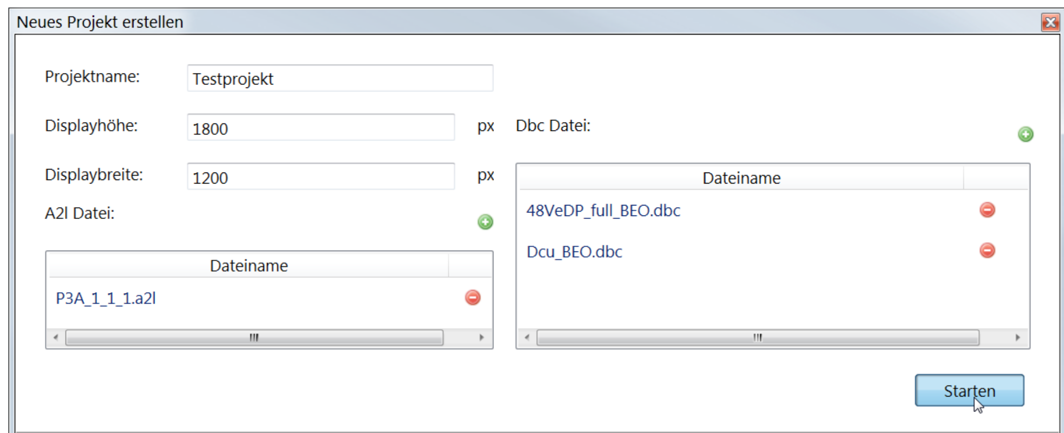

Figure 4.3: `CreateNewProject` View : Creation of a new project.

When name, dimensions and (optionally files) are entered and the button "Starten" ("start") is pressed, a `DisplayAppProject` object is created based on the user input. The `DisplayAppProject` holds all project data.

Signals are parsed from the *A2L file* via regular expressions.

```
private void ParseA2l(string a2lFile, ref Dictionary<String, int> measurements,
                      ref Dictionary<String, int> characteristics)
{
  Regex regex = new Regex(
    @"(begin\s(MEASUREMENT|CHARACTERISTIC)\s)([A-Za-z\d_-]*)(\s"")");
  var matches = regex.Matches(a2lFile);

  //only get the signal (group 1) without the pattern around it
  foreach (Match match in matches)
  {
    GroupCollection groups = match.Groups;
    if (groups[2].Value.Contains("MEASUREMENT"))
    {
      measurements.Add((groups[3].Value), 0);
    }
    else
    {
      characteristics.Add(((groups[3].Value)), 0);
    }
  }
}
```

After parsing, parsed measurements and characteristics are stored as string (key) int (value) dictionaries in the DisplayAppProject class. The string is the signal name, the integer is the number of uses of this signal in the project.

```
public Dictionary<string, int> _measurements;
public Dictionary<string, int> _characteristics;
```

This management of the selected signals makes it possible to add only signals to the exported list which are actually used in the project, thus not overtaxing *GIGABOX Beo*. This is also crucial when the users wants to exchange the *A2L file* at a later stage of the program (see autorefchap:replacea2limplementation).

All data of the application is stored in the *AppData* files. A folder is created for every project. The project folder consists of JSON file which holds all project data and the files which were added to the project (images, *A2L file*, *DBC files*). The JSON file is a serialization of the `DisplayAppProject` object.

4.2.2 Opening a Project

To open an existing project, the button "Bestehende App bearbeiten" ("edit existing project") on the start screen is pressed by the user. The View OpenProject is opened in a modal window where all existing projects are listed by name in a combobox (see Figure 4.4).

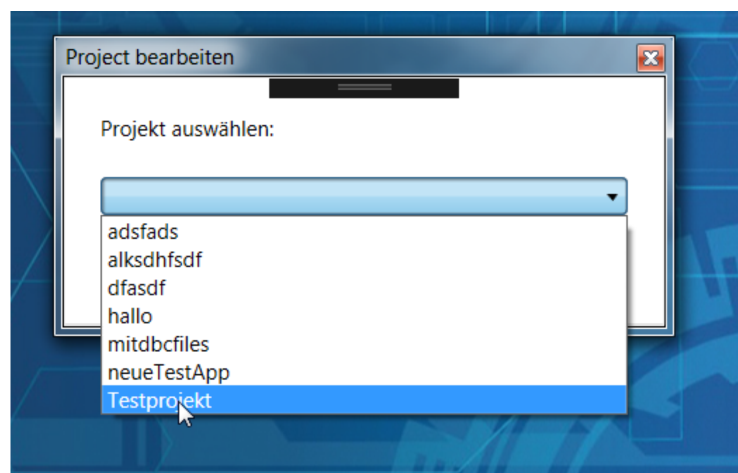

Figure 4.4: OpenProject View : Opening an existing project.

After the desired project is chosen by the user and the button "Öffnen" ("open") is clicked, the JSON file in the AppData folder is deserialized to a DisplayAppProject object. The RegionManager is called to navigate to EditorMain (see subsection 4.2.3) with the DisplayAppProject object as NavigationParameter.

```
var filePath =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData),
           "AppBuilder", projectname, (projectname + ".json"));

DisplayAppProject project =
JsonConvert.DeserializeObject<DisplayAppProject>(File.ReadAllText(filePath));
```

4.2.3 EditorMain

`EditorMain` is the main View of the application (see Figure 4.5). Upon navigating to `EditorMain`, the `EditorMainViewModel` receives and stores all project data from the model `DisplayAppProject`. `EditorMain` is subdivided into four main parts: The **Ribbon** (section 4.2.3) which holds all options. The **Visualization** (section 4.2.3) which shows how the Android application will look on the tablet. The **Item Overview** (section 4.2.3) and the **Property Window** (section 4.2.3). All of these parts do operations on items. Therefore, it is crucial to understand what an item and its structure is. The first paragraphs of this subsection will explain the concept of an item. Subsequently, all four parts of `EditorMain` will be described in detail.

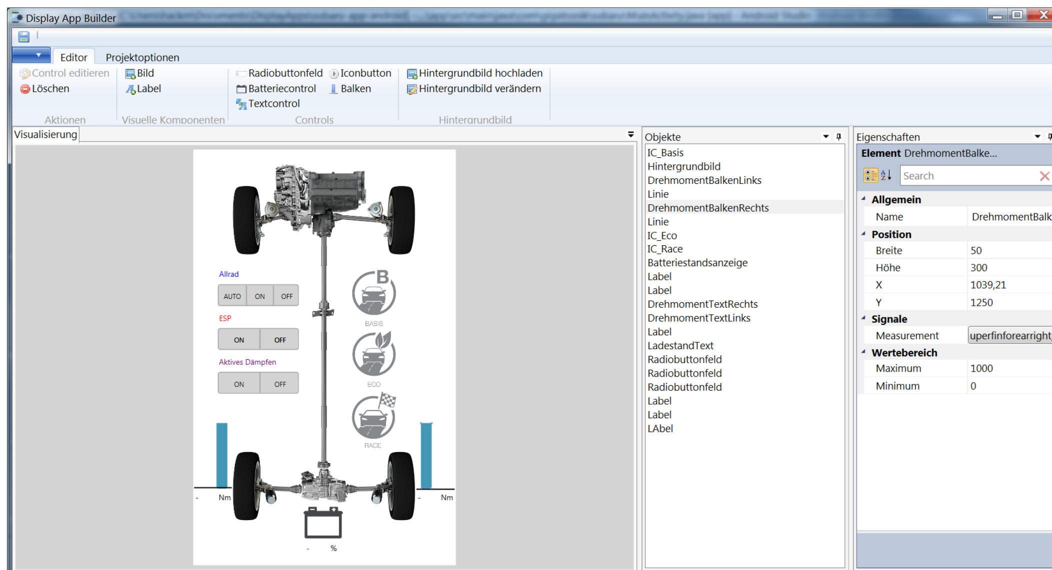

Figure 4.5: `EditorMain` View.

Items

Everything that can be placed on the Visualization is an item, whether it is a background image, a text field or a button. Items can be created

and interacted with by users in order to be able to build their own intelligent *DisplayApp*. An item can either be a simple visual item without an attached signal (for example an image or a label) or a control which displays, processes and/or manipulates vehicle data.

Figure 4.6 shows a class diagram of the item hierarchy. Blue classes are ViewModel classes, green classes are Model classes. In the `DisplayAppProject` class (Model), the project's items are stored in a list of `Item` objects. The `EditorMainViewModel` stores the project's items in a ObservableCollection property (see) of `ItemViewModel` objects. The currently selected item is stored as a `ItemViewModel` property.

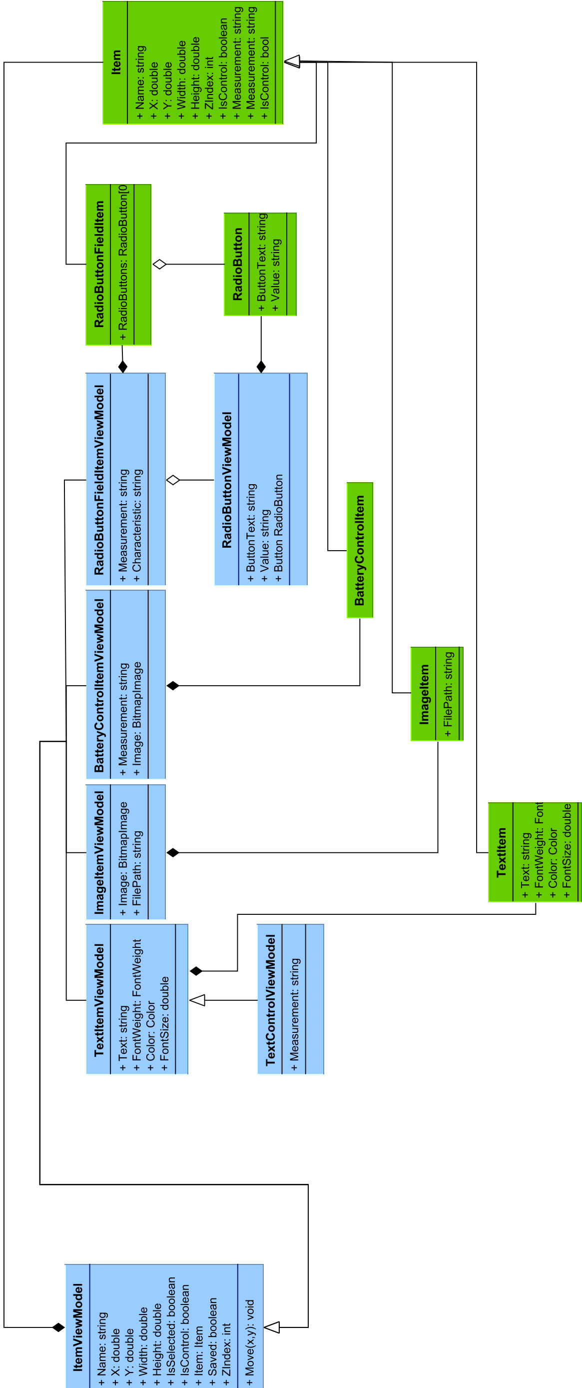

Figure 4.6: Item class diagram.

Ribbon

The **Ribbon** is the panel at the top of `EditorMain` view (see Figure 4.7). It is realized with the *WPF* Ribbon control. The ribbon tab "Editor" on the ribbon contains all buttons that enable the user to **add**, **edit**, and **remove** items. All buttons on the Ribbon are bound to `DelegateCommand` objects in the `EditorMainViewModel` by Data Binding (see subsection 3.2.3).

The ribbon group "Aktionen" (actions) contains buttons for editing and removal of the selected element/control. The ribbon group "Visuelle Elemente" (visual items) contains buttons for creating items that cannot be linked to signals. Available visual items are images and labels which can be used as descriptive elements (titles etc.) or for stylistic purposes. The ribbon group Controls contains buttons for creating visual elements which can be linked to signals. Depending on the control, they may feature complex configuration options (see subsection 4.2.6). Controls can be added in one click and configured afterwards. An optional background image (button "Hintergrundbild bearbeiten") can be set. It has the lowest z-index, i.e. it is shown behind all other visual elements. If the background picture is already set, the user can choose a new image which replaces the old one.

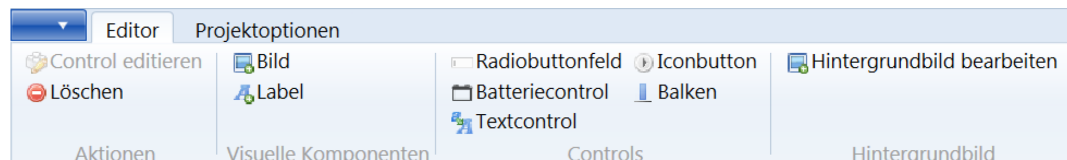

Figure 4.7: Ribbon.

Visualization

The **Visualization** displays the user interface of the configured app as it would on the tablet after an export (see Figure 4.8). The user can select an element with one click. It can be dragged and placed within the borders of the tablet dimensions. By double clicking on a control, the specific configuration window depending on the type of control is opened (see section 4.2.3). In the configuration window, a measurement and/or characteristic can be

assigned by the user. For some types of control (Radiobutton field, Icon Button), more complex configurations are possible.

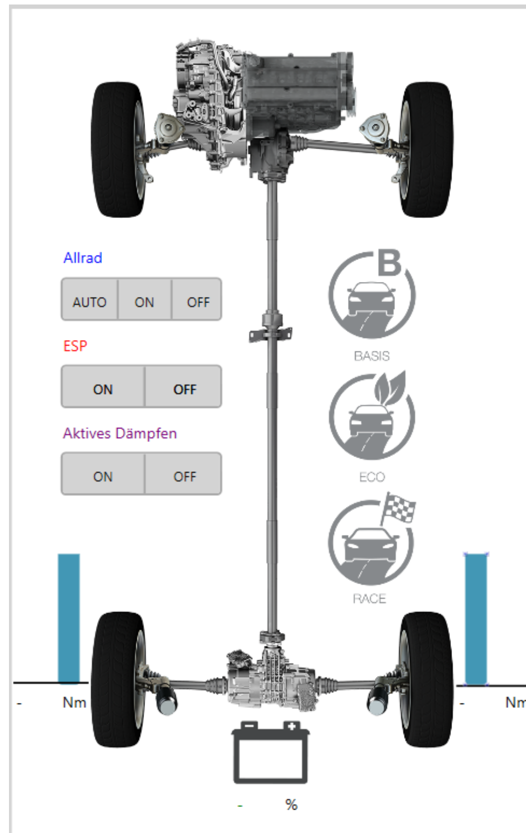

Figure 4.8: Visualization.

The Visualization is realized with the custom *WPF* control `EditorControl` which is a class derived from the canvas control. It converts items to visuals according to their properties and displays them on the canvas. As described above, the user can interact with the items by selecting, resizing, dragging and double clicking them. The next paragraphs are an exemplary description of the implementation of the `EditorControl`. It is based on the creation of and interaction with a Radiobutton field.

The `EditorControl` has a dependency property for a two-way data binding connection to the `ObservableCollection<Itemviewmodel>` property of the

project's items in the `EditorMainViewModel`. It also has a dependency property for two way data binding connection to the selected `ItemViewModel` property in the `EditorMainViewModel`. Every change the ViewModel or the View makes to an item triggers a function which handles the situation. When an item is modified in the View, the ViewModel is notified to update the modified items.

```
public static readonly DependencyProperty ItemsProperty =
DependencyProperty.Register("Items", typeof(ObservableCollection<ItemViewModel>),
typeof(EditorControl), new PropertyMetadata(OnItemsChangedCallback));

public static readonly DependencyProperty CurrentItemProperty =
DependencyProperty.Register("CurrentItem", typeof(ItemViewModel),
typeof(EditorControl), new PropertyMetadata(OnSelectionChangedCallback));
```

Listing 4.1: `DependencyProperties` with callback functions in `EditorControl`

When the user presses the a button on the ribbon to create a `TextControl` (see section 4.2.3), the function `AddTextControl()` is called by the DelegateCommand to which the button is bound. This function initiates a `TextControlViewModel` object in default size and position, adds it to the `Items` property and sets it as the currently selected object.

```
private void AddTextControl()
{
  var textItem = new TextControlViewModel();
  textItem.X = (_project._width + 10) / 2;
  textItem.Y = (_project._height + 10) / 2;
  textItem.Height = 100;
  textItem.Width = 200;
  textItem.ZIndex = 100;
  textItem.FontSize = 35;
  textItem.Text = "<<Wert>>";
  textItem.IsControl = true;

  Items.Add(textItem);
  CurrentItem = textItem;
  CanSaveItems = true;
}
```

Listing 4.2: `AddTextControl()` method `EditorMainViewModel`

The `EditorControl` is notified when the collection of items changes and calls

an EditorElementFactory function to create and place a FrameworkElement on the canvas. The ItemViewModel object is stored in the "Tag" property of its newly created FrameworkElement (see Figure 4.6). This is important to be able to update the ItemViewModel when the FrameworkElement changes and vice versa.

```
private static FrameworkElement CreateText(TextItemViewModel item)
{
  var text = new TextBlock();
  text.Text = item.Text;
  text.FontSize = item.FontSize;
  text.FontWeight = item.FontWeight;
  text.Foreground = new SolidColorBrush(item.Colour);
  text.Tag = item;
  PlaceElement(text, item);
  return text;
}

private static void PlaceElement(FrameworkElement element, ItemViewModel item)
{
  Canvas.SetZIndex(element, item.ZIndex);
  Canvas.SetLeft(element, item.X);
  Canvas.SetTop(element, item.Y);
  element.Width = item.Width;
  element.Height = item.Height;
}
```

Listing 4.3: Creation and Placement of the Battery FrameworkElement

Text controls and labels are handled by the same creation and update methods. They are indistinguishable for the View as TextControlViewModel is derived from TextItemViewModel and the EditorControl only convers visual aspects of controls. Movement and resizing of items in the Visualization is tracked via mouse events and handled by EditorControl.

Item Overview

The **Item Overview** is a list of all items, displayed by name (if assigned) or type (see Figure 4.9). Items can be single-selected by click. A name can be assigned to the selected element in the property window. This is very helpful in larger projects as the user is able to find elements quickly in the Item Overview.

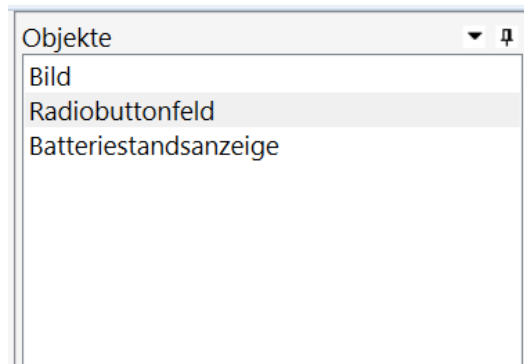

Figure 4.9: Item Overview.

Property Grid

The **Property Grid** is realized with the Xceed PropertyGrid class and gives an overview over the selected item by displaying all public properties of the `ItemViewModel` (see Figure 4.10) (Xceed, 2019). Most properties can be edited directly in the Property Grid by the user.

If the item is a control, i.e. it can have a measurement and/or characteristic, these properties are displayed as buttons with the signal names as button text (see Figure 4.10). If the item is a control and there are no assigned signals yet, the buttons are blank. Clicking on a signal button opens the control configuration window (see subsection 4.2.4).

4.2.4 Controls

There are three ways how the configuration window of a control can be reached:

- Double-clicking on the control in the visualization.
- Selecting the control and clicking the edit control button on the ribbon.
- Selecting the control and clicking on the measurement/characteristic button in the property window.

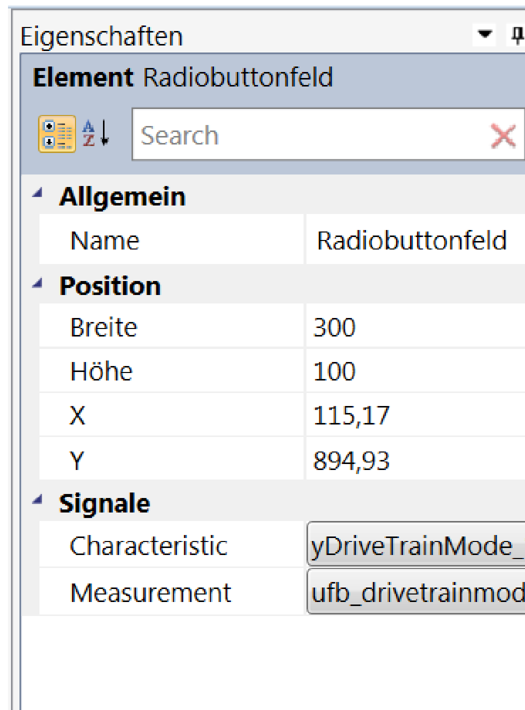

Figure 4.10: Properties.

4.2.5 TextControl, BatteryControl

A measurement can be selected from the measurements in the uploaded A2L file in a searchable combobox (see Figure 4.11). As mentioned in section 4.4.3, measurements are stored as `Dictionary<string, int>`, where the `string` is the measurement and the `int` is the count of controls the measurement is linked to. When the control does not have a previous measurement, the count of this measurement is increased by one in the dictionary. When the control does have a previous measurement that is changed by the new selection, the old measurement's count is additionally decreased by one.

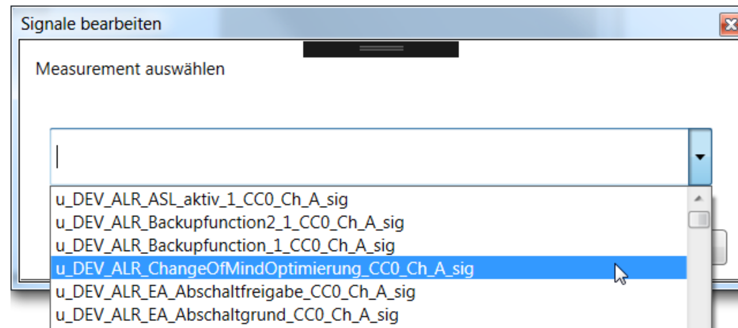

Figure 4.11: Signal Selection.

4.2.6 Radiobutton Field

Signals for the selected radiobutton field can be chosen in the comboboxes at the top of the configuration dialog (see Figure 4.12). A Radiobutton field can be linked to a characteristic with an optional measurement. It is typically linked to both if the button requires feedback from the *GIGABOX Beo* to be sure that the command was effective.

In the lower area of the window, buttons can be added to the radiobutton field by clicking on the green plus icon. Every button can be configured with a button text and the value which is sent at the button click. An example from the field that is created in Figure 4.12: When the user of the built app presses the button with the text ON, the value 1 will be sent to the chosen signal (characteristic).

The user can remove a button from the radiobutton field by pressing the red minus icon next to the relevant button.

4.2.7 Icon Button

Signals for the selected icon button can be chosen in the comboboxes at the top of the configuration dialog (see Figure 4.13). An icon button can be linked to a characteristic with an optional measurement. The value which is sent at a button click is entered in the field S̈ignalbelegung.̈ The checkbox

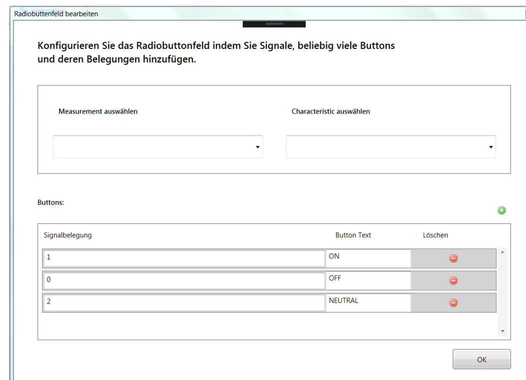

Figure 4.12: Configuration of a Radiobutton Field.

"Auf Feedback warten" specifies if the button should require feedback from the *GIGABOX Beo* to be sure that the command was effective.

In the lower area of the window, three different version of the icon image can be uploaded. The disabled icon is shown when there is no connection to *GIGABOX Beo*. The normal icon is shown when the application is connected but the icon was not pressed (the signal's current value does not equal the specified value of the icon button). The active icon is shown when the icon button was pressed effectively.

4.2.8 Progress Bar

The minimum and maximum of a selected progress bar are set via property grid, 0 being default minimum and 1000 the default maximum (see Figure 4.14). A measurement can be selected in a searchable combobox as already shown in the paragraph about text controls and battery controls. The visualization always shows the progress bar at its maximum value (see Figure 4.8).

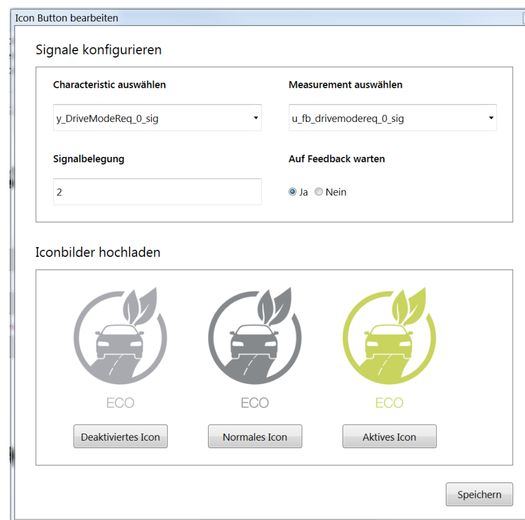

Figure 4.13: Configuration of an Icon Button.

4.2.9 Adding and Replacing DBC and A2L Files

A2L and DBC files can be replaces at all stages of the project. The option for this is located at the file („Dateien“) section on the ribbon (see Figure 4.15).

Usually, *A2L files* are exchanged in the case that the collection of available signals is expanded, adding new signals to the existing ones. Nevertheless, it is important to check if any signals which are linked to controls are missing in the new *A2L file* for the sake of error prevention. To achieve this, the newly uploaded A2L file is parsed and the resulting signals are compared to the existing ones.

If there are missing signals which are used in controls, a warning dialog lists all affected signals and asks the user if they still want to exchange the *A2L file*. The user can subsequently choose to abort the replacement. If they confirm, the signals are removed from the affected controls.

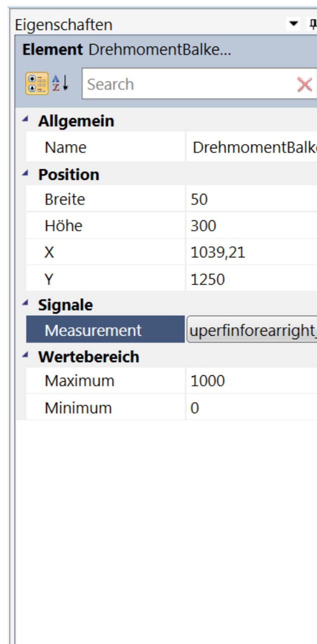

Figure 4.14: Configuration of a Progress Bar in the Property Grid.

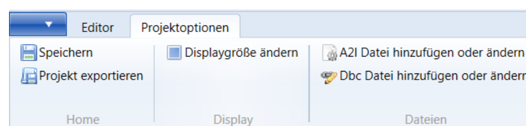

Figure 4.15: Project Options.

4.2.10 Export of a Project

The configured project can be exported by clicking the export project („Projekt exportieren“) button (see Figure 4.16). This option is available in the quick option dropdown or in on the ribbon, in the tab project options (“Projektoptionen).

All project data is saved to an object of a class containing project data tailored for the export. This object is serialized to a JSON file. In contrast to the class `DisplayAppProject`, the class `DisplayAppProjectAndroidExport` contains only data which is needed for constructing the Android application.

Signals are stored in lists. All signals which are not used in the project are removed.

All used images and the JSON file containing project data are saved in an export folder in the *AppData*. The export folder is compressed in a .zip archive.

To start the built app, the user has to copy the exported archive to the internal storage of the tablet of smartphone and open the *AppBuilder* Android application.

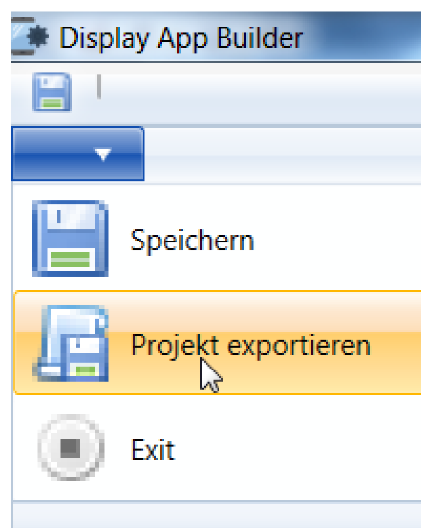

Figure 4.16: Export of a Project.

4.3 Testing

The *AppBuilder* is developed with the test-driven development method (see section 3.1). MSTest, the standard unit testing framework for Visual Studio was used for testing.

Besides writing tests, measuring code coverage is an important part of TDD. Although the used development environment, Visual Studio Professional,

has built in support for executing unit-tests it lacks a framework for generating code-coverage reports. Therefore, an external library was used to monitor code coverage.

OpenCover is an open-source code coverage tool specifically developed for *.NET*. It has a command-line only user interface and produces an XML output file. *OpenCover* provides measurement of the metrics statement coverage, method coverage and branch coverage (OpenCover, 2019). These features make it a great tool to calculate code coverage of the *AppBuilder* code base, although the lack of a GUI and a visual report is inconvenient.

Therefore, the extension was *AxoCover* was chosen for generating and displaying code coverage reports. *AxoCover* is an open source extension to Visual Studio which integrates *OpenCover* and uses its calculated metrics to generate reports amongst other features (AxoCover, 2019). In contrast to OpenCover, it has a GUI which is integrated into Visual Studio (see Figure 4.18). The "report" tab in the AxoCover window shows all relevant information about code coverage of the solution, broken down into projects, namespaces, files and methods. AxoCover also highlights uncovered lines directly in the source code with red bars. For example, most of the method shown in Figure 4.17 is covered by tests, as shown by the green bars next to the line numbers. The case that the background image is deleted is not covered though, as shown by the red bars.

```
544    private void OnDelete()
545    {
546        if(CurrentItem != null)
547        {
548
549            Items.Remove(CurrentItem);
550            CurrentItem.State = ModelState.Deleted;
551            if(CurrentItem == BackgroundImage)
552            {
553                BackgroundImage = null;
554            }
555            CurrentItem = null;
556            CanSaveItems = true;
557        }
558    }
```

Figure 4.17: Identifying uncovered code parts with AcoCover.

The open source library Moq was used to create mocking objects in unit tests. Moq provides an easy way to mock classes and interfaces using the *.NET* LINQ syntax and lambda expressions (Moq, 2019). Mocking objects are necessary in any cases in the process of writing unit tests. The following paragraphs give some mocking examples from testing the *AppBuilder* with code snippets. As apparent in the code snippet below, services are added via dependency injection in the constructor of classes.

```
public EditorMainViewModel(IEventAggregator ea, IRegionManager regionManager,
IMessageBoxWrapper messageBox, IOpenFileDialogWrapper openFileDialog,
IFolderBrowseDialogWrapper folderBrowseDialog)
```

Listing 4.4: Constructor of the class `EditorMainViewModel`. Many services are provided per dependency injection.

To create objects of these classes for testing purposes, their dependencies are mocked. An example is provided in the snippet below where a mocking object of the *Prism* RegionManager is created.

```
var mockRegionManager = new Mock<IRegionManager>();
mockRegionManager.Setup(rm => rm.RequestNavigate(It.IsAny<string>(),
It.IsAny<string>(), It.IsAny<NavigationParameters>())).Callback(delegate { });
var regionManager = mockRegionManager.Object;
```

Listing 4.5: Creation of a mocking object of the *Prism* RegionManager

Mocking is also necessary for all kinds of tests which involve file or folder selection by the user. To enable testability, C# classes for choosing files and folders were wrapped in interfaces. Interfaces are easy to mock, methods can be overwritten by Moq to return desired values or do something different altogether. In the code snippet below, the `OpenFileDialogWrapper` is mocked to return a chosen file path to a testfile.

```
var mockOpenFileDialogWrapper = new Mock<IOpenFileDialogWrapper>();
mockOpenFileDialogWrapper.Setup(o => o.ShowDialog()).Returns(true);
mockOpenFileDialogWrapper.Setup(o => o.FileName).Returns(testfilepath);
var openFileDialogWrapper = mockOpenFileDialogWrapper.Object;
```

Listing 4.6: Creation of a mocking object of the interface `OpenFileDialogWrapper`

By passing this mocking object to a class like in the code snippet above, it is possible to test methods using input/output operations on real files.

Another case where mocking objects are useful is to check if a certain task without a return value are run. For example, the `Action` `FinishInteraction` closes a *Prism* modal dialog. To check if the dialog was closed by running `FinishInteraction`, `FinishInteraction` is overwritten with a delegate that changes the value of a test variable to true (see code snippet below).

```
var finishInteractionCalled = false;
vm.FinishInteraction = delegate { finishInteractionCalled = true; };
Assert.IsTrue(finishInteractionCalled);
```

Listing 4.7: Checking if `FinishInteraction` was run.

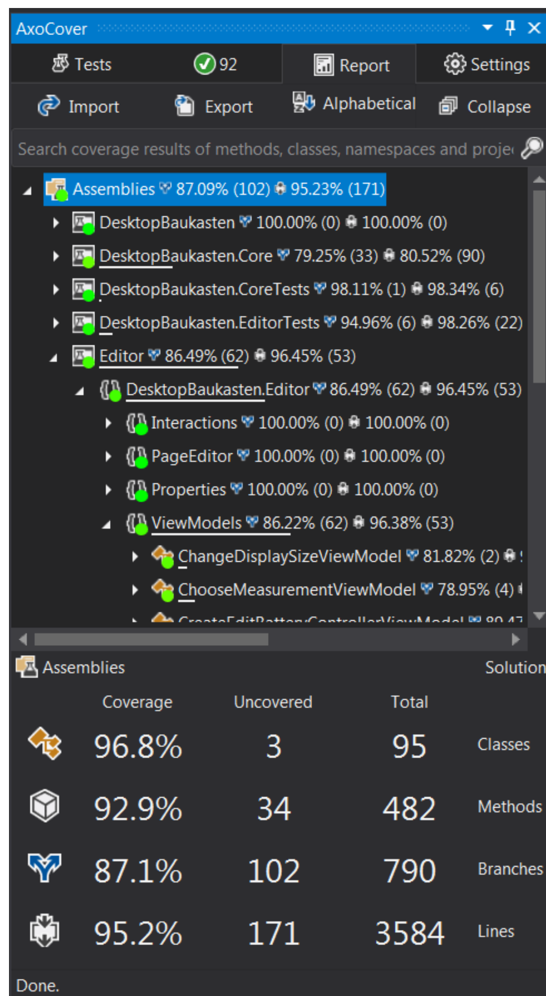

Figure 4.18: Overview over code coverage with the tool AxoCover.

4.4 Usability Inspection

Usability in software is a measure which describes how pleasant to use a user interface is and whether it provides what the user needs. Jakob Nielsen defines usability by the qualities learnability, efficiency, memorability, errors and satisfaction (Nielsen, 2012).

Usability Inspection is a technique to assess a user interface of a software system to find usability problems. Most effectively, usability tests are conducted empirically (with real representative users), although formal and automated approaches exist (Nielsen, 1994).

4.4.1 Setup

The domain and the potential users of the *AppBuilder* are very specific since knowledge about vehicle busses and used file formats is required to use the program. Thus, a usability test was conducted with three potential users who know the domain.

All three test users are familiar with the CAN protocols and Display Apps. They had never seen the application before the usability test. All test users were instructed to carry out exactly the same use cases and to speak their thoughts while performing them. They were especially told to mention all problems as well as positive aspects of using the program while experiencing them. All tests were filmed.

The following use cases were carried out in the course of the test in this order:

1. Create a new project.
2. Enter all required data into the form and upload a prepared A2L file.
3. Upload a background image.
4. Resize the background image to fill out the tablet.
5. Create a radiobutton field. Assign any signal, create an ON and an OFF button. Assign the value “1” to the ON button and the value “0” to the OFF button.
6. Position the radiobutton on the centre (vertical) left (horizontal).

7. Create a label with the text "Motor" and position it above the radiobutton field.
8. Save the project.
9. Upload a small image from the test files. Position it above the radiobutton field and the label.
10. Create a battery control and position it on the above the center of the screen. Assign any signal to it.
11. Create a text control and position it below the battery control. Assign the same measurement as to the battery control.
12. Close the program (a confirmation dialog is opened, asking if the user wants to close the application without saving). Cancel.
13. Save the program.
14. Close the program.
15. Reopen the program.
16. Choose the previously created project and open it.
17. Replace the A2L file.
18. Export the project.
19. Find the zip archive.
20. Copy the archive on the smartphone.
21. Open the *AppBuilder* application.
22. Open the created app.

4.4.2 Test Result

The results of the usability test were analysed, put in writing and structured. This subsection presents the positive aspects and problems which were found by test users.

Positive Aspects

The most mentioned positive aspect was the Windows style ribbon user interface which was described as very intuitive and easy to use by 100% of users. The second most mentioned positive (66.7%) was that the program was perceived as effective and efficient to use, making it easy to create a pleasant interface quickly. Other mentioned positive aspects were error

prevention (warnings when closing the application without saving first, warnings when switching to a new A2L file that lacks used signals) and the easy process to exporting a project.

Problems

All problems were structured and put into a table. Table 4.1 gives information about the use case where the problem was mentioned, a short problem description, the percentage of users who mentioned the problem, the severity and if the problem was fixed. Fixes which are described in the section subsection 4.4.3 are linked in the last column. There were many minor problems which were fixed quickly, like a missing program icon in a dialogue. Therefore, not all fixes are relevant enough to be documented. Occasionally, one refactoring subsection covers multiple problem fixes which belong to the same area of the program.

Use Case	Problem	% of users	Severity	Fixed?
Create a new project	Starting screen is too busy. Cannot spot buttons at first glance.	33.3	Medium	yes 4.4.3
Create a new project	Project creation dialogue is badly aligned and looks unpleasant.	33.3	Low	yes 4.4.3
Upload of A2L and DBC files.	Names of uploaded files are not visible for users. It is very confusing not to be able to see what was uploaded.	66.7	High	yes 4.4.3
Upload of A2L and DBC files.	Multiselection should be possible for DBC files for faster selection.	66.7	Medium	yes 4.4.3
Upload of A2L and DBC files.	Dialogue "A2L file was added" after file selection is unnecessary and annoying.	66.7	Medium	yes 4.4.3
Open and existing project.	Two screens (one for "open project", one for choosing) are too many. One screen with selection and "open project" button would be better.	66.7	Medium	yes 4.4.3

Table 4.1: Problems discovered in the course of the usability test.

Use Case	Problem	% of users	Severity	yes
Upload of a new A2L file (overwriting the old one).	Warning when overwriting the old A2L file is badly designed and annoying.	33.3	Medium	yes 4.4.3
Upload of a new A2L file which misses already used signals.	Warning when overwriting A2L file is unstructured, hardly legible and overflows screen.	100	High	yes 4.4.3
Adding a control.	Controls which are often used should be placed on the left of the ribbon.	33.3	Medium	yes 4.4.3
Configuration of radiobutton field.	Controls cannot be configured by double click. Difficult to find control configuration.	66.7	High	yes
Configuration of radiobutton field.	Arrangement and layout are confusing and look very unpleasant.	66.7	High	yes 4.4.3
Configuration of radiobutton field.	Large "Add button" button looks bad and is at a bad position. "plus" icon next to the list would be better.	66.7	Medium	yes 4.4.3
Configuration of radiobutton field.	Information text can be easily overlooked.	33.3	Medium	yes 4.4.3

Table 4.2: Problems discovered in the course of the usability test.

Use Case	Problem	% of users	Severity	yes
Configuration of radiobutton field.	Visual preview of the radiobutton field missing. The user doesn't know which button is which by looking at the visualization.	66.7	Medium	yes 4.4.3
Assign signal to control.	Signals cannot be changed in the property window. Difficult to find Signal Selection	100	High	yes 4.4.3
Assign signal to control.	Configuration window is arranged badly, information text is too long.	66.7	Medium	yes 4.4.3
Assign signal to control.	No text search for signals. It can take a long time to find a signal.	66.7	High	yes 4.4.3
Configuration of text control.	Text is not editable in property grid. Annoying to do several clicks to change the text.	66.7	High	yes 4.4.3
General	Application icon is missing on the top left of the program.	33.3	Low	yes
Saving the project.	The popup dialogue "changes were saved" after saving is annoying.	33.3	Low	yes 4.4.3

Table 4.3: Problems discovered in the course of the usability test.

Use Case	Problem	% of users	Severity	yes
Export of the project.	Changes shouldn't be saved automatically when exporting the project. The program should ask first.	33.3	Medium	yes
Property Grid.	When an item can have no signals or only a measurement, no empty "Measurement" and "Characteristic" fields should be displayed in the Property Window. this is confusing!	100	Medium	yes 4.4.3
Closing the program.	"Save before exiting?" dialogue is not centred	33.3	Low	yes

Table 4.4: Problems discovered in the course of the usability test.

4.4.3 Refactoring

The results of the test uncovered serious shortcomings of the user interface regarding usability. Extensive changes in the UI were planned and implemented in order to tackle those shortcomings. This section presents the results of these changes.

Starting Page

The main problem regarding the starting page was the overwhelming background image. Two of the three test subjects found the starting page confusing because the buttons hardly stood out against the background (see Figure 4.19).

Figure 4.19: Old starting page.

As Figure 4.20 shows, the background picture was exchanged to a more minimalistic, modern design to improve usability. The contrast of the buttons against the background was increased and they were placed more

prominently. Additionally, a title containing the name of the program was added.

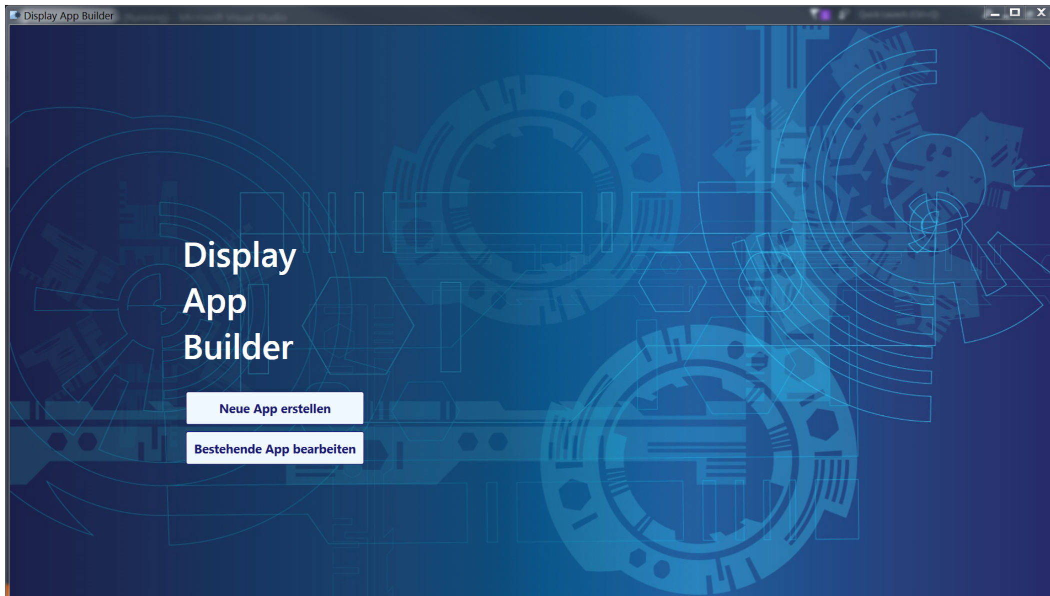

Figure 4.20: New starting page.

Project Creation

The first problem regarding project creation was the alignment and the large, unpleasant-looking buttons (see Figure 4.21). To fix this problem, the GUI was restructured with new alignment and small "+" icon buttons for adding files (see Figure 4.22).

The rest of the problems concerned file upload. In the old version, the user could not see which files they had already uploaded. After every upload, there was a popup dialogue which confirmed that the file was uploaded. Also, DBC files had to be uploaded one by one. In the new starting screen, the names of already uploaded files are displayed in a list and can be deleted by clicking the "-" icon button". The dialogue was eliminated. Multiselect is enabled for the upload of DBC files.

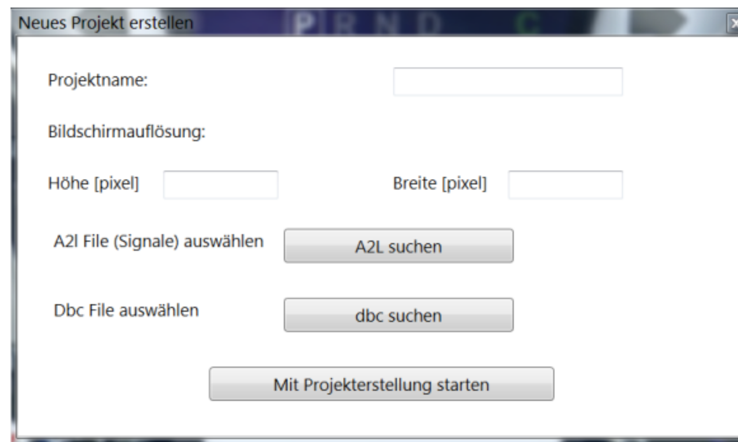

Figure 4.21: Old project creation interface.

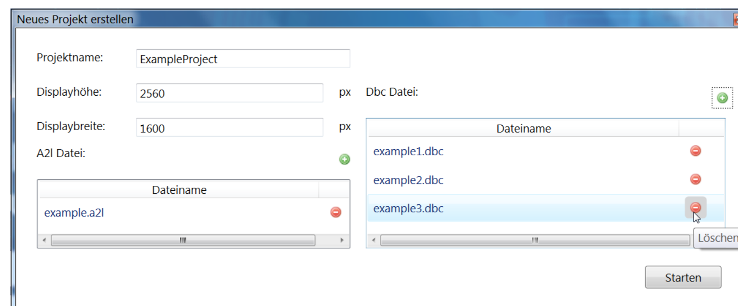

Figure 4.22: New project creation interface.

Opening an Existing Project

The process of opening a project was criticised in the usability test. In the old version, there were two screens for opening a project. When the "Edit Project" button was clicked by the user, a modal dialogue with a "Choose Project" button was opened. When the "Choose Project" was clicked by the user, a modal dialogue with a combobox was opened (see Figure 4.23). This combobox was bound to a list of projects by data binding. When a project was selected by the user, the application navigated to the `EditorMain` View without any button click.

Figure 4.23

The new process of opening a project is more straightforward. When the "Edit Project" button is clicked by the user, a modal dialog with a combobox (bound to a list of existing projects), and an "Open" button is shown (see Figure 4.24). After choosing a project and clicking the button, the application navigates to the `EditorMain` View.

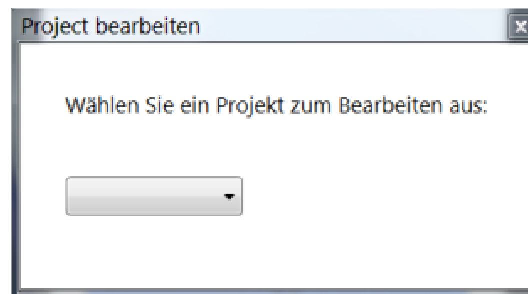

Figure 4.23: Old interface to open a project.

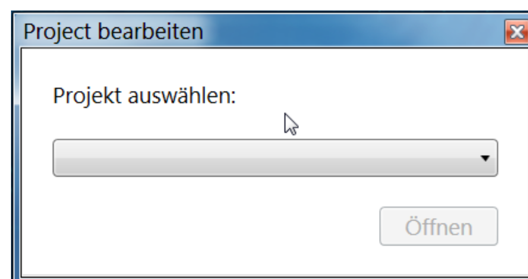

Figure 4.24: New interface to open a project.

Placement of Buttons on the Ribbon

The old ribbon was criticised for its button placement. The option to upload or exchange a background image was far on the left, even though it is usually not frequently needed (seeFigure 4.25). The new ribbon layout places buttons which the user will frequently need on the left (see Figure 4.7).

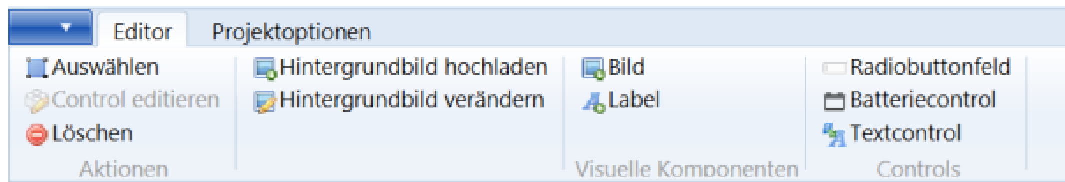

Figure 4.25: Old placement of buttons on the ribbon.

Property Grid

The first criticism was that signals, unlike other properties, could not be edited in the property grid. The reason for this problem is that signals have to be chosen from a list of measurements and characteristics respectively, unlike text and name properties which can be set to arbitrary strings. Given the dense layout of the property grid and the high number of signals, comboboxes are not feasible for signal selection in the property grid. Instead, this problem was solved via signal selection buttons (see Figure 4.10). The signal properties are displayed as buttons with the signal names as button labels If there are no assigned signals yet, the buttons are blank. Pressing one of these buttons opens the control configuration dialog.

The second criticism regarding the property grid was that purely visual items like labels and images had empty signal property fields although they cannot be linked to signals. Also, properties that can only be linked to measurements (not to characteristics) had an empty characteristic property field (see Figure 4.27). The reason for this problem was the class hierarchy of items. Figure 4.26 shows this with the example of labels and text controls. As one can see on the left, the class `ItemViewModel` had properties for signals. Specific items were directly derived from `ItemViewModel`. Labels and text controls were objects of the same class with the value of the IsControl property as their only distinction. Therefore, labels had signal properties although they could not be assigned, battery controls had a ”Characteristic” property which could not be assigned. The property grid displayed these empty, not assignable properties.

The new class hierarchy is on the right side of Figure 4.26. `ItemViewModel` does not have properties for signals. Labels are objects of the class `TextItemViewModel` which does not have properties for signals either. Text controls are objects of

the class `TextControlViewModel` which has a "Measurement" property and no "Characteristics" property. Thus, the property grid only displays fields for signals that items can actually be linked to.

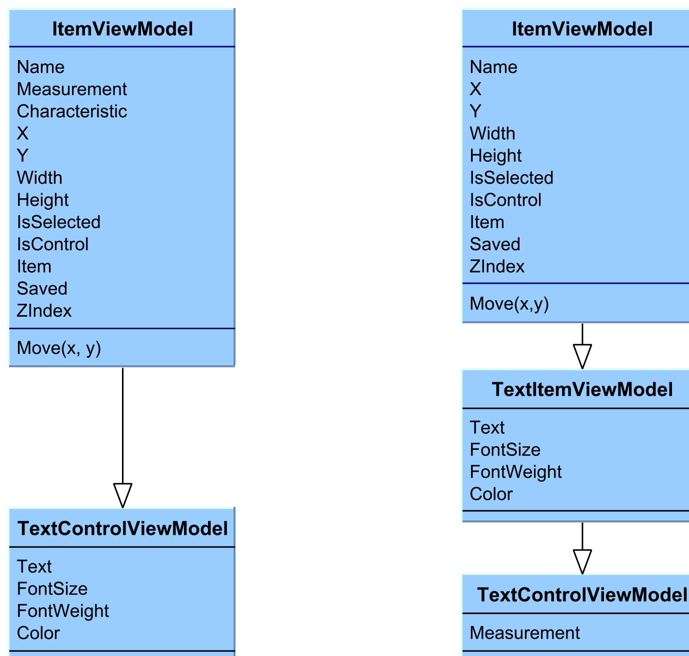

Figure 4.26: Refactoring of item class hierarchy, example based on label and text control.

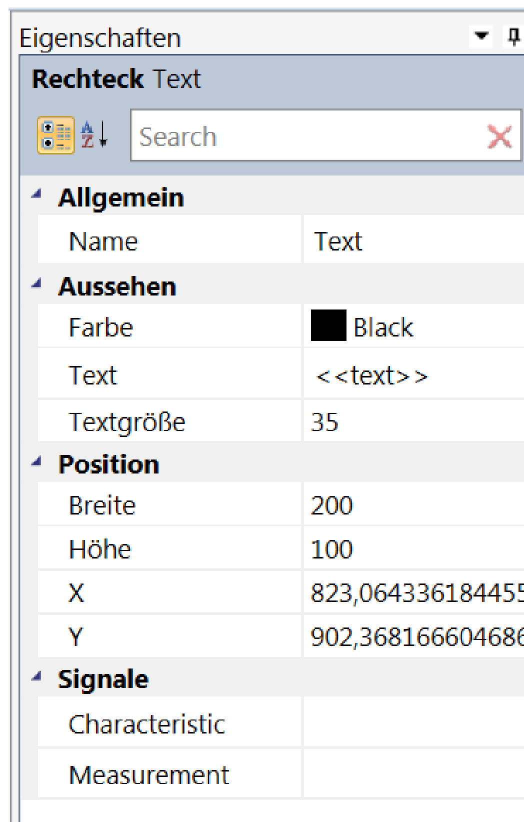

Figure 4.27: Old Property Grid. Although `BatteryControl` cannot be linked to a characteristic, an empty field is displayed. Also, signals cannot be edited from the Property Grid.

Measurement Assignment

The first discovered usability problem regarding the measurement assignment interface was the structure of the modal window and the overly long description text (see Figure 4.28). The second usability problem was that users had difficulties finding the desired measurement among dozens or hundreds of measurements.

The measurement assignment window was restructured, the explanation text was shortened (see Figure 4.11). Also, a searchable combobox was implemented in order to make signal assignment fast, efficient and pleasant

for the user.

Figure 4.28: Old interface for signal configuration of text controls and battery controls.

Configuration of Radiobutton Fields

Many usability problems regarding the configuration of radiobutton fields were discovered during the usability test. Firstly, the layout was criticised as unpleasant and confusing (see Figure 4.29). Secondly, the explanation text was almost overlooked by users because it was too small and unnoticeable. Thirdly, the "Add" button was criticised for not having an icon and having an unintuitive position. Lastly, the lack of a visual preview for radiobutton fields was perceived as confusing.

The modal window was restructured and realigned with distinct areas for different tasks, a more noticable description text and searchable comboboxes for signal selection (see Figure 4.30). Buttons are added by clicking a green "+" icon buttons right above the icon list. Finally, previews for buttons are rendered as bordered *WPF* grids in the Visualization (see Figure 4.8).

Save

The usability problem in this use case was the save message dialogue, which was perceived as annoying by users (see Figure 4.31). Therefore, the dialog

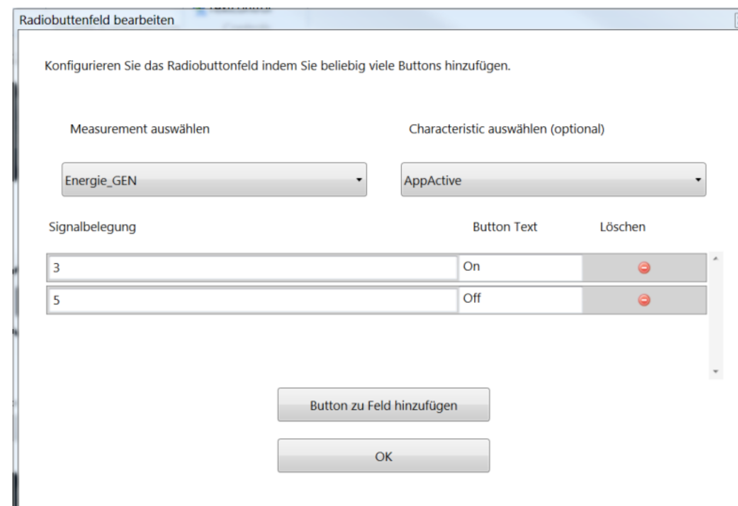

Figure 4.29: Old interface for configuration of radiobutton fields.

was removed. In order for users to have some feedback about the success of their actions, the "IsEnabled" property of the save button was bound to the "CanSave" property of the ViewModel. Thus, the button is greyed out after saving as long as there are no new changes (see Figure 4.32).

A2L File Replacement Warning

The discovered usability problem regarding A2L replacement was the warning message dialogue which was opened when an A2L file was overwritten by a new file which missed signals already linked to controls. Users criticised that the message dialogue was unstructured, hardly legible and overflowed the screen (see Figure 4.33).

Said message dialogue was centred and restructured, displaying missing signals in a list (see Figure 4.34).

Radiobuttenfeld bearbeiten

Konfigurieren Sie das Radiobuttonfeld indem Sie Signale, beliebig viele Buttons und deren Belegungen hinzufügen.

Measurement auswählen

u_fb_enableactivedampingreq_1_sig

Characteristic auswählen

y_EnableActiveDamping_1_sig

Buttons:

Signalbelegung	Button Text	Löschen
1	ein	
0	aus	

OK

Figure 4.30: New interface for configuration of radiobutton fields.

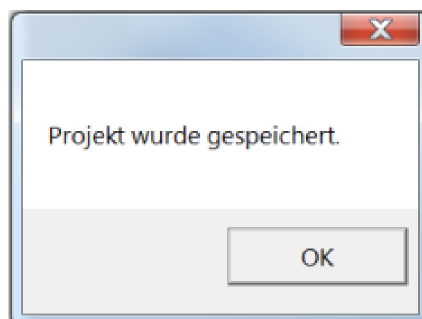

Figure 4.31: Old way to confirm that the project is saved.

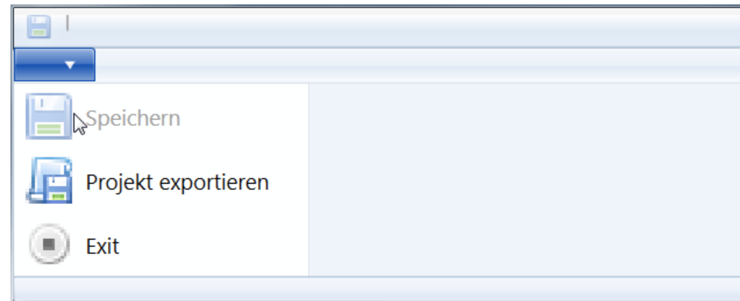

Figure 4.32: New way to confirm that the project is saved.

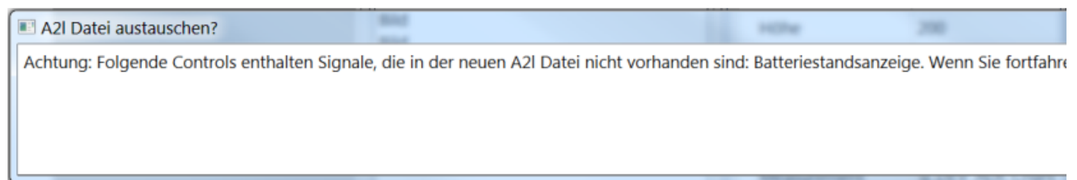

Figure 4.33: Old A2L File Replacement Warning.

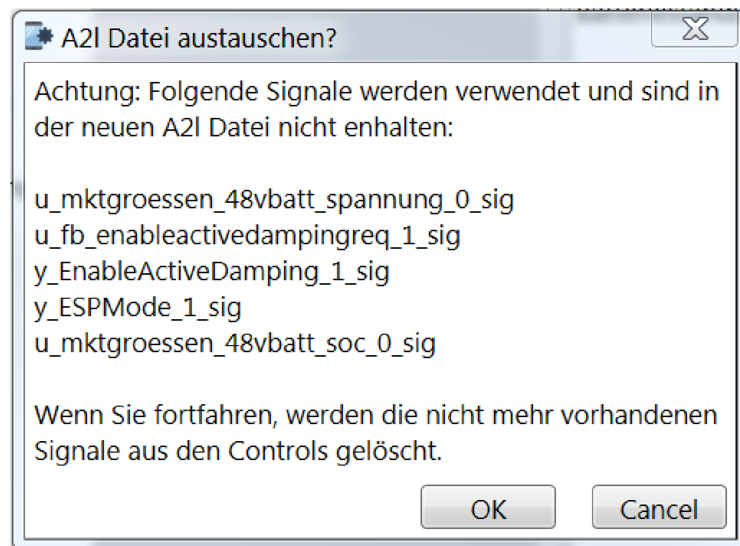

Figure 4.34: New A2L File Replacement Warning.

5 AppLoader (Android Application)

This chapter describes the development of the Android application that builds a user interface based on the configuration which is exported from the *AppBuilder*.

DisplayApps are built for a broad range of different cars. Therefore, the main components were moved to libraries (see Figure 5.1). The AKKA XCP libraries provide all functionalities concerning communication with *GIGABOX Beo*. This includes the establishment of a connection with *GIGABOX Beo*, the decoding of messages and the initialization of data acquisition from certain data locations (see section 3.5.1). The AKKA app utilities library mainly consists of a set of preconfigured classes for UI elements that display and manipulate data on the *GIGABOX Beo*. They only have to be parametrized with the UI element, the relevant signal and other data depending on the element.

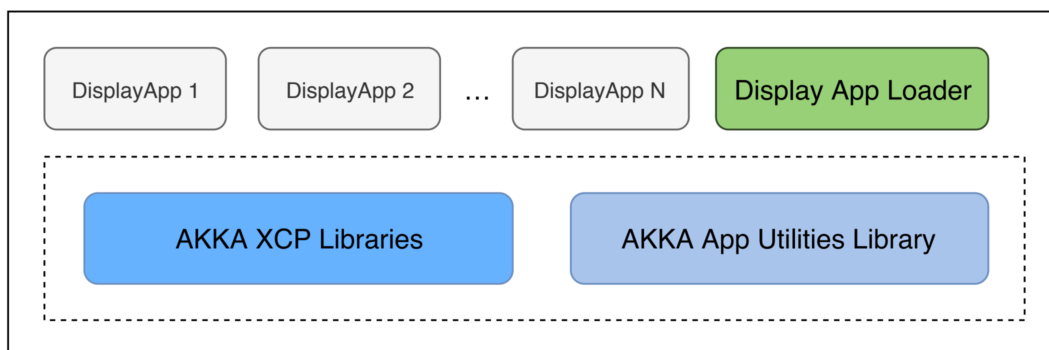

Figure 5.1: First draft of the graphical user interface.

On top of these libraries, every *Display App* was individually programmed as a new Android App. The *AppLoader* uses the same libraries as the individual Display Apps. However, instead of programming a completely new application for every new car, the *AppLoader* builds *Display Apps* based on imported configuration files from the *AppBuilder*.

5.1 Design

5.1.1 Requirements

The following requirements were specified for the *AppLoader* together with AKKA Austria. Each requirement consists of a description and a list of acceptance criteria.

List available projects

When the app is launched, a list of available projects on the internal memory is displayed. The user may select one of these projects to open it.

Acceptance criteria:

- List of project names is displayed in the app.
- The projects are read from a predefined folder on the internal memory.

Open project

The project selected from the previous project list opened and the project file is fully parsed. The Android App creates the currently supported components as described in the project file. After the all components are loaded, the connection the *GIGABOX Beo* (control unit) is established. The addresses of the memory blocks that should be read are collected and sent to the *GIGABOX Beo*.

Acceptance criteria:

- The screen with currently supported controls is displayed correctly.
- The connection to *GIGABOX Beo* is established.
- DAQ is initialised with the correct memory addresses from the project file.

Display image

Every image component of the project configuration is displayed on the screen.

Acceptance criteria:

- The image files specified in the project file are displayed on the screen.
- The image files are displayed with the correct dimensions.

Display text field:

Every text field of the project configuration is displayed on the View. The component is listening for updates from the control unit. Values received from the control unit are displayed in their respective text fields.

Acceptance criteria:

- The text fields specified in the project configuration is displayed on the screen.
- The text fields are displayed with the correct dimensions.
- The text fields display their corresponding values they received from *GIGABOX Beo*
- The changes of these values are reflected in their text fields.

Display button

Every button of the project configuration is displayed on the screen. When the button is pressed, the correct value is sent to the control unit.

Acceptance criteria:

- The buttons specified in the project file are displayed on the screen.
- The buttons are displayed with the correct dimensions on the specified position on the screen.
- The buttons display their corresponding labels (from the project file).
- Pressing the buttons sends the correct value to the control unit.

5.1.2 User Interface

The user interface of the *AppLoader* has two tasks to fullfill: Project selection by the user and displaying the selected project. Therefore the UI is very simple. The selection of a project is achieved by a clickable list of available projects (see Figure 5.2). The UI design was never modified since the first mock-up.

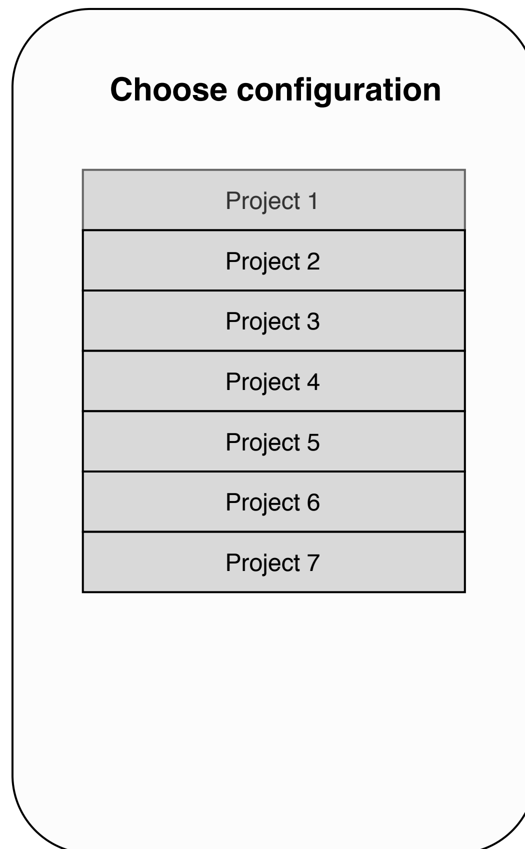

Figure 5.2: The first and final draft of the *AppLoader*'s graphical user interface.

5.2 Implementation

The Android application is implemented with the Model-View-Presenter pattern (see subsection 3.3.2). For every Activity[1] there is a corresponding Presenter object which communicates with the Model and performs input/output operations. The *AppLoader* consists of two Activities and therefore two main Views: MainActivity and CarActivity.

When the application is started, the `MainActivity` object (View) initializes a `MainPresenter` (Presenter) object and attaches itself to it. Read and write

[1] View logic of a single screen (see Android, 2019).

permissions for the external storage are requested by the `MainPresenter` in order to be able to display a list of files and subsequently open and unzip the configuration archive. If read and write permissions are denied, an error message is displayed an the application is terminated. Otherwise, internal storage of the device is scanned for archives files containing configured projects in `MainPresenter`. A clickable list of available projects is created and displayed to the user by `MainActivity` (see Figure 5.3). The archive of the clicked project is extracted and the JSON file is loaded into a local string variable. `CarActivity` is started with the JSON string as parameter.

When the `CarActivity` is started, a loading screen overlay is set to is placed over the rest of the GUI elements for the time of the setup (see Figure 5.4). The JSON string is deserialized into a `DisplayAppProject` data object.

The DBC file and the A2L file are parsed and the application tries to establish a connection with *GIGABOX Beo*. A DAQ measurement is initiated with the list of all relevant measurements from the `DisplayAppProject` as parameter (see section 3.5.1). All of the mentioned functions concerning communication with *GIGABOX Beo* are part of the AKKA XCP library (see chapter 5).

A GUI element is created for every item from the loaded `DisplayAppProject`. The following two functions create labels and text controls. Size, position, text color and other properties of the `TextView` are set to the information in the `TextItems`. If the `TextItem` is a label and therefore does not have a measurement, the method `getVariableDescriptionForItem` returns null. Subsequently, a method in the `CarActivity` is called to display and connect the configured `TextView`.

```
fun addTextViewsToLayout(rl : RelativeLayout){
  for(textItem in displayApp.TextItems){
    var tv = TextView(rl.context)
    tv.layoutParams = RelativeLayout.LayoutParams(textItem.Width.roundToInt(),
                                                  textItem.Height.roundToInt())
    tv.text = textItem.Text
    tv.x = textItem.X
    tv.y = textItem.Y
    tv.setTextColor(Color.parseColor(textItem.Color))
    tv.textSize = textItem.FontSize
    tv.id = generateViewId()

    val varDescr = getVariableDescriptionForItem(textItem.Measurements)
```

```
      uiScope.launch{
        getViewOrThrow().showTextBoxController(rl, tv, varDescr, false)
      }
    }
}
```

Listing 5.1: In `CarPresenter`

In the following code snippet, the configured `TextView` is displayed. For items which are controls (i.e. the `VariableDescription` is not null), a controller is created for the GUI element. The controllers are part of the AKKA app utilities library and provide all functionality for communication GUI element and *GIGABOX Beo* (see chapter 5).

```
override fun showTextBoxController(rl : RelativeLayout, tv: TextView,
            varDescr: VariableDescription?, valFromDbc: Boolean)
            : IXcpUiController?{
  rl.addView(tv)
  if(varDescr != null){
    val controller = TextBoxController(getXCPContext(), varDescr, tv.id, false)
    if (varDescr != null && varDescr.dataType == DataTypeEnum.FLOAT32_IEEE.name){
      controller.setFormat("%.0f")
    }
    Controllers.add(controller)
    return controller
  }
  return null
}
```

Listing 5.2: Configuring a label or text control in the `CarPresenter` object.

When all items have been created, the loading overlay is removed and the configured user interface is shown. Figure 5.5 shows a screenshot of the previously configured application.

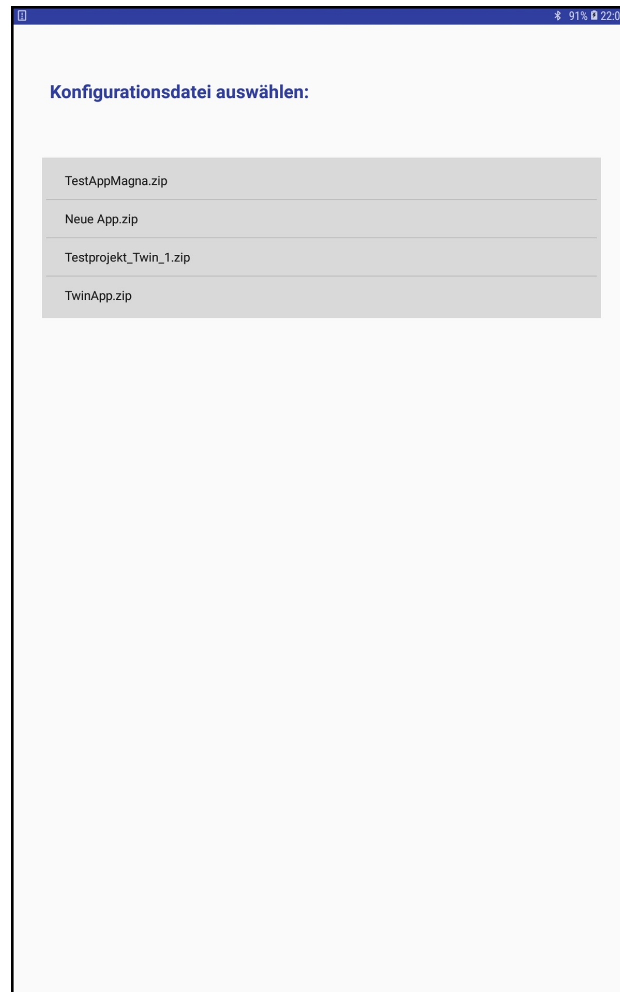

Figure 5.3: Clickable list of available projects.

Figure 5.4: The loading screen is shown while the GUI is configured.

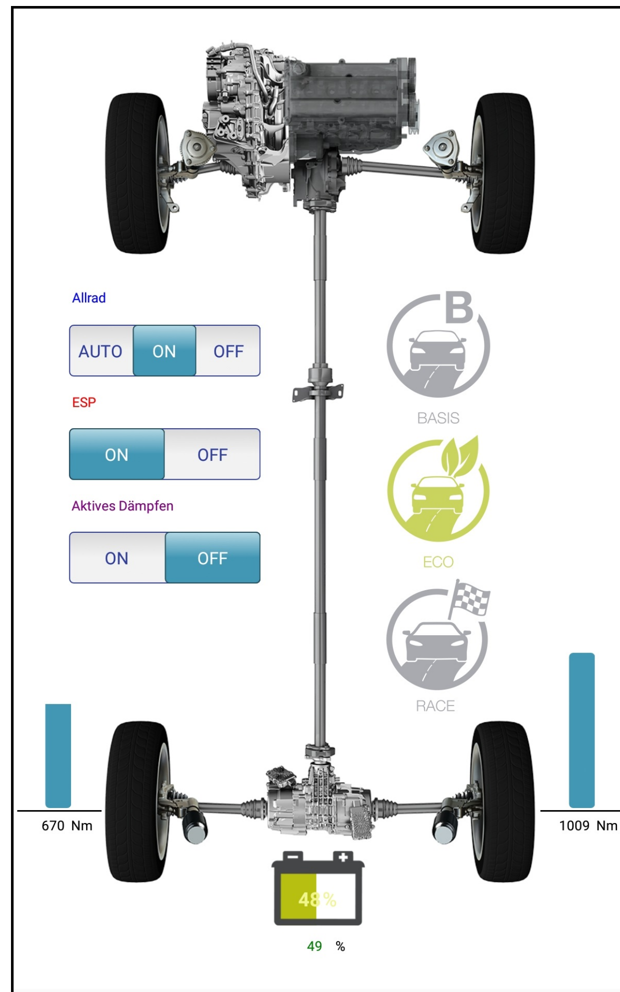

Figure 5.5: The configured application.

6 Summary and Future Work

6.1 Summary

The point of this thesis was the development of a visual coding tool for the creation of *AKKA Display Apps*. Chapter 1 introduced the reader to the background and motivation of the thesis. Chapter 2 examined the state of the art, comparing and analysing two popular visual coding tools. Chapter 3 explained methods and frameworks which were used during development as well as the technological background of vehicle busses and automotive network protocols. Chapter 4 described the design, development and usability inspection of the *AppBuilder* in detail. Chapter 5 examined the design and development process of the *AppLoader*.

The result of this thesis is a modular construction kit which enables users to create intelligent Android applications without writing any code. The constructed Android applications have runtime access to a defined vehicle's CAN bus and allow users to visualize and modify data.

6.2 Future Work

After a company-internal presentation, further development of the *AppBuilder* and *AppLoader* was decided. The set of preconfigured GUI elements will be expanded by several new controls including speedometers and diagrams. The possibility of multi-view applications will be implemented, allowing users to construct separate Views for debugging and showcasing of new features. Also, widgets and a map View will be added to the set of possible interface elements.

By the use of the *AppBuilder*, testing and demonstration of ECUs in cars will be more efficient and flexible. The resulting reduction of delays during the test season saves time and money.

Appendix

Bibliography

AKKADigital (Aug. 2013). *GIGABOX beo*. URL: https://www.akka-digital.com/fileadmin/Download_Products/gigabox_beo/giga_flyer_beo_akka.pdf (visited on 04/17/2019) (cit. on p. 25).

Andreas Patzer, Rainer Zaiser (2016). *XCP – The Standard Protocolfor ECU Development* (cit. on p. 26).

Android (2019). *Activity*. URL: https://developer.android.com/reference/android/app/Activity (visited on 06/27/2019) (cit. on p. 79).

AppyBuilder (2019). URL: appybuilder.com (visited on 05/28/2019) (cit. on pp. 5, 9, 11).

ASAM (Nov. 30, 2017). *ASAM MCD-1 XCP*. URL: https://www.asam.net/standards/detail/mcd-1-xcp/ (visited on 06/24/2019) (cit. on p. 26).

AxoCover (2019). *AxoCover*. URL: https://github.com/axodox/AxoCover (visited on 06/22/2019) (cit. on p. 53).

Beck, Kent (Nov. 8, 2002). "Test-Driven DevelopmentBy Example." In: Addison-Wesley (cit. on pp. 13, 14).

Bosch (Sept. 1991). *CAN Specification*. URL: http://esd.cs.ucr.edu/webres/can20.pdf (visited on 05/08/2019) (cit. on p. 25).

Cervantes, Edgar (Mar. 18, 2016). *Thunkable: coding for the masses and profits for the makers*. URL: https://xceed.com/wp-content/documentation/xceed-toolkit-plus-for-wpf/PropertyGrid%20class.html (visited on 06/13/2019) (cit. on p. 5).

CiA (2019). *History of CAN technology*. URL: https://www.can-cia.org/can-knowledge/can/can-history/ (visited on 05/08/2019) (cit. on p. 25).

Crispin, L. (Nov. 2006). "Driving Software Quality: How Test-Driven Development Impacts Software Quality." In: *IEEE Software* 23.6, pp. 70–71. ISSN: 0740-7459. DOI: 10.1109/MS.2006.157 (cit. on p. 14).

George, Boby and Laurie Williams (2003). "An Initial Investigation of Test Driven Development in Industry." In: *Proceedings of the 2003 ACM Symposium on Applied Computing*. SAC '03. Melbourne, Florida: ACM, pp. 1135–1139. ISBN: 1-58113-624-2. DOI: 10.1145/952532.952753. URL: http://doi.acm.org/10.1145/952532.952753 (cit. on p. 14).

ISO (Dec. 2015). *ISO 11898-1:2015*. URL: https://www.iso.org/standard/63648.html (visited on 05/08/2019) (cit. on p. 25).

Karac, I. and B. Turhan (July 2018). "What Do We (Really) Know about Test-Driven Development?" In: *IEEE Software* 35.4, pp. 81–85. ISSN: 0740-7459 (cit. on p. 15).

Kühnel, Andreas (2013a). "Visual C# 2012." In: Fourth. Rheinwerk Verlag. Chap. Das MVVM-Pattern. ISBN: 0335216846 (cit. on p. 16).

Kühnel, Andreas (2013b). "Visual C# 2012." In: Fourth. Rheinwerk Verlag. Chap. Konzepte von WPF. ISBN: 0335216846 (cit. on p. 20).

Lardinois, Frederic (2019). *Kotlin is now Google's preferred language for Android app development*. URL: https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/ (visited on 06/15/2019) (cit. on p. 23).

Microsoft (2019a). *Code-Behind and XAML in WPF*. URL: https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/code-behind-and-xaml-in-wpf (visited on 06/27/2019) (cit. on p. 16).

Microsoft (2019b). *The MVVM Pattern*. URL: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10) (visited on 05/13/2019) (cit. on pp. 15, 16).

MITAppInventor (2019). *About Us*. URL: https://appinventor.mit.edu/explore/about-us.html (visited on 05/19/2019) (cit. on pp. 7–9).

Moq (2019). *Moq*. URL: https://github.com/moq/moq4 (visited on 06/22/2019) (cit. on p. 54).

Müller, M. et al. (2018-11). "Enabling Teenagers to Create and Share Apps." In: *2018 IEEE Conference on Open Systems (ICOS)*, pp. 25–30. DOI: 10.1109/ICOS.2018.8632815 (cit. on pp. 5, 6).

Nielsen, Jakob (1994). "Usability Inspection Methods." In: *Conference Companion on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: ACM, pp. 413–414. ISBN: 0-89791-651-4. DOI: 10.1145/259963.260531. URL: http://doi.acm.org/10.1145/259963.260531 (cit. on p. 57).

Nielsen, Jakob (Jan. 4, 2012). *Usability 101: Introduction to Usability*. URL: https://www.nngroup.com/articles/usability-101-introduction-to-usability/ (visited on 06/16/2019) (cit. on p. 57).

OpenCover (2019). *OpenCover*. URL: https://github.com/OpenCover/opencover (visited on 06/22/2019) (cit. on p. 53).

Prism (2019a). *Commanding*. URL: https://prismlibrary.github.io/docs/commanding.html (visited on 06/15/2019) (cit. on p. 21).

Prism (2019b). *Using the ViewModelLocator*. URL: https://prismlibrary.github.io/docs/viewmodel-locator.html (visited on 06/15/2019) (cit. on p. 20).

Syromiatnikov, A. and D. Weyns (Apr. 2014). "A Journey through the Land of Model-View-Design Patterns." In: *2014 IEEE/IFIP Conference on Software Architecture*, pp. 21–30. DOI: 10.1109/WICSA.2014.13 (cit. on p. 17).

Vector (Jan. 2007). *DBC File Format Documentation*. URL: http://read.pudn.com/downloads766/ebook/3041455/DBC_File_Format_Documentation.pdf (visited on 06/24/2019) (cit. on p. 26).

Vector (2019). *Beschreibung der Kommunikationsnetze*. URL: https://www.vector.com/de/de/produkte/anwendungsgebiete/steuergeraete-kalibrierung/datenbeschreibung/#c25783 (visited on 06/24/2019) (cit. on p. 25).

Wolber, David, Harold Abelson, and Mark Friedman (Jan. 2015). "Democratizing Computing with App Inventor." In: *GetMobile: Mobile Comp. and Comm.* 18.4, pp. 53–58. ISSN: 2375-0529. DOI: 10.1145/2721914.2721935. URL: http://doi.acm.org/10.1145/2721914.2721935 (cit. on p. 7).

Xceed (2019). *PropertyGrid class*. URL: https://www.androidauthority.com/thunkable-coding-678467/ (visited on 05/28/2019) (cit. on p. 46).