FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	ETCS – DMI display II
Student:	Yury Udavichenka
Supervisor:	Ing. Jiří Chludil
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

ETCS (European Train Control System) je jednotný celoevropský vlakový zabezpečovací systém. Cílem této práce je pro simulátor železničního vozidla vytvořit displej strojvedoucího (DMI – Driver Machine Interface).

1. Analyzujte dostupnou dokumentaci k ETCS (subset ERA-ERTMS-015560v2.3) s ohledem na fungování DMI.
2. Analyzujte bakalářskou práci Jana Stejskala (FIT ČVUT) zaměřenou na DMI displej a dalších prací věnovaným DMI.
3. Na základě konzultací se zadavateli z Fakulty dopravní definujte funkční a nefunkční požadavky rozšiřujících funkcionalit předchozí verze DMI displeje.
4. Pomocí metod softwarového inženýrství a s využitím externí grafické knihovny navrhněte vybrané rozšiřující funkcionality jednotlivých obrazovek displeje.
5. Implementujte vybrané funkcionality prototypu DMI displeje, musí být zachována funkčnost všech předchozích verzí včetně BP Ondřeje Měšťana (FIT ČVUT), která vzniká paralelně k této práci.
6. Výsledný prototyp podrobte uživatelským, akceptačním a integračním testům.

Electronically approved by Ing. Michal Valenta, Ph.D. on 14 February 2023 in Prague.

Bachelor's thesis

ETCS – DMI DISPLAY II

Yury Udavichenka

Faculty of Information Technology
Department of software engineering
Supervisor: Ing. Jiří Chludil
May 11, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Udavichenka Yury. *ETCS – DMI display II*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

Acronyms ix

Introduction 1

1 Goals 3

2 Analysis 5
- 2.1 ERTMS/ETCS Analysis . . . 5
  - 2.1.1 Overview . . . 5
  - 2.1.2 ERTMS components . . . 5
- 2.2 DMI display simulator . . . 8
  - 2.2.1 Overview . . . 8
  - 2.2.2 Development history . . . 8
- 2.3 DMI display technical analysis . . . 14
  - 2.3.1 App architecture . . . 14
  - 2.3.2 Functionality . . . 15
  - 2.3.3 Source code analysis . . . 16
- 2.4 Results . . . 18
- 2.5 Functional and non-functional requirements . . . 19
  - 2.5.1 Functional requirements . . . 19
  - 2.5.2 Non-functional requirements . . . 20

3 Design 21
- 3.1 Runtime settings configuration . . . 21
  - 3.1.1 Motivation . . . 21
  - 3.1.2 Configuration file format . . . 21
  - 3.1.3 Accessing the configuration data . . . 22
- 3.2 Communication with lecturer's PC . . . 22
  - 3.2.1 Overview . . . 22
  - 3.2.2 Network configuration . . . 24
- 3.3 Multi-language support . . . 24
  - 3.3.1 Overview . . . 24
  - 3.3.2 String mapping . . . 25
  - 3.3.3 Language file format . . . 25
  - 3.3.4 Changing the display language . . . 26
- 3.4 Changing display resolution . . . 26
  - 3.4.1 Overview . . . 26
  - 3.4.2 Resolution configuration . . . 27

3.4.3 Model-View-Controller modifications . . . 27
3.4.4 Graphics library modifications . . . 27
3.5 Documentation . . . 28
3.6 Logging . . . 28
3.6.1 Overview . . . 28
3.6.2 Third-party logging libraries . . . 29
3.6.3 Custom logging library . . . 29
3.6.4 Choosing a library . . . 29
3.6.5 Logging implementation . . . 29
3.7 Extending functionality . . . 29
3.7.1 Overview . . . 29
3.7.2 Distance scale . . . 30
3.7.3 Orders, announcements, speed discontinuities . . . 31
3.7.4 Gradient profile . . . 31
3.7.5 Planning Area speed profile (PASP) . . . 31
3.8 Coding standards and guidelines . . . 31
3.8.1 Motivation . . . 31
3.8.2 Coding style guidelines . . . 32
3.8.3 General code quality advice . . . 32
3.8.4 Clang-Format . . . 32

4 Implementation **35**
4.1 Simulator layout . . . 35
4.2 Splitting responsibilities . . . 35
4.3 Installation manual . . . 36
4.4 User manual . . . 36
4.4.1 Changing a language . . . 37
4.4.2 Installing translations . . . 37
4.4.3 Setting up MQTT communication . . . 41
4.4.4 Changing the resolution . . . 41
4.5 Developer manual . . . 41
4.5.1 Setting up the environment . . . 41
4.5.2 Documentation . . . 42
4.6 Translator manual . . . 44

5 Testing **45**
5.1 User testing - application walkthrough . . . 45
5.1.1 Launching the application . . . 45
5.1.2 Changing display resolution . . . 45
5.1.3 Filling in train and driver data . . . 46
5.1.4 Starting the journey . . . 47
5.1.5 Changing the display language . . . 48
5.1.6 Adding and a new language . . . 48
5.2 Acceptance testing . . . 49
5.3 Integration testing . . . 49
5.4 Automated testing . . . 49

6 Conclusion **51**
6.1 Future work . . . 52

Contents of enclosed media **57**

List of Figures

2.1 Subsystems and components of ERTMS. Upscaled from [4] 6
2.2 Balises on Orivesi-Jyväskylä railway in Muurame, Finland (Leppänen 2009). Sourced from [5] 6
2.3 The default window of the DMI display. Sourced from [6] 8
2.4 An example of the DMI inside a train operator cabin. Sourced from [7] 9
2.5 ETCS simulator structure. Sourced from [12] 10
2.6 The DMI display wireframe. Sourced from [6] 10
2.7 The main window of the DMI display. Sourced from [6] 11
2.8 The DMI display application domain model. Sourced from [12] 13
2.9 Libraries used by the DMI display application. 15
2.10 The DMI application architecture. 16
2.11 An example of code with inconsistent formatting. Every single if statement in this section of the code is formatted slightly differently. 17

3.1 The Singleton design pattern. 22
3.2 The usage of the application context singleton by the application. 23
3.3 The interaction between the lecturer PC and the DMI display application. 24
3.4 The dictionary and how it interacts with the rest of the application. 25
3.5 The process of changing the display language. 26
3.6 An example of logging various events using the developer console. 30
3.7 The main elements of the planning information. Sourced from [6] 30
3.8 A screenshot of the DMI display application with the planning area shown. . . . 32

4.1 Git blame annotating individual lines of code inside Visual Studio. 36
4.2 An example of the DMI display application running on a Windows machine. . . . 36
4.3 The language settings window. 37
4.4 The train data window with the Czech localization selected. 38
4.5 The train data window with the English localization selected. 39
4.6 The train data window with the Russian localization selected. 40
4.7 The workloads necessary to build and run the project. 42
4.8 The welcome window of Visual Studio. 42
4.9 The window for specifying the repository path and location. 43
4.10 The solution explorer window. 43
4.11 The startup item selection. 43
4.12 The DMI display application running inside Visual Studio. 44

5.1 The train data input window. 46
5.2 The second part of the train data input window with all the required information filled. 47
5.3 The language selection window. 48
5.4 The list of languages after adding a new configuration file. 49
5.5 The new title of the settings window. 50

I would like to thank my supervisor Ing. Jiří Chludil for his invaluable advice, guidance and his directing me throughout the process of my writing this thesis. I would also like to thank my parents for giving me the opportunity to study in the Czech Republic and my family in general for supporting me for all these years. I'm very grateful to my English teacher back in Belarus for helping me achieve the level of English proficiency that I have today. And last but not least, a very special kind of thanks goes to my partner, K., for helping me stay sane during these past few stressful years of my academic career and giving me a reason to keep going despite all the hardships.

Declaration

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

In accordance with Section 2373(2) of Act No. 89/2012 Coll., Civil Code, as amended, I hereby grant a non-exclusive authorisation (licence) to use this copyright work, including all computer programs and all their documentation (hereinafter collectively referred to as "the Work"), to all persons who wish to use the Work. Such persons shall be entitled to use the Work in any manner that does not diminish the value of the Work, and for any purpose (including use for profit). However, any person who makes use of the above licence undertakes to grant a licence to any work that is based (even in part) on the Work, by adapting the Work, combining the Work with another work, incorporating the Work into an ensemble work, or adapting the Work (including translation), at least to the extent set out above, and also to make the source code of such work available in at least a comparable manner and to a comparable extent to the source code of the Work. This authorisation is unlimited in time, territory and quantity.

In Prague on May 11, 2023 .....

Abstract

This thesis deals with the development of the DMI display application, a component of a larger ETCS simulator project which fulfills the role of a visual interface between the driver and the train. This thesis concerns itself with the extension of an already existing DMI display application that was created as a collaboration between CTU in Prague's Faculty of Information Technology and Faculty of Transportation. Some of the goals include the analysis of all the previous work done on the application, the adding of missing functionality and the improvement of already existing functionality. Some of the focal points of the work deal with standardizing design and programming conventions, reducing technical debt caused by sub-optimal decisions during earlier development, and steering the project into something more manageable and easier to work on by subsequent teams of developers. Another goal was to modularize the project, making it less reliant on compile-time definitions and introducing better support for configuration files and localizations. Various software development and project management techniques were employed for this purpose. This thesis is being developed in parallel with a bachelor's thesis by Ondřej Měšťan that also concerns itself with the DMI display application.

Keywords DMI - driver machine interface, ETCS - European Train Control System, train simulator, technical debt reduction, language configuration files, graphics library extensions

Abstrakt

Tato práce se zabývá vývojem aplikace DMI displeje, která je součástí rozsáhlejšího projektu simulátoru ETCS. Aplikace displeje DMI plní úlohu vizuálního rozhraní mezi strojvedoucím a vlakem. Tato práce se zabývá rozšířením již existující aplikace DMI displeje, která vznikla v rámci spolupráce Fakulty informačních technologií a Fakulty dopravní ČVUT v Praze. Mezi cíle patří analýza veškeré předchozí práce na aplikaci, doplnění chybějících funkcionalit a vylepšení již existujících featur. Některé stěžejní body práce se zabývají standardizací návrhových a programovacích konvencí, snížením technického dluhu způsobeného neoptimálními rozhodnutími během předchozího vývoje a usměrněním projektu do podoby, která bude lépe zvládnutelná a na které budou moci lépe pracovat další týmy vývojářů. Dalším cílem je rozdělit aplikace na jednotlivé moduly, aby projekt byl méně závislý na definicích v době kompilace, a zavést lepší podporu konfiguračních souborů a různých lokalizací. K tomuto účelu byly použity různé techniky vývoje softwaru a řízení projektů. Tato práce vzniká souběžně s bakalářskou prací Ondřeje Měšťana, která se rovněž zabývá zobrazovací aplikací DMI.

Klíčová slova DMI - driver machine interface, ETCS - European Train Control System, vlakový simulátor, snížení technického dluhu, jazykové konfigurační soubory, rozšíření grafické knihovny

Acronyms

CTU	Czech Technical University in Prague
FIT	Faculty of Information Technology
DMI	Driver Machine Interface
ETCS	European Train Control System
ERTMS	European Rail Traffic Management System
EVC	Euro Vital Computer
JRU	Juridical recorder
RBC	Radio Block Centre
TIU	Train Interface Unit
LPC	Lecturer's PC
UI	User Interface
GUI	Graphical User Interface
API	Application Programming Interface
MVC	Model-View-Controller
SDL2	Simple DirectMedia Layer 2.0
PASP	Planning Area speed profile
IDE	Integrated Development Environment
OS	Operating system
PC	Personal Computer

Introduction

Trains play a vital part in our day-to-day lives. In the Czech republic, many people use them regularly to get around the country, be it travelling long distances for work, visiting family, going hiking, or simply as part of their daily commute. Trains are also invaluable for transporting large quantities of goods fast and efficiently. A robust train network means less reliance on cars, which in turn means less traffic congestion, more efficient usage of fuel, faster transportation and fewer emissions. Trains are also incredibly helpful for international travels, especially in Europe. Currently, Europe has several different standards for train networks currently in use, which often leads to inefficiencies, slowdowns and inconveniences when doing international travel. To address these concerns, the European Commission has headlined the European Rail Traffic Management System, a project that seeks to introduce a modern universal standard across the continent.

The European Train Control System simulator project is the result of cooperation between two faculties of the Czech Technical University in Prague: the Faculty of Information Technology and the Faculty of Transportation. The goal is to create a prototype that can simulate the behaviour of various parts of the ETCS that could be used as a tool for training the train drivers in the Czech Republic and teaching them how the system works and how to interact with it. In 2021, the project became one of the options for FIT's BI-SP1 subject, which required students to team up and work on something in cooperation with one another in order to meet a set of goals and requirements. Over time, the project became a staple for that subject as well as its successor, BI-SP2, and more and more teams of students got to contribute to its evolving feature set. The train simulator's goals and requirements have also kept evolving over time.

I was part of the original team that worked on the simulator's DMI display component. I then stepped away from it for a year, focusing on fulfilling other parts of my study plan as well as looking into other potential topics for my bachelor's thesis. Ultimately, I decided to come back to it, as I couldn't come up with a better topic and felt that my prior experience with the project would help me orient myself faster, as I also had to worry about maintaining a part-time job to sustain myself while working on the thesis. Another important factor is that, from my perspective, this project has actual real world value and application, and I generally prefer these types of projects to something that is done to fulfill a certain set of requirements but is otherwise completely pointless. My work experience in software development helped me identify various flaws in my and my team's old code, as well as the code that got added during the time of my absence. As a result, I decided to focus on improving the robustness and design of the currently existing application, to allow for easier expansion and maintenance by future developers as well as a better and more stable experience for the end user.

The thesis will focus on the DMI display component of the ETCS simulator. The individual chapters of the thesis shall cover analysis, design, implementation and testing of the application. This thesis is created in parallel with the bachelor's thesis of Ondřej Měšťan concerning the same component of the application, and as such his work will be referenced throughout mine.

Chapter 1
Goals

The goal of this thesis is to expand and improve the existing functionality of the DMI display application that will support the Windows and Linux operating systems. The goals of the analysis are to explore the current state of the project, the project's development history and some of its architectural design choices. This will include the analysis of existing theses related to the application's development. Functional and non-functional requirements are to be defined based on this analysis in coordination with the thesis supervisor representing the faculty as the application's customer. The goal of the design is to then extend the application's feature set. The way that will be done is by selecting some of the analyzed requirements in a way that prioritizes the most important missing features and features that would make the application better in the long run. Various methods of software engineering, such as design strategies and patterns, are to be used in order to aid development when appropriate. The subsequent goal is to then implement the proposed design while closely following the defined requirements and the supplied specification while also consulting with the application customer. Finally, the goal is to verify that the implemented features work as intended by using a range of user tests, acceptance tests and integration tests. All of the above should be done in coordination with Ondřej Měšťan, who's working on different extensions to the same application, in order to integrate our modifications into the main application.

The results of this thesis will be useful for future developers working on the project, and the goal of the project itself is to eventually create a fully-featured training simulator to help train drivers learn how to interact with the ETCS system, with the DMI display application playing a vital part in that training simulator.

Chapter 2
Analysis

This chapter concerns itself with the analysis of the work that has gone into the project over the course of its existence. First, it explores the European Train Control System and its various concepts and components. It then moves on to summarize the application's development history and does an in-depth analysis of Jan Stejskal's thesis concerning the application. Following that is a technical analysis of the project's current state. All of the aforementioned is then briefly summarized and discussed. The final section of the chapter is a set of functional and non-functional requirements that have been identified based on the results of the analysis.

2.1 ERTMS/ETCS Analysis

2.1.1 Overview

ERTMS, which stands for "European Railway Traffic Management System", is the European standard for the Automatic Train Protection (ATP) and command and control systems. Its stated goals are to create an interoperable railway system in Europe that is safer and more efficient than the systems used today. ERTMS is also a safety system that enforces compliance from trains with speed restrictions and signalling status. [1]

ETCS, which stands for "European Train Control System", is a train control standard, based on in-cab equipment able to supervise train movements and to stop it according to the permitted speed at each line section, along with calculation and supervision of the maximum train speed at all times. Information is received from the ETCS equipment beside the track (balises or radio) depending on the operation level. The driver's response is continuously monitored, and if necessary the emergency brakes would be taken under control. [2] Within the standard there are different ETCS levels and ETCS operation modes defined [3], each having different sets of functionalities with regard to signaling, train movement supervision, communication, etc.

2.1.2 ERTMS components

The ERTMS system is divided into several different components. These components belong to either of two subsystems: onboard and trackside. [4] Figure 2.1 gives an overview of the different components and how they are connected to one another. Below are the selected definitions of ETCS components that are relevant within the scope of this thesis.

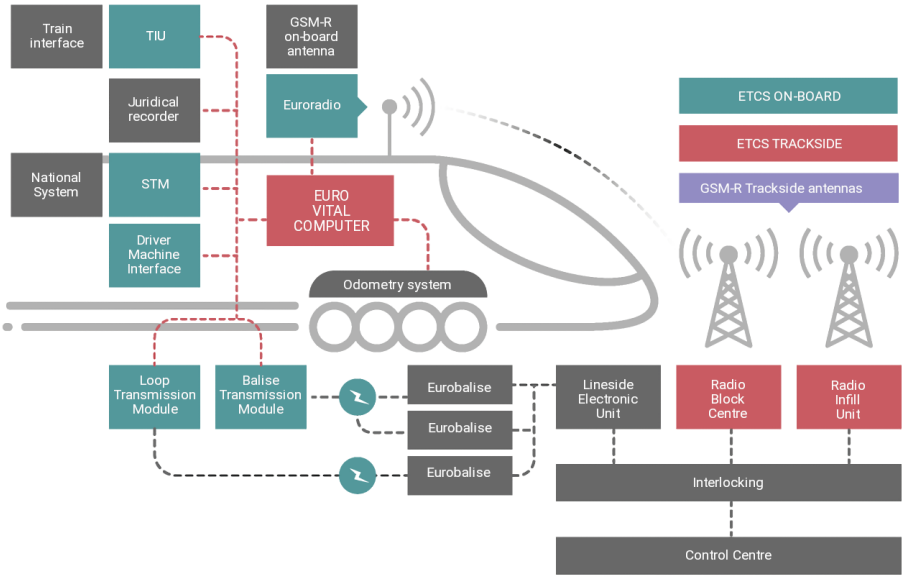

Figure 2.1 Subsystems and components of ERTMS. Upscaled from [4]

Figure 2.2 Balises on Orivesi-Jyväskylä railway in Muurame, Finland (Leppänen 2009). Sourced from [5]

Eurobalise

"The Eurobalise is a passive device that is installed on the track, storing data (fixed or switchable, i.e. with the possibility of changing information content) related to the infrastructure, such as speed limits, position references, gradients, etc. It is a passive device because it does not need an electric supply since it is the train antenna (BTM) that energises it when passing over it." [4] An example of the Eurobalise can be viewed on figure 2.2.

RBC (Radio Block Centre)

"The RBC is a device used at ETCS Level 2 and Level 3 acting as a centralised safety unit, which, using radio connection via GSM-R, receives train position information and sends movement authority and further information required by the train for its movement. The RBC interacts with the interlocking to obtain signalling-related information, route status, etc. It is also able to manage the transmission of selected trackside data and communicate with adjacent RBCs." [4]

EVC (Euro Vital Component)

"The EVC is the core of the ETCS onboard device. It is part of the Automatic Train Protection logic and is the unit with which all the other train functions interact, such as the odometer or the GSM-R data reception." [4]

DMI (Driver Machine Interface)

"The DMI is the interface between the driver and the ETCS. In most cases, it is an LCD touch screen panel for control and indication functions, allowing the driver to enter the required input data into the system and to visualise the output data." [4]

TIU (Train Interface Unit)

"The TIU is the interface that allows the ETCS to exchange information (e.g. ETCS will receive the status of the direction controller in the cab: forward, neutral or backward, which represents the direction of the train movement, through the TIU) and issue commands to the rolling stock (e.g. ETCS will send the rolling stock the command to apply the brakes)." [4]

JRU (Juridical Recording Unit)

"The Juridical Recording Unit provides 'black box' functions, i.e. it stores the most important data and variables from train journeys, allowing later analysis." [4]

BTM (Balise Transmission Module)

"The BTM is a module inside the ETCS onboard equipment for intermittent transmission from track to train, which processes signals received from the onboard antenna and retrieves data messages from a Eurobalise." [4]

Odometry system

"The odometer is responsible for calculating the distance run by the train, typically consisting of redundant tachometry and radar, able to calculate distance, speed and acceleration." [4]

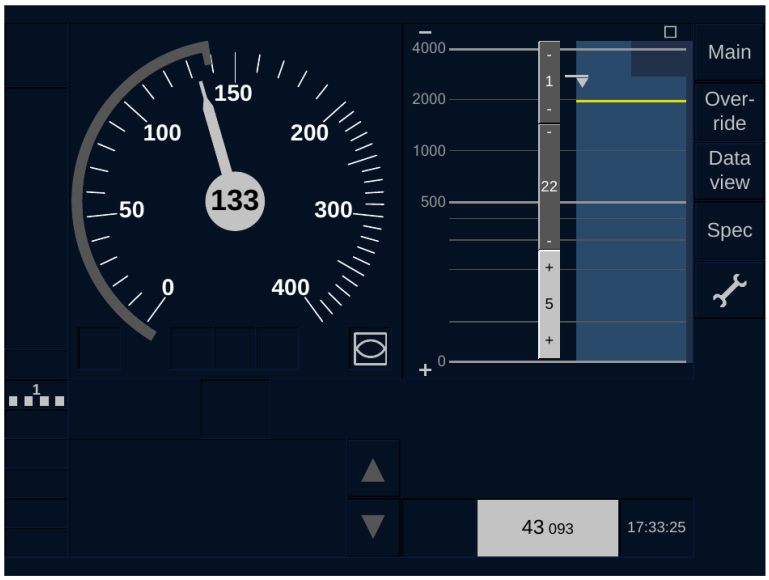

Figure 2.3 The default window of the DMI display. Sourced from [6]

2.2 DMI display simulator

2.2.1 Overview

The DMI display simulator is a component of the larger ETCS simulator being developed at FIT CTU. Historically the application is based on the ERA-ERTMS-015560 subset version 2.3 [6]. This thesis will also be focusing on this version of the subset. Although there are newer versions available, they do not significantly differ from the one that's currently used for the project. A lot of the changes are visual, such as adding more icons for acknowledgements or changing certain colour values.

2.2.2 Development history

This section of the analysis will cover the development history of the application.

2.2.2.1 Start of development

The DMI display simulator project began development in the spring of 2021, with the design proposal being finished in May [8]. The design proposal outlined a demo version of the DMI display that would contain the default window and the main window with the "Start" button. The demo application was to use the SDL-based graphics library provided by Petr Stříteský [9] for drawing and extract dimensions of various screen elements from configuration files. Communication with EVC was to be done through the MQTT protocol and the JSON file format. The MVC (Model-View-Controller) design pattern was settled on for the purpose of defining and drawing individual parts of the display. C++ was chosen as the programming language for the development of the application and Microsoft Visual Studio 2019 was the development environment used, although the application was intended to be cross-platform.

Over the course of the next several months, a large part of the original team went on to work on implementing the demo, myself included. To aid with implementing certain parts of the application, several new libraries were introduced to the project, such as Mosquitto [10]

Figure 2.4 An example of the DMI inside a train operator cabin. Sourced from [7]

and JSON for Modern C++. [11] The graphics library was also heavily modified to better suit the purposes of the application. The DMI team worked with representatives of the Faculty of Transportation to align on requirements as well as other teams working on the simulator (most notably EVC) to agree on the details of interplay with the rest of the simulator. The ERA ERTMS subset specification [6] was closely followed.

2.2.2.2 Jan Stejskal's thesis

Jan Stejskal is one of the people from the original team of students working on the project. The result of his work is the "ETCS — DMI Display" bachelor's thesis [12]. In it, he described a lot of the work done during the initial development period as well as the work done by him while writing his thesis. He analyzed the available documentation, the ERA ERTMS subset [6], and described in detail a lot of the individual components making up the display.

Analysis of the available documentation The first section of Stejskal's thesis covered the documentation of the subset version 2.3 [6] and the various concepts that the subset's documentation introduced. He covered the basic concepts introduced in and used by the documentation, such as the recommended fonts and font sizes, as well as sounds and symbols packaged with the subset. He then described the size and resolution of the display, the colours used by the application, brightness and volume levels, and the display's ability to change languages. The analysis covered what the purpose of different windows and buttons were and how they were supposed to work.

The default window is covered in more detail by this section. The default window can be considered the primary window of the DMI display, and the window that the driver will spend the most time looking at. It provides information such as breaking curve data, the planning area, various acknowledgements and notifications, the speedometer, the message list, monitoring information, as well as a set of buttons used for navigating to other windows. The default window is pictured on figure 2.3. The default window section of the analysis covers all of the window's parts in more detail while also providing references to the documentation.

Various sub-level windows were also covered in that section of the analysis. Such sub-level

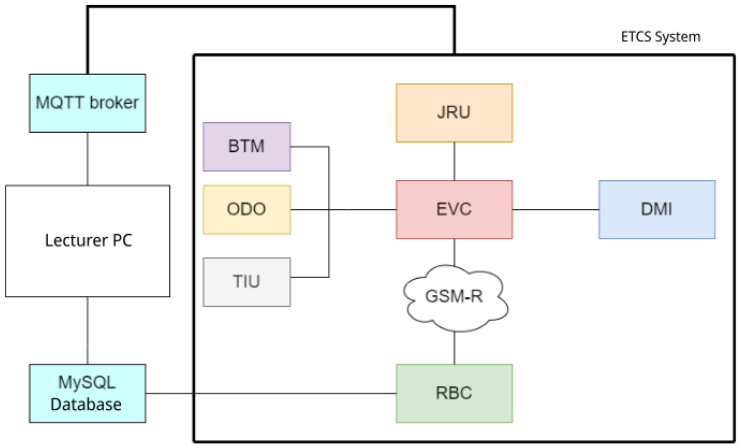

Figure 2.5 ETCS simulator structure. Sourced from [12]

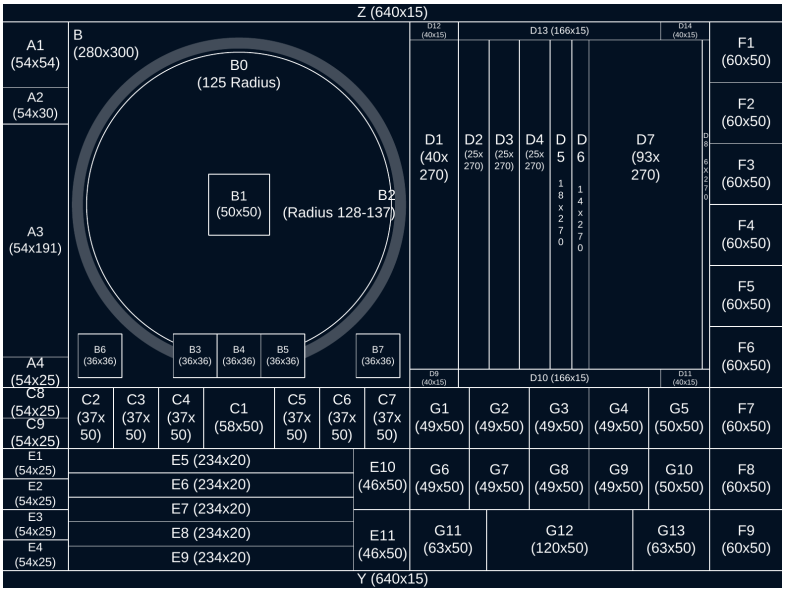

Figure 2.6 The DMI display wireframe. Sourced from [6]

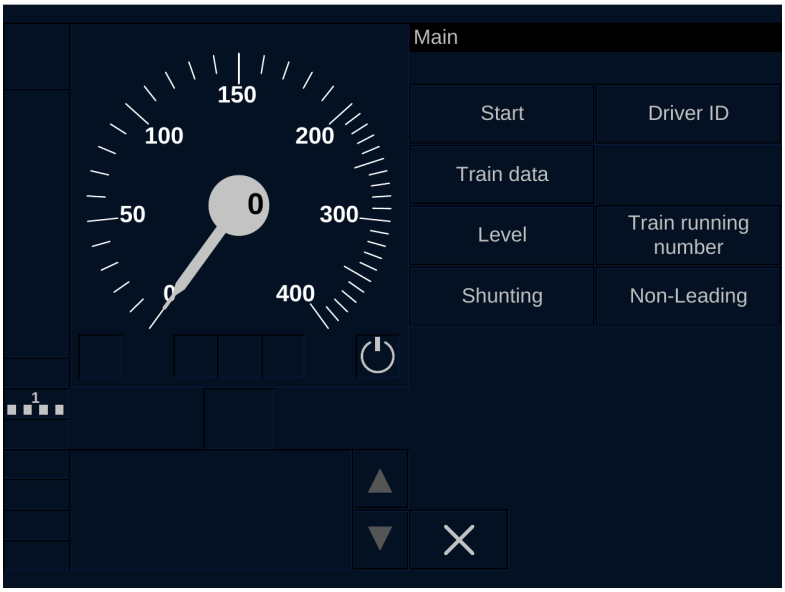

Figure 2.7 The main window of the DMI display. Sourced from [6]

windows include menu windows, data input windows, data validation windows and data overview windows. Their individual components as well as the method of data input and validation is then elaborated on further. The main window is given as an example of such sub-level window. Figure 2.7 shows an example of the main window. Finally, the section closes by describing various dialogue sequences present in the design of the DMI display.

Analysis of the graphics library The next section of Stejskal's thesis contains the analysis of Petr Stříteský's graphics library that was the end result of his respective thesis [9]. Stejskal describes how the graphics library was built and mentions the DMI display application's use of it. He further dissects the library's dependencies, file structure and functionality, describing what can be achieved by utilizing various functions that the library contains. Stejskal also describes some recommendations for how to best utilize the library. He closes the section by describing the various problems discovered within the library. He gives one such example of the text drawing functions needlessly recreating the textures for drawing the text every frame instead of creating those textures once and reusing them afterwards.

Functional and non-functional requirements This section contains the functional and non-functional requirements gathered from the analyses performed as well as consultations with the clients at CTU's Faculty of Transportation. Stejskal then goes on to sort the functional requirements according to their priority in order to help choose what to implement within the time constraints of the project.

Design This section concerns itself with the design of the DMI display application. The author notes that the design described in this section stems from the original design of the application that was created during the 2020 run of the BI-SP1 subject by our team, with some changes that were made later on in the project's development cycle. The visual design of the application was not altered or described in this chapter, as it had been previously described in the section concerning the analysis of the available documentation.

MQTT was chosen as the communication protocol between the DMI and the EVC. MQTT is a lightweight publish/subscribe messaging protocol. It's based on the principle of publishing

messages and subscribing to topics. Clients connect to a broker and subscribe to topics they are interested in. Clients also connect to the broker and publish messages to topics. Many clients can subscribe to the same topics, and MQTT as well as the broker act as a simple, common interface for everything to connect to. Adding new sensors to topics is trivial. [13]

Overall, MQTT is a good choice for the purposes of the ETCS simulator. It's lightweight, easily extensible and easy to work with, leading to low development complexity and runtime overhead. The specific MQTT implementation used by the DMI display application uses the Mosquitto library [10] and the TCP/IP communication protocol. The client implemented on the DMI display side connects to the MQTT broker on the EVC side and exchanges data with it in the form of a JSON file. [14] This means that the client expects to receive data formatted as a JSON file from the broker and formats its own data into that format before sending it to the broker. JSON is also a great choice for the format for exchanging communication data. It's simple, easy to parse and work with, and is also readable to humans due to how minimalistic and concise it is. These characteristics make it a popular choice for web APIs around the world.

To make parsing and serializing JSONs easier, the JSON for Modern C++ [11] library was chosen. Its professed design goals are intuitive syntax, trivial integration, serious testing, memory efficiency and speed. [15] The library proved to be great for the purposes of the application thanks to its ergonomics and performance.

According to Stejskal, the provided graphics library required some modification and fixes to make it more suitable for the application. To improve its performance, certain functions were modified and made to create certain textures only once during the initialization process of the application, instead of every time the function is called.

The individual windows of the application are maintained by the coordinating component. The purpose of the coordinating component is to enable switching between and displaying individual windows and their components. The coordinator is based on the Observer design pattern. [16] For the core architecture of individual windows and window sections, the choice was the Model-View-Controller (MVC) design pattern, to help with decoupling the display logic, the data model and the user interface from each other.

The responsibility of the *Controller* is to react to various events and update the necessary data stored inside the Model with the newly received data from EVC. The controller is also able to send data to the EVC if necessary. The *Model* concerns itself with storing all the required data of the individual component. A lot of that data is stored in and read from the JSON configuration files. Stejskal provides the positions, dimensions and colours of various objects as examples of such data. Lastly, the *View* component is responsible for drawing the contents of the component based on the data currently stored inside the Model of the component.

The default window was separated into 9 different areas with objects to display. As a result, the default window contains the MVCs for each one of those areas, that work completely independently of each other and only concern themselves with controlling their own respective areas. This makes working with individual areas and objects within those areas easier and more straightforward. The Observer pattern [16] was then used to enable communication between controllers, with individual controllers listening to changes on certain other controllers and updating their own state if needed. Other windows, on the other hand, for the most part only contain their own MVCs that control the entire window, as their internal structure is less complex.

The MVC pattern is common in user interface design. Examples of its use can be found in technologies such as Microsoft's ASP.NET Core. [17] There are variations of implementation details of the pattern with no clear winner, and as such the usage of the pattern for the purposes of the DMI display application was good and functional. Another benefit of the Model-View-Controller pattern is that it can scale up or down in complexity depending on the individual needs to every component, as proven by its slightly different application between various application windows.

On the other hand, the Observer pattern was not used correctly. The Observer is supposed to be a thin interface that can be used to enable communication between otherwise independent

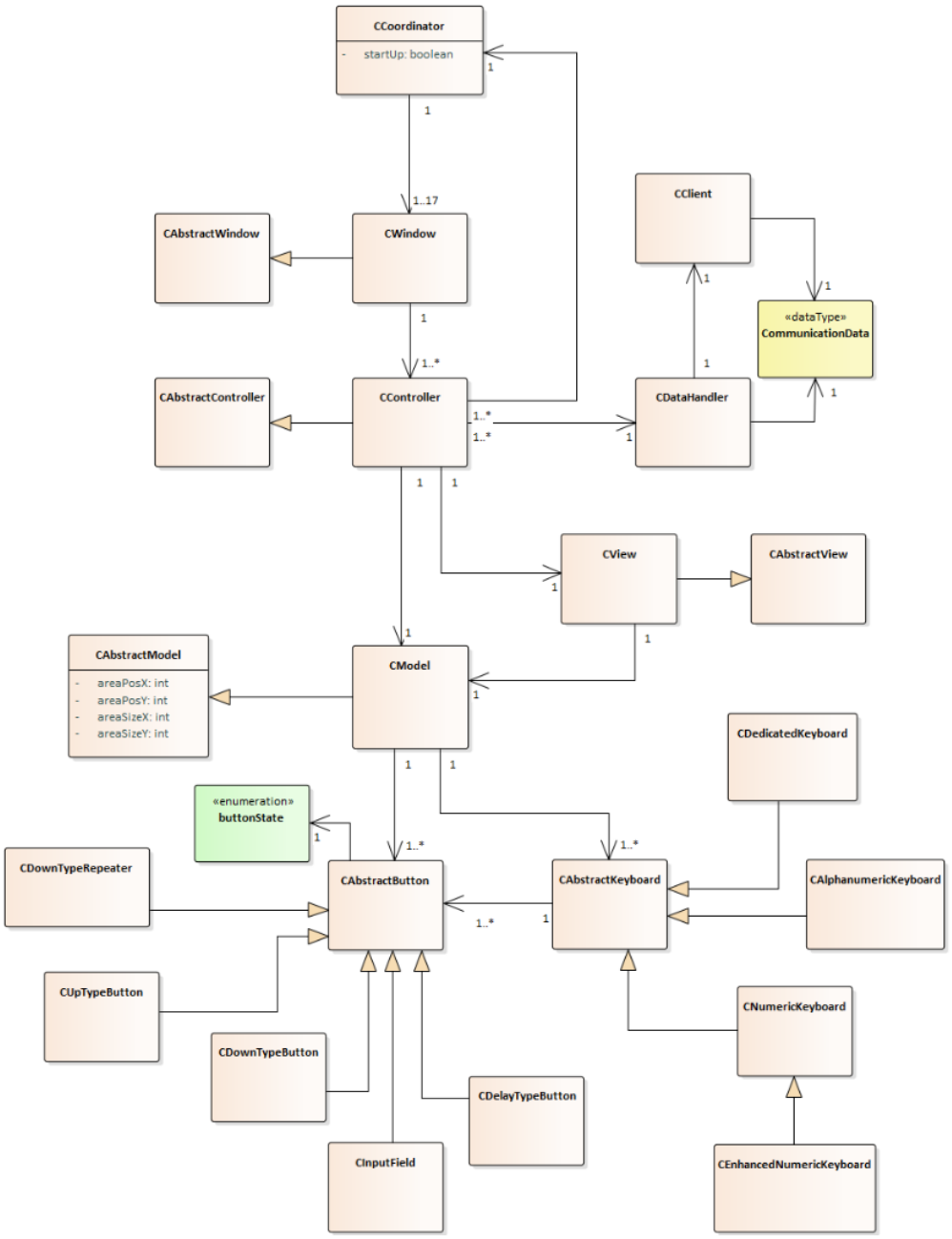

Figure 2.8 The DMI display application domain model. Sourced from [12]

objects. While the coordinator does have Observer functionality, it also handles a lot of other things, such as handling user input and rendering. This violates the separation of concerns principle of object-oriented programming and complicates further development and maintenance.

The next part of the section describes various processes present in the application, such as communication from EVC to DMI, communication from DMI to EVC, and switching between windows. Stejskal closes the section by describing the changes made to the original design of the application. These changes include moving user input detection logic from View components into Controller components and removing the communication data validation object in favour of simply processing the data received by the DMI from the EVC and logging errors into a file.

Implementation In this section Stejskal put manuals for installing, running and developing the application. While the installation manual has sections for both Linux and Windows, the developer manual focuses primarily on the Windows platform. The programmer's manual also contains a description of the project's file structure. Following is a demonstration of the application's user interface and a user manual containing step-by-step guides on performing various operations with the application. The section closes with a schema of the ETCS application architecture, as shown on figure 2.5.

Testing This section describes the user tests, acceptance tests and integration tests that the application was subjected to. It describes the steps taken for several individual tests, with each having a structure of estimated time, scenario purpose, starting point, ending point, instructions, expected steps and critical points. Stejskal then describes the acceptance tests done by Ing. Martin Leso, Ph.D. from CTU's Faculty of Transportation. After that follows a table describing the state of each implemented functional requirement at the time of the acceptance testing, with a note that the state described in that table is not up-to-date with the actual state of the application, since development of the application continued after that point. The section closes with a description of how integration testing was done.

2.2.2.3 Further development

The result of Jan Stejskal's work was a prototype of the DMI display application. Work on the application was then continued by various teams from FIT CTU's BI-SP1 and BI-SP2 subjects. The current state of the application was analyzed and described by the team of students working on the DMI display project during the 2022-2023 academic year run of the BI-SP2 subject.

2.3 DMI display technical analysis

The following section will describe the architecture of the current iteration of the DMI display application, the libraries and design patterns used, the functionality implemented and the quality of the code.

2.3.1 App architecture

2.3.1.1 Libraries

The libraries used by the project remain unchanged. This list contains the following:

- Simple DirectMedia Layer 2.0 [18]
- SDL2-based graphics library [9]
- Eclipse Mosquitto [10]

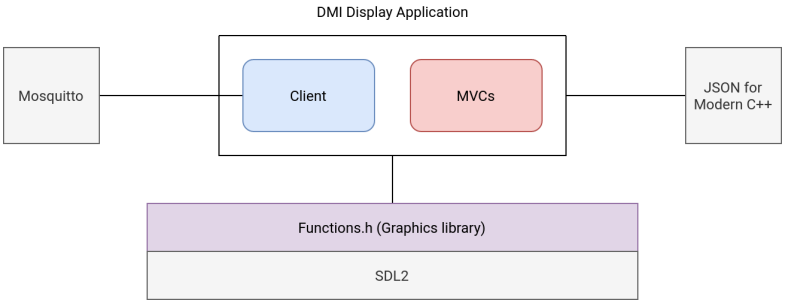

Figure 2.9 Libraries used by the DMI display application.

- JSON for Modern C++ [11]

It should be noted that the graphics library source file contains not only the functions used for drawing various objects, but also functions used to initialize or close the application, functions for processing user input, as well as some other miscellaneous functionality. An overview of external library usage is illustrated on figure 2.9.

2.3.1.2 Application

The DMI display application can be logically split into two parts: the network client and the display itself. The network client handles communication with other parts of the ETCS simulator by means of exchanging messages with the simulator's EVC component. The client receives regular messages from the EVC containing data, such as the current and permitted train speed, the target speed and distance, the gradient profile, the speed profile, ETCS mode and level changes, fixed text messages, and so on and so forth. The client is also responsible for sending messages over to the EVC. Examples of such messages include the driver ID, the ETCS level at the start of mission, radio network data, train data, etc. The messages are structured as JSON files. The specific contents of these messages are described in the project's internal documentation.

These messages can have relatively complex structures, and some messages can be very different from one another. Since MQTT is the protocol used for communication by both parties and every message goes through the same topic, message types don't get automatically resolved once they reach their destination. To help with differentiating and parsing these messages, each one contains an `NID_MESSAGE` field containing the ID of the type of that message. Messages received by the DMI have their IDs in the `2XX` range, while messages received from the DMI by the EVC have their IDs in the `3XX` range.

The second part of the application, the display itself, is comprised of MVCs of various display areas, windows containing these MVCs, and various components used inside the MVCs, such as buttons or input fields. The "glue" that connects the Model-View-Controllers and windows together with the client and enables exchange of information from the network to the user and back is the coordinator. The coordinator initializes display components, keeps the display components up-to-date with the current state of the application and the data received from the client, switches between different windows and presents them to the screen for the user to see and interact with. Figure 2.10 showcases this relationship.

2.3.2 Functionality

Since the publication of Jan Stejskal's thesis, [12] the DMI display application and its feature set have evolved significantly, with several different teams contributing to the project's codebase. The state of the project at the end of the B222 semester is covered in detail in Ondřej Měšťan's thesis that's being written in parallel to this one. [19]

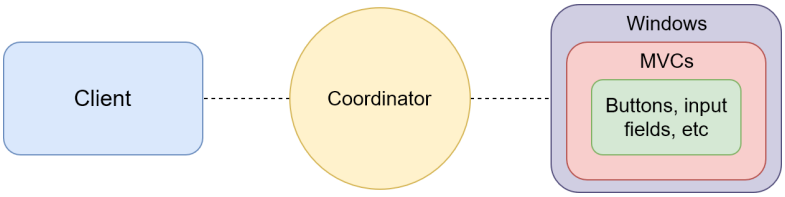

Figure 2.10 The DMI application architecture.

2.3.3 Source code analysis

Code quality is something the end user never sees, but it's something that still affects the user every time they use any piece of software. A program written in high-quality code is generally going to be more performant and reliable than one with worse coding standards, even if both programs contain an identical set of features. Code quality also matters a lot for the developer. A well-written, high-quality code base is easier to parse, maintain and expand. Bad and short-sighted code can prevent the developer from introducing certain features and frequently introduces technical debt. Technical debt is a concept that reflects the extra development work that arises when sub-optimal code is introduced. [20] In a project with a lot of technical debt, developers have to spend extra time working around or refactoring the problematic areas of the code, and sometimes the short-term solutions to technical debt end up piling on more and more of it. For these reasons, as part of my analysis, I wanted to explore the code quality of the DMI display application.

2.3.3.1 Coding style

Coding style, or coding conventions, refers to the general naming, formatting, code structuring and indentation conventions utilized for a software project. A good and consistent coding style helps make the code more readable and easier to understand. Some programming languages have clearly defined and globally recommended coding conventions. Java is an example of one such language [21]. Other languages (for example, Rust) even go as far as to ship a formatting tool together with their compiler toolchain to ensure uniformity and ease the cognitive load on programmers having to apply that style. [22] Some older languages, such as C or C++, don't have a commonly agreed upon coding style, and as a result the way the source code looks varies from organization to organization and frequently even from individual to individual. Sometimes programming language developers even advocate against automatic formatters, arguing that sometimes it's important to break the consistency when appropriate. [23]

The DMI display application project did not originally have a set of agreed-upon coding conventions. It was started by a team of very inexperienced students, for the majority of which the project was their first foray into working on software together with other people, so the idea to establish some set of code style guidelines did not surface during the design and implementation stages of the first prototype. This, coupled with the fact that the project was done in C++, resulted in a lot of the code being very messy and inconsistent with regards to naming and indentation. Inconsistent code then in turn creates a barrier of entry for anyone working with the project, as they have to spend time not only familiarizing themselves with the inner workings of the application, but also adapting to the code style and structure changing from source file to source file. A snippet of particularly inconsistent formatting from the application's code base can be seen on figure 2.11.

```
    while (!q.empty()) {
        temp = q.front();
        q.pop_front();
        if (temp.D_STATIC == 0) continue;
        if (temp.V_STATIC > prevSpeed) {
            isSpeedIncreaseFound = true;
        }
        if (isSpeedIncreaseFound && temp.V_STATIC < prevSpeed && temp.V_STATIC != 0)
        {
            continue;
        }
        int sect = CalculatePASPSection(prevSpeed, permSpeed);
        height = temp.D_STATIC;
        CalculateSSPPosition(height);
        if (height == m_Model->outOfRangeBottom)
            height = m_Model->D7SizeY;
        DrawPASPRectangle(height, rectWidths[sect]);
        if ( ! isSpeedIncreaseFound && temp.V_STATIC != 0 ) prevSpeed = temp.V_STATIC;
    }
```

Figure 2.11 An example of code with inconsistent formatting. Every single if statement in this section of the code is formatted slightly differently.

2.3.3.2 Code quality

As mentioned above, code quality makes it easier for developers to catch bugs and introduce new features. A well-thought-out class structure in object-oriented applications makes the app's individual components modular and easily swappable, reducing the amount of time developers have to spend on decoupling existing parts of the code from one another in order to make changes. A badly-designed class structure leads to unnecessary complexity and makes developers spend large amounts of time on refactoring existing code instead of introducing new code. However, software design is a very challenging and treacherous process. Some decisions may seem to make sense at the start of the project's lifetime, but as the code base evolves, problems tend to surface more often than not. This necessitates regular refactoring in order to maintain old code and remove dead, unnecessary code.

In general, a lot of code quality principles applied in software development today can be considered quite subjective, as it's difficult (if not impossible) to measure the positive impact of adhering to them, nor are they based on any hard data showcasing that a certain way of doing things is better than others. However, a certain set of measurements can be tracked across several approaches to software design:

1. Good code needs to be readable and easy to understand,
2. Good code needs to be consistent,
3. Good code needs to be easily testable,
4. Good code needs to be extensible,
5. Good code needs to be reusable,
6. Good code needs to be well-documented.

The readability and consistency of code has already been touched on in the previous section. The following section will explore the remainder of these requirements.

Testability There are various different types of software testing, and code can be described as testable when it lends itself easily to a wide variety of them. Unit tests validate that each software unit performs as expected. A unit is the smallest testable component of an application.

Integration tests validate that software components or functions are able to operate together as intended. Regression tests check whether new features break or degrade functionality. [24]

Currently, the DMI display application has no automated tests. Classes are generally not designed in a way so as to facilitate unit or integration testing, and no test frameworks are used. The only way to verify whether the application works correctly is via manual user testing.

Logging is another large contributor to the testability of any code base. A robust logging system can help trace errors and figure out which parts of the code are problematic. As of today, the project has some very rudimentary logging capabilities, such as printing out messages received by the client from the EVC into the developer console, but most components lack any logging functionality, making them harder to debug in case problems arise.

Extensibility The DMI display application has functionality that lets it load certain parameters from configuration files on startup. Unfortunately, the application doesn't expect a lot of these parameters to be different from what they are currently set to, and as such problems may arise if the configuration was to be changed. Furthermore, certain variables necessary for the functioning of the application are defined as compile-time constants, meaning that if a user wants to change some of them, they have to either have to be provided with a custom binary by the developer, or to have access to the source code as well as the development tools and to change the parameters themselves. One example of such parameter is display resolution and the related `SCALE` parameter. Various different layers of the application depend on that parameter, and as a result changing the resolution of the application is not as trivial as one would expect. Another example is the fact that the list of available languages is strictly defined inside the application's source code and translations cannot be added by regular users.

Reusability Code duplication should generally be limited as much as possible. Unfortunately, currently it's easy to find many examples of duplicated code in the code base. For instance, the same several lines or code are used to read out of a JSON configuration file in virtually every display area, however, those lines of code are always copy-pasted instead of being extracted into a function.

Documentation Significant parts of the DMI display application aren't documented, with comments missing where they should be present. On the other hand, the source code files also contain large chunks of commented out code that does nothing but hurt readability and potentially even confuse the reader. Perhaps most crucially, the graphics and utility library used by the DMI display app has quite poor documentation.

2.4 Results

Based on the results above, we can paint a picture of the DMI display application as an application with plenty of good and bad decisions alike. For example, the choice of the configuration file format and the library for their processing, the network communication protocol and the Model-View-Controller design pattern were quite successful. On the other hand, the implementation of the Observer design pattern wasn't done correctly, leading to unnecessary complexity. The fact that changing the resolution of the application requires recompilation of the entire project is an obvious design flaw. The inconsistent coding style and quality coupled with lacking documentation cause big hurdles in developers' ability to improve and expand the application. As such, the code base would benefit from some standardization. The lack of proper logging makes it difficult to find and resolve bugs, and the lack of an automated testing suite can make it hard to notice regressions. Adding more language options is a challenge in the current iteration of the application. In general, a lot of design decisions in this project feel either rushed due to time constraints or badly thought out due to inexperience.

2.5 Functional and non-functional requirements

Based on the analysis detailed above, the following functional and non-functional requirements have been defined:

2.5.1 Functional requirements

F1 - Language changes

The application should be available in multiple different languages. It should also allow users to easily add more language options through configuration files.

F2 - Resolution settings

The application should support resolutions higher than 640x480. In addition to that, the application should correctly scale its interface depending on the selected resolution, making sure that click areas do not drift away from their corresponding elements displayed on the screen.

F3 - EVC communication

The application should be able to properly process and respond to every message it gets from the EVC.

F4 - Network heartbeat

The application should be able to continuously send heartbeat messages to the lecturer's PC as long as the DMI display application is running.

F5 - Network control messages

The application should be able to receive and process messages from the lecturer's PC regarding the app's control flow (turn on, turn off, etc).

F6 - Maintaining current functionality

The application should retain all the functionality it had in its previous iterations, as defined by the ERA-ERTMS subset v2.3. [6]

F7 - Introducing new functionality

The application should introduce functionality that it is currently lacking compared to the specification in the ERA-ERTMS subset v2.3. [6]

F8 - Logging

The application should have expanded logging capabilities, making it easier to debug and identify problems.

2.5.2 Non-functional requirements

N1 - Programming language, libraries and the operating system

The project should continue to be developed using C++, communication should continue to be handled using the MQTT protocol, and the application should continue to use the JSON file format for configuration files. The SDL2, Mosquitto and JSON for Modern C++ libraries should continue to be used for developing the application. The resulting application should retain cross-platform capabilities, enabling it to be ran on Linux and Windows operating systems.

N2 - Code style guidelines

Clear and consistent code style guidelines should be defined and applied to both newly-introduced as well as previously-existing code. ClangFormat should be used to help enforce the defined guidelines across the entire project and all of its present and future contributors.

N3 - Code quality standards

Code quality standards should be improved to aid app stability, performance, testability and maintainability as well as make it easier for developers to familiarize themselves with the code base, introduce new features and improve existing ones.

N4 - Library documentation

The graphics and utility library should be well-documented, with explanations for what individual functions do and how to properly utilize them.

Chapter 3
Design

This chapter concerns itself with the design of the selected extensions to the application and the architectural decisions that will govern their eventual implementation. The various sections of the chapter cover processing configuration files at runtime, communication with the lecturer's PC, support for multiple languages and dynamically changing the display resolution. The chapter also covers various changes to enable all that, such as improving the graphics library layer, error logging and expanding documentation. Finally, the chapter closes with a description of features implemented by me before starting work on this thesis and the establishment of consistent coding standards and guidelines to enable better coordination between people working on the project.

3.1 Runtime settings configuration

3.1.1 Motivation

According to the specification, the minimum resolution of the DMI display should be based on a grid of 640x480 cells. The 4:3 aspect ratio should be the basis for object proportions, but the resolution itself is expected to be changeable depending on the parameters of any given screen. [6] Furthermore, this requirement makes even more sense for a simulator of the component, as it could be used on a much wider range of devices.

The idea of this change is to allow the resolution to be set via a configuration file instead of a compile-time constant. The application will then read from the configuration file upon launch and recalculate the scale and sizes of individual components of the display accordingly. This will help avoid recompiling the entire program for such a simple adjustment as well as let end-users change the setting to their liking instead of having to go into the code and rebuild the application. The system used for implementing this change should also be flexible enough to allow for extensions and moving other hard-coded parameters that the display program requires during its initialization procedure into configuration files.

3.1.2 Configuration file format

In the past, the JSON file format was selected as the format of choice for configuration files that the DMI display application uses. This has proven to work quite well over the course of the project's lifetime, and so the resolution configuration shall also be stored as a JSON file and processed using the JSON for Modern C++ library [11].

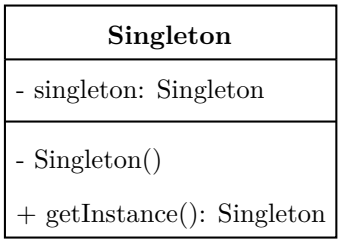

Figure 3.1 The Singleton design pattern.

3.1.3 Accessing the configuration data

3.1.3.1 The problem

The main design challenge of this feature is that variables such as screen width, height or logical scale should be accessible from any point of the application, as they're used not only during the app's initialization for creating the application window, but also throughout the application's runtime by various display area MVCs. At the same time, it cannot be a global compile-time constant, because that would go against the very point of this feature. Finally, we don't want to be passing an object containing this data everywhere where we could potentially want to use this, as it would be a nightmare to work with while also violating the high cohesion, loose coupling design principle of object-oriented programming. For these reasons, the singleton design pattern was chosen.

3.1.3.2 Singleton

The singleton is a creational design pattern that ensures an object can only have one instance, while also having that object be globally accessible [25]. This is perfect for our use case, as there can only be one screen resolution, and having that guarantee means that we can access this data in a safe an reliable way throughout our application's life cycle. Having the object be globally accessible will help us access it anywhere we need, such as inside the methods and functions performing the display initialization or the underlying graphics library layer translating the logical size to pixels. Having the variables be encapsulated in an object also gives us an additional guarantee of stability that a globally defined struct wouldn't.

We will apply the pattern to introduce an application context class to our application, which will contain the aforementioned variables as well as toggles for various debug levels, screen modes such as windowed or fullscreen, and potentially any other parameter that we would want to have similar access to. Adding more parameters is trivial, allowing for flexibility and extensibility further down the line. The application context singleton shall read the configuration file upon instantiating the singleton and store the necessary variables, which will then get accessed from the singleton object using getters.

3.2 Communication with lecturer's PC

3.2.1 Overview

The DMI display application should be able to handle communication between itself and the lecturer's PC. This communication can be broadly split into two categories. Firstly, the application needs to be able to send out a periodic heartbeat to the lecturer's PC to indicate that the application is currently up and running. Secondly, the application needs to be able to receive and handle various control messages from the lecturer's PC, such as start, stop, and so on.

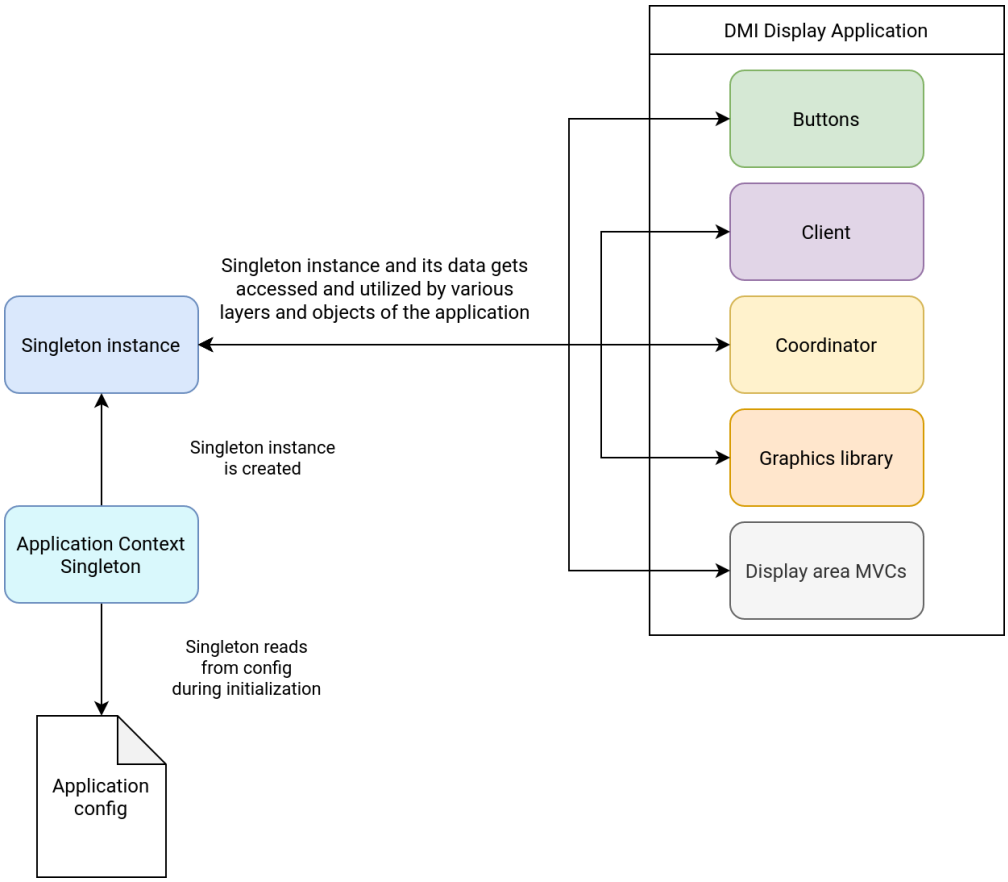

Figure 3.2 The usage of the application context singleton by the application.

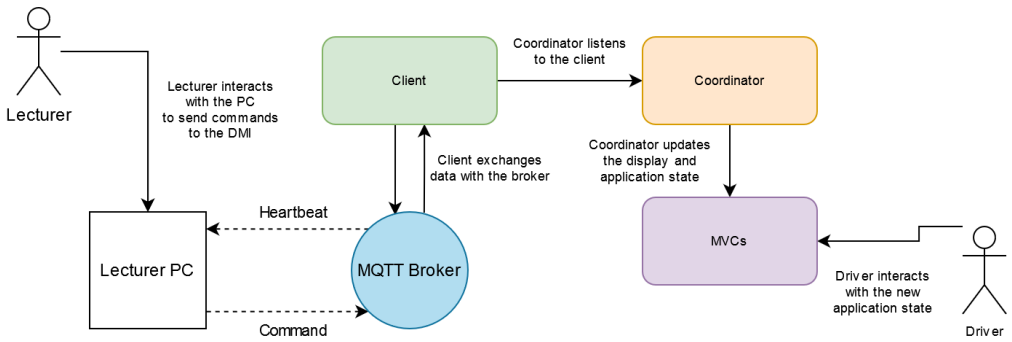

Figure 3.3 The interaction between the lecturer PC and the DMI display application.

The current DMI display application has an MQTT client implementation that uses the Eclipse Mosquitto message broker [10]. The functionality enabling communication with the lecturer's PC shall be added into the already existing client.

3.2.1.1 Heartbeat

A heartbeat is a term to describe an application sending regular messages to another place on the network. In our case, the purpose of a heartbeat is to let the lecturer PC know whether the application is up or down. This will then let the lecturer know whether to send the application various control messages or to try and diagnose the reason for the absence of heartbeat messages, be it due to a crash, problems with the connection, or something else.

3.2.1.2 Control messages

Control messages are messages that are sent from the lecturer's PC to the DMI display application to change the application's state in a certain way. For example, they could be used to start, stop or reload the application, load different configurations or subsets. The control message structure will consist of a specific tag describing the generic type of a message followed by various relevant data fields, depending on the control message type. Examples of such data fields include the name of a subset or a specific configuration file.

3.2.2 Network configuration

To enable effective network communication as well as expand on the efforts described in the previous section, the network address of the lecturer PC will be fully configurable. The rationale for this is similar to the one behind extracting resolution parameters into a config file as described in the previous section.

3.3 Multi-language support

3.3.1 Overview

The DMI display application should be able to handle multiple different languages, adding new languages should be easy, and it should support the addition of an arbitrary number of languages. This necessitates the development of a system flexible enough to support all of this. This can be done through the means of modularizing the configuration files responsible for providing translations, reading the list of languages at runtime, splitting every language localization into its own separate configuration file, and using string mapping.

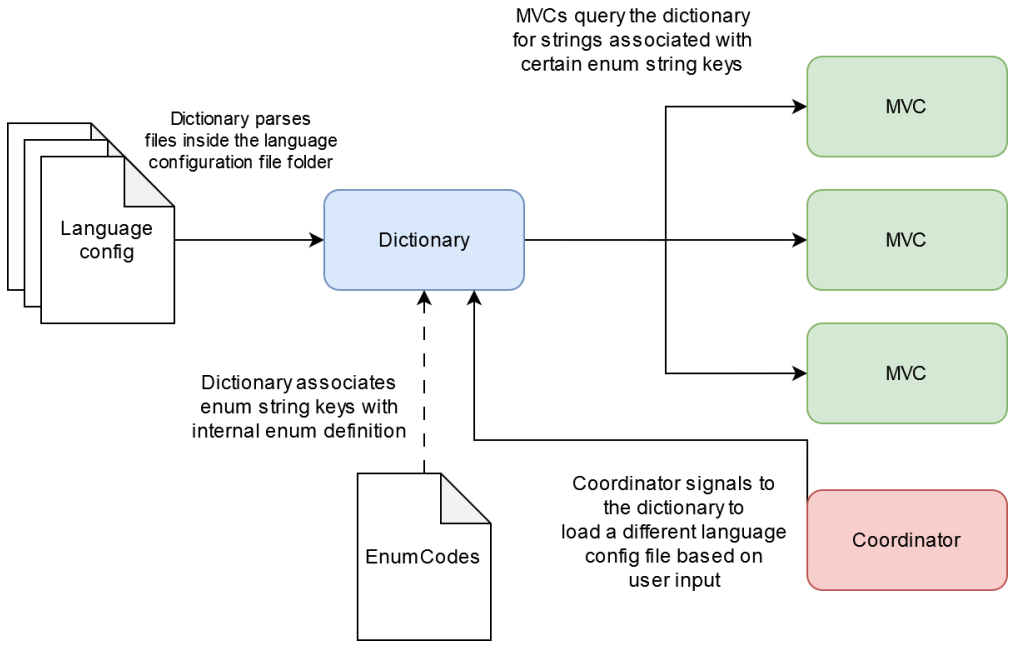

Figure 3.4 The dictionary and how it interacts with the rest of the application.

3.3.2 String mapping

String mapping is the main thing enabling mutli-language support. It involves having a pre-determined list of key codes, mapping them to strings that those key codes represent, and using the key codes in the UI code to reference the strings themselves. For the most part, the DMI display application doesn't have any arbitrary user input, and all the messages that it can receive from EVC are known beforehand. This means that we can pre-define all of the possible text strings that will be used throughout the application in advance.

When the application is started, the application will scan the `EnumCodes.json` file to associate human-readable string keys with their numeric values defined inside the application. It will then scan the language configuration folder for any language translation files it can use. It will parse each file in that folder and determine whether or not it's a valid localization file. It will then store a list of all the valid localization files to refer to later on and display for the user at the language selection screen. The fallback language will be determined by the application context config file. English will be the default fallback language, it will also be possible to change the default fallback language inside the application context configuration file.

To perform string mapping, the application will go through the language config file, scan for individual key codes and extract the corresponding text value from the file. It will then store the key code-text value pairs inside a so-called language map, which the rest of the UI will then reference for determining what string of text to display where. Upon launch, the application will also perform this matching and storing process one extra time to form a fallback language map, which the UI will access in case it can't find the necessary string in the currently active language map.

3.3.3 Language file format

The language file format will have an easily editable flat structure. To keep consistency with the rest of the configuration structure, and to not introduce extra complexity or libraries to the

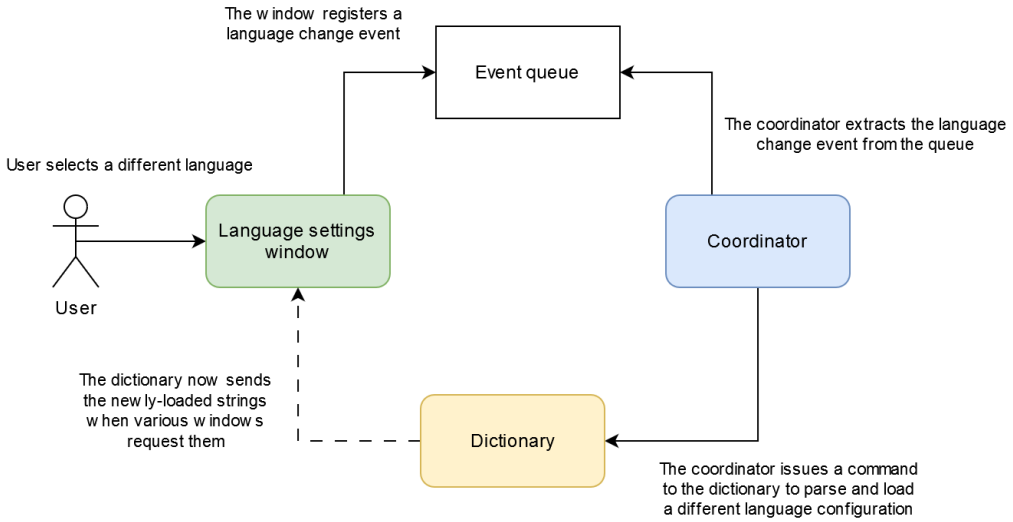

Figure 3.5 The process of changing the display language.

application logic, the language file will be stored as a JSON file and accessed using the JSON for Modern C++ library [11]. The first value in the file will be the name of the language that will be displayed on the button corresponding to that language, after which other values will follow. For ergonomic purposes, the key codes will not be numbers, as they were in the previous version of the application, but strings, that will be parsed to determine which key code is being referenced. The key codes will not be case-sensitive.

Any key code listed in the config that doesn't correspond to any UI string in the application will be ignored by the parser. The language file doesn't have to have every key code defined. In this case, the application will refer to the fallback language map instead when getting the UI string associated with that particular key code.

3.3.4 Changing the display language

The user will be able to change the display language of the application at any time by going into the language settings window and selecting their preferred language. Once a user chooses a language that's different from the one that's currently active, the application will clear out the active language map and parse the localization file corresponding to the button the user clicked on. Parsing will be performed just as described above, the result of which will be a newly formed language map that the application will query for the display of UI strings from that point on.

3.4 Changing display resolution

3.4.1 Overview

In the previous version of the DMI display application, resolution settings were baked into the application's source code. This meant that resolution was not configurable and the entire program had to be recompiled in order to change it. This is sub-optimal for several reasons. The user needs access to the source code and development tools in order to adjust the resolution of the application, which requires a certain degree of technical know-how from the user. If the user doesn't have access or the technical know-how to use these tools, they would have to contact the developer and request a custom binary set to the desired resolution. Finally, changing the resolution with the current setup requires recompiling the entire application, which can take

several minutes and is not ideal for debugging. For these reasons, the resolution should be truly configurable instead of a compile-time constant.

3.4.2 Resolution configuration

The resolution configuration will be implemented as an extension of the runtime settings configuration described in section 3.1. Resolution width and height are perfect candidates for storage in the application context singleton due to them being accessed throughout the application and not modified after initializing the application.

3.4.3 Model-View-Controller modifications

The MVC display logic needs to be changed to be more modular and responsive to different screen resolutions. Since the pixel width and height of the display are no longer compile time constants, we can no longer use the `SCALE` preprocessor macro to rescale user interface elements as needed to adjust for the chosen resolution. For this, the resolution parameters were split between two values: the fixed 640x480 logical resolution as defined by the subset [6] and the adjustable actual resolution of the window the application is rendering to. All references of the `SCALE` macro will be removed from the code, and the display components will refer to the logical sizes of various objects as opposed to their actual sizes in pixels in their display logic. The MVC layer will be refactored to be completely independent from the actual screen resolution, which will make using arbitrary resolutions and introducing changes to the application significantly easier. The developer will no longer need to worry about forgetting to rescale some element of the display based on the compile time resolution constant.

3.4.4 Graphics library modifications

To accommodate the changes described above, the graphics library functions will be changed to take care of the scaling based on the actual screen resolution. For this, as a replacement of the `SCALE` macro, the application context singleton will be equipped with additional functions for calculating the screen scale in relation to the logical resolution of 640x480, assuming the maintenance of a 4:3 aspect ratio.

Our goal presents us with an interesting problem. The way the graphics library works is that the display components call individual functions from the library in order to draw various elements on the screen - things like rectangles, lines, text, and so on. For this, scaling is very straight-forward – all that needs to be done is to multiply the elements by the scale parameter. But what if those functions are called by other functions inside the library? How do we avoid bugs caused by scaling the objects multiple times over?

The solution I came up with was to use a feature of the C++ language called function overloading. Many of the library functions were split into two separate functions with the same name: the “raw” function and the “scaled” function. The “raw” function kept the signature of the original function and called the “scaled” variant of that function after performing the necessary scaling. The “scaled” function then performed the necessary render logic and draw calls. The key detail here is that the “scaled” versions of the functions use a newly introduced object as their parameter called `ScaledRect`. A `ScaledRect` object is only scaled once upon initialization, which is how we make sure that the rescaling process can only happen at most once between any particular function getting called from the application logic layer and the resulting draw calls to SDL2.

In my opinion, this solution is very elegant because it solves several problems at once. First of all, it makes the developer’s life easier, as they won’t need to worry about calling certain functions in a certain order or adding confusing conditionals to make something scale properly.

Secondly, it's trivial to work with, as the developer working on a display area doesn't need to think about which variant of a function to call, thanks to function overloading. And finally, it's also easy to maintain, as the overloaded variants of drawing functions simply call the variant containing the drawing implementation instead of having to duplicate the drawing code.

3.5 Documentation

In software, documentation is written text or illustration that accompanies computer software or is embedded in the source code. The purpose of documentation in code is generally to give the reader more context for what a certain piece of code does and how exactly it does it. Documentation can come in different forms – for example, it can come in the form of comments inside source code, or in the form of separate documents describing how a certain component works. Different projects have different approaches when it comes to documentation. One extreme would be to have little to no documentation at all and solely rely on "self-documenting code", meaning code that is written in such a clear way that it doesn't require any extra explanations. The other end of the extreme would be adding comments to every single line of code written, such as explaining that a function called `ChangeResolution()` does, in fact, change the resolution.

As far as this project is concerned, there was already some documentation in place. For example, there are knowledge base repositories, where various message and error codes are described. However, the documentation is definitely not extensive enough. Some parts of the application have no explanation at all, so a new developer has to spend a considerable amount of time figuring out how such parts work and why they do the things that they do. There are also some parts which are documented in a misleading way. Since the documentation was originally written, several changes to the functionality of that piece of code were made without updating the comments.

The approach I took for solving this problem was to delete or update outdated documentation and try to explain newly-introduced code as best as I could. I believe that generally speaking, code should be written in a "self-documenting" manner, meaning that how a particular piece of code works or what it does should be obvious without too much effort from the side of the developer while analyzing it. This is usually achieved by means of using sensible naming conventions. However, even if it's clear to the programmer what a particular piece of code does, what's not always clear is the reasoning for why it does the thing that it does, why it is in place or what it's trying to achieve. I tried to solve that issue by adding extra clarifications via comments in the parts that I felt could be obscure to other people, and even to me if I were to step back from the project and return from it a few months or years afterwards.

3.6 Logging

3.6.1 Overview

Almost every program introduces logging at some stage of its development life cycle. Logging is an excellent way to see what the program is doing at any point without having to attach a debugger and set up breakpoints. It's also great for enabling communication between the user and the developer, as in the event that the user runs into a crash or an issue, the user can send the developer their log data to help the developer identify, reproduce and address the problem.

Different programming languages and frameworks have different ways of handling logging. Some languages have logging functionality built into their standard library, unfortunately however, C++ is not one of those languages. This results in C++ developers either developing their own logging library or using one of the available libraries that provide logging functionality.

3.6.2 Third-party logging libraries

The advantage of using third-party libraries is that it can save the developers a lot of work. Such libraries typically come with a host of features, such as coloured console output, multi-threading support, support for multiple platforms, custom formatting and the like. As C++ is quite a mature language with a large community, and logging is relatively simple in its essence, there are many solid third-party options available.

3.6.3 Custom logging library

Another option is to use a custom library for logging. The project can opt for this choice if it has some specific set of requirements that isn't sufficiently met by the options available online. Some developers also opt for this method because they aren't particularly constrained for time and find the process of manually implementing their own library to suit their needs fulfilling and enjoyable. Sometimes security can be a concern, too — one recent example of this is the Log4j vulnerability. [26]

On the other hand, being the developer of your own library will not necessarily prevent you from accidentally creating security vulnerabilities. Especially in relatively low-level languages such as C or C++, these things can be very error-prone and risky. In many cases, a privately-developed library will be a lot more likely to have security issues compared to a widely-used third-party one. For this reason as well as time constraints, we will be using a third-party open-source logging library for the DMI display application.

3.6.4 Choosing a library

A library has to fulfill a certain set of criteria if it is to be a good fit for the project. It has to be well-documented, easy to use and easy to integrate into existing code. It needs to be fast, so that using it won't hurt the application's performance. It also has to be cross-platform, as the DMI display application needs to support both Windows and Linux. The library should also work well with multi-threaded code, because even though the project currently doesn't utilize threads very much, it will definitely become more relevant as time goes on.

Having looked at the options available, I have decided to utilize the spdlog library. [27] It's popular, which means it's likely to be well-supported and secure. It's very easy to use and get started with and has a wide range of supported features that we can utilize later on to suit the project's needs.

3.6.5 Logging implementation

I decided that for a project of such scope and complexity as this one, a gradual approach to introducing changes of this kind would be best. The spdlog library will be added to the list of the libraries used by the project, and some very basic logging will be handled using it, such as printing out information about the initialization of individual components. The library's design and different available levels of logging should then make it easy for developers to introduce logging into other parts of the application as more code is being added and modified.

3.7 Extending functionality

3.7.1 Overview

This part will cover some of the work that was done by me to make sure the DMI display application adheres to the specification at the time of the project's inception. A large and very

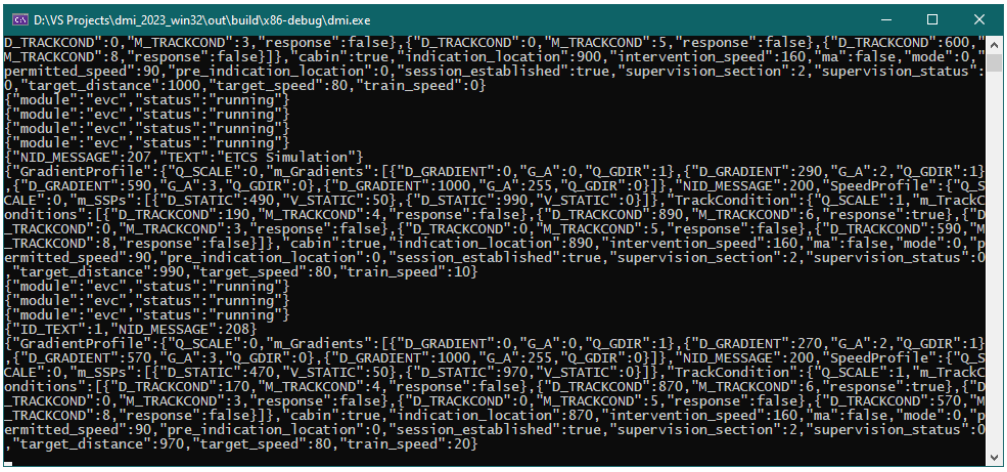

Figure 3.6 An example of logging various events using the developer console.

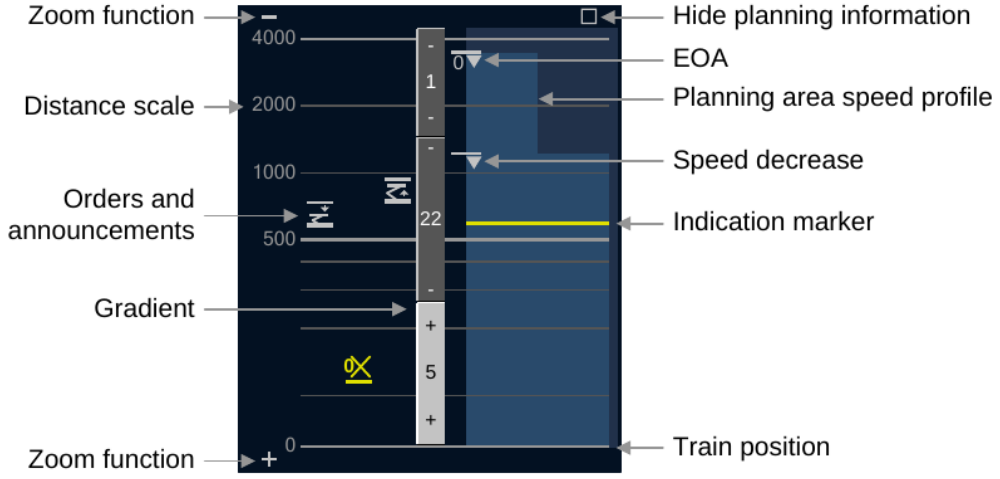

Figure 3.7 The main elements of the planning information. Sourced from [6]

complex part of that was the displaying of planning information shown on figure 3.7.

3.7.2 Distance scale

The distance scale has several zoom levels, with the highest representing the range between 0 and 1000 meters and the lowest representing the range between 0 and 3200 meters. The distance scale contains seven distance scale lines, which stay at the same level graphically no matter the zoom level selected. The scale is not linear, meaning that the scale is smaller the closer a particular object is to the train, and gets progressively larger the further away the object is, with the train position being represented by the first distance line. For example, on figure 3.7 the areas between the first and second lines, the second and third lines and the third and fourth lines from the bottom represent distances of 100m each, even though the spacing between each pair is visibly different. This is also apparent when looking at the displayed distance numbers: the space from 0m to 500m is visually equal to the space from 500m to 4000m.

Each element mentioned further in this section is placed in the area while taking this logarithmic scaling into account. The driver can also use the zoom function buttons to change

the zoom level of the area at any point, and the position of each element will be recalculated accordingly. The driver can also choose to hide the planning information by pressing the button labeled as such on figure 3.7.

3.7.3 Orders, announcements, speed discontinuities

The application is capable of displaying various orders and announcements sent to it by the EVC. These orders and announcements can have different icons depending on the individual order/announcement, as defined by the subset [6]. On the side opposite to the one displaying orders and announcements, speed discontinuity symbols are displayed. These can be speed increases, speed decreases and a special type of a speed decrease for a decrease to the target at zero speed.

3.7.4 Gradient profile

The DMI display application is capable of displaying various gradients sent to the application by the EVC component. These gradients can be of two types: uphill and downhill. The uphill gradient is represented by the "+" sign on either end of the gradient and is drawn in light grey, while the downhill gradient is instead drawn in dark grey and is represented using the "-" sign. The display application will also show the number value of the gradient in the middle of it in case the size of the gradient as well as the zoom level chosen by the driver makes the gradient object large enough for it. The gradient display area doesn't have to be continuous, meaning the gradients can have gaps between them.

3.7.5 Planning Area speed profile (PASP)

The DMI display application displays the planning area speed profile as a speed-distance diagram within the distance range selected by the driver for the planning information and within the movement authority. This diagram is represented by a graph consisting of two shades of blue on the right side of the planning area. The graph is split into four sections at fixed ratios of 1/4, 1/2 and 3/4 of PASP's size relative to the ceiling permitted speed at the current train location. The display of this graph is based on the information sent to the DMI by the EVC. The indication marker is then drawn on top of the PASP graph.

3.8 Coding standards and guidelines

3.8.1 Motivation

The majority of software projects adhere to some set of coding standards and guidelines. This helps the code look more uniform, and as a result easier to parse and to read. Furthermore, as this project is being worked on by many inexperienced students, a set of guiding principles will be very useful in helping to elevate the overall quality of the code.

At the same time, the standards should be easy to follow for any contributor to the project. I did not want to introduce heavy auto-formatting partly for this purpose. I believe that sometimes it makes sense to break from the convention in order to express yourself more clearly, and I also believe that letting people decide when to do it for themselves can be a valuable learning experience. I did not want to enforce my own personal style onto everyone else, and strove to find a happy medium that nobody would have too much difficulty adjusting to.

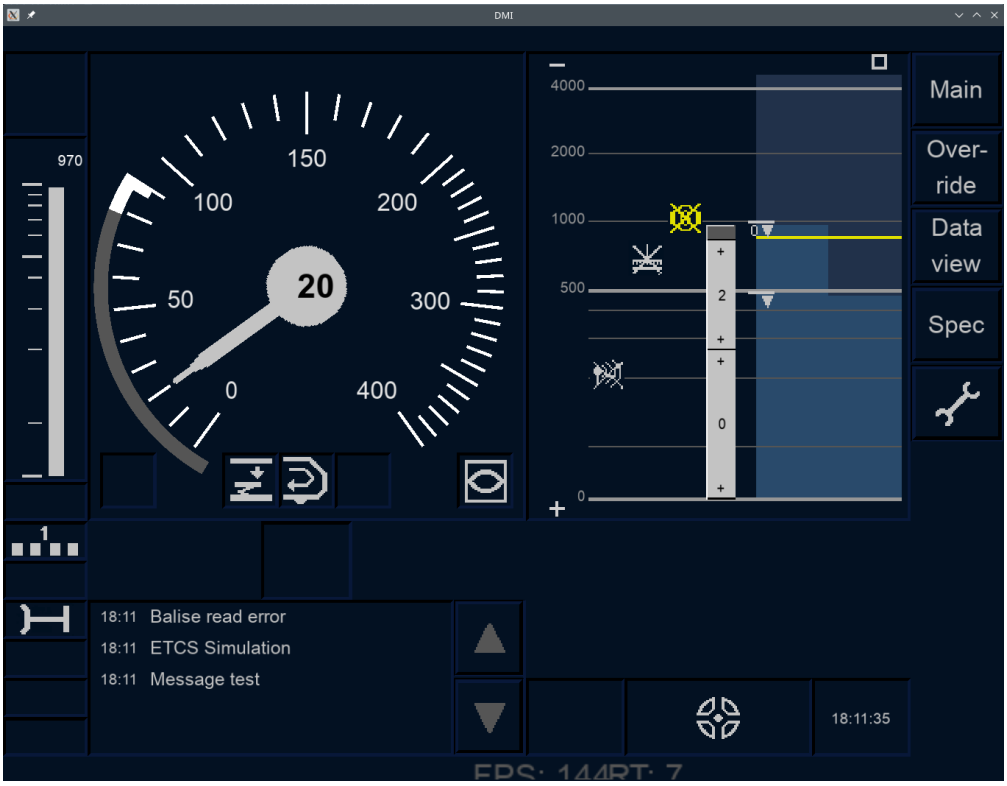

Figure 3.8 A screenshot of the DMI display application with the planning area shown.

3.8.2 Coding style guidelines

Clearly defined and easily accessible code style guidelines are necessary for unifying the coding style of the project. As such, there was a need for a document outlining all these guidelines in a way that is easy for developers to understand and follow. Having researched several documents of this kind from other projects, I decided to write my own in a style loosely based on the Linux kernel coding style guidelines. [28] The code style document is named `CODING_CONVENTIONS.md` and has a multitude of examples of "good" and "bad" coding style. The style document is open for anyone to modify as project requirements change over time and gaps in knowledge are filled.

3.8.3 General code quality advice

As part of describing the coding style for the DMI display application project, I included a list of good practices to follow when writing code. This includes things like preferring immutable definitions, passing objects by reference instead of by value, a guide on how to write useful comments and how to name variables and functions.

3.8.4 Clang-Format

Clang-Format is a widely-used C++ code formatter. It provides options to define code style options in formatted files that can be included as part of the project where code style rules are kept. [29] The purpose of using an auto-formatter is to keep code style standards consistent across the project. On a project with more consistent code style rules, it's relatively easy for developers to simply follow the style of whatever source code file they are currently working

on, and carry that style over to newly-created files. However, since the project never had any consistent set of formatting guidelines, following the style of whatever is already in place is pretty much impossible. At the same time, going over every source file manually and fixing problems one by one is a tedious, time-consuming and error-prone process. Because of this, it made sense to me to combine the defined style guidelines into a `.clang-format` file and let the developers simply hit the "Reformat" hotkey in their development environment of choice upon coming across a file that doesn't meet the project's style guide. It also helps give a quick reference to how certain parts of the code should look like without having to consult the style guidelines document every time.

Chapter 4

Implementation

This chapter contains manuals with information for installing, developing and using the DMI display application. An instruction guide for how to add more languages to the application is also included, with a showcase for how newly-added languages could look. The installation manuals will target the Windows operating system. Manuals targeting Linux systems will be included as part of Ondřej Měšťan's thesis. [19]

4.1 Simulator layout

The layout of the entire ETCS simulator project is shown on figure 2.5. A description of each individual component is covered in detail in one of the sections inside the implementation chapter of Jan Stejskal's thesis. [12] The work done as part of this particular thesis was done solely on the DMI component of the simulator, and the overall layout of the project's components has not significantly changed.

4.2 Splitting responsibilities

The current iteration of the project is a result of my and Ondřej Měšťan's work, while the project overall is the result of the work of us and our predecessors from the BI-SP1 and BI-SP2 subject teams. As a result, there was a need to coordinate our changes so as to not overlap our domains of responsibility and work more efficiently. Our solution was to have me work on the general structure and architecture of the application, while Měšťan focused on user interface changes and implementing support for different subsets. Since we were both working on the same application, there needed to be a way to tell who worked on what. One easy way to do that is by using `Git blame` — a function of Git, the version control system used by the project, that shows authors of each individual line of code.

According to the tool, I have contributed 107,895 line of code changes to the project, of which 53,811 were additions and 54,084 were removals. It is worth noting that Git records line changes as both additions and removals, which is why I have more total removals than total additions, as a lot of my time was spent on refactoring already existing code. Even still, the general principle in software development is that removing code while maintaining or expanding the program's feature set is a good thing, as it reduces the total code complexity and technical debt.

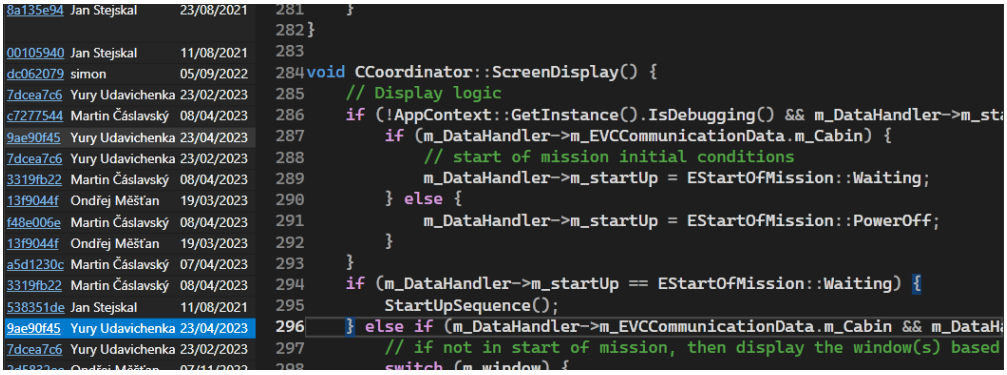

Figure 4.1 Git blame annotating individual lines of code inside Visual Studio.

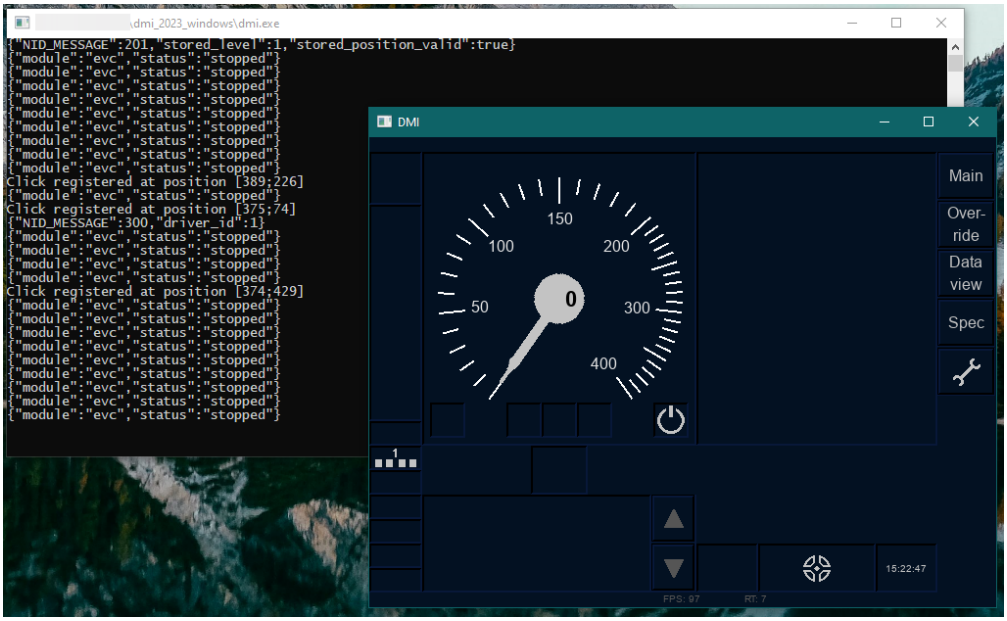

Figure 4.2 An example of the DMI display application running on a Windows machine.

4.3 Installation manual

To install the DMI display application on a Windows PC, one would only need to download the latest version of Visual C++ Redistributable by Microsoft, which can be found on their official website. [30] After that, the user needs to navigate over to the releases page in the repository and download the Windows version of the application. To start the application, the user needs to extract the archive into a convenient folder and run the `dmi.exe` file from within the folder. The installation manual for Linux systems will be contained inside Měšťan's thesis. [19]

4.4 User manual

This section will cover some of the scenarios a user might run into. This tutorial isn't intended for drivers, as supplying a full manual is out of scope for the purposes of this work.

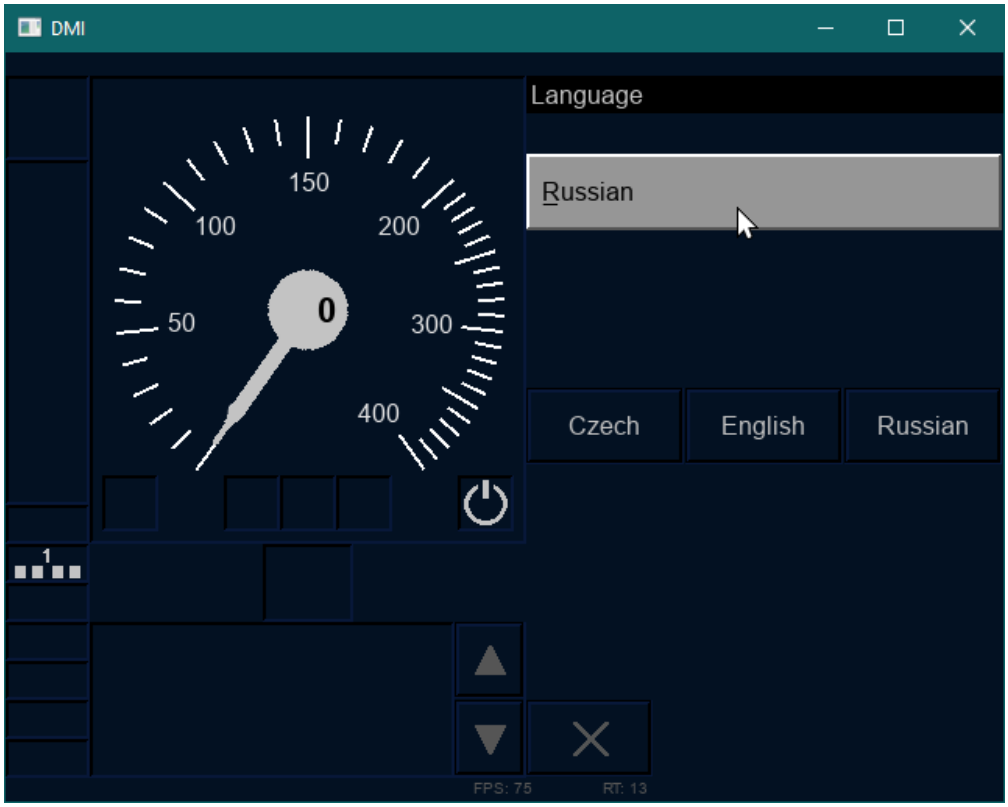

Figure 4.3 The language settings window.

4.4.1 Changing a language

Inside the application package, three languages are supplied by default: English, Czech and Russian. To switch between them, the user needs to open the application, fill all the data required by the application and navigate to the default window. From there, the user needs to open the language settings menu by clicking on the button displaying a wrench icon and then another button displaying a flag. From there, the user will be presented with several language options to choose from. To change the language, the user must click on the button with the name of their language of choice and then confirm their choice by clicking on the grey input field up above.

By default, the language of the application is set to English. To change it to something else, the user needs to navigate to the `app.json` configuration file inside the `config/global` directory and find the `default_language` variable. By default it is set to "English", the user can change it to "Czech" or "Russian" to have the application be set to the Czech or the Russian language on launch respectively.

4.4.2 Installing translations

To install new translations for the application, the user must put the supplied language configuration file into the `config/languages` directory. The next time the user starts the application, a new language option should show up in the language settings menu.

Figure 4.4 The train data window with the Czech localization selected.

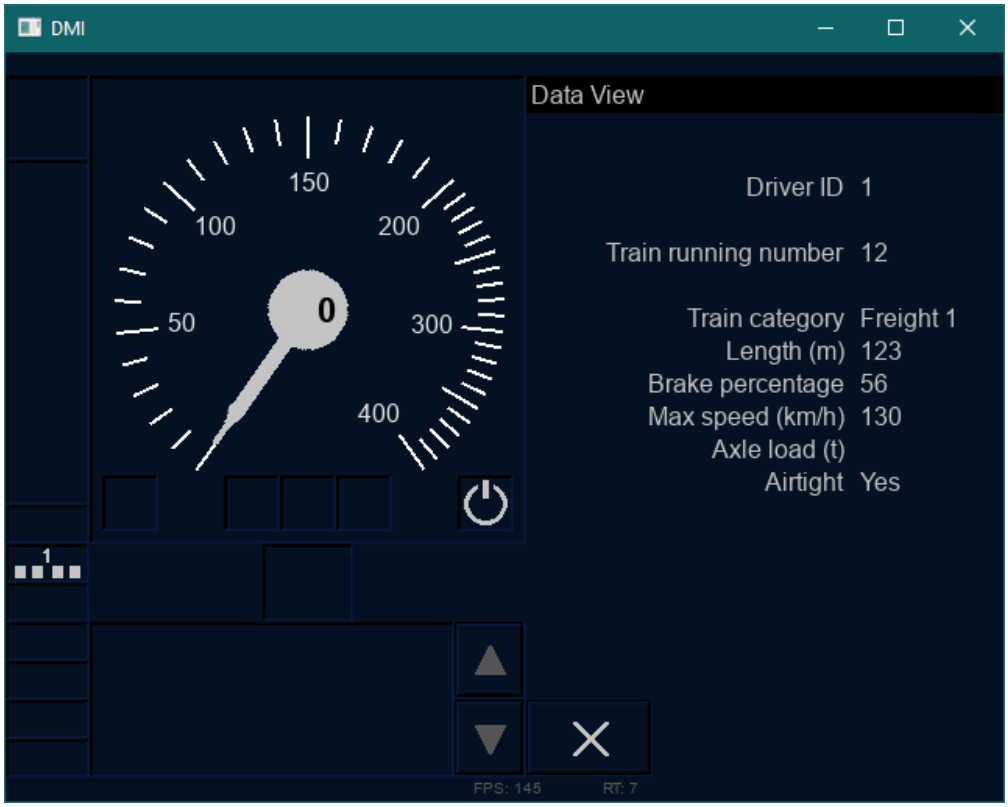

Figure 4.5 The train data window with the English localization selected.

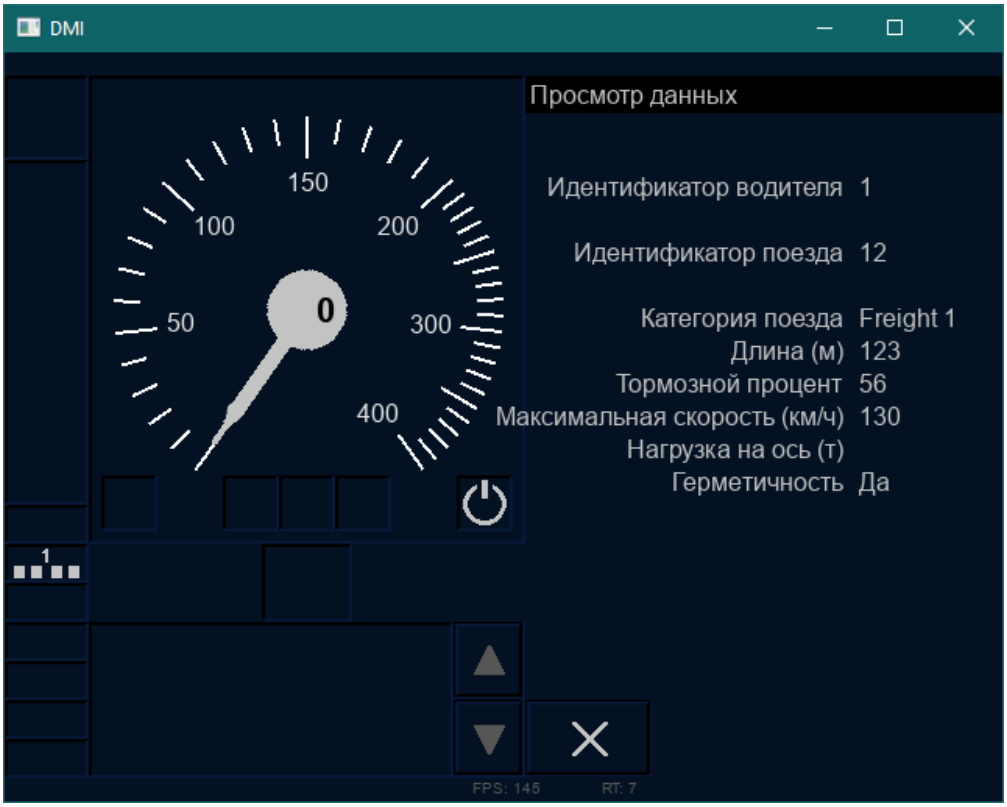

Figure 4.6 The train data window with the Russian localization selected.

4.4.3 Setting up MQTT communication

To change the address of the MQTT client the application is supposed to communicate with, the user needs to navigate into the `config/global` directory and modify the `app.json` file. Inside the file, they should look for two variables: `mqtt_host_address` and `mqtt_port`. Both variables are set to certain values by default. The user simply needs to change the values with the ones supplied to them by the people operating the component the DMI display application is supposed to be communicating with, save the file and relaunch the application. Once done, the application will now try to connect to the MQTT client using the newly supplied address.

4.4.4 Changing the resolution

To change the app resolution, the user can simply drag the corner of the window that opens once they launch the DMI display application, and the text should get scaled accordingly. However, the user must be careful when doing that, as the application is written with the 4:3 resolution in mind, and the button click mapping doesn't support other aspect ratios at the present. Instead, the preferred way of changing the resolution is through the aforementioned `app.json` configuration file by modifying the `actual_width` and `actual_height` variables to values equal to or higher than 640x480, while keeping with the 4:3 aspect ratio. To switch the program into fullscreen mode, `app.json` has a switch for that too.

4.5 Developer manual

One hurdle that I ran into during the project's implementation phase was that, while technically the application did support both Windows and Linux, the project setup instructions for Windows were very outdated and complicated. The Linux version of the project used CMake as the project's build tool, while the Windows version was relying on Visual Studio's default MSBuild system. The massive difference in these setups, as well as the fact that MSBuild is only supported by the Windows version of Microsoft Visual Studio, has historically caused developers many issues as they would attempt to get the code base up and running on their Windows setups. To fix this, I employed Visual Studio's native CMake support functionality in order to massively streamline the process. At the time of writing this, the setup process for Visual Studio is arguably the one with the fewest steps and the least amount of friction, whereas before it was the complete opposite.

4.5.1 Setting up the environment

This section will describe the development environment setup process for Windows 10 and 11 using Visual Studio 2022. While this specific toolchain is not strictly necessary and other options are available, Visual Studio has the most straightforward setup and is the easiest to get started with. The process of setting up a development environment on Linux will be covered in Ondřej Měšťan's thesis. [19]

First, download and install the latest version of Microsoft Visual Studio from Microsoft's website with the appropriate license. [31] When prompted for selecting various available workloads, make sure "Desktop development with C++" is selected, as shown on figure 4.7. Once installation concludes, launch Visual Studio. On the welcome screen, select the "Clone a repository" option. For the repository location, fill in its GitLab address. For the path, select any folder you find convenient. After that, press "Clone" and wait for the project to load. If prompted for authentication, fill in your FIT GitLab email and password.

Once the IDE window shows up, navigate to the Solution Explorer sidebar. If it's not immediately visible, you can open it by clicking on the "View" – "Solution Explorer" option at

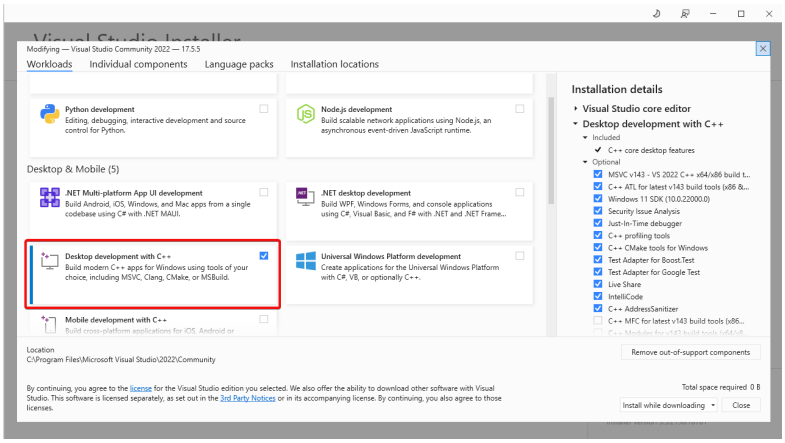

Figure 4.7 The workloads necessary to build and run the project.

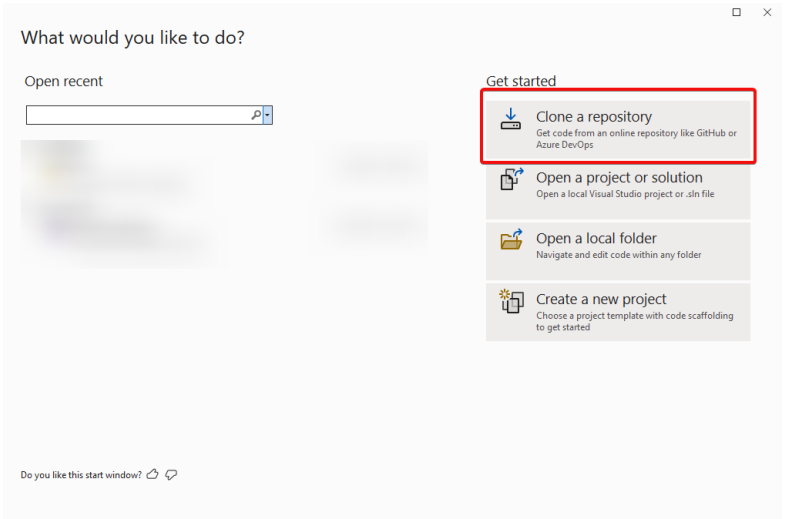

Figure 4.8 The welcome window of Visual Studio.

the top bar. Once there, double-click on the "Folder view" option and the project should get auto-configured by CMake. To build and run the project, you need to select `dmi.exe` as your startup item at the top panel and click "Run".

4.5.2 Documentation

The developer documentation in the repository features several important files. `README.md` serves as the central hub for everything else, describing how to set up and build the project inside various development environments on the supported operating systems. It also has links to the other documentation files that provide information for the developer on how to get started.

`PROGRAMMER_MANUAL.md` is where most of the documentation is located. It gives an overview of the project's structure and describes the way various components and layers of the application work. For example, there the developer can find information about the language dictionary system or the graphics library layer, as well as guidelines for how to use them.

Guides on submitting new changes to the repository are contained within two Markdown files named `CONTRIBUTING.md` and `CODING_CONVENTIONS.md`. The former describes the general

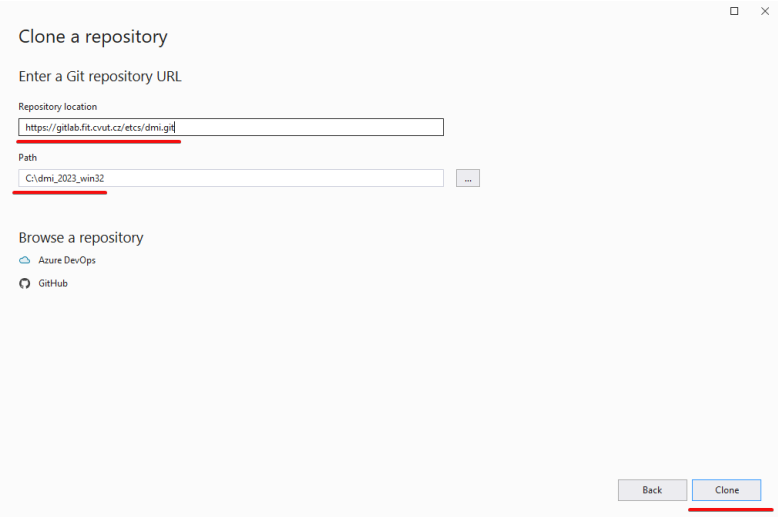

Figure 4.9 The window for specifying the repository path and location.

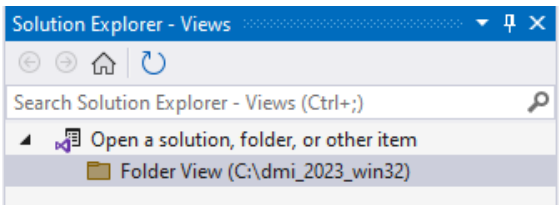

Figure 4.10 The solution explorer window.

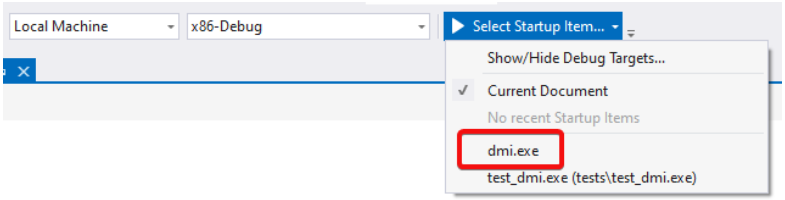

Figure 4.11 The startup item selection.

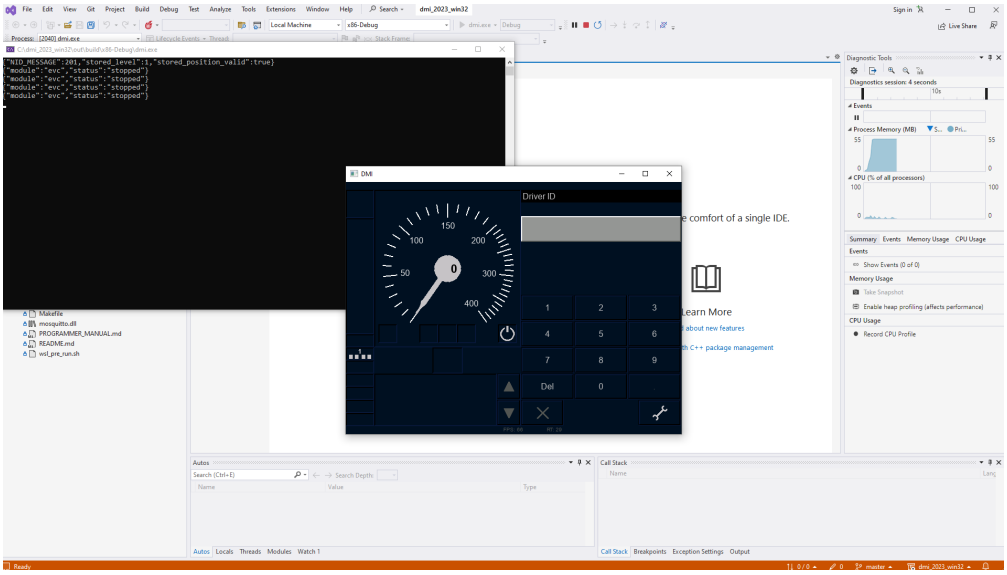

Figure 4.12 The DMI display application running inside Visual Studio.

process a developer should follow when contributing to the project, such as working within a separate branch and submitting merge requests instead of committing changes straight to master. The latter gives an overview of the coding style and conventions employed by the project, as well as some practical C++ tips that an inexperienced developer might not necessarily be aware of.

4.6 Translator manual

The current version of the DMI display application supports adding translations for more languages without having to modify the source code. The translator would instead need to add a new language configuration file to the `config/languages` directory. An easy way to do that is to refer to the `TEMPLATE.json` file, which has every possible language string token listed inside it.

The language file name can be completely arbitrary. The only necessary requirements are that it needs to follow the JSON file format and it needs the `language_name` parameter filled up. Everything else is optional, as the application will fall back on the default language if it fails to find a string token inside the currently active language file, as is demonstrated by the Russian localization file not having EVC message strings defined. However, the translator needs to not keep empty string tokens inside the language configuration file if they don't want an empty string to show up in the UI.

Chapter 5
Testing

This chapter describes the testing scenarios and processes employed during user, acceptance and integration testing. It also describes the importance of automated testing and the state of automated testing within the current iteration of the application.

5.1 User testing - application walkthrough

The purpose of this testing scenario is to have the user go through the DMI display application's functionality, verifying that everything works as intended. The starting point of the testing should be having the folder where the application is located opened and the application itself ready for use. Together with the individual steps the scenario also describes the expected outcomes of these steps while also giving some context behind the actions, so that whoever is doing the testing can be more informed and do more thorough checks of application functionality.

5.1.1 Launching the application

Upon launch, the user should be prompted with two windows: the DMI window itself as well as the debug console window. Inside the debug console window, the user should see logs, including logs of a periodic heartbeat being sent out. The heartbeat's module should be equal to "evc" and its status should be equal to "stopped". In the DMI window, the user should see a speedometer displayed on the left side of the window and a numerical keyboard together with an input field on the right side of the window.

5.1.2 Changing display resolution

The user should try dragging the corner of the DMI display application window and making the window larger while maintaining the original 4:3 aspect ratio. The application is expected to scale the size of the text and UI elements up to fit the new size of the window. The buttons on the screen should still register clicks in the areas they are drawn in. Please note that the click areas of the buttons won't exactly match the buttons themselves should the chosen aspect ratio be anything other than 4:3, which is a known limitation and a deliberate design choice. The user should then drag the window corner inward to make the window smaller and verify that it cannot go below the original resolution of 640x480.

The user should now close the application and navigate to the `app.json` config file located inside the `config/global` folder. In it, the user should look for two parameters: `actual_width` and `actual_height`. The value of the former parameter should be set to 640 by default, while

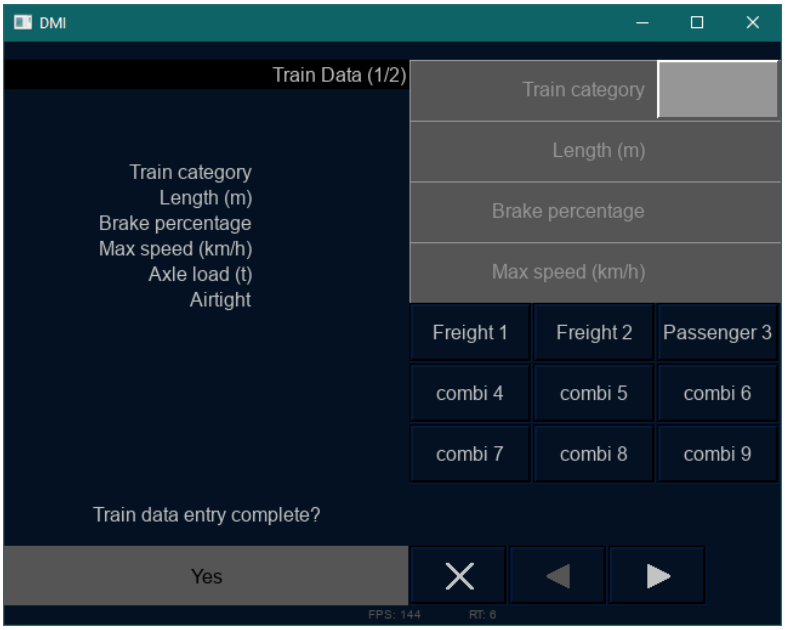

Figure 5.1 The train data input window.

the default value of the latter parameter should be at 480. The user should change these values to a larger 4:3 resolution by setting `actual_width` to 1024 and `actual_height` to 768. After doing that, the user should save their changes to the configuration file, close it, navigate back to the application executable and launch it. Upon launch, the user should be presented with a bigger window, but all the buttons, fields and other screen elements should be scaled proportionately to the newly chosen resolution.

5.1.3 Filling in train and driver data

The user should attempt to fill in the driver ID as prompted. Upon clicking the numerical keyboard buttons, the associated symbols should start showing up in the grey input area, and the debug console window should show a message in the form of "Click registered at position [X; Y]". Pressing "Del" should result in the rightmost symbol being removed from the input area. In the case of no symbol being present, the "Del" button shouldn't do anything. Once the user is satisfied with the ID, they should click the input field to submit and proceed.

The user should navigate to the train data window by clicking on the button with the same name. After that, they are expected to be prompted with an empty train data input window, as showcased on figure 5.1. The user should now attempt to fill in the data as suggested by the UI by choosing the desired option or filling in the desired value using the on-screen numeric keyboard, and as they proceed in doing that, the submitted data should start showing up inside the input fields as well as the parameter list on the left. The user should also be able to fill the data fields in an arbitrary order, not necessarily top to bottom. Upon clicking on the left and right arrows at the bottom of the screen, the user should be able to navigate between the two sets of input fields.

Once the user is done with putting in the desired values, they should be prompted with whether or not the data entry is complete at the bottom left of the screen. Upon clicking "Yes", they should be presented with an option to validate the train data. The user should agree and then fill in the train running number once prompted by the application. After doing all of these steps, the user should be returned to the application's main window.

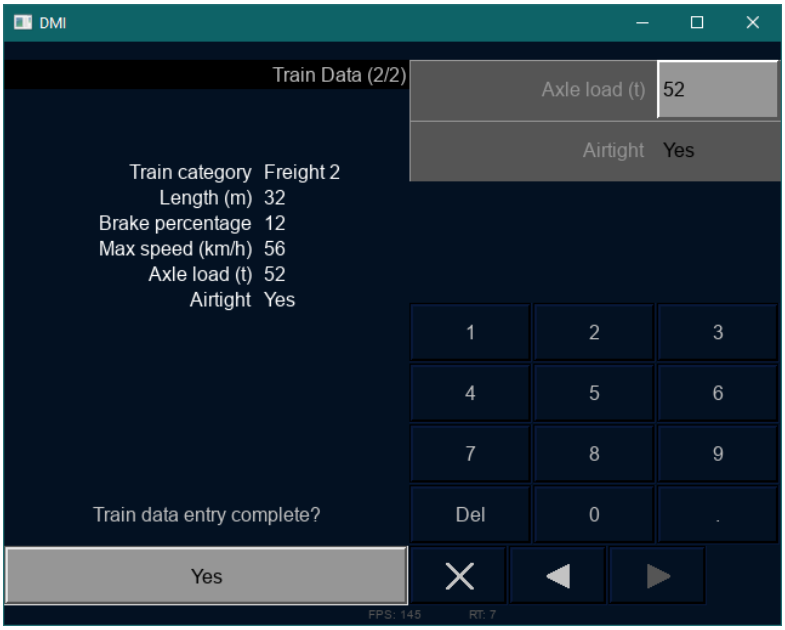

Figure 5.2 The second part of the train data input window with all the required information filled.

The user should now verify whether or not the data they had input got saved. To do that, they need to press the button with the "X" symbol to navigate to the default window and press the "Data view" button on the sidebar. The data showing up in the data view window that will open as the result of this sequence of actions is expected to be identical to the data submitted by the user in the previous dialogue sequence.

5.1.4 Starting the journey

To return to the main window, the user must once again press the button with the "X" symbol and click on the sidebar button labeled "Main". To verify whether or not heartbeat status will change upon starting the journey, the user should click on the "Start" button. After this, the heartbeat status parameter in the console window logs should change from "stopped" to "running" and the application itself should now automatically navigate the user to the default window.

Shortly after starting the journey, the user will be presented with the planning area display, the speedometer will start changing its values and the message box below the speedometer will start displaying messages. These changes are based on a set of mock data supplied together with the application. Clicking on the "+" and "-" buttons to the left hand side of the planning area should change its zoom level and update the size and position of each element inside it accordingly. Clicking on the square button in the top right corner of the planning area should toggle the planning area display on and off.

Once enough messages show up in the message box below the speedometer, the down arrow next to it should light up and become clickable. Upon clicking on the arrows, the user should be able to scroll up and down the list of received messages. The received messages and updates should also show up in the debug console before they are presented to the screen inside the message box. For the purpose of this testing scenario, it is not important to dig into the details of these messages, as they are quite difficult for a human to understand in their unprocessed text form.

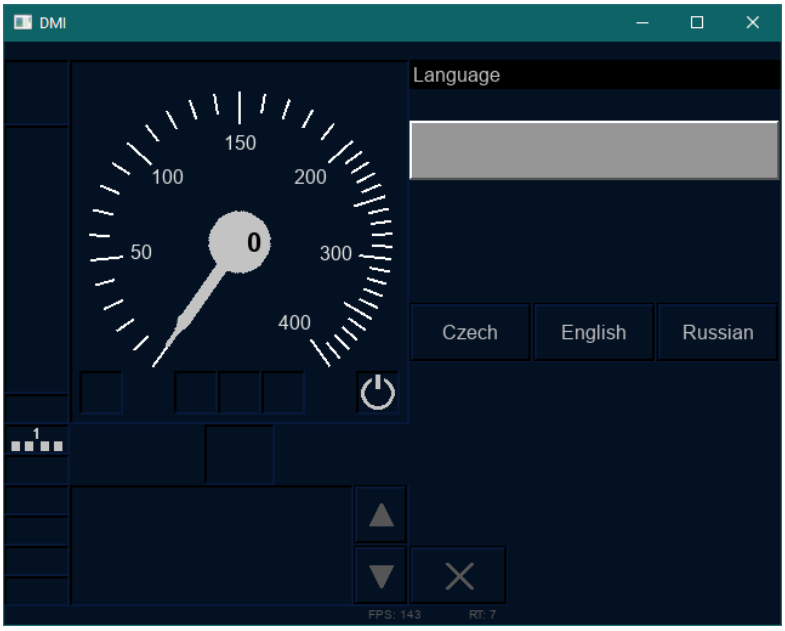

Figure 5.3 The language selection window.

5.1.5 Changing the display language

The user should attempt to change the display language and verify that this functionality works as intended. To do that, they should first open the settings window by clicking on the wrench icon on the default window's sidebar. From there, they should open the language window by clicking on the button with the flag icon. The user should now be presented with several language options: English, Czech and Russian.

The user should first select the Czech language option and confirm their choice by clicking on the input field above. Upon getting returned to the settings window, the user should then observe two changes: the window's title will go from "Settings" to "Nastavení" and the messages inside the message box below the speedometer should now be displayed in Czech. The user should now go back to the language selection window and choose Russian instead. Now, the window title is expected to be set to a word in the Cyrillic script while the text inside the message box should be set back to English. This is intended behaviour, as the provided Russian language file does not have messages defined in the translations, and as such the application will be using English as its fallback language in case of missing translations.

5.1.6 Adding and a new language

The user should now test the language addition functionality. To do that, they will need to close the application and navigate to the `config/languages` folder inside the application directory. The user should see four files in total, a reference template file as well as the three language files serving as options within the application. To test the language addition functionality, the user should duplicate `EN.json` and rename the new file to `AM.json`. Then, the user should open the newly created JSON file. To verify the fact that language file parsing works as intended, the user should change the `language_name` value from "English" to "American" and the `settings` value from "Settings" to "Options".

The user should then save the file and re-open the DMI display application. Once the application launches, the user should click on the wrench icon below the numerical keyboard. The

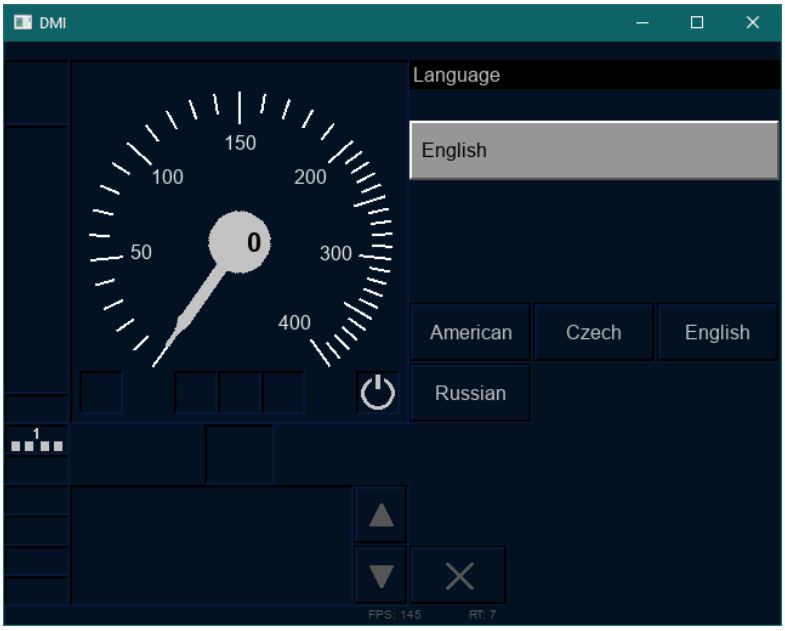

Figure 5.4 The list of languages after adding a new configuration file.

user should now see four language option buttons, with one of them being American. The user should select the newly added button and confirm their choice by clicking on the input field. The user should now see the settings window with the "Options" headline instead.

5.2 Acceptance testing

The goal of acceptance testing is to determine whether the product adheres to the given specifications and whether its requirements have been met. This type of testing is done by the customer, which in this case is doc. Ing. Martin Leso, Ph.D. from CTU's Faculty of Transportation. Acceptance testing will be performed under his supervision on the premises of the faculty.

At the time this thesis was submitted, acceptance testing has not yet been performed, however according to the agreement with the thesis's supervisor, acceptance testing will be conducted before the thesis's defense.

5.3 Integration testing

The goal of integration testing is to verify that the individual components of the project are able to communicate between one another and work together in a coherent and reliable way. For the purposes of this project, integration testing would involve testing the way the DMI display application interacts with the EVC and LPC components. Unfortunately, at the time of this thesis's submission, neither component is actively running or is in the state that would allow integration testing to be conducted.

5.4 Automated testing

Most team projects use some form of automated testing, usually with the help of a testing framework. Having automated testing in place helps prevent regressions, where a new addition

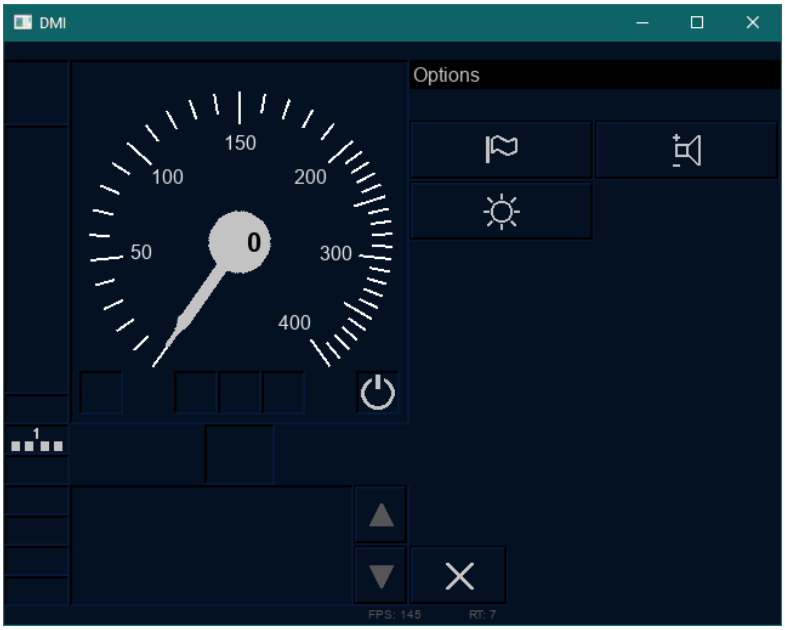

Figure 5.5 The new title of the settings window.

or change actually breaks certain already-existing functionality, and is often a faster, easier and more reliable way of testing compared to manual tests performed by users. However, the project itself must be structured in a way that allows for automated testing to be conducted, and some applications are easier to test than others. For example, a GUI application is generally considered far more difficult to automatically test compared to a web service with a web API layer, a business logic layer and a database layer.

During the creation of this thesis, CMake's CTest [32] framework was introduced to the DMI display application to faciliate automated testing. However, as of today, there are currently no automated tests actually written. I haven't been able to create any test cases, partly due to time constraints, and partly due to the fact that the current design of the application has too much coupling between objects. Because of this, isolating individual parts of the application and verifying their functionality using unit tests is pretty much impossible. The hope is that by continuing the process of modularizing the DMI display application and decoupling its various components from one another, eventually it will be in a state that is genuinely testable by automated tests at least on some level, but currently that is not the case.

Chapter 6
Conclusion

The goal of this thesis was the expansion and improvement of the DMI display application, bringing it closer to the chosen subset's specification as well as making general improvements to the code base so as to facilitate further work and improvement. This expansion was to be based on the analysis of previous works concerning the application and was to utilize various methods of software development. The application was supposed to be cross-platform, supporting both Windows and Linux operating systems. The resulting changes to the application were to be coordinated with Ondřej Měšťan and the results of his bachelor's thesis.

The results of the analysis chapter of my thesis was a bird's-eye view of the project's successes and failures and a set of functional and non-functional requirements that the project should be fulfilling. The design and implementation chapters of the thesis then used this set of requirements as foundation for introducing and implementing the proposed changes. Finally, the resulting application was then subjected to user testing in order to verify its reliability and ability to function as intended. In accordance to the agreement made with the thesis's supervisor, acceptance testing for the application will be conducted following this thesis's submission.

The resulting iteration of the DMI display application is a considerable improvement on the previous version of the application. There's significantly less reliance on hard-coded pre-defined constants in favour of dynamically processing and incorporating data from configuration files at the program's runtime. The introduced coding standards and conventions helped standardize the code base and made it easier for new developers to get familiar with the code base and produce better, higher quality contributions to it. The application supports both Windows and Linux operating systems, and the development setup process has been significantly streamlined so as to remove as much friction between the developer and their development environment of choice as possible.

The newly introduced dynamic language configuration file support allows for easy additions of new translations as well as straightforward updates of existing translations by re-organizing the config files to be more human-friendly. The DMI display application's resolution can now be easily changed at runtime as well as on startup through the use of configuration files, and working on new user interface additions has been made a lot simpler by the removal of the `SCALE` compile-time constant.

Communication with the LPC has been introduced to the project, as the DMI display application now continuously sends a heartbeat with its current status. The means of receiving and processing commands from the lecturer's PC has been introduced to the application's network layer, however as of right now, the rest of the application does not otherwise react to LPC's commands. A new logging library has been introduced and the application's flow of execution has been made more transparent by the improvements done to the logging system.

The application has been expanded in accordance to the ERA ERTMS subset version 2.3.

More specifically, the planning area has been fully implemented with all of its small details and various scaling algorithms to reposition its various elements. The underlying graphics layer has been significantly improved and made a lot easier to use as a high-level abstraction inside the Model-View-Controllers comprising the DMI display application. A significant amount of code has been decoupled during the process of the code base's standardization and refactoring by utilizing various software design patterns and development methods, allowing for easier extensions and testing later down the line.

6.1 Future work

Despite all of the aforementioned improvements, the project still needs a lot of work before it can function as a proper component of the ETCS training simulator. One significant point is testing – an application of such scope with so many developers needs to have a testing suite covering the code base in order to prevent the accidental introduction of bugs. There is still more refactoring that could be beneficial. One such example is the network layer. In my opinion, the class hierarchy of the MQTT client does not adhere to object-oriented programming principles. The Coordinator class also has a lot of issues for similar reasons, as pointed out in the analysis. Documentation, logging and error should be expanded. As the project expands its feature set, parallelizing its various components by means of multithreading could also be a good idea. Finally, the network communication part of the application could be improved by adding encryption and enabling the application to react to commands sent to it from the lecturer's PC.

Bibliography

1. *ERTMS* [online]. European Commission Directorate-General for Mobility and Transport, 2020. [visited on 2023-03-20]. Available from: https://transport.ec.europa.eu/transport-modes/rail/ertms_en.
2. *ERTMS - What is ERTMS?* [Online]. European Commission Directorate-General for Mobility and Transport, 2020. [visited on 2023-03-20]. Available from: https://transport.ec.europa.eu/transport-modes/rail/ertms/ertms-what-ertms_en.
3. *ETCS Levels and Modes* [online]. European Commission Directorate-General for Mobility and Transport, 2020. [visited on 2023-03-20]. Available from: https://transport.ec.europa.eu/transport-modes/rail/ertms/how-does-it-work/etcs-levels-and-modes_en.
4. *Sybsystems and Constituents of the ERTMS* [online]. European Commission Directorate-General for Mobility and Transport, 2020. [visited on 2023-03-20]. Available from: https://transport.ec.europa.eu/transport-modes/rail/ertms/how-does-it-work/subsystems-and-constituents-ertms_en.
5. LEPPÄNEN, Antti. *Balises on Orivesi-Jyväskylä railway in Muurame, Finland* [online]. 2009. [visited on 2023-05-10]. Available from: https://commons.wikimedia.org/wiki/File:Balises_in_Finland.jpg.
6. *ERA_ERTMS_015560 v2.3* [online]. EUROPEAN UNION AGENCY FOR RAILWAYS, 2020. [visited on 2023-03-20]. Available from: https://www.era.europa.eu/system/files/2022-11/index034_-_era_ertms_015560_v23.zip.
7. *Railway signalling since the birth to ERTMS* [online]. railwaysignalling.eu, 2013. [visited on 2023-05-10]. Available from: https://www.railwaysignalling.eu/railway-signalling-pdf.
8. KADLČEK, David; STEJSKAL, Jan; JAHODA, Petr; UDAVICHENKA, Yury; MACHÁČEK, Jiří; VEJVODA, Štěpan. *Návrh projektu DMI displej pro simulátor ETCS rev. 0.1.1* [online]. 2021. [visited on 2023-03-29]. Available from: https://drive.google.com/file/d/1z36wVkQNMtTMDwFy7yTjKPnNGgGuXCMf/view.
9. STŘÍTESKÝ, Petr. *Generic Design of Operational Displays of Railway Vehicles*. 2021. Available also from: http://hdl.handle.net/10467/94818.
10. LIGHT, Roger A. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software*. 2017, vol. 2, no. 13, p. 265. Available from DOI: 10.21105/joss.00265.
11. LOHMANN, Niels. *JSON for Modern C++* [online]. 2022. [visited on 2023-03-29]. Available from: https://json.nlohmann.me/.

12. STEJSKAL, Jan. *ETCS - DMI Display.* 2022. Available also from: http://hdl.handle.net/10467/101907.

13. LIGHT, Roger A. *MQTT man page* [online]. Eclipse Foundation, 2017. [visited on 2023-04-10]. Available from: https://mosquitto.org/man/mqtt-7.html.

14. PEZOA, Felipe; REUTTER, Juan L; SUAREZ, Fernando; UGARTE, Martín; VRGOČ, Domagoj. Foundations of JSON schema. In: *Proceedings of the 25th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, 2016, pp. 263–273.

15. LOHMANN, Niels. *JSON for Modern C++* [online]. 2022. [visited on 2023-03-29]. Available from: https://json.nlohmann.me/home/design_goals/.

16. *Observer* [online]. Refactoring Guru, 2023. [visited on 2023-04-02]. Available from: https://refactoring.guru/design-patterns/observer.

17. *Overview of ASP.NET Core MVC* [online]. Microsoft, 2022. [visited on 2023-04-10]. Available from: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview.

18. *Simple DirectMedia Layer* [online]. SDL Community, 2022. [visited on 2023-04-10]. Available from: https://www.libsdl.org/index.php.

19. MĚŠŤAN, Ondřej. *ETCS DMI II - Implementace rozdílů na lokomotivách provozovaných v České republice oproti normě ETCS.* 2023.

20. ROUSE, Margaret. *What is Technical Debt? – Definition from Techopedia* [online]. Techopedia, 2017. [visited on 2023-04-16]. Available from: https://www.techopedia.com/definition/27913/technical-debt.

21. *Java Code Conventions* [online]. Sun Microsystems, Inc., 1997. [visited on 2023-04-16]. Available from: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf.

22. *Rust Style Guide* [online]. Rust Foundation, 2020. [visited on 2023-04-16]. Available from: https://github.com/rust-lang/style-team/blob/master/guide/guide.md.

23. VAN ROSSUM, Guido; WARSAW, Barry; COGHLAN, Nick. *PEP 8 - Style Guide for Python Code* [online]. 2013. [visited on 2023-04-16]. Available from: https://peps.python.org/pep-0008/.

24. *How does software testing work?* [Online]. IBM, 2023. [visited on 2023-04-16]. Available from: https://www.ibm.com/topics/software-testing.

25. *Singleton* [online]. Refactoring Guru, 2023. [visited on 2023-04-02]. Available from: https://refactoring.guru/design-patterns/singleton.

26. *Apache Log4j Security Vulnerabilities* [online]. The Apache Software Foundation, 2021. [visited on 2023-04-19]. Available from: https://logging.apache.org/log4j/2.x/security.html.

27. MELMAN, Gabi. *spdlog: Fast C++ logging library* [online]. 2023. [visited on 2023-04-24]. Available from: https://github.com/gabime/spdlog.

28. *Linux kernel coding style* [online]. The Linux kernel development community, 2022. [visited on 2023-04-22]. Available from: https://www.kernel.org/doc/html/v4.12/process/coding-style.html.

29. *Using Clang-Format* [online]. JetBrains s.r.o, 2022. [visited on 2023-04-22]. Available from: https://www.jetbrains.com/help/rider/Using_Clang_Format.html.

30. *Microsoft Visual C++ Redistributable latest supported downloads* [online]. Microsoft Corporation, 2023. [visited on 2023-05-11]. Available from: https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist?view=msvc-170.

31. *Visual Studio IDE* [online]. Microsoft Corporation, 2023. [visited on 2023-05-06]. Available from: https://visualstudio.microsoft.com/.

32. *ctest(1) - CMake 3.26.3 Documentation* [online]. Kitware, Inc, 2023. [visited on 2023-05-11]. Available from: `https://cmake.org/cmake/help/latest/manual/ctest.1.html`.

Contents of enclosed media

- `readme.txt` ..... the file with the description of the contents
- `exe` ..... references to an executable binaries of the application
- `src`
  - `impl` ..... references to source code of the implementation
  - `thesis` ..... LaTeX source code of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in the PDF file format