FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	ETCS - Aktualizace a nová architektura komponenty DMI
Student:	Tereza Nepraśová
Vedoucí:	Ing. Jan Matoušek
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

ETCS (European Train Control System) je jednotný celoevropský vlakový zabezpečovací systém. V rámci ČVUT se ve spolupráci dvou fakult (Fakulta informačních technologií a Fakulta dopravní) vyvíjí simulátor tohoto systému.

DMI (Driver Machine Interface) je displej lokomotivy zobrazující údaje o trati a lokomotivě. Cílem práce je implementačně sjednotit tuto komponentu s již existujícími komponentami a zajistit její budoucí udržitelnost a snadnou rozšiřitelnost.

Pokyny pro vypracování:
1) Analyzujte současný stav komponenty DMI a předchozí bakalářské práce k této komponentě.
2) Analyzujte rozdíly v implementaci DMI a komponent RBC (Radio Block Center) a EVC (European Vital Computer)
3) Pomocí metod softwarového inženýrství navrhněte novou architekturu komponenty DMI, jejíž základ bude konzistentní s architekturou RBC a EVC s důrazem na budoucí udržitelnost a rozšiřitelnost.
4) Nově navrženou architekturu implementujte alespoň do stavu, kdy komponenta adekvátně reaguje na zprávy od ostatních komponent systému.
5) Nově napsanou komponentu řádně zdokumentujte a podrobte vhodným testům.
6) Získané zkušenosti shrňte a navrhněte možná rozšíření.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 25. ledna 2024 v Praze.

Bakalářská práce

ETCS – AKTUALIZACE A NOVÁ ARCHITEKTURA KOMPONENTY DMI

Tereza Neprašová

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Matoušek
13. května 2024

Odkaz na tuto práci: Neprašová Tereza. *ETCS – Aktualizace a nová architektura komponenty DMI.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěla bych poděkovat především svému vedoucímu práce, Ing. Janu Matouškovi, za jeho vstřícnost, trpělivost a cenné rady. Děkuji také Martinu Čáslavskému a Ondřeji Veselému za významnou pomoc v zorientování se v projektu a kontrolu mého kódu. V neposlední řadě bych chtěla poděkovat i své rodině a svému příteli za všechnu jejich podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

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

V Praze dne 13. května 2024 .....

Abstrakt

Tato práce se zabývá návrhem nové architektury a aktualizací komponenty DMI, která je součástí simulátoru ETCS. Projekt simulátoru vzniká ve spolupráci Fakulty informačních technologií a Fakulty dopravní ČVUT a je založen na reálném evropském vlakovém zabezpečovacím systému ETCS. Komponenta DMI představuje grafický displej, se kterým strojvedoucí vlaku interaguje. Na základě analýzy nedostatků předchozí architektury DMI a architektury zbylých komponent simulátoru představuje tato práce nový návrh, který se zaměřuje zejména na budoucí rozšiřitelnost a udržitelnost aplikace. Zároveň aktualizuje DMI z verze 2.3.0 na 4.0.0 podle oficiální dokumentace Evropské unie. Výstupem práce je implementace jádra aplikace, komunikace s ostatními komponentami simulátoru, vznik grafického frameworku pro uživatelské rozhraní a implementace několika obrazovek DMI. Aplikace je otestovaná i zdokumentovaná. Hlavním přínosem této práce je výrazné usnadnění a zrychlení budoucího vývoje aplikace a architektonické sjednocení s ostatními komponentami simulátoru.

Klíčová slova ETCS, DMI, C++, SFML, vlakový simulátor, uživatelské rozhraní, návrh architektury, grafický framework

Abstract

This thesis focuses on a new architectural design and update of the DMI component, which is part of the ETCS simulator. The simulator project is developed in cooperation between the Faculty of Information Technology and the Faculty of Transport of CTU and is based on a real European train control system ETCS. The DMI component is a graphical display the train driver interacts with. Based on the analysis of the deficiencies of the former DMI architecture and the architecture of the remaining simulator components, this thesis presents a new design focusing mainly on the future extensibility and sustainability of the application. It also updates the DMI from version 2.3.0 to 4.0.0 according to the official documentation of the European Union. The output of this thesis is the implementation of the application core, communication with other simulator components, the creation of a graphical framework for the user interface and the implementation of several DMI windows. The application is both tested and documented. The main contribution of the thesis is a significant facilitation and acceleration of future application development and architectural unification with other components of the simulator.

Keywords ETCS, DMI, C++, SFML, train simulator, user interface, architecture design, graphical framework

Seznam zkratek

ATO	Automatic Train Operation
BMM	Big Metal Mass
BPMN	Business Process Model and Notation
BTM	Balise Transmission Module
CEM	Common Enums and Messages
DMI	Driver Machine Interface
EDA	Event Driven Architecture
EOA	End of Authority
ERA	European Union Agency for Railways
ERTMS	European Rail Traffic Management System
ETCS	European Train Control System
EVC	European Vital Computer
GSM-R	Global System for Mobile Communications – Railways
ID	Identificator
JRU	Juridical Recording Unit
JSON	JavaScript Object Notation
LPC	Lektorské PC
MQTT	Message Queuing Telemetry Transport
MVC	Model–View–Controller
ODO	Odometry system
RBC	Radio Block Centre
RV	Reversing
SDL	Simple DirectMedia Layer
SFML	Simple and Fast Multimedia Library
SH	Shunting
SM	Supervised Manoeuvre
SOA	Service Oriented Architecture
SR	Staff Responsible
TAF	Track Ahead Free
TIU	Train Interface Unit
UI	User Interface
VBC	Virtual Balise Cover
WSL	Windows Subsystem for Linux

Úvod

ETCS (European Train Control System) je celoevropský vlakový zabezpečovač, který je součástí evropského standardu pro automatickou ochranu vlaků a systému řízení ERTMS (European Rail Traffic Management System). Ten si klade za cíl vytvořit efektivnější a bezpečnější interoperabilní evropskou železniční síť. Systém ERTMS má postupně nahradit různorodé evropské řídící systémy a umožnit tak jednotnou mezinárodní železniční dopravu bez nutnosti výměny strojvedoucího či lokomotivy na hranicích států. Pomocí ETCS lze vzdáleně dohlížet na pohyb vlaku, určit jeho přesnou geografickou polohu, zastavit vlak při překročení maximální povolené rychlosti v každém úseku tratě, či aktivovat nouzové brzdy. [1]

Projekt simulátoru ETCS

Od letního semestru akademického roku 2020/2021 vzniká projekt simulátoru výše zmíněného systému ETCS ve spolupráci dvou fakult ČVUT: Fakulty informačních technologií a Fakulty dopravní. Hlavním cílem projektu je umožnit strojvedoucím školení na tento evropský standard. Předpokládá se zvýšený zájem o taková školení, jelikož od počátku roku 2025 by po železničních tranzitních koridorech v České republice měly jezdit pouze vlaky vybavené ETCS. [2]

S projektem jsem se poprvé setkala v zimním semestru akademického roku 2023/2024 v rámci předmětu Softwarový týmový projekt 2. Přidala jsem se k němu s jasným zájmem o komponentu DMI (Driver Machine Interface). Jedná se o grafický displej na palubě vlaku, se kterým strojvedoucí interaguje. Tato komponenta je jako jediná ze všech vyvíjených součástí simulátoru ETCS původní od doby založení projektu. Na jejím vývoji se podílelo mnoho studentů a byly o ní sepsány tři bakalářské práce. Při vzniku komponenty však byla zvolena architektura, která za roky vývoje velmi zchátrala. Dnes si studenti podílející se na vývoji simulátoru ETCS stěžují na velmi špatnou rozšiřitelnost, nesrozumitelnost kódu a obtížnou udržitelnost komponenty DMI. Má práce vznikla z potřeby řešení tohoto problému.

Kapitola 1
Cíle práce

Hlavním cílem mé práce je navrhnout novou architekturu komponenty DMI, která bude klást důraz na udržitelnost, budoucí rozšiřitelnost a jednoduchost vývoje. Dále je cílem tuto architekturu z důvodu zachování konzistence mezi všemi komponentami simulátoru co nejvíce přiblížit architektuře komponent RBC (Radio Block Centre) a EVC (Euro Vital Computer), které v nedávné době prošly aktualizací. Dalším cílem je navrhnout solidní grafický *framework*, který se bude využívat pro vykreslování uživatelského rozhraní. Návrh je třeba směřovat na specifikační verzi 4.0.0 DMI displeje popsanou v oficiální dokumentaci vydanou Evropskou unií. [3]

Cílem analýzy je seznámit se s doménou simulátoru ETCS a dosavadní architekturou komponenty DMI prostřednictvím tří bakalářských prací, které o ní dosud byly napsány, identifikovat hlavní nedostatky této architektury a porovnat rozdíly v jejím návrhu a návrhu RBC a EVC. Dále je nutné důkladně zanalyzovat komunikaci DMI s ostatními komponentami.

Implementace se bude soustředit na vytvoření jádra aplikace a zajištění správného zpracování všech komunikačních zpráv definovaných v komponentě CEM (Common Enums and Messages). Po domluvě s vedoucím práce a díky dostatku času bude implementován i grafický *framework* pro vykreslování uživatelského rozhraní aplikace. Zároveň budou naprogramovány i některé obrazovky DMI podle dostupné dokumentace od Evropské unie. [3] Cílem mé práce však není implementovat celou aplikaci, a proto implementace zbylých obrazovek a hlavně logiky zobrazování grafických objektů na základě dat získaných z komunikace s ostatními komponentami simulátoru bude úkolem pro budoucí vývojáře.

Cílem testování je zejména pokrýt implementované objekty automatickými jednotkovými testy a manuálními testovacími scénáři.

V neposlední řadě jako součást praktické části vzniknou instalační a vývojářské příručky. Tato práce samotná bude také sloužit jako příručka k návrhu nové architektury, aby budoucí vývojáři podílející se na vývoji DMI mohli pokračovat s implementací tam, kde má práce skončí.

Kapitola 2
Analýza

Velmi důležitou součástí každého softwarového návrhu je analýza celé problematiky. Proto se v této kapitole budu věnovat nejprve analýze domény ETCS simulátoru jako takového, následně historii vývoje komponenty DMI a popisu uživatelského rozhraní. Poté DMI architektonicky zanalyzuji a porovnám s architekturou ostatních komponent, a nakonec provedu analýzu komunikace DMI s ostatními komponentami simulátoru. Celou analýzu zakončím definováním požadavků pro přepis, které vzniknou na jejím základě.

2.1 Analýza simulátoru ETCS

Než začnu s hlavní částí analýzy, je třeba si pro pochopení tématu představit a zadefinovat, co je to ETCS a z jakých částí se skládá.

„*ETCS je standard pro kontrolu vlaků založený na palubní výbavě, která dokáže dohlížet na pohyb vlaku a případně jej zastavit v závislosti na maximální povolené rychlosti v každém úseku tratě. Informace o trati jsou získávány z ETCS zařízení nainstalovaného podél trati, eurobalíz anebo rádia, v závislosti na operační úrovni systému. Tyto informace jsou pak využity k výpočtu maximální povolené rychlosti a k jejímu neustálému monitoringu. Reakce strojvedoucího jsou neustále sledovány a v případě nutnosti může systém ETCS aktivovat nouzové brzdy.*“ [1] (překlad vlastní)

Zabezpečovač ETCS lze rozdělit na dvě části – traťovou a palubní. Celý systém se skládá z mnoha komponent, pro vznikající simulátor ETCS však byly implementovány pouze některé. Ty představím v následující kapitole. Nejprve popíšu komponenty, které jsou součástí reálného systému ETCS a poté i komponenty, které jeho součástí nejsou a vznikly pouze pro účely simulátoru. Schéma celého simulátoru lze nalézt na obrázku 2.3.

2.1.1 Traťová část ETCS

Zde uvádím krátký popis komponent traťové části zabezpečovače ETCS.

RBC (Radio Block Centre)

„*RBC je zařízení, které se využívá na 2. úrovni zabezpečení ETCS a chová se jako centralizovaná bezpečnostní jednotka. Pomocí rádiového spojení získává pozici vlaku a následně odesílá povolení k pohybu a další informace potřebné pro pohyb vlaku. Je také schopno vysílat vybraná data o trati a komunikovat se sousedními RBC.*“ [4] (překlad vlastní)

Eurobalíza

„*Eurobalíza je pasivní zařízení nainstalované přímo na trati. Ukládá data týkající se infrastruktury, jako je rychlostní limit, ukazatele polohy, stoupání a další. Zařízení nepotřebuje zdroj energie, protože jej napájí BTM anténa při průjezdu vlaku přes něj.*“ [4] (překlad vlastní)

Obrázek 2.1 ETCS eurobalíza na trati [5]

2.1.2 Palubní část ETCS

Zde uvádím krátký popis komponent palubní části zabezpečovače ETCS.

EVC (Euro Vital Computer)

EVC je jádrem celé palubní části ETCS a propojuje všechny komponenty. Zpracovává vlakové informace jako jsou údaje přijaté z traťového zařízení ETCS, data zadaná strojvedoucím či údaje z palubních snímačů a zasílá je k zobrazení DMI. Sleduje také bezpečné dodržování jízdních parametrů, jako je povolená rychlost anebo dodržení povolení k vjezdu do úseku. [6]

DMI (Driver Machine Interface)

„*DMI je grafické rozhraní mezi strojvedoucím a ETCS systémem. Ve většině případů se jedná o dotykový LCD displej pro ovládací a indikační funkce, který umožňuje strojvedoucímu zadávat data do systému a zobrazovat výstupní data (např. povolenou rychlost).*“ [4] (překlad vlastní)

Komponenta zobrazuje strojvedoucímu tachometr s aktuální rychlostí vlaku, plánovací informace jako je stoupání či klesání tratě, brzdné křivky atd. Dále má mnoho obrazovek menu, ve kterých se zadávají zmíněná data. DMI je hlavním předmětem mé bakalářské práce a budu se jím do hloubky zabývat v následujících kapitolách.

JRU (Juridical Recording Unit)

„*JRU se chová jako černá skříňka vlaku. Ukládají se do ní nejdůležitější data a proměnné jízdy, čímž pak umožňuje jejich pozdější analýzu.*“ [4] (překlad vlastní)

V projektu ETCS simulátoru JRU představuje logovací komponentu. Ukládá si záznamy o veškeré komunikaci mezi komponentami a další důležité informace o simulaci.

BTM (Balise Transmission Module)

„*BTM je modul uvnitř palubního zařízení ETCS pro přerušovaný přenos z trati na vlak, který zpracovává signály přijímané palubní anténou a získává data z Eurobalízy.*“ [4] (překlad vlastní)

V projektu simulátoru ETCS jsou data z BTM simulována komponentou SimEmulator.

Obrázek 2.2 BTM anténa zespodu vlaku projíždějícího přes eurobalízu [7]

ODO (Odometry system)

„*ODO je počítadlo ujeté vzdálenosti, které je zodpovědné za výpočet rychlosti, zrychlení a ujeté vzdálenosti vlaku. Typicky se skládá z tachometru a radaru.*“ [4] (překlad vlastní)

TIU (Train Interface Unit)

„*TIU je komponenta uvnitř palubního zařízení ETCS, která poskytuje rozhraní mezi ETCS a samotným vlakem pro výměnu informací (např. směr pohybu vlaku) a pro vydávání příkazů kolejovým vozidlům (v případě, že ETCS převezme kontrolu nad vozidlem).*“ [4] (překlad vlastní)

2.1.3 Rozšiřující komponenty simulátoru ETCS

Následuje popis komponent, které vznikly pouze pro účely simulátoru a nejsou tedy součástí oficiálního standardu ETCS.

LPC (Lektorské PC)

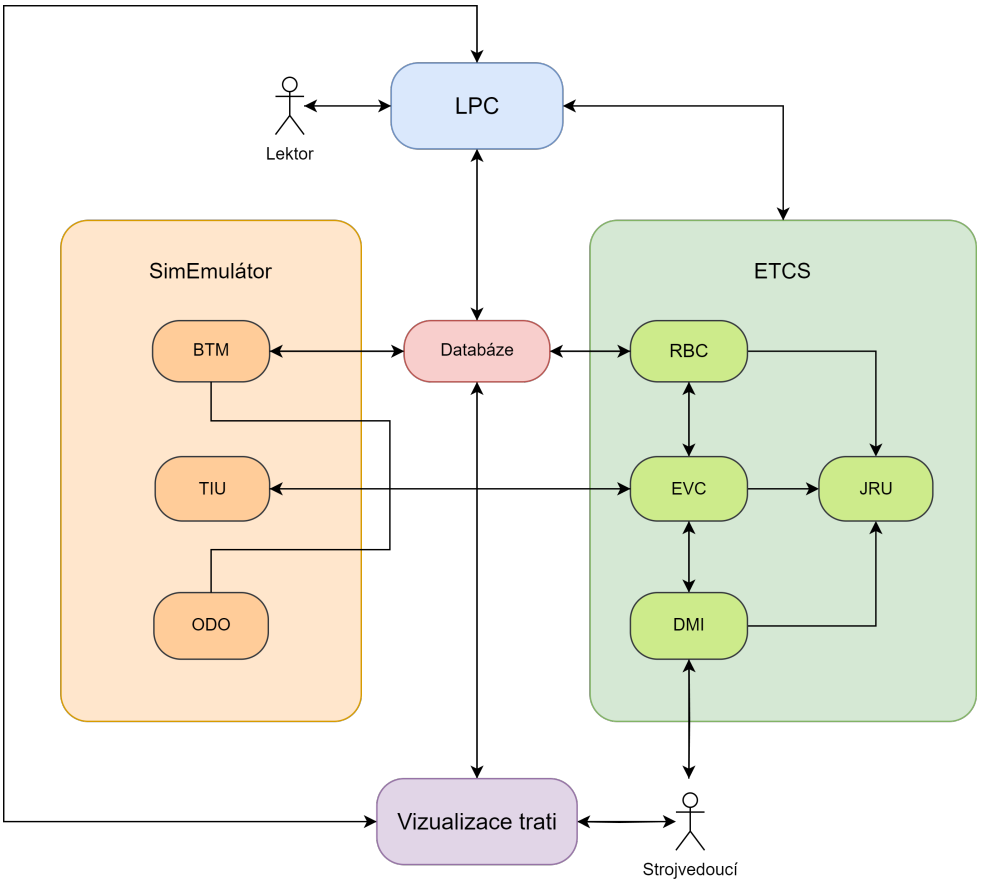
LPC je hlavní řídící komponenta celého simulátoru. Konfiguruje ostatní komponenty, zahajuje a zastavuje celou simulaci a následně ji vyhodnocuje.

CEM (Common Enums and Messages)

CEM je modul definující formu a obsah zpráv, které si komponenty ETCS simulátoru mezi sebou posílají. Používá se typicky jako submodul v ostatních komponentách.

SimEmulator

SimEmulator je emulátor pohybu vlaku. V současné době jsou komponenty BTM, ODO a TIU jeho součástí. Do budoucna se plánuje je oddělit.



Obrázek 2.3 Schéma celého ETCS simulátoru. Zdroj vlastní.

2.2 Analýza historie vývoje komponenty DMI

Počátek vývoje DMI se datuje již od jara roku 2021. Pracovalo na něm každoročně mnoho studentů v rámci předmětů Softwarový týmový projekt 1 a 2 a vývoj pokračuje dodnes. Zároveň vznikly tři bakalářské práce implementující mnoho funkcionalit DMI. Historie aplikace je tedy velmi bohatá a je třeba ji zanalyzovat, aby bylo možné vytvořit kvalitní návrh nové architektury vyvarující se nedostatkům a nesprávným rozhodnutím v architektuře původní. V této kapitole zanalyzuji již zmíněné bakalářské práce pro splnění tohoto cíle. Budu dávat důraz zejména na architekturu, technologie a funkcionality, které tyto práce do projektu přinesly.

2.2.1 Bakalářská práce Jana Stejskala

Jan Stejskal se s projektem setkal již v úplných začátcích. Jeho práce tak popisuje postupný vznik původního DMI. [8]

Analytická část

Velká část práce Jana Stejskala se zabývá analýzou dostupné dokumentace k DMI, tedy konkrétně dokumentem ERA-ERTMS-015560v2.3. [8] Jedná se o oficiální dokument vydaný Evropskou unií pro DMI displej ve specifikační verzi 2.3.0. [9] Tento dokument však pro mou práci již

není relevantní, jelikož je mým cílem začít aktualizaci DMI na verzi 4.0.0 podle příslušné dokumentace. [3] Specifik této novější dokumentace se budu držet po celou dobu návrhu i vývoje. Existují rozdíly mezi verzemi 2.3.0 a 4.0.0, na které bude třeba dbát. Ty, které jsou pro mou práci klíčové, popíši v kapitole 2.3.

Dalším bodem analýzy je diplomová práce Petra Stříteského, který se zabýval tvorbou generické knihovny pro vykreslování grafických displejů drážních vozidel. [10] Tato knihovna, využívající jazyky C, C++ a grafickou knihovnu SDL 2.0, byla poté zvolena jako základ pro vykreslování grafiky DMI. [8] Ve své práci bych však chtěla zvolit jinou cestu. Důvody pro změnu grafické knihovny popíši v sekci 3.1.4.

Návrh architektury

Celá aplikace DMI je napsána v programovacích jazycích C a C++. Na obrázku 2.4 představuje Stejskal jádro architektury původního DMI. V této podobě se udrželo dodnes, ačkoli aplikace narostla o mnoho dalších tříd. Stěžejním prvkem této architektury je třída *CCoordinator*, která zajišťuje přepínání mezi jednotlivými obrazovkami DMI. Další důležitou třídou je *CClient*, již se stará o komunikaci s ostatními komponentami ETCS systému. Prostředníkem mezi klientem a uživatelským rozhraním je *CDataHandler*, který drží data předávaná při komunikaci. [8]

Samotná architektura obrazovek je řešená pomocí návrhového vzoru MVC (Model–View–Controler). Tento vzor slouží k oddělení datového modelu, uživatelského rozhraní a řídící logiky. *Controllery* se starají o zpracování uživatelského vstupu a zpráv od EVC. Jsou zároveň schopny zprávy EVC odesílat. V neposlední řadě také aktualizují data v příslušném *Modelu*. *Views* zajišťují vykreslování uživatelského rozhraní na obrazovku. [8]

Procesy

Poslední část Stejskalovy bakalářské práce, která je pro můj návrh relevantní, je specifikace důležitých procesů původního DMI. Konkrétně zde autor popisuje datový tok mezi EVC a DMI a přepínání mezi obrazovkami aplikace. [8]

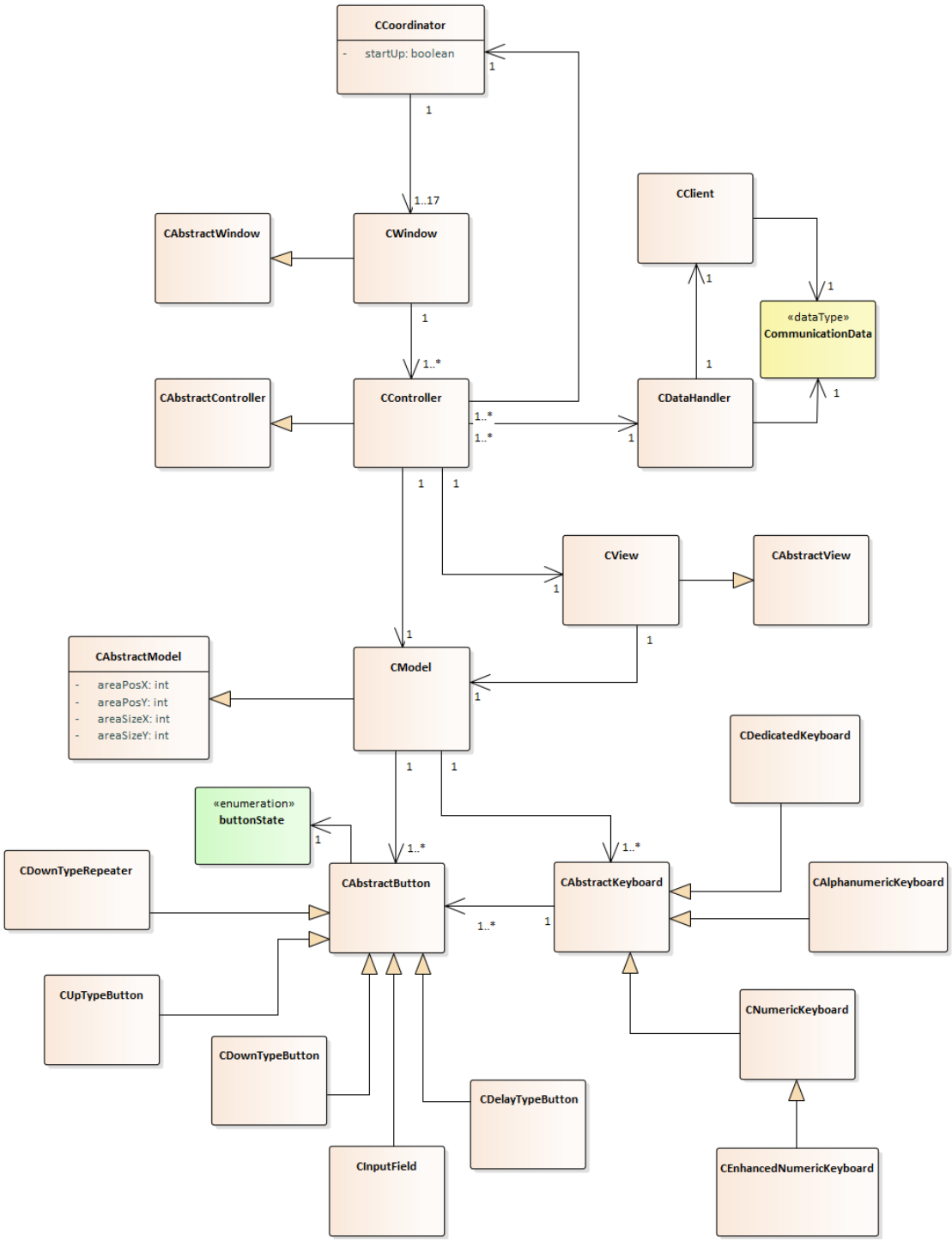
Komunikace mezi jednotlivými komponentami a tedy i EVC a DMI probíhá pomocí MQTT protokolu, jenž popíši v sekci 3.1.2. Pro její realizaci byl využit Eclipse Mosquitto MQTT Broker. [11] [8]

Jak jsem již dříve avizovala, přepínání obrazovek DMI realizuje komponenta *CCoordinator*. Podrobnější popis toho, jakým způsobem se mohou obrazovky mezi sebou přepínat zachycuje přechodová mapa na obrázku 2.5. Způsob přechodu je dán barvou šipky v diagramu. [8]

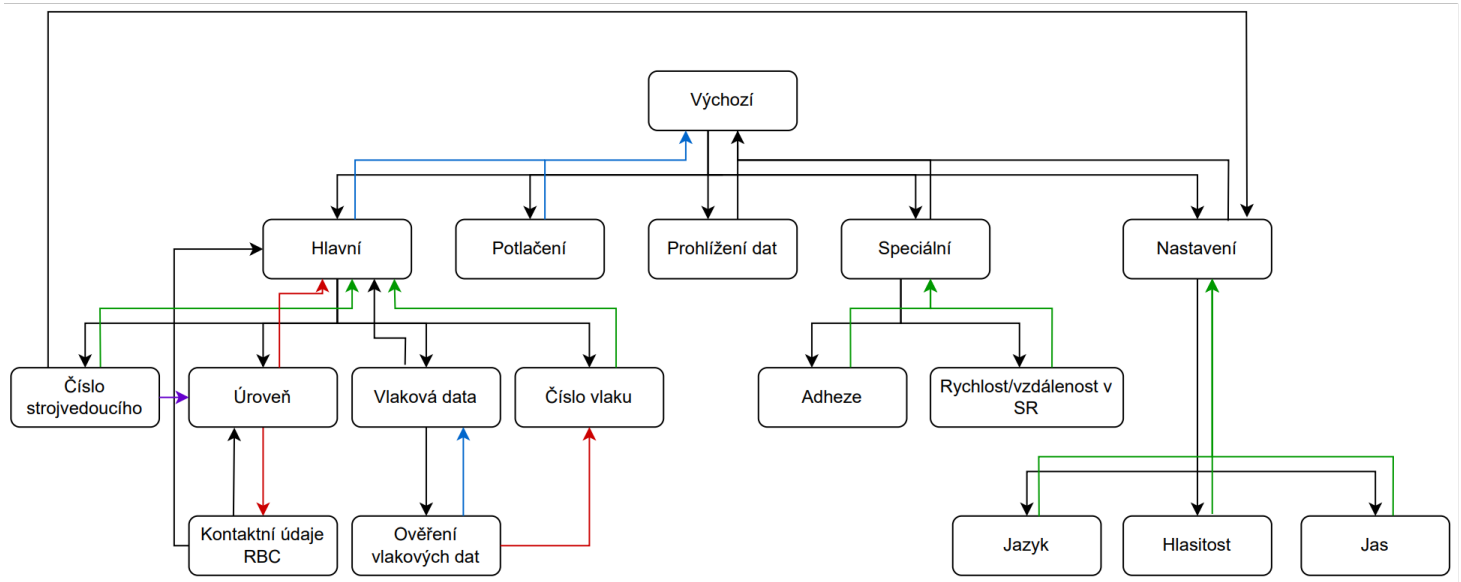
Specifikační verze 4.0.0 však zavedla oproti verzi 2.3.0 do způsobu přepínání obrazovek nuance. Zároveň představuje mnoho nových obrazovek. Tyto změny popíšu v kapitole 2.3.

Závěr

DMI displej je napsán v jazycích C a C++ a využívá pro vykreslování grafickou knihovnu založenou na knihovně SDL 2.0. Pro komunikaci s ostatními komponentami využívá protokol MQTT. Základem architektury je návrhový vzor MVC a několik dalších tříd zastřešujících komunikaci s ostatními komponentami ETCS simulátoru a přepínání obrazovek DMI. Původní DMI displej byl implementován ve verzi 2.3.0, a tak je třeba dbát při návrhu nové architektury na změny zavedené ve verzi 4.0.0.



Obrázek 2.4 Původní architektura DMI [8]



Obrázek 2.5 Přepínání obrazovek. Černá šipka znamená přepnutí pomocí jednoho tlačítka, modrá pomocí několika různých tlačítek. Fialová šipka označuje přepnutí za specifických podmínek, zelená přepnutí po ukončení procesu zadávání či validace dat, a nakonec červená přepnutí na základě určité hodnoty zadané do vstupního pole. [8]

2.2.2 Bakalářská práce Yuryho Udavichenky

Yury Udavichenka se stejně jako Jan Stejskal setkal s projektem již na jeho úplném začátku. Na rozdíl od něj však začal svou bakalářskou práci o rok později a navázal tak na tu Stejskalovu. Hlavním cílem jeho práce bylo rozšířit funkcionality DMI jmenovitě o podporu více jazyků, změnu rozlišení displeje, korektní komunikaci s LPC, dokončení implementace plánovací oblasti DMI a logování. K tomu navíc definoval konvence pro psaní kódu. [12]

Podpora více jazyků

Prvním doplněním funkcionalit DMI byla podpora více jazyků aplikace. Udavichenka vymyslel systém, pro který vytvořil šablonu pro slovník pojmů zobrazovaných v aplikaci. Tato šablona se doplní překlady v příslušném jazyce a přidá ke konfiguračním souborům, odkud si aplikace načítá jazyky. Ty je možné přidávat bez nutnosti rekompilace projektu. Uživatel si poté může po spuštění DMI v okně nastavení jazyků vybrat požadovaný jazyk. [12]

Změna rozlišení displeje

V dřívější verzi DMI bylo rozlišení displeje napevno svázáno s kódem, což je velmi nežádoucí praktika pro konfigurovatelnost projektu. Z toho důvodu Udavichenka implementoval možnost konfigurace rozlišení v konfiguračním souboru, bez nutnosti rekompilace celého projektu. V návaznosti na to také musel upravit původní grafickou knihovnu Petra Stříteského a přidat k ní funkce, které budou umožňovat škálování vykreslovaných objektů. [12]

Komunikace s LPC

Kromě komunikace s EVC je pro DMI důležitá také komunikace s LPC, který řídí simulaci. LPC potřebuje nějakým způsobem průběžně ověřovat, zda je DMI aktivní a zda nedošlo k chybě. Proto byl implementován tzv. *Heartbeat*. Jedná se o techniku, kdy aplikace periodicky odesílá zprávy na jiné místo v síti. [12]

Součástí požadavku bylo také zasílání kontrolních zpráv od LPC do DMI, pomocí kterých se bude měnit stav aplikace. Například by mělo být možné DMI spustit, zastavit anebo restartovat z LPC, či do budoucna přidat další kontrolní zprávy. [12]

Plánovací oblast DMI

V rámci práce došlo také k dokončení k zobrazení plánovacích informací jízdy podle dokumentace specifikační verze 2.3.0. Ve verzi 4.0.0 však došlo k drobným grafickým změnám, které bude třeba do nového DMI zahrnout. [3] Tyto změny jsou popsány v uvedené dokumentaci a nebudu je zde kvůli jejich irelevanci uvádět. Implementace plánovací oblasti totiž není součástí mé práce.

Logování

Poslední funkcionalitou, kterou přinesla Udavichenkova práce, je logování. Pro jeho realizaci byla vybrána knihovna spdlog. Bylo však implementováno pouze logování inicializace projektu. [12]

Po analýze projektového GitLab repozitáře jsem zjistila, že tato implementace logování v DMI nakonec nikdy nebyla využita. Namísto toho se dnes pro souhrnné logování všech komponent projektu využívá komponenta JRU.

Závěr

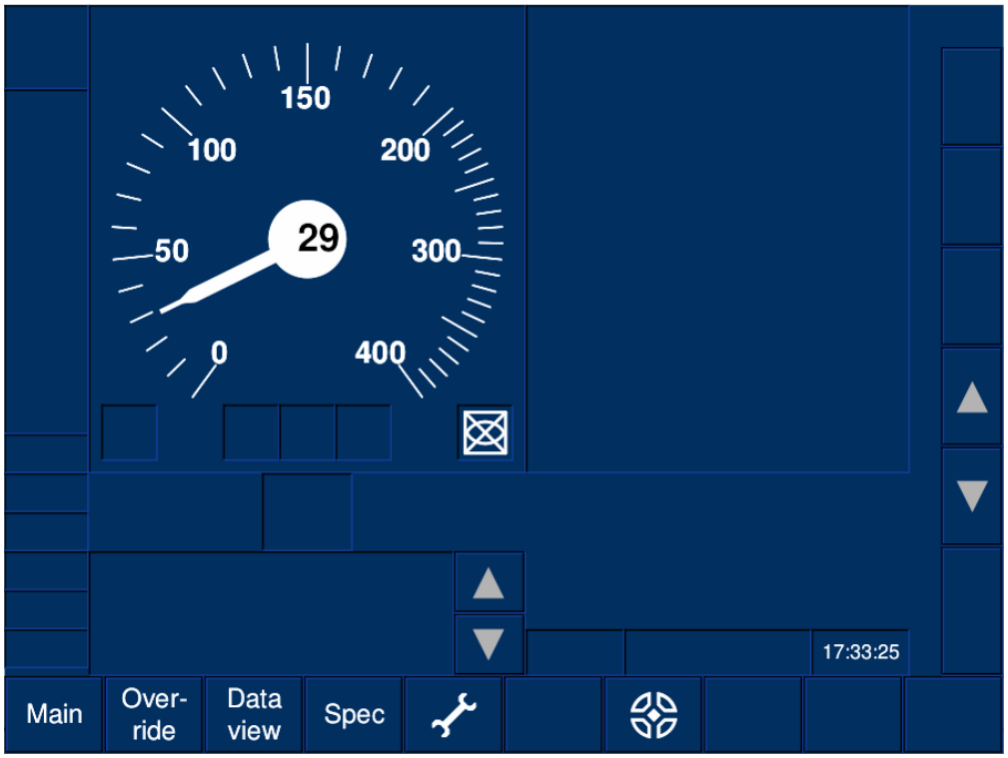
Udavichenka rozšířil DMI o několik důležitých funkcionalit. Za největší přínos z hlediska návrhu považuji zavedení konfigurovatelnosti aplikace bez nutnosti rekompilace projektu. Tuto ideu bych chtěla reflektovat i ve svém návrhu nové architektury.

2.2.3 Bakalářská práce Ondřeje Měšťana

Ondřej Měšťan psal svou bakalářskou práci paralelně s Yurym Udavichenkou a odkazují se tak ve svých pracích jeden na druhého. Měšťan se zabýval hlavně výzkumem displejů pro lokomotivy typu Bombardier Traxx MS2 a Siemens Vectron. Vytvořil softwarový návrh pro displej Bombardier Traxx MS2 a následně jej implementoval. Spolu s ním také rozšířil projekt o emulaci ovládání nedotykového displeje technologií soft-key. [13]

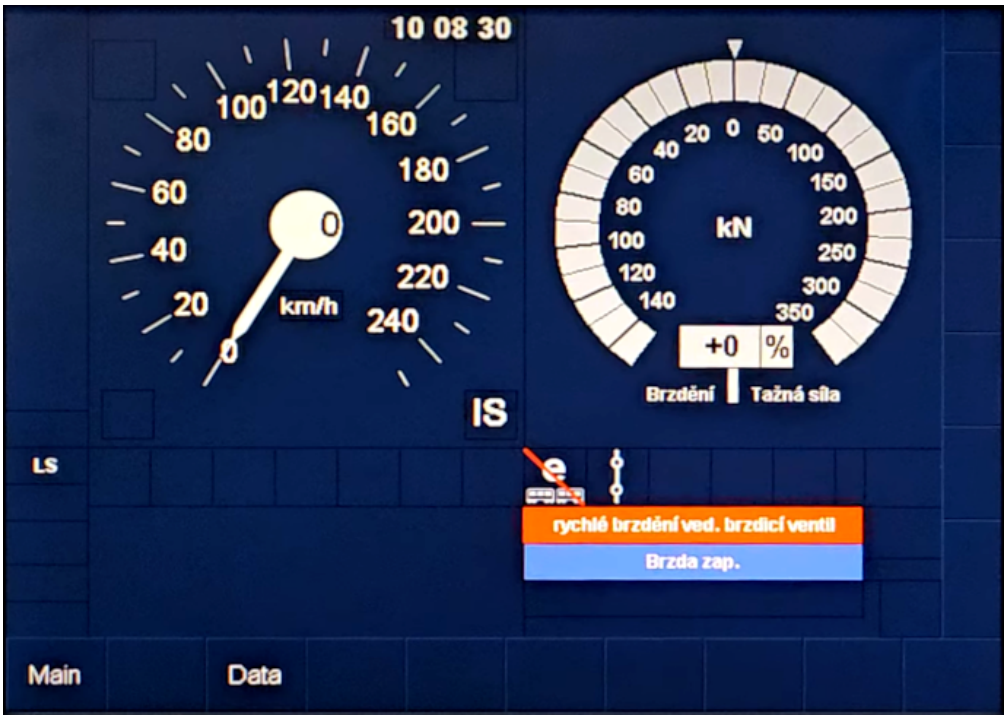
Různé typy displejů

Měšťan ve své práci popisuje, že rozdíly mezi DMI displejem dle dokumentu ERA-ERTMS-015560 verze 2.3.0 [9] (obrázek 2.6) a displeji lokomotivy Bombardier Traxx MS2 (obrázek 2.7) a Siemens Vectron (obrázek 2.8) jsou velké. Liší se nejen vzhled grafického rozhraní a rozložení objektů na obrazovce, ale i některé funkcionality. Obrazovky lokomotiv typu Traxx MS2 i Vectron například umožňují zobrazovat kruhový diagram aktuálního kumulativního výkonu motorů a brzd, což dokumentace ERA nezahrnuje. Přidaných funkcionalit se zde objevuje více, pro podrobnější analýzu však odkazuji na Měšťanovu bakalářskou práci. [13]



Obrázek 2.6 Výchozí obrazovka DMI dle normy ERA s technologií soft-key [13]

Obrázek 2.7 Výchozí obrazovka DMI lokomotivy Traxx MS2 z videa z kabiny [13]



Obrázek 2.8 Výchozí obrazovka DMI lokomotivy Vectron z videa z kabiny [13]

Implementace

V rámci implementace došlo k rozšíření proektu o emulaci soft-key technologie, definovanou v dokumentaci ERA [9]. Jedná se o rámeček s tlačítky kolem původního grafického rozhraní, kterým je možné displej ovládat. V režimu soft-key je původní dotykové ovládání neaktivní. [13]

Dále byla aplikace rozšířena o mnoho konfigurací, jako je nastavitelnost barevné sady displeje, podoby a umístění prvků na displeji a možnost změny vzhledu existujících obrazovek pomocí konfiguračních souborů tak, aby byla zachována jejich funkčnost. [13]

V neposlední řadě došlo k implementování displeje lokomotivy Bombardier Traxx MS2 a jeho přidaných funkcionalit a grafických prvků. [13]

Nevyužitá práce

Po analýze projektového GitLab repozitáře jsem dospěla ke zjištění, že velká část Měšťanovy práce nikdy nebyla sloučena do hlavní větve projektu a nikdy se tedy nedostala do produkce.

Práce se nachází na větvi `feature/traxx_ms2_model`, která má kořen ve větvi `master`. Je zde celá implementace displeje lokomotivy Bombardier Traxx MS2. Dnes už bude obtížné, a hlavně časově náročné ji začlenit do produkce, protože projekt se od doby jejího dokončení dál rozvíjel.

Závěr

Měšťanova práce přinesla do projektu významné rozšíření, ze kterého je třeba si odnést to, že vzhled a funkcionality DMI displeje podle dokumentace ERA do budoucna nemusí být jediné, které bude třeba v projektu implementovat. Můj návrh nové architektury by tedy měl být snadno rozšiřitelný o modely dalších displejů.

2.3 Analýza uživatelského rozhraní DMI

Před tím, než se pustím do analýzy architektury DMI, je třeba zanalyzovat jeho uživatelské rozhraní. Vzhled UI (User Interface) je sice rozebírán v předešlých bakalářských pracích, ty však implementují verzi 2.3.0 DMI. Ve verzi 4.0.0, na kterou mám DMI aktualizovat, bylo přidáno několik obrazovek, které je potřeba pro úplnost tohoto textu představit, jelikož některé z nich budu v rámci praktické části své práce implementovat. V tabulkách 2.1, 2.2 a 2.3 tedy krátce představuji všechny obrazovky DMI verze 4.0.0 a následně na obrázku 2.12 uvádím, jakým způsobem dochází k přechodům mezi nimi. Obrazovky musí být implementovány striktně podle specifikace v dokumentaci od ERA verze 4.0.0, ve které lze nalézt detailní popis vzhledu všech níže představených obrazovek. [3] Ten zde pro jeho rozsáhlost nebudu uvádět.

Menu obrazovka DMI	Popis
Hlavní menu	Tlačítkové menu, které slouží jako rozcestník k dalším obrazovkám. Některá jeho tlačítka však místo přepnutí obrazovky rovnou vykonají akci (tlačítka *Shunting*, *Non-Leading*, *Maintain shunting*, *Initiate SM* a *Exit SM*). [3] Pro více informací, co mají tlačítka provádět, odkazuji do přílohy C.
ID strojvedoucího	Menu pro zadávání dat na polovinu obrazovky. Slouží ke specifikaci alfanumerického identifikátoru strojvedoucího. [3]
Úroveň (*Level*)	Menu pro zadávání dat na polovinu obrazovky. Slouží pro upřesnění úrovně zabezpečení ETCS. [3]

Tabulka 2.1 Přehled obrazovek menu DMI (1. část)

Menu obrazovka DMI	Popis
Vlaková data	Menu pro zadávání dat na celou obrazovku. Zde se zadává mnoho různých dat klíčových pro jízdu vlaku. Například kategorie vlaku, jeho délka, maximální rychlost, vzduchotěsnost, kategorie zatížení nápravy atd. [3]
Validovat vlaková data	Validační menu vlakových dat. Slouží k potvrzení odeslání dat. [3]
Číslo vlaku	Menu pro zadávání dat na polovinu obrazovky. Slouží k zadaní či revalidaci čísla vlaku. [3]
Rádiová data	Tlačítkové menu, které slouží jako rozcestník k dalším obrazovkám. Tlačítka *Enter RBC data*, *Radio network type*, *GSM-R network ID* a *Mission with one radio system* slouží k přepnutí na příslušnou obrazovku. Tlačítka *Contact last RBC* a *Use short number* rovnou vykonávají nějakou akci. [3] Pro více informací, co mají tlačítka provádět, odkazuji do přílohy C.
ID GSM-R sítě	Menu pro zadávání dat na polovinu obrazovky. Slouží ke specifikaci či revalidaci identifikátoru GSM-R sítě (Global System for Mobile Communications – Railways). ID je jeden z pevně daných řetězců. Při přepnutí na tuto obrazovku se napřed musí získat dostupné sítě od EVC. Pokud se je nepovedlo získat, tato obrazovka se nezobrazí a vypíše se chybová hláška. [3]
RBC data	Menu pro zadávání dat na celou obrazovku. Slouží pro specifikaci identifikátoru a telefonního čísla RBC. [3]
Typ rádiové sítě	Menu pro zadávání dat na polovinu obrazovky. Slouží ke specifikaci či revalidaci typu rádiové sítě. [3]
Mise s jedním rádiovým systémem	Menu pro zadávání dat na polovinu obrazovky. Slouží pro specifikaci, zda se jedná o misi s jedním rádiovým systémem či nikoliv. [3]
Potlačení (*Override*)	Tlačítkové menu s jediným tlačítkem označeným *EOA* (End of Authority). Toto tlačítko spustí funkci potlačení a přejde na výchozí obrazovku. [3] Pro více informací, co tlačítko provádí, odkazuji do přílohy C.
Prohlížení dat	Menu pro výpis dat. Nachází se zde tabulka s přehledem všech dat, které strojvedoucí v UI zadal. [3]
Speciální	Tlačítkové menu se čtyřmi tlačítky. Tlačítka *SR speed/distance* a *Adhesion* slouží k přepnutí na příslušnou obrazovku. Tlačítka *Train intergrity* a *BMM reaction inhibition* rovnou vykonávají nějakou akci. [3] Pro více informací, co mají tlačítka provádět, odkazuji do přílohy C.
Adheze	Menu pro zadávání dat na polovinu obrazovky. Slouží ke specifikaci či revalidaci typu adheze. [3]
SR rychlost/vzdálenost	Menu pro zadávání dat na celou obrazovku. Slouží pro specifikaci SR (Staff responsible) rychlosti a vzdálenosti. [3]

Tabulka 2.2 Přehled obrazovek menu DMI (2. část)

Menu obrazovka DMI	Popis
Nastavení	Tlačítkové menu, které slouží čistě jako rozcestník k dalším obrazovkám. [3]
Vybrat ATO	Menu pro zadávání dat na polovinu obrazovky. Slouží ke specifikaci či revalidaci zapnutí ATO (Automatic Train Operation). [3]
Jazyk	Menu pro zadávání dat na polovinu obrazovky. Slouží pro nastavení jazyka celého uživatelského rozhraní. [3]
Hlasitost	Menu pro nastavení určité hodnoty. Zde se nastavuje hlasitost zvuku uživatelského rozhraní. [3]
Jas	Menu pro nastavení určité hodnoty. Zde se nastavuje jas uživatelského rozhraní. [3]
Verze systému	Menu pro výpis dat. Nachází se zde pouze údaj o čísle verze systému. [3]
Nastavit VBC	Menu pro zadávání dat na celou obrazovku. Slouží pro zadání či revalidaci VBC kódu (Virtual Balise Cover). [3]
Odstranit VBC	Menu pro zadávání dat na celou obrazovku. Slouží pro odstranění VBC kódu. [3]
Validovat nastavení VBC	Validační menu nastavení VBC kódu. Slouží k potvrzení odeslání kódu. [3]
Validovat odstranění VBC	Validační menu odstranění VBC kódu. Slouží k potvrzení odstranění kódu. [3]

Tabulka 2.3 Přehled obrazovek menu DMI (3. část)

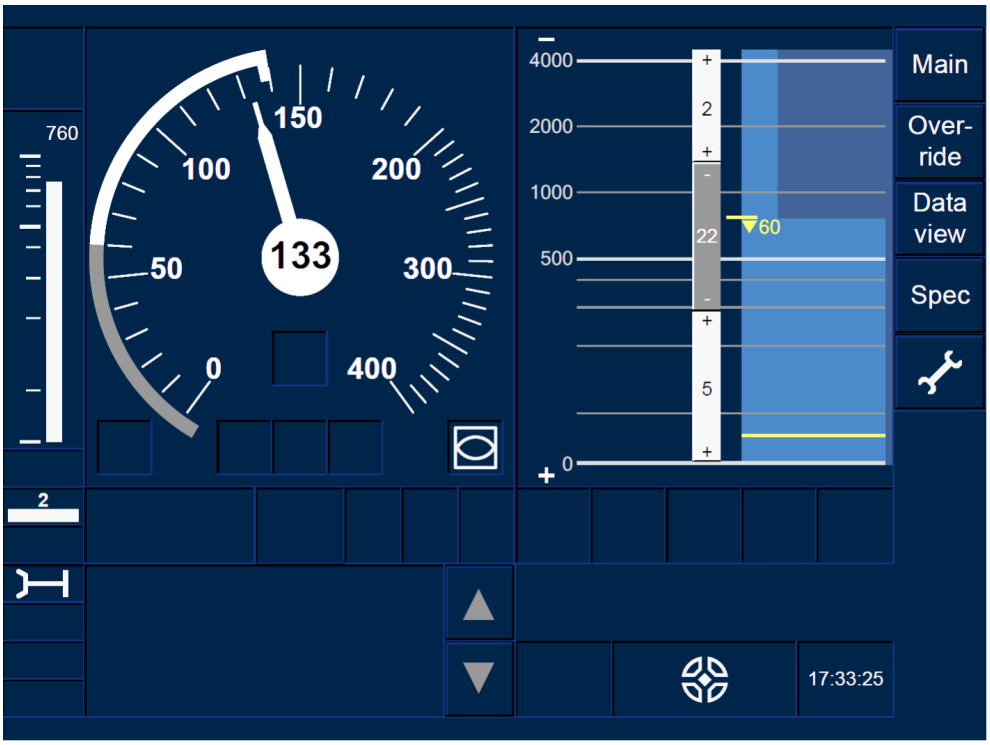
Nyní už schází jen představit poslední a zároveň nejdůležitější obrazovku, a tou je výchozí obrazovka. Zde se soustředí klíčové prvky DMI, jako je tachometr, ukazatel vzdálenosti k příští balíze, příchozí zprávy, plánovací oblast a další. Tuto obrazovku lze vidět na obrázku 2.9. Oficiální dokumentace ji rozděluje na několik menších částí podle obrázků 2.10 a 2.11. V tabulce 2.4 a 2.5 pak popisuji, co jednotlivé části obsahují.

Část výchozí obrazovky	Popis
Část A	V této části lze nalézt ukazatel vzdálenosti k cíli a doby do indikace, dále ukazatel adhezivního faktoru a nejnižší kontrolované rychlosti. [3]
Část B	Zde se nachází tachometr, na kterém se zobrazuje aktuální rychlost vlaku, maximální povolená rychlost, kruhový ukazatel rychlosti kolem tachometru a další. V této části se také zobrazují symboly s informacemi o aktuálním módu ETCS, příkazech a ohlášeních o podmínkách trati anebo autorizaci kontrolovaného manévru. [3]

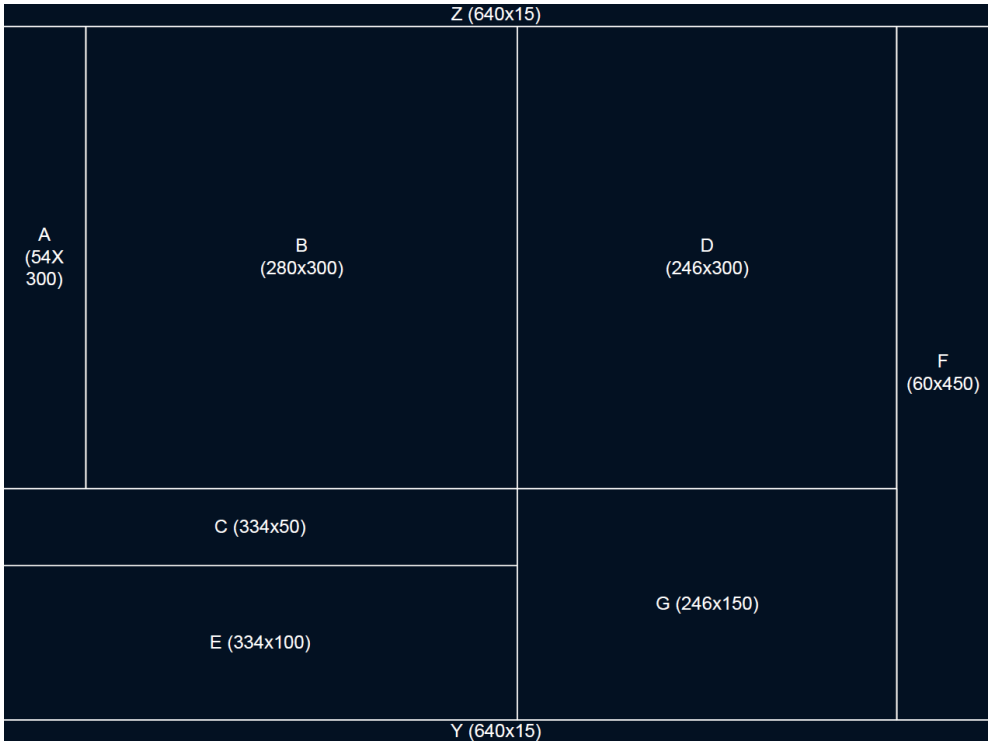
Tabulka 2.4 Přehled jednotlivých částí výchozí obrazovky DMI (1. část)

Část výchozí obrazovky	Popis
Část C	V této části dochází k potvrzování přechodu do jiného módu ETCS či úrovně zabezpečení. Zobrazují se zde symboly potlačení (pokud je potlačení aktivní), momentální úrovně zabezpečení, stavu trati v prostoru zastávky v tunelu, povolení couvání a další. [3]
Část D	Tato část slouží pro zobrazení plánovacích informací. Poskytuje strojvedoucímu přehled o podmínkách trati před vlakem. Nachází se zde informace o stoupání a klesání trati (gradient), o změně nejvyšší povolené rychlosti, příkazy a oznámení o podmínkách trati a indikátor vzdálenosti k cíli. [3]
Část E	Zde se nachází výpis zpráv obdržených od trati. Některé zprávy vyžadují potvrzení od strojvedoucího podobně jako přechody do jiných módů ETCS. [3]
Část F	Jedná se o jednoduchou postranní lištu, s nabídkou tlačítek. Tato tlačítka umožňují strojvedoucímu otevřít další obrazovky menu – *Hlavní menu*, *Potlačení*, *Prohlížení dat*, *Speciální* a *Nastavení*. [3]
Část G	V této části se nachází informace o aktuálním lokálním čase, geografická pozice vlaku a informace související s ATO. [3]

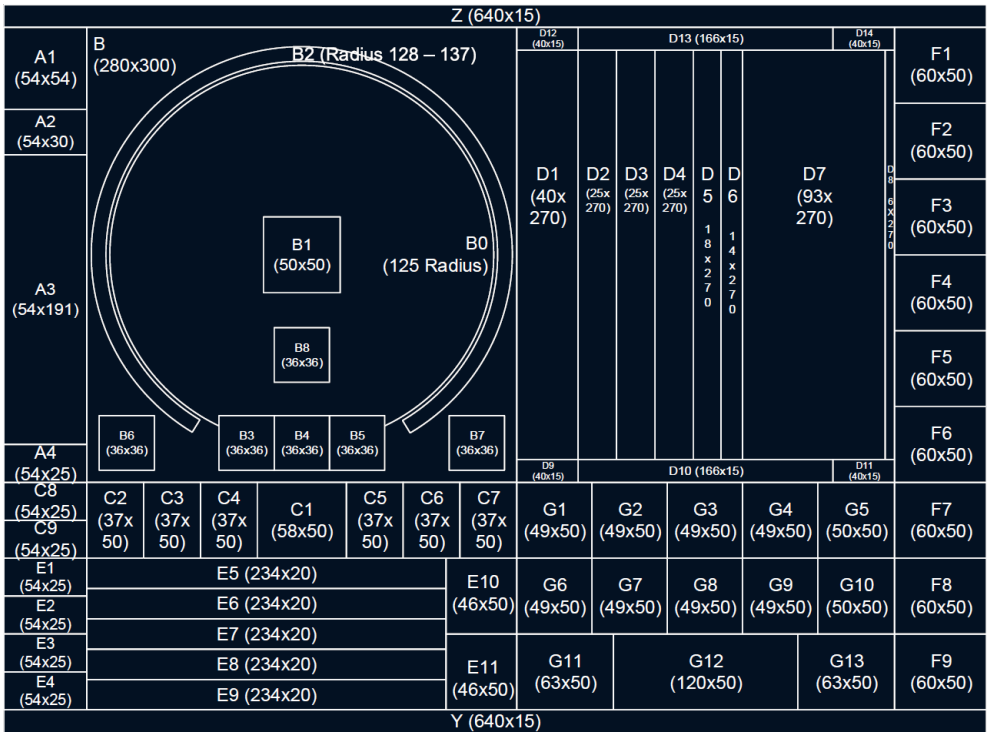
Tabulka 2.5 Přehled jednotlivých částí výchozí obrazovky DMI (2. část)



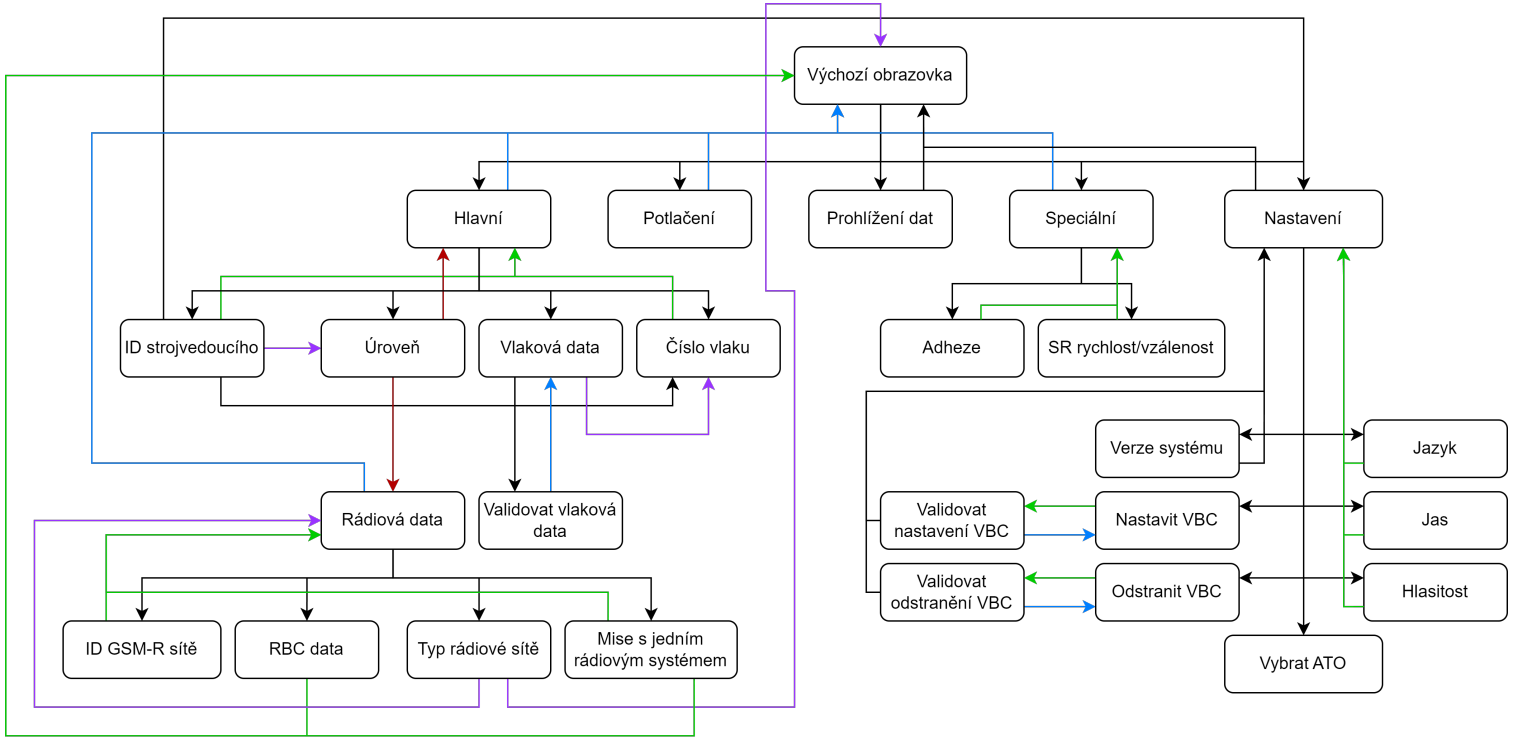
Obrázek 2.9 Ukázka vzhledu výchozí obrazovky DMI. Zdroj je oficiální dokumentace ERA [3].



Obrázek 2.10 Rozdělení výchozí obrazovky na jednotlivé části. Zdroj je oficiální dokumentace ERA [3].



Obrázek 2.11 Detailní rozdělení výchozí obrazovky na podčásti. Zdroj je oficiální dokumentace ERA [3].



Obrázek 2.12 Přepínání obrazovek verze 4.0.0. Zachovala jsem pro lepší porovnání notaci, kterou ve své práci využil Jan Stejskal pro popis verze 2.3.0. Černá šipka znamená přepnutí pomocí jednoho tlačítka, modrá pomocí několika různých tlačítek. Fialová šipka označuje přepnutí za specifických podmínek, zelená přepnutí po ukončení procesu zadávání či validace dat, a nakonec červená přepnutí na základě určité hodnoty zadané do vstupního pole. [3]. Pro detailnější popis přechodů odkazuji do přílohy C.

2.4 Analýza rozdílů v návrhu DMI a ostatních komponent simulátoru

Důležitou součástí mého návrhu nové architektury pro DMI je zajistit, aby tato architektura byla pokud možno co nejvíce konzistentní s architekturami již existujících komponent. Jmenovitě s komponentami RBC, EVC a JRU, které také v nedávné době prošly aktualizací a přepisem. Konzistence při vývoji hraje důležitou roli, protože vývojářům usnadní porozumění a orientaci v kódu. V následující kapitole se zaměřím na hlavní rozdíly mezi návrhem DMI a návrhem ostatních komponent.

2.4.1 Původní architektura DMI

Základ architektury DMI byl zanalyzován a popsán v sekci 2.2.1, tudíž se nebudu opakovat. Nicméně, než budu pokračovat k rozboru architektur ostatních komponent simulátoru, je třeba identifikovat důvody pro nutnost návrhu nové architektury. Díky tomu si jasně vytyčím cíle a problémy, které by měl můj návrh vyřešit.

Nedostatky a důvody pro návrh nové architektury

V průběhu analýzy zdrojových kódů DMI jsem narazila na mnoho prohřešků vůči kvalitnímu softwarovému návrhu a korektnímu využívaní programovacího jazyka C a C++. Studenti podílející se na vývoji simulátoru ETCS mi také nahlásili mnoho problémů s aplikací. Níže tedy identifikuji hlavní nedostatky původního návrhu, které vedly k rozhodnutí, že přepis komponenty na novou architekturu je pro budoucí vývoj vskutku nutný.

Vysoká provázanost: Cílem kvalitního softwarového návrhu je, aby byl každý objekt maximálně zapouzdřen a zároveň, aby při změně jeho vnitřní implementace ideálně nebylo potřeba měnit žádný jiný objekt. Tento jev se označuje za nízkou provázanost (anglicky *Low Coupling*). [14] Návrh DMI ovšem disponuje přesným opakem. Na mnoha místech v kódu objekty přímo přistupují k členským proměnným jiných objektů. V projektu je také definováno mnoho globálních proměnných (například ve výpisu kódu 2.1), ke kterým přistupují objekty napříč celou aplikací. [13] Pokud by se taková proměnná změnila, bylo by potřeba přepsat velké množství zdrojových souborů, což není žádoucí.

```
1  /* INPUT VARIABLES */
2  int MOUSEx, MOUSEy, TOUCHx, TOUCHy;
3  bool click;
4
5  /* EVENTS */
6  SDL_Event e;
7  SDL_KeyboardEvent k;
8
9  std::queue <ClickEvent> eventQueue;
10 std::queue <ESoftKey> g_softKeySignals;
```

Výpis kódu 2.1 Globální proměnné definované v souboru `Definitions.cpp`, ke kterým přistupuje mnoho objektů za účelem detekce kliknutí myši a získání informací o událostech spojených s grafikou [13]

Špatná rozšiřitelnost: Zdrojovému kódu je velmi obtížné porozumět, neboť není téměř vůbec zdokumentován. Proměnné využívané ve funkcích a metodách často svými názvy nevypovídají nic o tom, co znamenají a k čemu se používají. [13] Do takového kódu je opravdu složité přidávat nové funkcionality. Musela by tomu předcházet několik dnů až týdnů dlouhá analýza, a i přesto bude implementace náchylná na chybovost. Jako příklad těžko rozšiřitelného a pochopitelného kódu může posloužit výpis 2.2.

```
1  static void SDL_RenderDrawCircle(SDL_Renderer *renderer, int midpointX, int midpointY,
2                                   int radius) {
3      int oX, oY, d;
4      int errors;
5
6      oX = 0;
7      oY = radius;
8      d = radius - 1;
9      errors = 0;
10
11     while (oY >= oX) {
12         errors += SDL_RenderDrawPoint(renderer, midpointX + oX, midpointY + oY);
13         errors += SDL_RenderDrawPoint(renderer, midpointX + oY, midpointY + oX);
14         errors += SDL_RenderDrawPoint(renderer, midpointX - oX, midpointY + oY);
15         errors += SDL_RenderDrawPoint(renderer, midpointX - oY, midpointY + oX);
16         errors += SDL_RenderDrawPoint(renderer, midpointX + oX, midpointY - oY);
17         errors += SDL_RenderDrawPoint(renderer, midpointX + oY, midpointY - oX);
18         errors += SDL_RenderDrawPoint(renderer, midpointX - oX, midpointY - oY);
19         errors += SDL_RenderDrawPoint(renderer, midpointX - oY, midpointY - oX);
20
21         if (errors < 0) {
22             // TODO use this for error handling
23             errors = -1;
24             break;
25         }
26
27         if (d >= 2 * oX) {
28             d -= 2 * oX + 1;
29             oX += 1;
30         } else if (d < 2 * (radius - oY)) {
31             d += 2 * oY - 1;
32             oY -= 1;
33         } else {
34             d += 2 * (oY - oX - 1);
35             oY -= 1;
36             oX += 1;
37         }
38     }
39 }
```

Výpis kódu 2.2 Funkce pro vykreslování kruhu ze souboru `Functions.cpp` [13]

Nízká soudržnost: Každý objekt objektově orientovaného návrhu by měl ideálně řešit pouze jeden konkrétní úkol. Tomuto jevu se říká vysoká soudržnost (anglicky *High Cohesion*), protože všechny části jednoho objektu spolu úzce souvisí a řeší jeden společný problém. [14] To zároveň podporuje vyšší míru modularity. V DMI je však soudržnost nízká. Velká část implementovaných objektů zde řeší mnoho různých nesouvisejících problémů. Příkladem by mohla být třída `CClient` ve výpisu kódu 2.3

```
1  class CClient : public MQTTClient {
2  public:
3      void ProcessEVCMessage(const std::string& json);
4      void ProcessEvcGeneralDataMessage(nlohmann::json& json);
5      void ProcessEvcStoredLevelPositionMessage(nlohmann::json& json);
6      void ProcessEvcSessionEstablishedMessage(nlohmann::json& json);
7      void ProcessEvcStoredTrainDataMessage(nlohmann::json& json);
8      void ProcessEvcModeProposed(nlohmann::json& json);
9      void ProcessEvcShowTAFMessage(nlohmann::json& json);
10     void ProcessEvcEmergencyBrakingMessage(nlohmann::json& json);
11     void ProcessGeneralTextMessage(nlohmann::json& json);
12     void ProcessTranslatedTextMessage(nlohmann::json& json);
13     void ProcessInformationAboutCabin(nlohmann::json& json);
14
15     void ProcessETCSMessage(const std::string& json);
16     void ProcessLPCMessage(const std::string& json);
17     void ProcessSimulationStatusMessage(nlohmann::json& json);
18 }
```

Výpis kódu 2.3 Ukázka části hlavičkového souboru třídy `CClient`. Tato třída řeší veškerou komunikaci se všemi komponentami simulátoru na jednom místě. Množství posílaných zpráv se ale v průběhu vývoje rapidně zvyšuje a třída tak rychle roste. [13]

Nízká modularita: Soubory zdrojových kódů jsou velmi dlouhé, špatně se v nich orientuje a zbytečně se snaží řešit mnoho nesouvisejících problémů na jednom místě. Zejména narážím na soubory `Functions.cpp` a `Definitions.cpp`, které obsahují základní funkce pro vykreslování grafického rozhraní a jeho správu – od inicializace grafické knihovny a vykreslování objektů, přes nastavování jasu obrazovky až po funkce pro detekování kliknutí myši. Tyto věci spolu vůbec nesouvisí a měly by být od sebe logicky odděleny. Z důvodu délky těchto souborů je zde jako ukázku nepřikládám. Lze je však nalézt v příloze bakalářské práce Ondřeje Měšťana. [13]

Obtížná udržitelnost: Tento bod úzce souvisí se všemi výše zmíněnými. Kód s takovými nedostatky se bude zkrátka velmi špatně udržovat. Spolu s tím je také třeba zmínit, že aplikace není pokryta žádnými automatickými testy. To může vést k mnoha neodhaleným chybám v kódu, které na udržitelnosti dál ubírají.

Nevyužívání polymorfismu: Velkou součástí programovacího jazyka C++ je technika zvaná polymorfismus, která je důležitá pro objektově orientované programování. Jedná se o schopnost objektů poskytovat různou implementaci stejného rozhraní pomocí dědičnosti. [15] Kód by se dal na mnoha místech zpřehlednit a zkrátit tím, že by se polymorfismus správně využíval. Názorný příklad, kde by jej šlo využít, je ve výpisu kódu 2.4.

```
switch (m_window) {
            case EWindow::defaultWindow:
                this->m_DefaultWindow->Display();
                break;
            case EWindow::mainWindow:
                this->m_DefaultWindow->DisplayOnlyLeftHalf();
                this->m_MainWindow->Display();
                break;
            case EWindow::mainWindowDisabled:
                this->m_DefaultWindow->DisplayOnlyLeftHalf();
                this->m_MainWindow->DisplayDisabled();
                break;
            case EWindow::driverIDWindow:
                this->m_DefaultWindow->DisplayOnlyLeftHalf();
                this->m_DriverIDWindow->Display();
                break;
            case EWindow::dataViewWindow:
                this->m_DefaultWindow->DisplayOnlyLeftHalf();
                this->m_DataWindow->Display();
                break;
            case EWindow::levelWindow:
                this->m_DefaultWindow->DisplayOnlyLeftHalf();
                this->m_LevelWindow->Display();
                break;
}
```

Výpis kódu 2.4 Ukázka logiky pro přepínání obrazovek ze souboru `CCoordinator.cpp`. Tento dlouhý switch blok, který má celkem téměř 70 řádků by šlo nahradit jednoduchým polymorfním voláním, jelikož všechny objekty `Window` dědí z třídy `CAbstractWindow`. [13]

2.4.2 Nová architektura RBC, EVC a JRU

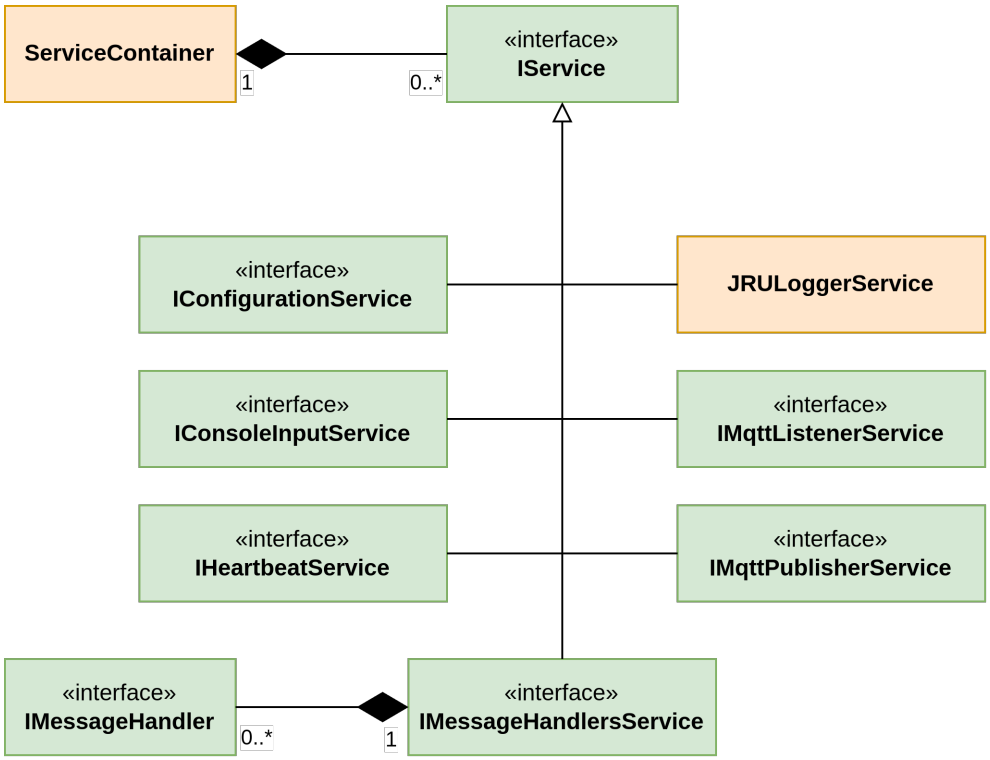
Architektura RBC, EVC a JRU se snaží všem výše zmíněným nedostatkům, kterými DMI disponuje, předcházet. Všechny tyto komponenty mají své jádro navrženo stejným způsobem a pouze k němu přidávají funkcionality specifické pro konkrétní komponentu. Návrh jádra komponent simulátoru ETCS se jako první začal implementovat v EVC v zimním semestru akademického roku 2022/2023. Zanedlouho následoval i přepis architektury RBC a JRU. Implementace RBC je předmětem současně vznikající bakalářské práce Ondřeje Veselého, na kterou se v následující sekci budu odkazovat, jelikož detailněji popisuje jádro společné architektury. [16]

Rozbor jádra architektury

Základem architektury jsou dva návrhové vzory:

Service Oriented Architecture (SOA): Rozděluje architekturu na menší logické celky v podobě služeb (angl. *services*). Jedná se o samostatné jednotky softwaru, které jsou navrženy pro plnění jednoho konkrétního úkolu. Jsou znovupoužitelné, na sobě nezávislé a dokáží spolu komunikovat. Princip tohoto návrhového vzoru podporuje rozšiřitelnost, modularitu, nízkou provázanost i vysokou soudržnost. [17]

Služby tvoří významnou část jádra architektury. Diagram znázorňující služby společné pro všechny komponenty simulátoru, které zmiňovanou architekturou disponují, lze nalézt na obrázku 2.13.



Obrázek 2.13 Diagram jádra architektury RBC, EVC a JRU

Event Driven Architecture (EDA): Umožňuje odděleným aplikacím asynchronně odesílat a reagovat na události (angl. *events*) pomocí tzv. *event brokeru*. [18] Každá událost se potom zpracovává příslušnou třídou označovanou jako *event handler*. Tento návrhový vzor podporuje zejména nízkou provázanost a rozšiřitelnost.

EDA se v případě komponent simulátoru ETCS využívá pro vzájemnou komunikaci. Namísto událostí se odesílají datové zprávy (angl. *messages*) prostřednictvím MQTT brokeru (popsaného v sekci 3.1.2) a jednotlivé komponenty je poté zpracovávají svými *message handlery*.

Vybrané třídy architektury

Na následujících řádcích bych ráda přiblížila pár nejdůležitějších tříd architektury. Pro podrobnější popis těchto i dalších tříd však odkazuji na bakalářskou práci Ondřeje Veselého. [16]

ServiceContainer: Neboli také kontejner služeb, je třída, která ukládá instance všech aktivních služeb. Při spuštění aplikace je inicializuje a při ukončení po nich zase uvolní paměť. Jeho hlavní rolí je však poskytování těchto služeb dalším částem aplikace.

Všechny jednotlivé služby mají definované rozhraní (angl. *interface*), které musí implementovat. Je to z toho důvodu, aby bylo možné provádět tzv. *dependency injection*. To je pojem označující programovací techniku, která pomocí rozhraní snižuje provázanost mezi na sobě závislými třídami. [19]

JRULoggerService: Slouží pro logování vnitřního stavu aplikace do logovací komponenty JRU.

ConfigurationService: Jedná se o službu, která vytváří, spravuje a získává konfigurační soubory. Funguje tak, že pokud požadovaná konfigurace nemá vytvořen svůj příslušný JSON soubor, bude pro ni vytvořen výchozí soubor definovaný v kódu. Pokud soubor vytvořen je, tak se z něj pouze načtou konfigurační soubory. Jinak řečeno, po prvním spuštění aplikace se vygenerují konfigurační soubory, pomocí kterých lze poté konfigurovat aplikaci bez nutnosti rekompilace.

HeartbeatService: Slouží k pravidelnému zasílání zpráv komponentě LPC, čímž jí aplikace dává najevo, že v pořádku běží. Tato služba pracuje na svém vlastním vláknu a funguje tak asynchronně vzhledem ke zbytku aplikace.

MessageHandlersService: Drží si všechny aktivní *message handlery*. Každý *handler* slouží ke zpracování právě jedné konkrétní zprávy, kterou aplikace obdrží.

MqttListenerService: Poslouchá na MQTT brokeru příchozí zprávy. Jakmile zpráva přijde, předá ji příslušnému *message handleru*. Tato služba běží na svém vlastním vláknu, aby mohla zprávy odbavovat ihned po jejich obdržení a funguje tak asynchronně vzhledem ke zbytku aplikace.

MqttPublisherService: Odesílá zprávy přes MQTT broker dalším komponentám.

2.5 Analýza komunikace DMI s ostatními komponentami systému

Posledním tématem k analýze je komunikace DMI s ostatními komponentami. Jelikož je mým cílem naimplementovat tuto komunikaci, je třeba důkladně zanalyzovat, jakým způsobem probíhá, se kterými komponentami se komunikuje a jaké zprávy se předávají.

DMI potřebuje komunikovat pouze se třemi komponentami. S LPC, který řídí simulaci, s JRU, kterému pouze zasílá zprávy k logování, a nakonec s EVC, které je jádrem ETCS.

2.5.1 LPC

LPC, jakožto hlavní řídící komponenta simulátoru, vydává DMI informace o stavu simulace. Může komponentám posílat i konfigurační zprávy, ale žádné takové v tuto chvíli pro DMI neexistují. Následuje popis všech zpráv, které jsou pro komunikaci LPC s DMI definovány.

Start Of Mission: Zpráva, kterou LPC předává DMI informaci o běhu simulace. Ta může být buď zahájena, zastavena anebo restartována.

Heartbeat: Tuto zprávu DMI odesílá v pravidelných intervalech. Dává pomocí ní LPC najevo, že v pořádku běží. Zpráva se začíná posílat zpravidla po zahájení simulace a přestává po ukončení simulace.

2.5.2 EVC

Po analýze zdrojových kódů a dotázání se vývojářů, kteří na ETCS simulátoru pracují déle než já, jsem zjistila, že pro komunikaci mezi DMI a EVC neexistuje žádná dokumentace. Na rozdíl třeba od komponenty RBC, která má komunikační zprávy pevně zadefinovány oficiálními *subsety* od Evropské unie. Komunikace mezi DMI a EVC byla tedy pro účely simulátoru definována pravděpodobně podle diagramů popisujících interakci strojvedoucího s displejem v oficiální dokumentaci ERA-ERTMS-015560 verze 2.3.0. [9] (kapitola 10.7)

Kdy a jaké zprávy si původní DMI s EVC vyměňuje jsem vypozorovala z logů vytvořených při simulaci jízdy vlaku a také ze zdrojových souborů obou komponent. Ačkoliv je mým cílem aktualizovat DMI na verzi 4.0.0, EVC je v tuto chvíli ve verzi 2.3.0. A protože mám také zajistit, aby DMI dokázalo komunikovat se stabilní verzí EVC, je třeba dodržet momentální podobu komunikace i přes to, že verze 4.0.0 zavádí do komunikace oproti minulé verzi rozdíly. Níže popisuji veškeré existující zprávy, které si DMI a EVC posílají a následně i průběh komunikace.

Příchozí zprávy od EVC

Cab Status: Zpráva, která předává informaci o tom, zda je kabina vlaku aktivní. Pokud ano, displej se rozsvítí, pokud ne, tak naopak zhasne. Zpráva může přijít kdykoliv během simulace.

Level And Position: Předává informaci o aktuální úrovni zabezpečení ETCS a pozici vlaku.

Emergency Braking: Informuje DMI o aktivaci či deaktivaci nouzových brzd. Odeslání této zprávy momentálně není v EVC naimplementováno, a tak ji ani DMI nemůže dostat. Mělo by se s ní ale počítat a naimplementovat v novém návrhu její přijetí.

Waiting for Emergency Stops Revocation: Předává informaci o tom, že ETCS čeká, dokud nebudou vyřízena všechna nouzová zastavení.

Emergency Stops Revoked: Tato zpráva informuje o tom, že došlo k odvolání všech nouzových zastavení a je možné znovu zahájit jízdu.

General Data: Zpráva, která se začne periodicky posílat po zahájení jízdy. Obsahuje všechna důležitá data pro vykreslování tachometru a průběhu jízdy.

Mode Proposed: Zašle se ve chvíli, kdy dojde ke změně operačního módu ETCS.

Outbound Train Data: Předává uložená data o vlaku.

RV Possible: Zašle informaci o možnosti přechodu do módu *Reversing* (RV) neboli couvání.

Session And Train Data: Informuje o stavu navázání spojení s RBC a přijetí dat o vlaku.

TAF Show: Předá informaci o nutnosti zobrazení dialogového okna volné trati před vlakem. Momentálně není její odeslání v EVC naimplementováno, ale mělo by se s ní počítat.

Waiting for SH response: Informuje DMI, že EVC čeká na odpověď od RBC na požadavek k přechodu do módu *Shunting* (SH) neboli manévrování, či posunu vlaku.

Odchozí zprávy do EVC

Driver ID: Zašle identifikátor strojvedoucího.

Level: Předá požadovanou úroveň zabezpečení ETCS.

Proposed Mode Ack: Používá se k potvrzení změny operačního módu ETCS.

Radio Network Data: Zašle data pro navázání spojení s RBC.

SH Selected: Informuje o stisknutí tlačítka *Shunting*.

Start Selected: Informuje o stisknutí tlačítka *Start*.

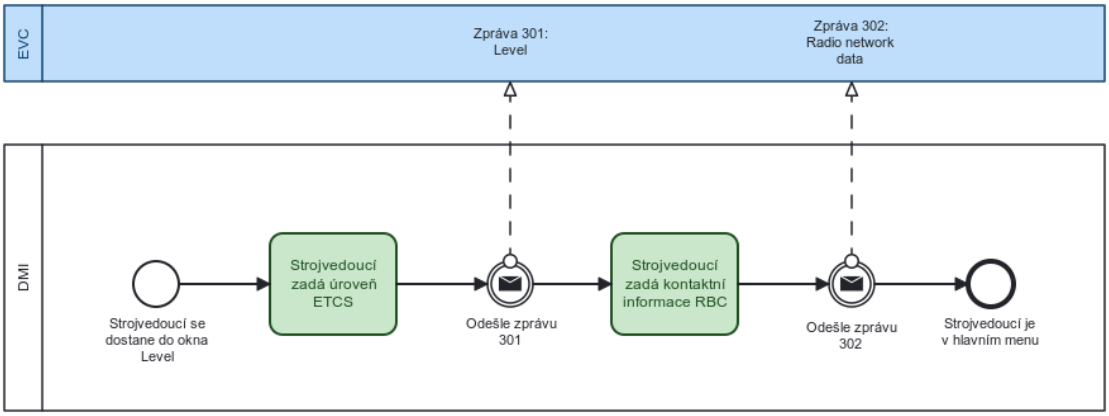
TAF Ack: Předá informaci o potvrzení dialogového okna volné trati před vlakem.

Train Data Entry Selected: Informuje o stisknutí tlačítka *Train Data*.

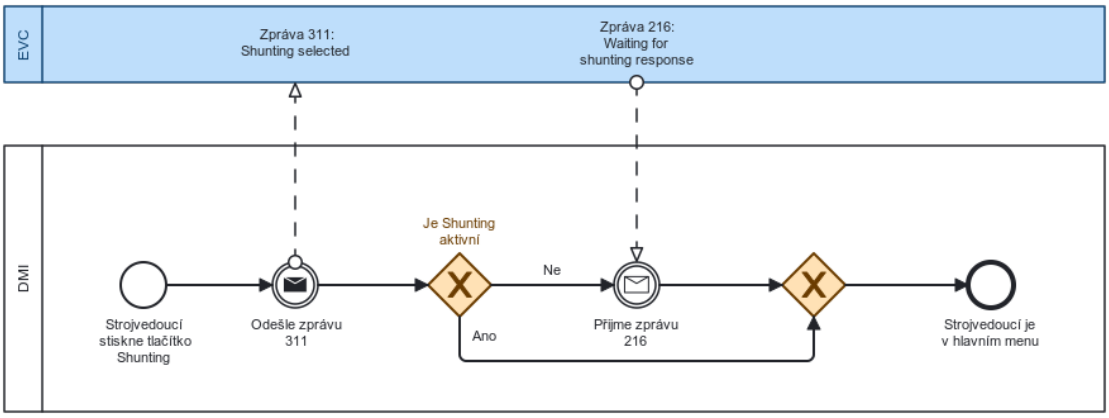
Train Data: Předá data o vlaku.

Průběh komunikace s EVC

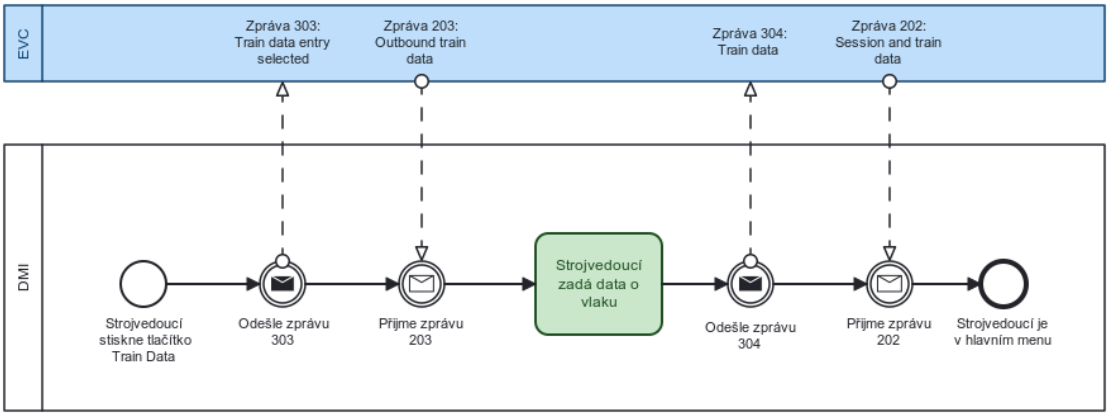
Na BPMN (Business Process Model and Notation) diagramech na obrázcích 2.14, 2.15, 2.16, 2.17 a 2.18 je znázorněn průběh komunikace s EVC tak, jak v tuto chvíli doopravdy probíhá.



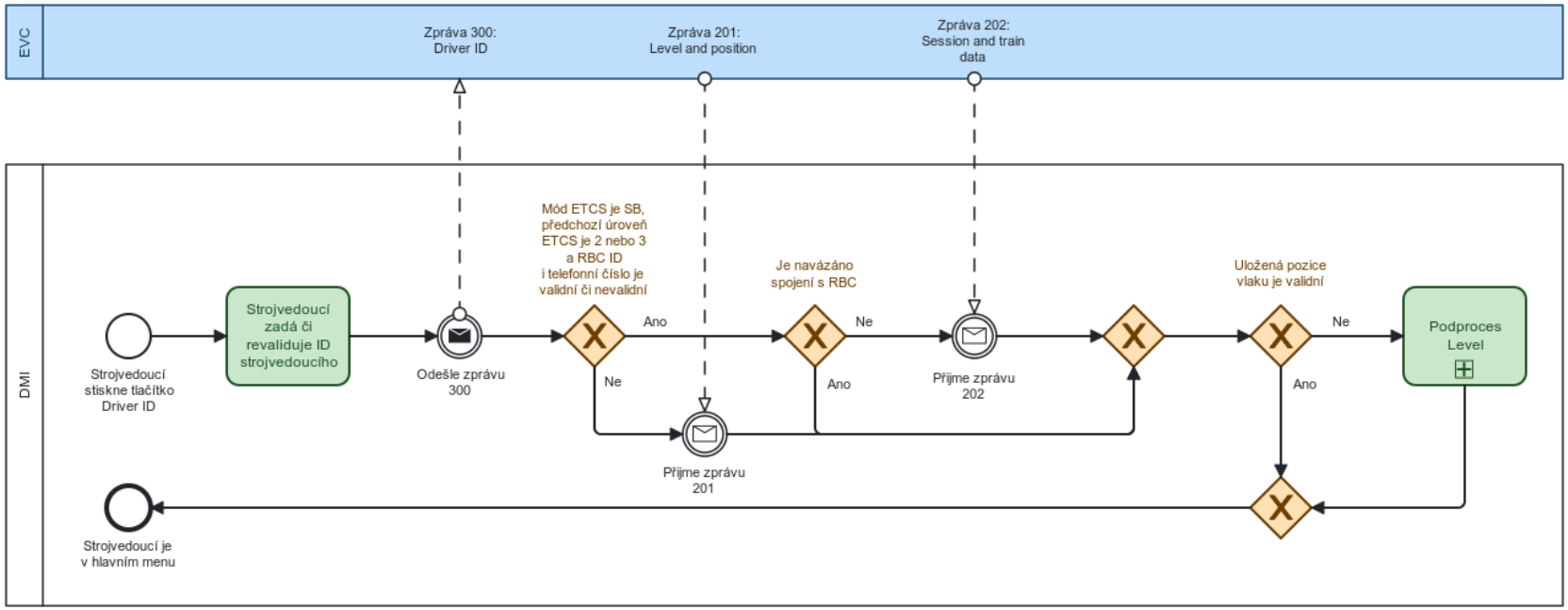
Obrázek 2.14 Průběh komunikace s EVC při zadávání úrovně ETCS



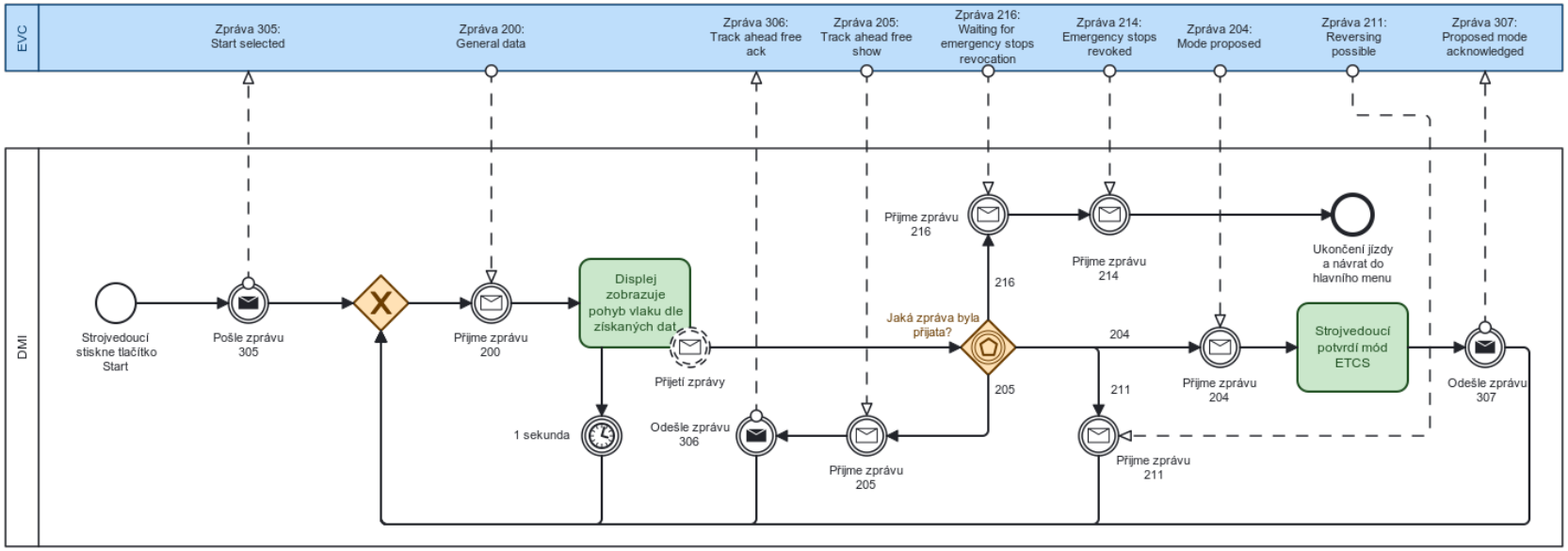
Obrázek 2.15 Průběh komunikace s EVC při stisknutí tlačítka Shunting



Obrázek 2.16 Průběh komunikace s EVC při zadávání informací o vlaku



Obrázek 2.17 Průběh komunikace s EVC při zadávání čísla strojvedoucího



Obrázek 2.18 Průběh komunikace s EVC po zahájení jízdy vlaku

2.6 Požadavky na novou architekturu DMI

Nyní bych chtěla shrnout všechny získané poznatky z analýzy a definovat požadavky na návrh. Jelikož se má práce soustředí zejména na návrh a přepis architektury a na vznik nového grafického *frameworku* pro DMI, a ne na funkcionality aplikace jako takové, uvedu požadavky jako celek a nebudu je rozdělovat na funkční na nefunkční. Definice všech požadavků je k nahlédnutí v tabulce 2.6.

Požadavek	Popis
[P1] Kompletní přepis	Z analýzy jasně plyne, že stávající DMI je ve velmi špatném stavu a bude potřeba jej kompletně přepsat.
[P2] Rozšiřitelnost a udržitelnost	Nová architektura by měla být maximálně rozšiřitelná a udržitelná do budoucna.
[P3] Nízká provázanost a vysoká soudržnost	Je třeba, aby jednotlivé třídy návrhu byly co nejméně provázané, ale naopak maximálně soudržné. Tento požadavek dále podporuje rozšiřitelnost aplikace.
[P4] Nastavení vhodné modularity	Ve stávajícím DMI je modularita velmi nízká a bere si tak vysokou daň na rozšiřitelnosti aplikace. Příliš vysoká modularita však také může ztížit vývoj. Proto je třeba vhodně zvolit, jak rozsáhlé budou jednotlivé třídy návrhu.
[P5] Specifikační verze 4.0.0	Po domluvě se zákazníkem projektu, panem doc. Ing. Martinem Lesem, Ph.D., bude třeba návrh směřovat na verzi ETCS 4.0.0, která je pro DMI definována v oficiální dokumentaci ERA. [3]
[P6] Konzistence s architekturou ostatních komponent	Z důvodu usnadnění vývoje by architektura měla být co možná nejpodobnější architektuře ostatních komponent simulátoru.
[P7] Návrh grafického *frameworku*	V analýze architektury stávajícího DMI jsem zjistila, že využívaná grafická knihovna je jedním z jejích největších problémů. Jedná se pouze o sadu nekvalitně navržených funkcí, která se velmi obtížně využívá v kódu. Můj nový návrh by tedy měl zavést objektově orientovaný a snadno použitelný grafický *framework*, ze kterého se bude stavět UI aplikace.
[P8] Více modelů displeje	Z analýzy bakalářské práce Ondřeje Měšťana v kapitole 2.2.3 plyne, že návrh by měl pamatovat na možnost vícero různých modelů displeje.
[P9] Překlady	Z analýzy bakalářské práce Yuryho Udavichenky v kapitole 2.2.2 plyne, že je třeba v návrhu vymyslet způsob překládání všech textových řetězců vykreslovaných v grafickém rozhraní aplikace. Mělo by být snadné přidávat i další překlady.
[P10] Komunikace s ostatními komponentami	Aplikace musí být schopna komunikovat s ostatními komponentami simulátoru ETCS, podle analýzy v kapitole 2.5. Zároveň je pro účely této komunikace třeba navrhnout způsob ukládání komunikačních dat.

Tabulka 2.6 Tabulka požadavků na novou architekturu DMI

Kapitola 3
Návrh

Z předešlé analýzy jasně vyplývá, že bude nutné původní DMI kompletně přepsat. Jak z důvodu neudržitelnosti kódu, tak z důvodu zavedení konzistence s ostatními existujícími komponentami. V této kapitole bych chtěla přiblížit svůj návrh nové architektury DMI, jaké technologie a návrhové vzory jsem využila a jak budu řešit problémy identifikované v předešlé analýze.

3.1 Zvolené technologie

Než budu pokračovat k samotnému návrhu architektury, je třeba popsat, jaké technologie v něm budu využívat.

3.1.1 Jazyk C++

Valná většina komponent simulátoru ETCS je napsána v jazyce C++. Konkrétně EVC, RBC, JRU, CEM i původní DMI. Dává tedy jedině smysl tento programovací jazyk využít i pro můj návrh a zbytečně neztěžovat vývoj simulátoru tím, že by vývojáři museli využívat více programovacích jazyků. Dalším důvodem zachování tohoto jazyka je i komponenta CEM, která definuje komunikační zprávy mezi komponentami a která je navržená jako knihovna pro C++ aplikace. Pro implementaci svého návrhu tedy využiji jazyk C++20.

3.1.2 Komunikační protokol MQTT

Všechny komponenty simulátoru ETCS využívají pro vzájemnou komunikaci protokol MQTT. Jedná se o jednoduchý a nenáročný protokol, který prostřednictvím centrálního bodu (tzv. *MQTT brokeru*) umožňuje předávání zpráv mezi jednotlivými klienty. Přenos probíhá pomocí protokolu TCP a samotná výměna zpráv funguje na principu *publish – subscribe*. Tento princip rozděluje zprávy do tzv. témat (angl. *topics*). Každý klient má potom možnost publikovat zprávy v daném tématu (*publish*), tedy zaslat je *brokeru*, který je následně distribuuje dalším klientům, anebo se přihlásit k odběru určitých témat (*subscribe*) a přijímat tak od *brokeru* všechny jeho zprávy. [20]

Komponenty projektu využívají pro realizaci komunikace Eclipse Mosquitto MQTT Broker, který pro účely zachování konzistence a jednoduchosti ve svém návrhu také využiji.

3.1.3 Knihovna JSON for Modern C++

DMI bude hojně pracovat se soubory typu JSON. Ať už kvůli komunikaci, při které se data předávají právě v tomto formátu anebo kvůli konfiguračním souborům. S knihovnou JSON for Modern C++ se snadno pracuje a využívají ji všechny ostatní komponenty projektu, proto ji pro manipulaci s JSON soubory také využiji.

3.1.4 Grafická knihovna SFML

V původním DMI byla grafická část aplikace postavena na knihovně SDL2. Jedná se sice o mocnou knihovnu, která dává programátorovi hodně kontroly nad generováním a vykreslováním grafických objektů, je však velmi nízkoúrovňová a obtížná na naučení. Doba, za kterou se programátor dokáže naučit s knihovnou pracovat, je v projektu simulátoru ETCS klíčová. Každý akademický rok totiž přichází noví studenti, kteří se na vývoji podílí. Někteří u vývoje projektu setrvají pouhý jeden semestr. Je tedy důležité, aby nestrávili mnoho času na pochopení práce s knihovnou a mohli rychle zahájit vývoj. Tento požadavek knihovna SDL2 hrubě porušuje. Práce s ní je pro neznalého obtížná a zdlouhavá. Navíc je její oficiální dokumentace špatně přehledná a nedá se v ní jednoduše vyhledávat. [21]

Z těchto důvodů jsem pro grafickou část aplikace zvolila knihovnu SFML (Simple and Fast Multimedia Library). Tato knihovna je napsána v jazyce C++ a poskytuje vysokoúrovňové objektově orientované rozhraní pro práci s grafikou i audiem. Krásně tak zapadne do mého objektově orientovaného návrhu nové architektury. Knihovna je zároveň oproti SDL2 velmi snadná na použití, její jednotlivé objekty jsou dobře zapouzdřeny a poskytuje přehlednou a informativní oficiální dokumentaci. SFML lze sice využívat pouze pro vývoj 2D grafiky, to je však pro účely DMI naprosto dostačující. Podporuje také vývoj pro více operačních systémů, a to jak pro Linux, Windows, tak i macOS. [22]

3.2 Jádro architektury

Nyní se dostávám k samotnému návrhu. Z důvodu již několikrát skloňovaného zachování konzistence mezi komponentami postavím jádro architektury stejně, jako je nyní například v RBC a EVC. Jeho princip jsem již popsala v kapitole 2.4.2. Kód jádra DMI bude až na malé odchylky totožný s kódem jádra ostatních komponent. Asi nejvýraznějším rozdílem bude, že jádro DMI nebude využívat službu `ConsoleInputService`, která je jinak v ostatních komponentách přítomna. Důvodem je, že DMI nemá být konzolová aplikace, ale grafická. Další důvod k této změně je implementační. Pokud by totiž aplikace neustále poslouchala na vstupu konzole, nebylo by možné ji jednoduše vypnout zavřením grafického okna displeje. V C++ je čtení vstupu takzvaně blokující, což znamená, že aplikace či její vlákno, které přijímá vstup, nebude pokračovat ve svém chodu, dokud nějaký vstup nedostane. Při pokusu o zavření okna DMI by se tedy aplikace neukončila, protože by stále čekala na vstup z konzole. Implementace neblokujícího vstupu v C++ je poměrně složitá záležitost a není předmětem mé práce.

3.3 Správa komunikačních dat

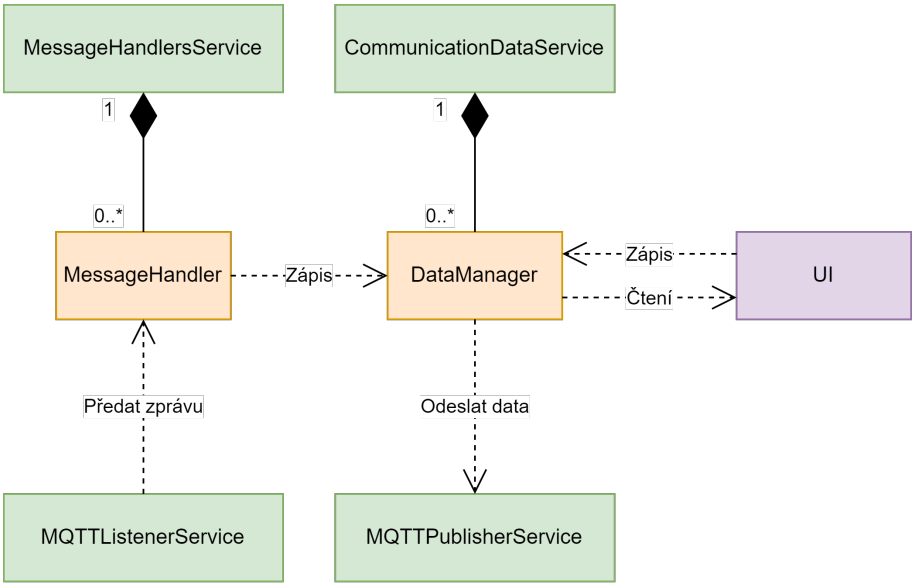
Důležitou součástí architektury je komunikace s ostatními komponentami, jejíž analýzu jsem provedla v kapitole 2.5. Přijímání a odesílání zpráv přes protokol MQTT je již vyřešeno ve společném jádru architektury komponent simulátoru, a tudíž jej zde nebudu podrobně popisovat (služby zajišťující komunikaci jsou popsány v kapitole 2.4.2). Co je ale třeba vyřešit, je ukládání a správa komunikačních dat. Přijatá data od ostatních komponent je z velké většiny nutné někam uložit, aby se podle nich později mohlo vykreslit grafické rozhraní. Z tohoto důvodu do návrhu zavádím službu `CommunicationDataService` a datové kontejnery.

3.3.1 CommunicationDataService

Tato služba bude mít za úkol spravovat jednotlivé kontejnery s komunikačními daty. Při inicializaci se do ní všechny dostupné kontejnery zaregistrují a ona je pak za běhu bude moct poskytovat jiným částem aplikace. Tímto způsobem bude od každého kontejneru existovat pouze jedna sdílená instance a nebude docházet ke zbytečnému kopírování dat.

3.3.2 DataManager

`DataManager` bude představovat jeden datový kontejner ukládající komunikační data. Z něj potom bude moct UI načítat potřebná data pro vykreslování. Způsob, jakým budou kontejnery pracovat se zbytkem aplikace je znázorněn na obrázku 3.1.



Obrázek 3.1 Diagram správy komunikačních dat

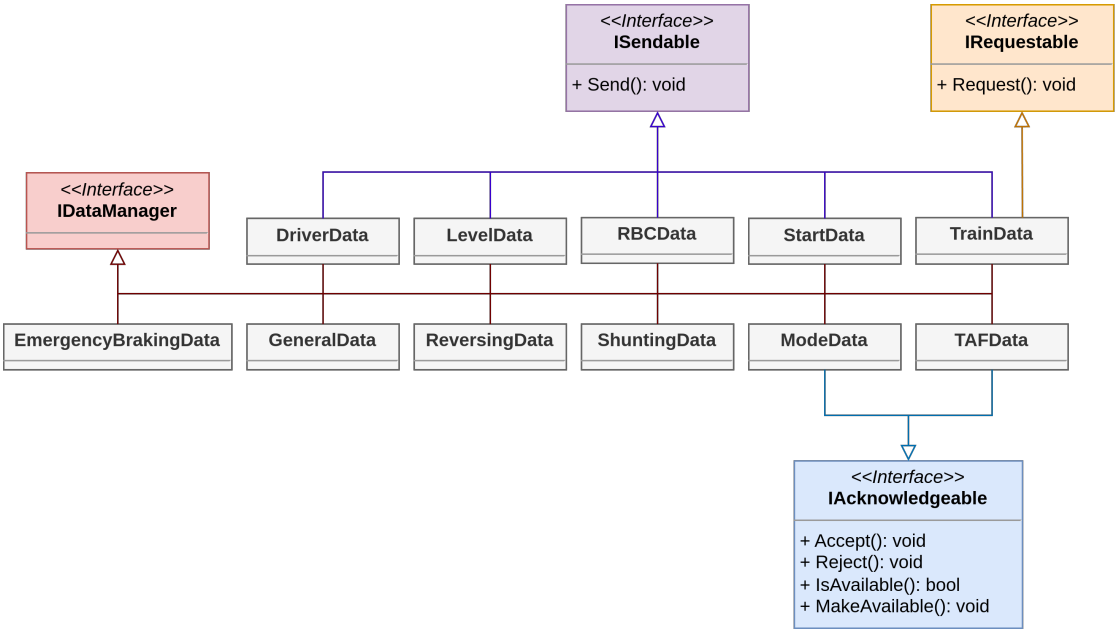
Každá zpráva, která lze od ostatních komponent přijmout a jejíž obsah je třeba uložit, bude mít pro jednoduchost svůj vlastní kontejner, do kterého se budou komunikační data po složkách ukládat. Data v kontejneru musí být asynchronní položky, protože se předpokládá, že k nim bude přistupovat více vláken aplikace – konkrétně vlákno, na kterém poběží příslušný *message handler* a vlákno UI. Všechny kontejnery, které v rámci mé práce vzniknou, jsou na obrázku 3.2. Každý `DataManager` bude také moct implementovat různá rozhraní, pomocí kterých bude manipulovat s daty. Pro současnou podobu komunikace budou stačit tato čtyři rozhraní:

IDataManager: Základní rozhraní každého kontejneru, které bude nabízet společné metody.

ISendable: Implementací tohoto rozhraní bude datový kontejner moct odesílat svá data jiné komponentě simulátoru.

IRequestable: Umožní kontejneru vyžádat si data od jiné komponenty. Pošle dané komponentě zprávu žádající o data a ta je zašle jako odpověď, kterou zpracuje *message handler* a uloží do daného kontejneru.

IAcknowledgeable: Toto rozhraní dá kontejneru možnost odeslat zprávu o přijetí či zamítnutí dat (angl. *acknowledge* nebo také *ack*).



Obrázek 3.2 Diagram všech kontejnerů a jejich rozhraní, které vzniknou v rámci implementační části práce

3.4 Grafická část aplikace

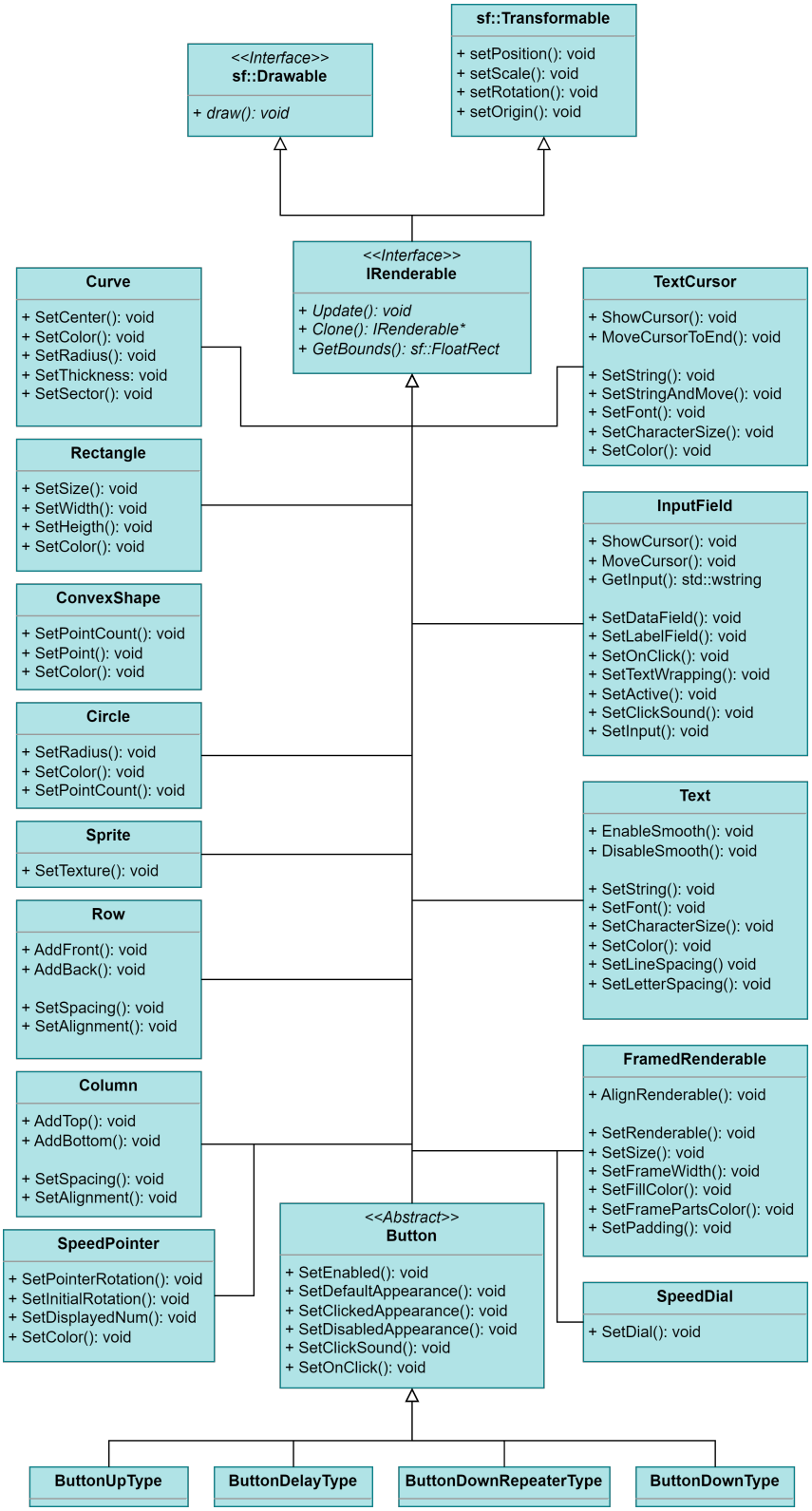
Nejdůležitější částí DMI je jeho displej. Právě grafická část aplikace byla v původní architektuře navržena velmi nerozšiřitelně a neudržitelně. V následujících kapitolách tedy představím svůj udržitelnější návrh.

3.4.1 Renderovatelné objekty

Hlavní idea návrhu je rozdělit grafiku na malé, vysoce přizpůsobitelné a generické objekty, které se budou skládat do sebe a tvořit tak větší celky. Proto vzniknou tzv. *renderables*, tedy renderovatelné objekty.

Všechny tyto objekty budou implementovat společné rozhraní `IRenderable`, které bude dědit z tříd `sf::Drawable` a `sf::Transformable`. Obě zmíněné třídy jsou součástí knihovny SFML. `sf::Drawable` poskytuje objektu možnost ho vykreslit na obrazovku a `sf::Transformable` zas grafickému objektu dává základní metody ke změně jeho pozice, počátku, rotace a škálování. Pomocí této struktury bude možné renderovatelné objekty snadno generalizovat a využívat polymorfismus.

Pro lepší představu přikládám třídní diagram na obrázku 3.3, který znázorňuje renderovatelné objekty, které budu implementovat. Uvádím zde pouze nejdůležitější veřejné metody bez jejich parametrů pro lepší přehlednost. Podrobnější popis fungování těchto objektů vysvětlím v kapitole 4.



Obrázek 3.3 Třídní diagram renderovatelných objektů, které vzniknou v rámci implementační části práce

3.4.2 Návrh vybraných renderovatelných objektů

Nyní bych ráda přiblížila návrh nejdůležitějších renderovatelných objektů. Cílem jejich návrhu je poskytnout solidní základ, ze kterého půjde snadno skládat jednotlivé části UI.

Row a Column

Tyto dvě třídy představují řádek (objekt `Row`) a sloupec (objekt `Column`). Budou sloužit zejména ke snazšímu rozložení jednotlivých *renderables* na obrazovce. Bude možné na jejich začátek či konec polymorfně přidat libovolný objekt implementující `IRenderable`. Tedy bude možné například do prvního pole sloupce přidat kruh, do druhého tlačítko a do třetího další sloupec. Zde se ukazuje dříve zmíněná idea skládání renderovatelných objektů do sebe.

Řádek i sloupec dynamicky přizpůsobí velikost polí rozměrům daného renderovatelného objektu v konkrétním poli (tedy při změně velikosti *renderable* v řádku či sloupci se změní i velikost pole a tím pádem budou vždy všechny položky v řádku či sloupci korektně rozloženy a nebudou se překrývat). Zároveň bude možné nastavit zarovnání renderovatelných objektů v polích na střed, doleva, doprava, nahoru či dolů. V neposlední řadě půjde nastavit i velikost mezer mezi jednotlivými poli objektu.

FramedRenderable

Tento renderovatelný objekt, česky ho nazvu rámeček, vzniká ze specifické potřeby DMI. Když si v dokumentaci prohlédnu, jak má UI aplikace vypadat, zjistím, že se skoro celé skládá z rámečků, které často mají uvnitř sebe vycentrovaný jiný renderovatelný objekt. Pro lepší pochopení přikládám obrázek 3.4 z oficiální dokumentace. Je zde zřetelně vidět, co tím myslím. Tachometr DMI se skládá z mnoha rámečků, které za určité situace mohou obsahovat jiný objekt. Tlačítka klávesnice napravo nejsou nic jiného než dva rámečky v sobě s textem uvnitř, stejně tak lišta s názvem okna i vstupní pole jsou pouhé rámečky s textem, který je zarovnaný doleva.

`FramedRenderable` bude ve výchozím stavu zcela průhledný, ale bude mít možnost nastavit barvu výplně a jednotlivých částí okraje. Také bude možné polymorfně nastavit vnitřní renderovatelný objekt a zarovnat ho na střed, doleva či doprava. Půjde nastavit i jeho odsazení zprava a zleva.

InputField

Vstupní pole je velmi důležitou součástí několika menu obrazovek DMI. Po analýze oficiální dokumentace ERA jsem zjistila, že tento objekt má několik různých podob (jednu z nich lze vidět na obrázku 3.4). Vždy je možné do tohoto pole zadávat vstup z klávesnice aplikace. Co se však liší, je, že někdy se zadává po znacích, někdy po celých textových řetězcích (které se vždy přepíší na rozdíl od zadávání znaků), někdy ve vstupním poli bliká kurzor, jindy ne a nakonec, někdy se objekt skládá pouze ze vstupní části a někdy má i štítek popisující dané pole. [3] Proto bude nutné objektu vstupního pole poskytnout několik funkcí pro jeho přizpůsobení. Nastavení řetězce uvnitř vstupního pole deleguji vně objektu.

Tlačítka

Tlačítka jsou naprosto klíčovou součástí každého uživatelského rozhraní, DMI nevyjímaje. Přestože v aplikaci mají tlačítka vždy vzhled rámečku s dalším renderovatelným objektem uvnitř, chci je navrhnout co nejvíce přizpůsobitelné a generické. Proto v mém návrhu bude vzhled klidového, stisknutého i vypnutého stavu tlačítka možné polymorfně nastavit na libovolný renderovatelný objekt.

Uživatel potřebuje s tlačítkem interagovat, ať už myší anebo dotekem. Tlačítko může reagovat různými způsoby popsanými v oficiální dokumentaci ERA [3] (kapitola 5.3.2.6). Bude tedy potřeba je navrhnout polymorfně (jak lze vidět na obrázku 3.3). Dokumentace rozlišuje čtyři typy tlačítek – tzv. *up-type*, *down-type*, *up-type* s opakovačem a *delay-type* (při analýze jsem zjistila, že tlačítka v původním DMI se často nechovala korektně). Jejich popis uvádím níže. V návrhu je ale třeba počítat i s tím, že do budoucna mohou přibýt nové typy.

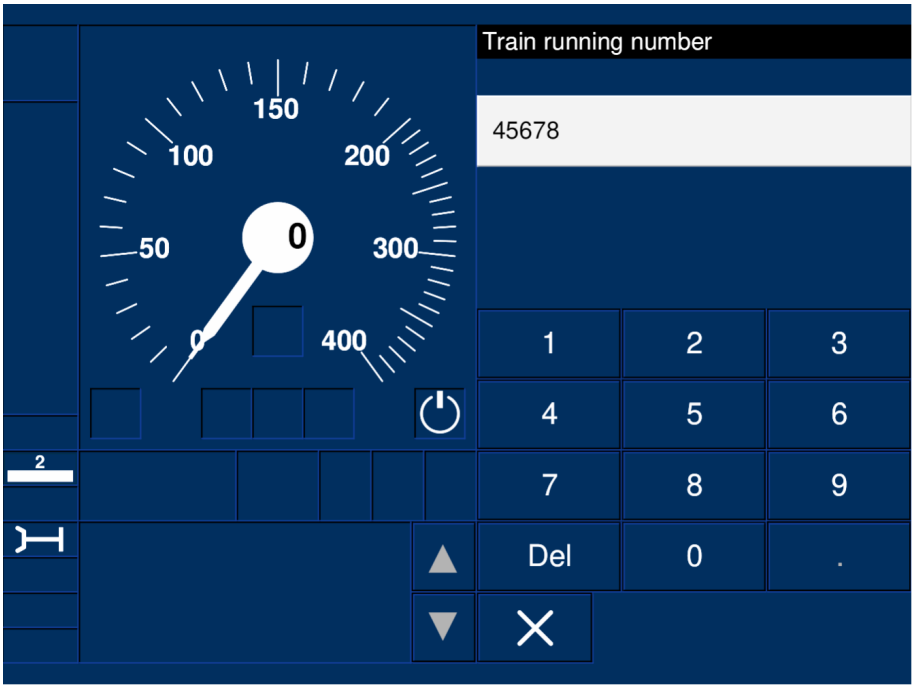
ButtonUpType: Po kliknutí zůstává stisknuté po celou dobu stisku. Při uvolnění stisku dojde k vykonání akce přiřazené tlačítku. Pokud by uživatel uhnul myší z tlačítka za stálého stisku a poté stisk uvolnil, akce se nevykoná. Pokud by však vrátil myš zpět na tlačítko a až tehdy stisk uvolnil, akce se provede.

ButtonDownType: Po kliknutí se s malou odezvou vrátí hned do původního nestisknutého stavu, i pokud uživatel nadále drží stisk a provede se akce.

ButtonDownRepeaterType: Funguje stejně jako `ButtonDownType`, s tím rozdílem, že po 1,5 s nepřetržitého držení tlačítka začne opakovaně po 0,3 s intervalech mačkat tlačítko.

ButtonDelayType: Funguje podobně jako `ButtonUpType`, s tím rozdílem, že po stisku nejprve po dobu 2 s po intervalech 0,25 s problikává ze stisknutého stavu do nestisknutého. Pokud uživatel v průběhu uvolní stisk, akce se neprovede a musí stisknutí opakovat znovu. Jestli však 2 s uplynou, tlačítko se začne chovat stejně jako `ButtonUpType`.

Důležitým aspektem každého tlačítka je akce, která se provede při jeho stisku. Proto půjde v mém návrhu každému tlačítku předat funkci, která se zavolá při jeho stisku. To, jakým způsobem se tlačítko při interakci s uživatelem bude chovat, je jasně definováno v oficiální dokumentaci ERA [3] (kapitola 5.3.2.6). Konkrétně se zde rozlišují čtyři různé typy tlačítek. Jejich implementační rozdíly popíší v kapitole 4.1.4.



Obrázek 3.4 Ukázka množství rámečků v uživatelském rozhraní aplikace [3]

3.4.3 MVC

Kromě renderovatelných objektů využiji pro grafické rozhraní návrhový vzor MVC, který byl taktéž využit v původním DMI. Tento návrhový vzor slouží k oddělení dat (*model*) od grafického rozhraní (*view*) a logiky (*controller*). Spolu s MVC využiji i vzor *Observer*. Ten funguje na principu toho, že existuje nějaký objekt *observable*, který mohou sledovat objekty označené jako *observery*. *Observable* si zaregistruje své *observery* a v případě nutnosti je může všechny notifikovat, typicky na základě nějaké události, jako je třeba změna dat. *Observery* po notifikaci provedou nějakou akci závislou na změně *observable*. [23] Tento vzor ve svém návrhu využiji pro aktualizaci *views* při změně modelu. Důvodem je, že *views* touto cestou nemusí neustále kontrolovat data v modelech a také se tím bude předcházet zbytečnému znovu generování grafických objektů, které by byly totožné s těmi předchozími.

Model: Datový model budou představovat již zmíněné *data managery*. Navíc jim přidám dědičnost ze třídy `Observable`, která jim umožní si zaregistrovat *observery* a notifikovat je o své změně v případě potřeby.

View: V mém návrhu bude *view* také jedním z renderovatelných objektů. Tedy bude implementovat rozhraní `IRenderable`. Bude složeno z menších renderovatelných objektů (případně i přímo z *views*). Hlavním rozdílem oproti *renderables* je to, že view je komplexnější objekt s mnoha renderovatelnými podobjekty, které již mají jasně definované nastavení a může na rozdíl od *renderables* využívat `ServiceContainer` a jím poskytované služby.

View bude zároveň implementovat rozhraní `IObserver`, pomocí nějž se bude moct zapsat ke sledování určitých *observables*. Každé *view* by mělo být maximálně generické, aby bylo možné ho znovu využít vícero *controllery* (například pro menu DMI, jež má mnoho různých obrazovek, které jsou si ale vzhledem vysoce podobné – tj. stačilo by implementovat jedno view pro daný typ menu obrazovky, např. tlačítkové menu).

Controller: Inicializuje si všechna potřebná *views* a nastaví je pro své účely (například nastaví tlačítkům akce, které se po jejich stisku provedou). Modely inicializovat nemusí, protože to zajišťuje `CommunicationDataService`. Každý *controller* také bude mít aktualizační metodu, ve které bude probíhat hlavní logika UI.

3.5 Služby

Nyní popíši svůj návrh nových služeb, které nebyly součástí architektury společného jádra komponent. Tyto služby jsou pro správný chod DMI klíčové.

3.5.1 RendererService

Služba `RendererService` bude obsluhovat samotné vykreslování grafiky. Bude spravovat instanci okna aplikace, s čímž souvisí jeho inicializace, zachytávání událostí způsobených interakcí uživatele (např. zavření anebo změna velikosti okna) a také vykreslování grafických objektů. Bude si držet všechny renderovatelné objekty, které je třeba v daném snímku vykreslit, ty pro každý snímek aktualizuje a následně vyrenderuje na obrazovku.

3.5.2 AssetStorageService

Tato služba bude spravovat a ukládat všechny grafické a audio *assety* – tedy textury, fonty a zvuky. Vznikne z toho důvodu, aby od každého zdroje existovala právě jedna sdílená instance a nedocházelo tak ke zbytečnému kopírování. Při inicializaci si služba načte všechny specifikované zdroje a následně je bude poskytovat dalším částem aplikace.

3.5.3 RenderableFactoryService

Služba `RenderableFactoryService` vznikne pro generování často využívaných renderovatelných objektů. Některé objekty se ve stejné podobě a nastavení objevují napříč celou aplikací (např. tlačítka, rámečky atd.) a proto je vhodné nastavit je na jednom místě, aby se kód neopakoval. V případě potřeby takového objektu se pouze zavolá příslušná metoda této služby na jeho vygenerování.

3.5.4 UIManagerService

`UIManagerService` zastřeší celé UI. Bude držet hlavní aplikační smyčku, ve které aktualizuje registrované *controllery* a zavolá *renderer*. Všechny *controllery* si manažer inicializuje při startu aplikace. Každý z nich potom bude možné buď zaregistrovat anebo odregistrovat z manažeru. K těmto akcím bude typicky docházet při nutnosti změny obrazovky UI, například po stisknutí tlačítka.

3.5.5 TranslatorService

Tato služba bude mít za úkol překládat texty v UI. Bude pro ni existovat výčtový typ `TextType`, který bude označovat konkrétní text. Služba si při inicializaci načte všechny dostupné překlady z konfigurací a přiřadí ke každému `TextType` jeho překlady. Ve chvíli, kdy někde v UI bude potřeba vykreslovat přeložitelný text, poskytne se ukazatel na tento text spolu s jeho `TextType` službě `TranslatorService`. Jakmile služba obdrží příkaz pro přeložení na zadaný jazyk, přepíše všechny zaregistrované texty podle jejich `TextType`. Ve výchozím nastavení bude vypisovat texty anglicky.

3.5.6 Konfigurace

Ačkoliv konfigurační služba `ConfigurationService` popsaná v kapitole 2.4.2 je součástí architektury společného jádra komponent, je dle mého názoru vhodné ji přesto zmínit, pro svou velkou důležitost. Při implementaci bude třeba brát důraz na to, aby byla aplikace v co nejvyšší míře konfigurovatelná. Například rozložení a rozměry objektů ve *view*, velikost okna aplikace anebo umístění zdrojů jako jsou obrázky a fonty je velmi nežádoucí mít natvrdo definované v kódu. I z toho důvodu, že může být více různých modelů displeje, jak jsem zjistila v analýze bakalářských prací. Níže ve výpisu 3.1 uvádím návrh struktury konfigurací.

```
Configurations ..................................................Obecné konfigurace
└─UI ..................................... Společné konfigurace pro všechny modely DMI
  ├─Assets ..............................Konfigurace umístění obrázků, fontů a zvuků
  └─DisplayModels ..................................Konfigurace konkrétních modelů
    └─ERA_ERTMS_TouchScreen .........Konfigurace pro dotykový model ERA ERTMS
      ├─Areas ....................... Konfigurace pro sekce výchozí obrazovky DMI
      └─Views .....................Konfigurace pro ostatní, jako jsou obrazovky menu
```

Výpis kódu 3.1 Návrh struktury konfigurací

3.6 Dokumentace

Dokumentování svého kódu je naprosto klíčové pro usnadnění budoucího vývoje aplikace. Ve své implementaci využiji dokumentační nástroj Doxygen, který umožňuje z kódu okomentovaným jeho notací vygenerovat HTML anebo LaTeX dokumentaci. Tento nástroj se hojně využívá při programování v jazyku C++. [24]. Notace byla zvolena pro všechny komponenty simulátoru ETCS, a tak jsem ji pro svou implementaci zvolila taktéž.

Každý hlavičkový soubor by měl mít okomentované všechny své metody (vyjma těch triviálních jako jsou *setter* nebo *getter*). Zároveň všechny soubory s kódem, ať už hlavičkové anebo zdrojové, budou mít na svých prvních pár řádcích dokumentační hlavičku, jejíž podoba byla dohodnuta pro všechny komponenty simulátoru. Bude zde uvedeno následující:

1. Název souboru
2. Komponenta simulátoru, které je soubor součástí
3. Specifikační verze *subsetu* (dokumentace), na jehož základě soubor vznikl
4. Krátký popis souboru
5. Vývojáři, kteří se na tvorbě souboru významně podíleli

Kapitola 4
Implementace

Nyní bych se chtěla přesunout k implementaci svého návrhu. Jelikož je zdrojový kód praktické části mé práce velmi rozsáhlý, popíši zde pouze vybrané třídy a objekty, které považuji za vhodné zmínit. Nebudu zde vůbec popisovat implementaci jádra aplikace. Tuto část implementace jsem pouze převzala z ostatních komponent simulátoru a pro její podrobnější popis odkazuji na bakalářskou práci Ondřeje Veselého. To zahrnuje například implementaci kontejneru služeb, komunikace přes MQTT, *message handlerů*, konfigurací atd. [16] Zároveň bych ráda avizovala, že uvedené ukázky kódu jsou pro účel tohoto textu často zkrácené, upravené či zčásti vypuštěné pro lepší čitelnost. Zbytek implementace je ve svém plném rozsahu součástí přílohy spolu s instalační A i vývojářskou příručkou B. Přiložené obrázky vykresleného UI mají barvy upravené tak, aby měly dostatečný kontrast pro účely tisku této práce.

4.1 Grafická část aplikace

Nejprve bych chtěla objasnit implementaci renderovatelných objektů, *views* a *controllerů*.

4.1.1 IRenderable

Abstraktní třídu `IRenderable` implementuje každá třída, která představuje grafický objekt a má být možné ji vykreslit na obrazovku. Vykreslovací metodu dědí z třídy `sf::Drawable` a je nutné ji implementovat pro každý renderovatelný objekt zvlášť. Typicky v ní dojde k aplikování transformací objektu na obraz a jeho vykreslení. Tuto metodu potom volá *renderer*. Příklad její implementace je na ukázce 4.1. Transformační metody poděděné z třídy `sf::Transformable` však nejsou virtuální a není je tedy třeba implementovat.

```
1 void Row::draw(sf::RenderTarget& target, sf::RenderStates states) const {
2     states.transform *= getTransform();
3
4     for (auto& renderable : row)
5         target.draw(*renderable, states);
6 }
```

Výpis kódu 4.1 Ukázka implementace vykreslovací metody ze souboru `Row.cpp`

V ukázce kódu 4.2 je vidět metoda `Update`, která přijímá jako parametr pozici myši. V této metodě bude typicky docházet k aktualizaci renderovatelného objektu. Některé objekty ji nemusí

implementovat a bude se tak volat výchozí metoda s prázdným tělem. Typicky ji implementují objekty, které potřebují reagovat na pozici myši (jako třeba tlačítka) a objekty, které se skládají z vícero dalších renderovatelných objektů. Například třída `Row` (řada), která musí aktualizovat každý objekt, který vykresluje (protože drží polymorfní objekty a neví, zda je třeba je aktualizovat či nikoliv).

Dále se zde nachází metody `GetBounds` a `Clone`, které je nutné implementovat v každém renderovatelném objektu. `GetBounds` slouží pro získání pozice a rozměrů daného objektu. Metoda `Clone` vznikla z potřeby snadno kopírovat renderovatelné objekty. Pokud chci stejný objekt vykreslovat vícekrát s různými vlastnostmi (například odlišné transformace), je třeba jej naklonovat, aby se objekty vykreslovaly správně.

```
class IRenderable : public sf::Drawable, public sf::Transformable {
public:
    virtual void Update(const sf::Vector2f& mousePos);
    virtual sf::FloatRect GetBounds() const = 0;
    virtual IRenderable* Clone() const = 0;

protected:
    void ApplyTransformations(const IRenderable* source);
    sf::Vector2f GetTransformedMousePos(const sf::Vector2f mousePos);
};
```

Výpis kódu 4.2 Ukázka části hlavičkového souboru třídy `IRenderable` ze souboru `IRenderable.hpp`

Nakonec bych chtěla objasnit existenci *protected* metod. `ApplyTransformations` se využívá jako pomocná metoda v metodě `Clone`. Aplikuje transformace zdroje na jiný objekt a tím pádem umožní vytvořit kopii i s transformacemi zdroje. Ukázka jejího využití je ve výpisu kódu 4.3. Metoda `GetTransformedMousePos` se zase využívá v metodě `Update` a to vždy, když je třeba aktualizovat dceřiný renderovatelný objekt dané třídy. Tato metoda opět souvisí s transformacemi. Pokud totiž aplikuji transformace z třídy `sf::Transformable` na rodičovský objekt, budou se vztahovat pouze na tento objekt jako celek a jednotlivé dceřiné objekty se o nich nedozví. Proto pokud by dceřiné objekty ve své aktualizační funkci využívaly parametr předávající pozici myši, bude potřeba tuto pozici transformovat na základě transformací rodičovského objektu. Přesně to pak řeší tato metoda. Ukázka jejího využití je ve výpisu kódu 4.4.

```
Row* Row::Clone() const {
    auto clone = new Row;
    clone->ApplyTransformations(this);
    clone->spacing = spacing;
    for (auto& renderable : row)
        clone->row.push_back(std::shared_ptr<IRenderable>(renderable->Clone()));

    return clone;
}
```

Výpis kódu 4.3 Ukázka využití metody `ApplyTransformations` ze souboru `Row.cpp`

```
void Row::Update(const sf::Vector2f& mousePos) {
    for (auto& renderable : row)
        renderable->Update(GetTransformedMousePos(mousePos));

    CheckAndReposition();
}
```

Výpis kódu 4.4 Ukázka využití metody `GetTransformedMousePos` ze souboru `Row.cpp`

4.1.2 Jednoduché grafické objekty

Renderovatelné objekty `Rectangle`, `ConvexShape`, `Circle`, `Text` a `Sprite` jsou v podstatě pouhé obaly nad objekty z knihovny SFML, jmenovitě pak `sf::RectangleShape`, `sf::CircleShape`, `sf::Text`, `sf::Sprite`. Důvod vzniku těchto obalů je prostý. Potřebovala jsem všechny renderovatelné objekty sjednotit tím, že budou implementovat stejné rozhraní (abstraktní třídu `IRenderable`). Pracuje se s nimi díky tomu jednodušeji. Ukázku implementace těchto tříd neuvádím pro její trivialitu.

4.1.3 Grafické kontejnery

Název grafický kontejner označuje renderovatelné objekty, které slouží primárně ke změně rozložení jiných renderovatelných objektů. Těmito objekty jsou `Row`, `Column` a `FramedRenderable` (detailně popsány v kapitole 3.4.2). Jejich hlavním rozdílem oproti zbylým renderovatelným objektům je, že se skládají z dceřiných objektů, které se mohou dynamicky měnit – hlavně svými rozměry a polohou. Tedy například `Row` i `Column` musí při každém volání jejich `Update` metody kontrolovat, zda nedošlo ke změně velikosti dceřiného objektu a pokud ano, tak změnit své rozložení, aby se jednotlivé objekty nepřekrývaly. Stejně tak v metodě `GetBounds` se musí rozměry počítat podle momentálních rozměrů dceřiných objektů. Její implementace spolu s metodou `CheckAndReposition`, pro dynamickou změnu rozložení, je ve výpisu kódu 4.6. Na ukázce kódu 4.5 je pak vidět, jak se při přidávání objektu do řády musí počítat jeho počátek v závislosti na již existující řadě.

```
void Row::AddBack(const std::shared_ptr<IRenderable>& added) {
    if (!added)
        return;

    if (!row.empty()) {
        added->setOrigin(
            added->getOrigin().x - GetBounds().width,
            added->getOrigin().y
        );
        added->move(spacing, 0);
    }
    row.push_back(added);
}
```

Výpis kódu 4.5 Ukázka metody `AddBack` ze zdrojového souboru `Row.cpp`

```cpp
void Row::CheckAndReposition() {
    float xPos = 0;
    for (int i = 1; i < row.size(); ++i) {
        xPos += row[i - 1]->GetBounds().width;
        if (i != 1)
            xPos += spacing;

        if (row[i]->getOrigin().x != -xPos)
            row[i]->setOrigin(-xPos, row[i]->getOrigin().y);
    }
}

sf::FloatRect Row::GetBounds() const {
    if (row.empty())
        return {{0, 0}, {0, 0}};

    float width = (row.size() - 1) * spacing;
    float height = 0;

    for (auto& renderable : row) {
        if (renderable->GetBounds().height > height)
            height = renderable->GetBounds().height;

        width += renderable->GetBounds().width;
    }
    return {getPosition(), {width, height}};
}
```

Výpis kódu 4.6 Ukázka zdrojového souboru `Row.cpp`

4.1.4 Tlačítka

Tlačítka určitě byla složitějšími renderovatelnými objekty na implementaci. Kromě toho, že jim bylo třeba implementovat mnoho metod pro nastavení jejich vzhledu, zvuku, či akce provedené při stisku, bylo hlavně potřeba naprogramovat metodu, která určuje, v jakém stavu se vzhledem k myši uživatele tlačítko nachází – metoda `CheckMouse`. Tato metoda se volá v metodě `Update` a je odlišná pro každý typ tlačítka (definované v kapitole 3.4.2). Fundamentálně ale dostane pozici myši a pak kontroluje, v jaké pozici je myš vzhledem k tlačítku, zda je stisknutá a na základě toho nastaví stav tlačítka na stisknutý či nestisknutý. Případně nastaví booleovskou hodnotu pro vyslání signálu provedení akce při stisku (u té bylo třeba dbát na to, aby se při držení tlačítka nastavila pouze na jeden snímek obrazovky). Ve zbytku metody `Update` se podle těchto stavů a hodnot nastaví vzhled, případně spustí zvuk či provede akce při stisku tlačítka. Metodu `Update` periodicky volá *renderer* před vykreslením dalšího snímku. Metody `CheckMouse` jednotlivých typů tlačítek jsou pro mnoho různých mezních případů dlouhé a nebudu je zde uvádět. Ukázku hlavičkového souboru třídy `Button` ale uvádím ve výpisu kódu 4.7.

Akce, kterou má tlačítko provést při stisku, se typicky nastavuje zvenku, buď v *controlleru* anebo ve *view*. Tlačítko se zde vytvoří a předá se mu lambda funkce, kterou vždy zavolá při stisku. Tato funkce nesmí brát žádné parametry ani nic vracet. Příklad toho, jak může takové nastavení akce vypadat, je ve výpisu kódu 4.8.

Po tom, co se tlačítko inicializuje, nastaví se mu akce a předá se *rendereru* (o to se typicky postará *view*, které dané tlačítko spravuje), si žije vlastním životem. Samo si řídí změny svého vzhledu, když s ním uživatel interaguje i provedení akce. Není třeba zvenku složitě zjišťovat, v jakém stavu tlačítko je (jako tomu bylo dříve v původním DMI).

```
class Button : public IRenderable {
public:
    Button() = default;
    void Update(const sf::Vector2f& mousePos) override;
    sf::FloatRect GetBounds() const override;
    Button* Clone() const override;
    void draw(sf::RenderTarget& target, sf::RenderStates states) const override;

    void SetDefaultAppearance(const std::shared_ptr<IRenderable>& renderable);
    void SetClickedAppearance(const std::shared_ptr<IRenderable>& renderable);
    void SetDisabledAppearance(const std::shared_ptr<IRenderable>& renderable);
    void SetClickSound(const std::shared_ptr<sf::Sound>& sound);
    void SetOnCLick(const std::function<void()>& action);
    void SetEnabled(const bool enable);

protected:
    virtual void CheckMouse(const sf::Vector2f mousePos) = 0;
    virtual Button* CloneInstance() const = 0;
};
```

Výpis kódu 4.7 Ukázka hlavičkového souboru třídy `Button` ze souboru `Button.hpp`

```
view->AddButton<ButtonUpType>(L"Driver ID", [&](){
    uiManager->UnregisterController(ControllerType::MainMenu);
    uiManager->RegisterController(ControllerType::DriverID);
});
```

Výpis kódu 4.8 Ukázka nastavení akce tlačítka. Toto tlačítko je součástí hlavního menu DMI a slouží ke změně obrazovky z hlavního menu na *DriverID* menu. Ze souboru `MainMenuController.cpp`

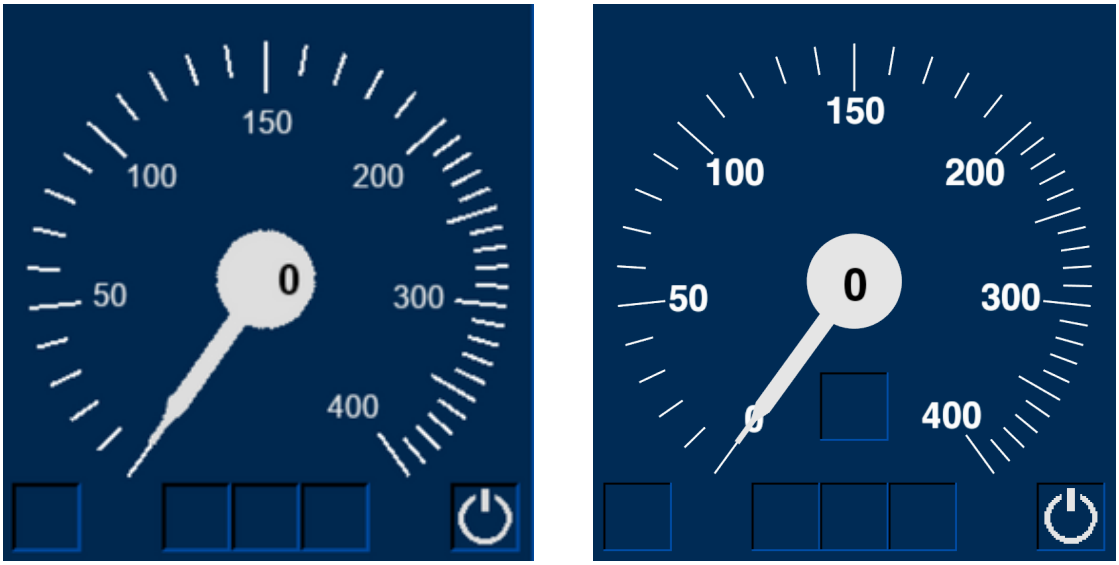
Tlačítko může mít vzhled libovolného renderovatelného objektu. V DMI však většinou vypadá jako na obrázku 4.1. U výchozího a vypnutého vzhledu jde o dva rámečky v sobě, s dalším renderovatelným objektem uvnitř, u stisknutého tlačítka jde pouze o jeden rámeček.

Obrázek 4.1 Popořadě příklady vzhledů výchozího, stisknutého a vypnutého tlačítka. Toto tlačítko je součástí několika obrazovek DMI. Zdroj je přímo z mnou implementované aplikace.

4.1.5 Tachometr

Tachometr se skládá z renderovatelných objektů `SpeedPointer` (ručička) a `SpeedDial` (ciferník). Ručičku tachometru je možné maximálně nastavit – barvou, rozměry, vykresleným číslem i rotací ručičky. Ciferník na rozdíl od ní není tolik přizpůsobitelný. Je to z toho důvodu, že jeho vzhled je natolik svázán se specifikací z oficiální dokumentace, že implementovat ho maximálně nastavitelný by bylo zbytečně složité a kontraproduktivní. Dokumentace ERA definuje celkem čtyři typy ciferníku a pro každý z nich má objekt `SpeedDial` k dispozici metodu, která nastaví jeho celkový vzhled na daný typ. [3]

Při analýze původního DMI jsem zjistila, že jeho tachometr vzhledem přesně neodpovídá požadavkům oficiální dokumentace, a navíc nedokáže při změně velikosti okna aplikace škálovat čísla na ciferníku. Zároveň zde byl implementován pouze jeden typ ciferníku. Ve své implementaci jsem se tedy zaměřila na to, aby tachometr odpovídal vzhledovým požadavkům, korektně se škáloval a aby byly k dispozici všechny jeho typy. Pomocí zapnutí tzv. *antialiasingu* jsem docílila vyhlazeného vzhledu ciferníku. Tuto metodu více popisuji v sekci 4.2.2.



Obrázek 4.2 Vlevo tachometr původního DMI, vpravo mnou vykreslený tachometr v novém DMI.

4.1.6 Views

Každé *view* implementuje abstraktní třídu `BaseView`, ve které jsou implementovány společné funkcionality všech *views*. Na výpisu kódu 4.9 je část implementace, která pokrývá nejdůležitější z těchto funkcionalit.

Na zmíněné ukázce je vidět, jakým způsobem dochází k propojení UI s datovými modely. *View* se ve svém konstruktoru zaregistruje jako *observer* daného modelu a následně v metodě `Notify` (jakmile se zavolá) provede akci, která souvisí s daty modelu. Konkrétně na této ukázce dochází k zapínaní, resp. vypínání vykreslování *view* v závislosti na tom, jaký stav kabiny se nachází v datovém manažeru `GeneralData`, který se nastaví přijmutím zprávy `CabStatusMessage`.

Dále je na ukázce znázorněno, jak dochází k aktualizaci a vykreslování *view*. Každé *view* si totiž drží svou kolekci renderovatelných objektů (třídní proměnná `renderables`), do které musí přidat všechny objekty, které chce vykreslovat. Jakmile *renderer* obdrží *view*, zavolá metodu `draw` na *view* a napřed aktualizuje a potom vykreslí všechny jeho renderovatelné objekty. Je však důležité dbát na pořadí objektů v kolekci. Pokud by se obraz objektů překrýval, pak ten, který je

v kolekci jako první, se bude vždy vykreslovat nejvíce v pozadí a ten, který je poslední, zas nejvíce v popředí.

```
BaseView::BaseView(ServiceContainer& container) {
    auto generalData = communicationData->FetchManager<GeneralData>();
    generalData->AddObserver(this);
    isActive = false;
}

void BaseView::Update(const sf::Vector2f& mousePos) {
    if (!isActive)
        return;

    for (auto& renderable : renderables)
        if (renderable)
            renderable->Update(GetTransformedMousePos(mousePos));
}

void BaseView::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    if (!isActive)
        return;

    states.transform *= getTransform();
    for (auto& renderable : renderables)
        if (renderable)
            target.draw(*renderable, states);
}

void BaseView::Notify() {
    auto generalData = communicationData->FetchManager<GeneralData>();
    isActive = generalData->GetCabin();
}
```

Výpis kódu 4.9 Ukázka zdrojového souboru třídy `BaseView` ze souboru `BaseView.cpp`

Na ukázce kódu 4.10 lze vidět hlavičkový soubor jednoho z implementovaných *views*. Konkrétně jde o obrazovku tlačítkového menu DMI. Ve *view* lze nastavit název obrazovky a libovolně přidávat tlačítka různých typů s textovým či obrázkovým štítkem a tak ho přizpůsobit více obrazovkám DMI.

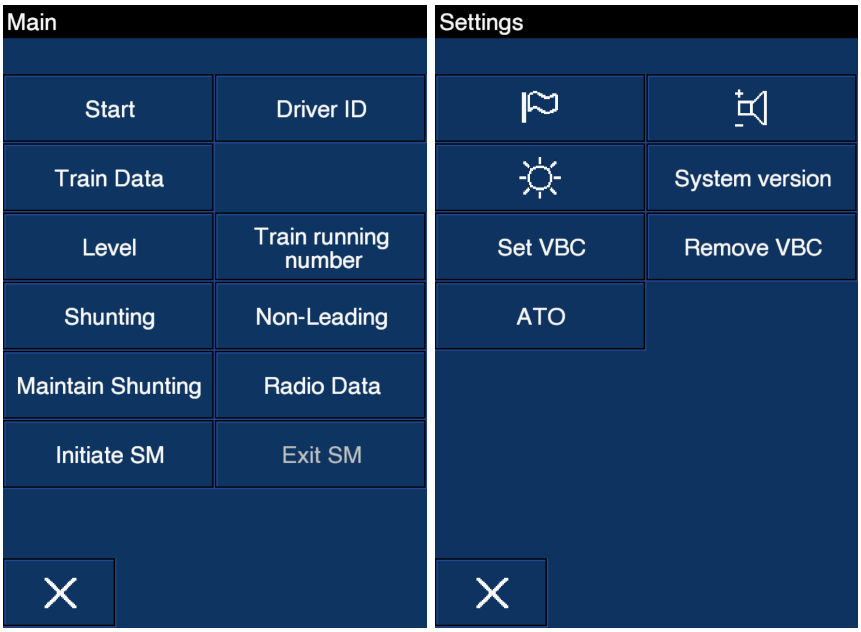
```
class ButtonMenuView : public BaseView {
public:
    template <typename T>
    void AddButton(const std::wstring& label, const std::function<void()>& onClick);
    template <typename T>
    void AddButton(const TextureName enabled, const std::optional<TextureName> disabled,
                   const std::function<void()>& onClick);
    void AddEmptySlot();
};
```

Výpis kódu 4.10 Ukázka hlavičky třídy `ButtonMenuView` ze souboru `ButtonMenuView.hpp`

Na ukázce kódu 4.11 je vidět konstruktor tlačítkového menu. Zde lze nahlédnout, jak dochází ke konstrukci jednotlivých objektů *view* a jak je lze nastavovat přes konfigurace. *View* se může skládat i z dalších *views*, jako je v tomto případě `NavigationView` (lišta ve spodní části menu). Na obrázku 4.3 jsou pro ukázku vykreslena dvě tlačítková menu, která jsou součástí DMI.

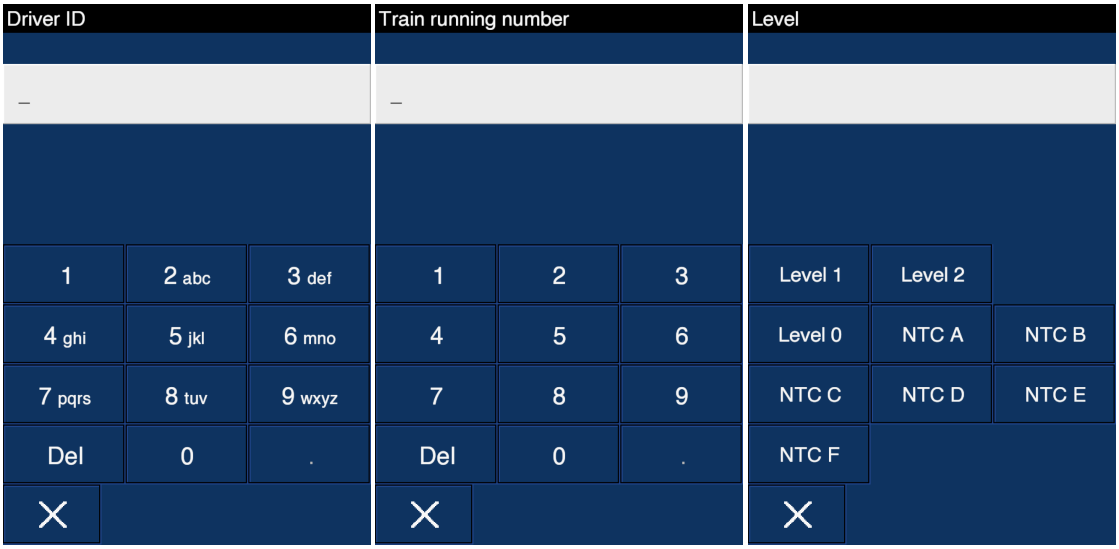
```
1  ButtonMenuView::ButtonMenuView(ServiceContainer& container,
2                                 const std::shared_ptr<NavigationView>& _navigation,
3                                 const std::wstring& titleStr) : BaseView(container) {
4
5      menuConfig = std::make_shared<ButtonMenuConfiguration>(
6          configurationService->FetchConfiguration(ConfigType::ButtonMenu)
7      );
8
9      navigation = _navigation;
10     navigation->SetClose([](){});
11     navigation->setPosition(menuConfig->navigation.x, menuConfig->navigation.y);
12     renderables.push_back(navigation);
13
14     titleBar = renderableFactory->CreateWindowTitleBar(titleStr, Align::LeftCenter);
15     renderables.push_back(titleBar);
16
17     setPosition(menuConfig->view.x, menuConfig->view.y);
18 }
```

Výpis kódu 4.11 Konstruktor třídy `ButtonMenuView` ze souboru `ButtonMenuView.cpp`



Obrázek 4.3 Vlevo hlavní tlačítkové menu, vpravo tlačítkové menu nastavení DMI. Zdroj je mnou implementovaná aplikace.

Kromě *view* pro tlačítkové menu jsem implementovala i `HalfGridInputMenuView` pro menu určené pro zadávání dat vykreslené na polovinu obrazovky (ve specifikaci DMI existuje i menu pro zadávání dat, které pokrývá celou obrazovku [3], to však zatím implementované není). Na obrázku 4.4 jsou některé obrazovky DMI, které `HalfGridInputMenuView` využívají. Pro účely tohoto *view* vzniklo i `KeyboardView` pro klávesnici. Klávesnice má několik druhů – numerická, alfanumerická a dedikovaná. Dedikovaná klávesnice může nastavit libovolné textové popisy jednotlivým klávesám. Všechny tyto klávesnice jsou vidět na zmíněném obrázku (nutno podotknout, že alfanumerická klávesnice v původním DMI vůbec neexistovala, i když podle specifikace [9] měla).



Obrázek 4.4 Vlevo *Driver ID* menu (alfanumerická klávesnice), uprostřed *Train Running Number* menu (numerická klávesnice) a vpravo *Level* menu (dedikovaná klávesnice). Zdroj je mnou implementovaná aplikace.

Klávesnice zadávacích menu by měly fungovat přesně tak, jak jsou popsány v oficiální specifikaci. Tedy numerická i alfanumerická je složena z tlačítek druhu *down-type* s opakovačem a dedikovaná z tlačítek téhož druhu bez opakovače. Alfanumerická klávesnice navíc funguje tak, že při opakovaném stisku téhož tlačítka, které má vícero symbolů, na vstupním poli tyto symboly iterují nad nehybným kurzorem. Až pokud uživatel stiskne jiné tlačítko anebo nestiskne nic po dobu 2 s, se zadaný symbol potvrdí a kurzor se posune o jednu pozici. [3]

4.1.7 Controllers

Controllery stejně jako *views* implementují společnou abstraktní třídu, a to `BaseController`. Ten ovšem na rozdíl od `BaseView` není příliš zajímavý, pouze inicializuje ukazatele na několik služeb, které všechny *controllery* využívají.

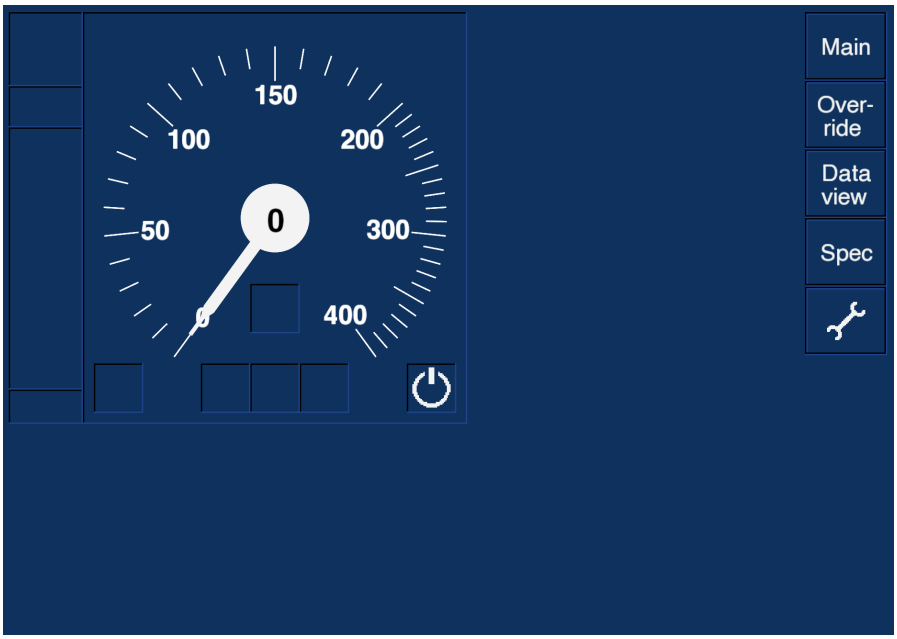
Co bych však chtěla vysvětlit je, jak jednotlivé *controllery* fungují. V ukázce kódu 4.12 je vidět část konstruktoru *controlleru* a jeho metoda `Update` (jedná se o *controller* hlavního menu na obrázku 4.3). Tímto způsobem zachází s *view* každý *controller*. Nejprve si jej inicializuje, potom nastaví podle svých požadavků – zde je například vidět, jak se nastavují tlačítka a jejich akce pro přepínání obrazovky – a následně v metodě `Update` pošle *view rendereru*. Jeden *controller* může spravovat i více *views* najednou. V metodě `Update` se bude typicky nacházet také logika za UI, specifikovaná v dokumentaci od ERA [3]. Ta však není předmětem mé práce.

```
MainMenuController::MainMenuController(ServiceContainer& container,
                                       const std::shared_ptr<ButtonMenuView>& _view)
                                    : BaseController(container) {
    view = _view;
    view->AddButton<ButtonUpType>(L"Start", [](){});
    view->AddEmptySlot();
    view->AddButton<ButtonUpType>(L"Level", [&](){
        uiManager->UnregisterController(ControllerType::MainMenu);
        uiManager->RegisterController(ControllerType::Level);
    });
    view->AddButton<ButtonDelayType>(L"Shunting", [](){});

    view->SetCloseButtonOnClick([&](){
        uiManager->UnregisterController(ControllerType::MainMenu);
        uiManager->RegisterDefaultWindowRight();
    });
}

void MainMenuController::Update() {
    renderer->RenderInForeground(view);
}
```

Výpis kódu 4.12 Ukázka části konstruktoru a metody `Update` třídy `MainMenuController` ze souboru `MainMenuController.cpp`

4.1.8 Ukázka výchozí obrazovky DMI

Zde bych chtěla na obrázku 4.5 ukázat, jak nyní vypadá výchozí obrazovka DMI. Interakcí s tlačítky v pravé svislé liště se lze prokliknout na zbylé obrazovky menu.



Obrázek 4.5 Výchozí obrazovka DMI. Zdroj je mnou implementovaná aplikace.

4.2 Služby

V této sekci bych chtěla vysvětlit jednotlivé implementované služby.

4.2.1 UIManagerService

Tato služba je hlavním správcem celého UI. V hlavní aplikační smyčce periodicky volá *renderer* a spojuje ho s MVC architekturou. Při svém startu si inicializuje a uloží všechny dostupné *controllery*. Jakmile se aplikace spustí, vytvoří si aplikační okno a zaregistruje *controllery*, které spravují tu část UI, která se má na obrazovce zobrazit. Následně spustí smyčku a nechá aplikaci, aby si žila svým životem. Celý tento proces je vidět na ukázce kódu 4.13.

Na zmíněné ukázce kódu je také vidět metoda `UpdateControllers()`. Tato metoda existuje z toho důvodu, že *controllery* mohou za běhu aplikace registrovat a odregistrovat libovolné další *controllery* z UI manažeru. Tato technika je vidět například na ukázce 4.12. Zde je třeba podotknout, že pouze momentálně zaregistrované *controllery* se ve smyčce aktualizují a posílají svá *views* *rendereru*. Tj. registrování, resp. odregistrování *controllerů* slouží ke změnám obrazovky UI. Samotné přidávání či odebírání aktivních *controllerů* probíhá právě v metodě `UpdateControllers()`.

```
1  void UIManagerService::Run() {
2      renderer->CreateWindow();
3
4      jruLoggerService->Log(true, MessageType::Info, "Started rendering DMI");
5      RegisterController(ControllerType::MainMenu);
6      RegisterDefaultWindowLeft();
7
8      // The main UI loop
9      while (!exit) {
10         UpdateControllers();
11         SetWindowBackground();
12         renderer->RenderFrame();
13     }
14 }
15
16 void UIManagerService::UpdateControllers() {
17     activeControllers.merge(toRegister);
18
19     for (auto& controller : toUnregister)
20         activeControllers.erase(controller);
21
22     toRegister.clear();
23     toUnregister.clear();
24
25     for (auto& controller : activeControllers)
26         if (allControllers.find(controller) != allControllers.end())
27             allControllers[controller]->Update();
28 }
```

Výpis kódu 4.13 Ukázka zdrojového souboru`UIManagerService.cpp`

4.2.2 RendererService

Hlavní úloha této služby je zcela zřejmá – stará se o vykreslování grafických objektů na obrazovku a správu okna aplikace. Okno inicializuje na základě získaných konfigurací, vždy ve středu obrazovky displeje zařízení. Ve výchozím nastavení je zapnutý tzv. *antialiasing*, což je technika, která zajišťuje, že vykreslené objekty se jeví být vyhlazené, bez hrbolatých okrajů. [25] Původní DMI toto postrádalo. *Antialiasing* má ovšem vyšší nároky na výkon grafické karty počítače, proto je možné, že bude potřeba jej na méně výkonných zařízeních vypnout. To lze učinit přes konfiguraci.

Služba `RendererService` zároveň ve výchozím nastavení inicializuje okno aplikace s horní lištou. Pomocí konfigurací se však dá zařídit, aby jej inicializovala v módu bez lišty a pokrývalo celou obrazovku – tzv. *fullscreen* (do tohoto módu se dá přepnout i stiskem klávesy F11).

Nejdůležitější metodou služby je však `RenderFrame`, ve které dochází k vykreslení jednoho snímku obrazovky. V každém snímku je nejprve třeba odchytit události spojené s interakcí uživatele a okna aplikace (jako je změna velikosti či zavření okna), zareagovat na ně, pak aktualizovat všechny grafické objekty a následně je vykreslit. Tato metoda je na ukázce kódu 4.14. Rychlost vykreslování jednotlivých snímků je omezena na 60 FPS.

```
1  void RendererService::RenderFrame() {
2      for (auto event = sf::Event(); window.pollEvent(event); )
3      {
4          switch (event.type) {
5              case sf::Event::Closed:
6                  ExitWindow();
7                  return;
8              case sf::Event::Resized:
9                  if (!fullscreen)
10                     windowSize = window.getSize();
11
12                 ResizeViewAspectRatio(resizableView);
13                 window.setView(resizableView);
14                 break;
15             case sf::Event::KeyPressed:
16                 if (event.key.code == sf::Keyboard::F11) {
17                     fullscreen = !fullscreen;
18                     CreateWindow();
19                 }
20             default:
21                 break;
22         }
23     }
24
25     window.clear(background);
26     for (auto& item : renderables) {
27         item->Update(GetMousePos());
28         window.draw(*item);
29     }
30     window.display();
31     renderables.clear();
32 }
```

Výpis kódu 4.14 Ukázka metody `RenderFrame` ze souboru `RendererService.cpp`. Při změně velikosti okna dochází k jejímu ukládání proto, aby při zrušení módu *fullscreen* přešlo okno do původní velikosti.

4.2.3 AssetStorageManager

Tato služba je určená pro správu všech textur, fontů a zvuků, které si při své inicializaci načte a pak pomocí metod v ukázce 4.15 je dosazuje na místa, kde jsou potřeba. Typicky se tato služba volá ve *views* anebo v `RenderableFactoryService`.

```
class IAssetStorageService : public IService {
public:
    virtual std::shared_ptr<sf::Font> GetFont(const FontName name) = 0;
    virtual std::shared_ptr<sf::Texture> GetTexture(const TextureName name) = 0;
    virtual std::shared_ptr<sf::Sound> GetSound(const SoundName name) = 0;
};
```

Výpis kódu 4.15 Ukázka části hlavičkového souboru `IAssetStorageService.hpp`

4.2.4 RenderableFactoryService

Služba `RenderableFactoryService` má velmi jednoduchou implementaci. Pouze poskytuje programátorovi sadu metod, pro vytvoření často používaných renderovatelných objektů, jako jsou rámečky, tlačítka a texty nastaveny na požadované parametry. Služba se volá typicky ve *views*. Některé poskytované metody lze vidět na ukázce 4.16.

```
class IRenderableFactoryService : public IService {
public:
    virtual std::shared_ptr<FramedRenderable> CreateDefaultFrame(
        const float width, const float height, const std::shared_ptr<IRenderable>& renderable
    ) const = 0;

    virtual std::shared_ptr<Text> CreateText(
        const FontName font, const uint32_t& fontSize, const std::wstring& str
    ) const = 0;

    virtual std::shared_ptr<Button> CreateDefaultButton(
        const float width, const float height, std::shared_ptr<Button> buttonInstance,
        const std::shared_ptr<IRenderable>& defaultLabel,
        const std::shared_ptr<IRenderable>& disabledLabel
    ) const = 0;

    virtual std::shared_ptr<InputField> CreateInputField(const std::wstring& label) const = 0;
    virtual std::shared_ptr<InputField> CreateDataOnlyInputField() const = 0;
};
```

Výpis kódu 4.16 Ukázka části hlavičkového souboru `IRenderableFactoryService.hpp`

4.2.5 CommunicationDataService

Úkol této služby je pouze správa datových manažerů. Při své inicializaci si načte všechny manažery a následně je vydává různým částem kódu. Typicky se tato služba využívá v *message handlerech*, *views* a *controllerech*. Její hlavička je na ukázce 4.17.

```
1 class ICommunicationDataService : public IService {
2 public:
3     virtual std::shared_ptr<IDataManager> FetchManager(DataManagerType type) const = 0;
4     virtual std::map<DataManagerType, std::shared_ptr<IDataManager>> GetAllManagers() const = 0;
5 };
```

Výpis kódu 4.17 Ukázka části hlavičkového souboru `ICommunicationDataService.hpp`

4.2.6 Datové manažery

Manažery jsou svým základem jednoduché datové modely. Pro každou datovou položku mají *getter* a *setter*. K tomu navíc mají možnost implementovat rozhraní `ISendable`, `IRequest-able` anebo `IAcknowledgeable` (popsáno v kapitole 3.3.2). Tyto třídy poskytují deklarace abstraktních metod, které manažerům umožňují svá data odesílat, vyžádat či potvrdit pomocí komunikace přes MQTT protokol. Na ukázce 4.18 je hlavička jednoho z manažerů. Každý manažer zároveň dědí ze společné třídy `IDataManager`, která dále dědí z `Observable`, protože datové manažery mohou být sledovány *views* a *controllery* při změnách dat.

```
1 class DriverData : public IDataManager, public ISendable {
2 public:
3     explicit DriverData(ServiceContainer& container);
4
5     void Send() override;
6
7     void SetDriverID(const uint32_t id);
8     uint32_t GetDriverID() const;
9 };
```

Výpis kódu 4.18 Ukázka části hlavičkového souboru `DriverData.hpp`

Kapitola 5
Testování

Nyní se přesouvám k poslední kapitole praktické části mé práce – testování. Jedná se o velmi důležitou součást vývoje každé aplikace. Důkladně provedené testování pomáhá zvýšit spolehlivost, a tedy i kvalitu. V této kapitole se zaměřím na technologii využitou pro testování a jednotlivé typy testů, které jsem na DMI aplikovala.

5.1 GoogleTest

Pro testování DMI jsem zvolila testovací *framework* GoogleTest, který slouží k testování aplikací napsaných v jazyce C++. Vybrala jsem ho z důvodu již mnohokrát zmiňovaného zachování konzistence s ostatními komponentami simulátoru ETCS, které jej pro testování taktéž využívají. Tento *framework* umožňuje psát mnoho druhů testů (jednotkové, integrační, systémové...), které rozděluje do tzv. testovacích sad (angl. *test suites*). Jedna testovací sada by měla ideálně obsahovat testy týkající se spolu souvisejících funkcionalit. GoogleTest využívá k testování tzv. aserce (angl. *assertions*), pomocí kterých vyhodnocuje výsledek testu. Tvrzení může být buď úspěšné, nebo může skončit fatální či nefatální chybou. Pokud dojde k fatální chybě, test se v jejím místě zastaví a nebude dál probíhat. V ostatních případech test vždy doběhne do konce. [26]

Ačkoliv mi zmíněná knihovna testování velmi ulehčila, i přesto to nebyl snadný úkol. Testování zdrojového kódu napsaného v jazyce C++ může naskýtat mnoho výzev. Například pro každý *mock* je třeba explicitně napsat vlastní třídu, která bude dědit z *mockované* třídy a bude definovat všechny *mockované* metody. Zároveň jsem z tohoto důvodu musela upravit několik zdrojových souborů, aby je vůbec bylo možné testovat. Každá metoda, kterou chci *mockovat*, totiž musí být virtuální. Toto automaticky přináší další problém, čímž je testování *template* metod, které v C++20 virtuální být nemohou. Jediné řešení bylo nechat *template* metodu volat jinou virtuální metodu. Pokud to však nešlo provést, tak jsem *template* úplně odstranila a zvolila jiný postup.

5.2 Jednotkové testování

Prvním druhem automatických testů, které jsem provedla, jsou jednotkové testy. Tyto testy se zaměřují na funkcionality malých částí kódu, typicky na jednotlivé třídy (mohou to ale být i funkce, v závislosti na modularitě aplikace). Každý test by měl být nezávislý na kódu mimo testovanou třídu, aby jeho výsledky byly zcela jednoznačné – tedy pokud je testovaná třída závislá na jiných, je třeba je *namockovat*. Jednotkové testy se snaží narazit na chyby co nejdříve v průběhu vývoje. Slouží také k ověření funkčnosti jednotlivých částí aplikace, podpoře nízké provázanosti a v neposlední řadě mohou posloužit i jako dokumentace, jelikož v nich lze nahlédnout na to, jakým způsobem se má daný kód využívat. [27]

Snažila jsem se jednotkovými testy pokrýt co největší část DMI. Některé části jsem jimi nepokryla, jako například renderovatelné objekty a *views* – důvod k jejich nepokrytí uvádím v kapitole 5.4. Zároveň jsem netestovala ani triviální části tříd, jako jsou *gettery* a *settery*. Testy jsem pokryla zejména služby aplikace, *message handlery* a datové manažery.

5.3 Integrační testování

Některé třídy či metody aplikace jsou implementovány tak, že nedává velký smysl je testovat jednotkovými testy. Například takové *controllery*, které vydávají pokyny pro přechod mezi obrazovkami DMI. Samotný *controller* nemá jak zjistit, že k přechodu obrazovky doopravdy došlo. Tuto informaci si nese služba UI manažeru a je třeba otestovat, že *controllery* s ní v tomto směru správně interagují. Z tohoto důvodu jsem zavedla druhý typ automatických testů, kterým jsou integrační testy. Tyto testy se zabývají pravě ověřením interakcí mezi více různými objekty aplikace. Pomocí nich jsem otestovala všechny implementované přechody mezi obrazovkami a také interakci datových manažerů s *message handlery* při ukládání dat z přijatých zpráv.

5.4 Testovací scénáře

DMI jako takové je poměrně obtížné automaticky testovat, a to hlavně z toho důvodu, že se jedná o aplikaci s grafickým rozhraním. Některé programovací jazyky či grafické knihovny poskytují prostředky k testování UI, u C++ a SFML tomu tak bohužel není. Celé aplikační okno i všechny vykreslené grafické objekty jsou automatickými testy v podstatě netestovatelné a je třeba je tedy testovat manuálně. Z tohoto důvodu jsem sepsala několik manuálních testovacích scénářů, které pokrývají funkcionality, jež nelze automaticky testovat. Popisuji je na následujících řádcích.

[TC1] Potvrzení zadaných dat a odeslání EVC

Popis: Testuje, zda se korektně uloží a pošlou data zadaná uživatelem v menu.

Předpoklady: Aplikace byla právě zapnuta v debugovacím módu.

Data: *Level 0*

Kroky:

1. Kliknout na tlačítko *Level*.
2. Pomocí klávesnice na obrazovce zadat přiložená data.
3. Potvrdit vstup kliknutím na vstupní pole.

Očekávaný výsledek: Aplikace se vrátí zpět do hlavního menu a v konzoli se vypíše logový záznam o odeslání dat EVC. Při navrácení do *Level* menu budou ve vstupním poli stále vyplněna vstupní data.

[TC2] Zrušení zadaných dat

Popis: Testuje, zda se data korektně smažou při zrušení zadávání vstupu.

Předpoklady: Aplikace byla právě zapnuta v debugovacím módu.

Data: 1234

Kroky:

1. Kliknout na tlačítko *Train running number*.
2. Pomocí klávesnice na obrazovce zadat přiložená data.
3. Zrušit vstup kliknutím na tlačítko s křížkem.

Očekávaný výsledek: Aplikace se vrátí zpět do hlavního menu. Při navrácení do *Train running number* menu bude vstupní pole opět prázdné.

[TC3] Správná funkce delay-type tlačítka

Popis: Testuje, zda se *delay-type* tlačítko chová očekávaným způsobem.

Předpoklady: Aplikace byla zapnuta v debugovacím módu a bylo stisknuto tlačítko *Radio data* pro přechod do příslušné obrazovky.

Kroky:

1. Kliknout a držet tlačítko *Radio network type*.
2. Čekat alespoň 2 sekundy a pak tlačítko pustit.

Očekávaný výsledek: Po dobu držení tlačítko problikává z výchozího vzhledu na stisknutý vzhled. Jakmile uplynou 2 sekundy, tlačítko zůstane napořád ve stisknutém vzhledu. Po puštění tlačítka se aktivuje a přepne obrazovku do *Radio network type menu*. Pokud se tlačítko pustí dříve než uplynou 2 sekundy, nedojde k jeho aktivaci.

[TC4] Správná funkce down-type tlačítka s opakovačem

Popis: Testuje, zda se *down-type* tlačítko s opakovačem chová očekávaným způsobem.

Předpoklady: Aplikace byla zapnuta v debugovacím módu a bylo stisknuto tlačítko *Train running number* pro přechod do příslušné obrazovky.

Kroky:

1. Kliknout a držet tlačítko klávesnice s číslem 1.
2. Čekat alespoň 2 sekundy a pak tlačítko pustit.

Očekávaný výsledek: Po stisknutí tlačítka se vypíše číslo 1 do vstupního pole. Při jeho držení se po 1,5 sekundě začne tlačítko opakovaně mačkat a vypisovat další jedničky do vstupního pole. Pokud by se tlačítko pustilo dříve než uplyne 1,5 sekundy, opakovač se neaktivuje.

[TC5] Správná funkce alfanumerické klávesnice

Popis: Testuje, zda se alfanumerická klávesnice chová očekávaným způsobem.

Předpoklady: Aplikace byla zapnuta v debugovacím módu a bylo stisknuto tlačítko *Driver ID* pro přechod do příslušné obrazovky.

Kroky:

1. Třikrát kliknout na tlačítko klávesnice s nápisem „2 abc“.
2. Počkat dvě sekundy.
3. Jednou kliknout na tlačítko klávesnice s nápisem „2 abc“.

Očekávaný výsledek: Ve vstupním poli je zadáno „b2“. Při opakovaném klikání na tlačítko na výstupu iterují znaky nad nehybným kurzorem. Po uplynutí 2 sekund se kurzor pohne o jednu pozici a je možné zadat další znak.

5.5 Výsledky testů

Kromě výše zmíněných testů jsem po celou dobu vývoje pravidelně testovala běh aplikace s pomocí nástroje Valgrind, který pomáhá odhalit špatnou práci aplikace s pamětí počítače. Zároveň jsem testovala běh aplikace na operačním systému Linux a Windows. V průběhu testování jsem narazila na řadu chyb, které jsem samozřejmě všechny opravila. DMI tedy předávám dalším vývojářům v odladěném stavu. Níže uvádím výčet nalezených chyb.

- **Vícenásobné uvolňování a úniky paměti přidělené objektům** – mohlo způsobit zhoršený výkon a pády aplikace.
- **Chyby závislé na použitém operačním systému** – jednalo se o chybějící či špatně přidané knihovny, protože Windows používá v některých případech jiné knihovny než Linux (například pro práci s *filesystémem*).
- **Špatně inicializované objekty využívající zvuk** – ukázalo se, že třída si nikdy nesmí přímo držet objekt `sf::Sound` nebo `sf::SoundBuffer` z knihovny SFML, protože při vytvoření instance takové třídy se automaticky inicializuje daný zvukový objekt a s ním i zvukový výstup. Toto způsobovalo problémy při automatickém testování na GitLabu, kde se projekt spouští ve virtuálním prostředí, které žádný zvukový výstup nemá. Při využívání těchto zvukových objektů je tedy třeba je ukládat jako ukazatel a inicializovat je až ve chvíli, kdy je doopravdy potřeba – tedy ne při testování.
- **Nevyužité konstruktory** – v kódu jsem nalezla několik čistě abstraktních tříd, které měly bezvýznamné konstruktory.
- **Nízké využívání *dependency injection* v konstruktorech** – tato chyba se projevila při testování *controllerů*, které využívají různá *views*. Původně si každý *controller* všechna svá *views* inicializoval ve svém konstruktoru. Toto však znemožňovalo *views* při testování *mockovat*. Po opravě se všechna *views* předávají *controllerům* jako parametr konstruktoru.
- **Využívání virtuálních metod v konstruktorech** – toto je praktika, která způsobuje nedefinované chování aplikace (mohla by se zavolat jiná implementace metody, než kterou si třída definuje). [28]
- **Špatně deklarované výčtové typy** – namísto `enum class` byly výčtové typy deklarovány pouze jako `enum`. Toto způsobovalo kolize, pokud dva různé výčtové typy obsahovaly položku se stejným názvem.

5.6 Spuštění testů

Všechny automatické testy (tedy jednotkové a integrační) proběhnou spuštěním binárního souboru `DMI_test`, který je součástí přílohy. Tento soubor se také vygeneruje při sestavení celé aplikace podle návodu v příloze A. Po spuštění tohoto souboru proběhnou všechny testy a jejich výsledek se vypíše do konzole. U každého testu se vypíše, zda proběhl v pořádku. Pokud ne, bude uvedena chyba, ke které došlo. Všechny automatické testy se zároveň samy spouští při *pushnutí* kódu na GitLab projektu prostřednictvím CI/DC.

Kapitola 6

Shrnutí praktické části práce

V této kapitole bych chtěla shrnout výstup praktické části mé práce. Nejprve uvedu, do jaké míry se mi podařilo v práci splnit požadavky definované v kapitole 2.6. Následně uvedu rozsah odvedené implementace a na závěr se budu věnovat doporučením pro budoucí vývoj aplikace.

6.1 Splnění definovaných požadavků

V praktické části práce jsem se opírala o požadavky definované v kapitole 2.6. K jejich míře splnění se vyjadřuji v tabulce 6.1 a 6.2.

Požadavek Stav	Poznámky
[P1] Kompletní přepis Splněn	Zdrojový kód původního DMI byl velmi neudržitelný, kompletně jsem přepsala architekturu komponenty a započala reimplementaci obrazovek UI.
[P2] Rozšiřitelnost a udržitelnost Splněn	Nová architektura DMI je navržena s důrazem na budoucí rozšiřitelnost, udržitelnost a snadnost vývoje.
[P3] Nízká provázanost a vysoká soudržnost Splněn	Jednotlivé třídy architektury jsou maximálně zapouzdřeny a snaží se vždy řešit jeden problém.
[P4] Nastavení vhodné modularity Splněn	Jádro architektury je rozděleno do menších logických celků s využitím návrhových vzorů SOA a EDA. Grafická část aplikace využívá návrhový vzor MVC, který vhodně odděluje datové modely, uživatelské rozhraní a logiku aplikace.
[P5] Specifikační verze 4.0.0 Splněn	Aplikace je implementována podle oficiální dokumentace ERA specifikační verze 4.0.0 [3].
[P6] Konzistence s architekturou ostatních komponent Splněn	Jádro aplikace je sjednoceno s jádrem ostatních komponent. Zbytek aplikace je postaven na stejném principu.

Tabulka 6.1 Stav splnění požadavků P1-P6 definovaných v kapitole 2.6

Požadavek Stav	Poznámky
[**P7**] Návrh grafického *frameworku* Splněn	Vznikla sada renderovatelných objektů založených na grafické knihovně SFML. Objekty jsou maximálně přizpůsobitelné a generické menší celky, které se mohou skládat do sebe, nebo do *views*. Každý objekt plní svůj vlastní úkol. Využila jsem návrhový vzor MVC pro stavbu uživatelského rozhraní. Vznikly také podpůrné služby, které se starají o chod UI, vykreslování grafiky a ukládání zdrojů.
[**P8**] Více modelů displeje Splněn	Návrh architektury počítá s možností více různých podob grafického displeje. Jejich modely půjdou přidávat prostřednictvím konfigurací.
[**P9**] Překlady Částečně splněn	Překlady textů aplikace jsem neimplementovala, nicméně můj návrh s nimi počítá a definuje, jak by měla implementace vypadat.
[**P10**] Komunikace s ostatními komponentami Splněn	Aplikace je schopna komunikovat s ostatními komponentami simulátoru podle analýzy v kapitole 2.5. Navrhla a implementovala jsem systém pro ukládání komunikačních dat.

Tabulka 6.2 Stav splnění požadavků P7-P10 definovaných v kapitole 2.6

6.2 Rozsah implementace DMI

Každý soubor repozitáře s implementací v příloze prošel mýma rukama, nicméně přesný rozsah práce se hůře určuje, protože jádro implementace jsem si vypůjčila od ostatních komponent simulátoru. V každém hlavičkovém či zdrojovém souboru, který jsem alespoň z většiny vytvořila já, jsem však v dokumentaci uvedena jako tvůrce. Takových souborů je celkem 292 se součtem 17690 řádků (i s komentáři).

Cílem práce nebyla kompletní reimplementace UI, vzniklá částečná reimplementace však ověřuje můj návrh architektury. Pro upřesnění stavu implementace jednotlivých obrazovek uvádím v tabulkách 6.3, 6.4, 6.5 a 6.6 přehled, který má sloužit zejména jako souhrn pro budoucí vývojáře. U nedokončených položek v tabulkách zmiňuji konkrétní objekty, které bude třeba implementovat. Ty popíši v kapitole 6.3.

Menu obrazovka DMI Stav implementace	Poznámky
Hlavní menu Implementováno	Je třeba přiřadit akce některým tlačítkům.
ID strojvedoucího Částečně implementováno	Chybí tlačítka pro přechod do nastavení a obrazovky čísla vlaku. Zadaná data se odesílají EVC, ale komunikace mezi komponentami momentálně podporuje pouze numerický identifikátor, ačkoliv by měl být textový.
Úroveň (*Level*) Implementováno	Úroveň ETCS se po zadání odešle EVC. Úrovně NTC se zatím neodesílají, protože v simulátoru zatím neexistují.

Tabulka 6.3 Stav implementace menu obrazovek (1. část)

Menu obrazovka DMI Stav implementace	Poznámky
Vlaková data Neimplementováno	Implementace obrazovky chybí, nicméně měly by být připraveny všechny renderovatelné objekty, ze kterých se obrazovka poskládá. Povede na implementaci `FullGridInputMenuView`.
Validovat vlaková data Neimplementováno	Povede na implementaci `ValidationMenuView`.
Číslo vlaku Implementováno	Data se zatím neodesílají EVC, jelikož je EVC neumí přijmout.
Rádiová data Implementováno	Je třeba přiřadit akce některým tlačítkům.
ID GSM-R sítě Implementováno	Data se zatím neodesílají EVC, jelikož je EVC neumí přijmout.
RBC data Neimplementováno	Povede na implementaci `FullGridInputMenuView`.
Typ radiové sítě Implementováno	Data se zatím neodesílají EVC, jelikož je EVC neumí přijmout.
Mise s jedním radiovým systémem Neimplementováno	Povede na implementaci `ConfirmationMenuView`.
Potlačení (*Override*) Implementováno	Data se zatím neodesílají EVC, jelikož je EVC neumí přijmout.
Prohlížení dat Neimplementováno	Povede na implementaci `DataMenuView`.
Speciální Implementováno	Je třeba přiřadit akce některým tlačítkům.
Adheze Implementováno	Data se zatím neodesílají EVC, jelikož je EVC neumí přijmout.
SR rychlost/vzdálenost Neimplementováno	Povede na implementaci `FullGridInputMenuView`.
Nastavení Implementováno	Je třeba přiřadit akce některým tlačítkům.
Vybrat ATO Implementováno	Data se zatím neodesílají EVC, jelikož je EVC neumí přijmout.
Jazyk Částečně implementováno	Zatím není implementována `TranslatorService` a tedy překlady nejsou k dispozici. V nabídce je pouze angličtina.

Tabulka 6.4 Stav implementace menu obrazovek (2. část)

Menu obrazovka DMI Stav implementace	Poznámky
Hlasitost Neimplementováno	Povede na implementaci `AdjustmentMenuView`.
Jas Neimplementováno	Povede na implementaci `AdjustmentMenuView`.
Verze systému Neimplementováno	Povede na implementaci `DataMenuView`.
Nastavit VBC Neimplementováno	Povede na implementaci `FullGridInputMenuView`.
Odstranit VBC Neimplementováno	Povede na implementaci `FullGridInputMenuView`.
Validovat nastavení VBC Neimplementováno	Povede na implementaci `ValidationMenuView`.
Validovat odstranění VBC Neimplementováno	Povede na implementaci `ValidationMenuView`.

Tabulka 6.5 Stav implementace menu obrazovek (3. část)

Část výchozí obrazovky Stav implementace	Poznámky
Část A Částečně implementováno	Pouze připraveny rámečky pro jednotlivé prvky této části.
Část B Částečně implementováno	Implementován tachometr a rámečky pro symboly. Chybí kruhový měřič rychlosti (křivka kolem tachometru), nicméně samotný renderovatelný objekt pro jednoduchou kruhovou křivku existuje. Není napojeno na logiku aplikace.
Část C Neimplementováno	Obsahuje pouze několik rámečků se symboly a tlačítka – půjde poskládat z již existujících renderovatelných objektů.
Část D Neimplementováno	Tato část má mnoho prvků, které se mění v závislosti na přijatých datech od EVC, a tak bude její implementace složitější. Všechny potřebné renderovatelné objekty však již existují.
Část E Neimplementováno	Bude třeba vytvořit text schopný zalamování – povede na implementaci renderovatelného objektu `MultilineText`.
Část F Implementováno	Je třeba přiřadit akci tlačítku *Data view*.
Část G Neimplementováno	Obsahuje pouze několik rámečků se symboly – půjde poskládat z již existujících renderovatelných objektů.

Tabulka 6.6 Stav implementace všech částí výchozí obrazovky DMI specifikovaných na obrázku 2.10.

6.3 Návrhy na budoucí rozvoj projektu

Nyní bych se chtěla věnovat tomu, jakým směrem by se budoucí vývoj projektu měl ubírat. Logická cesta je samozřejmě implementace zbytku obrazovek DMI a logiky vykreslování UI na základě získaných komunikačních dat. Níže uvádím několik svých konkrétních návrhů.

6.3.1 Views

Z kapitoly 6.2 o rozsahu implementace vyplývá z nedokončených obrazovek DMI hned několik *views*, které bude třeba implementovat. Níže přibližuji, jak by měla vypadat.

DataMenuView

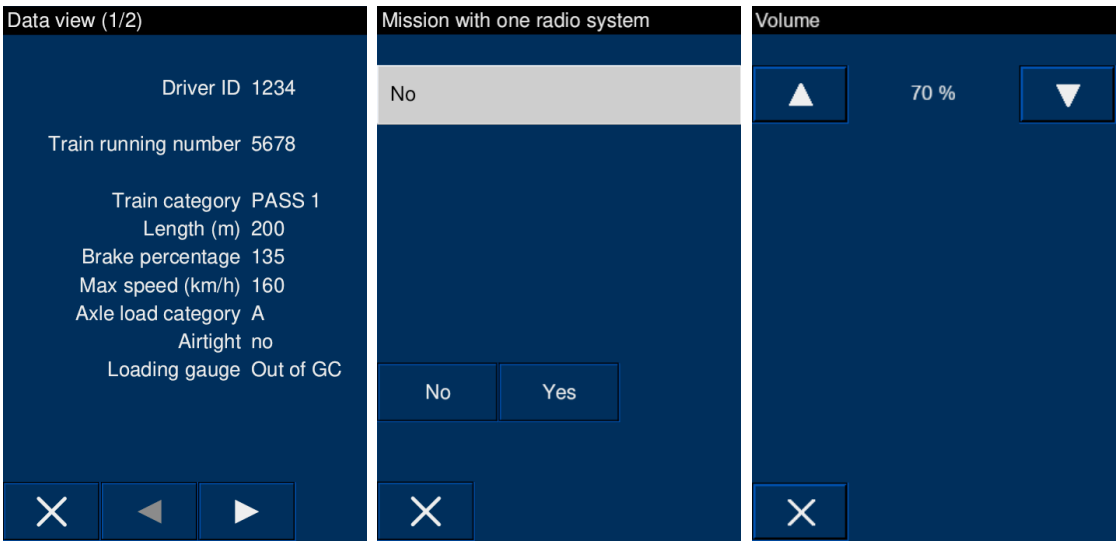
Toto *view* bude sloužit k prostému výpisu dat. Bude mít horní lištu s názvem obrazovky a bude obsahovat dva renderovatelné objekty sloupců s variabilním počtem řádků. *View* bude mít ve své spodní částí již implementované `NavigationView`. `DataMenuView` bude přímo využívat obrazovka *Prohlížení dat* a *Verze systému*. Vzhled tohoto *view* je na obrázku 6.1.

ConfirmationMenuView

View `ConfirmationMenuView` bude pouze nadstavba již existujícího `HalfGridInputMenuView`, které bude mít nastavená pouze dvě tlačítka dedikované klávesnice. `ConfirmationMenuView` bude přímo využívat obrazovka *Mise s jedním rádiovým systémem*. Vzhled tohoto *view* se nachází na obrázku 6.1.

AdjustmentMenuView

Toto *view* bude sloužit k nastavení nějaké hodnoty pomocí tlačítek. Bude mít horní lištu s názvem obrazovky, dvě tlačítka a hodnotu mezi nimi, která se bude stiskem tlačítek ovlivňovat. Ve spodní části *view* bude `NavigationView`. `AdjustmentMenuView` bude přímo využívat obrazovka *Hlasitost* a *Jas*. Vzhled tohoto *view* je k nahlédnutí na obrázku 6.1.

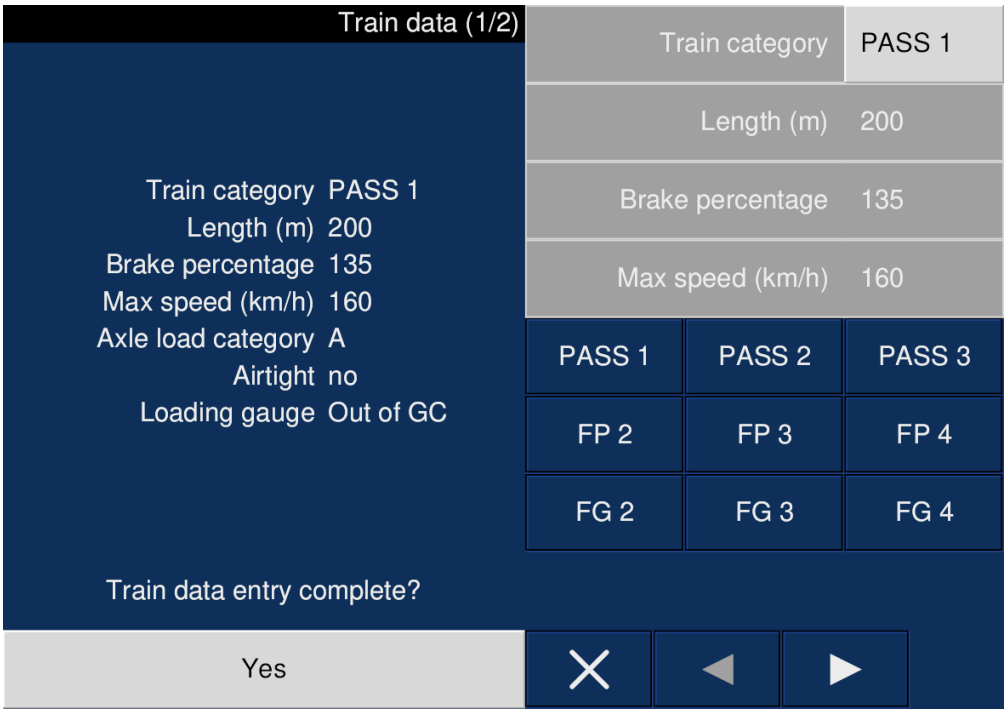


Obrázek 6.1 Vlevo příklad `DataMenuView`, uprostřed `ConfirmationMenuView` a vpravo `AdjustmentMenuView`. Obrázky vlevo a uprostřed byly získány z oficiální dokumentace ERA [3]. Obrázek vpravo je z původní aplikace DMI.

FullGridInputMenuView

Toto *view* se bude skládat ze dvou částí. Jeho pravá část bude mít ve sloupci variabilní množství `InputField` objektů (vstupních polí), které budou mít viditelný štítek s popiskem dat. Každé z těchto polí bude mít možnost nastavit typ klávesnice, pomocí které se budou zadávat data. Při kliknutí na aktivní pole se zadaná data potvrdí, pole se stane neaktivním a aktivuje se pole, které následuje za ním. Ve spodní části se bude nacházet navigační panel.

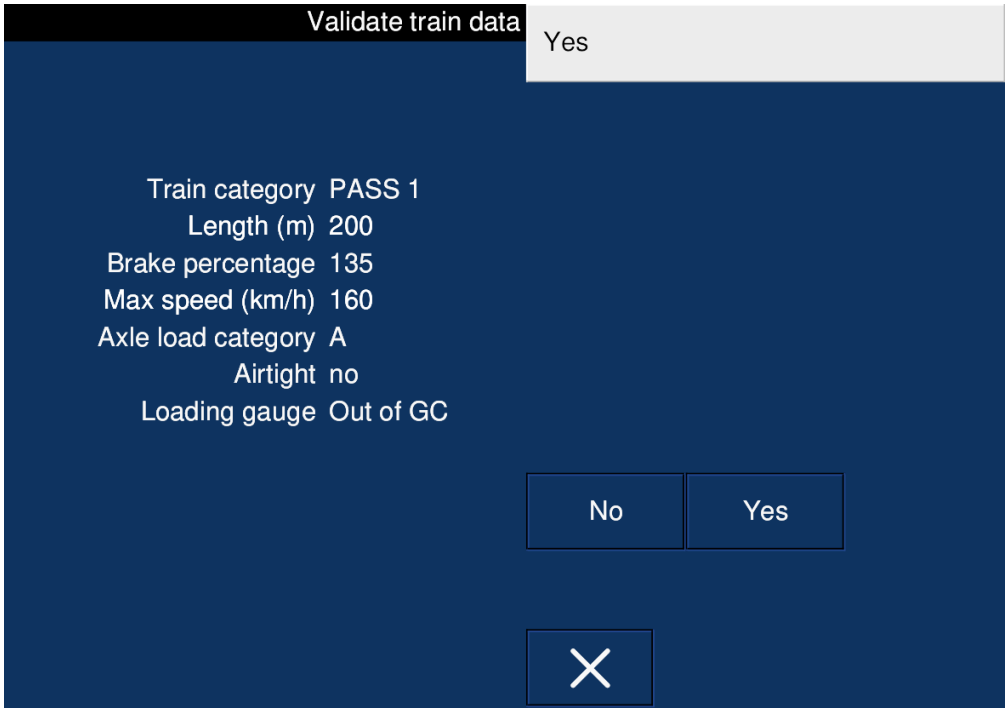
Levou část `FullGridInputMenuView` bude tvořit separátní *view*. Nazvu ho `DataOverviewView`. Bude velmi podobné `DataMenuView`, akorát bude mít název obrazovky zarovnaný doprava a místo navigačního panelu bude tlačítko pro potvrzení všech zadaných dat. `FullGridInputMenuView` bude přímo využívat obrazovka *Vlaková data*, *RBC Data* a několik dalších. Vzhled tohoto *view* je k nahlédnutí na obrázku 6.2.



Obrázek 6.2 Příklad `FullGridInputMenuView`. Obrázek získán z oficiální dokumentace ERA [3].

ValidationMenuView

View `ValidationMenuView` bude taktéž tvořeno ze dvou částí. Levou část bude tvořit `DataOverviewView`, které bude mít skryté tlačítko pro potvrzení dat. Pravou část pak bude tvořit `ConfirmationMenuView`. `ValidationMenuView` bude přímo využívat obrazovka *Validovat vlaková data*, *Validovat nastavení VBC* a několik dalších. Vzhled tohoto *view* je k nahlédnutí na obrázku 6.3.



Obrázek 6.3 Příklad `ValidationMenuView`. Obrázek získán z oficiální dokumentace ERA [3].

6.3.2 TranslatorService

V kapitole 3.5.5 představuji ideu služby `TranslatorService`, která bude sloužit pro překládání textů v aplikaci. Tuto službu bude třeba implementovat, jelikož se jedná o důležitou funkci aplikace.

6.3.3 MultilineText

Navrhuji implementovat nový renderovatelný objekt `MultilineText`, který by umožňoval automaticky zalamovat text, jakmile by překročil specifikovanou délku. Mělo by být také možné určit, kam se bude text zarovnávat – doleva, doprava anebo na střed. Zároveň bude vhodné také umožnit objektu volbu, zda má zalamovat text po celých slovech a nebo zda rozdělit poslední slovo spojovníkem (například text tlačítka *Override* v části F výchozí obrazovky DMI se zalamuje se spojovníkem). Víceřádkový text využívají zejména tlačítka UI. Momentální řešení, kdy se do víceřádkových textů uměle přidává odřádkování a bílé znaky není ideální.

Kapitola 7
Závěr

Hlavním cílem mé práce bylo navrhnout novou architekturu komponenty DMI, která je součástí simulátoru vlakového zabezpečovače ETCS. Architektura měla dávat důraz na budoucí rozšiřitelnost a udržitelnost a měla být konzistentní s architekturou ostatních komponent.

V teoretické části práce jsem popsala doménu simulátoru jako takovou, zanalyzovala historii vývoje komponenty prostřednictvím předešlých bakalářských prací a na jejím základě identifikovala nedostatky původní architektury, přiblížila návrh architektury jádra ostatních komponent simulátoru, zanalyzovala komunikaci DMI s těmito komponentami a na závěr ze všech těchto poznatků definovala požadavky na můj architektonický návrh.

Praktická část práce se dělila na návrh, implementaci a testování. V návrhu jsem představila technologie, které jsem se rozhodla využít pro implementaci. Následně jsem popsala způsob komunikace aplikace s ostatními komponentami simulátoru, vysvětlila architekturu grafické části komponenty a kapitolu zakončila návrhem několika klíčových služeb. Návrh jsem vymýšlela tak, aby předcházel architektonickým nedostatkům v původním DMI, a aby aplikaci povýšil z původní verze 2.3.0 na 4.0.0. V implementační kapitole jsem poté popsala, jak fungují jednotlivé části aplikace a jak probíhala jejich implementace. Praktickou část své práce jsem zakončila popisem automatických testů, které jsem nad implementací uskutečnila, a sadou testovacích scénářů pro manuální testování uživatelského rozhraní.

Výstupem mé práce je návrh, implementace a otestování solidního základu navržené architektury – tedy jádro celé aplikace, komunikace s ostatními komponentami, grafický *framework* pro snadnou tvorbu uživatelského rozhraní a část samotného UI. Práce byla založena na požadavcích definovaných v kapitole 2.6, k jejichž míře splnění se vyjadřuji v kapitole 6.1. Kód byl řádně zdokumentován a vznikly také instalační a vývojářské příručky, které jsou součástí přílohy. Cíle stanovené na začátku práce se mi tedy podařilo splnit. Její výsledek nyní významně usnadní a zrychlí budoucí vývoj komponenty DMI a zvýší tak celkovou kvalitu projektu simulátoru ETCS.

Příloha A

Instalační příručka

Instalační příručka je po domluvě s vedoucím práce přiložena v anglickém jazyce, neboť je vystavena společně se všemi zdrojovými soubory na GitLab repozitáři projektu, který je celý v angličtině. Původní formát příručky je Markdown, pro účely tohoto textu však byla převedena do formátu LaTeX. Příručka je v původním znění s mírnými, převážně estetickými úpravami. Jedná se o mé dílo.

Linux

Download dependencies

```
sudo apt install libmosquitto-dev libsfml-dev
```

Build

Use the following commands from the root of the project. You can optionally enable debug mode by specifying the flag bellow. The debug mode just logs some events in the console and adds warning flags to compiler. You can also disable connection to MQTT for easier debugging.

```
git submodule update --init
cmake -B build [-DENABLE_DEBUG=ON] [-DMQTT=OFF]
cmake --build build -j <number of threads to build on>
```

Run

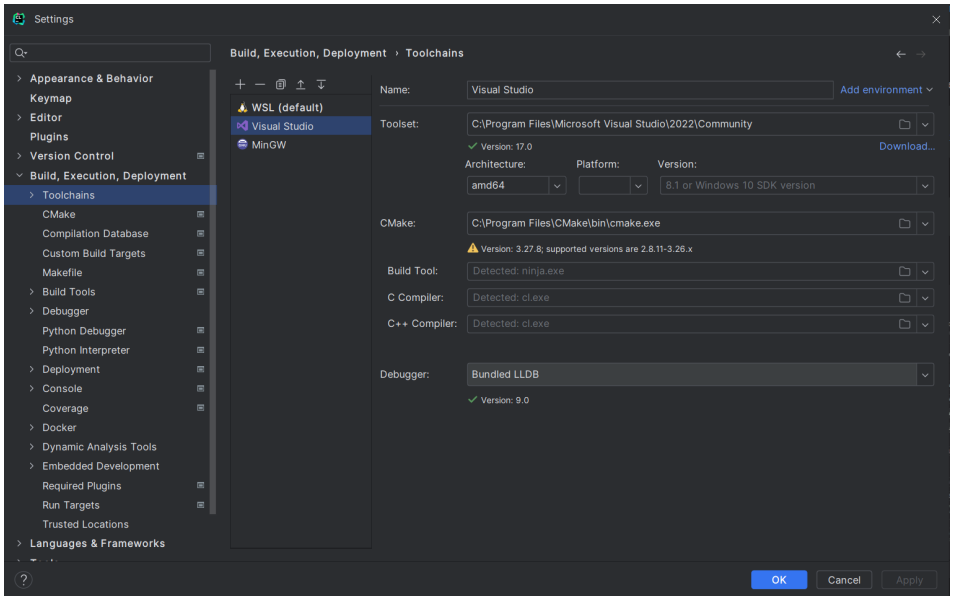
```
cd build
./DMI_run
```

Windows (CLion)

In order to start building on Windows, we need to install the Visual Studio compiler, which is a part of the Visual Studio for C++ (available for download here [29]).

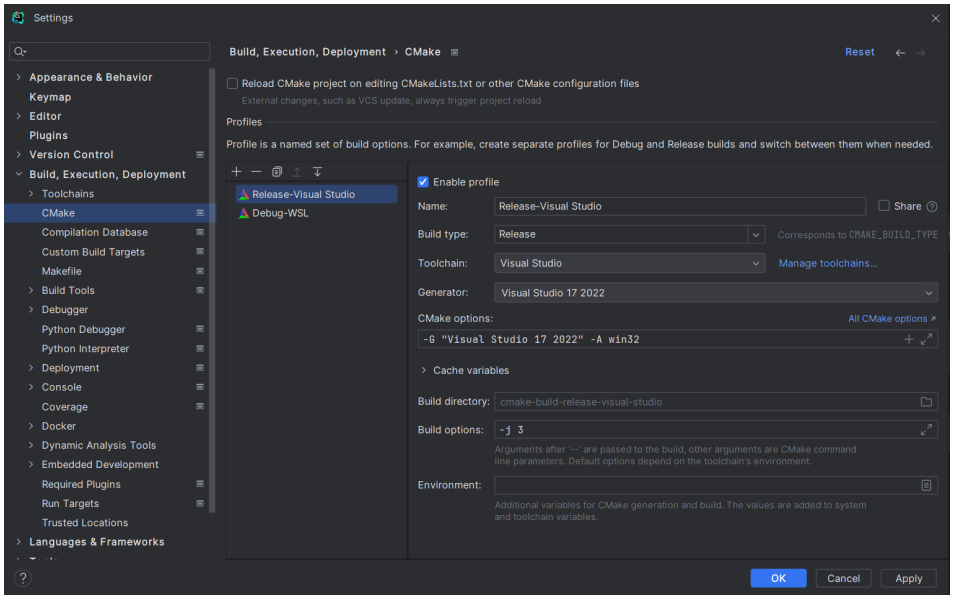
After running the installer, choose to install **Desktop development with C++**. In case you don't want to install the entire IDE, you can choose to only install **MSVC**, **C++ Cmake tools for Windows** and the **Windows SDK** (its version is dependent on the version of your Windows).

After installation, we need to set up a Visual Studio toolchain in our project. In CLion, go to *File > Settings > Build, Execution, Deployment > Toolchains*. Click on the plus button to create a new toolchain and choose Visual Studio. The IDE should automatically detect Visual Studio compiler. It should look something like this:



Obrázek A.1 Visual Studio toolchain

Finally, we will set up a CMake profile for the build. Go to *File > Settings > Build, Execution, Deployment > CMake*. Click on the plus button and set it up as shown on the picture bellow. Don't forget to add the **`-A win32`** flag in the CMake options. Afterwards, run the `git submodule update --init` in the console and then run the project.



Obrázek A.2 Cmake build profile

WSL2

Download dependencies

```
sudo apt install libmosquitto-dev libsfml-dev
```

Build

Use the following commands from the root of the project. You need to specify the WSL flag (it will disable antialiasing, as WSL is not able to do it). You can optionally enable debug mode by specifying the flag bellow. The debug mode just logs some events in the console and adds warning flags to compiler. You can also disable connection to MQTT for easier debugging.

```
git submodule update --init
cmake -B build -DWSL=True [-DENABLE_DEBUG=ON] [-DMQTT=OFF]
cmake --build build -j <number of threads to build on>
```

Graphical output

WSL does not have any graphical output, so we will have to download the VcXsrv Windows XServer [30], which will provide it for us. First, install the server. Then add the following line of code into your `.bashrc` file.

```
export DISPLAY="$(cat /etc/resolv.conf | grep nameserver | awk '{print $2}'):0"
```

After that, run the XLaunch you just installed. There will be a series of configuration windows. Keep everything as it is except for the option ”Disable access control”in the third window - check this option on.

Now you should be able to run the DMI application. If you get the *Failed to open X11 display; make sure the DISPLAY environment variable is set correctly* error, something is wrong with the `DISPLAY` variable in the `.bashrc`.

Audio output

Same as the graphical output, you don’t get any audio in your WSL. To set things up follow the tutorial here [31]. It is not necessary to have the audio output configured in order to run the project, but you will get a bunch of warnings each time an audio object gets created or called.

Příloha B

Vývojářská příručka

Po vzoru instalační příručky, i tato je vystavena na GitLab repozitáři projektu a je tedy napsána v anglickém jazyce. I tato příručka je v původním znění s mírnými, převážně estetickými úpravami, a jedná se o mé dílo.

About

This is a manual on how to work with the graphics framework developed for this project. It will cover the creation of individual UI parts, what needs tobe done to create them and what to watch out for. This is a very simple guide, covering the most important things. For more in depth information about the architecture, explanation of the implementation etc. read this entire bachelor thesis.

Creating new renderable object

Each renderable object has to implement the `IRenderable` abstract class, so that the application can work with it correctly. It MUST implement these methods:

```
1  sf::FloatRect GetBounds() const override;
2
3  Text* Clone() const override;
4
5  void draw(sf::RenderTarget& target, sf::RenderStates states) const override;
```

GetBounds(): Determines the position and the bounding rectangle of the object in global coordination system. It is needed for example by objects like Row or Column, whose task is to change the layout of the objects.

Clone(): Utility method for correct object cloning. Do not forget to copy the transformations of the source object as well.

draw(): Needed by the renderer. If something is not rendering on the right place or missing on the screen, the problem is probably in this method.

The object can optionally implement this method:

```
1 void Update(const sf::Vector2f& mousePos) override;
```

This method will be periodically called by the renderer, and it will update the object. There should not be any insane reallocation or generation of graphical objects. Minimize in as much as possible. If you need to regenerate the renderable object here, do it only when it actually needs to change its graphics.

If your new renderable object consists of other renderable objects, the `Update()` method needs to be implemented ALWAYS. You need to call the `Update()` methods of the child renderable objects here. But before you do so, do not forget to transform the mousePos parameter using the protected method `GetTransformedMousePos()`.

If you need to render the object on screen, just pass it to the `IRendererService` using the `RenderInForeground()` or `RenderInBackground()` method. However, the renderables themselves should not use the `ServiceContainer` and its services! They should be simple, generic and as customizable as possible. Give them setters for everything.

For more information on how to implement renderables, just look into some other renderable's code (for something simple, see the `Circe` or `SpeedPointer`, for something more complex, see the `Row` or `FramedRenderable`).

Creating new view

View is just another renderable object. The main difference from basic renderables is, that it is more complex, consists of many renderables or even other views, and it also can use the `ServiceContainer` and its services. It needs to implement the `BaseView` abstract class. It does not have to reimplement the `Update()` or the `draw()` methods. However, it needs to implement these methods:

```
1 sf::FloatRect GetBounds() const override;
2
3 void Notify() override;
```

GetBounds(): Works the same as with renderables.

Notify(): Used by the Observer pattern. This method gets called if a data manager observed by the view triggers it. It usually gets triggered in message handlers, when the data in managers change. By default, the implementation of this method should at least call the `Notify()` method from the parent `BaseView` class.

In the view's constructor, you should initialize and set up the child renderables and views. You can also use the `IRenderableFactory` for creating an often used renderable (such as frames, buttons or texts). If you need to access a certain texture, font or a sound, use the `IAssetStorageService` to inject it. Once you set up a child renderable, just add it to the member variable of the view called `renderables`. It is a vector of renderables, that gets iterated over inside the `Update()` and `draw()` method of the `BaseView` class. So this is how you register the object for rendering and updating. Make the view as customizable as possible, so it can be reused for more DMI windows.

For more information on how to implement views, just look into some other view's code (for example the `KeyboardView` or the `ButtonMenuView`).

Creating new controller

Before creating a new controller, you need to add its name to the `ControllerType` enum. You will need it for its initialization and registering to the UI manager. A controller needs to implement the `BaseController` abstract class. It should also implement this method:

```
void Update(const sf::Vector2f& mousePos) override;
```

In the `Update()` method, the controller will usually send its views to the renderer. There should also be the logic behind the UI. Such as when to enable (or disable) which button, when to show some objects onscreen etc.

The controller should in its constructor initialize and set up all the views it wants to use. Such as their appearances and most importantly, the actions of the buttons in the views. It should not initialize any data models, as they are initialized elsewhere (in the `CommunicationDataService`).

After creating the controller, go to the `UIManagerService`, then to the `InitControllers()` method and add your controller here. Now it should be wired into the app correctly. If you need to pass the controller to the app (so you can start its `Update()` method and render its views onscreen), just call this method of the UI manager:

```
uiManager->RegisterController(ControllerType:YourController);
```

For more information on how to implement controllers, just look into some other controller's code (for example the `MainMenuController` or the `DriverIDController`).

Data Managers

In this MVC architecture, data models are referred to as data managers, as they can not only store data, but also manage it using the communication services. They can either send, request or acknowledge data from other ETCS components. For this, the data managers can implement these classes: `ISendable`, `IRequestable` and `IAcknowledgeable`. The data gets written into the managers in message handlers, which handle the incoming communication messages.

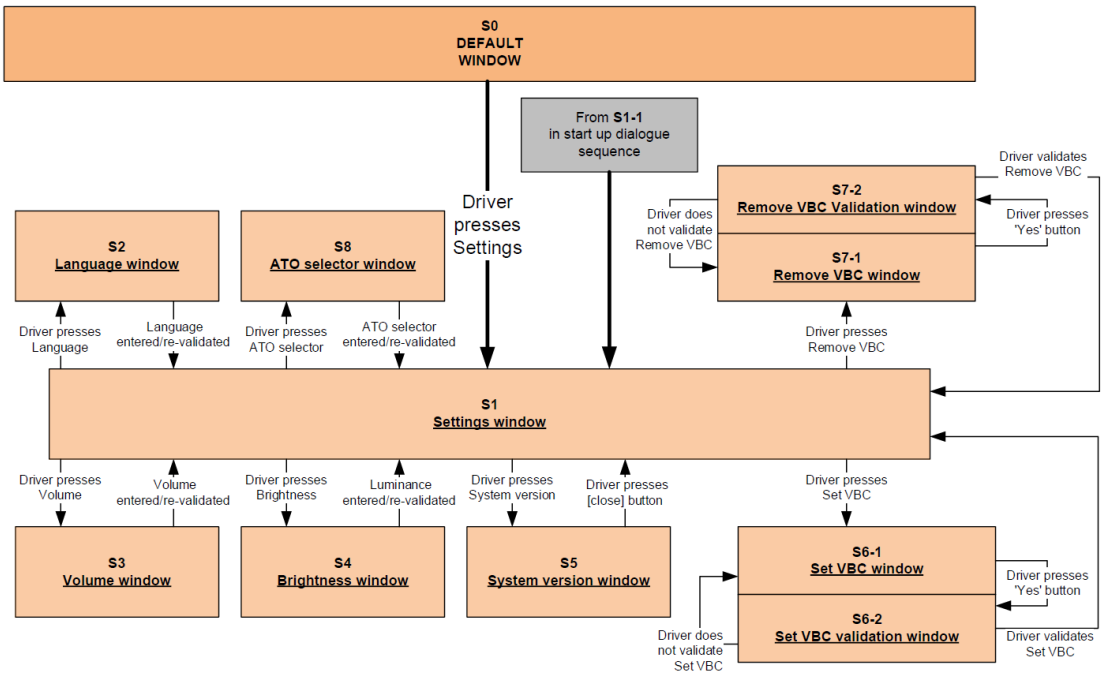
Data managers are also observables (from the Observer architecture pattern). They can be observed with classes, that implement the `IObservable` abstract class. This is usually used, when the data in the manager gets rewritten in the message handler. Once it gets rewritten all the observers of the manager will get notified of the change (the observers are usually views, but controllers also could be). Observers will then perform something in their `Notify()` method.

For more information on how to implement data managers, just look into some other data managers' code (for example the `DriverData` or the `TrainData`).

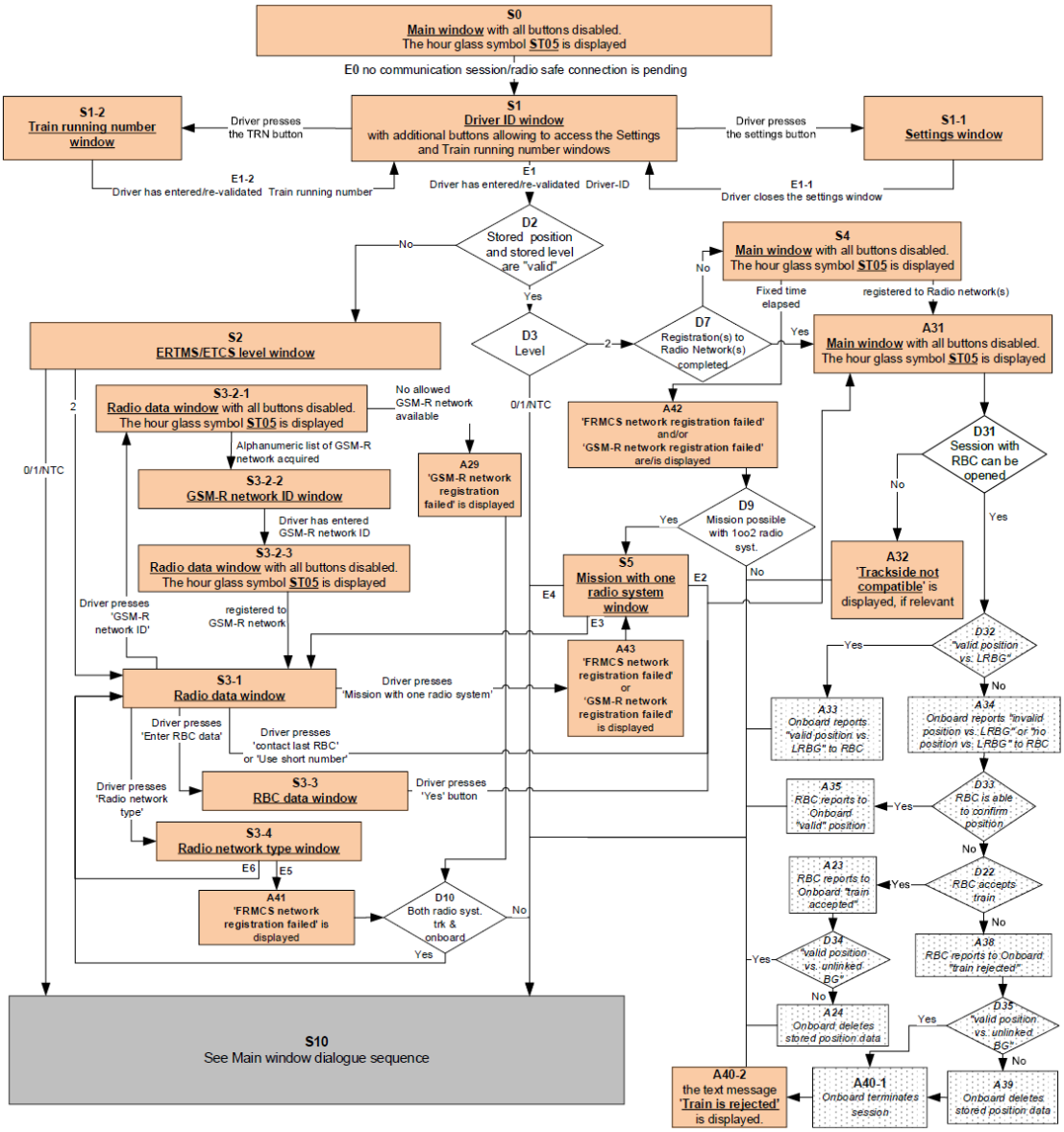
Příloha C

Diagramy přechodů mezi obrazovkami DMI

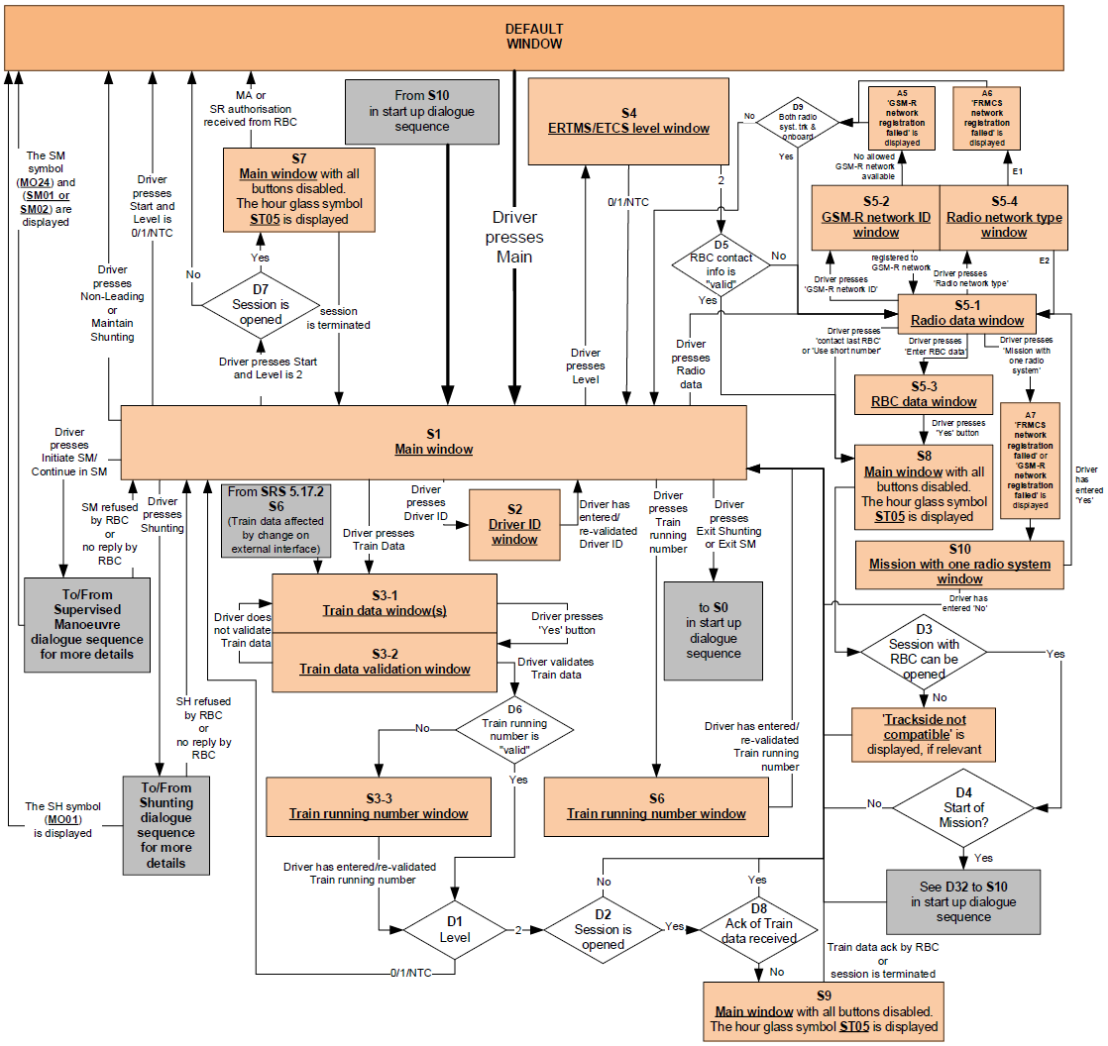
Zde se nachází několik diagramů z oficiální dokumentace DMI verze 4.0.0 od ERA [3]. Diagramy slouží k lepšímu pochopení přechodů mezi DMI obrazovkami. Pro více detailů a vysvětlení některých přechodů však odkazuji do zmíněné dokumentace.



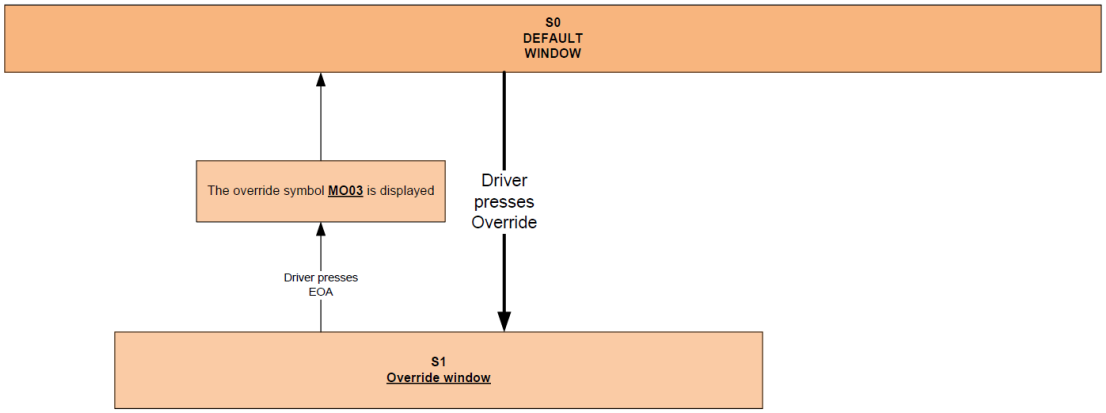
Obrázek C.1 Přepínání obrazovek v menu *Nastavení* [3]



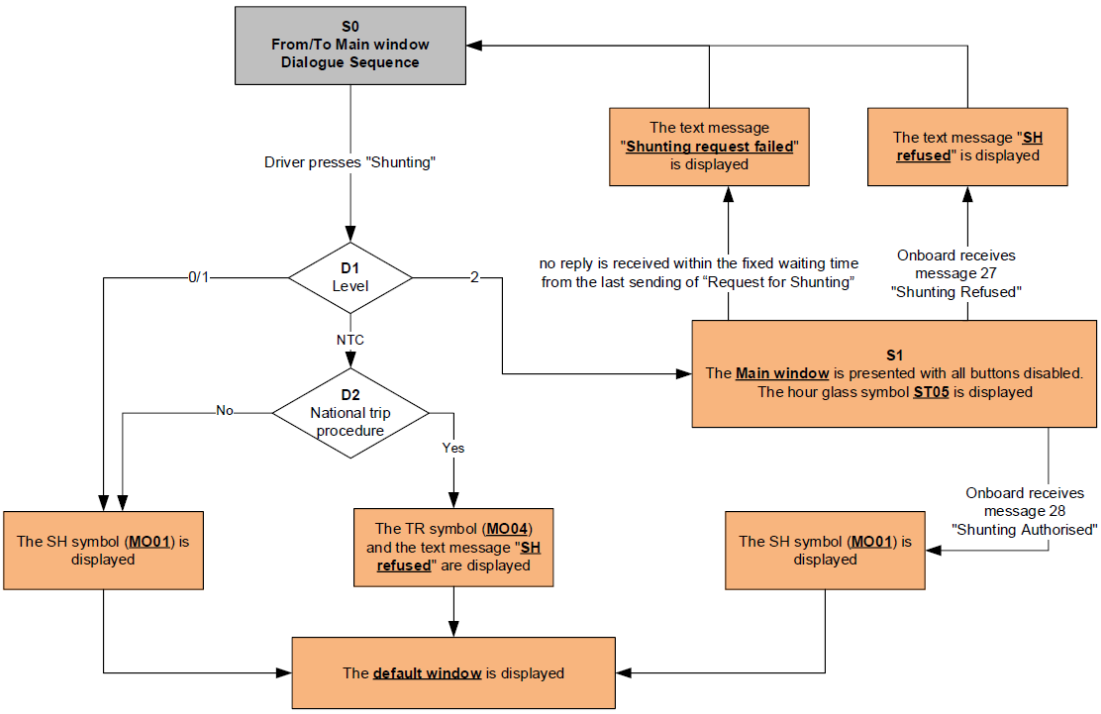
Obrázek C.2 Způsob změny obrazovek po spuštění DMI [3]



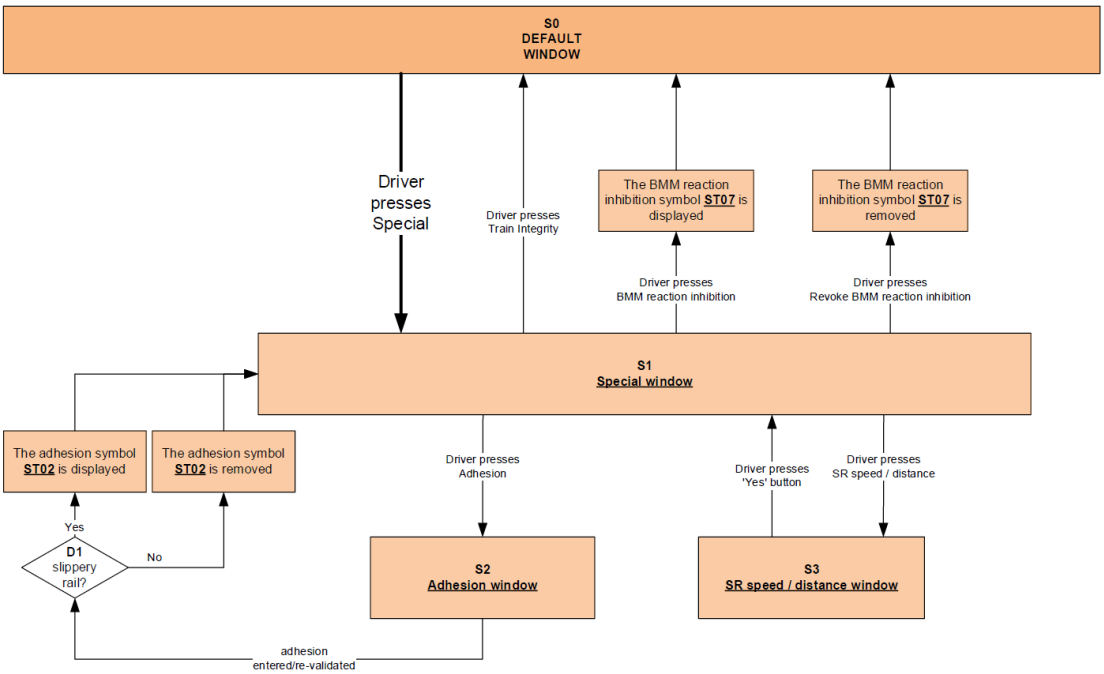
Obrázek C.3 Přepínání obrazovek v hlavním menu [3]



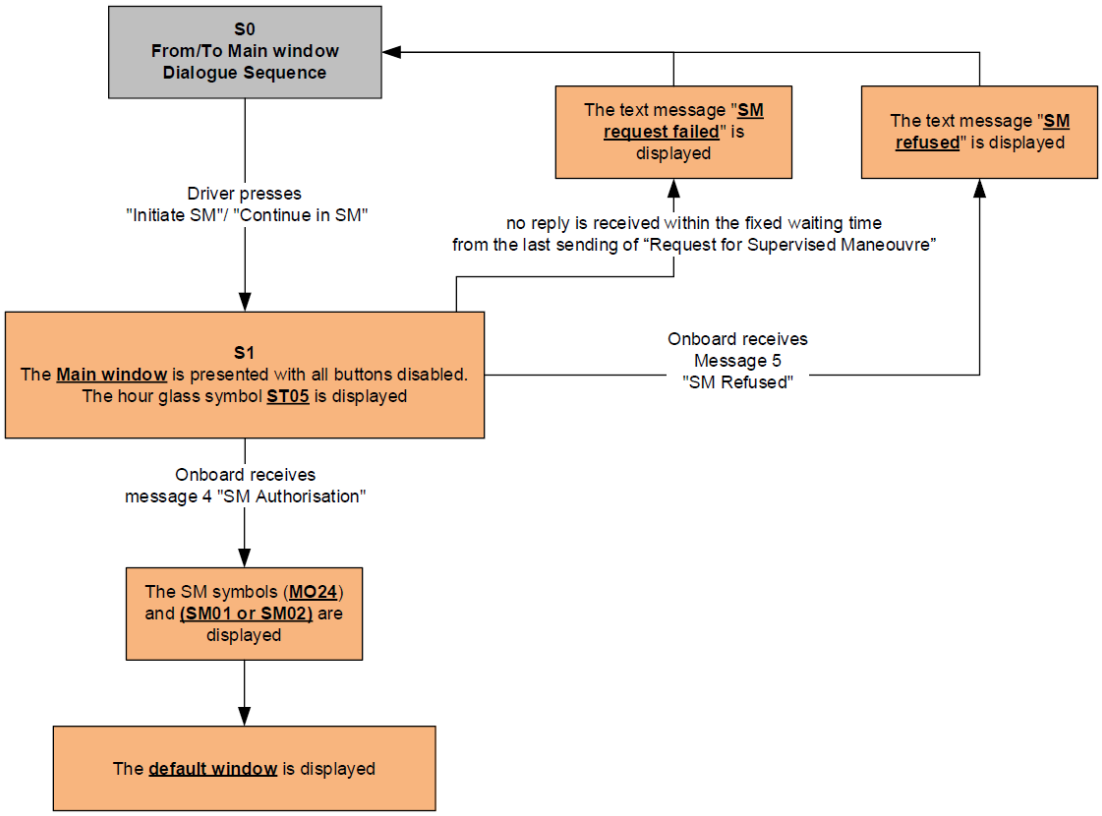
Obrázek C.4 Přepínání obrazovek v menu *Potlačení* [3]



Obrázek C.5 Přepínání obrazovek po stisku tlačítka *Shunting* v hlavním menu [3]



Obrázek C.6 Přepínání obrazovek v menu *Speciální* [3]



Obrázek C.7 Přepínání obrazovek po stisku tlačítka *Initiate SM* v hlavním menu [3]

Bibliografie

1. EUROPEAN UNION AGENCY FOR RAILWAYS. *What is ERTMS and how does it work?* [online]. 2024. [cit. 2024-03-07]. Dostupné z: https://transport.ec.europa.eu/transport-modes/rail/ertms/what-ertms-and-how-does-it-work_en.
2. PŘINESDOMOVÁ, Lucie. *Na koridorech do roku 2025 jen vlaky s ETCS. Jednotný systém zabezpečení nově funguje na dalších tratích – Ekonomický deník* [online]. 2021. [cit. 2024-03-03]. Dostupné z: https://ekonomickydenik.cz/na-koridorech-do-roku-2025-jen-vlaky-s-etcs-jednotny-system-zabezpeceni-nove-funguje-na-dalsich-tratich/.
3. EUROPEAN UNION AGENCY FOR RAILWAYS. *ETCS Driver Machine Interface v4.0.0* [online]. 2023. [cit. 2024-03-20]. Dostupné z: https://era.europa.eu/system/files/2023-12/index006_-_ERA_ERTMS_015560_v400.zip.
4. EUROPEAN UNION AGENCY FOR RAILWAYS. *Subsystems and Constituents of the ERTMS* [online]. 2024. [cit. 2024-03-07]. Dostupné z: https://transport.ec.europa.eu/transport-modes/rail/ertms/what-ertms-and-how-does-it-work/subsystems-and-constituents-ertms_en.
5. RAIL EXPRESS. *A better bet for balise* [online]. 2019. [cit. 2024-03-09]. Dostupné z: https://railexpress.com.au/a-better-bet-for-balise/.
6. RAILWAYSIGNALLING.EU. *The ERTMS/ETCS signalling system* [online]. 2013. [cit. 2024-03-09]. Dostupné z: https://railwaysignalling.eu/wp-content/uploads/2016/09/ERTMS_ETCS_signalling_system_revF.pdf.
7. RAILSYSTEM. *Balise* [online]. 2024. [cit. 2024-03-09]. Dostupné z: https://railsystem.net/balise/.
8. STEJSKAL, Jan. *ETCS – DMI display*. 2022. Dostupné také z: https://dspace.cvut.cz/handle/10467/101907. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
9. EUROPEAN UNION AGENCY FOR RAILWAYS. *ETCS Driver Machine Interface v2.3.0* [online]. 2009. [cit. 2024-03-07]. Dostupné z: https://era.europa.eu/system/files/2022-11/index034_-_era_ertms_015560_v23.zip.
10. STŘÍTESKÝ, Petr. *Generický návrh provozních displejů drážních vozidel*. 2021. Dostupné také z: https://dspace.cvut.cz/handle/10467/94818. Bakalářská práce. České vysoké učení technické v Praze, Fakulta dopravní.
11. CEDALO. *Eclipse Mosquitto* [online]. 2024. [cit. 2024-03-09]. Dostupné z: https://mosquitto.org/.

12. UDAVICHENKA, Yury. *ETCS – DMI display II*. 2023. Dostupné také z: https://dspace.cvut.cz/handle/10467/109543. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
13. MĚŠŤAN, Ondřej. *ETCS DMI II – Implementace rozdílů na lokomotivách provozovaných v České republice oproti normě ETCS*. 2023. Dostupné také z: https://dspace.cvut.cz/handle/10467/109655. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
14. MONTIEL, Ivan. *Low Coupling, High Cohesion* [online]. 2018. [cit. 2024-03-19]. Dostupné z: https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6.
15. REFSNES DATA. *C++ Polymorphism* [online]. 2024. [cit. 2024-03-19]. Dostupné z: https://w3schools.com/cpp/cpp_polymorphism.asp.
16. VESELÝ, Ondřej. *ETCS – Aktualizace a nová architektura komponenty RBC*. 2024. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
17. BARNEY, Nick; NOLLE, Tom. *Service-oriented architecture (SOA)* [online]. 2023. [cit. 2024-03-22]. Dostupné z: https://techtarget.com/searchapparchitecture/definition/service-oriented-architecture-SOA.
18. SEETHARAMUGN. *The Complete Guide to Event-Driven Architecture* [online]. 2023. [cit. 2024-03-22]. Dostupné z: https://medium.com/@seetharamugn/the-complete-guide-to-event-driven-architecture-b25226594227.
19. THORBEN. *Design Patterns Explained – Dependency Injection with Code Examples* [online]. 2024. [cit. 2024-03-23]. Dostupné z: https://stackify.com/dependency-injection.
20. MALÝ, Martin. *Protokol MQTT: komunikační standard pro IoT* [online]. 2016. [cit. 2024-04-02]. Dostupné z: https://root.cz/clanky/protokol-mqtt-komunikacni-standard-pro-iot/.
21. SDL COMMUNITY. *Simple DirectMedia Layer* [online]. 2024. [cit. 2024-03-07]. Dostupné z: https://libsdl.org/.
22. GOMILA, Laurent. *SFML* [online]. 2024. [cit. 2024-03-07]. Dostupné z: https://sfml-dev.org/.
23. MARIĆ, Predrag. *The Observer Pattern in Java* [online]. 2024. [cit. 2024-04-15]. Dostupné z: https://baeldung.com/java-observer-pattern.
24. AVENWEDDE, Stephan. *Document your source code with Doxygen on Linux* [online]. 2022. [cit. 2024-04-19]. Dostupné z: https://opensource.com/article/22/5/document-source-code-doxygen-linux.
25. MARAK, Sylvia. *What is Anti-Aliasing? Ultimate Guide* [online]. 2024. [cit. 2024-04-17]. Dostupné z: https://selecthub.com/resources/what-is-anti-aliasing/.
26. GOOGLETEST. *GoogleTest Primer* [online]. 2024. [cit. 2024-05-05]. Dostupné z: https://google.github.io/googletest/primer.html.
27. BASUMALLICK, Chiradeep. *What is Unit Testing? Top Tools and Best Practices* [online]. 2022. [cit. 2024-05-05]. Dostupné z: https://spiceworks.com/tech/devops/articles/what-is-unit-testing.
28. GEEKSFORGEEKS. *Calling virtual methods in constructor/destructor in C++* [online]. 2024. [cit. 2024-05-08]. Dostupné z: https://geeksforgeeks.org/calling-virtual-methods-in-constructordestructor-in-cpp/.
29. MICROSOFT. *Visual Studio C/C++ IDE a kompilátor pro Windows* [online]. 2024. [cit. 2024-04-15]. Dostupné z: https://visualstudio.microsoft.com/cs/vs/features/cplusplus/.

30. MARHA. *VcXsrv Windows X Server download* [online]. 2024. [cit. 2024-05-06]. Dostupné z: https://sourceforge.net/projects/vcxsrv/.

31. WANG, Mianzhi. *Setting Up WSL with Graphics and Audio – Mianzhi Wang* [online]. 2017. [cit. 2024-05-06]. Dostupné z: https://research.wmz.ninja/articles/2017/11/setting-up-wsl-with-graphics-and-audio.html.

Obsah přiloženého archivu

Zde se nachází popis přiloženého archivu:

- `build` — spustitelné soubory aplikace pro Windows a Linux
  - `linux` — spustitelné soubory pro Linux
  - `windows` — spustitelné soubory pro Windows
  - `README.txt` — popis jednotlivých binárních souborů
- `src` — zdrojové soubory práce
  - `impl` — všechny zdrojové kódy praktické části práce
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `thesis.pdf` — text práce ve formátu PDF
- `README.txt` — stručný popis obsahu média