FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	ETCS - Studie řízení projektu a kontroly kvality výstupů
Student:	Martin Čáslavský
Vedoucí:	Ing. Jan Matoušek
Studijní program:	Informatika
Obor / specializace:	Manažerská informatika 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.

Cílem této práce je analýza současných procesů vývoje, ať už z hlediska plánování, či kontrolování celého postupu a implementace aplikací, které mají za cíl ulehčit práci vývojářům a budoucím vedoucím týmů.
Těmito pomůckami jsou jednak komponenta JRU, která bude o rozšířena o logovací systém, a skript na kontrolu odpracovaného času v rámci předmětů SP1 a SP2.

Pokyny pro vypracování
1) Analyzujte současný stav projektu, zaměřte se na projektové řízení.
2) Analyzujte způsob promýšlení strategií projektu simulátoru.
3) Analyzujte interní i externí procesy v rámci týmů pracujících na tomto projektu.
4) Navrhněte a zaveďte vylepšení, případně nové procesy řízení projektu.
5) Navrhněte a použijte, případně vytvořte nové podpůrné nástroje; v rámci tohoto bodu se zaměřte především na vylepšení záznamů z testování vyvíjeného systému a získávání informací o zapojení členů týmu.
6) Nově zavedené postupy ověřte na novém týmu v rámci běhu předmětu SP1 v letním semestru 2023/2024.
7) Zjištěné poznatky shrňte a navrhněte další pokračování.

Elektronicky schválil/a Ing. David Buchtela, Ph.D. dne 25. ledna 2024 v Praze.

Bakalářská práce

ETCS – STUDIE ŘÍZENÍ PROJEKTU A KONTROLY KVALITY VÝSTUPŮ

Martin Čáslavský

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

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2024 Martin Čáslavský. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Čáslavský Martin. *ETCS – Studie řízení projektu a kontroly kvality výstupů.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Poděkování viii

Prohlášení ix

Abstrakt x

Seznam zkratek xi

1 Úvod 1

2 Teoretická část 3

2.1 Uvedení do problematiky ETCS . . . 3
2.1.1 Historie . . . 3
2.1.2 Aplikační úrovně . . . 4
2.1.3 Výhody . . . 4
2.1.4 Části ETCS . . . 5
2.2 Projektové řízení v softwarovém vývoji . . . 6
2.2.1 Základní definice v projektovém řízení . . . 6
2.2.2 Podmínky úspěšného projektu . . . 9
2.2.3 Způsoby projektového řízení . . . 11
2.3 Logovací systémy . . . 13
2.3.1 Log . . . 13
2.3.2 Zobrazení logů . . . 15

3 Projektové řízení 17

3.1 Analýza projektového řízení simulátoru ETCS . . . 17
3.1.1 SWOT analýza aktuálního stavu . . . 17
3.1.2 Komunikace mezi týmy . . . 19
3.1.3 Střet školních předmětů s projektem . . . 20
3.1.4 Procesy v rámci projektu . . . 20
3.1.5 Požadavky pro podpůrné nástroje týmu . . . 22
3.1.6 Strategie projektu ETCS simulátoru . . . 23
3.2 Návrh pracovních postupů v systému správy verzí . . . 24
3.2.1 Zvolený postup práce s Git . . . 24
3.2.2 Správa úkolů . . . 26
3.3 Návrh skriptu na měření času . . . 28
3.4 Návrh lepšího řešení projektového řízení . . . 29
3.4.1 SWOT analýza budoucího stavu . . . 29
3.4.2 Zbavení se závislosti na předmětech SP1 a SP2 . . . 29
3.4.3 Rozšíření týmové struktury . . . 30
3.4.4 Povinnosti projektového manažera na úrovni vedení . . . 32
3.4.5 Povinnosti vedoucích jednotlivých týmů . . . 34
3.5 Implementace postupů do webové aplikace GitLab . . . 35
3.5.1 Implementace práce s Gitem . . . 35

3.5.2 Implementace správy úkolů . . . 36
3.6 Implementace skriptu na měření času . . . 40
3.6.1 Získávání dat z webové aplikace GitLab . . . 40
3.6.2 Vytvořené funkce . . . 40
3.6.3 Hlavní průchod skriptem . . . 42
3.6.4 Používání skriptu . . . 43

4 Komponenta JRU **47**
4.1 Analýza stavu vývoje ETCS simulátoru . . . 47
4.1.1 DMI . . . 47
4.1.2 EVC . . . 48
4.1.3 RBC . . . 48
4.1.4 Komunikace mezi komponentami . . . 48
4.2 Analýza komponenty JRU . . . 49
4.2.1 Požadavky na komponentu JRU . . . 49
4.2.2 Požadavek na JRUViewer . . . 50
4.3 Návrh komponenty JRU . . . 51
4.3.1 Logovací systém . . . 51
4.3.2 Pomocné třídy . . . 53
4.4 Návrh JRUViewer . . . 54
4.4.1 Rozložení obrazovky . . . 54
4.4.2 Architektura . . . 56
4.5 Návrh komunikace mezi JRU a JRUViewer . . . 58
4.5.1 TCPServer . . . 58
4.5.2 TCPClient . . . 58
4.5.3 Formát zpráv . . . 58
4.6 Implementace JRU . . . 59
4.6.1 InternalStateMessage . . . 59
4.6.2 JRULoggerService . . . 62
4.6.3 TimeStamp . . . 63
4.6.4 TimerService . . . 63
4.6.5 SaveLogService . . . 64
4.6.6 MessageTypeCheckService . . . 64
4.6.7 JRUMessageDataService . . . 65
4.6.8 HeartbeatControlService . . . 66
4.6.9 BrakingCurvesDataService . . . 67
4.6.10 TCPServer . . . 67
4.7 Implementace JRUViewer . . . 67
4.7.1 MainWindow . . . 67
4.7.2 Manager . . . 68
4.7.3 QModel . . . 69
4.7.4 Dialog . . . 69
4.7.5 Widget . . . 70
4.7.6 Worker . . . 70
4.7.7 Repository . . . 70
4.7.8 TCPClient . . . 70
4.8 Testování . . . 71
4.8.1 Interakce s ostatními komponentami . . . 71
4.8.2 Zobrazování logů . . . 71
4.9 Návrh na budoucí rozšíření . . . 72

5 Závěr **73**

A Používání logovacího systému 75

B Používání aplikace JRUViewer 77

Obsah příloh 83

Seznam obrázků

2.1 Eurobalíza v kolejnici. Získáno z [8] 5
2.2 Obrázek znázorňující rozsahový trojúhelník z [10] 8
2.3 Sekvence iterací v rámci vodopádové metody 11
2.4 Popis metodiky Scrum z SWI přednášky [15] 12

3.1 Životní cyklus úkolu 26
3.2 Štítky priority ve webové aplikaci GitLab 36
3.3 Štítky náročnosti ve webové aplikaci GitLab 37
3.4 Štítky stavu ve webové aplikaci GitLab 37
3.5 Štítky typu ve webové aplikaci GitLab 38
3.6 Nepovinné štítky ve webové aplikaci GitLab 39
3.7 Nastavení plánovače spouštění pipeline 44
3.8 Výsledky timetracking skriptu ve webové aplikaci GitLab 45

4.1 Průchod logu skrz JRU 53
4.2 Návrh JRUViewer v aplikaci Qt Designer 55
4.3 JRUViewer architektura 57

B.1 Tlačítko sloužící pro připojení k JRU serveru 77
B.2 Ukázka stavu jednotlivých komponent 77
B.3 Barevné rozlišení zpráv, které JRUViewer zobrazuje 78
B.4 Obrazovka JRUViewer během používání 79

Seznam tabulek

3.1 SWOT analýza projektu ETCS simulátoru 19
3.2 Větve pro repositáře v ETCS projektu 26
3.3 SWOT analýza projektu budoucího stavu 29

4.1 Výčtový typ `MessageType` 51
4.2 Ukázka použití formátu zpráv pro TCP komunikaci 59
4.3 Stručný popis proměnných ve třídě `InternalStateMessage` 59

Seznam výpisů kódu

3.1 Funkce get_time_from_issue 40
3.2 Funkce parse_time_tracking_string 41
3.3 Funkce check_date 41
3.4 Funkce get_issues_from_date 41
3.5 Funkce get_time_value 41
3.6 Funkce print_time_track 42
3.7 Funkce get_statistics 42
3.8 Definování úkolu v .gitlab-ci.yml 43

4.1 Všechny varianty konstruktorů pro `InternalStateMessage` 60
4.2 Metody `AddArgument` pro parsování argumentů v `InternalStateMessage` 60
4.3 Metoda pro deserializaci v `InternalStateMessage` 61
4.4 Metoda pro formátování textu v `InternalStateMessage` 61
4.5 Metody `Log()` pro logování zpráv v `JRULoggerService` 62
4.6 Vytvoření definici `DEBUG` pro logování do konzole v *CMakeLists.txt* 63
4.7 Metoda pro ukládání zpráv do souboru v `SaveLogService` 64
4.8 Metoda pro filtrování zpráv v `MessageTypeCheckService` 65
4.9 Metoda pro přidání zprávy v `JRUMessageDataService` 65
4.10 Metody pro získání dat z `JRUMessageDataService` 66
4.11 Metoda pro kontrolu stavu komponent `GetComponentStateInString` 66
4.12 Deklarace třídy `MainWindow` v JRUViewer 68
4.13 Možná podoba zástupce skupiny *Managers* 69

A.1 Jednoduché vytvoření logu 75
A.2 Porovnání možností pro výpis jména funkce v logu 76
A.3 Logování aktuální rychlosti 76

Na začátku bych chtěl vyjádřit své upřímné díky svému vedoucímu, Ing. Janu Matouškovi, za jeho skvělé vedení práce a přístup k celému projektu. Dále děkuji panu doc. Ing. Martinu Lesovi za cenné konzultace v průběhu práce a za jeho pomoc při vysvětlování některých odborných pojmů. Rád bych rovněž poděkoval všem svým kolegům z projektu ETCS za jejich snahu a ochotu. Zvláštní uznání patří Ondřeji Veselému a Tereze Neprašové, kteří v rámci svých bakalářských prací odvedli mimořádnou práci. Nakonec bych chtěl vyjádřit vřelé díky mé rodině a přítelkyni za jejich trpělivost a nekonečnou podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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 16. května 2024

Abstrakt

Cílem této práce je zanalyzovat projektové řízení simulátoru ETCS a implementovat podpůrné programy pro usnadnění vývoje. Práce se soustředí na identifikaci aktuálních problémů v řízení projektu a jejich následnou analýzu s cílem najít efektivní řešení. Mezi implementované podpůrné programy patří skript na měření odpracovaného času a rozšířená komponenta JRU s vlastním logovacím systémem. Tato aplikace, včetně grafického uživatelského rozhraní, má potenciál pomoci budoucím studentům při vývoji nových funkcionalit. Práce tak přináší nový pohled na projekt a otevírá otázky týkající se jeho budoucnosti.

Klíčová slova ETCS, implementace Juridical Record Unit, logovací systém, studie projektového řízení, C++, Qt

Abstract

The goal of this thesis is to analyze the project management of the ETCS simulator and implement supportive programs to make development easier. The thesis focuses on identifying current issues in project management and analyzing them to find effective solutions. The implemented support programs include a script for time tracking and an extended JRU component with its own logging system. This application, including the graphical user interface, has the potential to assist future students in developing new functionalities. The thesis brings a new perspective to the project and raises questions regarding its future.

Keywords ETCS, implementation of Juridical Record Unit, logging system, study of project management, C++, Qt

Seznam zkratek

API Application Programming Interface
BTM Balise Transmission Module
CEM Common Enums and Messages
CI/CD Continuous Integration / Continuous Deployment
ČVUT České vysoké učení technické
DDOS Distributed Denial-Of-Service
DMI Driver Machine Interface
ERTMS European Railway Traffic Management System
ETCS European Train Control System
EVC European Vital Computer
FIT Fakulta informačních technologií
GSM-R Global System For Mobile Communications - Railways
GUI Graphic User Interface
IoT Internet of Things
IT Informační Technologie
JRU Juridical Recording Unit
LPC Lektorské pracoviště
L1 Level 1
L2 Level 2
L3 Level 3
MA Movement Authority
MVC Model-View-Controller
MQTT MQ Telemtry Transport
ODO Odometrické snímače
OS Operační systém
PM Projektový manažer
PPR Projektové řízení
RBC Radio Block Center
SDL Simple DirectMedia Layer
SMART Specific, Measurable, Achievable, Realistic, Timely
SP1 Softwarový týmový projekt 1
SP2 Softwarový týmový projekt 2
STM Specific Transmission Module
SW Software
SWI Softwarové inženýrství
SWOT Strenghts, Weaknesses, Opportunities, Threats
TIU Train Interface Unit
UI User Interface

Kapitola 1
Úvod

V České republice je více než 9000 km železniční trati [1]. Jedná se tak o jednu z nejhustších infrastruktur v rámci celé Evropy. V dnešním technologickém světě je dán velký důraz na zabezpečení evropské železnice a tuto problematiku Evropská unie řeší pomocí společného evropského vlakového zabezpečovacího systému – ETCS.

Tento systém má vejít v platnost na českých dráhách v blízké době a spolu se zavedením bude potřeba provést školení strojvedoucích. Pro tyto potřeby je vhodné vytvořit simulátor, který bude věrně implementovat chování tohoto zabezpečovacího systému.

V rámci ČVUT v Praze se ve spolupráci Fakulty informačních technologií a Fakulty dopravní tento simulátor vyvíjí a tato bakalářská práce je součástí celého projektu.

Výsledek této práce je především určen pro studenty, kteří budou v budoucnu pracovat na tomto projektu. Bude jim sloužit jako zdroj pro analýzu a využívání nových částí simulátoru, na které se tato práce zaměřuje. Dále se práce může hodit pro zlepšení řízení celého projektu a ke zvýšení pravděpodobnosti jeho úspěšného zakončení.

Struktura práce

Na začátku práce se nachází „Teoretická část“, která má za cíl představit čtenáři základní pojmy potřebné k pochopení dalších částí této práce. Navazují na ní dvě hlavní kapitoly „Projektové řízení“ a „Komponenta JRU“.

Cílem kapitoly o projektovém řízení je navrhnout vylepšení stávajících procesů nebo případně vytvořit nové procesy pro efektivní řízení projektu. Pro dosažení tohoto cíle je nezbytné provést analýzu současného stavu, jak interních, tak externích procesů týmu, a pečlivě promyslet strategii celého projektu. Některé procesy byly již nově implementovány během psaní bakalářské práce, a tato kapitola bude reflektovat výsledky jejich používání.

Následující kapitola o komponentě JRU má několik cílů, přičemž hlavním z nich je implementovat novou komponentu do ETCS simulátoru podle specifikací zákazníka a týmu. Stejně jako v předchozí kapitole je pro dosažení tohoto cíle nezbytná důkladná analýza současného stavu všech komponent, aby bylo možné navrhnout a implementovat novou komponentu v souladu s celkovým projektem. Kromě implementace oficiálních funkcí bude komponenta JRU rozšířena o vlastní logovací systém, který má usnadnit práci vývojářům v budoucnosti. Kapitola také poskytne informace o implementaci nových funkcí a provedených testech.

V závěrečné kapitole dojde k zhodnocení celé práce a nastínění dalších kroků, které mohou celému projektu pomoct.

Návaznost práce

Tato práce je součástí většího celku v rámci projektu ETCS simulátoru, ke kterému již v minulosti vznikaly bakalářské práce. Tento projekt běží od roku 2021, kdy do něho nastoupili první studenti v rámci předmětu SP1. Cílem od začátku bylo co nejdříve vytvořit funkční prototyp, který by bylo možné ukazovat potencionálním zákazníkům.

Z počátku byl projekt rozkouskován na vícero týmů, kdy byl každý tým zodpovědný pouze za jednu část celého simulátoru. To vedlo k mnoho odlišnostem v implementaci, které celý projekt zpomalily. V letním semestru 2023 došlo ke sjednocení vícero částí pod jeden jediný tým, jehož jsem byl já součástí. V týmu jsme většinu času museli věnovat sjednocování komponent. Simulátor nebyl většinu času funkční a až před začátkem letních prázdnin došlo k jeho zprovoznění. Tyto zkušenosti mě přivedly k myšlence, že je potřeba projekt lépe řídit nejen ze strategického pohledu.

Kapitola 2
Teoretická část

Tato část bakalářské práce se věnuje teoretickému základu. Cílem je vysvětlit a definovat klíčové pojmy, které budou důležité pro další části práce. Slouží jako pomoc pro ty, kdo nemají hlubší povědomí o tématech, kterými se práce zabývá.

2.1 Uvedení do problematiky ETCS

ETCS (European Train Control System) představuje evropský zabezpečovací systém vlaků, který je součástí ERTMS (European Railway Traffic Management System). Cílem ERTMS je sjednotit vlakové zabezpečovací systémy napříč Evropou. Zavedení ERTMS může vést k vytvoření bezproblémového železničního systému a zvýšit konkurenceschopnost ve srovnání se zbytkem světa. Vedle ETCS je součástí ERTMS také GSM-R (Global System For Mobile Communications – Railways), které zajišťuje komunikaci mezi vlakem a tratí. [2]

V rámci Evropské unie existuje více než 20 různých vlakových zabezpečovacích systémů, což vyžaduje, aby byly vlaky vybaveny různými systémy pro zajištění bezpečnosti na mezinárodních trasách. Jednotný zabezpečovací systém, jako je ERTMS, má za úkol nahradit tyto systémy a zefektivnit tak evropský železniční provoz. [2]

2.1.1 Historie

Historie ERTMS a tedy i ETCS sahá do začátku devadesátých let minulého století. V roce 1990 ERRI[1] sestavil skupinu železničních expertů, jejichž hlavním cílem bylo primárně sestavit základní požadavky pro jednotný evropský zabezpečovací systém. Došlo k vydefinování požadavků pro nové palubní zařízení (EUROCAB), systém pro přenos dat (EUROBALISE) a nový kontinuální přenosový systém (EURORADIO). [3]

V roce 1995 definovala Evropská komise strategie dalšího rozvoje ERTMS s důrazem na přípravu budoucí implementace v evropské železniční síti. S cílem dokončit kompletní specifikace vznikla v roce 1998 skupina s názvem UNISIG. [3]

25. dubna 2000, bylo oficiálně uvedeno ERTMS, čímž se poskytla železnici vyšší úroveň výkonu. Specifikace, které v tento den byly uvedeny do provozu, však bylo nutné časem revidovat, aby lépe odpovídaly potřebám železničního provozu a obsahovaly nové funkcionality.. To vedlo k vytvoření nové specifikace 2.3.0d, která byla schválena Evropskou komisí v dubnu 2008. [3]

V roce 2017 byl vydán soubor specifikací ERTMS Baseline 3 verze 2, který se stal dalším klíčovým milníkem ve vývoji ERTMS. Tato verze je vyspělejší a stabilnější oproti předchozí verzi 2.3.0d. Ve stejném roce se členové UNISIG opětovně zavázali k rozvoji ERTMS. Cílem tohoto

[1] Evropský železniční výzkumný ústav

závazku je zajistit stabilitu specifikací ERTMS po ERTMS Baseline 3 verze 2 a podpořit rychlé a koordinované zavádění v celé Evropě. Členské státy Evropské unie byly vyzvány k pravidelnému vypracovávání národních prováděcích plánů, které by měly být aktualizovány minimálně jednou za pět let. [3]

V současnosti je určitě potřeba zmínit, že i přesto, že je ERTMS primárně zaměřený na Evropu, jeho úspěch je celosvětový. ERTMS se povedlo prosadit jako světový standard, který se mimo jiné využívá třeba v Číně, Taiwanu, Mexiku, Novém Zélandu, Austrálii a v dalších státech. [3]

2.1.2 Aplikační úrovně

ERTMS se neustále vyvíjí a s každou verzí mohou být funkcionalita a úrovně odlišné. Ve verzi 2.3.0d, jejíž specifikace byla použita jako základ pro většinu implementace ETCS simulátoru, existují různé úrovně, z nichž nejvýznamnější jsou L1 (Level 1), L2 (Level 2) a L3 (Level 3):

L0 traťová část ETCS a ani traťová část národního zabezpečovacího systému není dostupná. Slouží pro kontrolu maximální konstrukční rychlosti vlaku a národní maximální rychlosti K udělení oprávnění k jízdě slouží návěstidla. [4, s. 12]

LSTM traťová část ETCS opět není k dispozici, vlak však může používat traťovou část národního zabezpečovacího systému pomocí STM (Specific Transmission Module) modulu. Detekce a integrita vlaku je prováděna zařízením mimo ETCS. [4, s. 14]

L1 je základní úroveň celého zabezpečovacího systému, kde významnou úlohu mají tzv. balízy (kapitola 2.1.4.1.2). Tato statická zařízení, která se nachází v kolejnici, slouží pro předávání oprávnění k jízdě a popisu trati. Z daného popisu vlak zjišťuje maximální povolenou rychlost, kterou poté kontroluje. [4, s. 16]

L2 rozšiřuje předchozí úroveň o RBC (Radio Block Center). Konkrétní funkčnost této komponenty je vysvětlena v sekci 2.1.4.1.1. Důležité je však říct, že díky této komponentě má vlak neustále aktuální informace o trati a povolení k jízdě, není tedy potřeba informace získávat z dalších balíz. [5]

L3 navazuje na úroveň předchozí. Hlavní změnou v této úrovni, je primárně to, že integrita vlaku je kontrolována přímo ve vlaku samotným. Díky tomu není potřeba mít obvody a vlak může jet bez blokování. [5]

2.1.3 Výhody

Zvýšená kapacita přepravy osob i zboží na již existujících linkách. Díky ERTMS může železniční doprava dobře reagovat na větší poptávku. ERTMS umožňuje zkrátit bezpečné rozestupy mezi vlaky a tím zvýšit kapacitu na dosavadní infrastruktuře až o 40 %. [6]

Vyšší rychlost v dopravě. Díky ERTMS můžou vlaky jet až rychlostí 500 km/h. [6]

Vyšší míra spolehlivosti na všech linkách. Díky jednotnému zabezpečovacího systému ETCS se může výrazně zvýšit spolehlivost a přesnost. Tyto dva prvky jsou klíčové nejen pro osobní, ale i nákladovou dopravu. [6]

Nižší výrobní náklady díky jednotnému systému. Není potřeba vytvářet, udržovat a instalovat více systémů, tím se rapidně sníží náklady. [6]

Snížení nákladů na údržbu díky ERTMS. Vzhledem k tomu, že od úrovně 2 již není potřeba traťová signalizace, může se ušetřit na nákladech na údržbu. [6]

Otevřený trh dodávek umožní zákazníkům nakupovat zařízení pro instalaci kdekoli v Evropě. Existuje šest dodavatelů ERTMS, kteří mohou vyrábět traťové i palubní zařízení. [6]

Zkrácení doby realizace zakázky díky jednotnému systému. [6]

Zjednodušení schvalovacího procesu v Evropě a výrazné snížení nákladů na certifikaci, která tradičně souvisí se zaváděním nového systému. [6]

Zlepšení bezpečnosti pasažérů je u zabezpečovacího systému zásadní. Ověřený, jednotný systém ERTMS je garancí bezpečnosti. [6]

2.1.4 Části ETCS

ETCS se rozděluje do dvou podskupin – traťová a vozidlová část. V této sekci bych rád popsal obě dvě části podrobněji.

2.1.4.1 Traťová část

Traťová část, jak už název napovídá, se skládá z komponent, které jsou spjaté s kolejnicí. Nejsou tedy součástí vlaku samotného. Pro náš simulátor jsou významný především RBC a balíza:

2.1.4.1.1 RBC (Radio Block Center) je základem traťové části ETCS od úrovně 2. Je zodpovědné za bezpečnost všech vlaků jedoucích ve spravované oblasti, se kterými byla navázána komunikace pomocí GSM-R. RBC odesílá vlakům tzv. MA (Movement Authority), což jsou povolení k jízdě, které získává na základě informací o trati. [7]

Díky komunikaci s vlakem, má RBC povědomí o jeho poloze, kterou si pravidelně ukládá do své databáze. Tímto způsobem je každý vlak monitorován a je pod dohledem prakticky kdykoli. [7]

2.1.4.1.2 Balíza je zařízení instalované na trati, které vysílá zprávy vlakům vybaveným ETCS. Mezi její hlavní funkce patří poskytování dat zařízením, které přes ni projíždí. Mezi tyto data patří rychlostní omezení, údaje o poloze či sklon koleje. Na obrázku 2.1 je vidět podoba balízi v kolejnici. [7]

Obrázek 2.1 Eurobalíza v kolejnici. Získáno z [8]

2.1.4.2 Vozidlová část

Všechny níže zmíněné části se už nachází ve vlaku a proto mluvíme o tzv. vozidlové části. Mezi nejdůležitější části patří:

2.1.4.2.1 BTM (Balise Transmission Module) je bezdrátová čtečka balíz, která se nachází na spodku vlaku. Kromě čtení informací z balíz slouží také k napájení těchto krabiček na kolejnici. [7]

2.1.4.2.2 DMI (Driver Machine Interface) je rozhraní mezi ETCS systémem a strojvedoucím. Slouží k zobrazování důležitých údajů od systému směrem k strojvedoucímu a k získávání informací od strojvedoucího. Má několik podob, může být na dotykovém displeji, ale i na displeji s pomocnými tlačítky. [7]

2.1.4.2.3 EVC (European Vital Computer) je palubní počítač, který je zodpovědný za chod vlaku. Bezpečně zpracovává všechny informace, které může získat jak z tratě, tak z vlaku samotného, ale i od strojvedoucího, s kterým komunikuje pomocí rozhraní DMI. [7]

2.1.4.2.4 JRU (Juridical Recording Unit) je ve své podstatě černou skříňkou vlaku. Zaznamenává informace o prováděných činnostech a různých stavech vlaku. [7]

2.1.4.2.5 ODO (odometrické snímače) se používají k určení polohy vlaku, především odesláním ujeté vzdálenosti a aktuální rychlosti vlaku. [7]

2.1.4.2.6 TIU (Train Interface Unit) je rozhraní mezi ETCS a lokomotivou. Umožňuje výměnu informací a udělování příkazů od EVC (například příkaz k brzdění). [7]

2.2 Projektové řízení v softwarovém vývoji

V této sekci bych rád nahlédl do teorie projektového řízení v softwarovém vývoji. Podívám se na samotnou definici projektového řízení, pokusím se popsat různé metody a nástroje pro řízení projektu, ale i jeho plánování. Tato kapitola slouží jako teoretická předloha pro analytické popsání současného stavu ETCS simulátoru z hlediska projektového řízení.

2.2.1 Základní definice v projektovém řízení

Projektové řízení jako takové není nikde přesně vydefinováno. Během provádění rešerše se mi nepovedlo najít vícero zdrojů, které by se shodovali ve všech věcech. Proto jsem z vlastního uvážení vybral definice, které nejlépe odpovídají mému pochopení této problematiky.

2.2.1.1 Projekt

Definice projektu není jednoduchou úlohou. V internetových i v knižních zdrojích se vyskystuje velké množství různých definic. Tato rozmanitost je patrná i z přednášky z předmětu PPR (Projektové řízení) [9], kde jsou prezentovány tři různé definice projektu:

Projekt *je jedinečný celek koordinovaných činností se stanoveným výchozím a konečným bodem, realizovaných jednotlivcem, nebo organizací s cílem dosažení určených časovým, nákladovým a výkonovým rozvrhem. (IPMA)*

Projekt *je úsilí, v rámci kterého dochází k organizování lidských, materiálových a finančních zdrojů, zaměřené na provedení určeného rozsahu práce při vymezených nákladech a čase, za účelem dosažení prospěšných změn. (PMI)*

Projekt *je jedinečný proces sestávající z řady koordinovaných a řízených činností s daty zahájení a ukončení, prováděný pro dosažení cíle, který vyhovuje specifickým požadavkům, včetně omezení daným časem, náklady a zdroji. (ISO)*

Mně osobně připadá jako nejlepší definice ta, která říká, že projekt je posloupnost několika činností, které směřují k jednotnému cíli. Musí být dokončen v daném termínu, s omezenými zdroji a podle specifikace, která je vydefinována před startem celého projektu. [10]

Tyto činnosti zpravidla bývají jedinečné, komplexní a úzce propojené. Jedinečnost činností je dána především tím, že projekt je dlouho trvající a může být tak do velké míry ovlivněn okolím: pandemie, ekonomická krize atd. [10]

Komplexnost činností v projektu se projevuje v tom, že nejsou rutinní a stereotypní, nýbrž vyžadují často kreativní a flexibilní přístup. Plnění těchto úkolů vyžaduje hlubší úvahy a analytické myšlení. Nicméně je třeba zdůraznit, že i přesto mohou existovat v projektu určité jednoduché a opakující se činnosti, jako jsou pravidelné schůzky. [10]

Propojené činnosti jsou velmi logickou vlastností projektu. Ten by měl totiž směřovat k jednotnému cíli a je pravděpodobné, že různé činnosti, které mají ke stejnému cíli směřovat, budou propojené. [10]

2.2.1.2 Rozsahový trojúhelník

Trojimperativ projektu popisuje vztah mezi rozsahem projektu, dobou trvání a náklady. Právě na něm je patrné, že změní-li se jedna ze tří proměnných, musí dojít ke změně i druhé, aby došlo k vyrovnání celého trojúhelníku. Rozsahový trojúhelník staví nad tímto principem, akorát ho rozšiřuje více do detailu. [10]

Rozsah slouží pro určení hranic celého projektu. Pomocí rozsahu lze vydefinovat, co je, ale hlavně i co není součástí projektu. Změny rozsahu mají velký vliv na celý projekt, i proto je tato veličina většinou tou nejdůležitější. V IT světě většinou bývá označován jako funkční specifikace. [10]

Kvalita se rozděluje na dva podtypy. Kvalita produktu a kvalita procesu.

Kvalita produktu primárně určuje, co a v jaké kvalitě bude výstupem celého projektu. Tím může být jak hardware, tak i software. Pro kontrolu kvality existuje velké množství nástrojů a metodik. [10]

Kvalita procesu se týká samotného řízení projektu. Důraz je kladen na to, jak dobře samotné řízení projektu funguje a dochází i k analýze, jak ho lze neustále zlepšovat. [10]

Náklady projektu jsou také velmi důležité. Tyto náklady je nejjednodušší si představit jako rozpočet, který je vhodné stanovit na začátku projektu. Úspěšnému projektu pomáhá, pokud se během analýzy povede odhadnout potencionální náklady co nejpřesněji. Budoucí změna rozpočtu může být velmi náročná, protože nemusí být ochota investovat do projektu více, než bylo původně vydefinováno. [10]

Čas je většinou označen nějakým termínem (často tzv. *deadline*), kdy je potřeba projekt mít dokončen. Je úzce spojen s náklady a to nepřímou úměrností. Chceme-li zkrátit termín dokončení, musí se zákonitě zvednout náklady. [10]

Zdroje jsou v kontextu softwarového projektu primárně zaměstnanci, ale do zdrojů můžou patřit i různá fyzická vybavení, zásoby a nebo placené služby. Do zdrojů je možné zařadit i testovací data. [10]

Rizika nejsou vyloženě součástí trojúhelníku, ale zásadně ovlivňují ostatní. Při řízení projektu je potřeba rizika hlídat a v případě potřeby je eliminovat. [10]

Kromě posledního bodu platí, že jsou všechny ostatní veličiny na sobě závislé. Změní-li se hodnota jedné, může dojít i ke změně některé z dalších (klidně i více), aby došlo k vyvážení celého projektu. [10]

Na obrázku 2.2 je vidět vztah, mezi jednotlivými veličinami. U většiny projektů jsou nejdůležitějším prvkem rozsah a kvalita projektu, které jsou vyznačeny obsahem celého trojúhelníku. Jednotlivé strany pak tvoří čas, náklady a dostupné zdroje. Při přípravě projektu je tedy vhodné tyto tři veličiny vydefinovat, aby byl splněn požadavek zákazníka (nebo kohokoliv jiného) na rozsah a kvalitu výsledku. [10]

Obrázek 2.2 Obrázek znázorňující rozsahový trojúhelník z [10]

V praxi neexistuje projekt, u kterého by nenastala jakákoliv změna těchto veličin. Zákazník si například může rozmyslet rozsah produktu či jeho kvalitu, klíčový člen týmu může celý projekt opustit atd. Jakákoliv z těchto změn může mít negativní vliv na celý systém a projekt. [10]

Aby tyto změny nevedly až k zániku celého projektu, je potřeba prioritizovat jednotlivé proměnné. Tím je myšleno, že je potřeba určit, které z těchto pěti veličin je možno upravovat a které jsou naopak fixní a není vhodné je měnit. Příkladem může být zafixování ceny, což může v důsledku vést k prodloužení doby, nebo snížení zdrojů, atd. [10]

2.2.1.3 Projektové řízení

Projektové řízení je proces využití zdrojů k realizaci projektu. Jedná se o naplánování a kontrolování dílčích cílů, které později vedou k zakončení celého projektu. Častokrát se můžou vyskytovat i termíny jako vedení či řízení projektů, může se používat i anglický výraz *project management*. [11]

Použití projektového řízení v projektu má velké množství výhod. Zásadní výhodou je to, že projektové řízení zvyšuje pravděpodobnost úspěchu celého projektu, zvlášť pokud je aplikováno už před samotným zahájením. Pomocí projektového řízení lze lépe definovat a naplánovat cíle projektu, které lze později lépe kontrolovat. [12]

Díky projektovému řízení jsou jednoznačně přidělené role a zodpovědnosti. Pomocí toho by pak mělo být jasné, kdo má co na starost, včetně klíčové eliminace vzniklých problémů. [12]

2.2.2 Podmínky úspěšného projektu

Aby byl projekt úspěšný, je potřeba mít ho dobře naplánovaný a je potřeba provést důkladnou analýzu problému a všech možných řešení. K zajištění úspěchu existuje i spousta metod a obecných praktik, na které je potřeba při plánování projektu myslet. V této kapitole bych rád jednotlivé metody a praktiky představil.

2.2.2.1 SMART metoda

Tato metoda je jednou z klíčových metod používaných k maximalizaci úspěšnosti projektů. Je navržena k definování jasných a konkrétních cílů, které by měl projekt splnit. Tato metoda je známá pod zkratkou *SMART*, který vychází z anglických slov:

Specific – cíle projektu by měly být konkrétní a přesně definované.

Measurable – cíle projektu by měly být měřitelné, aby se lépe kontroloval výstup projektu.

Achievable – cíle projektu by měly být dosažitelné, především z hlediska nákladů.

Realistic – cíle projektu by měly být realistické a splnitelné.

Timely – cíle projektu by měly být časově měřitelné, měly by mít jasně daný termín. [13]

2.2.2.2 Sestavení kvalitního týmu a určení pravomocí

Projekt jako takový je většinou velmi rozsáhlý a tak je nepravděpodobné, že by na jednom projektu pracoval jen jeden člověk. Je proto potřeba sestavit kvalitní tým, který bude nejlépe odpovídat rozsahu daného projektu. Kvůli tomu, že je každý projekt ze své podstaty jedinečný, nelze ani určit, jak by tým měl být sestaven. [12]

Základním kamenem každého týmu by měl být nějaký projektový manažer, který je zodpovědný za celý projekt a správné používání projektového řízení. Jeho hlavní náplní práce by mělo být plánování dílčích úkolů projektu, kritické rozhodování a reakce na neočekávané události. Měl by mít ponětí o tom, v jakém stavu se projekt momentálně nachází a co ostatní členové týmu dělají. [12]

V týmu je také potřeba vydefinovat pravomoci a zodpovědnosti jednotlivých členů. U menších projektů se pravidelně děje, že veškerou zodpovědnost (i pravomoci) přebírá PM (projektový manažer). Pokud je projekt většího rázu, tak PM častokrát svojí zodpovědnost částečně předává na ostatní členy, kteří jsou například zodpovědný za dílčí úkoly, komponenty, moduly atd. [12]

Pokud jsou takto rozděleny pravomoci a zodpovědnosti, je pak snazší se obracet se změnou či dodefinováním požadavků už na konkrétní lidi a celá práce na projektu se díky tomu může zefektivnit. [12]

2.2.2.3 Sledování aktuálního stavu projektu

Samostatné naplánování projektu, i kdyby bylo sebelepší, bohužel úspěch nezaručuje. Pravidelná kontrola aktuálního stavu je neméně důležitou součástí projektu a projektového řízení. Jenom ta, totiž může odhalit, zda projekt neodbočil od svého definovaného zadání. Díky včasnému odhalení nějakých nesrovnalostí či problémů, může PM či jiná zodpovědná osoba včas zareagovat a tím snížit případné negativní dopady na celý projekt. [12]

V dnešní době existuje velké množství nástrojů, které mají za úkol týmům ulehčit orientaci v projektu. Mezi základní patří různé systémy pro správu úkolů. Existují různé varianty, ať už placené nebo ty, které jsou bezplatné. [12]

Kromě těchto nástrojů, je však také důležité v pravidelných intervalech uspořádat nějakou schůzku, ve kterém dojde k shrnutí aktuálního stavu všem členům týmu. Mimo aktuální stav je

také vhodné si zhodnotit řízení projektu, zda všechny nastavené věci fungují tak, jak mají, nebo zda je potřeba je nějak poupravit. [12]

Pro lehčí sledování projektu je vhodné jeho dílčí cíle shlukovat do několika různých etap, ve kterých bude snazší orientace a úkoly se v těchto etapách budou lépe vyhodnocovat. [12]

2.2.2.4 Dokumentace projektu

Má-li být projekt úspěšný, je potřeba pro něho vést dostačující dokumentaci. Většinou se nejedná pouze o jeden soubor, který popisuje projekt z jedné části. Typů dokumentace totiž může být více a já bych je rád v této sekci popsal:

Projektová dokumentace vzniká buď na začátku a nebo v průběhu celého projektu. Používá se spíše pro interní záležitosti pro vývojáře, produktové manažery, designery atd. Jedná se o dokument pravdy, ve kterém mají být definovány veškeré požadavky, plány a technický návrh. V průběhu celého projektu by mělo docházet k pravidelné kontrole, že implementace se nachází v rámci mezí, které tento dokument stanovil. [14]

Produktová dokumentace se zaměřuje na vytvořený software. Slouží uživatelům, aby lépe porozuměli danému produktu. Může se jednat o různé instalační, administrátorské a uživatelské příručky. [14]

Procesní dokumentace slouží opět primárně k internímu používání. Má za cíl dokumentovat jednotlivé procesy a dílčí úkoly v rámci celého projektu. Všechny zaznamenané věci se mohou později použít na dalších projektech, což může do budoucna přinést efektivnější práci. Součástí této dokumentace jsou i různé plány. [14]

Technická dokumentace bývá v softwarových projektech nejčastější. Poskytuje informace o technických aspektech celého projektu: architektura softwaru, datové struktury, algoritmy a další technické detaily. Tento typ dokumentace se může hodit, pokud na projekt bude někdo se svojí prací navazovat. Může totiž jednodušeji pochopit všechny potřebné náležitosti. Mezi nejčastější technickou dokumentaci patří: dokumentace API (Application Programming Interface), datového modelu a architektury. [14]

Uživatelská dokumentace úzce souvisí s tou produktovou. Kromě již zmíněných věcí, sem spadají například různé tutoriály a *How-to* příručky. Jejich cílem je zjednodušit všem uživatelům používání daného softwaru. [14]

Častým problémem projektů je, že dokumentace není úplná. Já osobně jsem se několikrát setkal s problémem, že chyběla dokumentace projektová a nebo jednotlivé části z ní. Špatné vydefinování požadavků v tomto dokumentu může vést ke špatnému definování hranic mezi zákazníkem a dodavatelem, a tak může mít každý z nich jinou představu o výsledném produktu. To může vést k neustálému vytváření nových požadavků, které mohou vést až k neúspěchu celého projektu.

2.2.2.5 Podmínky akceptace

Pokud při vytváření projektu byla úspěšně použita metoda SMART (sekce 2.2.2.1), budou všechny dílčí úkoly dobře definovaný a bude jednodušší určit, jestli byly řádně splněny. To pak může ulehčit akceptaci celého projektu. Kromě toho, je také dobré do podmínek určit požadovanou kvalitu, funkční i nefunkční požadavky, které daný projekt zkonkretizují.

Dobře definované podmínky akceptace se dají v průběhu probíhání projektu dobře hlídat, lze reagovat na nějaké odchylky a projekt takový tím bude mít vyšší pravděpodobnost úspěchu.

2.2.3 Způsoby projektového řízení

I přesto, že je každý projekt jedinečný, je zřejmé, že dva různé projekty mohou obsahovat stejné procesy projektového řízení. Díky těmto podobnostem lze vytvořit obecný postup, kterému se říká metodika. Ty mají primárně za cíl usnadnit projektové řízení a vytvořit standart různých postupů v rámci projektu. [15]

2.2.3.1 Klasické metodiky

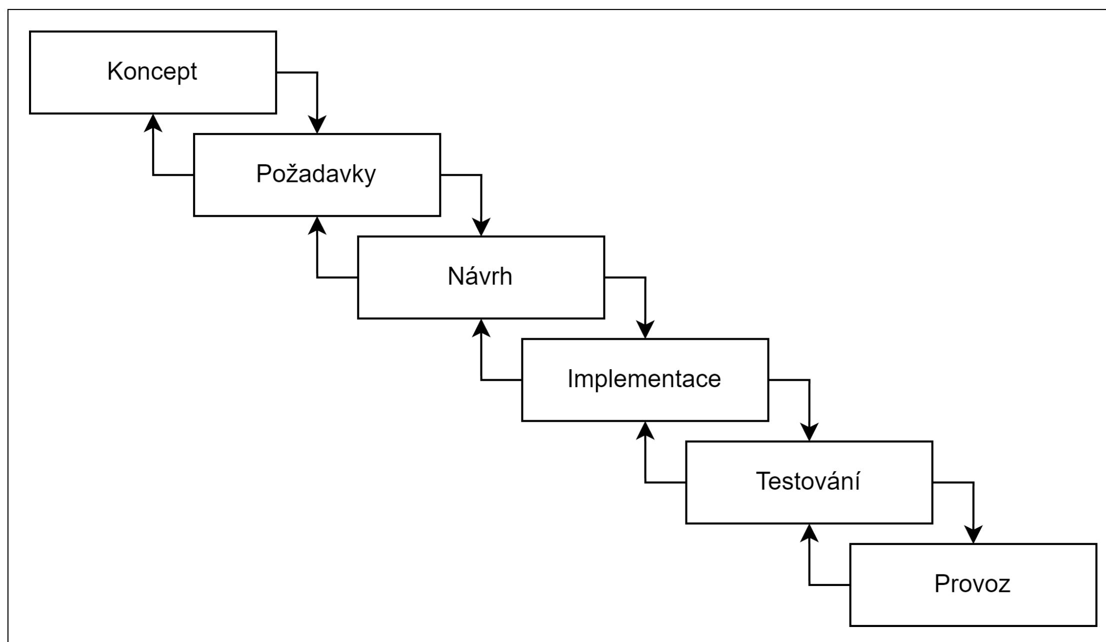
Klasické metodiky jsou historicky starší. Kladou velký důraz na tvorbu dokumentace, která má vznikat v rámci celého projektu. Jsou často velmi dobře propracované a velmi rozsáhlé, což může vést ke zvýšení pracnosti na daném projektu, případně i ke snížení kvality, protože tým se bude metodice věnovat příliš mnoho času. [15]

U klasické metodiky dochází k tomu, že na začátku projektu jsou pevně dané cíle, které není tak jednoduché za běhu změnit. Případný zákazník častokrát výsledky projektu vidí až na jeho konci a není součástí projektu jako takového. [15]

Vodopád

Vodopádová metoda je jednou z primárních metodik používaných v současnosti při vývoji softwarových projektů. Tento model je zástupcem klasických metodik projektového řízení. Je charakterizován sekvencí fází, které jsou často nazývány iteracemi a postupují lineárně, vzájemně se ovlivňují. [16]

Na obrázku 2.3 je možné vidět průběh vodopádové metody. Většina projektů začíná konceptem, z něhož vycházejí různé požadavky, jak funkční, tak nefunkční. Během procesu definování požadavků často dochází k úpravám původního konceptu, což je znázorněno dvousměrnou šipkou. Další fáze postupují lineárně a iterace jsou možné pouze u okamžitě následující fáze.



Obrázek 2.3 Sekvence iterací v rámci vodopádové metody

Z průběhu vodopádové metody je zřejmé, že je klíčové vytvořit požadavky pečlivě, aby se minimalizovalo riziko návratů a změn v pozdějších fázích projektu. Hlavní nevýhodou této metodiky

je fakt, že zákazník má možnost vidět výsledky projektu až v pozdějších fázích testování a provozu, což omezuje jeho možnosti aktivního zásahu do vývoje. Na druhou stranu však vodopádová metoda nabízí jednodušší řízení a celkové plánování projektu. [16]

2.2.3.2 Agilní metodiky

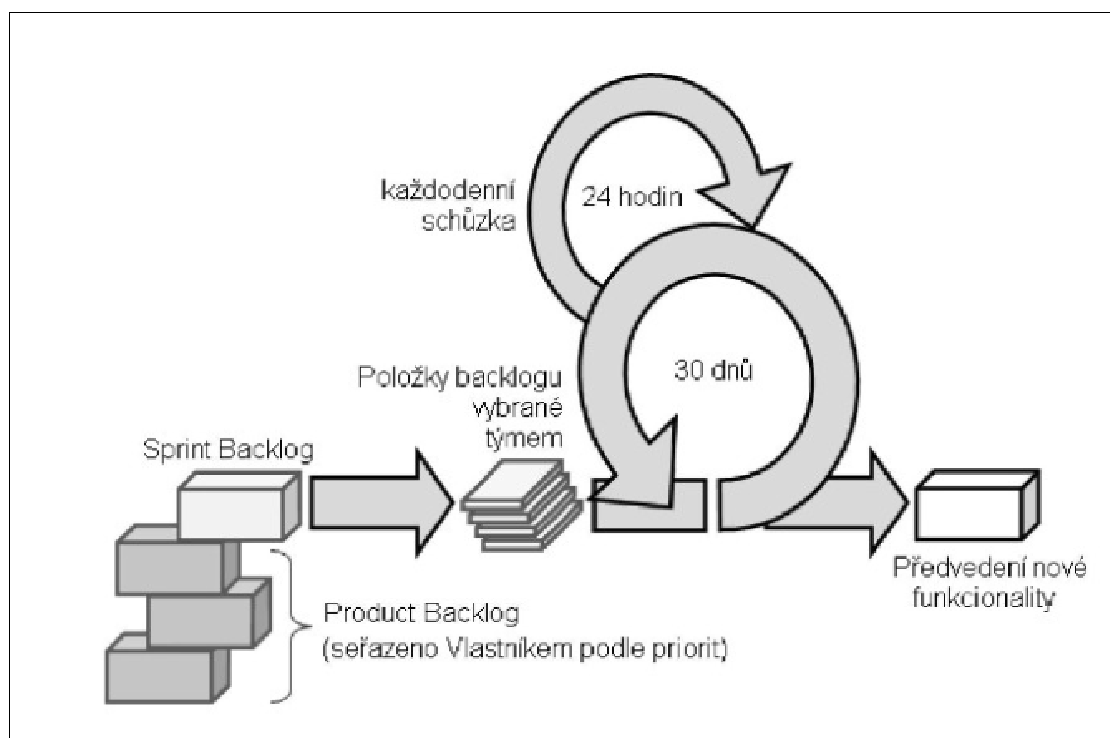
Agilní metodiky se v dnešní době těší velké oblibě a jsou vnímány jako modernější alternativa ke klasickým metodikám. V agilním přístupu je produkt klíčový a snižuje se důraz na vytváření rozsáhlé dokumentace. Na rozdíl od tradičních metodik je v agilním přístupu zapojení zákazníka větší a definované cíle jsou flexibilnější, což umožňuje provádět úpravy na základě zpětné vazby. Pro úspěšné fungování agilní metodiky je nezbytná silná komunikace, jak mezi členy vývojového týmu a produktovou částí, tak mezi zákazníkem a dodavatelem. [15]

Scrum

Scrum je nejznámějším zástupcem agilních metodik. Hlavní výhodou je především možnost rychlého zapracování změn požadavků od zákazníka. [16]

Jeho princip spočívá ve vytvoření tzv. produktového *backlog*, ve kterém se nachází seznam všech požadavků od zákazníka. Tyto požadavky mají svojí prioritu, na základě které se přidávají do *sprints* (tří až čtyř týdenní iterace), ve kterých musí tyto požadavky být zpracovány. O určení priorit se stará buď projektový manažer a nebo *scrum master*. [16]

Nedílnou, až kritickou součástí je pravidelné setkávání vývojářského týmu, který by měl každý den diskutovat nad aktuálním stavem projektu. Tyto schůzky můžou sloužit pro odstranění překážek, které mohou zpomalovat celý vývoj. [16]



Obrázek 2.4 Popis metodiky Scrum z SWI přednášky [15]

Na obrázku 2.4 je vidět popis této metodiky z SWI přednášky [15]. Zmíněné je zde, že backlog může sestavovat tým, což je vhodné pro menší týmy.

2.2.3.3 Výběr metodiky

Klasické i agilní metodiky mají své výhody i problémy. Před začátkem projektu, je tak potřeba stanovit, jakým způsobem má být projekt řízen. Pro výběr je potřeba brát v potaz tyto věci:

Velikost projektu může hrát velkou roli. Pokud je projekt velký a při analýze bylo potřeba vydefinovat velké množství cílů, můžou být časté změny požadavků (které jsou u agilního přístupu důležitou součástí) velkým problémem, který může vést až k neúspěchu celého projektu. [15]

Velikost týmu je znát především u agilních metodik. Vzhledem k tomu, že u těchto metodik je velmi důležitá každodenní komunikace, může být větší počet členů v týmu problémem. Komunikace v takto velkých týmech je velmi neefektivní. Pro větší týmy se tedy obecně doporučuje klasická metodika. [15]

Složení týmu může mít také zásadní podíl na výběru metodiky. Pokud je tým plný elánu, všichni se pravidelně vídají (nepracují vzdáleným přístupem) a mají všichni chuť se aktivně podílet na celém projektu, dává smysl použít metodiky agilní. Tyto metodiky však vyžadují zapojení všech, což může být problémem. [15]

Stabilita požadavků je věc, na kterou se přijde během analýzy před samotným spuštěním projektu. Pokud má zákazník všechno pořádně promyšlené, požadavky jsou srozumitelné a všechny cíle projektu splňují metodu SMART (sekce 2.2.2.1), nemá úplně velký smysl použít agilní metodiku. Ta by se uplatnila v případě, že zákazník si není jistý, co opravdu chce a až názorná ukázka mu pomůže lépe vydefinovat požadavky. [15]

Firemní kultura může mít také vliv. Jsou firmy, které mají projektové řízení nastaveny pro všechny projekty stejně, v tom případě pak nově vzniklý projekt musí dodržovat tuto kulturu. Obecně se dá počítat, že menší firmy a startupy mají blíže k agilním metodikám, zatímco větší firmy se spíše budou držet metodik klasických. [15]

Zvyklosti zákazníka jsou také důležitým rozhodovacím faktorem. Zvlášť pro agilní metodiky je potřeba, aby k nim byl zákazník otevřen, protože tyto metodiky pro něho znamenají zátěž a hlavně vyšší náklady. [15]

2.3 Logovací systémy

V této sekci bych se rád zaměřil na logovací systémy, které jsou v SW (Software) produktech velmi časté. Nejdříve zde vydefinuji log samotný, vypíšu jeho typy a co takový log může obsahovat. Na závěr zmíním jakým způsobem se logy zobrazují.

2.3.1 Log

V oblasti IT je log velmi často používán. Jak už to v tomto odvětví bývá, není úplně přesná definice, která by termín log dokonale popsala. Z webové stránky AWS od Amazonu lze log definovat jako automaticky vygenerovaný záznam v daném programu. Tento záznam může obsahovat informace o různých událostí, které v programu nastaly. [17]

Vícero logů dohromady může tvořit historii všech procesů, událostí a zpráv. K pečlivému sestavení historie často slouží časové značky, které můžou konkretizovat, kdy jaká událost nastala. Pokud se tedy v programu stane něco špatně, lze pomocí logů odhalit chybu. [17]

2.3.1.1 Typy logů

Událostní log je základním typem. Jak z názvu vyplývá, jedná se především o záznam událostí. Slouží k popisu chování systémů a je velmi nápomocný při odhalování chyb programu. [17]

Systémový log pochází ze samotného operačního systému a není definovaný aplikací. [17]

Přístupový log má především význam pro zabezpečení aplikace. Díky tomuto záznamu, lze kontrolovat, který uživatel provádí konkrétní autentizace. [17]

Serverový log vytváří automaticky server. Tento záznam nese například informace o počtu přihlášení, IP adresy klientů, typy požadavků atd. [17]

2.3.1.2 Využití logů

Identifikace a odstraňování chyb je jednou z hlavních vlastností logů. Program se může dostat do situací, kdy sám vyhodnotí, že nastala chyba (nepovedlo se připojit k serveru, byla vyhozena výjimka). Tuto chybu pak může zaznamenat do logu. Tato vlastnost může vývojářům ulehčit opravování a vylepšování celého systému. [17]

Zlepšení provozu je také potencionální využití. Logy mohou zaznamenávat informace o zpoždění či zátěži v různých částech aplikace. Tyto informace mohou odhalit potencionální problém a vývojáři mohou reagovat s předstihem. [17]

Zvýšení efektivity se dá vyčíst z četnosti logů v konkrétní moment. Díky tomu lze lépe plánovat využití různých zdrojů. Tímhle se může například vyřešit přetížení serveru, což vede ke zvýšení spokojenosti uživatelů. [17]

Pochopení chování uživatelů má především význam z hlediska marketingu a komercionalizace. Logy se v tomhle případě mohou použít k odhadnutí uživatelských návyků. Díky tomu mohou vznikat nové funkcionality a tím může dojít ke zvýšení spokojenosti zákazníka. Lze tuhle vlastnost využít i pro vytváření reklam, které mohou přinést určitý finanční obnos. [17]

Posílení zabezpečení pomocí logů je velmi jednoduché. Logy mohou zaznamenat neobvyklé chování uživatelů, jako větší množství neúspěšných pokusů či pokusy o DDOS (Distributed Denial-Of-Service). Díky logům lze zjistit, že se něco takového děje a kybernetický tým či vývojář tak může včas zareagovat. [17]

2.3.1.3 Obsah logů

Log obsahuje mnoho věcí, které mohou poté sloužit pro lepší orientaci mezi nimi. Já bych zde rád zmínil ty nejzásadnější, které by žádnému z logů neměly chybět:

Priorita logu nese informaci o tom, jak moc závažný záznam daný log prezentuje. To může sloužit k jednoduššímu odhalování chyb, protože takové záznamy budou mít vyšší prioritu, než obyčejné označení úspěšné operace. Mezi nejčastější používané schéma priorit patří:

- FATAL: chyba, která většinou vede k vypnutí celé aplikace
- ERROR: chyba, označující špatné chování aplikace
- WARNING: varování, které značí, že chování aplikace může být nevyzpytatelný
- DEBUG: záznam, který slouží při opravování chyb
- INFO: pozitivní záznam, může značit úspěšně provedenou operaci

Časový záznam slouží k pozdějšímu sestavení časové osy běhu daného programu. Díky této informaci lze vytahovat jednotlivé části. Vývojář může zpozorovat nějakou podezřelou aktivitu, která může značit potencionální problém.

Kategorie u logů slouží k možnému dělení. Lze pomocí toho shlukovat informace, které k sobě patří. Ideální situace nastává, pokud pro procházení logů je použit systém, který nabízí filtrování na základě kategorie.

Zpráva samotná, je také nedílnou součástí logu. Zprávou lze podat veškeré informace a většinou se jedná o tu nejzajímavější část.

Dodatečné hodnoty u logů můžou znamenat uložení nějakých proměnných. Tato technika je o něco pokročilejší, ale může pomoct pro sledování dění v programu.

2.3.2 Zobrazení logů

Opravdový význam logy dostávají až v případě, že si je vývojář, správce či tester zobrazí. V tom daném momentě je totiž možné výsledky z logů interpretovat. Aplikace nebo servery v závislosti na případu užití ukládají své záznamy do přístupných souborů, tzv. *logfiles*.

Takto vytvořené soubory nemusí být nutně textového typu. Může se stát, že aplikace budou pro zvětšení paměťové efektivity ukládat záznamy do binárních souborů. Problémem u takto uložených souborů je jejich nečitelnost.

Nabízí se tedy otázka, jak tyto uložené údaje zobrazovat. Obecně se k tomu využívají tzv. *Log Viewer*, aplikace, která pomáhá se zobrazováním logů pomocí pěkného uživatelského rozhraní, česky se tato aplikace dá nazvat „Zobrazovač logů". [18]

Tyto aplikace kromě jednoduchého zobrazení logů umožňují uživateli vytvářet různé pohledy nad daty, do určité míry umí i data řádně analyzovat. Často nabízejí možnost uživateli sledovat data živě za běhu programu, který záznamy posílá. Kromě toho je možné procházet i historická data z načtených *logfiles*. U lepších a dražších systémech se nachází i vizualizace dat, která opět napomáhá k lepšímu vyhodnocování běhu aplikace. [18]

Kapitola 3

Projektové řízení

Tato kapitola je zaměřena na projektové řízení v ETCS simulátoru a všechny související aspekty. Klíčovou součástí této kapitoly je analýza současného stavu, která zahrnuje SWOT analýzu. Cílem této analýzy je poskytnout co nejpřesnější popis stavu celého projektu. Výstupy z této analýzy jsou následně využity k navrhování vylepšení projektového řízení nebo dokonce k návrhu zcela nových procesů. Součástí je také návrh a implementace skriptu na měření odpracovaného času.

3.1 Analýza projektového řízení simulátoru ETCS

ETCS simulátor je v současnosti velmi rozsáhlým projektem. Jeho součástí je vícero týmů a je potřebná spolupráce dvou různých fakult ČVUT. Všechny tyto náležitosti značí, že řízení takového projektu není vůbec jednoduché. V této kapitole bych se rád věnoval tomu, jak řízení probíhá v současnosti. Poukážu na věci, které se za mě dělají dobře, ale zároveň označím věci, které by se daly dělat lépe. Výstupy z této části budou použity později pro návrh jak řízení zefektivnit.

3.1.1 SWOT analýza aktuálního stavu

SWOT (Strenghts, Weaknesses, Opportunities and Threats) analýza je základní metoda, která se běžně používá v různých projektech. Jejím cílem je zmapovat aktuální stav, aby všechny zúčastněné strany věděly o faktorech, které zásadně ovlivňují chod celého projektu. [19]

Základem jsou dva různé pohledy:

1. Pohled z hlediska původu: vnější a vnitřní,
2. Pohled z hlediska přínosu: pomocné a škodlivé. [19]

3.1.1.1 Vypracování analýzy

Při vypracovávání SWOT analýzy jsem vycházel z mých vlastních zkušeností, které jsem po více než roční práci na projektu získal. Informace se mi podařilo získat i od dalších členů týmu. Pomocí těchto informací se mi povedlo identifikovat nejen hlavní problémy, ale i to, co je v projektu velmi dobré.

Silné stránky se nachází vlevo nahoře ve světle zelené části. Tyto stránky značí, co je v současnosti projektu dobré. Některé věci jsou velkou konkurenční výhodou, což může vést k úspěšnému zakončení projektu.

Znalosti zákazníka souvisí s tím, že zadavatel celého projektu, pan doc. Ing. Martin Leso, Ph.D., je velmi znalý v dané problematice. To může urychlit všem týmům orientaci v oficiálních dokumentech pro ETCS, které jsou při implementaci simulátoru klíčové. Přístup zákazníka k celému projektu je velmi vřelý, což se dá využít pro zlepšení projektového řízení.

Bakalářské práce jsou vhodnou praktikou pro udržení studentů, aby na projektu pracovali delší dobu. Mohou díky tomu pomáhat novým týmům a opadá díky tomu čas, který musí nově příchozí strávit se seznamováním se s projektem. Zároveň se pomocí bakalářských prací dají udělat větší části implementace, což může urychlit celý projekt.

Nápad Lektorského pracoviště je originální věcí celého ETCS simulátoru. Komponenta LPC (Lektorské pracoviště) má být jakýmsi středobodem celého simulátoru. Pomocí ní je možné veškerý simulátor řídit a konfigurovat, komponenta by také měla umět vyhodnocovat jízdu. Tento prvek je konkurenční výhodou.

Slabé stránky jsou vypsány v oranžové části, která se nachází vpravo nahoře. Tyto slabé stránky popisují, co je v uvnitř projektu špatně a nebo celý projekt do nějaké míry negativně ovlivňuje. Případné odstranění těchto slabých stránek může mít pozitivní vliv na úspěch projektu.

Závislost na předmětech SP1 a SP2 zpomaluje celý projekt. Tyto dva předměty mají svá pravidla, které častokrát jdou proti samotnému projektu. Jedná se o větší problém, který více rozeberu v kapitole 3.1.3.

Slabší komunikace mezi týmy je způsobena rozsáhlostí projektu, na kterém pracuje vícero týmů. U nich je potřeba vynaložit velké úsilí na komunikaci, která podle mě neprobíhá ideálně. Více o tomto problému píšu v kapitole 3.1.2.

Složité předávání zkušeností je pro projekt velkým problémem. Na projektu z větší části pracují primárně studenti, kteří většinou po konci předmětu začínají pracovat na bakalářské práci (ne nutně v rámci ETCS projektu). Tento odliv zkušeností má negativní dopad na další týmy, neboť musí sami pochopit chod celého simulátoru.

Časté změny v implementaci jsou problémem, který je v SW projektech velmi častý. V tomto projektu dochází k časté obměně týmů, které mají jiné znalosti a dovednosti. To vede k častému přepracování práce předchozích týmů, což celý projekt zpomaluje.

Příležitosti ve světle hnědé části, která se nachází vlevo dole, značí vnější okolnosti, které mohou projektu pomoci. Tyto okolnosti fungují jako určitá motivace, která může sloužit k rychlejšímu vývoji. Příležitosti mohou do projektu přinést i nějaké finance, které momentálně chybí.

Nasazení ETCS na české železnice se každým dnem blíží. Povinnost mít ETCS aspoň na evropských koridorech způsobí, že bude potřeba mít strojvedoucí proškolené. Právě to může zvýšit poptávku po tomto simulátoru.

Hrozby jsou v dolní části tabulky úplně na konci, která je vybarvena fialovou barvou. Značí vnější okolnosti, které negativně ovlivňují celý projekt. Je velmi těžké je jakkoliv ovlivnit, většinou je potřeba minimalizovat následky těchto okolností.

Rychlejší konkurence bude s největší pravděpodobností reagovat na zvýšenou poptávku po proškolení. V železničním odvětví existují některé firmy, které mají silnou pozici a téměř monopolní postavení. Je pravděpodobné, že tyto firmy reagují na zvýšenou poptávku po školení vlastními řešeními simulátorů.

Prodloužení realizace ETCS v ČR je hrozba, kterou nelze dostatečně ovlivnit. Přesto, že je ČR v rámci Evropské unie vázaná k realizaci ETCS, může se stát, že nedojde k dodržení stanovených termínů a i kvůli tomu může klesnout poptávka po simulátoru.

	POMOCNÉ	ŠKODLIVÉ
VNITŘNÍ	Znalosti zákazníka Bakalářské práce Nápad Lektorského pracoviště	Závislost na předmětech SP1 a SP2 Slabší komunikace mezi týmy Složité předávání zkušeností Časté změny v implementaci
VNĚJŠÍ	Nasazení ETCS na české železnice	Rychlejší konkurence Prodloužení realizace ETCS v ČR

Tabulka 3.1 SWOT analýza projektu ETCS simulátoru

V tabulce 3.1 jsou zobrazeny všechny dříve identifikované výstupy. Horní část tabulky obsahuje vnitřní faktory, zatímco dolní část obsahuje faktory externí. Tyto faktory jsou dále rozděleny horizontálně podle jejich vlivu na celý projekt

3.1.1.2 Stanovení SWOT strategie

Po vypracování SWOT analýzy má dojít k jejímu vyhodnocení a především k určení strategie, kterou by měla být na projektu zvolena. Je potřeba vybrat jednu ze čtyř kategorií:

1. **MAX-MAX** maximalizace silných stránek při maximalizaci příležitostí. Cílem projektu je efektivně využít existujících silných stránek a současně identifikovat a využít veškeré dostupné příležitosti.
2. **MAX-MIN** maximalizace silných stránek při minimalizaci hrozeb. Projekt má být zaměřen na využití všech silných stránek a zároveň minimalizovat hrozby a jejich dopady, pomocí vyhodnocení rizik.
3. **MIN-MAX** minimalizace slabých stránek s maximalizací příležitostí. Úsilí v projektu by mělo být zaměřeno především na minimalizaci slabých stránek s vědomím, že je potřeba se soustředit na všechny příležitosti.
4. **MIN-MIN** minimalizace slabých stránek a hrozeb. Projekt je zaměřen na eliminaci známých slabých stránek a hrozeb, včetně jejich možných negativních dopadů.

3.1.2 Komunikace mezi týmy

Na ETCS simulátoru většinou pracuje více než jen jeden tým. Projekt je totiž velmi rozsáhlý a menší skupina lidí tento objem práce nemůže zvládnout. Při mém nástupu v rámci letního semestru v únoru 2023 byly na projekt navázaný čtyři týmy: jeden se zabýval samotným ETCS, jeden řešil vše okolo LPC a dva týmy se staraly o vizualizaci simulátoru.

Mezi takto velkým počtem týmů je potřeba zajistit správnou komunikaci a vhodné předávání informací, které se povedlo získat během analýzy či návrhu. Bohužel musím z mých osobních zkušeností říct, že se tak nedělo. Spousta věcí zůstala vždy pouze u jednoho týmu a sdílení informací neproběhlo.

Ani koordinační schůzky, které probíhaly každý týden, komunikaci nijak nezlepšily. Pro většinu týmů to byla ztráta času, protože nedošlo ke schválení společného cíle. Spíše se jednalo o zjištění stavu všech částí, což byla primárně informace pro zákazníka. Při většině schůzek se zákazník nemohl zúčastnit, protože mu o nich nebylo předem řečeno.

Celá komunikace byla také silně ovlivněna tím, že každá část projektu byla v jiném implementačním stavu. Zatímco týmy na vizualizaci začínaly na tzv. „zelené louce“, týmy pracující na LPC a ETCS už měly kód, se kterými musely pracovat. Propast mezi týmy se poté také vyskytla v plnění úkolů, kdy tým na ETCS se dokázal zorientovat rychleji a ostatním komponentám utekl.

Myslím si, že v projektu nám schází osoba, která by měla přehled o stavu všech jeho částí, nejen o tom, co každá zvládá, ale i o stavu kódu. Přítomnost takového jednotlivce by umožnila

eliminovat potřebu stavových schůzek, které často jen zbytečně zdržují studenty. Místo toho by koordinační meetingy mohly sloužit primárně pro plánování a koordinaci dalších kroků. Tato setkání by měla být systematicky vedena touto osobou společně se zákazníkem. Věřím, že by všem prospělo, kdyby tyto schůzky byly pečlivě naplánovány dopředu a agenda by byla týmům zaslána s dostatečným předstihem.

Nechci být jenom kritický, tak bych tu rád zmínil, že se mi líbí řešení přes Discord[1], kde existuje ETCS server. Přístup do tohoto serveru mají současní studenti, ale i ti bývalí, takže velkou výhodou je, že lze takhle komunikovat se zkušenějšími studenty. Věřím, že by se komunikace přes tuto platformu dala ještě zlepšit, ale obecně se mi tento princip líbí.

3.1.3 Střet školních předmětů s projektem

V této části se zabývám největším problémem celého projektu. Konkrétně jde o vázanost na školní předměty, především na SP1 (Softwarový týmový projekt 1) a SP2 (Softwarový týmový projekt 2), což má za následek zpomalení celého procesu. Tyto předměty stanovují specifická pravidla, která se musí dodržovat. Zejména u SP1 je tento problém velmi patrný.

Pokud vezmeme v úvahu, že studenti mají v rámci předmětu SP1 k dispozici pouze 13 týdnů, začnou se implementací zabývat až v posledních 3 týdnech. Tento časový stres se prohlubuje, když se v poslední fázi semestru současně konají zápočtové testy i u ostatních předmětů. V důsledku toho máme k dispozici pouze omezený čas na implementaci, a tak se projekt často zastaví v analýze a návrhu, s představou, že se implementace uskuteční v následujícím semestru. Co se týče implementace v rámci předmětu SP2, ta probíhá pouze tehdy, pokud stejný tým pokračuje, což není vždy zaručeno.

Častá změna týmů pochopitelně projektu taky nepomáhá. V celém ETCS simulátoru není nikdo, kdo by znal kód od počátku až do současnosti. Není zde autorita, která by jasně určila, jak bude vypadat architektura. V předmětech SP1 a SP2 jsou studenti tlačeni do vlastních návrhů, což vede k přepisům celých komponent. Předávání projektů, jako se děje vždy po konci SP2, je běžnou praxí i v *business* světě. Je však zapotřebí lepší koordinace a komunikace, což by se mělo promítnout i zde.

Celý ETCS simulátor není v současnosti projekt, ale studentské pískoviště, kde si každý postaví svůj hrad z písku, aby ten druhý mu to zbořil a udělal to po svém. Za mě je to rozhodně špatně. Chápu, že pro softwarové inženýry je potřeba dělat návrhy, vymýšlet architekturu, používat správně praktiky objektově orientovaného programování, ale pokud se naskakuje už do něčeho rozjetého, je potřeba se držet toho, co už existuje.

ETCS simulátor stojí na rozcestí. Jedna cesta vede směrem k profesionálnímu projektu, který může mít třeba i ekonomický přesah. Ta druhá vede k tomu, že bude zcela pohlcen předměty SP1 a SP2. Čím dřív dojde k rozhodnutí, jakou cestu projektu si zvolit, tím lépe bude pro všechny.

3.1.4 Procesy v rámci projektu

V této sekci bych rád popsal procesy, který tým musí v rámci svého působení na projektu plnit. Mnoho věcí vychází z předmětů SP1 a SP2, je zde ale pár speciálních případů, které v jiných projektech není. Sekci rozdělím dále do dvou částí, a to na procesy v rámci týmu a pak na procesy, do kterých se zapojuje vícero týmů.

3.1.4.1 Interní procesy týmu

Pro týmy v rámci předmětu platí, že mají vždy minimálně tři iterace, které je potřeba odevzdávat. Celý projekt, včetně jeho procesů, se těmto třem milníkům podřizuje pomocí vodopádové metodiky (kapitola 2.2.3.1). Milníky v SP1 jsou rozděleny následovně:

[1] Aplikace pro komunikaci

Analýza je součástí první iterace. Tým má během této fáze za cíl se seznámit s celým projektem, a to nejen s kódem, ale i s celou problematikou ETCS. Tomu jim můžou pomoct různé schůzky: se zákazníkem, svým vedoucím, ale i s bývalými studenty (pokud si najdou čas). Díky rozsáhlosti a náročnosti projektu se stává, že je tato fáze velmi zdlouhavá a zabírá nejvíc času v rámci předmětu.

Pochopení ETCS probíhá v několika fází. Nejdříve dochází k velmi rychlému seznámení a jsou vysvětlené jednoduché principy. K tomu je vhodné se spojit s bývalými studenty, kteří prošli stejnými procesy a o ETCS už něco vědí.

Po tomto seznámení následují další kroky týmu do Dopravního sálu Fakulty dopravní ČVUT (webové stránky [20]), kde dojde k první schůzce se zákazníkem. Zde studenti získají další znalosti o zabezpečovacích systémech železnicích, ale i požadavky od zákazníka, které poté musí zpracovat. Tyto požadavky definují náplň dalších týdnů v rámci předmětu.

Požadavky poté musí tým analyzovat, odhadnout jejich náročnost a zjistit, jak do dosavadního řešení požadavky pasují. Zde dochází už k té analýze, která je pro softwarové projekty typická a většinou jim už nikdo nepomáhá.

Na konci iterace je potřeba veškeré zanalyzované poznatky zaznamenat. Závěrečný dokument by měl obsahovat všechny zjištěné poznatky, včetně odhadnutí pracnosti všech požadavků a určení jejich priorit. Tento dokument pak bude sloužit jako podklad pro další dokumenty.

Návrh je hlavní částí prostřední iterace v rámci předmětu SP1. Studenti zde musí přijít s vlastním nápadem a ten poté zakreslit pomocí diagramů. V této fázi je i nadále potřeba čerpat informace z oficiální dokumentace ETCS a požadavky, které jsou v ní popsané vzít v potaz při tvorbě vlastních řešení.

Veškeré návrhy je potřeba řádně zaznamenávat a na konci iterace je prezentovat vedoucímu pomocí závěrečného dokumentu. Je vhodné se během těchto návrhů radit s bývalými studenty, aby došlo k zachování stávajícího řešení a nedocházelo k častým změnám kódu. Při tvorbě těchto návrhů je potřeba dodržovat prioritu jednotlivých požadavků, která byla stanovena v předchozí iteraci.

Implementace uzavírá celý předmět. I když se jedná o nejdůležitější část všech softwarových projektů, tak na ní bohužel moc času není, protože předchozí části jsou náročné. I tak by ale mělo dojít k implementování několika návrhů, které byly navrženy a prezentovány v závěrečném dokumentu předchozí iterace.

Při implementaci je potřeba dodržovat návrhy z předchozí iterace, ale i konvence, které jsou definované v rámci projektu. Během této fáze je velmi pravděpodobné, že vzniknou požadavky nové ze strany týmu. Tyto požadavky je potřeba (jako ty předchozí) řádně analyzovat a vytvořit k nim návrhy.

Ve všech těchto iteracích dochází k pravidelným týdenním schůzkám týmu s vedoucím ze strany učitelů. Tyto schůzky slouží k podávání informací o stavu a k případnému oznámení některých nedostatků nebo problémů. Vedoucí má možnost na těchto schůzkách tým posunout nějakým směrem, ať už ke správnému řešení nebo obecně k plánování celého projektu.

Předmět SP2 je trochu odlišný. Vedení tohoto projektu je vůči studentům vstřícné a nabízí jim možnost si tento předmět odchodit během léta. Výhodou je, že studenti mají víc volného času a můžou se projektu více věnovat. Nevýhodou jsou formality, který tento předmět má.

Opět se v něm nachází tři iterace, který je potřeba zakončit odevzdáním dokumentu. Kvůli školním pravidlům tyto dokumenty nelze odevzdat v létě a tak se některým studentům práce stejně přesouvá do zimního semestru. Každá z těchto tří iterací má určitý konkrétní dokument, který je vyžadován předmětem SP2:

Administrátorská příručka je hlavním výstupem první iterace. Slouží primárně pro budoucí studenty, aby se lépe orientovali v prostředí ETCS simulátoru. Je vhodné zde reflektovat

věci, které se v tomto prostředí změnily. Tím může být například vytvoření nového repositáře, změna nějakých konfigurací či procesů. Vhodné je také případně do dokumentu uvést informace o tom, jestli se nějak změnilo spouštění celého projektu.

Uživatelská příručka by měla navazovat na tu administrátorskou a zakončovat druhou iteraci. Měly by zde být uvedené nové funkcionality simulátoru a jak jednotlivé komponenty používat. Tento dokument by měl pak v budoucnu sloužit jako informace pro případné zákazníky.

Programátorská příručka je výstupem třetí a poslední iterace předmětu SP2. Jak je z názvu patrné, slouží primárně pro programátory, kteří se v budoucnu dostanou do projektu. Může se jednat o samotný dokument, nebo můžou jednotlivé informace pro programátory být uloženy na vícero místech (Wiki, kód, dokumenty v repositářích atd.). Součástí by měla být různá pravidla, popis architektury a dalších řešeních, aby se budoucí studenti jednoduše orientovali.

Pravidelný report stavu simulátoru není vyžadován přímo předmětem SP2, ale spíš vedením ETCS projektu. Je zde potřeba reflektovat aktuální stav, počet odpracovaných hodin a plány do dalších iterací.

Zvažování formátu dokumentů pro jednotlivé iterace předmětu SP2 je důležité s ohledem na budoucí čitelnost a správu dokumentů. S narůstajícím množstvím verzí jednotlivých typů dokumentů může být obtížné udržet přehlednost a strukturu. Proto by bylo vhodné po odevzdání každé iterace vytvořit jeden komplexní dokument, který bude obsahovat nejen aktuální informace, ale také relevantní informace z předchozích ročníků.

3.1.4.2 Externí procesy týmu

Je pravidlem, že tým pracující na ETCS simulátoru není jediným. Je tedy potřeba mít pravidelné informace o stavu v ostatních týmech, kde je velmi důležitá komunikace. Hlavním externím procesem jsou tedy koordinační schůzky, o kterých píšu v kapitole 3.1.2. Ke komunikaci se v posledních bězích začala více používat aplikace *Discord*. Mezi časté procesy tak patří i komunikace s ostatními týmy v rámci této aplikace.

3.1.5 Požadavky pro podpůrné nástroje týmu

Z hlediska projektového řízení je vhodné připravit týmu takové podmínky, ve kterých nebude zbytečně ztrácet čas. Proto došlo ze strany týmu k vytvoření několika požadavků, které jsem měl za úkol zpracovat.

3.1.5.1 Pracovní postupy týmu v systému správy verzí

Pro týmovou spolupráci se využívá fakultní GitLab, který umožňuje efektivní správu verzí kódu. Jednou z klíčových funkcí této platformy je možnost vytvářet a spravovat úkoly tzv. *issue*, což pomáhá lépe organizovat práci a plánovat postup projektu.

Nicméně, nativní funkcionality GitLabu ne vždy plně vyhovují potřebám týmu, a proto je občas potřeba upravit určité aspekty. Prvním krokem k tomu je zvolení vhodných postupů pro práci s verzovacím systémem Git, což si vyžaduje zvážení různých faktorů, jako je velikost týmu, frekvence úprav a rozsah kódu.

Dále je nutné promyslet, jakým způsobem se budou vytvářet a spravovat jednotlivé *issue*. Fakultní GitLab obsahuje pouze dva stavy – *open* (otevřený úkol) a *closed* (uzavřený úkol), což není dostatečné pro potřeby týmu. Je tedy potřeba implementace vlastních stavů, které budou lépe reflektovat postup řešení úkolů a pomohou týmu se orientovat v aktuálním stavu projektu.

Kromě stavů je vhodné zohlednit i další vlastnosti úkolů jako je typ, priorita a náročnost. Je tedy potřeba vymyslet systém pro kontrolu těchto vlastností, což pomůže lépe řídit pracovní postupy a zlepšit produktivitu celého týmu.

3.1.5.2 Měření času

Předměty SP1 a SP2 jsou zakončeny klasifikovaným zápočtem. Aby mohl být student ohodnocen, je potřeba prokázat, kolik práce na projektu udělal. Jednou z metrik, na kterou lze v tomto směru nahlížet, je odpracovaný čas v člověkohodinách. Fakultní GitLab, nabízí možnost přidávat odpracovaný čas k jednotlivým *issue*. Problémem však je, že nelze jednoduše zjistit kolik odpracovaných hodin má někdo na všech *issue*, které se v projektu nachází..

Sčítání času na jednotlivých *issue* je práce, která se dá velmi jednoduše zautomatizovat. Bylo by vhodné vytvořit skript, který pomocí GitLab *API* získá všechny potřebné informace. Tento skript by měl umět pracovat s časovým rozmezím, aby neprocházel již zaznamenané časy. Může běžet automaticky a pravidelně, aby vedoucí týmů z řad studentů i učitelů měl každý den nejaktuálnější informace.

3.1.6 Strategie projektu ETCS simulátoru

Po svém vstupu do projektu jsem nebyl dostatečně seznámen s žádnou celkovou strategií týkající se simulátoru. Teprve při analýze práce jsem měl možnost se seznámit s dokumentem nazvaným „Koncepce vlakového simulátoru ČVUT komponenta ETCS“. Tento dokument byl vytvořen s cílem zachytit základní informace a vytvořit jakousi dokumentaci pravdy. Z teoretického hlediska se nejvíce podobá projektové dokumentaci (viz kapitola 2.2.2.4) Tento dokument byl naposledy aktualizován dne 23. dubna 2021, což je rok, kdy celý projekt začal. V rámci analýzy je nezbytné ověřit jeho aktuálnost a případně provést úpravy, abychom zajistili jeho relevanci pro současný stav projektu.

3.1.6.1 Analýza koncepčního dokumentu

Rozebíraný dokument se skládá ze dvou hlavních částí: koncepce a architektura. První část je velmi krátká a stručně popisuje zasazení celého simulátoru. Vysvětluje důvody vzniku projektu a stanovuje požadavek na identické chování simulátoru s reálnými systémy. Druhá část je rozsáhlejší a podrobně se zabývá architekturou celého simulátoru.

Neřeší se jen architektura z hlediska SW, ale zabývá se i hardwarem. V dokumentu jsou zmíněný dva typy simulátoru: statický a dynamický. Jako vybraná platforma byl zvolen OS (Operační systém) Windows. Z hardwarového hlediska je systém rozdělen do tří částí:

Vizualizační PC musí být schopný kvalitně zobrazovat virtuální scény v minimálním rozlišení 4K. Pro statickou verzi simulace je potřeba mít alespoň jednu čelní obrazovku, typicky širokoúhlou televizi nebo monitor. Pro dynamickou verzi, známou také jako „full cab and motion“, je zapotřebí navíc postranních obrazovek na úrovni bočních oken kabiny.

Řídící PC je primárně určeno pro samotný simulátor. Na tomto počítači by měly být spuštěny vlakové počítače a zabezpečovací systémy včetně ETCS a národních systémů LVZ a Mirel. V případě, že simulátor běží na dynamickém stroji, je také zodpovědný za řízení servomotorů pohyblivé plošiny.

Lektorské PC je určeno jako operační pracoviště pro obsluhu simulátoru. Jedná se o počítač, na kterém se provádí řízení a vyhodnocování celé simulace. Toto pracoviště umožňuje nastavování scénářů simulace a poskytuje uživateli potřebné nástroje pro správu a monitorování průběhu simulace.

Dále se dokument primárně věnuje řídícímu PC a jeho čtyřem hlavním částem. První částí je aplikace „Vozidlový počítač“, která má za úkol implementovat provozní obrazovky v závislosti na typu simulovaného vozidla. Kromě toho je součástí této části i fyzikální model jízdy simulovaného vozidla.

Druhou částí řídícího počítače je aplikace „CANBus“, která slouží jako rozhraní pro komunikaci mezi elektronickými moduly a ostatními částmi simulátoru. Toto rozhraní je implementováno prostřednictvím průmyslové sběrnice CANBus 2.0. Pomocí této aplikace se data propisují do datového skladu.

Třetí částí řídícího počítače je „Datový sklad“. Tento sklad je provozován na platformě Mosquitto pomocí protokolu MQTT (MQ Telemtry Transport). Přístup k datovému skladu je definován na portu 1883 a odpovídající TCP/IP adrese.

Poslední částí je zabezpečovací systém samotný, což je hlavní cíl celého projektu a zbytek dokumentu je převážně věnován části s názvem „Aplikace ETCS“. Implementace softwaru probíhá pomocí jednotlivých modulů, které realizují jednotlivé části systému ETCS. Jako klíčové specifikace jsou zde zmíněny Subset-026, ERA ERTMS 015560 a Subset-027. Jako první krok byla zvolena implementace systémové specifikace verze 2.3.0d.

3.1.6.2 Aktuální stav koncepčního dokumentu

I přesto, že některé aspekty dokumentu zůstávají aktuální, existují určité odlišnosti, které je třeba zdůraznit. Například původní vize využití dvou různých modulů pro simulátor zůstává platná, stejně jako cílový operační systém Windows a rozdělení ETCS simulátoru do několika soběstačných modulů.

V průběhu realizace projektu došlo k některým změnám. Realita se trochu oddálila od plánovaného použití „datového skladu“. I když se pro komunikaci mezi komponentami stále využívá MQTT a platforma Mosquitto, není možné tvrdit, že tato data jsou uložena a kdykoliv načtena. Aplikace MQTT *Explorer* může být použita k čtení veškeré komunikace, která prochází daným brokerem, nicméně data získaná tímto způsobem nelze dále zpracovat.

Obecně se ale opravdu dá říct, že dokument vystihuje základní informace o celém projektu. V některých případech jsou dnes už konkrétnější informace, které by bylo vhodné časem doplnit. Zároveň je taky v blízké době v plánu povýšení na aktuálnější verzi ERTMS.

3.1.6.3 Harmonogram projektu

Aktuálně se projekt nachází v zajímavé situaci, protože 19. června 2024 se má na Vítězném náměstí konat tradiční VědaFest, kde má být i ukázka tohoto simulátoru. Tato událost představuje významnou propagaci projektu.

Úspěšná ukázka na VědaFestu by mohla zvýšit pravděpodobnost komercializace celého projektu. Všechny týmy, včetně studentů pracujících na svých bakalářských pracích, směřují veškeré své úsilí do přípravy demo ukázky.

Dlouhodobější harmonogram nemá konkrétní obrysy, vše směřuje k dané ukázce. Po ní dojde k vyhodnocení a k vytvoření projektového harmonogramu.

3.2 Návrh pracovních postupů v systému správy verzí

Tato kapitola vychází z požadavků uvedených v kapitole 3.1.5.1. Hlavním cílem návrhu je splnění těchto požadavků a vytvoření prostředí ve webové aplikaci, které bude pro vývojáře přehledné a zároveň užitečné pro plánování a orientaci v aktuálním stavu projektu. Velké množství těchto požadavků bude splněno správným používáním tzv. *Labels* (štítků), které lze jednotlivým úkolům přidělit. Na základě těchto štítků je pak možné používat různé filtry.

3.2.1 Zvolený postup práce s Git

Projekt ETCS simulátorů je týmovým projektem a pro práci s verzovacím systémem Git je potřeba vydefinovat několik pravidel, kterými se tým má řídit, tzv. *Gitflow*. V současnosti má

tým pod správou několik repositář, kde je potřeba tato pravidla nejen sjednotit, ale zároveň kontrolovat jejich dodržování.

Ve verzovacím systému Git se pracuje s tzv. větvemi, na kterých mohou existovat různé verze kódu. Na každé větvi může pracovat jiný vývojář, pak je potřeba tyto větve nějak sjednotit. Já pro práci na ETCS simulátoru navrhuji toto rozvržení větví:

master větev je hlavní větví celého repositáře. Kód v této větvi by měl být funkční se zbytkem simulátoru, protože je považován za aktuální vydanou verzi. K tomu je vhodné používat tzv. *Tag*, který slouží k rozlišení jednotlivých verzí. Tato větev je *protected*, což znamená, že do ní není možné přidávat změny na přímo, ale musí se k tomu použít *Merge request*, kde dochází ke slučování změn z jiné větve. Větev *master* se nesmí mazat.

develop_XXX větev je hlavní vývojářská větev. Slouží k postupnému shromažďování nových funkcionalit. Podobně jako větev *master* je *protected* a nikdy nedochází k jejímu odstranění. Označení *_XXX* slouží pro rozdělení funkcionalit v závislosti na semestru, kdy tato vývojářská větev existuje. Po skončení tohoto semestru se větev stává neaktivní, veškeré její změny je potřeba sjednotit do větve *master*, odkud následně vznikne nová *develop* větev.

release slouží pro jednodušší sjednocování změn z *develop* větví do *master*. Vzhledem k tomu, že obě dvě větve jsou *protected*, není možné je opravovat na přímo, což může být u *Merge request* procesu problém. Větev *release* by měla tento nedostatek odstranit. Její používání spočívá v tom, že se vytvoří na větvi *develop* a následně se vytvoří*Merge request* směrem do větve *master*. Po schválení dojde ke sloučení této větve do *master* větve, v případě, že došlo na této větvi k nějakým změnám, dojde ke sjednocení této větvě zpět do *develop* větve. Po tomto procesu dojde k jejímu smazání.

feature/ je předpona pro implementační větve. Takto označené větve vznikají nad větví *develop*, do které se potom slučují. Je vhodné větve pojmenovávat tak, aby co nejlépe popisovaly obsah změn. Tyto větve se nemažou, po čase se samy archivují v rámci webové aplikace GitLab.

bugfix/ slouží pro označení větví, jejiž úkolem je opravit nahlášenou chybu. Vznikají a následně se slučují do větve *develop*. Po úspěšné opravě chyby se nemažou, časem přestanou být aktivní.

refactoring/ označuje větve, které se zabývají úpravami, které nesouvisí s novou funkcionalitou ani s opravou chyb. Většinou se jedná o úpravy algoritmů, lepší strukturalizace kódu, atd. Stejně jako předchozí větve, tak pracují nad větví *develop* a nemažou se.

submodules/ slouží pro aktualizaci podrepositářů. Vzhledem k tomu, že v současnosti existuje mnoho repositářů, které mezi sebou sdílí kód pomocí repositáře CEM (Common Enums and Messages), je potřeba tuto část udržet aktuální. Pro její aktualizaci slouží takto označené větve, které se následně mažou. Jejich funkčnost je primárně omezena na *develop* větve.

hotfix/ oproti předchozím vzniká nad větví *master*. Slouží k rychlým úpravám nad vydanou verzí simulátoru. Po úspěšné opravě je potřeba změny sjednotit nejen do *master* větve, ale i do příslušné *develop* větve. Po opravě se tato větev maže.

documentation/ je poslední předpona, která slouží pro lepší orientaci ve větvích. Používá se pro tvorbu veškeré dokumentace, komentáře v kódu, ale i výstupní dokumenty k jednotlivým iteracím v rámci SP1 a SP2. Funguje nad *develop* větví a po sjednocení nezaniká.

V tabulce 3.2 jsou přehledně vypsány jednotlivé větve a jejich základní vlastnosti.

TYP	OCHRANA	MAZÁNÍ	POUŽITÍ
master	ano	ne	hlavní větev pro vydávání nových verzí
develop_XXX	ano	ne	hlavní vývojářská větev pro semestr *XXX*
release	ne	ano	vytvoření nové verze
feature/	ne	ne	implementace nové funkcionality
bugfix/	ne	ne	oprava chyb na vývojářské větvi
refactoring/	ne	ne	přepis stávajících funkcionalit
submodules/	ne	ano	aktualizace podrepositářů
hotfix/	ne	ano	oprava chyb na master větvi
documentation/	ne	ne	vytváření dokumentace

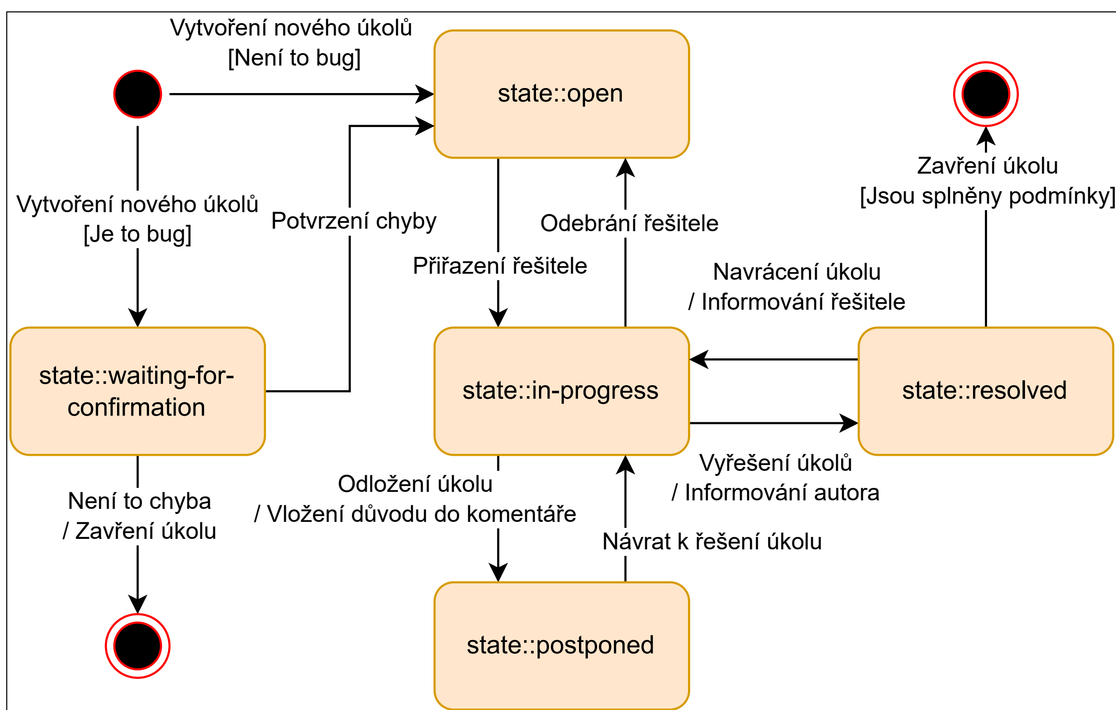
Tabulka 3.2 Větve pro repositáře v ETCS projektu

3.2.2 Správa úkolů

Jak zmiňuji v kapitole 3.1.5.1, nativní funkce pro správu úkolů ve webové aplikaci GitLab nejsou dostačující a je potřeba si repositáře přizpůsobit týmovým požadavkům. V této kapitole se budu věnovat návrhu nových funkcionalit, které splní požadavky a zefektivní týmovou práci.

3.2.2.1 Stav úkolů

Jednou ze základních věcí, které je potřeba u úkolů evidovat, je jejich aktuální stav. K tomu budou sloužit vytvořené štítky. Na obrázku 3.1 je vidět životní cyklus úkolu. Jednotlivé stavy jsou pak důkladně popsány v dalších odstavcích.



Obrázek 3.1 Životní cyklus úkolu

state::waiting-for-confirmation je pouze pro úkoly typu *bug* (chyby). *Issue*, které se nachází

v tomto stavu čeká na potvrzení, že se opravdu jedná o chybu. Po potvrzení chyby dochází ke změně na **state::open**, v opačném případě dochází k zavření daného *issue*.

state::open označuje úkol, který je připraven k řešení a může si ho kdokoliv vzít. Přiřazením řešitele se stav mění na **state::in-progress**.

state::in-progress značí, že na řešení daného úkolu někdo aktivně pracuje. K takto označeným úkolům je vhodné v pravidelných časových intervalech podávat aktuální stav v podobě komentářů. Pokud řešitel úkolu už nechce daný úkol řešit, odebere se jako řešitel a změní stav na **state::open**.

state::postponed je pro úkoly, které mají řešitele, ale momentálně se na nich aktivně nepracuje. Při přechodu do toho stavu je potřeba v komentářích *issue* uvést důvod.

state::resolved použije řešitel, pokud si myslí, že je daný úkol splněn. O daném vyřešení je potřeba informovat autora úkolu, který musí zkontrolovat, že je *issue* opravdu splněno. Pokud tomu tak je, může se úkol zavřít, v opačném případě se vrací do stavu **state::in-progress**.

3.2.2.2 Priorita úkolů

Další vlastnost, kterou je vhodné u úkolů hlídat, je jejich priorita. Štítky, které budou označovat prioritu jednotlivých *issue* je vhodné používat pro lepší plánování projektu. Já navrhuji používat tyto čtyři priority:

priority::low označuje úkoly s nejnižší prioritou. Pokud existují důležitější *issue*, není vhodné na těchto úkolech pracovat.

priority::medium označuje úkoly se střední prioritou. Tyto úkoly je vhodné dokončit do jednoho měsíce od jejich vytvoření.

priority::high označuje úkoly s vysokou prioritou. Takto označené úkoly by měly být označeny pomocí **state::in-progress** do jednoho týdne.

priority::critical označuje úkoly s nejvyšší prioritou. Toto označení by mělo být používáno pro úkoly, jejichž nevyřešení brání řešení dalších úkolů. Tyto úkoly by měly být vyřešeny co nejdříve.

3.2.2.3 Náročnost úkolů

Každý úkol má svojí náročnost, která slouží pro lepší odhadování stráveného času. Vývojářům může tento štítek naznačit, jestli stihne tento úkol vyřešit za víkend, nebo jestli bude na daném *issue* pracovat několik týdnů. Navrhuji používat tři různé štítky:

difficulty::friendly slouží pro označení nejjednodušších úkolů. Takové úkoly se dají většinou vyřešit do několika málo hodin. Takto označené *issue* je vhodné pro začátečníky, nebo pro rychlé odreagovaní zkušenějších programátorů.

difficulty::intermediate je pro středně složité úkoly. Je vhodné, aby na takto složitých úkolech pracovali už o trochu více zkušení programátoři v rámci projektu ETCS.

difficulty::advanced označuje nejsložitější úkoly. Často je potřeba důkladná znalost nějaké technologie, či důkladná práce s oficiální dokumentací ETCS.

3.2.2.4 Typ úkolů

Už z různých druhů větví (viz. tabulka 3.2) je patrné, že úkol může být různého typu. Pro jednodušší zjištění toho, na čem momentálně vývojáři pracují, jsem se rozhodl navrhnout tyto typy:

type::bachelor-support označuje úkoly, které slouží jako pomoc studentům pracujícím na svých bakalářských pracích v rámci projektu. Může se jednat o jednoduché testování, konzultaci, nebo reakci na nové funkcionality, které v rámci bakalářské práce vznikly.

type::bug je určen pro úkoly, které se zabývají objevováním, analýzou a následnou opravou existujících chyb. Větve, které se zabývají tímto úkolem, by měly být označeny pomocí prefixu *bugfix/* nebo *hotfix/*, v závislosti na tom, ve které větvi se chyba nachází.

type::documentation je pro úkoly, které se zabývají dokumentací. Většinou se bude jednat o vytváření výstupních dokumentů k jednotlivým iteracím v rámci předmětů SP1 a SP2.

type::enhancement označuje úkoly, které se týkají vylepšení stávajících funkcionalit. Může se jednat o efektivnější algoritmy nebo celkové vylepšení aktuální logiky.

type::feature je pro úkoly, které se zabývají novou funkcionalitou pro simulátor. Může se jednat o funkcionality definované v ETCS, či nějaké technické požadavky.

type::management je určen pro úkoly související s projektovým řízením v rámci projektu, ať už jde o plánování nebo kontrolu aktuálního stavu. Vedoucí týmu mohou toto označení použít pro další úkoly, které souvisejí s vedením projektu.

type::overhead je určen především pro měření času na různých schůzkách, nejenom pro potřeby předmětů SP1 a SP2.

type::refactoring označuje úkoly, ve kterých jde o přepis stávajících funkcí. Oproti **enhancement** by aktuální funkčnost měla být zachována.

3.2.2.5 Další štítky

Všechny vlastnosti, které jsem dosud představil, jsou u všech úkolů povinné. Je tedy potřeba mít u každého nově vytvořeného *issue* čtyři štítky, kdy každý z nich bude k dané vlastnosti.

Kromě těchto povinných vlastností, by bylo vhodné mít i další možnosti, jak jednotlivé úkoly od sebe rozlišit a tím usnadnit orientaci. Během vývoje se vývojář může dostat do stavu, kdy pro vyřešení úkolů potřebuje pomoc někoho jiného nebo má nějakou jinou překážku. Pro tyto případy jsem se rozhodl vytvořit speciální štítky **needs**. Pod tuto kategorii spadají tyto možnosti: **needs::analysis** (pro označení, že je potřeba *issue* nejdříve analyzovat), **needs::another-issue** (značí, že pro splnění tohoto úkolu je potřeba splnit nějaký jiný), **needs::discussion** (úkol potřebuje, aby se k němu vyjádřilo vícero lidí), **needs::review** (úkol je hotový a je potřeba zkontrolovat jeho řešení) a **needs::testing** (aktuální řešení potřebuje otestovat).

Druhou variantou jsou štítky označené pomocí kategorie **special**. Takovéto štítky slouží pro vyjádření čehokoliv jiného a je možné je jakkoliv upravit. Může se vytvořit pro signalizaci, že nějaké *issue* je součástí bakalářské práce, nebo že úkol je vhodný pro nováčky na projektu.

3.3 Návrh skriptu na měření času

V této kapitole se zabývám návrhem skriptu, který má primárně sloužit k měření času stráveného na projektu. Při tvorbě tohoto návrhu jsem vycházel z požadavků, které jsem analyzoval v kapitole 3.1.5.2 Jako preferovaný jazyk jsem zvolil Python, protože nabízí platformní nezávislost a podporu pro vytváření REST API dotazů, což bude klíčové pro získání dat z GitLabu. Zdrojem informací pro implementaci bude oficiální dokumentace GitLab API [21].

Filtrování *issue* na základě zadaného datového rozsahu. Vzhledem k pravidelným změnám týmu na projektu není vhodné, aby se sčítaly časy za celou dobu projektu. Uživatel bude mít možnost definovat datový rozsah, který určí *issue*, u nichž je potřeba měřit odpracovaný čas.

Spočítání naměřených časů proběhne pouze ve vyfiltrovaných *issue*. Některé úkoly mohou být rozsáhlejší a mohou se rozkládat přes více časových období. Proto bude potřeba sčítat pouze časy zaznamenané v daném datovém intervalu, který uživatel specifikuje. Časy z jednotlivých *issue* budou seskupeny podle jednotlivých osob.

Vhodné zobrazení sečteného času na konci skriptu. Výstup může být prezentován ve formě souboru nebo prostým vypsáním do konzole.

3.4 Návrh lepšího řešení projektového řízení

V této sekci se zaměřím na zlepšení aktuálního řízení projektu a také na možnosti jeho budoucího rozvoje. Navrhnu opatření, která mohou být implementována okamžitě, ale také se zaměřím na dlouhodobé strategie pro posílení projektového řízení. Případná aplikace těchto změn by měla být probrána především se zákazníkem, ale i s ostatními členy týmu. Všechny aplikované návrhy by pak měly být v samostatném dokumentu, kterým se budou moct budoucí týmy řídit.

3.4.1 SWOT analýza budoucího stavu

V praxi je velmi časté, že v případě analýzy stavu projektu se provádí SWOT analýzy dvě. Současný stav projektu je popsán v kapitole 3.1.1. V této kapitole se nachází SWOT druhá, která popisuje ideální stav, kam by měl projekt směřovat. Z následného porovnání těchto dvou analýz lze stanovit strategii, která povede ke zlepšení celého projektu.

Pro projekt ETCS simulátoru navrhuji použít strategii **MIN-MAX**. Slabé stránky jsou velmi výrazné a myslím si, že jejich eliminace by mohla velmi pomoct celému snažení. Zároveň je však potřeba myslet na to, že nasazení ETCS na české železnice se blíží a je potřeba tento milník nepromeškat. K tomu se může využít červnový Vědafest, kde je možné odprezentovat aktuální stav celého simulátoru. Případný úspěch by mohl vést k zisku potencionálního zákazníka.

	POMOCNÉ	**ŠKODLIVÉ**
VNITŘNÍ	Znalosti zákazníka Bakalářské práce Nápad Lektorského pracoviště Konzistence projektu pomocí PM	
VNĚJŠÍ	Nasazení ETCS na české železnice Červnový Vědafest	Rychlejší konkurence Prodloužení realizace ETCS v ČR

Tabulka 3.3 SWOT analýza projektu budoucího stavu

Na tabulce 3.3 je vidět žádoucí budoucí stav, do kterého by se měl projekt dostat, pokud by použil strategii **MIN-MAX**. Je na první pohled patrné, že by mělo dojít k odstraněních všech slabých stránek. O tom, jak se těchto slabých stránek zbavit, píšu v navazujících kapitolách.

3.4.2 Zbavení se závislosti na předmětech SP1 a SP2

Situace, ve které je projekt závislý na studentech a jejich účasti na předmětech SP1 a SP2, není ideální, zejména pokud to brání rychlému pokroku projektu. Protože projekt nemá dostatek financí na získání profesionálních vývojářů na plný úvazek, je třeba hledat alternativní řešení,

jak udržet současný tým studentů a zároveň zajistit pokračování projektu s minimální závislostí na těchto předmětech.

Jedním z potenciálních řešení je udržet si stávající studenty tím, že se jim poskytnou další příležitosti k práci na projektu. V současnosti je jedinou příležitostí tvorba bakalářské práce. Problémem však je, že vhodná témata se pomalu vyčerpávají a hlavně po dokončení bakalářské práce se nedaří studenty na projektu udržet. Podle mě, se dá využít jedna z následujících možností:

Propojení s dalšími předměty zní trochu divně, zvlášť, když závislost na předmětech SP1 a SP2 jsem označil v analýze jako jeden z největších problémů. Pokud by však došlo na navázání dalších předmětů, především těch, které jsou součástí magisterského studia, mohlo by dojit k udržení stávajících studentů, což je zatím v projektu palčivé téma. Studenti na magisterském studiu jsou o hodně zkušenější a celý vývoj by tak mohl probíhat efektivněji.

Kreditové stáže by mohly být velkým lákadlem, jak studenty na projektu aspoň částečně udržet. Je zřejmé, že kdyby stáže byly placené, motivace studentů zůstat by byla větší, ale nabídnout jim za jejich práci kredity by také mohla být možnost. Bylo by potřeba řádně vydefinovat náplň práce těchto stáží a v jaké pozici by byly vůči studentům z předmětů SP1 a SP2.

Komercionalizace projektu by velké množství problému vyřešila. Projekt je v současnosti od prodávání reálného projektu velmi daleko, ale dynamika trhu se může velmi měnit a pokud se v blízkých měsících povede úspěšná demo ukázka, může být vše jinak. Pokud dojde k rozhodnutí projekt komercializovat, muselo by dojít k velkým změnám v rámci řízení projektu.

V analýze (viz. kapitola 3.1.3) jsem zmínil, že projekt stojí na rozcestí, jak se k závislosti na předmětech SP1 a SP2 postavit. Za mě je simulátor v takovém stavu, že by bylo vhodné z něho udělat profesionální projekt. Může být nadále vázán na předměty SP1 a SP2 a tím poskytnout studentům zkušenosti, které v jiných předmětech nenajdou.

Kromě studentů však bude nezbytné mít také osobu, která je zapojena do projektu delší dobu, zná stav implementace všech částí a ví, kde se co nachází. Tato osoba by měla dohlížet na to, aby nedocházelo k větším změnám bez racionálního důvodu. Taková osoba by mohla být součástí spin-off firmy, která je založena za účelem využití a rozvoje duševního vlastnictví univerzity ve formě produktu. Měla by být náležitě placená za svou práci, nebo by mohla být zapojena do projektu v rámci jiných předmětů nebo kreditových stáží, jak je uvedeno v této kapitole výše.

3.4.3 Rozšíření týmové struktury

V současnosti na takto rozsáhlém projektu není nikdo, jehož náplň práce by odpovídala projektovému manažerovi nebo se nějak důkladně zabývala vedením týmů. Vedení projektu si je toho velmi dobře vědomo a do budoucna by o někoho takového celý tým rozšířila.

3.4.3.1 Vedoucí jednotlivých týmů z řad studentů

Projekt je primárně studentský a vedoucí jednotlivých týmů by měl také pocházet z řad studentů. Otázkou se nabízí, jakým způsobem do projektu studenty, kteří si tuto roli chtějí zkusit, získat. V tomhle směru se nejvíce nabízí využít oboru manažerské informatiky, který je součástí katedry softwarového inženýrství.

Studenti se v tomto oboru učí manažerský pohled na projekty, některé předměty jsou vyloženě zaměřený na řízení projektů a ETCS simulátor by jim mohl nabídnout rozšíření svých znalostí. Studenty z manažerské informatiky lze do projektu dostat pomocí dvou přístupů:

SP1 předmět je povinný nejen pro studenty softwarového inženýrství, ale i pro studenty z manažerského oboru. Tyto studenty by bylo vhodné do projektu přivést za každou cenu. Každý

tým, který v rámci SP1 (popřípadě SP2) vznikne, by měl mít jednoho studenta z tohoto oboru. Projekt je velmi rozsáhlý a je vhodné, aby součástí týmu byl někdo, kdo může plánovat a řídit různé procesy.

V následujících ročnících by tak bylo vhodné vypisovat v systému SOS[2], že jedno místo je rezervováno pro studenty vhodného oboru. Pokud by se povedlo domluvit s autory celého systému, mohlo by být výhodné udělat nějakou rezervaci v rámci základních funkcionalit systému.

Odborné předměty pro manažerskou informatiku se často točí kolem řízení projektu. Já sám jsem si těmito předměty prošel a připadaly mi velmi zajímavé. Jestli je však něco, co by mohlo být vylepšené, tak je to praktická část. Ta je zaměřena na práci v týmu, což je naprosto v pořádku, ale ten tým je složen pouze z lidí z manažerské informatiky. Díky tomu pak někteří studenti nemají možnost si pořádně vyzkoušet, jaké to je vést projekt.

Na tyto studenty bychom právě měli mířit a nabídnout jim možnost si vyzkoušet naučené věci na reálném projektu. Aby to mohlo fungovat, bude potřeba vést debatu s garanty jednotlivých předmětů a společně najít řešení, jak se s těmito předměty spojit.

Je potřeba si uvědomit, že toto řešení přináší určitá rizika. Jedním z nich je jednoznačně nedostatek zkušeností. Zároveň jsem v několika předchozích kapitolách zmiňoval, že velká závislost projektu na školních předmětech není vhodná. Tyto problémy by se daly vyřešit zkušenějším projektovým manažerem.

3.4.3.2 Projektový manažer

Do projektu by bylo velmi přínosné zapojit už zkušeného projektového manažera. Jeho působení na projektu by mělo být spíše globální než zasahovat do jednotlivých týmů. Otázkou však je, jak si může primárně školní projekt takového manažera finančně dovolit. V současnosti projekt nemá žádné finance, takže by externího manažera byla potřeba do projektu dostat jiný způsobem (např. možnost investice do založení firmy).

Druhou možností je vzít projektového manažera, který má určitou vazbu na univerzitu. V současnosti není prakticky škola nikterak zapojená, ačkoliv je projekt součástí dvou fakult. Na ČVUT existuje velké množství manažerů, kteří mají s řízením projektu velké zkušenosti a je možné, že se nám podaří s některým z nich domluvit na nějaké spolupráci.

Velkou výhodou takto získaného manažera jsou především jeho zkušenosti, které mohou projekt vylepšit a ten se tak může stát úspěšnějším. Pokud by projekt získal jakékoliv finance, myslím si, že by bylo vhodné je investovat do někoho, kdo by se projektu věnoval z tohoto pohledu.

3.4.3.3 Vzájemná spolupráce

Vzhledem k tomu, že na projektu zpravidla pracuje více než jeden tým, má smysl mít člověka zabývající se projektovým řízením z globálního pohledu nad všemi týmy, ale zároveň každý tým by měl mít svého vedoucího, který bude úzce spolupracovat s ostatními týmy a s užším vedením projektu.

Zkušenější manažer by měl mít na starost mimo jiné i přípravu koordinačních schůzek a obecně by měl zlepšit komunikaci mezi týmy (viz. kapitola 3.1.2). Společně s nejvyšším vedením by měl určovat dlouhodobou strategii projektu a následně kontrolovat dodržování definovaných cílů. Studentským manažerům by měl pomáhat a částečně je učit, jak vést projekt.

Vedoucí jednotlivých týmů by měly být z řad studentů. Vhodné by bylo, zdůraznit v aktuálním systému SOS, že je na tuto pozici potřeba někdo z oboru manažerské informatiky. Úkolem těchto vedoucích je primárně řídit tým samotný a přenášet požadavky od vedení projektu a nadřazeného projektového manažera. Takový člověk by se měl věnovat opravdu jen věcem, které se týkají řízení a plánování a programovat by měl opravdu jen minimum svého času.

[2]Systém pro správu projektů v rámci FIT ČVUT

3.4.4 Povinnosti projektového manažera na úrovni vedení

V předchozí sekci jsem se věnoval tomu, proč a jak by měl být tým rozšířen o lidi, jejichž hlavní prací by mělo být řízení celého projektu. V této části bych rád konkrétněji popsal povinnosti projektového manažera, který by měl projekt řídit ze strategického pohledu. Některé věci jsem už lehce naznačil dříve, ale zde je prostor se jim věnovat více.

3.4.4.1 Koordinační schůzky

Z mé analýzy je zřejmé, že koordinační schůzky v současné podobě nejsou tak efektivní, jak by mohly být. Ve světe velkých softwarových projektů je běžné, že tým stráví nezanedbatelný čas na různých schůzkách. Tyto schůzky je však potřeba řádně připravit a vést je správným směrem. To by za mě měla být povinnost projektového manažera na tomto projektu. Vedoucí ze stran učitelů nemají časovou kapacitu, aby byla taková schůzka dopředu připravena a studenti z jednotlivých týmu nejsou v pozici, kdy by něco takového měli dělat.

Za mě by koordinační schůzky měly probíhat jednou za dva týdny, případně i jednou za týden, zejména v případě blížící ho se důležitého termínu. Mezi schůzkami by tak měl projektový manažer shánět požadavky jak ze strany vedení (zákazník, učitelský vedoucí týmu), tak ze strany týmů (studentský vedoucí týmu, další členové). Tyto požadavky by měl vzít v potaz a vytvořit agendu, kterou s dostatečným předstihem (minimálně 2 dny před termínem schůzky) pošle všem zúčastněným stranám, aby se na schůzky připravily. Tato agenda pomůže dát koordinačním schůzkách řád a ty by tak mohly být kratšího rázu.

Po skončení koordinační schůzky by měl vytvořit záznam, který opět pošle všem stranám pro potvrzení, že uvedené údaje jsou v pořádku a žádné nechybí. Takto validovaný dokument je poté potřeba řádně uchovat, aby ho bylo jednoduché vždy dohledat.

3.4.4.2 Společné plánování projektu

Většina projektových manažerů jsou na projektech od toho, aby ho řádně plánovali a pak kontrolovali, zda je tento plán dodržován. Tato povinnost by tak pochopitelně neměla chybět ani na tomto projektu, protože je velmi komplexní a dlouhodobě mu přesnější plánování chybí. Problémem je však už tolik probíraná závislost na předmětech (viz. kapitola 3.1.3).

V tomto předmětu jsou studenti nabádání k tomu, aby sami odhadovali pracnost jednotlivých požadavků a ty poté naplánovali na celý semestr. Toto plánování by tak mohlo být v rozporu s tím plánem, který navrhl projektový manažer. Tato část v SP1 a SP2 je za mě velmi důležitá a není vhodné, abychom o to další týmy ochudili.

Proto plánování od projektového manažera nemusí být nějak extra důkladní a spíš, než plánovat jednotlivé týdny a měsíce, by měl plánovat na základě jednotlivých semestrů. Mimo to by bylo vhodné, kdyby pomohl i se sestavením plánu jednotlivým týmům. Tím jim může předat nejen zkušenosti, ale hlavně může zkontrolovat samotný plán a bude mít větší povědomí o tom, co se na projektu bude v nejbližších měsících dít.

Při plánování je potřeba aktivně komunikovat se zákazníkem a získat tím od něho různé požadavky. Požadavky by však měli chodit i od jednotlivých týmů. Může se jednat o různé technické věci, či další různá vylepšení, která můžou vést ke stabilizaci a k celkovému zlepšení projektu.

3.4.4.3 Kontrola dodržování plánů

Tento bod úzce souvisí s tím předchozím. Zatímco přesnější naplánování má probíhat i s pomocí vedení, kontrola dodržování tohoto plánu je už čistě jen na projektovém manažerovi. Každému vyhovuje přístup trochu jiný, takže zde nemá vyloženě cenu psát, jak by danou kontrolu měl kdokoliv provádět.

Důležitá však bude pravidelná komunikace s vedoucími týmy, případně s celým týmem. Není vhodné, aby zjišťování stavu jednotlivých týmů probíhalo vždy jen na koordinačních schůzkách, které by primárně měly mít jiný smysl.

Pravidelná kontrola je důležitá, aby byla možnost eliminovat případná rizika. Projekt by se tak nemusel kvůli případným problémům zastavovat, nebo se dokonce vracet zpět, jako tomu občas v aktuálním projektu při přepisu komponent bývá.

3.4.4.4 Osobní schůzky s členy týmu

V praxi bývají časté osobní schůzky nadřízeného s podřízeným. Obvykle jsou tyto schůzky označovány jako *one-to-one meetingy*. Pro projektového manažera může být časově náročné mít takové schůzky se všemi členy všech týmů. Proto by bylo vhodné tyto schůzky alespoň s vedoucími týmů, i když by bylo ideální setkat se individuálně s každým alespoň jednou za semestr. S vedoucími týmů by však bylo dobré se potkávat pravidelně, například po konci každé iterace, každý měsíc, nebo dokonce jednou za dva týdny.

Cílem těchto schůzek má být poskytnutí zpětné vazby, a to pro obě strany. Projektový manažer by měl objektivně ohodnotit práci osoby, se kterou má zrovna meeting. Je však potřeba, aby zde docházelo i k sebehodnocení a případné sebekritice. Projektový manažer by měl dostat i zpětnou vazbu o fungování celého projektu.

Mezi otázky, které by na těchto schůzkách mohli padnout, patří:

1. Za co by ses v poslední době pochválil?
2. Je něco, v čem naopak vidíš prostor ke zlepšení?
3. Jaká je nálada v týmu, pracuje se ti dobře?
4. Co ses za poslední dobu nového naučil?
5. Je něco, s čím ti mohu pomoct?

Tyto otázky by mohly poskytnout vhodnou zpětnou vazbu, není však nutné všechny použít.

3.4.4.5 Retrospektivní schůzky

Retrospektivní schůzky, jsou jedna z mnoho možností, jak od týmů získat zpětnou vazbu k celému projektu. V současnosti tento prvek chybí, protože není nikdo, kdo by se o něj vyloženě staral. To by právě mělo být povinností potencionálního nového projektového manažera.

Tyto retrospektivní schůzky by měly probíhat následovně:

1. Projektový manažer připraví veřejně dostupný dokument, který poskytne všem v týmu v dostatečném předstihu před konáním schůzky. Tento dokument by měl obsahovat alespoň tři různé kategorie, do kterých budou moct členové týmu přidávat své kartičky:

   MAD značí věci, které daného majitele kartičky štvou, například jak se některé věci (ne)řeší.

   SAD značí věci, na které by se podle majitele kartičky měl celý tým zaměřit.

   GLAD použije majitel kartičky v případě, že je s něčím opravdu spokojen.

   Nejedná se o konečný výčet, kategorií lze mít víc, pokud je manažer uzná za vhodné.

2. Každý člen týmu by měl ještě před schůzkou svoje kartičky vložit do příslušné kategorie. Věci, které hodnotí, se nemusí týkat jen práce v týmu, ale může se jednat i o technickou stránku projektu. Je zde zapotřebí vytvořit zpětnou vazbu, nad kterou bude možnost dále diskutovat.

3. Na dané schůzce, dojde k přerozdělování bodů k jednotlivým kartičkám. Díky tomu dojde k určení priorit pro jednotlivá témata, nad kterými bude poté prostor diskutovat.

4. Vytvoření akční kartičky je finálním krokem retrospektivních schůzek. Akční kartičky slouží k tomu, aby vzniklo nějaké řešení pro případné problémy. Tyto kartičky je poté potřeba vzít, vytvořit z nich případné úkoly a ty poté přerozdělovat. Na retrospektivní schůzce by mělo také dojít k určení toho, kdo je za tuto kartičku zodpovědný.

3.4.4.6 Vedení misí

Jak jsem už zmiňoval několikrát ve své analýze, na projektu funguje vícero týmů. Občas se stane, že nějaký požadavek, souvisí s vícero týmy. V tom případě, je potřeba, aby spolupráce mezi týmy byla daleko důslednější a můžeme se bavit o tom, že by měla být na každodenní úrovni. V tu chvíli je potřeba, aby vznikl menší tým (pracovní skupina), která bude pracovat na tzv. misi.

Vedoucí této mise, by měl být projektový manažer, neboť spolupráce mezi týmy spadá především do jeho kompetencí. Projektový manažer je zodpovědný za správné vydefinování požadavků a za vybrání členů týmu. Většinou by měl vybrat jednoho až dva členy z každé části projektu, která je do mise zainteresovaná.

Pro takto vybrané členy to znamená, že na chvíli nebudou dostupný svým týmům a budou se plně věnovat této misi. Součástí mise by měly být pravidelnější schůzky všech zúčastněných. Veškeré implementační změny, které během mise vzniknou, by měly být do kódu sjednoceny až v samotném závěru, proto by bylo vhodné, kdyby se vytvořila jedna větev, do které budou jednotlivé změny postupně přidávány.

3.4.4.7 Výpomoc pro vedoucí jednotlivých týmů

Vzhledem k tomu, že vedoucí jednotlivých týmů by měly být zpravidla studenti, je vhodné, aby jim byl projektový manažer k dispozici. Může se jednat o různé konzultace k jednotlivým problémům, které určitě nastanou. Rozhodně by však projektový manažer neměl dělat jejich práci.

3.4.5 Povinnosti vedoucích jednotlivých týmů

Vedoucí jednotlivých týmů jsou v každodenním kontaktu s týmem. V aktuálním stavu jsou vedoucí získáváni podobně jako ostatní členové týmu přes aplikaci SOS, která se používá pro předměty SP1 a SP2, a s největší pravděpodobností tomu tak bude i v dalších semestrech. Povinností vedoucího týmu je opravdu hodně a je důležité zmínit, že jeho úkolem na projektu určitě nemůže být primárně jen programování.

3.4.5.1 Plánování jednotlivých iterací

Jedním z hlavních úkolů vedoucího je plánování jednotlivých iterací. Od projektového manažera, zákazníka a učitelského vedoucího dostane tým vždy velké množství požadavků, které je potřeba priorizovat a řádně naplánovat. Za správnou prioritizaci je zodpovědný právě vedoucí. Měl by důkladně naplánovat úkoly na konkrétní týdny a poté kontrolovat jejich stav.

Je vhodné také přiřazovat jednotlivé úkoly ostatním členům týmu. Vedoucí je totiž zodpovědný za to, že každý v týmu má co dělat. Pro tuto kontrolu může použít jak schůzky, tak různé funkcionality webové aplikace GitLab.

3.4.5.2 Kontrolování stavu jednotlivých issue

S předchozím bodem úzce souvisí i tento. Jednotlivé issue mají možnost lépe evidovat svůj aktuální stav, což může vedoucímu usnadnit velké množství práce. Více informací o jednotlivých funkcionalit v GitLabu, které jsem implementoval pro jednodušší používání, je v kapitole 3.5.

Kontrolování stavu je důležité nejen pro jednotlivá issue, ale z takto získaných informací lze kontrolovat stav celé iterace, potažmo semestru. Lze takto odchytit případné rizika, nebo jiné

problémy, které během vývoje mohou nastat. O tom, že se něco nestíhá, nebo je nějaký jiný problém, je potřeba vždy informovat projektového manažera.

3.4.5.3 Odevzdání iterací

V kapitole 3.1.4.1 zmiňuji, že součástí předmětu SP1 jsou v aktuálním plánu tři iterace. Na konci těchto iterací dochází k odevzdání dokumentu s patřičným obsahem. Za tento dokument a především za jeho včasné odevzdání je primárně zodpovědný vedoucí týmu.

To neznamená, že celý dokument bude vytvářet on sám, ale je potřeba při plánování jednotlivých iterací na tuto podstatnou část myslet. Dokument se může klidně psát v průběhu iterace, není potřeba čekat až na poslední týdny. Je důležité komunikovat v tomto směru především s učitelským vedoucím, který může mít různé požadavky na to, co si v dokumentu přeje mít uvedeno.

Součástí iterace je také potřeba přerozdělit body. To je také v kompetenci samotného vedoucího týmu. On však musí být také ohodnocen, a to zpětnou vazbou týmu, případně projektovým manažerem.

Pro hodnocení je potřeba brát v potaz vícero věcí. Je pochopitelné, že strávený čas je jeden ze vstupních parametrů, které je vhodné v tomto směru použít. Někdo však může odvést velké množství práce, za méně času. Takový člověk by tak neměl být nikterak extra pokutován. Další věci pro hodnocení je například komunikace, správné plnění úkolů, dodržování interních pravidel atd.

3.4.5.4 Osobní schůzky se členy týmů

O této povinnosti jsem mluvil už v případě projektového manažera (viz. kapitola 3.4.4.4) a je pochopitelné, aby schůzky v podobném duchu měl i vedoucí týmu s ostatními členy. Schůzky by měly probíhat stejně, bylo by vhodné, o jejich výsledcích případně informovat projektového manažera.

3.4.5.5 Prezentování výsledků

Na konci semestru, kdy dochází k odevzdání závěrečné iterace a uzavření celého předmětu (SP1 i SP2) je vhodné, aby výsledná práce, byla představena většímu okruhu lidí, než jen vedoucímu z řad učitelů. Proto by měl vedoucí týmu (s pomocí ostatních) vytvořit prezentaci, kterou představí nejen zákazníkovi, ale i projektovému manažerovi a dalším lidem ve vedení projektu.

Prezentace by měla obsahovat představení všech nových funkcionalit, opravení chyb a další důležité informace a změny, které se během semestru odehrály. Bylo by skvělé, kdyby na konci prezentace bylo zmíněno i to, co by se mělo udělat v dalším běhu.

3.5 Implementace postupů do webové aplikace GitLab

V této části práce bych rád popsal, jak jsem implementoval návrh z kapitoly 3.2. Prakticky vše jsem mohl udělat rovnou ve webové aplikaci a nemusel jsem používat žádný vzdálený přístup pomocí API.

3.5.1 Implementace práce s Gitem

Práci s Gitem není tak přímočaré implementovat, protože to do značné míry je na zodpovědnosti jednotlivých lidí. Kvůli tomu jsem vytvořil wiki stránku, která je součástí ETCS_TOGETHER repozitáře. Ta je dostupná všem vývojářům a jedním z jejich prvních úkolů je seznámení se s touto stránkou.

Co se týká větví, tam jsem musel pár věcí změnit. Každý repozitář má svojí *master* větev, kterou bylo potřeba nastavit jako *protected* a zamezit různým špatným přístupům, jako je například upravování kódu přímo z této větve.

Dále bylo potřeba vytvořit *develop* větev s příslušným přídavkem, který označuje aktuální semestr. Za tu dobu, co jsem na projektu, jsem musel tento proces udělat třikrát. Vytvořená větev se musí stejně jako *master* nastavit jako *protected*, zároveň je potřeba tuto větev nastavit jako *default* větev celého repozitáře, což urychlí jiné procesy.

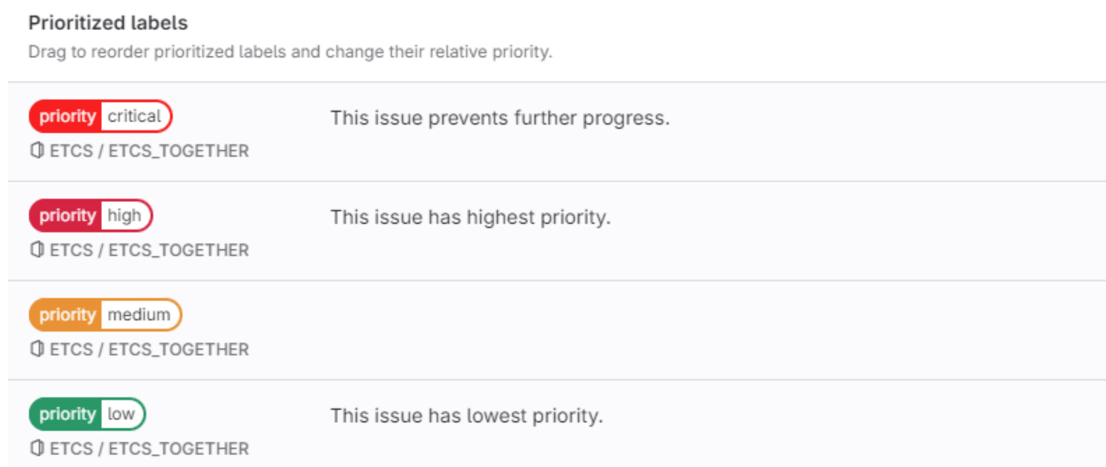
3.5.2 Implementace správy úkolů

Jak jsem už v návrhu naznačoval, spousta věcí bude fungovat díky nativní funkcionalitě, která se označuje jako *Labels*. Tyto štítky mi umožní splnit veškeré požadavky. Jejich funkcionality jsem konkrétně popsal v jednotlivých dílčích částech kapitoly 3.2.2.

Kromě štítků se však zaměřím i na dvě další nativní funkcionality, které jsou součástí webové aplikace GitLab. Tyto funkcionality slouží k přehlednějšímu zobrazování jednotlivých úkolů. Jedná se o *Issue board*, kde je možné vytvořit různé náhledy a filtry a o tzv. *Milestones*, které mohou pomoct při plánování úkolů k jednotlivým iteracím.

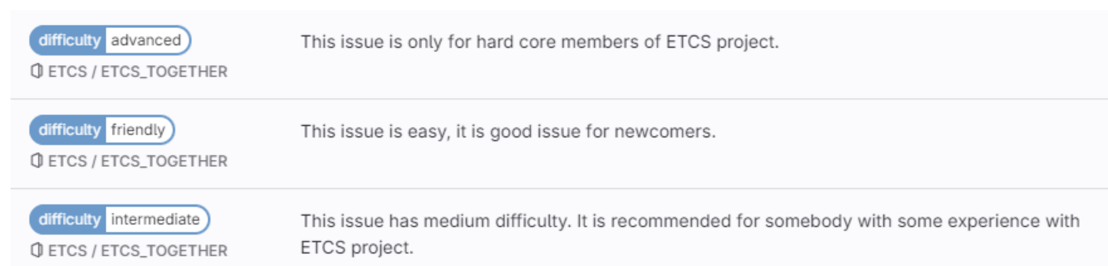
3.5.2.1 Štítky

Na obrázku 3.2 je vidět implementace priorit. Při vytváření těchto štítků jsem se snažil prioritu vyjádřit nejen pomocí názvu, ale i barvy, která může prioritu správně evokovat. Štítky jsou popsány v angličtině (hlavní jazyk pro vývoj v našich repozitářích), aby bylo jasné, co která priorita znamená.



Obrázek 3.2 Štítky priority ve webové aplikaci GitLab

Dále bylo potřeba implementovat i náročnost. Na obrázku 3.3 je vidět, v jaké podobě jsou momentálně štítky, které slouží těmto účelům. Oproti prioritě jsem se rozhodl v tomto případě použít jednotnou barvu, nemyslím si, že by různé barvy měly nějaký pozitivní vliv na rozlišitelnost jednotlivých úkolů.



Obrázek 3.3 Štítky náročnosti ve webové aplikaci GitLab

Asi nejdůležitější štítek, je ten, který slouží pro označení aktuálního stavu. Jeho implementaci můžete vidět na obrázku 3.4. Barevně jsem se rozhodl použít kombinaci předchozích prvků. Je tu různé barevné schéma, kdy stavy pro *resolved* a *postponed* jsou barevně odlišné. K tomu jsem se rozhodl především kvůli tomu, že tyto stavy úkolu potřebují vícero pozornosti. *Postponed* úkoly je potřeba pravidelně kontrolovat, jestli například už není možnost s úkolem zase něco udělat. *Resolved* zase značí, že úkol by mohl být hotový a je potřeba ho zkontrolovat.

Ostatní stavy mají stejnou barvu, protože není vyžadováno tolik pozornosti. Zvažoval jsem použití specifické barvy i pro stav *waiting-for-confirmation*, ale nakonec jsem se k tomu neodhodlal. Pokud by bylo potřeba se danému úkolu v tomto stavu více věnovat, je vhodné přiřadit správnou prioritu.

Obrázek 3.4 Štítky stavu ve webové aplikaci GitLab

Posledním povinným štítkem je ten, který označuje typ daného úkolu. Na obrázku 3.5 je opět k nahlédnutí, jak dané štítky vypadají ve webové aplikaci GitLab. Pro typ jsem se rozhodl použít jednotnou barvu, nevidím žádný důvod, proč mít rozšířené barevné schéma pro typ úkolu.

Štítek	Popis
type refactoring ETCS / ETCS_TOGETHER	This issue is for refactoring the code.
type overhead ETCS / ETCS_TOGETHER	This label is used for an issue, which is somehow related to the meeting, lessons in SWI etc.
type management ETCS / ETCS_TOGETHER	This issue is for team leaders.
type feature ETCS / ETCS_TOGETHER	This issue means that it has some new functionality.
type enhancement ETCS / ETCS_TOGETHER	This issue aims to improve or change something that is already part of a project.
type documentation ETCS / ETCS_TOGETHER	This issue is just documentation.
type bug ETCS / ETCS_TOGETHER	The issue is reporting as bug.
type bachelor-support ETCS / ETCS_TOGETHER	This issue means that we are supporting student working on their bachelor's thesis.

Obrázek 3.5 Štítky typu ve webové aplikaci GitLab

Kromě těch povinných štítků, jsem vytvořil i ty nepovinné, které slouží pro přidání dalších informací. Obě dvě varianty můžete vidět na obrázku 3.6. Pro štítky jsem použil opět jednotnou barvu.

Pro *needs* jsem zvolil o něco výraznější barvu, která na sebe upoutá větší pozornosti. To je vhodné především kvůli tomu, že daný úkol, vyžaduje nějakou akci a dokud tato akce neproběhne, tak se nemůže úkol splnit.

Special štítky mají nevýraznou žlutou, používají se pro speciální označení úkolů. Na obrázku je vidět aktuální použití, kdy existují štítky pro autory bakalářských prací a pro nově příchozí do projektu.

Label	Description
needs analysis ETCS / ETCS_TOGETHER	This issue needs some analysis before making it.
needs another-issue ETCS / ETCS_TOGETHER	This issue another issue to be done before working on this.
needs discussion ETCS / ETCS_TOGETHER	This issue needs some discussion with the team.
needs review ETCS / ETCS_TOGETHER	This issue needs review in merge request.
needs testing ETCS / ETCS_TOGETHER	This issue needs testing from somebody else.
special caslama1 ETCS / ETCS_TOGETHER	This label is for issue, which are for caslama1 bachelor thesis. Only him is responsible for this task and can do it.
special neprater ETCS / ETCS_TOGETHER	This label is for issue, which are for neprater bachelor thesis. Only her is responsible for this task and can do it.
special newcomers ETCS / ETCS_TOGETHER	This issue is meant for newbies so they can learn on that.
special veselo21 ETCS / ETCS_TOGETHER	This label is for issue, which are for veselo21 bachelor thesis. Only him is responsible for this task and can do it.

Obrázek 3.6 Nepovinné štítky ve webové aplikaci GitLab

3.5.2.2 Milestone

Označuje nějaký časový úsek, na jehož konci by měl být nějaký cíl. V kontextu tohoto projektu, se nabízí, udělat *Milestone* pro každou iteraci. Tohle umožní vhodně plánovat a přiřazovat jednotlivé úkoly k jednotlivým iteracím.

Není potřeba všechny úkoly přiřazovat do nějakého *Milestone*, můžou zůstat bez jakéholi zařazení. V případě, že se stihnou všechny potřebné úkoly, můžou se nový přidávat.

Tuto funkčnost by měli využívat pouze vedoucí týmu společně s projektovým manažerem. Není vhodné, aby úkoly byly rozřazovány někým jiným, neboť by to mohlo vést k nekonzistenci a nedorozuměním.

3.5.2.3 Issue board

Funkcionalita *Issue board* ve webové aplikaci umožňuje vytvořit různé pohledy na dané úkoly. Rozhodl jsem se tuto funkčnost využít a vytvořit několik různých pohledů.

Základní pohledy jsem vytvořil podle štítků zmíněných v předchozích odstavcích. Bylo potřeba použít jen některé z nich, a proto jsem vytvořil *board* pouze pro *difficulty*, *needs*, *priority* a *state*.

Z mého pohledu jsou nejpoužívanější *issue board* pro týmové meetingy (nazývaný *StandUps*) a aktuální *Milestone*. Tyto seznamy jsou užitečné pro plánování a kontrolu aktuální práce. V prvním z nich jsou úkoly strukturovány podle jednotlivých osob, což usnadňuje sledování, kdo na čem pracuje. Druhá varianta se zaměřuje na aktuální iteraci a obsahuje úkoly, které jsou pro danou iteraci nezbytné.

3.6 Implementace skriptu na měření času

V této sekci bych rád popsal (nejen) moje trápení s jazykem *Python*, který jsem použil pro vytvoření skriptu na měření času. Vycházet zde budu především z návrhu tohoto skriptu, který se nachází v kapitole 3.3.

3.6.1 Získávání dat z webové aplikace GitLab

V návrhu jsem zmiňoval, že bych rád použil GitLab API, které má dokumentaci na oficiálních stránkách. Používání ve výsledku nebylo tak přímočaré, jak jsem očekával a bylo pár problémů, se kterými jsem se musel vypořádat.

Samotné připojení bylo velmi jednoduché, bylo potřeba pouze ve webové aplikaci vytvořit *token*, který se poté při připojování použije. Tohle slouží k zabezpečení projektu, aby data z něho nemohl získat kdokoliv.

Problém nastal při získávání samotných dat ze serveru. Pro větší efektivitu, API dotaz, který má získat všechny úkoly daného projektu, vrací pouhý zlomek a je potřeba se pravidelně dotazovat v cyklu na další. Dotaz tak není přímočarý a je potřeba větší úsilí.

Největším problémem však pro mě bylo, z úkolu dostat konkrétní údaj o zaznamenaném čase. Dotaz pro jednotlivá *issue* sice vrací údaj o zaznamenaném čase, ale ten není použitelný pro náš požadavek. Tento údaj bere veškerý naměřený čas, není možné získat konkrétní osoby, který si daný čas zaznamenaly. Zároveň není možné tento čas filtrovat v závislosti na datu.

Rozhodl jsem se, že tento problém vyřeším čtením všech komentářů, které v daném úkolu jsou. Zjistil jsem totiž, že přidání času k úkolu se reflektuje nejen tím, že se zvedne celkový čas, ale zároveň se vytvoří komentář, který v sobě obsahuje všechny podstatné údaje. Řešení to není ideální, ale během implementace skriptu jiná možnost nebyla. Získávání potřebných informací popisuji níže.

3.6.2 Vytvořené funkce

Jedná se o skript malého rozsahu, proto nemělo smysl vytvářet třídu. Vznikly tedy jen funkce, které bych rád zde popsal:

get_time_from_issue je pro získání času z jednoho konkrétního úkolu. Jejím hlavním cílem je tedy projít všechny komentáře, které v daném *issue* jsou a z nich případně získat čas. Na ukázce kódu 3.1 můžete vidět popsanou funkci pomocí komentářů, které slouží k vytvoření představy, co se vnitřně v dané funkci děje.

```
def get_time_from_issue(issue_id, total_dictionary, from_date, to_date):
    # Vytvoření API dotazu a zisk odpovědi ze serveru pro dané issue (issue_id)
    # Iterace skrz všechny komentáře v daném úkolu
        # Kontrola, zda je z intervalu (from_date, to_date) a obsahuje čas
        # Zavolání funkce na parsování komentáře
```

Listing 3.1 Funkce get_time_from_issue

parse_time_tracking_string slouží pro získání přesného času z daného komentáře, který je ve formě textového řetězce. V této funkci dochází k převádění textového řetězce do desetinných čísel, která reprezentují naměřený čas v hodinách. Problém mi v tomto směru dělalo, že čas lze i odebrat, takže bylo potřeba na toto myslet. Vnitřní procesy ve funkci je vidět na ukázce kódu 3.2.

```
def parse_time_tracking_string(time_str, total_dictionary, username):
    # Použití regex pro hledání časového údaje v time_str
    # Zjištění, zda se jedná o přičítání nebo odebírání času
    # Iterování skrz všechny nalezené shody z regex hledání
        # Zjištění časové jednotky jednotky (minuta, hodina, den, měsíc atd.)
        # Převedení všech údajů do hodin
        # Přičtení (odečtení) hodnoty v total_dictionary pro username
```

Listing 3.2 Funkce parse_time_tracking_string

check_date slouží pro ověření, že datum je v konkrétním intervalu, včetně krajních bodů. Kompletní implementace je vidět na ukázce kódu 3.3.

```
def check_date(from_date_str, to_date_str, date_str):
    from_date = datetime.strptime(from_date_str, "%Y-%m-%d")
    to_date = datetime.strptime(to_date_str, "%Y-%m-%d")
    date = datetime.strptime(date_str[:10], "%Y-%m-%d")
    if from_date <= date <= to_date:
        return True
    else:
        return False
```

Listing 3.3 Funkce check_date

get_issues_from_date filtruje všechny *issue*, které se v projektu nachází, v závislosti na jejich datu poslední aktualizace. Tento proces se provádí na začátku celého skriptu. Vrací seznam identifikátorů všech *issue*, které vyhovují danému intervalu. Na ukázce kódu 3.4 je vidět, jak orientačně funkce funguje.

```
def get_issues_from_date(from_date, to_date):
    # Vytvoření API dotazu a načtení odpovědi, která obsahuje seznam úkolů
    # Iterace skrz všechny vrácené hodnoty
        # Použití funkce check_date pro kontrolu aktualizace
            # Vložení id daného issue do návratové hodnoty
```

Listing 3.4 Funkce get_issues_from_date

get_time_value slouží pro zjištění, zda daný čas se má k celkové části přičíst nebo odečíst. Osobně jsem měl s implementací této funkce největší problém. Musel jsem procházet jednotlivé komentáře a zjistit, jaké všechny možnosti se při práci s časem mohou vyskytnout. Zjistil jsem, že existují pouze tři možnosti a jenom jedna z nich („added“) čas přičítala. Implementaci funkce je vidět na ukázce kódu 3.5.

```
def get_time_value(time_str):
    if "added" in time_str: return 1
    else: return -1
```

Listing 3.5 Funkce get_time_value

print_time_track je čistě jen pro výpisové účely. Cílem této funkce je umožnit jednoduchý výpis získaných dat. Veškerý výpis probíhá do konzole. Dlouho jsem uvažoval nad tím, jaký formát výpisu si vybrat a nakonec jsem zvolil velmi jednoduchou variantu. Tato funkce tedy vykresluje dvojice *username* a celkově naměřený čas v daném časovém intervalu. Přesná implementace je vidět na kódu 3.6.

```
def print_time_track(total_dictionary):
    for username in total_dictionary:
        print(f"  {username}:\t{total_dictionary[username]:>7.2f} h")
```

Listing 3.6 Funkce print_time_track

get_statistics slouží pro získávání statistických dat. Především jde o celkový součet a o průměr. Tyhle data se můžou hodit pro případné hodnocení členů týmu a určení těch, kteří na projektu odpracovali nadprůměrné množství času. Přesnou implementaci můžete vidět na kódu 3.7.

```
def get_statistics(total_dictionary):
    sum = 0.0
    for username in total_dictionary:
        sum += round(total_dictionary[username],2)
    avg = sum / len(total_dictionary)
    return sum, avg
```

Listing 3.7 Funkce get_statistics

3.6.3 Hlavní průchod skriptem

V předchozích kapitolách jsem představil jednotlivé funkce, které se postupně v kódu používají. Vzhledem k tomu, že se jedná o jednoduchý skript, neexistuje ani žádná funkce *main*.

Nejdříve je potřeba získat vstupní údaje od uživatele. K tomu používám knihovnu *sys*. Skript bere celkem čtyři různé vstupní údaje. První se ukládá do proměnné *from_date* a používá se pro označení časového intervalu, ve kterém chceme čas sčítat. Druhým krajním bodem je poté proměnná *to_date*, která je druhým vstupním argumentem. Třetí argument je také časový, a označuje aktuální den (proměnná *today*). Ta se využívá pro filtrování jednotlivých *issue* na začátku skriptu, protože datum poslední aktualizace úkolu může být až v pozdějším termínu, než ve kterém chci časy sčítat. Kdybych tedy i úkoly filtroval jen na základě *to_date*, mohl bych některé zaznamenané časy minout. Poslední vstupní parametr je klíč k Rest API připojení k GitLab serveru. Tento klíč je uložen v proměnné ACCESS_TOKEN.

Po tomto načtení všech dat, už je možné provádět úkony. Nejdříve je potřeba vyfiltrovat úkoly, aby mi zůstaly jen ty, které byly naposledy aktualizovány ve vhodném období. K tomu používám funkci *get_issues_from_data*, kterou provolám s parametry *from_date* a *today*.

Následně skript prochází skrz všechny nalezené úkoly a postupně přičítá všechny naměřené časy do jednoho velkého *dictionary*. To se skládá z *username* jednotlivých členů pracujících na projektu, které se mapuje na hodnotu, která reprezentuje odpracovaný čas v hodinách. Z těchto hodnot se poté vypočítávají statistiky (celkový součet a průměr).

V poslední fázi dochází k tomu, že skript všechny data vypisuje v čitelném formátu do konzole. Při výpisu dochází k lehkému zaokrouhlení údajů, které však pro potřeby skriptu nemá žádné vedlejší následky.

3.6.4 Používání skriptu

Jedním z požadavků pro tento skript, byla jeho automatizace. S tímto požadavkem jsem tedy počítal a pro jeho realizaci jsem se rozhodl použít CI/CD (Continuous Integration / Continuous Deployment) v rámci projektového repositáře ETCS_TOGETHER. Zde jsem nastavil automatické spouštění skriptu každý den ve večerních hodinách, takže vedoucí daného týmu může mít vždy nejaktuálnější údaje.

V následujících odstavcích bych rád představil, co vše jsem musel ve webovém prostředí GitLab nastavit, aby vše fungovalo. Zároveň z tohoto textu bude patrné, jak některé údaje v budoucnosti aktualizovat.

3.6.4.1 Nastavení automatického spouštění pipeline

Všechny naše repositáře, které se používají na projektu, mají v sobě soubor *.gitlab-ci.yml*. Tento soubor v sobě obsahuje definici několika úkolů, které se mají stát, v případě, že někdo přidá do repositáře svoje změny. Já jsem do tohoto souboru v již zmíněném společném repositáři přidal definici úkolu, který má spustit daný skript (viz. ukázka kódu 3.8).

Jakmile jsem měl takto definovaný úkol, mohl jsem zařídit jeho automatické spouštění. K tomu jsem musel vytvořit v sekci *Settings* → *CI/CD* → *Variables* několik proměnných, který skript musí využívat. Jedná se o FROM_DATE, TO_DATA a USED_TOKEN. Tyto proměnné slouží pro určení intervalu, ve kterém uživatel chce časy sčítat. Tyto data musí být ve formátu YYYY-MM-DD. USED_TOKEN je klíč (token), který se používá pro připojení přes GitLab API. Tato proměnná musí být maskovaná.

```
time_tracking:
  stage: time_tracking # Zařazení do správné fáze
  script:
    - current_date=$(date "+%Y-%m-%d")
    - cd ./time-tracking
    - echo Today is $current_date.
    - python ./timetracking.py $FROM_DATE $TO_DATE $current_date $USED_TOKEN
  only:
    - schedules # Úkol se může spouštět jen pomocí plánování
```

Listing 3.8 Definování úkolu v .gitlab-ci.yml

Posledním krokem bylo potřeba nastavení automatického spouštění. K tomu jsem opět použil webové prostředí. Nastavení takto automatických spouštěných úkolů se dá udělat v sekci *Build* → *Pipeline schedules*. Zde jsem klikl na tlačítko *New schedule* a vyplnit údaje tak, jak je vidět na obrázku 3.7. Zde je vidět, že se spouští každý den ve 21 hodin středoevropského času. Větev, nad kterou se tento úkol spouští je *master*. Proměnné nejsou žádné, protože jsou definované v globálním nastavení projektu.

Edit Pipeline Schedule

Description

Time tracking

Interval Pattern

Every day (at 3:28pm)

Every week (Friday at 3:28pm)

Every month (Day 0 at 3:28pm)

Custom

0 21 * * *

Set a custom interval with Cron syntax. What is Cron syntax?

Cron timezone

[UTC+2] Prague

Select target branch or tag

master

Variables

Variable | Input variable key | Input variable value

Activated

Save changes | Cancel

Obrázek 3.7 Nastavení plánovače spouštění pipeline

3.6.4.2 Kontrolování výsledku

Vyhodnocení naměřených časů je klíčovou součástí skriptu a díky navrženému implementačnímu přístupu není problém. Výsledky jsou přístupné prostřednictvím webové aplikace GitLab.

Pro zobrazení těchto výsledků je třeba přejít do sekce *Build* → *Jobs* v rámci repozitáře ETCS_TOGETHER. Zde lze nalézt všechny provedené úkoly. Úkol s názvem *time_tracking* obsahuje relevantní data po dokončení běhu skriptu. Podobný výsledek je možné vidět na obrázku 3.8, kde jsou k dispozici všechna potřebná data pro určení časových intervalů.

V prostřední části je seznam všech vývojářů (a případně manažerů), kteří během daného časového intervalu vykonali práci, kterou zadali k daným úkolům. Osoby jsou řazeny abecedně podle svého uživatelského jména. Pod tímto seznamem jsou uvedeny jednoduché statistiky, které mohou být použity pro případné určení bodového ohodnocení.

```
$ python ./timetracking.py $FROM_DATE $TO_DATE $current_date $USED_TOKEN
######################### Time Tracking #########################
#  From:    2024-02-19                                          #
#  To:      2024-06-30                                          #
#  Today:   2024-04-24                                          #
#################################################################
  anonym01:      108.18 h
  anonym02:      124.15 h
  anonym03:       49.33 h
  anonym04:       52.88 h
  anonym05:        1.50 h
  anonym06:        8.82 h
  anonym07:       56.13 h
  anonym08:       88.48 h
  anonym09:      109.72 h
  anonym10:       32.92 h
#################################################################
  SUMMARY:       632.11 h
  AVERAGE:        63.21 h
#################################################################
```

Obrázek 3.8 Výsledky timetracking skriptu ve webové aplikaci GitLab

Kapitola 4

Komponenta JRU

Tato kapitola se zaměřuje na vše podstatné okolo komponenty JRU. Je v ní analýza aktuálního stavu celého simulátoru i samostatné komponenty včetně požadavků na její implementaci. Dále je v kapitole kompletní návrh samotné komponenty i aplikace pro zobrazení logů z nově vzniklého logovacího systému. V posledních částech je popis implementací obou částí a informace o testování nových funkcionalit.

4.1 Analýza stavu vývoje ETCS simulátoru

V této sekci bych se rád zaměřil na to, v jakém stavu byl ETCS simulátor před tím, než jsem do něho vstoupil v rámci předmětu SP1. Z pohledu projektového řízení, zde došlo ke spojení vícero samostatných projektů do jednoho. Za každou komponentu (DMI, EVC a RBC) byl v minulosti zodpovědný jiný tým a každá měla svá pravidla, které bylo potřeba sjednotit. Před sjednocením však muselo dojít k důkladné analýze, kterou zde popíšu.

4.1.1 DMI

Komponenta DMI byla v době přebírání projektu jedinou komponentou, která disponovala grafickým rozhraním. Díky tomu byla struktura kódu jedinečná a pro většinu z týmu špatně čitelná. Jako grafická knihovna zde byla použita knihovna SDL (Simple DirectMedia Layer), ke které byla vytvořena vlastní nadstavba, která se pak volala v jednotlivých částech. Jako architektura zde byla použita MVC (Model-View-Controller) architektura, kdy jednotlivé části obrazovky DMI byly rozděleny podle oficiální dokumentace.

Komunikace s EVC byla řešena pomocí příkazu `switch`, který rozděloval zprávy na základě jejich identifikátorů. Později se volaly jednotlivé metody, které byly zodpovědné za načtení informací ze zprávy. V DMI docházelo k dvojitému ukládání a následnému kopírování dat. Informace ze zpráv byly jednak uloženy ve třídě zodpovědné za komunikaci, ale i ve třídě, ze které čerpaly data jednotlivé modely.

Funkčnost komponenty byla velmi dobrá, úpravy byly minimální a jednoduché. Díky tomu byla možnost se věnovat přepisu některých špatných částí, jako například již zmíněné kopírování dat, nebo vyřešení narůstající náročnosti na paměť.

I přesto došlo k rozhodnutí, že je potřeba komponentu DMI přepsat, aby lépe zapadala do projektu se společnou architekturou. Více informací o přepisu je možné získat z bakalářské práce Terezy Neprašové [22].

4.1.2 EVC

Komponenta EVC nám byla představena jako nově přepsaná, s funkční architekturou a ve velmi dobrém stavu. Kód oproti ostatním komponentám opravdu vypadal lépe, byl ve všech souborech konzistentní, takže se v něm dalo lépe orientovat. Docházelo zde ke správnému použití polymorfismu, architektura byla postavená na dvou základních konceptech: *MessageHandler*[1] a *Service*[2].

Velkým nedostatkem však bylo, že tato komponenta nikdy nebyla testována. Víc jak osm týdnů jsme v rámci předmětu SP1 opravovali banální chyby, které v EVC byly. Probíhalo to tak, že jsme se snažili rozjet celý simulátor postupně. Avšak vždy jsme narazili na nějaké nesrovnalosti v EVC, které nás zastavily. Tato zkušenost později vedla k myšlence vytvořit vlastní logovací systém, který by mohl být použit pro snazší objevování chyb v kódu.

Po opravě chyb, jsme se rozhodli architekturu z EVC aplikovat i do jiných komponent. Při procesu očišťování funkcionalit za účelem zisku čisté architektury, jsme objevili zásadní nedokonalosti v celém návrhu. Díky tomu, běželo celé EVC v jednom jediném vláknu, i když bylo navržené pro vícevláknovost. Tyto poznatky byly později použity pro návrh architektury nové, která v mnoha věcech vychází z architektury použité v EVC.

4.1.3 RBC

RBC nám ze všech komponent připadalo nejhorší. Celý kód byl hodně postaven na skriptování, chyběl jakýkoliv náznak architektury. Nejhůř vypadala sekce, která řešila příchozí zprávy. Zde se nacházel jeden velký `switch`, který v závislosti na identifikátoru dané zprávy skočil do sekce kódu, kde se vykonala nějaká práce. Na stav RBC komponenty jsme byli předem varováni a po analýze jsme dospěli k závěru, že bude potřeba tuto komponentu přepsat. O tom, jak přepis komponenty probíhal se můžete dočíst v bakalářské práci Ondřeje Veselého [23].

Co se týká funkcionality, tak tam jsme větší nesrovnalosti nenašli. I když byl kód velmi špatně napsán, během testování byl takřka vždy funkční a úsilí vynaložené k opravám případných chyb tak bylo minimální.

4.1.4 Komunikace mezi komponentami

Za nás jedním z hlavních problémů, který pramenil z toho, že před námi pracovalo na komponentách více týmů, byla špatně vydefinovaná komunikace mezi komponentami. Jednotlivé zprávy měly například jinak pojmenované proměnné, mnohokrát se stalo, že byly i jiného typu. Tým strávil velké množství času, když se snažil objevit, proč simulátor nefunguje a proč jednotlivé komponenty zdánlivě bezdůvodně padají.

4.1.4.1 MQTT

Veškerá komunikace mezi komponentami aktuálně probíhá pomocí MQTT protokolu. V následujících odstavcích bych rád popsal tento protokol a analyzoval jeho výhody a nevýhody.

MQTT je nejpoužívanějším protokolem v rámci IoT (Internet of Things). Protokol je řízen událostmi a propojuje zařízení pomocí vzoru *publish-subscribe*. Odesílatel (*Publisher*) a příjemce (*Subscriber*) spolu komunikují pomocí tzv. *Topics*. Komunikace mezi nimi zprostředkovává MQTT *broker*. Ten přijímá všechny zprávy, které byly odeslány, a ty poté filtruje (na základě zaevidovaných *Topics*) jednotlivým příjemcům. Dvě strany spolu nejsou nikterak propojené, což má své výhody i nevýhody. [24]

Základní výhodou je, že daný *Topic* může odebírat vícero příjemců. Díky tomu není potřeba, pokud to dané použití vyžaduje, posílat zprávu dvakrát. Velký význam to má pro logovací systém a implementaci komponenty JRU.

[1] Třída zodpovědná za správné vyřešení příchozí zprávy
[2] Třída slučující funkcionalitu nebo data

Nevýhodou může být velká zátěž na MQTT *broker*. V ETCS simulátoru se během jedné sekundy pošle více jak 30 zpráv, což může mít negativní vliv na jeho efektivitu. Kromě toho má MQTT vnitřně zabudovanou kontrolu pro případný DDOS útok, častokrát se nám tak stalo, že jedna z komponent byla označena za potenciální hrozbu a došlo k její odpojení.

Do toho funkcionalita MQTT, kdy jsou všechny zprávy posílány do MQTT *broker*, neodpovídá reálné implementaci ETCS. V budoucnosti jsou určité náznaky, že by část simulátoru mohla komunikovat s reálnou implementací. V tom případě by se musel MQTT opustit.

4.2 Analýza komponenty JRU

Z předchozích odstavců vyplývá, že v projektu chyběl logovací systém, který by mohl vývojářům usnadnit hledání chyb. Po analýze problematiky ETCS jsem došel k závěru, že je možné využít komponentu JRU a upravit ji nad rámec ETCS tak, aby se stala logovacím systémem.

Před samotným začátkem implementace bylo potřeba udělat analýzu již vzniklého, ale nepoužívaného, kódu. I v této starší formě komponenty došlo k používání MQTT protokolu ke komunikaci se zbytkem simulátoru a veškeré zprávy se ukládaly do databáze.

Pro naši potřebu však mnoho věcí chybělo. Použitá architektura byla velmi jednoduchá a přibližovala se skriptování podobně jako třeba v RBC. Vzhledem k vytváření nové architektury pro RBC bylo rozhodnuto, že se tento přístup použije i v JRU.

V reálném ETCS dochází k tomu, že do JRU zprávy posílá pouze EVC. Pokud bychom se tohoto principu drželi i v rámci simulátoru, docházelo by k velkému množství kopírování informací a to může mít pro MQTT komunikaci fatální následky. Pro celý simulátor by se více hodilo, kdyby JRU poslouchalo veškerou MQTT komunikaci a komponenty tam mohli pouze logovat svůj vlastní stav.

Připojování do databáze je z hlediska vývojářů spíše zátěží. Pro naše interní testování není vhodné, aby se data ukládaly někde vzdáleně. Pro plánované užívání komponenty JRU z hlediska vývoje je tedy výhodnější ukládat zprávy a logy do souboru.

4.2.1 Požadavky na komponentu JRU

V této sekci bych rád bodově popsal požadavky na komponentu JRU. Požadavek jednoduše popíšu a určím, kdo je jeho autorem. V závěru celé bakalářské práce vyhodnotím, zda došlo ke splnění požadavku.

F1: Poslouchání veškeré komunikace

Jak z analýzy komponenty vzešlo, původní JRU se drželo oficiální implementace v rámci ETCS, kdy komunikovalo pouze s EVC. Pro použití v týmu to znamená kopírování dat (EVC pošle něco RBC i JRU zároveň), ale i nedostatek informací z ostatních částí simulátoru. Nové JRU by tak mělo mít možnost poslouchat nativně veškerou komunikaci, uchovávat data a rozumnou formou je poté prezentovat. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**

F2: Logování interních stavů

JRU má potenciál být v simulátoru využíváno jako logovací systém, který může vývojářům usnadnit práci. Aby byl tento systém efektivní, bude potřeba umožnit všem komponentám možnost zalogování jakékoli události či interního stavu. Bude tedy potřeba vymyslet systém, který bude jednotný a snadno používatelný. Autorem tohoto požadavku je **tým pracující na projektu v rámci předmětu SP1 a SP2**.

N1: Jednotná architektura

V rámci jednodušší orientace v kódu ve všech komponentách by JRU mělo také využívat novou architekturu. Mělo by reflektovat všechny změny, které se v architektuře udělají. V případě nějakých nesrovnalostí, může dojít i ke konzultaci a návrhům z mé strany. Autorem tohoto požadavku je **tým pracující na projektu v rámci předmětu SP1 a SP2**.

4.2.2 Požadavek na JRUViewer

JRUViewer má být GUI (Graphical User Interface) aplikace pro zobrazování zaznamenaných logů a dalších důležitých událostí v rámci ETCS simulátoru. Má to být hlavně pomůcka pro vývojáře, která umožní jednodušeji procházet logy a další data z celého ETCS simulátoru. V této sekci jsou veškeré požadavky pro tuto novou část simulátoru.

F3: Desktopová aplikace

Aplikace JRUViewer musí běžet na počítači a nesmí se jednat o webovou aplikaci. Důvodem je, aby používání nezáviselo na ničem a nebylo zapotřebí mít webový prohlížeč. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**

F4: Připojení k JRU pomocí vhodně zvolené komunikace

Veškerá data potřebná k zobrazení jsou uložena v komponentě JRU. Je tedy potřeba vytvořit připojení. Toto spojení by mělo být stabilní a mělo by umožnit zobrazovat výsledky za běhu programu. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**

F5: Vykreslování brzdných křivek

Brzdné křivky jsou základní funkcionalitou ETCS. Jejich vizualizací je totiž možné zjistit, že komponenta EVC funguje správně. JRUViewer by tedy měl nabídnout vývojářům možnost tyto křivky zobrazit. Tohle může pomoct při případném testování. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**

F6: Filtrování získaných informacích

Z požadavků JRU (viz. kapitola 4.2.1) je patrné, že se budou informace získávat z vícero zdrojů. Je tedy vhodné, aby JRUViewer mohl tyto zprávy a logy filtrovat. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**

F7: Zobrazení stavu komponent

V současnosti je v ETCS simulátoru vytvořen princip, kdy komponenty ohlašují svůj stav (aktivní, neaktivní). Jako všechny ostatní věci, i tato informace chodí přes MQTT komunikaci a JRU tak k tomu má přístup. Bylo by vhodné, kdyby JRUViewer rozumným způsobem stav komponent vykresloval. Autorem tohoto požadavku je **tým pracující na projektu v rámci předmětu SP1 a SP2**.

F8: Zobrazení odometrických statistik

Komponenta ODO posílá aktuální informace o vlaku do komponenty EVC. JRU má díky MQTT k těmto informacím přístup a může je dále zpracovávat. Lze tedy vypočítat informaci i o průměrné rychlosti. JRUViewer by tedy měl umět zobrazit aktuální rychlost vlaku, jeho celkovou ujetou vzdálenost, aktuální zrychlení a průměrnou rychlost. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**

N2: Musí běžet na OS Windows a Ubuntu

Vzhledem k tomu, že vývoj simulátoru probíhá na platformách Windows i Ubuntu, bylo by žádoucí, kdyby JRUViewer byl spustitelný na výše zmíněných operačních systémech. V současnosti je oficiální verze (používaná u zákazníka) simulátoru pouze pro OS Windows, má implementace na tuto platformu větší prioritu. Vhodně zvolené technologii by tento požadavek mohly vyřešit jednoduše. Autorem tohoto požadavku je **doc. Ing. Martin Leso, Ph.D.**.

4.3 Návrh komponenty JRU

Komponenta JRU je součástí oficiální dokumentace ETCS. Slouží jako černá skříňka vlaku, do které EVC zaznamenává veškeré údaje. Základním požadavkem tedy bylo uskutečnit EVC možnost s JRU komunikovat. Pro zachování konzistence v rámci projektu, jsem se rozhodl použít MQTT komunikaci.

Aby pro budoucí týmy bylo jednodušší se orientovat v kódu ve všech komponentách, bude i JRU využívat společnou architekturu. Ta je zpracována v jiné bakalářské práci [23], proto jí tady nebudu rozepisovat, pouze se zaměřím na odlišnosti, které jsou pro JRU specifické.

4.3.1 Logovací systém

Jak už jsem popisoval v analytické části (kapitola 4.2.1), pro vývojáře je potřeba mít nějaký logovací systém, který může urychlit vývoj simulátoru. Nabízí se tedy komponentu JRU rozšířit o možnost, aby do ní mohly zprávu posílat i ostatní komponenty.

4.3.1.1 MessageType

Typ	Význam
Internal	Tento typ logů může použít pouze komponenta JRU, slouží pro interní stavy.
Fatal	Vyskytla se chyba, která má velký vliv na simulátor – většinou znamená pád.
Error	Chyba, která negativně ovlivňuje simulátor. Může dojít k rozbití simulátoru.
Warning	Chyba menšího významu, simulátor funguje dál (může mít neobvyklé chování).
Debug	Slouží pro potřeby programátora při testování aplikace.
Info	Významnější událost pozitivního rázu, většinou přechod do stavu atd.
Note	Méně významné události, nebo pravidelně opakující se události.

Tabulka 4.1 Výčtový typ `MessageType`

V rámci CEM jsem musel definovat výčtový typ `MessageType`, který slouží k rozlišení logů na základě jejich významu pro chod simulátoru. Tabulka 4.1 obsahuje informace o všech definovaných typech. Ke každému typu je přiřazen i jeho význam, který určuje, jakým způsobem má být daný typ používán.

4.3.1.2 Definice logu

Po zvážení všech možností jsem dospěl k následující definici zprávy, která bude nést informace logu. Tuto zprávu jsem pojmenoval jako `InternalStateMessage`. Bude dědit od třídy `Message`, která je rodičem pro všechny zprávy přenášené přes MQTT komunikaci. Při návrhu je důležité zohlednit následující:

Konstruktor pro tuto zprávu by měl vývojáři umožnit inicializovat tři základní věci pro log:

1. Určit typ zprávy pro případné filtrování dle závažnosti.
2. Vložit textovou formu logu, která bude snadno čitelná pro uživatele.
3. Vložit libovolný počet proměnných pro výpis jejich hodnot.

Získání hodnot z dané zprávy by mělo být dostupné všude v kódu. Třída musí umožnit získat informace o typ zprávy, její textovou formu i hodnoty proměnných.

(De)serializace by se měla držet stejných pravidel, jako ostatní zprávy MQTT komunikace.

Při návrhu textového formátu jsem bral v potaz jeho snadné a přímočaré používání, které však nebude omezovat přidávání proměnných. Důležité bude odlišit, do kterého místa v logu má přijít hodnota z proměnné. Pro tento účel jsem se rozhodl použít symboly %, mezi které je potřeba dát klíčové slovo. Pro zapsání samotného symbolu % by bylo potřeba napsat „%%“.

Takový textový log by tedy mohl vypadat nějak takto: „Tohle je zpráva s %klicoveslovo% slovem na ukázku“. Pokud bychom k tomuto textu hodili proměnnou, jejíž obsahem bude „klíčovou“, po vypsání takto zpracovaného logu, je vhodné, aby se vypsalo: „Tohle je zpráva s klíčovým slovem na ukázku“.

4.3.1.3 Vytváření logů

Při návrhu, jak logovat informace z komponent, jsem se soustředil na několik věcí. Jednak jsem chtěl vývojářům usnadnit jejich práci, aby poslání logů bylo jednoduché a přímočaré, v druhé řadě jsem jim chtěl umožnit logy zobrazit bez použití JRU. Moje řešení muselo zapadat do dané architektury, proto jsem se rozhodl vytvořit `JRULoggerService`, která dědí z `IService` a také z `IInitializable`, čímž jí půjde inicializovat za běhu programu.

Tato služba bude primárně sloužit pro zaobalení jednoduchých funkcí, jako je vytvoření zprávy, její odeslání do komponenty JRU a případné vypsání do konzole dané komponenty. Musí reflektovat to, že daná zpráva umožňuje vložení neomezeného množství proměnných.

4.3.1.4 Ukládání logů

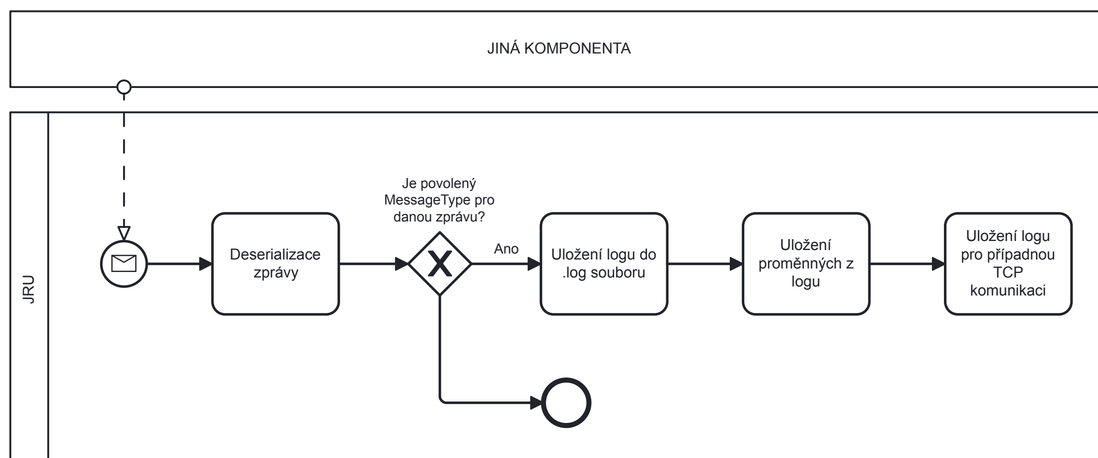
Přijatý log musí JRU, jakožto základ logovacího systému, správně uchovat. Nejjednodušším řešením je všechny příchozí zprávy ukládat v podobě *json* objektu do souboru. Toto řešení by spočívalo pouze v rozluštění toho, o jakou komponentu se jedná (na základě MQTT *Topic*) a následné uložení do správného souboru.

Řešení to není moc ideální, nemůže se tolik pracovat s daty a některé informace, jako například údaje k databázi, není vhodné ukládat. Proto je potřeba vytvořit tzv. `MessageHandler`, pro všechny definované zprávy. Tyto zprávy v něm řádně ošetřit a informace o nich uložit.

Bohužel je zpráv opravdu velké množství a je tedy pravděpodobné, že implementace nových zpráv může být rychlejší a stav v JRU nemusí být vždy aktuální. Proto jsem se rozhodl navrhnout i `MessageHandler`, který bude odchytávat zprávy, které zatím nemají žádný definovaný. Ten bude tyto zprávy ukládat v *json* objektu, jak jsem zmiňoval výše. Postupem času bude potřeba implementovat všechny zprávy.

Pro ukládání logů, ale i zpráv z MQTT komunikace, je nutné vymyslet společnou třídu, která bude v sobě uchovávat potřebné informace, jako jsou čas přijetí, zdroj přijetí a obsah přijetí. Pro tento účel jsem se rozhodl navrhnout `JRUMessage` a k ní přidružené třídy, které jí budou polymorfně rozšiřovat. Při návrhu jsem myslel na to, že tento princip se bude moct dát použít i pro případné zobrazování pomocí GUI aplikace.

Na obrázku 4.1 je zobrazeno, jak JRU pracuje s příchozím logem. Zahrnuje deserializaci zprávy a následné filtrování, zda je daný typ povolený pro *Topic*. Pokud kontrola proběhne v pořádku, dochází k ukládání informací.



Obrázek 4.1 Průchod logu skrz JRU

4.3.2 Pomocné třídy

Aby logy a komponenta JRU mohly správně fungovat, je potřeba navrhnout také další třídy, které dodají potřebné funkčnosti. Návrhy těchto tříd bych rád probral v této kapitole.

4.3.2.1 TimeStamp

Jak píšu v kapitole 2.3.1.3, důležitou součástí logů je i časová informace. Mým úkolem tak bylo navrhnout třídu, která bude tento čas reprezentovat.

Při návrhu bylo potřeba myslet na její budoucí používání, jako je vypisování do souboru či serializace pro případnou komunikaci s GUI aplikací. Vhodné by taky bylo vytvořit kontrolu časových intervalů mezi jednotlivými záznamy, které se mohou případně hodit pro zjištění, že systém funguje efektivně.

4.3.2.2 TimerService

Tato služba je zodpovědná za vytváření časových schránek. Pomocí jejího provolání by měl vývojář dostat aktuální čas od inicializace komponenty.

4.3.2.3 SaveLogService

Tohle je třída pro ukládání dat, které JRU zachytí ať už v MQTT komunikaci nebo pomocí logů od ostatních komponent. Kromě samotných zpráv je tato služba zodpovědná i za ukládání hodnot proměnných na základě klíčových slov, které byly do logu vloženy. Služba slouží pouze pro ukládání dat do souborů, případné poskytování zpráv pro GUI aplikaci bude řešit třída jiná.

4.3.2.4 MessageTypeCheckService

Jak jsem zmínil v kapitole 4.3.1.1, každý log může mít různý typ, který určuje vážnost daného logu. Na straně JRU, by bylo vhodné tyto logy filtrovat, aby nedocházelo k zatížení systému. Kvůli tomu jsem se rozhodl navrhnout tuto třídu, která je zodpovědná za filtrování přijatých logů od všech komponent na základě konfigurace.

4.3.2.5 JRUMessageDataService

V kapitole 4.3.1.4 jsem zmínil třídu `JRUMessage`, která slouží pro uchování informací z logů a veškeré MQTT komunikaci. Tato třída je zodpovědná za ukládání všech takových zpráv, které v rámci JRU vznikly. Využití bude především pro případné používání s GUI aplikací, která bude získávat data právě z této třídy. Kromě samotných zpráv se zde budou ukládat další statistiky, jako například data která chodí z ODO.

4.3.2.6 HeartbeatControlService

Heartbeat[3] chodí od každé komponenty jednou za sekundu. Díky této funkcionalitě tak chodí velké množství zpráv, které pro vývojáře při případném procházení logů nemají vypovídající hodnotu. Proto jsem se rozhodl tyto informace odklonit od `JRUMessageDataService`.

Tato služba, která se specializuje na jednotlivé stavy všech komponent, ukládá informace o posledně zaznamenaném heartbeatu a v případě potřeby vyhodnocuje, jestli je daná komponenta stále aktivní.

4.4 Návrh JRUViewer

Vzhledem k tomu, že je JRUViewer primárně pro interní použití a já nemám žádnou zkušenost s vytvářením GUI aplikací, rozhodl jsem se zvolit pro tento program Qt grafickou knihovnu [25]. Ta je velmi jednoduchá, běží na všech operačních systémech a nabízí příjemné uživatelské rozhraní, kterým se dá program jednoduše vytvořit. Zároveň tato knihovna obsahuje některé základní grafické prvky, které bude jednoduché v aplikaci použít.

4.4.1 Rozložení obrazovky

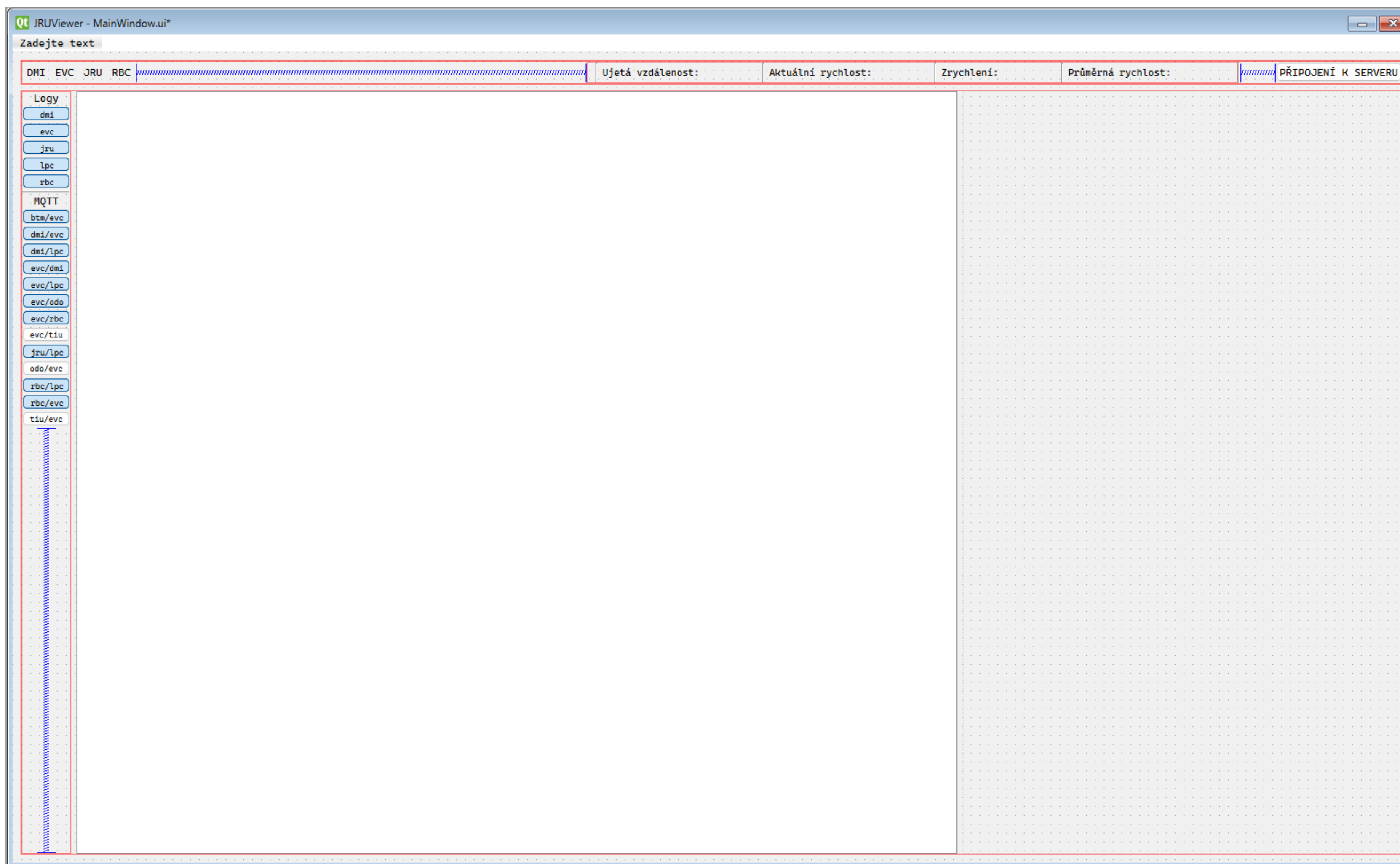
Moje schopnosti grafického návrháře nejsou na vysoké úrovni, přesto je potřeba obrazovku vhodně navrhnout. Knihovna Qt, kterou jsem se rozhodl pro tento program použít, nabízí svůj vlastní *Designer*, což je aplikace, kde je možné navrhnout svoje vlastní uživatelské rozhraní.

Při návrhu jsem vycházel především z požadavků a z vlastních zkušeností s různými formami uživatelského rozhraní. Nejdůležitější část JRUViewer, jsem se rozhodl dát doprostřed a věnovat tomu největší plochu. Do této části jednak patří výpisy samotných logů, ale také zobrazování brzdných křivek.

Filtrování těchto logů je také velmi důležitou funkcí. Filtry by měly být jednoduše přístupné, proto jsem se rozhodl věnovat této funkcionalitě levou lištu aplikace. Další důležitou částí je vykreslování různých statistik a stavu komponent. Pro tyto účely se mi zdá vhodné využít horní lištu, kterou lze plnit těmito funkcemi. Navíc jsem se rozhodl umístit do horní lišty částečnou interaktivitu – tlačítko pro připojení k serveru.

Na obrázku 4.2 je vidět podoba JRUViewer v aplikaci *Qt Designer*. Je možné, že v implementaci dojde k určitému odklonění od návrhu v závislosti na technické možnosti aplikace.

[3] pravidelné podávání informací o dané komponentě



Obrázek 4.2 Návrh JRUViewer v aplikaci Qt Designer

4.4.2 Architektura

Použitá architektura v rámci JRUViewer volně vychází z architektury představené společností Google, která je brána jako doporučená pro vývoj mobilních aplikací. Tuto architekturu jsem zvolil primárně kvůli zkušenostem, které ve vývoji mobilních aplikací mám. Důkladný popis architektury je dostupný na webových stránkách pro Android Developers. [26]

Architektura má základní rozdělení na dvě povinné a jednu nepovinnou vrstvu. Tyto vrstvy by od sebe měly být co nejvíce oddělené a komunikace mezi nimi by měla zásadně probíhat jen se sousední vrstvou. V následujících sekcí představím jednotlivé vrstvy společně s jejich zástupci z aplikace JRUViewer.

4.4.2.1 UI vrstva

Tato vrstva je jedinou vrstvou, ve které je dovolené využívat Qt knihovnu, která je součástí JRUViewer především kvůli grafickým komponentám. Pokud se však v budoucnu rozhodne kdokoliv využít Qt i pro síťovou komunikaci či další věci, které tato knihovna nabízí, je možné ji využít i v dalších vrstvách. Prvky z UI (User interface) vrstvy lze rozdělit na dvě části, podle jejich funkcionality.

UI prvky slouží pro samostatné vykreslování na displej uživatele. Tyto prvky by měly mít minimum informací o datech, pouze jim jsou předány informace, které mají vykreslit. V těchto prvcích se nesmí dít žádná logika.

State holders slouží k udržování informací o daných UI prvcích. Může se jednat o držení stavů, zachycování událostí, atd.

V JRUViewer se počítá s těmito třídami, které budou součástí UI vrstvy:

`Window` Jedná se o hlavní UI prvky celého JRUViewer. Jde o jedno celé okno, které v sobě obsahuje další Qt prvky (`Widgets`), jako jsou tlačítka, taby, různé `View` pro zobrazení informací. Qt aplikace počítá s existencí `MainWindow`, které je zatím jediným navrženým oknem.

`Dialog` je grafický prvek pro vyobrazení dialogových oken. Slouží pro předání rychlé informace uživateli, která má velký význam, nebo je vyžadována nějaká akce.

`QModel` slouží pro rozšíření implementace stávajících `QObjects`, aby lépe odpovídaly požadavkům JRUViewer. Některé UI prvky potřebují ke své funkčnosti třídy definované přímo v Qt knihovně. Takovým objektem je třeba `QTableView`, který potřebuje `QTableViewModel`, který v sobě uchovává data pro zobrazení v dané tabulce.

`Manager` je zodpovědný za udržení informací o stavu jednotlivých UI prvcích, které jsou ve `Window`. K tomu mu slouží prvky z kategorie `QModel`, ale i případný přístup do doménové vrstvy.

4.4.2.2 Doménová vrstva

Doménová vrstva je nepovinná, většinou se do aplikací přidává pro případ eliminace kopírování kódu. Spadají sem třídy, které se používají na vícero místech. Díky této definici sem patří všechny třídy, které jsou definovány v rámci repositáře CEM.

Já jsem se rozhodl do této vrstvy dodefinovat vlastní třídu typu `Worker`, kterou využívá třída `Manager`. Tato třída běží na vlastním vlákně a slouží k pravidelnému získávání informací z datové vrstvy. Jedná se o třídu, ve které se už nesmí vyskytnout cokoliv z Qt knihovny. V současném návrhu se zatím počítá s tím, že každý `Worker` bude v kódu moct použít pouze jeden `Manager`.

4.4.2.3 Datová vrstva

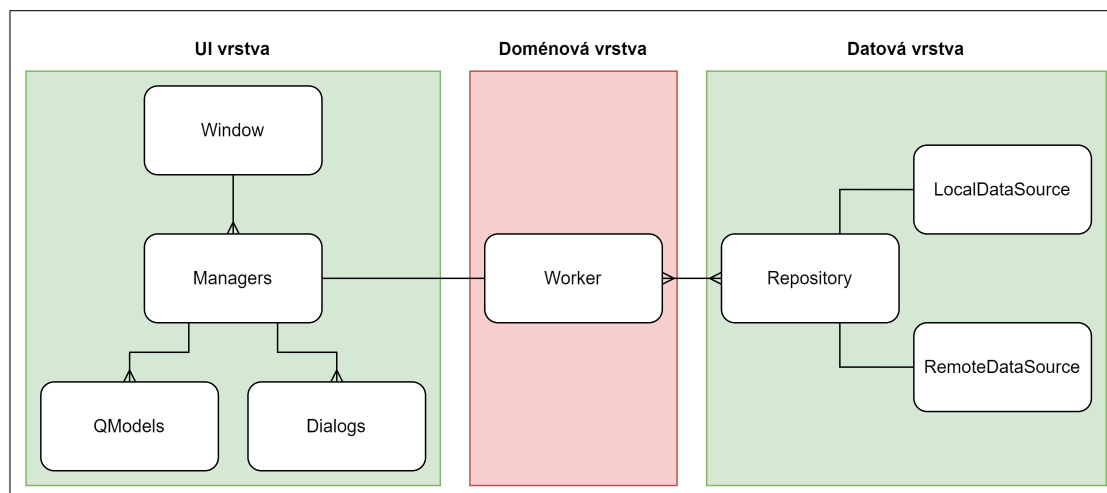
Datová vrstva je nejspodnější vrstvou z celé architektury. Jejím úkolem je nejen uchovávání dat, ale také získávání dat ze všech možných zdrojů. Pro JRUViewer využiji všechny definované prostředky v oficiální dokumentaci, na rozdíl od ostatních vrstev zde není žádná závislost na platformě a tak se můžou použít tyto části:

`Repository` slouží jako rozhraní mezi úložištěm dat a zbytkem aplikace. Pomocí této třídy je možné data nejen získat, ale v případě potřeby i ukládat. Vhodné je zvolit správnou míru rozdělování, aby v celé aplikaci neexistovala jen jedna tato třída. Opačný extrém také není vhodný, neboť příliš velká granularita by mohla vést ke zbytečné tvorbě nových tříd.

`DataSource` slouží v JRUViewer jako zdroj dat. Mohou být dvojího typu: *Remote* a *Local*. `RemoteDataSource` značí, že úložiště není součástí aplikace, ale je potřeba mít jiný zdroj, ke kterému je potřeba se připojit. Nejčastější v tomhle pohledu je například HTTP nebo jednoduchá TCP komunikace. `LocalDataSource` naopak používá data, která jsou součástí aplikace, nejčastěji lokální databáze či soubory.

4.4.2.4 Spojení vrstev

Na obrázku 4.3 je možnost vidět vizualizaci navrhované architektury pro JRUViewer. Nachází se v ní všechny vrstvy a jejich zástupci, které jsem zmínil v předchozích kapitolách. Na obrázku je taky patrné, jaký je vztah mezi jednotlivými částmi.



Obrázek 4.3 JRUViewer architektura

Část `Window` by měla obsahovat několik zástupců `Manager`. Ty v sobě mohou obsahovat buď nějaký `QModel`, který pomáhá obsluhovat věci v rámci Qt knihovny a nebo může obsahovat definici různých zástupců `Dialog`. Povinnou součástí `Manager` by však měl být jeden `Worker`, který se nachází v doménové vrstvě.

`Worker` má přístup k datům pomocí zástupce `Repository`. Jedno `Repository` není vázané pouze na jeden `Worker`, ale může být použito vícero zástupců. Zároveň může jeden `Worker` použít vícero `Repository`.

`Repository` jako takový se skládá ze dvou různých zdrojů dat `DataSource`. Ty jsou odlišné v závislosti na to, jestli jsou data získávané z lokálního úložiště, či je JRUViewer musí získat vzdáleným přístupem.

4.5 Návrh komunikace mezi JRU a JRUViewer

JRUViewer je plánovaný pouze jako zobrazovač logů a jiných dat. Bude tedy potřeba vyřešit problém, jak data, která jsou uložená v JRU, do této aplikace dostat. Vzhledem k tomu, že celý simulátor běží na MQTT komunikaci, nabízelo se použití stejného principu i zde. Na straně JRU by se přidal další *Topic*, přes který by komunikace probíhala. Toto řešení by bylo jistě velmi jednoduché a ušetřilo by mi mnoho času.

Použití MQTT se mi však zdá nevhodné. JRUViewer by měl posílat vždy dotaz směrem k JRU, které by mu mělo odpovědět pomocí daných dat. Za mě je zbytečné, aby do této komunikace zasahoval MQTT *broker*. Proto jsem se rozhodl, že ke komunikaci použiji vlastní jednoduchou implementaci TCP komunikace, která bude fungovat na operačních systémech Windows a Ubuntu.

4.5.1 TCPServer

Z popisu komunikace vyplývá, že v JRU by měl vzniknout server, který bude zprostředkovávat komunikaci pomocí TCP protokolu. Server musí splňovat několik požadavků:

1. Musí běžet na všech OS, který simulátor vyžaduje.
2. V případě, že nedojde k připojení žádného klienta, nesmí zbytečně zpomalovat JRU.
3. Musí zvládnout zpracovat různé požadavky od klienta.
4. Pro efektivní vyřizování požadavků od JRUViewer je nutné, aby `TCPServer` byl schopný obsloužit vícero připojení najednou.
5. Musí se vhodně používat v navržené architektuře pro všechny komponenty.

`TCPServer` nesmí mít v sobě uložená žádná data. Potřebné informace pro připojené klienty tak musí získávat z ostatních služeb. V tomto směru je nejdůležitější třída `JRUMessageDataService` (4.3.2.5), která poskytuje informace o všech zprávách (včetně těch interních), které JRU zachytí.

4.5.2 TCPClient

Pokud se na straně JRU počítá se vznikem serveru, je tedy jasné, že na straně JRUViewer musí vzniknout klient. Tento klient musí běžet na TCP komunikaci a musí být schopný posílat data na server. Požadavky na klienta tedy jsou:

1. Stejně jako celé JRUViewer, musí běžet na všech OS, který simulátor vyžaduje.
2. Musí reflektovat, zda se připojení k serveru povedlo nebo ne.
3. V rámci JRUViewer musí běžet vícero instancí této třídy.
4. Nesmí zpracovávat data, konkrétní čtení dat musí delegovat do jiných částí kódu.

4.5.3 Formát zpráv

Aby komunikace byla dobře čitelná a dalo se s ní v kódu dobře pracovat, je vhodné navrhnout společný formát zpráv, které mezi serverem a klientem budou chodit. Osobně jsem dlouho uvažoval nad použitím *json* formátu, který si všechny komponenty posílají pomocí MQTT komunikace. Vzhledem k tomu, že je potřeba data posílat po jednotlivých bytech, rozhodl jsem se *json* nakonec nepoužít a posílám zprávy v podobě textového řetězce, který v sobě obsahuje speciální znaky pro signalizaci jednotlivých částí zprávy.

Každou zprávu jsem se rozhodl zakončit řetězcem **\a\b**. Díky tomu, že znám konec každé zprávy, budu mít mnohem jednodušší načítání dat, ať už na straně klienta tak i serveru.

Zprávy však mohou mít v sobě uloženy vícero proměnných. Tyto proměnné od sebe odliším opět pomocí speciálního symbolu **\b**. Pokud proměnná je jakákoliv kolekce, která může obsahovat více než jeden prvek, jsou tyto prvky rozděleny pomocí **\a**.

V tabulce 4.2 je vidět zpráva, která v sobě obsahuje dvě proměnné: `ujeta_vzdalenost` a `balizy`. Obě dvě pracují s celočíselným datovým typem, proměnná `balizy` je kolekce. V posledním řádku tabulky je ukázka toho, jak by tato zpráva v mém navrženém formátu vypadala.

PROMĚNNÁ	HODNOTA
ujeta_vzdalenost: Int	123456
balizy: Array<Int>	230, 564, 789
123456\b230\a564\a789\b\a\b	

Tabulka 4.2 Ukázka použití formátu zpráv pro TCP komunikaci

4.6 Implementace JRU

V celé této části se budu věnovat mojí implementací komponenty JRU. Rád bych zde popsal implementační detaily jednotlivých tříd, poukázal na některé, za mě zajímavé, části a v neposlední řadě bych zde rád popsal moje procesy.

4.6.1 InternalStateMessage

Největší výzvou v celé komponentě JRU je určitě logovací systém. V této kapitole bych se zaměřil na třídu, která reprezentuje daný log. Nachází se ve společném repositáři CEM.

Instance této třídy, které budou vznikat v jednotlivých komponentách, je potřeba posílat přes MQTT komunikaci. Kvůli tomu musí tato třída dědit ze základní třídy `Message`. Z ní jednak podědí proměnnou `NID_MESSAGE` a zároveň musí vydefinovat virtuální metody pro serializaci zprávy: `to_json` a `from_json`.

4.6.1.1 Proměnné třídy

Každá třída má možnost si držet interní stav pomocí vlastních proměnných a jinak tomu není ani u `InternalStateMessage`. V tabulce 4.3 je možné vidět všechny privátní proměnné, včetně jejich stručného popisu.

NÁZEV	TYP	POPIS
`arguments`	`nlohmann::json`	uložené hodnoty argumentů z konstruktoru
`controlCntOfArguments`	`unsigned int`	kontrolní počet argumentů
`keyWords`	`vector<string>`	kolekce uložených klíčových slov
`messageText`	`string`	textový řetězec pro danou zprávu
`messageType`	`MessageType`	uložený typ logu

Tabulka 4.3 Stručný popis proměnných ve třídě `InternalStateMessage`

Než se pustím do popisu dalších vlastností třídy, rád bych se zaměřil na konkrétní proměnnou `messageText`. Její chování se liší v závislosti na tom, zda byla třída vytvořena před odesláním

nebo až po jejím přijetí na straně JRU. Před samotným odesláním obsahuje proměnná nezformátovaný řetězec. Jakmile je zpráva přijata, provede se formátování, kdy klíčová slova jsou nahrazena hodnotami argumentů, a výsledný řetězec je uložen do proměnné `messageText`.

4.6.1.2 Konstrukce zprávy

Velmi důležitou částí této třídy je její konstrukce. Dlouho jsem přemýšlel nad tím, jak nejlépe vytvářet logovací zprávy. Zvažoval jsem použití podobného přístupu jako standardní funkce `printf()` v jazyce C++, která pracuje s formátovanými řetězci.

Nakonec jsem tento přístup zavrhl. Chtěl jsem poskytnout programátorům jednoduchý nástroj, který jim ušetří úvahu nad typem proměnných. Toto jsem dosáhl použitím obecného (generického) přístupu v jazyce C++.

```
InternalStateMessage() = default;

InternalStateMessage(MessageID id, MessageType type, const std::string& text) {
    NID_MESSAGE = id;
    messageType = type;
    messageText = text;
}

template<typename ... Args>
InternalStateMessage(MessageID id, MessageType type,
                     const std::string& format, const Args&... args) {
    NID_MESSAGE = id;
    messageType = type;
    messageText = format;
    controlCntOfArguments = 0;
    AddArgument(args ...);
}
```

Listing 4.1 Všechny varianty konstruktorů pro `InternalStateMessage`

Na ukázce kódu 4.1 jsou zobrazeny tři různé konstruktory, kterými lze v současnosti inicializovat tuto třídu.

První varianta, nazývaná *default constructor*, slouží primárně k vytvoření daného objektu, když JRU přijímá daný log. Tento přístup funguje pro všechny definované MQTT zprávy, které používá celý simulátor, a nebyl důvod tento přístup změnit.

Druhý konstruktor slouží pro vytvoření zpráv, které neobsahují žádné argumenty. Tato varianta je velmi jednoduchá, protože není nutné provádět žádné specifické operace.

Třetí možnost je o něco složitější, protože zde dochází k volání metody `AddArgument`.

```
template<typename T, typename ... Args>
void AddArgument(const T& value, const Args&... args) {
    arguments[std::to_string(controlCntOfArguments++)] = value;
    AddArgument(args ...);
}

void AddArgument() {}
```

Listing 4.2 Metody `AddArgument` pro parsování argumentů v `InternalStateMessage`

Jak je patrné z kódové ukázky 4.2, tato metoda je rekurzivní a volá sama sebe vnitřně. Tímto způsobem efektivně prochází všechny dodané argumenty. Každý argument je uložen a současně se zvyšuje číslo kontrolního počtu argumentů. Prázdná varianta slouží k ukončení rekurze.

4.6.1.3 Serializace

Vzhledem k tomu, že třída `InternalStateMessage` dědí z `Message`, má k dispozici metodu `to_json`, kterou je vhodné upravit si k libosti. Serializace této zprávy je přímočará, dochází pouze k vytvoření `nlohmann::json` proměnné, do které se uloží hodnoty `NID_MESSAGE`, `messageType`, `messageText` a `arguments`. Není důvod sem přikládat ukázku z kódu.

4.6.1.4 Deserializace

Deserializace pomocí metody `from_json` je už zajímavější. Zde totiž kromě načtení hodnot klasických proměnných z `nlohmann::json` dochází i k formátování celé zprávy. Jak metoda `from_json` vypadá, je vidět na ukázce níže 4.3.

```
void from_json(const nlohmann::json& j) {
    keyWords.clear();
    NID_MESSAGE = j.at("NID_MESSAGE").get<MessageID>();
    messageType = j.at("messageType").get<MessageType>();
    messageText = j.at("messageText");
    arguments = j.at("arguments");
    controlCntOfArguments = 0;
    messageText = GetMessageText();
}
```

Listing 4.3 Metoda pro deserializaci v `InternalStateMessage`

Právě metoda `GetMessageText` je pro celý log klíčová. Ta má za cíl kontrolu textu, jestli souhlasí počet klíčových slov a argumentů a také provádí vložení argumentů na místo klíčových slov.

```
std::string GetMessageText() {
    std::string retStr;
    size_t pos = 0;
    while (pos < messageText.size()) {
        auto positions = GetPositionsInFormat(retStr, pos);
        if (positions.second == std::string::npos && positions.first != std::string::npos) {
            // vyhození výjimky
        } else if (positions.first == positions.second) {
            break;
        }
        pos = UpdateTextWithAnArgument(retStr, positions.first, positions.second);
    }
    if (keyWords.size() != arguments.size()) { // vyhození výjimky }
    return retStr;
}
```

Listing 4.4 Metoda pro formátování textu v `InternalStateMessage`

Na kódu 4.4 je vidět implementace této metody. Ta prochází skrz celý textový řetězec a pomocí metody `GetPossitionsInFormat()` hledá dvojici %, která slouží pro ohraničení klíčového slova. Zároveň zde dochází ke kontrole, že je formát zprávy v pořádku.

Pokud je vše vyhodnoceno správně, volá se metoda `UpdateTextWithAnArgument()`, která provede náhradu klíčového slova (včetně %) za patřičný argument, který se nachází jako další v pořadí. Návratovou hodnotou je v tomhle případě index pozice za druhým %. Implementační detaily těchto dvou metod mi nepřipadají pro kontext této práce důležité, proto je zde neuvádím.

4.6.2 JRULoggerService

Tuto službu implementuji především kvůli tomu, aby vytváření zpráv a jejich odeslání bylo zabaleno do jedné třídy. Tato třída návrhově spadá do společné architektury, takže některé věci jsou stejné, jako u ostatních. Kvůli mému použití generiky u `InternalStateMessage`, musím použít generiku i v této třídě. Tato vlastnost je bohužel na škodu, protože se nedá úplně jednoduše tato třída použít pro testování. Jsem si toho vědom, v současnosti se mi však nepovedlo vymyslet jiné optimální řešení.

Během implementace jsem myslel na to, aby vývojáři měli možnost vypisovat logy do konzole. V době psaní JRU ještě nebyla hotová aplikace JRUViewer, která by umožnila tyto logy zobrazit, takže výpis do konzole měl velkou prioritu. Kvůli výpisu i kvůli generickému vytváření `InternalStateMessage` existují čtyři různé metody `Log()`.

```
void Log(MessageType msgType, const std::string& text) const {
    Log(false, msgType, text);
}

template <typename... Args>
void Log(MessageType msgType,
         const std::string& text,
         const Args&... args) const {
    Log(false, msgType, text, args...);
}

void Log(bool toConsole, MessageType msgType, const std::string& text) const {
    auto msg = std::make_shared<InternalStateMessage>(id, msgType, text);
    if (toConsole) {
        // výpis zprávy do konzole
    }
    publisher->Publish(msg);
}

template <typename... Args>
void Log(bool toConsole, MessageType msgType,
         const std::string& text, const Args&... args) const {
    auto msg = std::make_shared<InternalStateMessage>(id, msgType, text, args...);
    if (toConsole) {
        // výpis zprávy do konzole
    }
    publisher->Publish(msg);
}
```

Listing 4.5 Metody `Log()` pro logování zpráv v `JRULoggerService`

Na ukázce kódu 4.5 jsou vidět všechny varianty pro `Log()`. Rozdíl mezi prvními dvěma a těmi dalšími, je v tom, že první argument metody je `bool toConsole`. V případě, že tento argument není přítomen, chová se metoda stejně, jako by byla předaná hodnota `false`.

4.6.2.1 Implementace v JRU

To, jak jsem popsal současnou implementaci tak úplně neplatí pro samotnou komponentu JRU. Pro tu nemá smysl, aby se zpráva posílala přes MQTT komunikaci pomocí `Publish()`. Místo toho zde dochází k internímu uložení zprávy a dalších hodnot.

4.6.2.2 DEBUG mód

Pokud má proměnná `bool toConsole` hodnotu `true`, dochází k výpisu do konzole. Tento proces je však velmi složitý. Za prvé dochází k vytvoření nové instance zprávy, která musí být deserializována, a za druhé samotný výpis může být pro program velmi náročný.

Aby nedocházelo ke zpomalení v produkčním kódu, rozhodl jsem se použít *#ifdef* direktivu jazyka C++. Před kompilací je tedy nutné specifikovat, že daný kód má být ve vývojářské formě, která umožní výpis požadovaných logů do konzole.

Ve všech komponentách je v jejich *CMakeLists.txt* obsažena následující část kódu 4.6. Tato část kódu zajistí, že pokud je *cmake* zavolán s přepínačem `-DENABLE_DEBUG=ON`, vytvoří *cmake* pro celý kód definici `DEBUG`. Díky této vlastnosti lze pak veškerý kód, který se nachází uvnitř podmínky `if (toConsole)`, obalit mezi *#ifdef DEBUG* a *#endif*.

```
if (ENABLE_DEBUG)
    add_definitions(-DDEBUG)
endif ()
```

Listing 4.6 Vytvoření definici `DEBUG` pro logování do konzole v *CMakeLists.txt*

4.6.3 TimeStamp

Časové záznamy jsou pro logovací systémy velmi důležité. Musel jsem implementovat třídu, která bude časovou schránku reprezentovat. Třída je součástí společného repozitáře CEM.

Základním konstruktorem, který by se měl používat při většině situací, je ten, který používá prvek z knihovny `chrono`, která je součástí standardní C++ knihovny. V něm získávám údaje pro minuty, sekundy a milisekundy, pomocí `std::chrono::duration_cast()`. Existuje taky varianta konstruktoru, kdy lze vložit konkretní čísla pro všechny zmíněné časové veličiny.

Pomocné metody, které jsem implementoval jsou operátory pro porovnání (`operator==()` a `operator!=()`), metoda `CheckInterval()`, která slouží pro ověření, že rozdíl mezi dvěma časovými záznamy je v daném milisekundovém intervalu. Vytvořil jsem také metodu `GetSeconds()`, která vrací počet sekund daného časového záznamu.

Vzhledem k tomu, že třídu `TimeStamp` bude potřeba posílat z JRU do JRUViewer, musí být součástí implementace i systém serializace a deserializace. K tomu pomáhá `operator<<()`, který vrací instanci třídy jako `std::ostream&`, a také metoda `FromString()`, která vyplní data z textového řetězce.

4.6.4 TimerService

Tato služba slouží k vytváření jednotlivých časových záznamů. Jedna se o třídu, která dědí od `IInitializable`, což vynucuje implementaci metody `Initialize`. V této metodě dojde k vytvoření aktuálního času pomocí `std::chrono::high_resolution_clock::now()`.

Pokud poté kdekoliv v kódu potřebuji získat aktuální časový záznam, zavolám metodu `GetTimeStamp()`, která vrací instanci třídy `TimeStamp`. Tu vytvářím tak, že od aktuálního času (získaný pomocí `std::chrono::high_resolution_clock::now()`) odečtu ten čas, který byl v metodě `Initialize` vnitřně uložen.

4.6.5 SaveLogService

Implementace služby, která má za cíl ukládat veškeré logy, byla velmi přímočará. Chci umožnit ukládání nejen samotné zprávy, ale i argumenty z logů. Třídu jsem udělal konfigurovatelnou pomocí systému konfigurací, který je součástí všech komponent s jednotnou architekturou. V konfiguraci by měly být dvojice: `Topic`, který JRU odebírá a název souboru, do kterého se mají logy ukládat.

```
void SaveMessage(std::shared_ptr<JRUMessage> message) const {
    auto topic = ConvertMessageIDToTopic(message->GetMessageId());
    std::ofstream outFile(files.at(topic), std::ios::app);
    if (outFile.is_open()) {
        outFile << message->DumpToFile() << std::endl;
        outFile.close();
    } else if (topic != Topic::JRU) {
        // zalogovani chyby
    } else {
        // vyhozeni vyjimky, aby nebyla cyklicka zavislost s logovanim
    }
}
```

Listing 4.7 Metoda pro ukládání zpráv do souboru v `SaveLogService`

Nejčastěji z této třídy provolávám funkci `SaveMessage()`. V ukázce 4.7 je vidět průchod během ukládání zpráv. V první řadě potřebuji získat `topic`, který poté můžu použít k získání názvu souboru, do kterého je potřeba daný log uložit. Tyto data se získávají v metodě `Initialize`, kde používám již zmíněnou konfiguraci. V případě, že je vše v pořádku, dojde k vypsání dat do souboru.

Naopak metoda `SaveValues()`, která slouží k ukládání argumentů jednotlivých logů, není v současnosti dostatečně využívaná. Implementaci jsem provedl především s ohledem na budoucnost, kdy očekávám její větší používání. Metoda se chová velmi podobně jako `SaveMessage()`, přičemž hodnoty proměnných jsou ukládány do souboru, který nese název *klicove_slovo.log*.

4.6.6 MessageTypeCheckService

Musel jsem implementovat systém, který umožní filtrovat zprávy rovnou na úrovni JRU. Možnost filtrace bude sice součástí JRUViewer, ale může se hodit i tady: jednodušší testování, odlehčení zátěže na komponentu. Jako u předchozí služby, mi k tomu velmi pomůže konfigurační systém.

V konfiguraci se nachází několik dvojic: `Component`, výčtový typ signalizující komponentu, od které daný log přišel a `MessageType` (viz. tabulka 4.1). Hodnota `MessageType` je pro určení nejméně vážné zprávy, kterou od dané komponenty lze přijmout. Zároveň dochází ke kontrole, že zpráva s typem `MessageType::Internal` přišla pouze od komponenty JRU.

Metoda, která se volá pro kontrolu, se jmenuje `ControlMessageType()`. Její implementaci je možné vidět na ukázce kódu 4.8.

```
bool ControlMessageType(Component component, MessageType msgType) const {
    return componentsControl.count(component) &&
           CheckInternal(component, msgType) &&
           componentsControl.at(component) >= msgType;
}

bool CheckInternal(Component component, MessageType msgType) const {
    return component == Component::JRU || msgType != MessageType::Internal;
}
```

Listing 4.8 Metoda pro filtrování zpráv v `MessageTypeCheckService`

4.6.7 JRUMessageDataService

Když jsem vytvářel komunikaci mezi JRU a JRUViewer, došlo mi, že ukládání dat nebude tak jednoduché, jak jsem si v návrhu představoval. Potýkal jsem se s několika problémy, musel jsem řešit, jak dlouho a jakým způsobem mám data v této třídě uchovávat.

Tato služba spadá do kategorie tzv. datových služeb, které jsou v ostatních komponentách velmi časté (v EVC se jedná o většinu služeb). Jejich základní, a ve výsledku i jedinou vlastností, je umožnit vývojáři uložit data a pak je v jiné části kódu opět získat. Oproti jiným službám zde však dochází v určitých situacích k velkému zpracování dat (viz kód 4.9).

```
void AddMessage(std::shared_ptr<JRUMessage> message) {
    if (message->GetSource() == ConvertTopicToString(Topic::ODOtoEVC)) {
        ODOStatistics statistics{};
        auto text = message->GetMessageText();

        auto dTrainStr = // získání ujeté vzdálenosti z proměnné text
        statistics.travelledDistanceInMeters = std::stoul(dTrainStr);

        auto vTrainStr = // získání aktuální rychlosti z proměnné text
        statistics.currentSpeed = std::strtod(vTrainStr.c_str(), nullptr);

        auto aTrainStr = // získání zrychlení z proměnné text
        statistics.currentAcceleration = std::stoul(aTrainStr);

        if (stamp == nullptr) {
            // vytvoření časové známky pro výpočet průměrné rychlosti
        } else {
            // výpočet průměrné rychlosti
        }
        odoStatistics = statistics;
    } else {
        messages.PushBack(message);
    }
}
```

Listing 4.9 Metoda pro přidání zprávy v `JRUMessageDataService`

Získávání dat je velmi přímočaré. Na ukázce kódu 4.10 je vidět, že existuje víc metod, které slouží k získávání různých dat. Metoda `PopMessages()` slouží k získání logů a zpráv chodící po

MQTT komunikaci. Je vidět, že data nejsou perzistentní a při každém provolání této metody se smažou. Metoda `GetOdoStatistics` se volá pro zisk odometrických statistik. Statistiky se posílají přes TCP komunikaci, a tak používám konkrétní formát.

```
std::vector<std::shared_ptr<JRUMessage>> PopMessages() {
    return messages.GetDataAndClear();
}

std::string GetOdoStatistics() {
    auto statistic = odoStatistics.GetValue();
    return std::to_string(statistic.travelledDistanceInMeters) + "\a" +
           std::to_string(statistic.currentSpeed) + "\a" +
           std::to_string(statistic.currentAcceleration) + "\a" +
           std::to_string(statistic.averageSpeed) + "\b";
}

bool UpdateEvcToTiuMessage(const TIUMessage& message) {
    auto retVal = !(tiuMessage == message);
    if (retVal) { tiuMessage = message; }
    return retVal;
}
```

Listing 4.10 Metody pro získání dat z `JRUMessageDataService`

Při testování jsem zjistil, že EVC posílá do komponentu TIU zprávu několikrát za sekundu. To vedlo ke zpomalení celého JRUViewer, který vypisoval všechny zprávy. Abych zefektivnil celý program, vytvořil jsem funkci `UpdateEvcToTiuMessage`, která slouží k filtrování, že se dvě stejné zprávy za sebou neposílají.

4.6.8 HeartbeatControlService

Implementace služby, která má za cíl kontrolovat aktuální stav jednotlivých komponent, byla velmi přímočará. Využil jsem vlastnosti třídy `TimeStamp`, která umožňuje kontrolovat časové intervaly mezi jednotlivými záznamy.

```
std::string GetComponentStateInString() {
    auto stamp = timer->GetTimeStamp();
    std::string retVal;
    std::lock_guard<std::mutex> lock(mutex);
    for (const auto& cmpState : componentState) {
        retVal += ConvertComponentToString(cmpState.first);
        if (stamp.CheckInterval(cmpState.second, HB_CHECK_IN_MS)) {
            retVal += "1\b";
        } else {
            retVal += "0\b";
        }
    }
    return retVal;
}
```

Listing 4.11 Metoda pro kontrolu stavu komponent `GetComponentStateInString`

Na ukázce kódu 4.11 je vidět, jak se tyto informace získávají. Je zde vidět jednak zamykání pomocí `std::lock_guard` a také provolávání `CheckInterval` pro zjištění, že je komponenta stále aktivní. Komponenty se hlásí jednou za sekundu, hodnotu `HB_CHECK_IN_MS` jsem nastavil na dvě sekundy.

4.6.9 BrakingCurvesDataService

Datová služba, která se zabývá brzdnými křivkami nemá bohužel žádný návrh. Požadavek na implementaci vykreslování přišel v pozdější fázi bakalářské práce a na pořádný návrh nebyl čas. Na štěstí se jedná primárně jen o ukládání dat a pak následné předávání dat ve formátu vyhovující TCP komunikaci.

4.6.10 TCPServer

Implementace tříd `TCPServer` a `TCPServerService` byla pro mě nejsložitější částí vlastního logovacího systému. Před psaním této části jsem neměl žádné zkušenosti s TCP připojením a musel jsem využít všechny dostupné zdroje.

Kvůli odlišné implementaci v závislosti na operačním systému se v kódu těchto tříd vyskytuje velké množství direktiv `#ifdef WINDOWS`, které rozlišují kód mezi platformami. Tato odlišnost může pro nezkušeného programátora činit kód méně čitelným, ale jiná možnost zde není.

Při implementaci jsem se zaměřil na podporu vícevláknovosti a možnost obsloužit více klientů současně. Třída `TCPServer` má jasně definované zprávy, které může přijmout, a odpovídá na ně patřičným způsobem získáním dat. Data získává z následujících služeb: `JRUMessageDataService`, `HeartbeatControlService` a `BrakingCurvesDataService`.

`TCPServerService` slouží k obalení celého serveru tak, aby fungoval ve společné architektuře. Základní funkcionalitou této služby je startování vlákna, na kterém poté celý server běží. Toto vlákno je uložené v třídě, aby k němu byl vždy přístup a šlo jej případně ukončit.

4.7 Implementace JRUViewer

V této sekci se budu věnovat hlavně implementaci desktopové aplikace JRUViewer. Kvůli tomu, že zkušenosti s aplikacemi, které obsahují grafické rozhraní, mám minimální, narazil jsem na spoustu problémů, které jsem musel řešit.

4.7.1 MainWindow

Třída `MainWindow` představuje základní uzel programu a je volána z funkce `main()`. Dědí z třídy `QMainWindow`, která vnitřně spouští své vlastní vlákno, jež udržuje program aktivní. Toto vlákno však přineslo řadu výzev, neboť jednotlivé prvky uživatelského rozhraní lze upravovat pouze z něj. Pro řešení tohoto problému jsem využil `SIGNALS` a `slots`, které umožňují propojit signály s funkcemi, a tak umožňují vyvolání určitých akcí.

Důležitou součástí této třídy je proměnná `Ui::MainWidnow* ui`, která obsahuje informace o zobrazených prvcích uživatelského rozhraní. Samotný popis rozložení obrazovky je uložen v souboru *MainWindow.ui*, který lze editovat pomocí Qt aplikace. Kromě této proměnné třída `MainWindow` také obsahuje `std::unique_ptr` pro všechny instance jednotlivých `Manager`.

Veškeré důležité inicializace jsou prováděny v konstruktoru této třídy. Zde dochází k inicializaci všech podstatných tříd a také se zde upravují některé prvky uživatelského rozhraní. Propojení jednotlivých signálů s metodami, které se mají vykonat, probíhá rovněž v konstruktoru.

Na ukázce kódu 4.12 je vidět deklarace této třídy. Je možné si zde všimnout i proměnné `brakingCurvesWidget`, což je specifický UI element pro vykreslování brzdných křivek (viz. kapitola 4.7.5).

```
class MainWindow : public QMainWindow {
    Q_OBJECT
public:
    explicit MainWindow(QWidget* parent = nullptr);
    ~MainWindow();

private slots:
    void OnBtnConnectToRemoteClicked();
    void OnBtnScrollDownClicked();
    void OnConfigurationClicked();
    void OnMessagesUpdated();
    void ChangeHeartbeatColor(Component component, bool isActive);
    void ChangeBrakingCurves(const BrakingCurvesData& data);
    void ChangeValueForOdoStatistics(const OdoStatistics& statistics);

private:
    Ui::MainWindow* ui;

    std::unique_ptr<BrakingCurvesPlotWidget> brakingCurvesWidget;

    std::unique_ptr<BrakingCurvesManager> brakingCurvesManager;
    std::unique_ptr<MessageManager> messageManager;
    std::unique_ptr<DialogManager> dialogManager;
    std::unique_ptr<HeartbeatManager> heartbeatManager;
    std::unique_ptr<RepositoryManager> repositoryManager;
    std::unique_ptr<OdoStatisticsManager> odoManager;
};
```

Listing 4.12 Deklarace třídy `MainWindow` v JRUViewer

4.7.2 Manager

Mají na starost získávat data k jednotlivým UI částem celé aplikace. V jejich implementaci je potřeba říct, že jsou závislý na Qt knihovně, protože dědí z třídy `QObject`. Tato dědičnost mi umožní vytvořit signály, které se poté provolají až do `MainWindow`, kde je možné daný UI elementy upravit nebo aktualizovat.

Každý ze zástupců této skupiny slouží k něčemu jinému, ale některé rysy mají společné. To je třeba možné vidět na ukázce 4.13. V konstruktoru je vždy předán `RepositoryManager`, který slouží ke správě všech úložišť v aplikaci. Obvykle jsou pak součástí `SomethingManager` dvě veřejné metody, pomocí kterých lze z `MainWindow` startovat a nebo vypínat vlákna, které běží uvnitř `SomethingWorker`.

Pro aktualizaci UI prvků obrazovky jsou důležité metody definované v sekci `signals`. Tyto metody nemají v `SomethingManager` definované tělo, je tedy potřeba je napojit na nějakou jinou metodu z třídy `MainWindow`. V kódu je pak možné provolat tento signál pomocí klíčového slova `emit`.

V sekci `private slots` se nachází metody, které lze na signál napojit. V `SomethingManager` se vyskytuje instance třídy `QTimer`, která v určených časových intervalech provolává signal `&QTimer::timeout()`. Právě na tento signál je potřeba napojit metodu ze sekce `private slots`.

Každý `SomethingManager` má svojí vlastní instanci `SomethingWorker`. Tato třída obstarává data, které poté může `SomethingManager` získat v metodě, kterou v pravidelných intervalech provolává nastavený časovač.

```
class SomethingManager : public QObject {
    Q_OBJECT
public:
    SomethingManager(const std::unique_ptr<RepositoryManager>& repositoryManager,
                     QObject* parent = nullptr);
    void StartSomething();
    void StopSomething();

signals:
    // signály, na které je potřeba v MainWindow udělat reakci

private slots:
    // funkce, které se volájí v závislosti na časomíře

private:
    std::unique_ptr<SomethingWorker> worker;
    std::unique_ptr<QTimer> timer;
};
```

Listing 4.13 Možná podoba zástupce skupiny *Managers*

4.7.3 QModel

Qt knihovna nabízí širokou škálu tzv. widgetů, které slouží jako UI elementy. Některé z těchto widgetů vyžadují další třídu, která má za úkol spravovat daný UI prvek. V mém případě jsem pro vypisování zpráv použil třídu `QTableView`, a proto jsem musel vytvořit třídu, která dědí vlastnosti od `QAbstractTableModel`. Tuto třídu jsem pojmenoval `MessageTableQModel`.

Hlavním úkolem `MessageTableQModel` je uchovávat data, která jsou následně vypisována do tabulky. Pro správnou funkčnost jsem musel implementovat čtyři poděděné metody: `rowCount()`, `columnCount()`, `data()` a `headerData()`.

V rámci požadavků pro JRUViewer jsem zdůraznil potřebu filtrování zpráv. Tuto funkcionalitu však nemůžu implementovat pouze pomocí `MessageTableQModel`, proto jsem se rozhodl vytvořit další třídu založenou na `QSortFilterProxyModel` nazvanou `MessageTableFilterQModel`.

Hlavní metoda, kterou jsem musel implementovat, se nazývá `filterAcceptsRow()` a slouží pro filtrování dat. Uvnitř této třídy existuje mapa, která na základě zdroje zprávy (komponenta nebo MQTT *Topic*) určuje, zda se daná zpráva má zobrazit. Tlačítka v levé části obrazovky jsou poté pouze napojena na tuto třídu, aby při stisknutí aktualizovala data v mapě.

Vše funguje tak, že `MessageTableFilterQModel` obsahuje instanci `MessageTableQModel`, ze které získává všechna data, jež následně filtruje. Samotný `QTableView`, který je umístěn v `MainWindow`, je poté spravován `MessageTableFilterQModel`.

4.7.4 Dialog

Je běžnou praxí, že různé aplikace obsahují dialogová okna, která upozorňují uživatele na potřebu větší pozornosti k určité části aplikace. I když v současnosti není mnoho případů užití v JRUViewer, je vhodné mít tento systém k dispozici. Celkový dialogový systém je postaven na třídě `DialogManager`, která obsahuje veřejné metody sloužící k vykreslování dialogového okna.

Samotné dialogové okno je reprezentováno třídou `SomethingDialog`, která dědí od `QDialog`. V konstruktoru této třídy je pomocí různých Qt funkcí a tříd vytvořen samotný dialog, který se zobrazuje.

4.7.5 Widget

Požadavek na vykreslování brzdných křivek přišel v pozdější fázi vývoje, a proto bylo nutné jej rychle vyřešit. Qt knihovna nabízí možnost vykreslovat různé grafy. Ačkoli jsem zvažoval jejich použití pro tyto účely, rozhodl jsem se vytvořit vlastní *widget* pro větší kontrolu nad vykreslováním.

Qt umožňuje vytvoření vlastní třídy, která dědí od třídy `QWidget`. Tato třída obsahuje metodu `paintEvent()`, která je volána při každé aktualizaci. Implementací této metody jsem mohl vytvořit vlastní vykreslovací mechanismus, což mi umožnilo vykreslovat i brzdné křivky (třída `BrakingCurvesPlotWidget`).

4.7.6 Worker

Třídy, které spadají do této kategorie, mají jeden hlavní úkol: vykonávají nějaký úkon ve vlastním vlákně, obvykle zahrnující získávání dat z `Repository`. Abych zajistil, že všechny tyto třídy budou mít podobné chování, vytvořil jsem rozhraní `IWorker`, které musí být implementováno všemi těmito třídami. Toto rozhraní definuje metody pro spuštění a zastavení vlákna.

4.7.7 Repository

Implementace datové vrstvy byla pro mě přímočará, protože jsem měl již zkušenosti s podobnými architekturami z vývoje mobilních aplikací. Cílem bylo organizovat související prvky do jednotlivých tříd, což zjednodušuje správu a zvyšuje modularitu aplikace.

Když se objeví potřeba vytvořit nové úložiště (*repository*), vytvoří se nová třída, jako je například `SomethingRepository`. Tato třída musí poskytovat veřejné metody umožňující jednotlivým prvkům z kategorie `Worker` snadný přístup k datům ve správných formátech.

`SomethingRepository` má schopnost získávat data z různých zdrojů v závislosti na jejich dostupnosti. V současné době se využívá především třída `SomethingTCPDataSource` pro komunikaci mezi JRU a JRUViewer přes TCP. Implementace lokálních datových zdrojů zatím nebyla potřeba, ale je důležité tuto možnost zohlednit pro budoucí vývoj.

RepositoryManager

Mimo to, ještě bych rád zmínil třídu `RepositoryManager`, která mi usnadňuje předávání jednotlivých instancí tříd `Repository` do příslušných částí aplikace. Není vhodné, aby instance `SomethingRepository` byla pevně svázána s konkrétní instancí `SomethingWorker`, protože je možné, že data budou potřeba sdílet i jinde. `RepositoryManager` také obsahuje důležité třídy pro správu připojení, jako je například `TCPClient`.

4.7.8 TCPClient

Implementace `TCPClient` je velmi podobná `TCPServer` u komponenty JRU. Při získávání dat od serveru jsem se rozhodl, že klient nebude data zpracovávat, pouze předá jednotlivé textové řetězce datovým zdrojům. Pro optimalizaci a rychlejší běh celého JRUViewer jsem se rozhodl použít více instancí klienta v celé aplikaci. Každý `SomethingTCPDataSource` bude mít svůj vlastní `TCPClient`.

4.8 Testování

V mé práci jsem se zaměřil i na testování nově implementovaných částí. Projekt intenzivně využívá knihovnu *googletest* pro usnadnění testování C++ kódu, zejména prostřednictvím tzv. *Unit* testů, které slouží k ověření menších částí kódu. Snažil jsem se uplatnit tento přístup i v komponentě JRU. Avšak narazil jsem na problém s `JRULoggerService`.

Tato služba se odlišuje od ostatních v tom, že nemá potřebné rozhraní (`IJRULoggerService`), čímž není možné vytvořit falešnou implementaci (takzvaný *mock*) nutnou pro testování. Tento nedostatek mi znemožňuje vytvářet efektivní *Unit* testy. Je zřejmé, že bude potřeba provést důkladnou analýzu a navrhnout lepší řešení, jak testovat aktuální implementaci. Pravděpodobně bude nezbytné najít kompromis.

Naštěstí se mi podařilo úspěšně otestovat třídy v rámci společného repozitáře CEM, konkrétně `InternalStateMessage` a `TimeStamp`. Pro tyto třídy jsem otestoval všechny veřejné metody, přičemž testy se spouštějí automaticky pomocí webové aplikace CI/CD v GitLabu při každé změně.

4.8.1 Interakce s ostatními komponentami

JRU jsem tak mohl testovat pouze manuálně a byl jsem nucen kontrolovat výsledky sám. Tento proces byl přímý a jednoduchý: spustil jsem celou simulaci a sledoval vytvářené *logfiles*, které JRU generovalo.

Během testování jsem se často setkával s problémy spojenými s nestabilitou MQTT připojení. Často docházelo k odpojení jedné z komponent (většinou EVC nebo JRU) od MQTT *brokera*. Toto chování bylo velmi znepokojující, zejména když implementace `MqttPublisherService` byla ve všech komponentách identická. Při testování funkcionality zaznamenávání brzdných křivek byla situace tak neúprosná, že jsem musel změnit implementaci v komponentě EVC úplně a použít jiné funkce pro připojování k MQTT.

I přes tuto změnu však občas docházelo k odpojení. Je tedy pravděpodobné, že někde nedostatečně využíváme knihovnu, která zprostředkovává připojení. Budoucí tým se proto bude muset zabývat tím, proč k těmto problémům dochází, a předložit nové, stabilnější řešení.

Kromě problému s MQTT, se na operačním systému Windows začal v pozdější fázi testování objevovat problém s knihovnou, která se používá pro serializaci zpráv: *nlohmann/json*. Na tomto operačním systému knihovna občas nefunguje při kopírování dat, což vede k pádu celé komponenty. Problém jsem se snažil vyřešit pomocí `try-catch` bloků, ale marně. Ve verzi pro Ubuntu se tato chyba nevyskytuje. Kód, který řeší tuto funkcionalitu je součástí společné architektury. Kvůli tomu byl problém komunikován s celým týmem, zatím se nepovedlo najít řešení.

Když však připojení k MQTT fungovalo a zároveň se neobjevila *json* chyba, běžel spolehlivě i logovací systém. Všechny zprávy z komunikace byly správně uloženy a vypsány do *logfiles*. Při nasazování nové verze ETCS simulátoru (10. května 2024) byla funkčnost JRU využita pro odhalení menších nedostatků. Svůj účel tak plní dobře a implementace logovacího systému je z mé strany úspěšná.

4.8.2 Zobrazování logů

Významná část implementace byla spojena s GUI pro zobrazování logů, nazvaným JRUViewer. Testování GUI pomocí *Unit* testů a jiných automatických testů je velmi obtížné. Stejně jako při testování interakce JRU s ostatními komponentami, muselo být i zde provedeno manuální otestování celé aplikace. Kromě mého vlastního testování jsem požádal i ostatní členy týmu o pomoc při testování a poskytnutí zpětné vazby.

Základní funkcionality, tedy zobrazování logů, fungovala od začátku správně. Všechny logy se zobrazovaly bez problémů, a bylo možné rozlišit různé barvy. Nedostatky týkající se aplikace

se především týkaly výkonu, protože JRUViewer byl často velmi přetížen veškerou komunikací.

Díky tomuto přetížení jsem byl schopen odhalit několik nesrovnalostí ve funkci simulátoru. Například jsem zjistil, že EVC příliš často posílá zprávy do TIU obsahující informace o brzdění vlaku. Tyto zprávy se posílají cyklicky a bohužel zatěžují tak celou komunikaci. Problém jsem prozatím vyřešil na straně JRU tím, že filtruji zprávy obsahující stejné informace jako ty předchozí. Toto řešení je dostačující, avšak do budoucna by mělo být přepracováno chování EVC.

Také jsem odhalil chybu v komunikaci mezi ODO a EVC. Data posílaná mezi těmito jednotkami neodpovídala komentářům z kódu a oficiální dokumentaci. Například zrychlení vlaku, posílané z komponenty ODO jako desetinné číslo, bylo v EVC interpretováno jako celé číslo, což vedlo k nepravdivým výsledkům (vždy 0). Tým pracující na EVC komponentě byl informován o všech chybách a plánuje je vyřešit v nadcházejícím období.

Testování mě také přivedlo k zjištění, že dochází ke zbytečnému zdvojování informací. Například když JRU zachytí zprávu z MQTT komunikace, zároveň se tato událost ohlásí i z komponent, které se týkají této komunikace. Je vhodné zvážit, co je skutečně potřeba zaznamenávat a co není.

Zároveň testování odhalilo možné nedostatky v návrhu grafického rozhraní, zejména ve způsobu zobrazení všech zpráv v jednom seznamu. Do budoucna by bylo vhodné rozdělit zprávy do více seznamů podle typu události, což by uživatelům umožnilo lépe sledovat a filtrovat důležité informace. Takové úpravy by mohly vylepšit uživatelskou zkušenost a zefektivnit práci s aplikací.

Ostatní funkcionality, jako zobrazování stavu komponent a brzdných křivek fungují velmi dobře. Připojení JRUViewer k JRU je bezproblémový. Až se vyřeší výkonnostní problémy, které souvisí i se zbytečným posíláním zpráv, bude aplikace v používatelném stavu. Týmy ji pak budou moct používat pro testování simulátoru a případné odhalování chyb.

4.9 Návrh na budoucí rozšíření

Doba určená pro vypracování bakalářské práce je omezená, a proto jsem nemohl stihnout všechny potřebné úpravy, které by byly vhodné pro JRU i JRUViewer. JRUViewer momentálně vnímám spíše jako prototyp, který vyžaduje další doladění.

V předchozí kapitole, která se věnovala testování grafické aplikace JRUViewer, jsem již zmínil problémy s výkonem a prolínáním různých zpráv, což může snižovat přehlednost uživatelského rozhraní. Knihovna Qt nabízí třídu `QTabWidget`, která by mohla tento problém efektivně řešit. Tato třída umožňuje implementaci více pohledů pomocí různých záložek. Toto rozdělení by efektivně umožnilo oddělit od sebe logy, MQTT komunikaci a brzdné křivky, čímž by JRUViewer zobrazoval menší množství informací než dosud.

S použitím různých záložek by bylo také nutné implementovat lokální datové zdroje (viz. kapitola 4.4.2.3), kde by se ukládaly všechny přijaté informace, které nejsou v daném okamžiku vykreslovány aplikací. Bude třeba přijít s optimálním řešením, jak aktualizovat tato data, aby nedocházelo ke zpomalení celého procesu.

V současnosti aplikaci chybí možnost jakékoli konfigurace. Pro tuto funkcionalitu je připraven dialogový systém, ale bohužel jsem nestihl cokoliv implementovat. Z mého pohledu by bylo dobré konfigurovat v současnosti alespoň dvě věci: připojení k serveru (zvolení adresy a portu) a nastavení jednotlivých komponent a filtrování jejich logů.

Aplikace JRUViewer je primárně určena pro vývojáře pracující na ETCS části celého simulátoru. Zákazník krátce zvažoval využití aplikace i pro účely Lektorského PC, nicméně tento program není založen na jazyce C++, což může ztížit kompatibilitu. Proto by se aplikace měla především rozšiřovat podle potřeb vývojářů, kteří ji mohou využít k usnadnění práce při hledání chyb a testování celého simulátoru. Pokud bude motivace, doporučuji využití implementaci vlastní třídy, která dědí z `QWidget`, pro vícero oblastí a tím udělat grafické rozhraní přehlednější.

Kapitola 5
Závěr

V teoretické části této práce jsem analyzoval základní znalosti, na nichž stojí zbytek práce. Vysvětlil jsem základní pojmy týkající se ETCS a nastínil historii tohoto zabezpečovacího systému. Dále jsem se zaměřil na projektové řízení a klíčové pojmy v této oblasti. Zjistil jsem, že definice pojmu projekt a dalších souvisejících pojmů mohou být různorodé a záleží na interpretaci každého jednotlivce. V závěru této části jsem provedl analýzu logovacích systémů a logů, které jsem poté využil při návrhu vlastního logovacího systému pro ETCS simulátor.

Praktická část práce je rozdělena do dvou částí. V první se zabývám samotným projektem ETCS, včetně analýzy současného stavu a navržením řešení pro jeho vylepšení. Druhá část se zaměřuje na komponentu JRU, pro kterou jsem provedl analýzu všech komponent projektu, abych zajistil co nejpodobnější implementaci a usnadnil tak orientaci v kódu budoucím vývojářům. Vytvořil jsem návrh a popsal současnou implementaci nejen komponenty JRU, ale i grafické aplikace pro zobrazování logů – JRUViewer.

Mým cílem v této práci bylo navrhnout vylepšení procesů řízení projektu ETCS simulátoru. K tomu jsem využil své znalosti z oblasti projektového řízení a prostředí ETCS simulátoru, abych navrhl řešení, která by mohla projektu prospět. Z mého pohledu je klíčové najít osobu, která bude mít na starosti řízení projektu a usnadní tak práci všem ostatním.

Implementované procesy do projektu, zejména do webové aplikace GitLab, jsou využívány současnými členy týmu ETCS. Je pro ně snadné porozumět tomu, jak tyto procesy používat, a vidí v nich přínos pro projektové řízení. Avšak není jim vždy snadné správně tyto procesy korigovat a zajistit jejich dodržování, což podporuje mé přesvědčení, že je třeba rozšířit současnou strukturu projektu.

Dalším cílem byla implementace komponenty JRU včetně vlastního logovacího systému. Při návrhu jsem dbal především na jednoduchost použití, aby logování nezpůsobovalo vývojářům zbytečné potíže, ale naopak jim usnadňovalo práci. Vytvořil jsem také grafickou aplikaci, která zobrazuje logy a další události, a je zde stále prostor pro její další vylepšení a zdokonalení funkcionality a spolehlivosti.

Z mého hlediska se podařilo splnit všechny požadavky, které vznikly pro komponentu JRU. Aktuálně však existuje problém s implementací zobrazovače (JRUViewer) na operačním systému Ubuntu. Přestože Qt knihovna, která byla vybrána, je multiplatformní a mělo by vše běžet i na Ubuntu, nedokázal jsem projekt správně nakonfigurovat.

Příloha A
Používání logovacího systému

Tato příloha slouží pro budoucí vývojáře k seznámení se s logovacím systémem, jeho fungováním a správným používáním. Obsahuje také informace o tom, na co by si vývojáři měli dávat pozor, aby nepřetížili logovací systém a mohli efektivně využít jeho vlastností.

Ve své práci jsem doposavad pořádně nevysvětlil, jakým způsobem se logy mají používat. První pravidlo, které bych rád zmínil je to, že pro používání je vhodné používat `JRULoggerService` (viz. kapitola 4.6.2). Tato služba má připravené rozhraní, které usnadní vytváření samotné zprávy.

Tuto službu je možné využívat všude, kam se posílá `ServiceContainer`, ve kterém je uložena. Nabízí se tedy jí použít v ostatních službách, v jednotlivých třídách, které spadají do kategorie `MessageHandler` a nebo v `TopicWorker`. V těchto třídách by bylo vhodné si `JRULoggerService` držet pomocí ukazatelů. Použití by pak vypadalo jednoduše:

```
// Získaní ukazatele na JRULoggerService
loggerService = container.FetchService<JRULoggerService>().get();

// V jine části kódu zavolání Log() metody
loggerService->Log(true, MessageType::Info, "Tohle je ukázkové použití.")
```

Listing A.1 Jednoduché vytvoření logu

Na ukázce kódu A.1 je vidět jednoduché vytvoření logu. Neobsahuje žádné parametry a je provolána s hodnotou `true`, která v případě splněných podmínek (viz. kapitola 4.6.2.2) zařídí vypsání textu do konzole.

Vypsání jména funkce

Pro jednodušší přehled při procházení jednotlivých logů, je vhodné zalogovat i jméno funkce (metody), ze které daný log je. Preferovaným formátem takových logů je, aby vždy začínal jménem metody (v případě, že metoda může být ve vícero třídách, tak i jméno třídy) a tu pomocí „:“ oddělil od samotného textu. Vývojář má dvě možnosti, jak toho může docílit:

1. Vložení názvu funkce do textového řetězce je přímočaré. Programátor vloží název do textu, jako zbytek celé zprávy.
2. Název funkce se do textu vloží pomocí parametru. Logovací systém je navržen, aby přijímal různé argumenty, takže vložit název funkce nemusí být tak složité. Tomu může pomoct i makro

__FUNCTION__, které slouží pro přístup jména funkce.

```
void SomeService::LoggingShowcase() {
    logger->Log(true, MessageType::Info,
        "SomeService::LoggingShowcase: Tohle je první ukázka.");
    logger->Log(true, MessageType::Info,
        "%function%: Tohle je druhá ukázka.", __FUNCTION__);
}
```

Listing A.2 Porovnání možností pro výpis jména funkce v logu

Obě dvě možnosti je možné vidět na kódové ukázce A.2.

Logování typu Debug

V produkčním kódu, ve větvích *master* a *develop* by se neměly objevovat logy, které mají svůj typ jako `MessageType::Debug`. Účelem těchto logů je vývojářům usnadnit hledání chyb a kontrolovat průchod aplikací. Tyto logy by měly být dočasné a při vyřešení problému by měly být smazány.

Logování více argumentů

V mnou navrženém logovacím systému je možné vkládat do logovací zprávy různé argumenty. Tato funkce umožňuje vytvářet flexibilnější záznamy, které mohou obsahovat různé hodnoty v závislosti na průběhu programu. V budoucnosti může tato vlastnost sloužit k ukládání hodnot těchto proměnných v systému, což pak může být využito k vizualizaci hodnot dané proměnné za běhu programu.

```
void SomeService::LoggingShowcaseWithArguments() {
    speed = odoService->GetCurrentSpeed();
    logger->Log(true, MessageType::Info,
        "%function%: Aktuální rychlost: %speed%.", __FUNCTION__, speed);
}
```

Listing A.3 Logování aktuální rychlosti

Na ukázce kódu A.3 je demonstrováno logování více argumentů. Vedle zaznamenání názvu funkce, jak bylo popsáno v předchozích odstavcích, je zde také zaznamenána aktuální rychlost. Získání této hodnoty může být realizováno různými způsoby, ale pro účely logování není podstatné jakým způsobem je hodnota získána.

Pokud je aktuální rychlost například 120 km/h, může výpis vypadat následovně: ”SomeService::LoggingShowcaseWithArguments: Aktuální rychlost: 120.”JRU si zároveň ukládá informaci, že hodnota proměnné `speed` je 120 a v budoucnu může být tato informace využita k vizualizaci hodnoty proměnné.

Příloha B

Používání aplikace JRUViewer

Tato příloha slouží jako uživatelská příručka pro grafickou aplikaci JRUViewer. Obsahuje vysvětlení a popis uživatelského rozhraní a poskytuje informace o tom, co uživatel může očekávat a vidět při používání aplikace. Vzhledem k současnému stavu aplikace, která nemá velké množství funkcionalit, není tato příručka příliš rozsáhlá.

Jako přílohu k této práci jsem přiložil spustitelnou aplikaci JRUViewer. Po zapnutí aplikace se zobrazí (zatím) jediné okno celého programu. V dalších odstavcích popíšu jednotlivé části obrazovky.

Připojení k serveru

V pravé horní části se nachází tlačítko (obrázek B.1) pro připojení k serveru. Jednoduchým stisknutím dojde k tomu, že se JRUViewer pokusí připojit k JRU. Pokud vše dopadne správně, začnou se zobrazovat logy a další údaje. Pokud však nastane chyba, zobrazí se dialogové okno s chybovou hláškou.

Obrázek B.1 Tlačítko sloužící pro připojení k JRU serveru

Kontrola stavu komponent

Jakmile dojde k připojení k serveru, začnou se získávat informace o komponentách, které v ETCS simulátoru fungují. Komponenty čekají na zprávu od Lektorského pracoviště, které značí, že má simulace začít. V tu chvíli začnou posílat tzv. *Heartbeat*, který podává informaci o jejich stavu. Stav komponent je tak vidět v levém horním rohu (obrázek B.2).

Obrázek B.2 Ukázka stavu jednotlivých komponent

Filtrování zpráv

Levá lišta programu obsahuje tlačítka, která slouží pro filtrování zpráv. Jednoduchým stisknutím a odkliknutím lze zprávy ze seznamu přidat nebo odstranit.

Procházení zpráv

Tou nejdůležitější částí je procházení skrz zaznamenané logy. Zprávám a logům je věnována většina obrazovky, aby byly vždy dobře čitelný a dalo se v nich dobře orientovat. Zpráv chodí opravdu velké množství a seznam, který vypisuje logy se posouvá směrem dolů jak logy přibývají. Lze však v seznamu kdykoliv vyjet nahoru a podívat se na logy staršího typu.

Pro větší přehlednost jsou jednotlivé zprávy barevně odlišeny v závislosti na jejich typu. To dobře reprezentuje obrázek B.3, kde jsou vidět testovací zprávy a jejich rozdíly.

```
00:00:015  jru      Tohle je interní zpráva.
00:00:018  jru      Tohle je fatalní chyba.
00:00:018  jru      Tohle je chyba.
00:00:019  jru      Tohle je varování.
00:00:019  jru      Tohle je debugovací zpráva.
00:00:020  jru      Tohle je informační zpráva.
00:00:020  jru      Tohle je poznámka.
00:32:780  lpc/evc  {"NID_MESSAGE":800,"simulationStatus":1}
```

Obrázek B.3 Barevné rozlišení zpráv, které JRUViewer zobrazuje

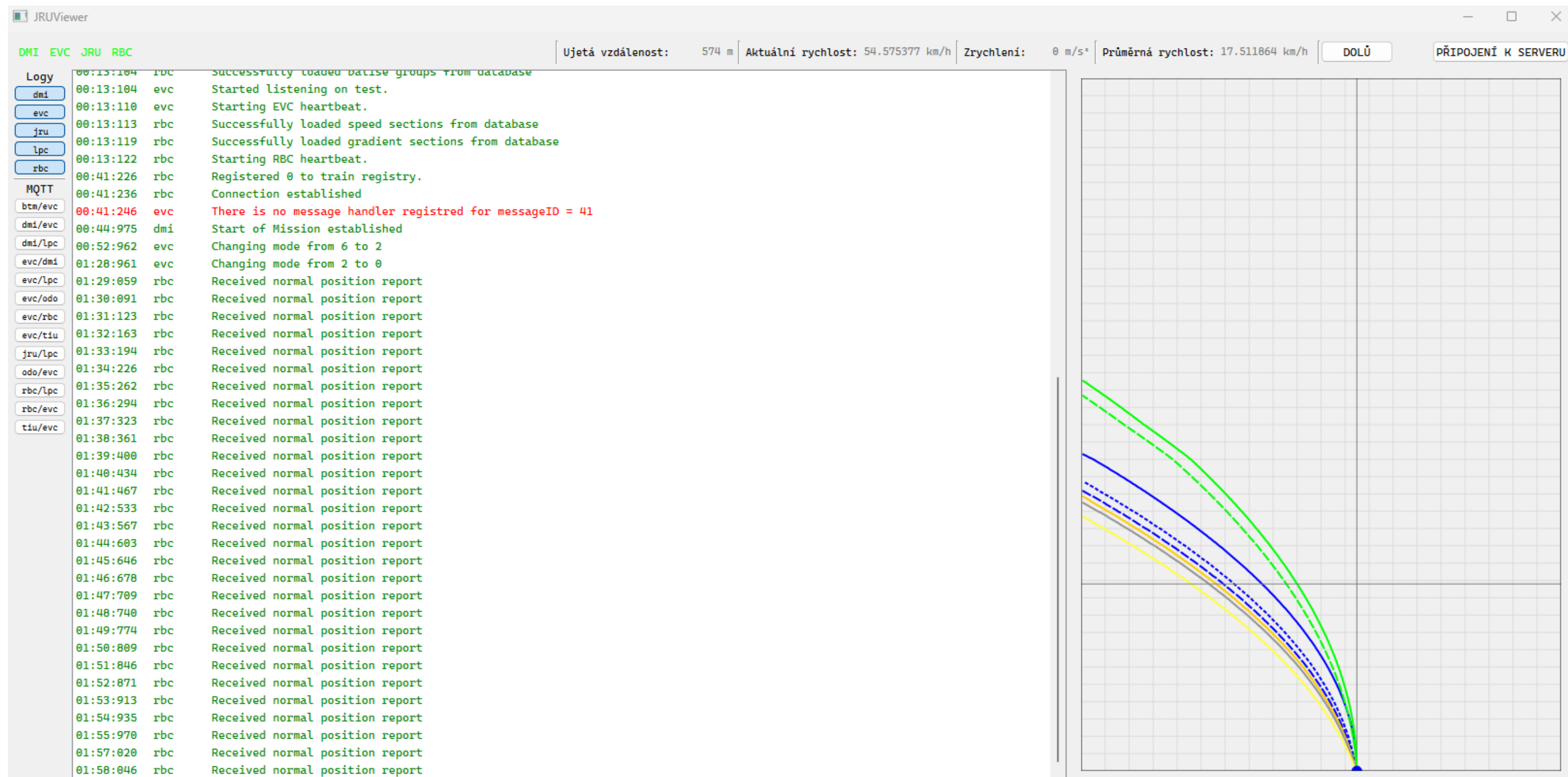
Vizualizace brzdných křivek

V pravé části vedle výpisu logů a zpráv se nachází prostor pro vykreslování brzdných křivek. Ten je při startu aplikace prázdný a vykresluje pouze osy. Jakmile ETCS systém přejde do FS módu[1], začnou fungovat brzdné křivky a je tedy možnost je začít vykreslovat. Jak vypadají brzdné křivky je možné vidět na obrázku B.4.

Odometrické statistiky

Po připojení k serveru dochází k výpočtu a k následnému výpisu statistiky, která souvisí s ODO. Na obrázku B.4 jsou tyto statistiky vidět v horní liště, vedle tlačítka „Dolů“, které slouží pro posunutí seznamu na úplný konec. Je možné si taky všimnout, že aktuální zrychlení je nulové, což odpovídá chybě, kterou jsem zmiňoval dříve.

[1] Mód, kdy má EVC plnou zodpovědnost nad vlakem



Obrázek B.4 Obrazovka JRUViewer během používání

Bibliografie

1. SPRÁVA ŽELEZNIC. *Základní charakteristika železniční sítě* [online]. Správa železnic, státní organizace, 2024 [cit. 2024-03-25]. Dostupné z: https://www.spravazeleznic.cz/o-nas/vse-o-sprave-zeleznic/zeleznice-cr/zeleznicni-sit-v-cr/.
2. UNIFE. *ERTMS in brief* [online]. UNIFE, 2024 [cit. 2024-03-01]. Dostupné z: https://www.ertms.net/about-ertms/ertms-in-brief/.
3. UNIFE. *ERTMS History* [online]. UNIFE, 2024 [cit. 2024-03-01]. Dostupné z: https://www.ertms.net/about-ertms/ertms-history/.
4. EUROPEAN UNION AGENCY FOR RAILWAYS. *SUBSET-026 System Requirements Specification Chapter 2 Basic System Description* [online]. European Union Agency for Railways, 2006. Ver. 2.3.0 [cit. 2024-05-15]. Dostupné z: https://www.era.europa.eu/era-folder/set-specifications-1-etcs-b2-gsm-r-b1.
5. UNIFE. *ERTMS Signaling levels* [online]. UNIFE, 2024 [cit. 2024-03-02]. Dostupné z: https://www.ertms.net/about-ertms/ertms-signaling-levels/.
6. UNIFE. *ERTMS Benefits* [online]. UNIFE, 2024 [cit. 2024-03-02]. Dostupné z: https://www.ertms.net/about-ertms/ertms-benefits/.
7. EVROPSKÁ KOMISE. *Subsystems and Constituents of the ERTMS - European Commission* [online]. Evropská komise, 2024 [cit. 2024-03-03]. Dostupné z: https://transport.ec.europa.eu/transport-modes/rail/ertms/what-ertms-and-how-does-it-work/subsystems-and-constituents-ertms_en.
8. PETER140001. *Balíza* [online]. Společnost přátel kolejové dopravy, 2015 [cit. 2024-03-04]. Dostupné z: https://www.k-report.net/ukazobrazek.php?soubor=1033495.jpg&httpref=.
9. PEŠEK, David. *BI-PRR: Úvodní přednáška* [online]. FIT CTU Courses Pages, 2023 [cit. 2024-03-04]. Dostupné z: https://courses.fit.cvut.cz/BI-PRR/lectures/01/BI-PRR.21-PR_01-Uvodni%20prednaska.pdf.
10. WYSOCKI, Robert K. *Effective Project Management - Traditional, Agile, Extreme, Hybrid.* 1.1 Defining a Project [online]. 8th. John Wiley & Sons, 2019 [cit. 2024-04-26]. ISBN 978-1-119-56280-1. Dostupné z: https://app.knovel.com/hotlink/khtml/id:kt012J03W2/effective-project-management/defining-a-project.
11. PILÁT, Marek. *Co je to projektové řízení?* [online]. SHEAN.cz, 2020 [cit. 2024-03-05]. Dostupné z: https://www.shean.cz/clanky/detail/co-je-to-projektove-rizeni.htm.

12. NÁPLAVA, Pavel. *BI-TIS - Přednáška č. 2* [online]. 2024. [cit. 2024-03-05]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/641572/mod_page/content/23/Prednaska02.pdf.
13. FIŠTRÓN. *SMART metoda: Jak správně definovat cíle* [online]. FISTRO, 2017 [cit. 2024-03-08]. Dostupné z: https://fistro.cz/aktuality/smart-metoda-jak-spravne-definovat-cile/.
14. ORAGUI, David. *Software Documentation Best Practices* [online]. Helpjuice, 2024 [cit. 2024-03-15]. Dostupné z: https://helpjuice.com/blog/software-documentation.
15. MLEJNEK, Jiří. *BI-SWI: 12. přednáška - Metodiky a agilní přístup* [online]. 2024. [cit. 2024-03-06]. Dostupné z: https://moodle-vyuka.cvut.cz/pluginfile.php/601011/mod_resource/content/8/12.prednaska.pdf.
16. DIMA, Alina Mihaela; MAASEN, Maria Alexandra. From Waterfall to Agile software: Development models in the IT sector, 2006 to 2018. Impacts on company management. *Journal of international studies* [online]. 2018 [cit. 2024-04-27]. Dostupné z DOI: 10.14254/2071-8330.2018/11-2/21.
17. AMAZON. *What is a Log File?* [online]. Amazon Web Services, Inc. or its affiliates., 2024 [cit. 2024-03-10]. Dostupné z: https://aws.amazon.com/what-is/log-files/.
18. BENSON, David. *Everything You Need To Know About Log Viewers* [online]. Logit.io, 2023 [cit. 2024-05-08]. Dostupné z: https://logit.io/blog/post/log-viewer/.
19. GENERALI ČESKÁ POJIŠŤOVNA. *Co je SWOT analýza a jak ji vypracovat* [online]. 2022. [cit. 2024-03-16]. Dostupné z: https://www.generaliceskaprofi.cz/ze-zivota/co-je-swot-analyza-a-jak-ji-vypracovat.
20. TÝM DSFD. *Dopravní sál fakulty dopravní* [online]. Tým DSFD, 2023 [cit. 2024-03-20]. Dostupné z: https://dsfd.fd.cvut.cz/.
21. GITLAB. *REST API* [online]. 2024. [cit. 2024-04-12]. Dostupné z: https://docs.gitlab.com/ee/api/rest/.
22. NEPRAŠOVÁ, Tereza. *ETCS – Aktualizace a nová architektura komponenty DMI.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.
23. VESELÝ, Ondřej. *ETCS – Aktualizace a nová architektura komponenty RBC.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.
24. HIVEMQ. *MQTT Essentials - All Core Concepts Explained* [online]. HiveMQ, 2024 [cit. 2024-03-15]. Dostupné z: https://www.hivemq.com/mqtt/.
25. THE QT COMPANY. *Qt | Tools for Each Stage of Software Development Lifecycle* [online]. The Qt Company, 2024 [cit. 2024-03-22]. Dostupné z: https://www.qt.io/.
26. GOOGLE. *Guide to app architecture* [online]. Google for Developers, 2023 [cit. 2024-04-02]. Dostupné z: https://developer.android.com/topic/architecture/.

Obsah příloh

- readme.txt .... stručný popis obsahu média
- exe
  - config.cfg .... konfigurační soubor pro MQTT komunikaci
  - RUN.exe .... soubor pro spuštění obou částí najednou
  - jru .... adresář se spustitelnou formou komponenty JRU
  - jruviewer .... adresář se spustitelnou GUI aplikací JRUViewer
- src
  - bp-martin-caslavsky .... zdrojová forma práce ve formátu LaTeX
  - jru .... zdrojové kódy komponenty JRU
  - jruviewer .... zdrojové kódy GUI aplikace JRUViewer
  - timetracking .... zdrojové kódy skriptu na měření času
- text .... text práce
  - bp-martin-caslavsky.pdf .... text práce ve formátu PDF
- zdroje-prezentace .... veřejně nedostupné zdroje