
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# SYSTÉM NA SPRÁVU PROGRAMOVACÍCH KONVENCÍ V PROJEKTU

**CODING CONVENTIONS MANAGEMENT SYSTEM**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** **Bc. MICHAL ORLÍČEK**
**AUTHOR**

**VEDOUCÍ PRÁCE** **Ing. JAN PLUSKAL**
**SUPERVISOR**



**BRNO 2021**

**Vysoké učení technické v Brně**
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

# Zadání diplomové práce


Student: **Orlíček Michal, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Softwarové inženýrství
Název: **Systém na správu programovacích konvencí v projektu Coding Conventions Management System**
Kategorie: Softwarové inženýrství
Zadání:

1. Zjistěte, jaké konvence psaní kódu používají programátoři při vývoji open source projektů a jakým způsobem zajišťují/vynucují jejich použití v projektu. Proveďte průzkum technologií, který tento proces zajišťují.
2. Analyzujte vybrané projekty z bodu 1 a zjistěte, jakým způsobem byly změny v konvencích komunikovány k programátorům a jak bylo ověřeno jejich dodržení. Dále zjistěte, jakým způsobem se nově přispívající programátoři seznamují s konvencemi.
3. Vyberte vhodné technologie, analyzované v bodě 1 a 2. Navrhněte systém, který zajistí správu konvencí v projektu. Systém bude schopný konvence vytvořit pro nově vznikající projekty, zjistit aktuálně používané a komunikovat jejich změny k programátorům. Systém musí být schopen ověřit, zdali jsou konvence dodrženy a vytvořit souhrnnou zprávu o stavu projektu. Respektujte připomínky vedoucího k navrženému systému.
4. Navržený systém dle bodu 3 implementujte.
5. Porovnejte implementovaný systém oproti existujícím alternativám, diskutujte jeho silné/slabé stránky a určete vhodné oblasti použití. Demonstrujte systém na vybraném open source projektu.

Literatura:

- Martin, R.C. (2008). *Clean Code: A Handbook of Agile Software Craftmanship.* Prentice Hall.
- Martin, R.C., Martin, M. (2006). *Agile Principles, Patterns, and Practices in C#*. Prentice Hall.
- Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Pluskal Jan, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 30. července 2021
Datum schválení: 26. října 2020

# Abstrakt

Cieľom práce je navrhnúť a implementovať systém na správu programovacích konvencií v projekte. Pred navrhnutím samotného systému bol vykonaný prieskum benefitov programovacích konvencií, analýza používaných konvencií v open source projektoch v službe GitHub a analýza existujúcich technológií spravujúcich programovacie konvencie. Na ich základe boli navrhnuté scenáre použitia, špecifikované požiadavky a určená architektúra. Následne bol systém implementovaný ako webová aplikácia postavená na technológiách Blazor a EditorConfig. Hlavným zámerom bolo vytvoriť systém, ktorý umožní uchovávať všetky druhy používaných programovacích konvencií a zároveň umožní užívateľovi dané konvencie automaticky kontrolovať a generovať. Zverejnený je pod open source licenciou v službe GitHub a nasadený v cloudovej platforme Azure.



# Abstract

The goal of this thesis is to design and implement coding conventions management system for project. Prior to the creation of the system itself, the research of coding conventions benefits, the analysis of used technologies in open source projects at GitHub service, and the analysis of existing technologies managing coding conventions was done. On the basis of that, usage scenarios were designed, requirements were specified and system architecture was determined. Then the system was implemented as web application based on Blazor and EditorConfig technologies. The main aim was to create a system that would allow to store all types of programming conventions and at the same time allows users to automatically control and generate them. It is published under an open source license within the GitHub service and deployed on the Azure cloud platform.



# Kľúčové slová

programovacie konvencie, čistý kód, open source, GitHub, EditorConfig, Blazor, ASP.NET Core

# Keywords

coding conventions, clean code, open source, GitHub, EditorConfig, Blazor, ASP.NET Core


# Citácia

ORLÍČEK, Michal. *Systém na správu programovacích konvencí v projektu.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pluskal

# Systém na správu programovacích konvencí v projektu

## Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Jana Pluskala. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Michal Orlíček
30. júla 2021

## Poďakovanie

Chcel by som sa poďakovať vedúcemu práce Ing. Janovi Pluskalovi za ochotu a odbornú pomoc. Priateľom a známym za pomoc s testovaním, prekladom a korektúrou textu. A v neposlednom rade mojej priateľke a rodine za psychickú podporu pri písaní tejto práce.

# Obsah

# Kapitola 1

# Úvod

V súčasnosti sa čoraz viac tímov rozhoduje vyvíjať projekty *agilným* spôsobom. Tento spôsob vývoja vyžaduje ešte väčšiu *čistotu kódu*, ako *lineárne* spôsoby, ktoré sa používali v minulosti. Vyššia úroveň čistoty umožňuje lepšiu udržateľnosť kódu, čo znižuje časovú zložitosť pridávania nových funkcionalít v ďalších iteráciách vývoja.

Jeden z najdôležitejších indikátorov kvalitného a *čistého kódu* je jeho konzistencia. Určením pokynov, ktoré sú odporúčané na použitie jednej alternatívy, z rôznych možností pri písaní určitej časti kódu, sa pri ich dodržiavaní konzistencia zvyšuje. Takto zapísané pokyny sa nazývajú programovacími konvenciami alebo programovacími štýlmi a môžu sa líšiť v závislosti od programovacieho jazyka, platformy, druhu vyvíjanej aplikácie, typu súboru alebo aj skúseností členov tímu. Pri správnom zvolení konvencií môže byť kód čitateľnejší, kvalitnejší, konzistentnejší a môže byť potlačovaná jeho zložitosť, čo môže viesť k menej častým alebo jednoduchším *refaktorizáciam*.

Cieľom práce je navrhnúť a vytvoriť systém, ktorý umožní správu programovacích konvencií v projekte. Je určený primárne pre projekty, ktoré využívajú *.NET* ekosystém a preto je postavený na moderných technológiách od spoločnosti *Microsoft*, avšak je možné ho použiť pre všetky typy projektov. Zdrojové súbory systému sú voľne dostupné v službe *GitHub* pod *MIT* licenciou, vďaka ktorej je možné ho upravovať a používať na komerčné aj súkromné účely. Výhodou zverejnenia pod voľnou licenciou je umožnenie opravovania chýb a následný vývoj systému po dokončení diplomovej práce.

Prvá časť práce (viď 2) sa zaoberá programovacími konvenciami používanými v *open source* projektoch. Rozširuje motiváciu z úvodu, obsahuje prehľad programovacích konvencií používaných v projektoch. Bližšie analyzuje vybrané projekty a zaoberá sa ich vytváraním, diskusiami o nich, komunikáciu k vývojárom po ich zavedení a aj následnými úpravami. Avšak cieli aj na spôsoby, akými sa dodržuje alebo vynucuje ich použitie a ako sa s nimi noví členovia projektov zoznamujú. Zameriava sa aj na prieskum technológií, ktoré zaisťujú správu programovacích konvencií a na technológie, ktoré pomáhajú konvencie v projekte dodržať. Ako prílohu, obsahuje práca zoznam *open source* projektov použitých pri analýze.

Druhá časť práce (viď 3) obsahuje návrh systému. Začína porovnaním existujúcich riešení a používaných technológií, pričom využíva poznatky z predchádzajúcej časti. Na ich základe špecifikuje scenáre použitia systému a formuluje požiadavky naň. Následne opisuje navrhnutý systém, jeho architektúru a jednotlivé architektonické celky. V neposlednej rade vysvetľuje, ako bol návrh implementovaný a aké technológie boli na implementáciu použité.

Tretia časť práce (viď 4) opisuje, akým spôsobom bol systém testovaný. Diskutujú sa jeho výsledky, zhodnocujú sa silné a slabé stránky, limity, vhodné oblasti využitia a mož-

nosti pre rozšírenia a zmeny. Následne je systém demonštrovaný na zvolenom *open source* projekte.

Poslednou časťou je samotný vytvorený systém s jeho dokumentáciou. Zdrojové súbory sa nenachádzajú v texte práce, ale sú dostupné online[1] a na priloženom kompaktnom disku. Dokumentácia[2] obsahuje manuál, ako do systému implementovať vlastné kontroly kódu, spolu s *API* dokumentáciou kódu. Okrem toho obsahuje návod ako aplikáciu nasadiť, avšak ten pre otestovanie systému nie je nutný, pretože v dobe jeho testovania a hodnotenia je už nainštalovaný systém dostupný online[3].

Práca obsahuje jednu prílohu (viď A), je ňou zoznam *open source* projektov v službe *GitHub*, ktoré sú použité na analýzu v časti 2.2.1.

Pre lepšiu čitateľnosť a menej opakujúcich sa vysvetlení používam skratky a slová v texte práce, ale ich vysvetlenie oddeľujem od jeho zvyšku. Na konci práce je dostupný zoznam skratiek a slovník s literatúrou, použitou na jeho tvorbu. V digitálnej verzii je možné na slová v práci kliknúť k presmerovaniu na ich význam. Práca obsahuje dva prehľady použitej literatúry, jednu k samotnej prácu a druhú k zmienenému slovníku a skratkám. V literatúre k slovníku sú využité prevažne terciáne informačné zdroje, ktorým som sa snažil v literatúre k práci vyhnúť.

[1] https://github.com/orlicekm/CodingConventionsManagementSystem
[2] https://orlicekm.github.io/CodingConventionsManagementSystem
[3] https://ccms.orlicek.net

# Kapitola 2

# Analýza

Kapitola sa zaoberá analýzami programovacích konvencií používaných v projektoch. Pred začatím tvorby návrhu systému je vhodné zanalyzovať programovacie konvencie z rôznych pohľadov. Prísluší sa skúmať, prečo a kedy je vhodné konvencie užívať, aké projekty a akými spôsobmi konvencie využívajú a aké technológie k tomu používajú. Vďaka analýzam je možné adekvátne špecifikovať požiadavky na vytváraný systém (pre viac informácií o návrhu systému viď kapitolu 3).

Prvá časť kapitoly sa venuje výhodám programovacích konvencií (viď sekcia 2.1). Rozširuje motiváciu z úvodu (viď kapitola 1) o teoretické podklady. Vysvetľuje, prečo sú programovacie konvencie súčasťou *clean code* a popisuje iné spôsoby udržovania čistoty kódu. Druhá časť kapitoly analyzuje rôzne druhy programovacích konvencií (viď sekcia 2.2). Popisuje, aké druhy sa používajú, pričom čerpá z rôznych zdrojov, medzi ktorými je aj päťdesiat analyzovaných *open source* projektov zo služby *GitHub*. Následne sa podrobnejšie zaoberá konkrétnymi projektami a pozoruje, akými spôsobmi sú konvencie v projektoch uchované, spravované, ako k nim programátori pristupujú, zoznamujú sa s nimi a ako sa overuje alebo vynucuje ich dodržiavanie. Posledná časť skúma existujúce technológie, ktoré sa používajú na správu programovacích konvencií (viď sekcia 2.3). Zisťuje, aké konvencie daná technológia umožňuje spravovať, aké možnosti dáva vývojárovi pri ich správe a ako daná technológia odporúča alebo vynucuje ich použitie v projekte.

## 2.1 Benefity programovacích konvencií

Prvým krokom, pri vytváraní práce je analýza teoretických podkladov programovacích konvencií, za účelom zistenia, aká je motivácia na ich používanie a aké výhody poskytujú. Prvá časť sa zameriava na rolu čistého kódu v agilnom vývoji (viď 2.1.1). Následne sa venuje vlastnostiam kvalitného kódu (viď 2.1.2) a problémom, ktoré môžu vzniknúť, ak sa kód pravidelne nečistí (viď 2.1.3). V ďalšej časti sa definuje čistý kód (viď 2.1.4) a programovacie konvencie s ich prepojením na čistotu (viď 2.1.5). Poslednú časť tvoria iné spôsoby, ako je možné zvyšovať čistotu kódu v projektoch (viď 2.1.6).

### 2.1.1 Čistota kódu pri agilnom vývoji

Ako je uvedené v úvodnej kapitole, v súčasnosti sa zvyšuje množstvo projektov, ktoré sú vyvíjané *agilným* spôsobom. Toto tvrdenie podporujú aj štatistiky z vyhľadávača *Google*, kde *agile scrum* postupne okolo roku 2010 prekonal *vodopádový model* [37]. Jeho početné využitie je možné vidieť aj v ankete z roku 2018 zo *StackOverflow*, kde na otázku, aké

metodológie vývoja používajú, odpovedalo 57 075 profesionálnych developerov [75]. Na prvom mieste skončil *agile* s 85.9% a na druhom *scrum*, ktorý patrí pod agilné metodológie, s 63.2% [75].

Metodiky *agilného* vývoja sa snažia o zvýšenie produktivity a efektivity práce a ich výhodou je možnosť rýchlo reagovať na zmenu požiadaviek zákazníka [1]. Vývoj je založený na krátkych iteráciách, medzi ktorými sa vytvorí časť požadovanej funkcionality a vytvorí funkčný a otestovaný software [73]. Kniha *Agile principles, patterns, and practices in C#* definuje agilný vývoj ako schopnosť vyvíjať software rýchlo v prostredí meniacich sa požiadaviek [51]. Ak chceme dosiahnuť túto schopnosť, musíme používať postupy, ktoré poskytnú potrebnú disciplínu a spätnú väzbu. Potrebujeme použiť princípy návrhu, ktoré udržia software flexibilným a udržateľným a potrebujeme poznať návrhové vzory, ktoré preukázateľne vyvažujú tieto princípy pre konkrétne problémy [51]. Z definície je pre nás najpodstatnejším pojmom disciplína, vďaka ktorej nepíšeme kód podľa toho, čo subjektívne považujeme za najlepšie, podriaďujeme sa nejakej vyššej vôli, pravidlám, ktoré sme si definovali. Prostredníctvom týchto pravidiel je snaha o dosiahnutie toho, aby bol software flexibilným a udržateľným, a teda snažia sa zvyšovať jeho kvalitu. O kvalite kódu je viac v časti 2.1.2.

V knihe *Clean Code* sa uvádza, ako sa *scrum* a *agile* v dnešnej dobe zameriavajú na rýchle uvedenie produktu na trh [50]. Časti kódu sa opúšťajú predčasne, nie však z dôvodu, že by boli dokončené, ale preto, že hodnotový systém sa zameriava viac na vonkajší vzhľad produktu, ako na podstatu toho, čo dodávame [50]. Preto sa v *scrume* odporúča, aby bola *refaktorizácia* (viď 2.1.6) súčasťou *DoD* [50]. Pri nedodržaní odporúčaní nám opravovanie chýb, rozširovanie a udržovanie časti kódu, napísaných týmto spôsobom, ktoré nie sú dokončené a teda vyčistené, zaberie omnoho viac času, ako keby sme čistenie vykonali pri ich vytváraní. Viac o problémoch špinavého kódu viď časť 2.1.3.

### 2.1.2 Kvalita kódu

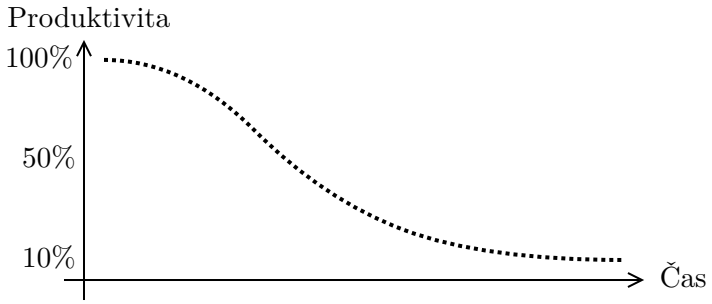
Na úvod by bolo vhodné zadefinovať, čo to vlastne kód je. Všeobecne známa definícia kódu, ktorú môžeme nájsť vo významových slovníkoch, ho opisuje ako systém slov, písmen, čísel alebo symbolov, ktoré reprezentujú správu alebo zaznamenávajú informáciu šifrovane alebo v krajšej forme. V tejto práci však budeme kódom označovať zdrojový kód projektu. *LINFO* definuje zdrojový kód ako verziu softvéru, ktorá je originálne napísaná človekom ako čitateľný, neformátovaný text [80]. Kódu sa v programovaní nikdy nezbavíme [50]. V *Clean Code* Robert C. Martin vysvetľuje, že dôvodom je reprezentácia detailov požiadaviek, ktoré na určitej úrovni už nie je možné ignorovať a je nutné ich špecifikovať [50]. Následne definuje kód ako jazyk, ktorými sú nakoniec tieto požiadavky vyjadrené [50].

Existuje viacero vlastností, ktorými sa dá určovať kvalita kódu. Avšak tieto vlastnosti nie sú striktne vyhradené a tím si pre projekt môže definovať svoje vlastné. Medzi často používané patrí spoľahlivosť, opakovaná využiteľnosť, prenosnosť, testovateľnosť [4]. K niektorým z týchto vlastností existuje jedna alebo viacero metrík, ktorými sa dá daná vlastnosť v kóde merať. Napríklad je možné zistiť, na aký podiel kódu sú vytvorené testy, ale čitateľnosť alebo pochopiteľnosť kódu je veľmi subjektívna. Prehľad analýz kvality zdrojového kódu a aj nástrojov, ktoré analýzu vykonávajú automaticky sa nachádza napríklad v bakalárskej práci *Analýza kvality zdrojových kódů* od Lukáša Voříška [85]. Niektoré súčasne používané *IDE* obsahujú prvky statickej analýzy a upozorňujú na možné problémy, už pri vývoji [85]. Kniha *Clean Code in C#* opisuje kvalitný kód, ako základnú vlastnosť softvéru [2]. Kód, ktorý dodržiava vysoké štandardy, bude mať kvalitnú výkonnosť, dostupnosť, bezpečnosť, škálovateľnosť, udržateľnosť, prístupnosť, nahraditeľnosť a rozšíriteľnosť [2].

### 2.1.3 Problémy špinavého kódu

Čistý aj špinavý kód je možné preložiť, špinavý kód je však špinavý z nejakého dôvodu [2]. Písanie čistého kódu nie je otázkou iba morálnych, etických hodnôt. Naviac, strávený čas, počas ktorého sa upravoval kód pre dodržanie vysokých kvalít, nie je vhodné ospravedlňovať dobrou programátorskou praxou, dôvody sú čisto ekonomické [18]. Kvalitný kód dokáže ušetriť čas, financie aj ľudské zdroje, v dlhšom časovom období. Nekonzistentný dizajn knižníc nepriaznivo ovplyvňuje produktivitu vývojárov [59].

Kód, ktorý je vyvíjaný, bez brania ohľadu na jeho čistotu, sa pridávaním ďalších funkcionalít stáva viac a viac špinavým. Čím je kód špinavším, tým ťažšie sa v ňom orientuje, pridávanie ďalších funkcionalít trvá dlhšiu dobu, a preto sa spomaľuje jeho vývoj. Miera spomalenia je výrazná, produktivita tímu klesá asymptoticky k nule [50]. Kód je vhodné čistiť postupne, čím dlhšie sa čistenie kódu odkladá, tým viac času zaberie. Zmena čistého kódu je oproti komplexnému špinavému kódu jednoduchšia [18]. Pomer času medzi čítaním a písaním je značne cez 10:1 [50]. Kód musíme čítať vždy, keď chceme písať ďalší a preto sa snažíme spraviť čítanie čo najjednoduchším, aj za cenu dlhšieho písania. Špinavý kód je významná prekážka a jediným spôsobom ako vyvíjať rýchlo, je udržiavať kód čistým [50].




Obr. 2.1: Klesanie produktivity pri špinavom kóde. Obrázok upravený z *Clean Code* [50].

Existujú rôzne metriky, ktoré dokážu vývojárom pomôcť merať čistotu kódu a tým určovať, ktoré časti kódu je vhodné *refaktorizovať* pre zvýšenie kvality (viď 2.1.2).

### 2.1.4 Čistý kód

Aký je to čistý kód? Robert C. Martin oslovil s touto otázkou viacero profesionálov v oblasti, ktorých následne v *Clean Code* cituje, čím ukazuje na fakt, že každý definuje čistý kód iným spôsobom [50]. On popisuje čistý kód ako remeslo a programátora ako umelca, ktorý dokáže premeniť prázdnu obrazovku radom transformácií, kým nezíska elegantne kódovaný systém [50]. Písanie čistého kódu podľa neho vyžaduje disciplínu použitia nespočetného množstva malých techník, aplikovaných prostredníctvom starostlivo získaného pocitu čistoty [50].

Existuje množstvo kníh a kurzov, ktoré učia technike písania čistého kódu. Niektoré z nich, ako napríklad *Clean Code*, boli použité aj pri písaní tejto práce. Keď si ich pozorne prečítame, tak zistíme, že sa v niektorých častiach líšia a niekde majú dokonca protichodné názory. Nie je jeden správny spôsob, ako napísať čistý kód, pretože kód nie je nikdy úplne čistý. Niekto môže za čistý považovať v danej situácii jeden konštrukt, ako niekto úplne iný. Všetko záleží od situácie. Aj v *Clean Code* sa píše, že nebudeme pravdepodobne súhlasiť 100%–ne so všetkým, čo v knihe nájdeme [50].

Záleží na detailoch [50]. Každá menšia časť kódu, ktorá nie je napísaná v súlade s definovanými pravidlami, otvára priestor pre rozšírenie, ktoré by viedlo k väčším nezrovnalostiam. Dobrí programátori vedia, že zriedkavo je prvotný kód tým čistým a preto trávia čas jeho čistením [18]. Pozeráme sa na to, ako rozdeliť program do menších modulov, ktoré spolu spolupracujú na jednom funkčnom riešení, ktoré je plne testovateľné, môže na ňom pracovať viacero tímov súčasne, a je omnoho jednoduchšie na čítanie, pochopenie a dokumentovanie [2]. Už v knihe *The C Programming Language* je odporúčané pracovať s viacerými menšími časťami, než s jednou veľkou, pretože nepodstatné detaily môžu byť zapúzdrené do funkcií a šanca na nežiadúce interakcie sa zmenšuje [74].

Kód časom degraduje, preto musíme byť aktívni v jeho prevencii [50]. Čistota kódu sa časom zhoršuje a nemusí to byť nutne spôsobené prepísaním jeho častí bez vyčistenia. Dôvodov je viacero. Môže sa jednať o zmenu pohľadu na to, čo je čisté, môžu byť do jazyka pridané nové, čistejšie konštrukty alebo sa zmení okolie kódu a tým pádom kód nie je čistý ako celok.

### 2.1.5 Dopad programovacích konvencií na čistotu kódu

Podľa Dave Thomasa, zakladateľa *OTI* a jedného z hlavných tvorcov nástroja *Eclipse*, poskytuje čistý kód skôr jeden spôsob, ako viacero, na vykonávanie jednej veci [50]. Kód môže obsahovať viacero programovacích štýlov, a teda jednotlivé časti kódu môže vývojár napísať rôznymi spôsobmi, čím sa kód stáva nekonzistentným. Konzistencia je definovaná ako rovnakosť, zhodnosť, uniformita [21]. Neochvejná konzistencia vedie ku kódu, pri ktorého spoznávaní môžete vytvoriť viacero predpokladov a predpovedí o jeho správaní, ktoré budu nakoniec pravdivé [21]. Ak na projekte pracuje viac vývojárov, čo je v dnešnej dobe pri rozšírení verzných systémov bežné, písanie čistého kódu je o konzistentnosti aj medzi nimi. Je oveľa jednoduchšie pochopiť veľký kód, keď je písaný konzistentným štýlom [36]. Jednotný, konzistentný kód pomáha k jeho skoršiemu pochopeniu, lepšej čitateľnosti a tým následne k rýchlejšiemu vývoju a jednoduchšej údržbe. Použitím programovacích konvencií sa zaručuje napísanie kódu v prijatom a dohodnutom formáte, čo pomáha ľudom sústrediť sa na podstatu kódu a tráviť menej času pochopením jeho usporiadania [2].

Každý väčší *open source* projekt má svojho vlastného sprievodcu štýlmi: množinu konvencií o tom, ako písať pre daný projekt kód [36]. Programovacie konvencie tvoria akúsi sadu zásad, ktorá sa dodržiava pri písaní kódu. Môžu byť podané iba ústne, ale aj formalizované do sady spísaných pravidiel. Programovacie štandardy stanovujú zásady ako písať a nepísať kód, ktoré je potrebné dodržiavať [2]. Pravidlá sa môžu deliť na mäkké a tvrdé, pri mäkkých sa dovoľuje v niektorých situáciách ich porušenie, zatiaľ čo tvrdé musia byť dodržané vždy. Napríklad Robert C. Martin považuje za mäkké pravidlo nevyužiť viac ako tri argumenty [84].

*Microsoft* má svoje programovacie konvencie (viď 2.2.3), ktoré zväčša bývajú preberané a upravené tak, aby vyhovovali potrebám konkrétneho podniku [2]. Každý projekt zvyčajne používa vlastné programovacie konvencie, ktoré bývajú bežne odvodené od štandardov vývojárskeho tímu a firemných pokynov, s prihliadnutím na využité technológie, programovacie jazyky a prostredie. Konvencie vytvárajú konzistentný vzhľad kódu, aby sa čitatelia mohli sústrediť na obsah, nie na jeho rozloženie [53]. Snažia sa umožniť čitateľom porozumieť kódu vytvorením predpokladov na základe prechádzajúcich skúseností a uľahčiť kopírovanie, zmeny a údržbu kódu, pričom uplatňujú osvedčené postupy [53]. Z pohľadu vývojového cyklu sú konvencie najčastejšie sledované pri kontrole navrhovaných zmien [2].

Zmienky o vhodnom písaní kódu je možné nájsť už v knihe *The C Programming Language* z roku 1978. Napríklad, podľa nej nie je vhodné písať kód, pri ktorom záleží na poradí vyhodnotenia alebo používať výraz *goto* ako náhradu iných konštruktov [74]. Odporúča použitie rekuzie a ukazovateľov. Využitie ukazovateľov väčšinou vedie k viac kompaktnému a efektívnemu kódu [74]. Kód využívajúci rekurziu je kratší a jednoduchší na napísanie a porozumenie, ako jeho nerekurzívny proťajšok [74]. Pre prehľad programovacích konvencií viď sekcia 2.2.

### 2.1.6 Iné spôsoby zvyšovania čistoty kódu

Programovacie konvencie sú jedným zo spôsobov zvyšovania čistoty kódu. Ako je uvedené v sekcii 2.1.5, nie vždy sú formalizované a zvyčajne sa na ne nekladie dôraz. Existujú ale aj rôzne iné mechanizmy, ktoré zvyšujú čistotu, medzi ktoré patrí napríklad refaktorizácia alebo testovanie.

*Refaktorizácia* je veľmi podobná optimalizácii výkonu, oboje upravujú kód, ale nemenia funkcionalitu, rozdielom je ich účel. Refaktorizácia je zmena vykonaná v internej štruktúre softvéru, vďaka ktorej je jednoduchší na pochopenie a jednoduchšie upraviteľný bez zmeny jeho pozorovateľného správania [18]. Je to kľúč k udržaniu kódu čitateľným a upraviteľným, preto je dôležitým prvkom v procese vývoja softvéru [18]. V dnešnej dobe sa čoraz častejšie používa slovo refaktorizácia pre akékoľvek čistenie kódu. Rôzne *IDE* pre rozličné programovacie jazyky obsahujú integrované formy refaktorizácie. Napríklad *ReSharper* obsahuje rýchle akcie na extrahovanie metódy alebo automatické vyčistenie kódu.

Na to, aby refaktorizácia bola vykonaná správne, je potrebná spoľahlivá sada testov, ktorá odhalí neodvratné chyby [18]. Čistotu a teda kvalitu kódu zvyšujú aj testy, no ich hlavným prínosom je hľadanie chýb vo funkcionalite a v dokumentovaní kódu. V metóde programovania *TDD* sa ako prvé píšu testy a až následne sa vytvára kód tak, aby testy končili úspechom [18]. Následne sa kód refaktorizuje, aby bol čo najčistejší, no stále s úspešne prechádzajúcimi testami [18]. Dôležitým efektom napísania testov ako prvých je, že testy fungujú ako neoceniteľná forma dokumentácie [51].

Ďalšími spôsobmi, ako zvýšiť čistotu je využitie návrhových vzorov a kontrola čistoty pri potvrdzovaní navrhovaných zmien. Polovica boja v programovaní čistého kódu je v správnej implementácii a použití návrhových vzorov [2]. V čistote vie pomôcť napríklad aj vhodná forma dokumentácie a pri agilnom vývoji nastavenie *DoD* s cielením na čistotu. Čistejší návrh na najvyššej úrovni aplikácie podporuje vytvorenie návrhu vo forme diagramov ešte pred začiatkom vývoja, napríklad pomocou *UML*.

## 2.2 Používané programovacie konvencie

Pred vytvorením návrhu aplikácie je vhodné zanalyzovať programovacie konvencie, ktoré používajú programátori pri vývoji. Prvá časť sa venuje analýze *open source* projektov, pričom skúma aké konvencie a technológie sa v nich používajú (viď 2.2.1). Nasledujúca časť podrobnejšie analyzuje konkrétne projekty z prvej časti a skúma spôsoby uchovávania a spravovania konvencií v nich (viď 2.2.2). Posledná časť analyzuje programovacie konvencie v ďalších zdrojoch, mimo analyzované projekty (viď 2.2.3).

Analýza sa nesnaží do detailov preskúmať všetky druhy programovacích konvencií a ani rozhodovať, aké konvencie sú v akej situácii vhodné. Usiluje sa iba o zistenie, aké konvencie sa v praxi reálne používajú, čo je následne využité pri návrhu systému na ich správu. Aké

konvencie nakoniec užívateľ použije, ponecháva na jeho vlastnom rozhodnutí, pretože on sám najlepšie vie, čo je pre jeho projekt v aktuálnej situácii vhodné.

### 2.2.1 Analýza programovacích konvencií v open source projektoch

Zistiť, aké programovacie konvencie sa používajú pri vývoji *open source* projektov je jednou z hlavných častí analýzy, ktorú je vhodné spraviť pred vytvorením návrhu systému. Z každoročnej správy *Octoverse*, ktorá ukazuje vývoj služby *GitHub* za posledný rok, je možné pozorovať, ako *open source* komunita rastie. V grafe z roku 2020 je možné vidieť, ako percentuálne rástol počet vytvorených *open source* projektov na aktívneho užívateľa, v porovnaní s minulým rokom [31]. Počet príspevkov do *open source* projektov sa oproti predchádzajúcemu roku zvýšil o dvadsaťpäť percent, čomu však zrejme pomohol aj COVID–19 lockdown, pri ktorom je značný nárast [31].

Ako zdroj *open source* projektov som zvolil službu *GitHub*. *GitHub* má najväčšiu *open source* komunitu na svete, ktorú tvoria milióny projektov [27]. Komunitu *GitHubu* tvorilo v roku 2020 cez päťdesiatšesť miliónov užívateľov a viac ako šesťdesiat miliónov repozitárov [31]. Za rok 2020 bolo vytvorených takmer dve miliardy príspevkov [31]. Ku *GitHub* projektom je možné nájsť v dátovom sklade *BigQuery* voľne prístupné dáta, z ktorých som pri analýze čerpal.

#### Výber analyzovaných projektov

Dátových sád ku *GitHub* projektom je viac. *GHTorrent* [22] je relačná dátova sada, prístupná z BigQuery, ktorú som ale nevyužil, keďže nie je aktualizovaná pravidelne a jej posledná aktualizácia prebehla v júni 2019. K dispozícii je aj databáza obsahujúca kópiu *ASCII* súborov menších ako 10 megabajtov, ak sa nachádzaju v *open source* projekte [40]. Tú som taktiež nevyužil, pretože posledná aktualizácia prebehla v marci 2019 a ani vtedy neobsahovala všetky *open source* repozitáre, ale iba ich vybranú časť. Rozhodol som sa použiť *GH Archive*, ktorý sleduje verejnú časovú os *GitHubu*, ktorú archivuje a sprístupňuje pre ďalšie analyzovanie [38]. *GH Archive* je aktualizovaný každú hodinu a je k nemu umožnený prístup okrem BigQuery[1] aj pomocou *HTTP* klienta.

Pre analýzu som sa rozhodol vybrať päťdesiat *open source* projektov, ktorých konvencie som následne skúmal. Získal som zoznam projektov, ktoré dostali najviac hviezdičiek (*stargazers*) za rok 2020. Následne som ich manuálne prechádzal a odfiltroval tie, ktoré nemali open source licenciu alebo boli v inom jazyku, ako v angličtine a teda nebolo možné ich pre analýzu použiť. Týmto spôsobom som prechádzal projekty, pokiaľ som nezískal prvých päťdesiat, ktorých zoznam je v dostupný v prílohe A. Zvolil som tento spôsob, pretože som chcel čo najmenej zasahovať do výberu projektov, aby som zamedzil skresleniu vyvodených výsledkov analýzy a získal čo najobjektívnejšie výsledky. Môj vplyv na výber projektov do analýzy teda minimalizoval a odfiltroval iba nepoužiteľné projekty.

Pri analýze som sa rozhodoval nad viacerými spôsobmi výberu projektov. Snažil som sa o výber väčších projektov, pretože pri nich je vyššia pravdepodobnosť že budú obsahovať programovacie konvencie (viď 2.1.5). Malé projekty často vyvíja jeden alebo malé množstvo programátorov, a preto v nich vo väčšine prípadov niesu programovacie konvencie explicitne uvedené a riešia sa osobitne (napríklad ústne alebo v *Pull requestoch*). Najväčšie projekty majú častejšie vlastné, jedinečné, riešenia spravovania programovacích konvencií. Výber podla hviezdičiek (*stargazers*) obsahoval prierez ako menších, tak aj väčších projektov, pri-

[1] https://console.cloud.google.com/bigquery?project=githubarchive&page=project

čom väčších obsahoval viac, a preto na analýzu boli považované za najvhodnejšie. Zároveň som sa snažil o projekty, ktoré sú stále aktívne a využívané, preto som zvolil projekty iba za rok 2020.

### Výsledky analýzy

Každý z vybraných projektov som skúmal manuálne, otvorením ich repozitára v službe *GitHub*, pretože z dát dostupných na BigQuery nebolo umožnené získať potrebné informácie. V tejto časti vysvetľujem, aké dáta som v projektoch analyzoval, uvádzam dôvody ich analýzy a aké hodnoty som získal. Pre bližšie informácie o konvenciách na konkrétnych vybraných projektoch viď časť 2.2.2, pre informácie o konkrétnych technológiách zaisťujúcich uchovávanie, dodržovanie a spravovanie programovacích konvencií viď časť 2.3. Pre využitie získaných dát pri návrhu viď časť 3.1.1.

V prvom kroku som zisťoval, či repozitáre obsahujú zdrojový kód a teda či vôbec dáva zmysel, aby obsahovali programovacie konvencie. Za repozitár, ktorý obsahuje kód som bral do úvahy taký, ktorého kód stojí za zmyslom repozitára. Ak napríklad repozitár obsahoval iba *Markdown* súbory a jeden krátky skript, používaný na vyčistenie ich formátovania, pričom v repozitári išlo primárne o obsahy *Markdown* súborov, tak som to nebral ako repozitár obsahujúci kód. A to z dôvodu, že by nedávalo zmysel, aby obsahoval programovacie konvencie. Tridsaťpäť z päťdesiat repozitárov obsahovalo zdrojový kód.

V repozitároch som hľadal aj nejaký druh sprievodcov prispievaním, čím som sa snažil zistiť, či sa v danom repozitári vôbec očakáva prispievanie a dáva zmysel, aby repozitár obsahoval programovacie konvencie. Tridsaťdeväť z päťdesiat repozitárov obsahovalo nejaký druh vysvetlenia ako prispievať, pričom najčastejšie ho obsahoval súbor `CONTRIBUTING.md`. Pri sprievodcoch prispievaním som pozoroval tri kategórie obsahu. Sprievodcov obsahujúcich konvencie určujúce ako pracovať s repozitárom, teda ako písať a čo majú obsahovať *Issue*, ako vytvárať *Pull requesty* a ostatné veci, týkajúce sa práce s *Gitom* a *GitHubom*. Sprievodcov s takýmto obsahom bolo tridsaťšesť. Sprievodcov prispievaním, ktorí obsahovali iný obsah, prevažne etické hodnoty a princípy udržiavané v repozitári, bolo tridsaťsedem a sprievodcov obsahujúcich programovacie konvencie bolo štrnásť, čo tvorí dvadsaťosem percent zo všetkých päťdesiatich repozitárov. Avšak, ak odhliadneme od repozitárov, ktoré neobsahovali žiadny kód alebo ani žiadneho sprievodcu prispievaním, tak zistíme, že repozitárov s programovacími konvenciami je polovica.

Tabuľka 2.1: Súhrn projektov podľa kategórie konvencií

| | Počet projektov |
|---|---|
| Celkovo | 50 |
| Obsahujúcich iné konvencie | 37 |
| Obsahujúcich Git/GitHub konvencie | 36 |
| Obsahujúcich programovacie konvencie | 14 |
| Bez konvencií | 8 |

Tabuľka 2.2: Súhrn projektov podľa technológii spravujúcich konvencie

| | Počet projektov |
|---|---|
| Celkovo | 50 |
| Obsahujúcich konvencie v Markdown súboroch | 34 |
| Obsahujúcich konvencie v Issue šablóne | 25 |
| Obsahujúcich konvencie v EditorConfig súbore | 18 |
| Obsahujúcich konvencie na externom odkaze | 8 |
| Obsahujúcich konvencie vo Wiki | 7 |
| Bez konvencií | 8 |

Medzi konvenciami v jednotlivých repozitároch je veľká diverzita, ktorú spôsobujú rôzne programovacie jazyky a prístupy, a preto je možné v nich nájsť prakticky všetky druhy konvencií, opísaných v odbornej literatúre (viď 2.2.3). Medzi najčastejšie nájdené patria menné konvencie a konvencie upravujúce formátovanie, čo je pravdepodobne spôsobené tým, že sú to všeobecnejšie kategórie, použiteľné na väčšinu kódu. Často sa konvencie zaoberajú aj knižnicami a testami, prípadne celkovo testovateľnosťou kódu. Pri jazykoch kde to dáva zmysel, je možné nájsť konvencie zaoberajúce sa typmi, premennými, dokumentáciou ku kódu a komentármi. Všeobecne, konvencie väčšinou prikazujú, akým konštruktom sa vyhnúť, ktoré použiť a v akých prípadoch tak učiniť. Zväčšia ide o nižšie konštrukty a iba v ojedinelých prípadoch riešili väčšie súvislosti, ako napríklad návrhové vzory obsahujúce viac prepojených tried v objektovo orientovaných jazykoch. Ku konštruktom na najnižšej úrovni, ako je napríklad ukončenie súboru novým riadkom, znak na konci riadku alebo kódovanie súborov sú používané *EditorConfig* súbory, prípadne iné súbory, používané na kontrolu špecifických konvencií daného jazyka pomocou *IDE* alebo externého programu. Pokiaľ konvencie nejakú tematiku neriešili explicitne, tak vo väčšine prípadov nabádali čitateľa k dodržiavaniu štýlu už existujúceho kódu.

### 2.2.2 Analýza vybraných projektov

Časť práce, ktorá sa zaoberá konkrétnymi projektami a využívaním programovacích konvencií v nich. Časť analyzuje konkrétne projekty, vybrané z analýzy *open source* projektov v službe *GitHub* (viď časť 2.2.1) a zároveň sleduje, akými spôsobmi sú konvencie v projektoch uchovávané a spravované, ako ku konvenciám programátori pristupujú, ako sa s nimi nový programátori zoznamujú a ako sa overuje alebo vynucuje ich dodržiavanie. Môj výber spočíval len v projektoch, ktoré obsahujú programovacie konvencie.

Pri výbere analyzovaných projektov som sa snažil dosiahnuť rôznorodosť, ako podľa obsahu, tak aj podľa spôsobu uchovávania programovacích konvencií. Mojou snahou bolo poukázať na to, ako sa reálne technológie, spravujúce programovacie konvencie na projektoch používajú, aké druhy konvencií sa využívajú a ako komunikujú s programátorom. Projekty som nevyberal iba na základe môjho subjektívneho názoru, ale aj podľa prvenstva v stanovených kategóriách. Ako prvý projekt analyzujem *The Algorithms — Python* obsahujúci rôzne algoritmy, implementované v jazyku *Python*. Druhým projektom je *Microsoft PowerToys* uchovávajúci sadu nástrojov pre efektívnejšiu prácu so systémom *Windows 10*. Posledným projektom je *React*, knižnica v jazyku *JavaScript* pre vytváranie užívateľských rozhraní.

### The Algorithms — Python

*The Algorithms — Python* [83] je repozitár, obsahujúci algoritmy, implementované v jazyku *Python*. Napísaný je v jazyku *Python* a prispelo do neho cez šesťstopäťdesiat užívateľov. Repozitár patrí do najväčšej *open source* knižnice algoritmov v službe *GitHub* pre rôzne programovacie jazyky. Jazyky, spolu aj s ich repozitármi, sú dostupné na oficiálnej stránke *The Algorithms*[2]. Algoritmy sú určené iba na edukačné a demonštračné účely, pričom nemusia byť vždy najoptimálnejšie. Súbor `DIRECTORY.md` obsahuje zoznam algoritmov v repozitári, s odkazmi na súbory, v ktorých sú implementované. Jeden algoritmus je napísaný v jednom *Python* súbore. Repozitár som zvolil na analýzu, pretože je prvým zo zoznamu analyzovaných projektov (viď časť 2.2.1), ktorý obsahuje programovacie konvencie.

V *Markdown* súbore `CONTRIBUTING.md` sa nachádzajú pokyny pre prispievateľov do repozitára. Pred samotným prispením sa očakáva ich prečítanie a dodržanie, pričom na prípadné otázky je možné použiť *Issue* alebo *Gitter*. Pre *Markdown* súbory ako technológiu, spravujúcu konvencie viď 2.3.1. Medzi pokynmi sa v *Markdown* súbore nachádzajú aj samotné konvencie. Konvencie riešia verziu programovacieho jazyka, správne pomenovania s odkazom na externé menné konvencie, spôsob komentovania a dokumentovania a preferenciu určitých konštruktov jazyka nad inými. Pokyny, okrem konvencií obsahujú aj všeobecné zásady, netýkajúce sa programovania, spôsob akým v projekte funguje automatické testovanie a teóriu algoritmov.

Zmeny v konvenciách je možné sledovať v histórii súboru, keďže je uložený priamo v *Git* repozitári projektu. Dodržanie konvencií je kontrolované pomocou pluginu *pre–commit* a pomocou programov *black* a *flake8* (viď 2.3.5), čím sa kód naformátuje do správneho tvaru. Následne je manuálne skontrolovaný v *Pull requeste*. Testy sú automaticky spúšťané na *Travis CI* s využitím *Pytest* a *Mypy* kontrolujúcim typy (viď 2.3.5). Repozitár obsahuje aj jednoduchú Wiki (viď 2.3.3) obsahujúcu základné informácie o repozitári a odkazy na manuál k prispievaniu a *Gitter*.

### Microsoft PowerToys

*Microsoft PowerToys* [63] je repozitár, uchovávajúci sadu nástrojov pre pokročilých užívateľov, ktorí chcú zefektívniť svoju prácu so systémom *Windows 10* a dosiahnuť tak vyššiu produktivitu. Medzi hlavné nástroje patrí *Color Picker*, umožňujúci získať farbu z akejkoľvek práve bežiacej aplikácie, *Fancy Zones* spravujúci okná a umožňujúci vytváranie ich komplexných rozvrhnutí a *PowerToys Run* určený na vyhľadávanie a spúšťanie aplikácií okamžite. Do repozitára prispelo cez stoosemdesiat užívateľov, pričom obsahuje kód prevažne v jazykoch C++ a C#. Prehľad nástrojov aj s návodom na inštaláciu sa nachádza v *Microsoft* dokumentácii[3]. K analýze som zvolil repozitár, pretože je prvým z analyzovaných projektov (viď časť 2.2.1), ktorý obsahuje programovacie konvencie a zároveň je od firmy *Microsoft* a obsahuje kód v programovacom jazyku C#. Cielenie na .NET technológie som rozoberal v časti 3.1.2.

V súbore `README.md`, ktorý sa zobrazuje na hlavnej stránke repozitára, sa nachádza časť pre prispievateľov. V časti sa píše o tom, ako projekt víta prispievanie všetkých druhov, od pomoci so špecifikáciou, dizajnom, dokumentáciou, hľadaním chýb s ktorými môže pomôcť každý, až po programovanie funkcií a opravovanie chýb. Následne časť obsahuje odkaz na *CLA*, návod pre prispievateľov a vývojársku dokumentáciu. Okrem toho repozitár obsahuje aj `COMMUNITY.md` súbor s užívateľmi, ktorí mali veľký prínos pre projekt

[2] https://thealgorithms.github.io
[3] https://docs.microsoft.com/sk-sk/windows/powertoys

a Wiki (viď 2.3.3) so základnými informáciami pre užívateľov s odkazmi na viac informácií v *Microsoft* dokumentácii.

V repozitári sa nachádza aj súbor `CODE_OF_CONDUCT.md` obsahujúci odkaz na *Microsoft* kódex správania, ktorý definuje správanie konvencií v projekte. V návode pre prispievateľov sa nachádzajú konvencie k vytváraniu a prispievanie do *Issue*. V závere návodu je odkaz do vývojárskej dokumentácie, ktorá obsahuje konvencie k práci so samotným repozitárom. Vývojárska dokumentácia je rozložená do viacerých *Markdown* súborov a sú v nej dokumentované významné triedy, štruktúra kódu, lokalizácia, logovanie, nastavenia, diagnostika, pracovný tok repozitára, kompilácia a aj programovacie konvencie. Programovacie konvencie sa zaoberajú najmä formátovaním, k čomu používajú *ClangFormat* (viď 2.3.5). Okrem toho obsahuje repozitár aj *EditorConfig* súbor (viď 2.3.4), ktorý v programovacích konvenciách spomenutý nie je, ktorý v dvoch prípadoch upravuje vážnosť problému nájdeného pri analýze.

Pri vytváraní *Issue*, je na výber zo štyroch šablón (viď 2.3.2), ktoré obsahujú základné body, ktoré by mali jednotlivé typy *Issue* obsahovať. Typmi šablón sú zadanie chyby, dokumentácie, požiadavka funkcie a lokalizácia. Okrem toho obsahuje okno vytvárania *Issue* aj odkaz na nahlasovanie bezpečnostných chýb, k čomu je v repozitári vytvorený samostatný *Markdown* súbor a odkazy na užívateľskú a vývojársku dokumentáciu. Okrem toho vytváranie *Issue* upravuje aj `SUPPORT.md` súbor a súbor pre prispievateľov. Zmeny v konvenciách, nachádzajúcich sa v *Markdown* súboroch alebo v súbore *EditorConfig* a *ClangFormat* je možné sledovať v histórii súboru, keďže sú uložené v systéme na kontrolu verzií *Git*. Zmeny v šablónach nie je možné sledovať, avšak ak k zmene dôjde, do všetkých existujúcich *Issue* je možné pridať komentár na vyžiadanie ich úpravy. Dodržanie kódexu správania je kontrolované pri každej komunikácii v projekte, programovacie konvencie sú kontrolované pomocou automatizovaných nástrojov *ClangFormat* a *EditorConfig* a následne manuálne kontrolované pri *Pull requeste*.

### React

*React* [17] je *JavaScriptová* knižnica pre vytváranie užívateľských rozhraní. Jej základnou vlastnosťou je deklaratívnosť, React umožňuje jednoducho vytvárať interaktívne užívateľské rozhrania, pre každý stav aplikácie, čo robí kód predvídateľnejším, zrozumiteľnejším a jednoduchšie sa v ňom hľadajú chyby. React je založený na zapuzdrených komponentoch, ktoré riadia svoj vlastný stav, efektívne sa vykresľujú a menia, pri zmene údajov na pozadí. Repozitár použilo cez päť miliónov užívateľov a cez tisíc päťsto do neho prispelo. Projekt som zvolil na analýzu, pretože ako prvý obsahuje odkaz na externú dokumentáciu a nepoužíva pre uchovávanie *Markdown* súbory. Zaujímavý je aj používaním viacerých automatizovaných nástrojov na kontrolu programovacích konvencií.

Repozitár víta užívateľov k prispievaniu, čo je uvedené priamo na úvodnej stránke. K repozitáru je vytvorená Wiki (viď 2.3.3) obsahujúca odkaz na externú dokumentáciu[4]. Vytváranie *Issue* obsahuje jednu šablónu pre chybu, obsahujúcu nutné časti, ostatné druhy issue sú bez šablóny (viď 2.3.2). *Markdown* súbory neobsahujú konvencie, ale odkazujú sa na externú dokumentáciu, v prípade kódexu správania na *Facebook* konvencie, určujúce správanie sa v projekte[5], v otázkach bezpečnosti na *Facebook* whitehat[6].

[4] https://reactjs.org/
[5] https://engineering.fb.com/codeofconduct/
[6] https://www.facebook.com/whitehat

Externá stránka obsahuje rozsiahlu dokumentáciu k celému projektu, jej časť je definovaná pre prispievateľov. Táto časť obsahuje konvencie k vytváraniu *Issue* s chybou, žiadaním o zmenu alebo návrhom novej funkcionality. Okrem toho sa zameriava aj na vytvorenie *Pull requestu* s novým kódom, pričom popisuje celý proces, ktorý je nutné dodržať, čo je vhodné si pred prispievaním prečítať. Programovacie konvencie sa v procese kontrolujú pomocou programu *Linter* a formátovanie sa nastavuje pomocou programu Prettier, pre viac informácií o fungovaní programov viď 2.3.5. Okrem toho repozitár obsahuje aj *EditorConfig* súbor (viď 2.3.4) obsahujúci konvencie pre všetky súbory s upresnením konvencií pre *Markdown* súbory. Zmeny konvencií je možné v prípade programov kontrolovať v histórii súborov v repozitári, ktoré dané programy nastavujú. V prípade zmeny v procese na externej stránke nie je možné zmeny sledovať. Dodržanie programovacích konvencií sa kontroluje okrem automatizovaných nástrojov aj v *Pull requestoch*, správanie podľa kódexu je kontrolované vo všetkej komunikácii.

### 2.2.3 Ďalšie zdroje programovacích konvencií

K vytvoreniu komplexného prehľadu využívaných programovacích konvencií je vhodné zanalyzovať aj konvencie z ďalších zdrojov, nie iba z vybraných *open source* projektov analyzovaných v prechádzajúcich častiach. Vzhľadom na veľké množstvo dostupných konvencií k rôznym jazykom, som sa rozhodol spracovať zvlášť všeobecné zdroje, využiteľné pri všetkých jazykoch a zvlášť zdroje k jednému, referenčnému jazyku. V prvej časti analyzujem programovacie konvencie použiteľné všeobecne pre ľubovoľný jazyk a v druhej časti sa venujem programovacím konvenciám ku jednému konkrétnemu programovaciemu jazyku.

#### Všeobecné programovacie konvencie

Časť práce skúmajúca programovacie konvencie použiteľné všeobecne pre ľubovoľný programovací jazyk. Cieľom je získať na používané programovacie konvencie aj iný pohľad, mimo analýzy *open source* projektov z časti 2.2.1 a 2.2.2. Skúma, aké druhy konvencií by mohol chcieť použiť v projekte, čo je následne vhodné zohľadniť pri návrhu systému na ich správu. Pri analýze som cielil na odbornú literatúru, z ktorej väčšina konvencií vychádza.

**Clean Code** [50] s podtitulom *A Handbook of Agile Software Craftsmanship* je kniha od Robert C. Martina, ktorá sa zameriava na písanie čistého kódu. Knihu autor rozdeľuje do troch častí. Niekoľko prvých kapitol popisuje princípy, vzorce, postupy a vhodné praktiky pri písaní čistého kódu, ktoré obsahujú príklady v zdrojovom kóde jazyka Java [50]. Druhá časť knihy pozostáva z niekoľkých prípadových štúdií, učiacich techniku čistenia a transformácie kódu na jeho čistejšiu, menej problémovú variantu [50]. Tretia časť obsahuje zoznam heuristík a problémov zhromaždených z prípadových štúdií z druhej časti, ktoré slúžia ako základňa znalostí pre ďalšie použitie [50].

V knihe je možné nájsť veľké množstvo rôznych druhov programovacích konvencií. Obsahuje menné konvencie, konvencie k funkciám, objektom, triedam, dátovým štruktúram a k písaniu testov [50]. Následne rozoberá formátovanie kódu, komentovanie kódu, prácu s výnimkami, ale aj s kódom tretích strán a vytváraním hraníc medzi nim a vlastnými časťami kódu [50]. V neposlednom rade rieši aj konvencie na vyšších úrovniach, ako sú vzťahy medzi objektami, asynchrónnosť, návrhové a architektonické vzory [50].

**Clean Code in C#** [2] s podtitulom *Refactor your legacy C# code base and improve application performance by applying best practices* je kniha venujúca sa zaužívaným prakti-

kám pri písaní kódu, a teda aj programovacím konvenciám. Zaoberá sa správnym písaním metód, tried, čistých funkcií alebo správnemu využitiu dedičnosti a vyhadzovaniu výnimiek [2]. Opisuje spôsoby ako testovať kód, používať asynchrónnosť, a ako vhodne vytvárať *API*, kontrolovať kód a spravovať projekt [2]. Kniha okrem iného opisuje aj správne a nesprávne spôsoby komentovania kódu, organizácie tried, objektov, menných priestorov a dátových štruktúr [2].

A okrem opísaných kategórií konvencií, kniha obsahuje aj všeobecné princípy, používané pri písaní kódu. Princíp *KISS (keep it simple, stupid)* cieli na písanie jednoduchého, až primitívneho kódu [2]. *YAGNI (you aren't gonna need it)* je spôsob programovania, ktorý dovoľuje pridať ďalší kód len keď je absolútne nutný a teda drží množstvo kódu na absolútnom minime [2]. Princíp *DRY (don't repeat yourself)* sa snaží o zníženie opakujúcich sa častí kódu pomocou *refaktorizácie* [2]. *SOLID* je sada piatich princípov návrhu, ktorých cieľom je vytvoriť softvér rozšíriteľný bez nutnosti úprav veľkých častí už existujúceho kódu, ktorý je jednoduchý na čítanie, pochopenie a udržiavanie [2]. Occamova britva je fráza, ktorá hovorí, že najjednoduchšie riešenie je najpravdepodobnejšie to správne [2].

**Agile principles, patterns, and practices in C#** [51] je kniha ilustrujúca základy *agilného* vývoja a dizajnu na sérii prípadových štúdií. Okrem konvencií zameriavajúcich sa na správne a nesprávne riadenie projektu *agilným* spôsobom obsahuje kniha aj konvencie pre vyššie, abstraktnejšie časti rozsiahlych systémov a konvencie návrhových a architektonických vzorov. Taktiež rozoberá jednotlivé princípy *SOLID* (viď časť Clean Code in C#) a konvencie využívané pri návrhu aplikácií.

**Refactoring** [18] s podtitulom *Improving the Design of Existing Code* je kniha, ktorá cieli na vylepšenie dizajnu stávajúcej verzie kódu. Obsahuje príklady, ako nevhodne napísať kód a vysvetľuje, ako ho správne upraviť. Okrem úprav funkcií, mien alebo komentárov sa zameriava aj na jednotlivé prvky ako sú cykly, podmienené príkazy a odložené vyhodnocovanie [18]. Taktiež sa venuje úpravám dedičnosti, zapuzdrenia, testov alebo *API* [18].

### Programovacie konvencie k jazyku c#

V úvode som spomínal, že systém je primárne určený pre projekty, ktoré využívajú *.NET* ekosystém. Ako kandidáta som zvolil programovací jazyk C#, pričom som kládol dôraz na dobrú dostupnosť relevantných zdrojov a na časté použitie pri vývoji *enterprise aplikácií*. Následne som vyhľadal konvencie a štýly dostupné k tomuto jazyku.

*.NET* a C# som zvolil aj preto, pretože vytváraný systém je napísaný primárne v ňom a v ankete z roku 2019 zo *StackOverflow* skončil *.NET* na druhom a *.NET Core* na treťom mieste medzi najviac populárnymi rámcami [76]. Zároveň sa *.NET Core* umiestnil ako prvý v najobľúbenejších rámcoch mimo web a C# ako desiaty najobľúbenejší jazyk [76].

**Microsoft dokumentácia** [62] je oficiálna technická dokumentácia pre konečných užívateľov, vývojárov a IT špecialistov, ktorí pracujú s *.NET* ekosystémom. Teda aj pre jazyk C#, ktorý bol vyvinutý okolo roku 2000 spoločnosťou *Microsoft*. Väčšina obsahu dokumentácie je *open source*. Článok dokumentácie je reprezentovaný ako *Markdown* súbor v *GitHub* repozitári. K jazyku C# obsahuje dokumentácia rôzne kategórie článkov: dokumentuje základy práce s jazykom, prehliadku jeho funkcionalít, zmeny v jednotlivých verziách a iné [54]. V dokumentácii sa nachádza aj stránka, rozoberajúca možnosti nastavovania štýlov kódu pomocou *EditorConfig* súborov (viď 2.3.4).

Časť dokumentácie sa zaoberá programovacími konvenciami k jazyku C#, ktoré odporúča autor jazyka jeho užívateľom [53]. Rozdeľujú sa na menné, konvencie rozvrhnutia a konvencie komentovania [53]. Následne obsahujú pokyny k jazyku, v ktorých informuje o vhodných konštruktoch a vítaných oblastiach ich využitia [53]. V poslednej časti sa nachádza odkaz na pokyny k bezpečnému programovaniu [53].

V dokumentácii sa nachádza aj sekcia, ktorá poskytuje pokyny pre navrhovanie knižníc, ktoré rozširujú a komunikujú s *.NET* rámcom [59]. Odporúča pokyny pre návrh a snaží sa pomôcť zaistiť konzistenciu *API* a jednoduché použitie poskytnutím jednotného programovacieho modelu, ktorý nie je závislý na programovacom jazyku použitom pri vývoji [59]. Sekcia sa delí na pokyny pomenovania, pokyny pre návrh typov, pokyny pre návrh členov, pokyny pre rozšíriteľný návrh, pokyny pre použitie výnimiek, pokyny použitia bežných typov vo verejne prístupných *API* a na návrhové vzory [59].

**Google Style Guides** [36] je repozitár[7] v službe *GitHub*, ktorý je exportovaný do internetovej stránky. Obsahuje sprievodcov programovacími štýlmi, ktoré používa spoločnosť *Google* pre svoj kód [36]. Štýly sú uložené v *XML*, *HTML* a *Markdown* súboroch pre mnoho programovacích jazykov, medzi ktorými je aj C#. Repozitár je pod *CC* licenciou, ktorá umožňuje jeho používanie, pod podmienkou uvedenia autora.

Štýly pre programovací jazyk C# boli vyvinuté interne v spoločnosti a sú používané ako východiskové pre ich kód [34]. Poskytujú štylistické voľby, ktoré zodpovedajú iným v *Google*, napríklad pre programovacie jazyky C++ a Java, pričom vychádzajú z menných konvencií pre jazyk C# od spoločnosti *Microsoft* a z konvencií v *dotnet runtime* repozitári[8] [34]. Rozdelené sú sa na dve časti: formátovanie jazyka a kódovanie. Časť o formátovaní obsahuje menné konvencie, organizáciu poradia kódu a pravidlá používania neviditeľných znakov, nasledovanú rozsiahlym príkladom súboru, na ktorom sú konvencie použité a vysvetlené v komentároch. Kódovanie obsahuje pravidlá vhodnosti a nevhodnosti použitia jednotlivých komponentov jazyka v určitých situáciách.

**Dofactory** [9] je americká firma, poskytujúca rámce návrhových vzorov pre *.NET*, *SQL* a *JavaScript*. Snaží sa poskytovať jednoduchú a čistú architektúru, pre rýchly vývoj aplikácií. Na ich stránkach je možné nájsť programovacie konvencie, rozdelené na sedemnásť častí, ktoré pri vývoji dodržujú a odporúčajú dodržiavať aj vývojárom pracujúcim s ich produktami [7]. Každá z nich opisuje, ako daný konštrukt písať, nepísať alebo čomu sa treba radšej vyhnúť [7]. Pri každej časti je príklad použitia, odôvodnenie a v niektorých prípadoch aj príklad nesprávneho použitia alebo výnimiek [7]. Obsahovo sa jedná o pravidlá, konzistentné s odporúčanými konvenciami od spoločnosti *Microsoft* pre .NET rámec, pričom niektoré konvencie sú rozšírené.

**GeeksforGeeks** [20] je internetový portál, vytvorený pre poskytovanie dobre napísaného, premysleného a vysvetleného riešenia otázok z oblasti programovania, algoritmizácie a pohovorov. Na portáli sa nachádza množstvo článkov, ako k programovaciemu jazyku C#, celému ekosystému .NET, tak aj k iným technológiám. Okrem článkov je v portáli možné nájsť aj pracovné ponuky, online kurzy a iné.

Článok s programovacími štandardami na začiatku opisuje jazyk C# a jeho základné charakteristiky a históriu [69]. Následne popisuje jedenásť praktík, ktoré je vhodné pri písaní

[7] https://github.com/google/styleguide

[8] https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/coding-style.md

kódu v jazyku využívať [69]. Ide prevažne o menné konvencie, ktoré vychádzajú z konvencií spoločnosti *Microsoft*. Okrem nich rieši aj zarovnanie textu, zapuzdrenie a vhodné použitie niektorých konštruktov.

K programovaciemu jazyku C# je možné nájsť online veľké množstvo konvencií, ktoré si vývojári upravujú podľa vlastných potrieb. Príkladom je príspevok o programovacích štandardoch na stránke *C# Corner*, ktorého cieľom je definovať pokyny, ktoré zabezpečia konzistentný štýl a formátovanie kódu, čo má pomôcť vývojárom vyhnúť sa bežným chybám [3]. Ďalším príkladom je článok na *SubMail* blogu, ktorý uvádza do základných konvencií rozdelených do deviatich častí [86]. Väčšina konvencií vychádza z vyššie zmienených alebo všeobecných konvencií (viď prvá časť 2.2.3) aplikovaných priamo na jazyk C#.

## 2.3 Analýza technológií spravujúcich programovacie konvencie

Súčasťou analýzy, ktorú je vhodné spraviť pred návrhom samotného systému, je analýza technológií, ktoré slúžia na uchovávanie, dodržiavanie a spravovanie programovacích konvencií v projektoch. Analýza slúži k zhodnoteniu súčasného stavu v oblasti technológií, používaných k spravovaniu programovacích konvencií a využíva poznatky z časti 2.2.1, v ktorej som analyzoval päťdesiat projektov v službe *GitHub*.

Táto časť práce analyzuje konkrétne technológie, spravujúce programovacie konvencie v projektoch, pričom uvádza príklady projektov, v ktorých sa dané technológie na správu konvencií využívajú. V prvej časti sa zaoberá *Markdown* súbormi (viď 2.3.1), v druhej šablónami pre *Issue* (viď 2.3.2) a v ďalšej Wiki stránkami, spolu s externými webovými stránkami (viď 2.3.3). V neposlednom rade preberá *EditorConfig* s jeho rozšíreniami (viď 2.3.4) a ostatné, menej často používané technológie (viď 2.3.5).

### 2.3.1 Markdown súbory

*Markdown* [79] je nástroj na konverziu textu na HTML pre tvorcov webových stránok. Umožňuje tvoriť obyčajný text, ktorý sa jednoducho píše a číta. Ten sa potom prevádza na štrukturálne platný *XHTML* (alebo *HTML*) kód. Markdown sa skladá z dvoch častí: syntaxe formátovania obyčajného textu a programu napísaného v Perl jazyku, ktorý text prevádza do *HTML*. Hlavným cieľom syntaxe je dosiahnuť, aby bol jazyk čo najčitateľnejší. Existuje niekoľko jednoduchých značkovacích jazykov, ktoré sú nadmnožinou Markdownu [5].

Markdown [29] je teda jednoduchá a ľahko použiteľná syntax pre štýlovanie všetkých foriem písma na platforme *GitHub*. *GitHub* používa svoju vlastnú verziu syntaxe[9], nazývanú *GitHub Flavored Markdown*, ktorá poskytuje ďalšiu sadu užitočných funkcionalít, z ktorých mnohé uľahčujú prácu s obsahom. Nie však všetky funkcionality sú dostupné kdekoľvek, niektoré sú dostupné iba v popisoch a komentároch *Issue* a *Pull requestoch*. Markdown môže byť použitý na väčšine miest na *GitHube*, medzi ktorými sú aj súbory s príponami *.md* alebo *.markdown*, ktoré je možné upravovať online, priamo na stránkach *GitHubu* a v reálnom čase pozorovať zmeny.

[9] https://github.github.com/gfm

Najčastejšie, v tridsiatichštyroch prípadoch z päťdesiat, boli Markdown súbory použité ako sprievodcovia prispievaním a teda obsahovali programovacie konvencie (viď 2.2.1), príkladom projektu, ktorý uchováva programovacie konvencie v Markdown súboroch je *The Algorithms — Python* (viď časť 2.2.2). Časté použitie Markdown súborov prisudzujem jednoduchému použitiu, ktoré je priamo integrované do platformy *GitHub*, ako aj vysokej flexibilite, ktorá umožňuje napísať konvencie a štýly akýmkoľvek spôsobom, keďže ide o textové súbory s formátovaním. To podporuje aj analýza, pri ktorej som zistil, že medzi jednotlivými konvenciami je veľká diverzita. Ďalšou výhodou použitia Markdown súborov je vkladanie blokov kódu so syntaxou jazyka alebo rozdelenie konvencií do viacerých logických častí, súborov, ktoré na seba navzájom odkazujú, čo tiež môžeme pozorovať vo viacerých repozitároch z analýzy.

### 2.3.2 Issue šablóny

*Issue* [28] na platforme *GitHub* je spôsob, ako sledovať úlohy, vylepšenia a chyby týkajúce sa projektov. Je to niečo ako e–mail, ale na rozdiel od neho, je tu možnosť aj zdieľať a diskutovať o ňom so zvyškom tímu. Väčšina softvérových projektov má nejaký sledovač chýb. Sledovač na plaforme *GitHub* sa volá *Issues* a má vlastnú sekciu v každom repozitári. Typický Issue obsahuje názov a popis obsahu, farebné štítky, ktoré ho kategorizujú a dá sa podľa nich filtrovať priradenú osobu, zodpovednú zaň a komentáre, ktoré dovoľujú každému s prístupom k repozitáru poskytovať spätnú väzbu. Issue sa zhromažďujú do míľnikov, čo je užitočné pri ich spájaní s rôznymi funkcionalitami a fázami projektu. V Issue je možné upovedomiť iných užívateľov alebo tímy, čo poskytuje flexibilný spôsob, ako zapojiť tých správnych ľudí do efektívneho riešenia problémov. Text Issue je formátovaný pomocou *Markdown* technológie a *GitHub* obsahuje vyhľadávač, ktorý podporuje ich prehľadávanie.

Issue može byť vytvorená na základe kódu z *Pull requestu*, ale aj priamo z komentára iného Issue alebo kontroly *Pull requestu* [26]. Pokiaľ projekt obsahuje projektovú dosku (board) na určovanie priorít práce, môžu byť poznámky z nej prekonvertované na jednotlivé Issue [26]. Repozitáre môžu obsahovať šablóny, ktoré pomáhajú k vytváraniu kvalitnejších Issue a *Pull requestov* [32]. Šablóny k Issue je možné vytvárať v nastaveniach repozitára [25]. GitHub umožňuje nastavenie mena, popisu, obsahu a štítkov v šablóne, ktoré sa po jej zvolení pri vytváraní nového Issue prednastavia [25].

Práve možnosť nastavenia predvoleného obsahu Issue v šablóne v *Markdown* formáte umožňuje ukladanie sprievodcom prispievaním a teda aj programovacích konvencií. Výhodou šablón je okrem použitia syntaxe *Markdown*, ktorej výhody som opisoval pri *Markdown* súboroch, je už predvolený obsah Issue, ktorý stačí iba upraviť a teda je jednoduchšie dodržať konvencie jej štruktúry. Issue šablóny v prieskume programovacích konvencií boli ako druhé najčastejšie použité a obsahovali sprievodcov prispievaním v dvadsiatichpiatich prípadoch z päťdesiat (viď 2.2.1), pre príklad projektu používajúceho Issue šablóny viď *Microsoft PowerToys* (2.2.2).

### 2.3.3 Wiki a externé stránky

Wiki stránky a externé stránky boli ďalším výrazne používaným spôsobom, ako v *GitHub* projekte skladovať sprievodcov prispievaním a tým pádom aj programovacie konvencie. Externé stránky boli využité v ôsmich z päťdesiatich analyzovaných projektov a Wiki stránky v siedmich, pre viac informácií o analýze viď časť 2.2.1. Príkladom projektu využívajúceho Wiki je napríklad *Microsoft PowerToys* (viď 2.2.2), príkladom projektu odkazovaného sa na externé stránky je *React* (viď časť 2.2.2).

Každý *GitHub* projekt môže obsahovať sekciu pre uchovávanie dokumentácie nazývanú Wiki, pomocou nej je možné zdielať detailné informácie o projekte, napríklad ako ho používať, ako je navrhnutý alebo jeho základné princípy [23]. Wiki podporuje, rovnako ako Issue alebo *Markdown* súbory formátovanie v *Markdown* syntaxi [24]. Rozdiel oproti využitiu Markdown súborov je v tom, že Wiki je uložená v separátnom *Git* repozitári [24]. Wiki teda zachováva výhody *Markdown* súborov, k čomu pridáva výhodu samostatného repozitára, určeného na dokumentovanie projektu, na ktorý je zvyčajne odkazované z primárneho repozitára s kódom.

Externými stránkami sú myslené webové stránky, na ktoré projekt odkazuje a slúžia ako sprievodcovia prispievaním do projektu a teda môžu obsahovať programovacie konvencie. Výhodou využitia webových stránok je určenie vlastnej dostupnosti a teda môžu byť dostupné aj užívateľom, ktorí nemajú prístup k samotnému projektu. Takisto oproti iným riešeniam, založených na *Markdown* syntaxi, poskytujú viac možností vo formátovaní a obsahu. Externými stránkami v analýze boli zväčša detailné dokumentácie k projektom, určené ako pre koncových užívateľov, tak pre vývojárov.

### 2.3.4 EditorConfig

*EditorConfig* [41] pomáha udržiavať konzistentný štýl kódu pre viacerých vývojárov pracujúcich na rovnakom projekte naprieč rôznymi editormi a *IDE*. *EditorConfig* projekt pozostáva z formátu súboru, slúžiaceho na definovanie programovacích štýlov a zbierky doplnkov do rôznych textových editorov, ktoré umožňujú čítať formát súboru a dodržiavať v ňom definované štýly. Súbory *EditorConfigu* sú jednoducho čitateľné a fungujú dobre s verznými systémami.

V prípade *EditorConfigu* sa oproti prechádzajúcim technológiám neukladajú programovacie konvencie ako formátovaný text, ale ako *EditorConfig* súbor, vďaka ktorému vedia editory automaticky upraviť modifikovaný súbor tak, aby spĺňal konvencie zadané v *EditorConfig* súbore. V analýze projektov (viď časť 2.2.1) bol najčastejším nástrojom používaným k uchovávaniu programovacích konvencií, pokiaľ nepočítame aj formátované texty. Využitý bol v osemnástich z päťdesiat projektov, pre projekt využívajúci *EditorConfig* viď *Microsoft PowerToys* (2.2.2). Ďalšie technológie umožňujúce automatickú kontrolu obsahuje časť 2.3.5.

#### Štandard

*EditorConfig* súbory sú štandardne pomenované `.editorconfig` a sú vyhľadávané pri otvorení súboru pomocou *IDE* alebo textového editoru, ktorý má *EditorConfig* vstavaný alebo je do neho nainštalované *EditorConfig* rozšírenie [41]. *EditorConfig* súbor je vyhľadávaný v adresári, v ktorom sa nachádza modifikovaný súbor s editorom a v každom nadradenom adresári, až pokiaľ sa nenájde koreňový súbor obsahujúci pravidlo `root=true` [41]. Súbory s konvenciami sú čítané zhora nadol, pričom najnovšie nájdené pravidlá majú prioritu. Pri viacerých súboroch majú prioritu pravidlá bližšie k modifikovanému súboru [41].

*EditorConfig* súbory obsahujú páry kľúča a hodnoty (v závislosti od zdroja nazývané pravidlá alebo vlastnosti), ktoré charakterizujú konvencie. Tieto páry sú umiestnené pod sekciou, ktorá určuje, na aké súbory sa konvencie aplikujú [12]. Pokiaľ kľúč nie je rozpoznaný editorom alebo rozšírením, tak je ignorovaný, čo je užitočné pri vytváraní vlastných dodatkov. Medzi široko rozšírené, a teda takmer vždy podporované, kľúče konvencií patrí štýl a veľkosť odsadenia, šírka tabu, znak konca riadku, kódovanie súboru, orezávanie koncových medzier a vloženie nového riadka na koniec súboru [11]. Súbory *EditorConfigu* sú založené

na *INI* formáte a mali byť kódované UTF–8, pre viac informácií o formáte súborov viď špecifikáciu *EditorConfigu*[10].

#### Rozšírenia

Ako je spomenuté v prechádzajúcej časti o štandarde, pokiaľ kľúč *EditorConfig* pravidla nie je rozpoznaný, tak je ignorovaný, čo umožňuje jednoduché vytváranie vlastných rozšírení pravidiel. Podporu rozšírení si rieši každé *IDE* alebo plugin samostatne. Vďaka tomu existuje do *EditorConfigu* viacero rozšírení.

Rozšírenie pravidiel existuje pre napríklad pre *.NET*. Rozdelené je do štyroch kategórií: pravidlá ovplyvňujúce spôsob použitia rôznych jazykových konštruktov, pravidlá identifikujúce zbytočný kód, pravidlá formátovania a pravidlá pomenovania [55]. Niektorým pravidlám je možné nastaviť úroveň závažnosti[11]. Rozšírenie je podporované viacerými *IDE*, medzi ktorými je aj Visual Studio (viď 3.3), ale pravidlá je možné vynucovať aj počas kompilácie zdrojového kódu [55].

Ďalšie rozšírenia pre viaceré jazyky podporujú produkty od firmy *JetBrains*. Napríklad v prostredí *IntelliJ IDEA* je možné po pridaní pluginu používať okrem základných pravidiel aj rozšírené, začínajú prefixom `ij_` [43]. Rozšírenie *ReSharper* pridáva ďalšie pravidlá pre *.NET*, umožňuje exportovať svoje nastavenia do *EditorConfig* súboru a umožňuje ho upravovať interaktívne [44].

### 2.3.5 Iné technológie

Časť obsahujúca iné technológie, používané na správu programovacích konvencií, ktoré v analýze *GitHub* projektov (viď časť 2.2.1) boli zastúpené sporadicky alebo majú významné postavenie medzi technológiami, zaisťujúcimi ich uchovávanie a dodržiavanie. Technológie sú špecializované na určité jazyky a vedia konvencie kontrolovať automaticky.

#### Linter

*Linter* je malý program, ktorý kontroluje kód kvôli štylistickým alebo programovacím chybám [82]. *Linter* rámce sú k dispozícii pre väčšinu syntaxí programovacích jazykov [82]. Prvý *Linter* rámec vznikol v roku 1978, pre analýzu kódu v programovacom jazyku C [77]. Cieľom pôvodného nástroja bolo analyzovať zdrojový kód a nájsť optimalizácie kompilátora, k čomu sa v priebehu času začali pridávať ďalšie typy kontrol a overovaní [77]. Vo všeobecnosti Lintre poskytujú viac typov kontrol, medzi ktoré patrí využitie statickej analýzy, kontrola bezpečnosti alebo programovacích konvencií a formátovania [77]. Nie všetky *Lintre* podporujú všetky typy kontrol a zväčša sa zameriavajú iba na nejakú ich časť, jednotlivé kontroly sú rozdelené do pravidiel, ktoré môžu byť zapínané a vypínané podľa potreby.

*ESLint* je *open source JavaScript Linter*. *JavaScript* je vďaka jeho vlastnostiam náchylný k vytváraniu chýb, ktoré sa typicky hľadajú spustením kódu [68]. *ESLint* a iné *Lintery* umožňujú vývojárom odhaliť problémy s kódom bez jeho spustenia [68]. *Cpplint* umožňuje kontrolu súborov voči C++ štýlom od spoločnosti *Google* (viď 2.2.3) [49].

[10] https://editorconfig-specification.readthedocs.io/#file-format

[11] https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/configuration-options#severity-level

### Prettier

*Prettier* [71] je dogmatický program, slúžiaci na formátovanie kódu podporujúci viacero jazykov, medzi ktoré patrí *JavaScript*, *HTML*, *CSS*, *JSON* a ďalšie. Pre prehľad všetkých podporovaných jazykov viď oficiálnu dokumentáciu[12]. Prettier odstráni všetko originálne formátovanie a zabezpečí, že celý výstupný kód bude v konzistentnom štýle. Je možné ho inštalovať lokálne do projektu ako aj volať priamo z kódu, pričom umožňuje nastavenie niekoľkých druhov formátovania, ako aj ignorovanie určitých súborov.

Oproti *Linterom* je určený k formátovaniu, pričom *Linter* je určený predovšetkým k zachytávaniu chýb [70]. Formátovacie pravidlá z *Linteru*, ako napríklad maximálna dĺžka riadku, nie sú vôbec pri použití potrebné, keďže Prettier celé formátovanie zmaže a vytvorí nové, konzistentné [70]. S pravidlami v *Linteroch*, ktoré zvyšujú kvalitu kódu, ako napríklad žiadne nevyužité premenné, Prettier nemá nič spoločné a slúžia prevažne k odhaľovaniu chýb [70].

### Mypy

*Mypy* [81] je dobrovoľne použiteľný nástroj kontrolujúci statické typovanie pre jazyk *Python*, ktorého cieľom je spojiť výhody dynamického a statického programovania. Kombinuje silu výrazov a pohodlie *Pythonu* so systémom pre kontrolu typov a s kontrolou typov pri kompilácii [81]. Funguje pre štandardné *Python* programy, ktoré spúšťa pomocou ľubovoľnej *Python VM* s takmer žiadnou réžiou [81].

### Black

*Black* [48] je nástroj, formátujúci kód v jazyku *Python*. Jeho používaním užívateľ prichádza o možnosť vlastnoručného formátovania kódu, avšak šetrí čas a vytvára deterministicky naformátovaný kód, ktorý je rovnaký v každom projekte [48]. Užívateľ nástroja sa môže viac sústrediť na obsah kódu ako na jeho formátovanie, kontrolovanie kódu je rýchlejšie a produkuje menšie zmeny [48].

### ClangFormat

*ClangFormat* [78] popisuje množinu nástrojov, formátujúcich programovacie jazyky, ktoré sú postavené na *LibFormat*[13]. Podporuje prácu viacerými spôsobmi, vrátane samostatnej aplikácie alebo integrácie v rôznych editoroch a *IDE* [78]. Samostatná aplikácia umožňuje formátovanie jazykov C, C++, C#, Java a ďalších [78].

### Flake8

*Flake8* [6] je nástroj spustiteľný cez príkazový riadok, ktorý vynucuje konzistentný štýl v *Python* projektoch. V základe obsahuje viacero kontrol konzistentnosti, avšak podporuje aj rozšírenia tretích strán [6].

[12] https://prettier.io/docs/en/index.html
[13] https://clang.llvm.org/docs/LibFormat.html

# Kapitola 3

# Návrh

Pred vytvorením návrhu systému je dôležité zanalyzovať požiadavky, ktoré bude daný systém plniť. Kapitola nadväzuje na predchádzajúcu, ktorá analyzuje programovacie konvencie používané v projektoch (viď 2). Pomocou analýz z predchádzajúcej kapitoly určuje požiadavky na vytváraný systém a následne systém pomocou vhodných technológií a architektúry navrhuje, spĺňajúc dané požiadavky.

Prvá časť kapitoly identifikuje požiadavky na vytváraný systém na základe analýz z predchádzajúcej kapitoly (viď 3.1). Porovnáva analyzované technológie a konvencie, ktorých výsledky zhŕňa do scenárov použitia a špecifikovaných požiadaviek. Ďalšia časť navrhuje architektúru systému (viď 3.2). Systém rozdeľuje do vrstiev a vrstvy následne do menších celkov. Navrhuje ich úlohu, vhodné technológie a popisuje ich význam. Posledná časť vykresľuje ako bol systém implementovaný, aké problémy pri implementácii vznikli a ako bol výsledný systém nasadený (viď 3.3).

## 3.1 Požiadavky na vytváraný systém

Časť práce, definujúca požiadavky na navrhovaný systém, spravujúci programovacie konvencie. V prvej časti porovnáva funkcie existujúcich systémov a riešení spravujúcich programovacie konvencie (viď 3.1.1), pričom zohľadňuje získané dáta z analýz z prechádzajúcej kapitoly. Následne uvažuje s rôznymi scenármi použitia (viď 3.1.2), pričom vychádza najmä zo spôsobov, ako sa obdobné technológie v analýze najčastejšie používajú. V poslednej časti definuje priame požiadavky na systém, vychádzajúce primárne zo scenárov použitia a analýzy existujúcich technológií a *open source* projektov (viď 3.1.3).

### 3.1.1 Porovnanie funkcií existujúcich technológií

Pred spísaním požiadaviek na systém je vhodné porovnať funkcie existujúcich technológií spracujúcich programovacie konvencie, ktoré boli analyzované v časti 2.3 a pracovať s dátami získanými z analýzy *open source* projektov v časti 2.2.2. V tejto časti si kladiem za cieľ porovnať funkcie a možnosti jednotlivých technológií a zamyslieť sa nad ich výhodami a nevýhodami z viacerých pohľadov. Zameriavam sa na porovnanie typov spravovaných konvencií, spôsoby prístupu a uloženia a na históriu zmien. Porovnanie funkcií existujúcich technológií je následne použité pri vytváraní scenárov použitia (viď 3.1.2) a špecifikácii požiadaviek (viď 3.1.3).

### Porovnanie typov konvencií

V prvom rade je vhodné porovnať, aké konvencie je možné pomocou ktorých technológií spravovať, a teda aké možnosti nám rôzne technológie ponúkajú. Technológie je možné rozdeliť podľa konvencií na dva segmenty. Prvým sú technológie. umožňujúce spracovanie akýchkoľvek konvencií, keďže sú uložené v ľudskom jazyku pomocou formátovaného textu, čoho následkom je nemožnosť automatizovanej kontroly. Druhým segmentom sú technológie, spracúvajúce konvencie vo forme jazyka, ktorému technológie rozumejú, sú schopné ho spracovať a konvencie kontrolovať, avšak jazyk je obmedzenejší ako pri prvom type. Medzi technológiami sa nenachádzala žiadna komplexnejšia, ktorá by ponúkala výhody z oboch druhov, teda možnosť uloženia akýchkoľvek konvencií, ale zároveň ich umožnila automaticky kontrolovať.

*Markdown* súbory, spolu s *Issue* šablónami a Wiki stránkami (viď 2.3) podporujú správu akýchkoľvek konvencií, keďže ich ukladajú vo forme formátovaného textu. Tieto technológie boli v analýze programovacích konvencií v *open source* projektoch zastúpené výrazne častejšie (viď 2.2.1). Formátovaný text umožňuje pracovať s akýmikoľvek konvenciami, ľubovoľne ich zoradiť, združiť a formulovať, vďaka čomu je ich použitie všeobecnejšie, dajú sa použiť pre akýkoľvek jazyk, akokoľvek užívateľ vyžaduje. Pri analýze sa mi podarilo zistiť, že v použití programovacích konvencií medzi projektami je veľká diverzita a teda využitie uloženia vo forme textu je výhodou. Výhodou je aj možnosť pracovať so zložitejšími konvenciami, ktoré by aj tak nebolo možné jednoduchým spôsobom kontrolovať automatizovane. Nevýhodou uloženia vo forme textu je teda nemožnosť kontrolovať konvencie priamo technológiou, čo by bolo aj zložité, pri voľnom spôsobe ich zápisu. Technológie dávajú užívateľom možnosť formulovať text akokoľvek potrebujú, v podstate sú to technológie určené na správu textu. v ktorých sa užívatelia rozhodli spravovať konvencie.

Technológie ako *EditorConfig*, *Linter*, *Prettier* a ďalšie (viď 2.3) umožňujú kontrolu konvencií automatizovane a ohraničujú, aké konvencie je možné v nich spravovať. V analýze programovacích konvencií boli zastúpené v menšej miere (viď 2.2.1). Zväčša sa obmedzujú na kontrolu konkrétnych konvencií v určitých jazykoch, v ktorých sa sústredia na jednoduchšie, automaticky kontrolovateľné prvky (ako je napríklad formátovanie a mená premenných, funkcií a podobne) alebo umožňujú kontrolu nižšie úrovňových konvencií, spoločných pre viaceré jazyky (napríklad maximálnej dĺžky riadku textu v súbore). Pridávanie vlastných konvencií väčšina technológií neumožňuje, *Editorconfig* dovoľuje ich pridávanie do konfiguračných súborov, avšak je nutné ich kontrolu naprogramovať. Nastavenie kontrolovaných konvencií majú rozličné formy, v závislosti od technológie. Niektoré technológie nie je možné nijako nastavovať a kontrolujú sadu predpripravených konvencií, ale pri iných je možné vyberať, ktoré konvencie budú kontrolované. *Editorconfig* umožňuje okrem výberu konvencií aj výber priečinkov a prípon súborov, na ktorých budú konvencie kontrolované. Od rozličných foriem nastavení sa odvíjajú aj možnosti radenia a združovania konvencií. Technológie, ktoré dovoľujú vyberať konvencie použité na kontrolu sú zväčša logicky usporiadané. V *Editorconfigu* sú konvencie združené podľa prípon súborov, v ktorých je možné jednotlivé konvencie vkladať ľubovolne.

### Porovnanie uloženia konvencií

Ako je spomínané v prechádzajúcej časti, nie všetky technológie kontrolujúce konvencie ukladajú nastavenia. Technológie, ktoré ich neukladajú sa buď nijak nenastavujú a kontrolujú predvytvorenú sadu konvencií alebo sa musia nastavovať vždy znovu, čo nie je užívateľsky prívetivé. Rôzne technológie ukladajú konvencie rôznymi spôsobmi. Technológie

spravujúce a kontrolujúce konvencie lokálne u užívateľa zväčša používajú lokálne databázy alebo konfiguračné súbory. Nevýhodou lokálnych nastavení je ich zdieľanie. Konfiguračné súbory, ktoré používa napríklad *Editorconfig*, sú často zdieľané pomocou verzných systémov, ale nastavenia z databáz musia byť exportované a znovu importované u iného užívateľa.

Technológie pracujúce s formátovaným textom sa zvyčajne ukladajú ako *Markdown* súbory, ktoré sú uložené v *Git* repozitári, čo umožňuje prístup všetkým jeho užívateľom. *Issue* šablóny a *GitHub* Wiki stránky sú dostupné v rámci GitHub repozitára. Programovacie konvencie uložené na externých stránkach, majú rôzne spôsoby uloženia, zvolené danou stránkou alebo službou.

#### Porovnanie obmedzenia prístupu ku konvenciám

Medzi analyzovanými technológiami sa nachádzali také, ktoré vedeli obmedziť prístup ku konvenciám určitým ľudom, ale aj také, ktoré boli verejné a mohol si ich prehliadnuť každý. *Open source* projekty poskytujú používané konvencie typicky verejne, aby sa s nimi mohli prispievajúci užívatelia zoznámiť.

Technológie spravujúce a kontrolujúce technológie lokálne, zväčša poskytujú prístup ku konvenciám iba ich užívateľovi. Uložené sú v lokálnych databázach a je nutné ich pre získanie prístupu iného užívateľa exportovať. *Editorconfig* pracujúci s konfiguračnými súbormi síce umožňuje kontrolovať zmeny lokálne, ale konfiguračné súbory sa zväčša ukladajú do verzného systému, a teda ku konvenciám majú prístup užívatelia, ktorí majú prístup k repozitáru. Ku konvenciám uloženým v *Markdown* súboroch, *Issue* šablónach a *GitHub* Wiki stránkach majú taktiež prístup užívatelia repozitára. Ak je repozitár verejný, aj prístup ku konvenciám je verejný. Pri konvenciách uložených na externých stránkach záleží od konkrétnej služby. Stránky zhromažďujúce programovacie konvencie sú zväčša verejne dostupné všetkým, ale môžu byť dostupné iba vybraným užívateľom po prihlásení.

#### Porovnanie zmien v konvenciách a histórie

Pre technológiu je výhodou, ak zaznamenáva zmeny v konvenciách, históriu ich úprav a umožňuje užívateľom, aby sa ku konvenciám vyjadrovali. Vďaka tomu je možné spätne zisťovať dôvody jednotlivých zmien a zaznamenávať si, ktoré konvencie už boli aplikované na kód, ako aj viesť diskusiu o budúcich zmenách v konvenciách. História zmien pomáha aj k lepšiemu pochopeniu konvencií pre nových vývojárov.

Konvencie v *Markdown* súboroch ukladajú svoju históriu pomocou verzného systému, rovnako je možné ukladať históriu aj konfiguračných súborov rôznych technológií, umožňujúcich automatickú kontrolu. Lokálne technológie pracujúce s databázou, určené pre jedného užívateľa často neumožňujú sledovať históriu zmien. Prikladám to tomu, že výhody sledovania zmien v lokálnych technológiách strácajú význam, keďže sú primárne určené pre jedného užívateľa. Technológie spravujúce konvencie na externých stránkach implementujú históriu zmien a komentovanie v závislosti na type služby.

### 3.1.2 Uvažované scenáre použitia

Na základe porovnaní funkcií existujúcich technológií spravujúcich programovacie konvencie (viď 3.1.1) a na základe analýzy *open source* projektov v časti 2.2.2 je možné definovať niekoľko základných scenárov použitia systému. Scenáre budú následne využité v časti 3.1.3 pre formálne definovanie požiadaviek na systém.

#### Správa konvencii

Systém slúži ako databáza konvencií pre projekt v ľubovoľnom jazyku, pričom na jednom projekte môže pracovať viacero vývojárov. Konvencie sa zadávajú do systému pomocou formátovaného textu, čo umožňuje spísanie rôznorodých konvencií. Systém spravuje konvencie a umožňuje sledovať ich históriu a je schopný vytvoriť konvencie pre novovznikajúce projekty, ako aj zistiť aktuálne používané. Pri každej konvencii je možné komunikovať v diskusnom vlákne pre riešenie možných zmien a ujasnenie si nejasností. Celý systém funguje ako báza znalostí konvencií v projekte pre nových členov tímu a ako spôsob zoznamovania sa s novými a zmenenými konvenciami.

#### Kontrola a generovanie konvencií

V systéme je možné ku každej konvencii pridať formálny zápis, pomocou ktorého je možné konvenciu na projekte automatizovane kontrolovať. Automatizovaná kontrola konvencií vytvára súhrnnú správu dodržania konvencií v projekte, tak ako je uvedené v zadaní práce. Formálne zápisy je možné generovať z aktuálneho stavu projektu. Kontrolu a generovanie je možné rozšíriť o vlastné definované pravidlá.

#### Open source systém

Programovacie konvencie k projektu sú voľne prístupné, keďže systém je primárne určený pre *open source* projekty. Užívateľ sa do systému prihlasuje pomocou *GitHub* účtu. Kód systému je verejne dostupný pod *open source* licenciou, čo umožňuje systém upravovať a pridávať vlastné kontroly konvencií.

### 3.1.3 Špecifikácia požiadaviek

Pred výberom implementačných technológií, špecifikáciou architektúry a návrhom jednotlivých modulov systému, je vhodné formálne špecifikovať požiadavky, ktoré by mal cieľový systém spĺňať. Požiadavky majú byť jednoznačné, testovateľné, uskutočniteľné, merateľné a nezávislé na implementácii, pričom sú definované dostatočne detailne pre návrh systému [87]. Požiadavky boli špecifikované s ohľadom na uvažované scenáre použitia a analýzu existujúcich technológií. Následne som pri špecifikácii vychádzal zo zadania práce, analýzy a uskutočnených konzultácii.

#### Požiadavky na výsledný systém

Požiadavky na výsledný systém definujú predovšetkým požiadavky, ktoré súvisia s vývojom a nasadením systému, ako je napríklad počítačové a programové vybavenie, prenositeľnosť a bezpečnosť [47]. V prípade vytváraného systému, ide primárne o definovanie druhu systému, licencie a spôsobu nasadenia.

- Systém je vytvorený ako webová aplikácia, ktorá komunikuje s užívateľom pomocou webového rozhrania. Aplikácia taktiež komunikuje s databázou, v ktorej uchováva perzistentné dáta a s aplikačným programovacím rozhraním služby *GitHub*.
- Systém je zverejnený pod *open source* licenciou v službe *GitHub* a obsahuje všetky potrebné náležitosti, umožňujúce vlastné nasadenie a ďalší vývoj komunitou. Systém je zverejnený v celom rozsahu a teda je možné ho stiahnuť, ľubovoľne upraviť a nasadiť

užívateľom. K systému je vytvorená dokumentácia, obsahujúca návod ako systém nasadiť.

- Inštancia systému bude nasadená a dostupná pri jeho testovaní a následnom hodnotení (pre nasadenie inštancie viď 3.3.3).

#### Funkcionálne požiadavky

Ide o základný typ požiadaviek, ktorý určuje, čo má vyvíjaný systém robiť a aké sú jeho funkcionality [47]. Pri špecifikácii som sa zameral hlavne na tento typ požiadaviek, keďže pre navrhovaný systém je primárne definovať jeho funkcionality.

- Do systému sa užívatelia prihlasujú pomocou užívateľských účtov v službe *GitHub*. Následne užívateľ pristupuje k repozitárom zo služby *GitHub*, v ktorých spravuje programovacie konvencie. V systéme užívateľ môže pristupovať ako k verejným, tak ku privátnym repozitárom, v ktorých má administrátorské práva alebo práva na zápis.
- Konvencie môže užívateľ v systéme uchovávať vo forme formátovaného textu. Okrem toho, každá konvencia obsahuje vlastné diskusné vlákno a históriu zmien.
- Každá konvencia môže obsahovať aj formálny zápis v *EditorConfig* formáte. Systém umožňuje automatizovanú kontrolu konvencií v projekte, na základe ich formálneho zápisu, z ktorého vypíše súhrnnú správu. Systém taktiež umožňuje vygenerovať formálny zápis konvencií zo zdrojových súborov, pričom systém zistí pravdepodobnosť jej použitia a konečný výber nechá na užívateľovi. Systém dovoľuje sledovať históriu súhrnných správ a aj históriu zmien formálneho textu.
- Generovanie formálneho zápisu konvencií z projektu, ako aj ich kontrolu v projekte, je možné rozšíriť o vlastne definované formálne pravidlá. Vlastne pravidlá je možné vytvárať pomocou nových tried, implementujúc preddefinované rozhrania v kóde systému. Využitím rozhraní, systém zaručuje nepovinné implementovanie generovania formálneho zápisu a aj kontroly na projekte. K vytváraniu vlastných pravidiel je vytvorený návod, pre užívateľov systém obsahuje zoznam už implementovaných pravidiel s ich popisom.

## 3.2 Architektúra systému

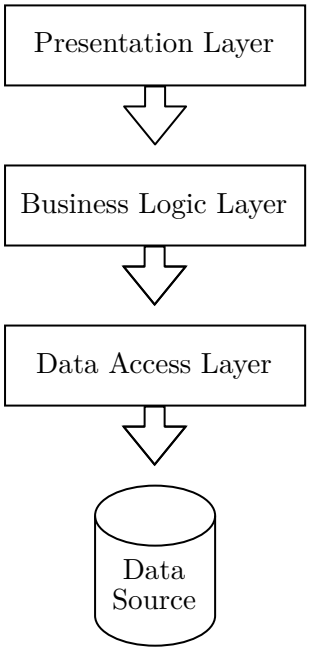
Časť práce popisujúca architektúru navrhovaného systému na správu programovacích konvencií. Navrhnutá architektúra vyplýva zo špecifikácie požiadaviek (viď 3.1.3) a z uvažovaných scenárov použitia (viď 3.1.2), pričom cieli na využitie moderných technológií a prístupu k vývoju.

Systém bude serverová aplikácia používajúca štandardný klient–server prístup v *.NET* prostredí, ktorá bude postavená na technológii *Blazor* (viď 3.2.2), pomocou server hosting modelu. Pre *.NET* som sa rozhodol kvôli mojej osobnej znalosti prostredia a preto, že veľmi rýchlo narastá popularita technológie Blazor, čo je možné pozorovať na jej použitiach[1]. *Blazor* som sa rozhodol použiť ako kľúčovú pre vývoj aplikačnej vrstvy.

[1] https://trends.builtwith.com/framework/Microsoft-Blazor

### 3.2.1 Trojvrstvová architektúra

Systémové riešenie je rozdelené do troch vrstiev, projektov. Trojvrstvová architektúra rozdeľuje aplikácie na tri výpočtové vrstvy, je prevládajúcou softvérovou architektúrou pre tradičné klient–server aplikácie [42]. Jednotlivé vrstvy sú od seba logicky oddelené. Hlavnou výhodou trojvrstvovej architektúry je jednoduché inovovanie alebo nahradzovanie vrstvy, bez vplyvu na ostatné [72].




Obr. 3.1: Trojvrstvová architektúra napojená na databázu použitá v projekte.

Najvyššia, prezentačná vrstva je užívateľským a komunikačným rozhraním systému, pomocou ktorého koncoví užívatelia intereagujú s aplikáciou [42]. V systéme má za cieľ zobrazovať užívateľovi informácie, zbierať ich od neho a posielať ich do nižšej, logickej vrstvy. Tá obsahuje služby a v prípade návrhu tohto systému Viewmodely (viď 3.2.3), na ktoré sa prezentačná vrstva napája.

V strede sa nachádza logická vrstva, tiež nazývaná aplikačná, biznisová alebo stredná vrstva. Vrstva riadi základné funkcionality systému spracovaním dát [72]. Vo vrstve sa informácie zhromaždené z prezentačnej vrstvy spracúvajú pomocou špecifikovaných pravidiel [42]. Vrstva sa v prípade potreby napája na databázovú, z ktorej čerpá perzistentné dáta, alebo na iné služby, s ktorými systém komunikuje. Výstupom vrstvy sú služby a Viewmodely, na ktoré sa napája prezentačná vrstva.

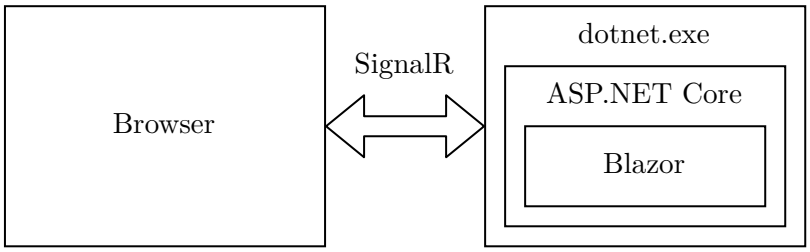
Najnižšie je položená databázová vrstva. Slúži na oddelenie databázovej logiky od logiky aplikácie [72]. V prípade tohto systému komunikuje s databázou pomocou *ORM* a vytvára rozhrania, na ktoré sa v prípade potreby komunikácie s databázou napája logická vrstva. V trojvrstvovej aplikácii musí ísť celá komunikácia z databázovej do prezentačnej vrstvy cez logickú, dátová vrstva s prezentačnou nemôže komunikovať priamo [42].

### 3.2.2 Prezentačná vrstva

Ako je spomínané v úvode k trojvrstvovej architektúre, prezentačná vrstva slúži ku komunikácii konečného užívateľa s aplikáciou. K tomu je v systéme využitá technológia Blazor. Okrem užívateľského rozhrania vrstva zabezpečuje aj autorizáciu užívateľa a spravuje Dependency Injection (*DI*) kontajner, do ktorého si pridáva služby z logickej vrstvy.

#### Blazor

Blazor je *open source* vývojársky rámec pre vývoj jedno stránkových aplikácií, ktorý vznikol z kombinácie slov Browser a Razor (*.NET HTML* modul generujúci zobrazenia) [65]. Blazor ponúka dva modely hostingu. WebAssembly model, ktorý beží na klientskej strane v prehliadači pomocou *WebAssembly*. Blazor stránka a jej závislosti sa pri *WebAssembly* inicializácii stiahnu do prehliadača klienta, ktorý následne spúšťa kód [52]. Druhým modelom je Server model, pri ktorom je aplikácia vykonávaná na serveri.




Obr. 3.2: Blazor Server model použitý v aplikácii. Aplikácia sa vykonáva na serveri v ASP.NET Core. Užívateľské rozhranie, udalosti a *JavaScript* volania sa spracúvajú pomocou SignalR spojenia. Obrázok upravený z *Microsoft* dokumentácie [52].

Rozhodol som sa použiť Server hosting model, pričom využitie každého z modelov má svoje výhody aj nevýhody. Server hosting model neodosiela celú stránku ku klientovi, takže sa načítava rýchlejšie a pre spustenie aplikácie nie je v prehliadači nutná podpora *WebAssembly* [52]. Všetky výpočty bežia na serveri, takže nie je nutné výpočty validovať a postačuje riešiť validáciu z užívateľského rozhrania. Pri ASP.NET core (*open source* web rámci, ktorý využíva Blazor Server hosting model) som rozhodol pre verziu 5.0, keďže v čase návrhu je najnovšom stabilnou.

#### Užívateľské rozhranie

Blazor využíva k vytváraniu stránok Razor Pages. Ide o syntax kombinujúcu *HTML* kód s kódom v jazyku C# [61]. V jednom súbore je možné striedať oba jazyky [61].

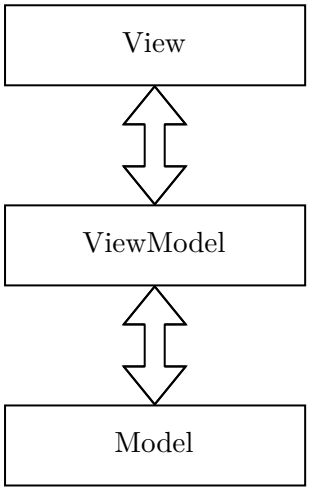
Pri tvorbe užívateľských rozhraní, je možné využívať štandardné *HTML* komponenty a *CSS*. Avšak pre vytvorenie stránky s kvalitnejším a responzívnejším dizajnom som sa rozhodol využiť *open source* knižnicu *Radzen*[2] komponent pre Blazor. Existuje viacero knižných, ktoré pridávajú vlastné komponenty, ako napríklad *Blazorise*, *BlazorStrap* alebo *MudBlazor*. Jedine *Radzen* pridáva komponent *HTML* editoru, ktorý je vhodné použiť pri práci s formátovaným textom konvencií. Okrem knižnice *Radzen* som sa rozhodol využiť *Bootstrap*[3], *open source CSS* rámec, ktorý je už implicitne vložený v základnej šablóne Bla-

[2] https://blazor.radzen.com
[3] https://getbootstrap.com

zor aplikácie. Ikony a obrázky som sa rozhodol čerpať z Google Fonts[4] a z Material Design Icons[5].

### Architektonický vzor Model–View–ViewModel

Model–View–ViewModel (*MVVM*) je softvérový architektonický vzor, ktorý uľahčuje oddelenie vývoja *GUI* (View) od vývoja biznis logiky. Pomáha mať lepší *SoC*, meniť jednotlivé View za iné bez zmeny ViewModelu, navrhovať dizajn aplikácie bez nutnosti siahať do logiky aplikácie a naopak [19]. Využívaný je prevažne s technológiou *WPF* [19].




Obr. 3.3: Architektonický vzor Model–View–ViewModel skladajúci sa z troch častí: View, ViewModel a Model. View si kladie za úlohu reprezentovať dáta koncovému užívateľovi, definuje štruktúru, rozloženie a vzhľad užívateľského rozhrania. ViewModel je prostredníkom medzi View a Modelom, udržiava v sebe aktuálny stav systému a dostupné operácie (napríklad odkaz aktuálne zvolený repozitár). Model predstavuje dátovú časť programu (napríklad objekt repozitára, na ktorý sa odkazuje ViewModel).

Kvôli značným výhodám v separácii jednotlivých celkov systému som sa rozhodol aplikáciu navrhnúť v súlade s architektonickým vzorom Model–View–ViewModel. Prezenčná časť aplikácie obsahuje jednotlivé pohľady (View), ktoré sú v prípade technológie Blazor reprezentované pomocou Razor Pages. Pohľady obsahujú iba užívateľské rozhranie, bez vnútornej logiky. Pre získavanie stavu aplikácie, dát a spúšťanie operácii sa odkazujú na ViewModely, ktoré sú uložené spolu s Modelmi v logickej vrstve.

Užívateľské rozhranie som rozdelil do niekoľkých pohľadov/stránok, s ohľadom na požiadavky na systém:

- Úvodná stránka, domov aplikácie
- Menu celej stránky, hlavička a pätka
- Zobrazenie dostupných reporitárov
- Zobrazenie zvoleného repozitára, konvencií v ňom

[4] https://fonts.google.com/icons
[5] https://materialdesignicons.com

- Zobrazenie zvolenej konvencie a dát v nej:
  - Zobrazenie aktuálneho formátovaného/formálneho textu
  - Zobrazenie komentárov
  - Pridanie komentára
- Pridanie/mazanie a úprava mena konvencie
- Importovanie formálnych pravidiel z repozitára
- Kontrola formálnych pravidiel na repozitári
- Zobrazenie histórie formátovaného/formálneho textu
- Upravenie formátovaného/formálneho textu konvencie
- Zobrazenie implementovaných formálnych pravidiel
- Neexistujúca, nenájdená alebo chybová stránka

### Autorizácia

Funkcionálne požiadavky požadujú prihlasovanie užívateľov do systému pomocou služby *GitHub*. Okrem toho je však vhodné, aby po prihlásení systém naďalej vedel komunikovať s užívateľovým účtom v službe *GitHub* a pracovať s dátami užívateľa v nej. To je vhodné docieliť pomocou protokolu OAuth, ktorý *GitHub* podporuje a umožní následné spytovanie nad *API*.

OAuth je autorizačný protokol alebo rámec. Je otvoreným štandardom popisujúcim ako môžu nesúvisiace servery a služby bezpečne povoliť overený prístup bez zdieľania svojich počiatočných prihlasovacích údajov [39]. Ten spôsob overenia je známy ako treťou stranou zabezpečená delegovaná autorizácia [39].

Protokol bude použitý k prístupu do vytvoreného systému pomocou účtu v službe *GitHub*. ASP.NET core, nad ktorým beží Blazor, obsahuje už vstavané možnosti ako s protokolom pracovať, ktoré je následne možné rozšíriť o *NuGet* balíček obsahujúci implementáciu jednotlivých *OAuth* poskytovateľov[6]. Samotná autorizácia bude prebiehať na prezentačnej vrstve, keďže obsahuje vhodné prostredie na jej realizáciu, ale prístupový token získaný autorizáciou bude uložený na logickej vrstve a ku *GitHub API* sa bude pristupovať z nej.

### Vkladanie závislostí

ASP.NET Core podporuje návrhový vzor pre vkladanie závislostí, Dependency Injection (*DI*), čo je technika na dosiahnutie Inversion of Control (*IoC*) medzi triedami a ich závislosťami [56]. Výhodou je registrácia závislosti v kontajneri služieb, služby môžu byť následne vkladané do konštruktov tried, pričom zodpovednosť za konštrukciu a dekonštrukciu tried preberá framework [57].

Existujú aj rôzne iné balíčky, zaisťujúce Dependency Injection, avšak pre jednoduchosť a natívne prepojenie som sa rozhodol pre použitie Dependency Injection z ASP.NET Core. Priamo v šablóne Blazor aplikácie je kontajner zaregistrovaný Startup súbore. V logickej a dátovej vrstve je možné vytvoriť poskytovateľa služieb, ktorý služby z danej vrstvy pridá do kontajnera. Následne si môže vrstva pridať do kontajnera služby z využívanej nižšie postavenej vrstvy pomocou poskytovateľa služieb nižšej vrstvy.

[6] https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers

Pri registrácii služby do kontajnera, je možné vybrať jednu z nasledujúcich druhov životností:

- Prechodná (Transient) životnosť, s ktorou sa vytvára nová inštancia služba vždy, keď je o ňu požiadané [57].
- Rozsahová (Scoped) životnosť, s ktorou sa pri webových aplikáciach vytvára jedna inštancia pre jedno pripojenie klienta [57]. Ten vďaka tomu dostáva po požiadaní vždy rovnakú inštanciu.
- Jedináčik (Singleton), ktorý sa vytvára jeden a jeho inštancia je vždy pri požiadaní vrátená [57].

Služby Radzenu pre fungovanie upozornení, dialógov alebo menu sú zaregistrované ako Scoped.

### 3.2.3 Logická vrstva

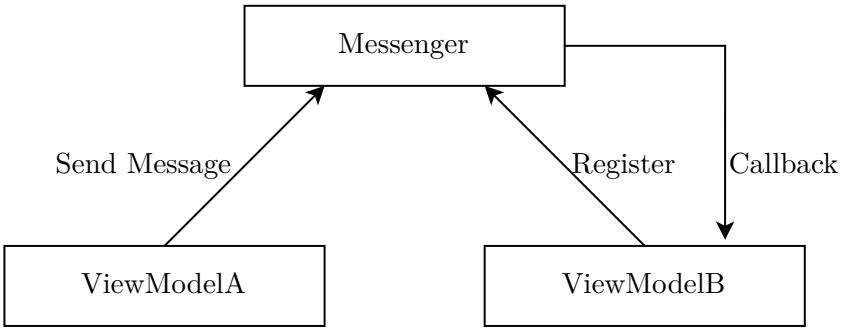
Logická vrstva obsahuje základné funkcionality systému, rozdelené do logických celkov, služieb. Keďže služby neuchovávajú žiaden stav, sú v zaregistrované do *DI* kontajnera ako Singleton. Výnimkou sú ViewModely, ktoré uchovávajú stav aplikácie pre každého užívateľa, preto sú zaregistrované ako Scoped. Aby nebolo nutné každú službu po jej vytvorení registrovať osobitne, budú v logickej vrstve vytvorené rozhrania pre jednotlivé druhy živostnosti. Následne bude stačiť, aby trieda reprezentujúca službu rozhranie implementovala. Pri registrovaní vrstvy sa prejde pomocou knižnice *Reflection* menný priestor vrstvy a služby sa zaregistrujú do kontajnera s príslušnou životnosťou.

V prípade potreby sa vrstva napája na databázu pomocou databázovej služby, ktorá využíva databázovú vrstvu systému. Pre viac informácií o tom, ako funguje napojenie na databázu viď 3.2.4.

#### ViewModely

Ako je spomínané v prezentačnej vrstve (viď 3.2.2), ViewModely z architektonického vzoru Model–View–ViewModel (*MVVM*) sú uložené v logickej vrstve. Nie sú navrhnuté tak, aby existovali pre každý View z aplikačnej vrstvy, ale sú skonštruované podľa druhu spravovaných dát. Napríklad pre pohľad zobrazenia histórie formálneho/formátovaného textu a pre jeho úpravu je použitý jeden ViewModel spravujúci formálne a formátované texty.

ViewModel je vstupným bodom prezentačnej vrstvy systému. Stará sa o volanie ďalších služieb z logickej vrstvy, prípadne o získavanie dát z iných ViewModelov. ViewModely medzi sebou komunikujú a posieľajú si správy o zmenách pomocou návrhového vzoru Messenger, niekedy nazývaného mediátor.




Obr. 3.4: Jednotlivé ViewModely si do Messengera registrujú správy, ktoré chcú prijímať. Keď ViewModel odošle správu, nastane spätné volanie a ViewModely, ktoré správu zaregistrovali, ju dostanú. Vďaka tomu nemusia mať ViewModely medzi sebou závislosti a môžu spolu komunikovať pomocou správ. Obrázok upravený z DotNetPattern [10].

### Modely

Modely predstavujú v MVVM dátovú časť programu, tak ako bolo vysvetlené v prezentačnej vrstve (viď 3.2.2). Sú to objekty, uchovávajúce dáta systému. Môže sa na ne referovať aj ako na doménové objekty.

Modely sa v systéme rozdeľujú do dvoch druhov, detailné modely a modely pre zoznamy. Detailné modely obsahujú všetky dáta, ktoré k modelu existujú, zatiaľ čo modely pre zoznamy obsahujú iba ich časť, využívanú pre zobrazenie v zozname. Napríklad, pri zozname používateľových repozitárov nie je nutné, aby model obsahoval všetky svoje konvencie, ale stačí poznať ich počet, čo môže byť užívateľovi v zozname zobrazené. Primárnym dôvodom využívania dvoch druhov konvencií je zníženie veľkosti dát, jednak kvôli ich prenosu, ale aj pre ich spracovávanie a mapovanie. V prípade rozsiahlejších systémov je možné modely rozdeľovať na viacero druhov, presne podľa aktuálne potrebných dát.

Pretváranie jedného objektu na iný sa nazýva mapovanie. Všetky objekty, ktoré logická vrstva získa z iných služieb pretvára na modely, s ktorými ďalej pracuje. Napríklad databázová vrstva vráti objekt vo formáte entity, ktorý sa následne premapuje na vhodný model. K mapovaniu systém využije *NuGet* balíček AutoMapper[7]. AutoMapper je jednoduchá malá knižnica, vytvorená na zbavenie sa kódu, ktorý mapoval jeden objekt na druhý [45]. Pri použití balíčku stačí v kóde vytvoriť mapu medzi dvoma objektami, ktoré je následne možné pomocou automappera mapovať. Pri vytváraní mapy je možné definovať, ako sa mapovanie bude správať. Vlastnosti objektov s rovnakými názvami sa vedia medzi sebou mapovať bez nastavení.

### GitHub API služba

Navrhnutou autorizáciou z prezentačnej vrstvy (viď 3.2.2) získa systém prístupový token užívateľa ku *GitHub API*. Token bude systém využívať k získavaniu informácií o užívateľovi a jeho repozitároch, ku ktorým sa v systéme spravujú konvencie.

Logická vrstva obsahuje službu zabezpečujúcu volania do *GitHub API*. Služba je navrhnutá tak, aby využívala *NuGet* balíček Oktokit[8]. Octokit je klientská knižnica, ktorá poskytuje jednoduchý spôsob interakcie s *GitHub API* [30]. Služba získa požadované in-

[7] https://www.nuget.org/packages/automapper
[8] https://www.nuget.org/packages/Octokit

formácie, premapuje ich na príslušné modely a vráti ich ViewModelom alebo inej službe, ktorá dáta požadovala.

### Služba zmien textov

Ako je špecifikované v požiadavkách, užívatelia môžu konvencie ukladať pomocou formátovaného textu, ku ktorému je dostupná história zmien. Každá konvencia môže taktiež obsahovať formálny zápis pravidiel, ku ktorému je história dostupná. Ukladanie celého formálneho a formátovaného textu do databázy je obzvlášť pri väčšom množstve textov a ich väčšej dĺžke nepraktické.

Do databázy sa teda budú ukladať iba zmeny medzi jednotlivými textami. K tomu navrhnujem použiť *NuGet* balíček diff–match–patch[9]. Balíček, ktorý v origináli vznikol v roku 2006 na podporu Google dokumentov, ponúka robustné algoritmy na vykonávanie operácií, potrebných na synchronizáciu textu [35]. Medzi ne patrí aj vytváranie patchu, listu zmien medzi dvoma textami. Patch je následne možné aplikovať na text, aby z neho vznikol text druhý.

Databáza bude obsahovať iba patche na predchádzajúce verzie a aktuálny text. Keď užívateľ požiada o históriu zmien textu, je možné ju pomocou aktuálneho textu a patchov vygenerovať, spolu s príslušnými zmenami.

### Služba zvýrazňujúca časti formálneho textu

Konvencie obsahujú formálny text v *EditorConfig* formáte, ktorý je založený na *INI* formáte (pre viac informácií viď 2.3.4). Pre zlepšenie užívateľskej skúsenosti je vhodné, aby boli jednotlivé časti formálneho textu v užívateľskom rozhraní zvýraznené. Farebné zvýraznenie odlíši jednotlivé druhy textu vo formáte, napríklad sekcie budú inou farbou a v inom formáte ako komentáre. Tým by mal užívateľ jednoduchšie identifikovať, že nejde o prostý, ale o formálny text. Zvýrazňovanie prípadne pomôže aj pri hľadaní chýb vo formáte textu.

Pre implementáciu zvýrazňovania jednotlivých častí formálneho textu priamo v systéme by bolo nutné implementovať analýzu INI formátov, čo by bolo časovo náročné. Preto som sa rozhodol o využitie extérneho riešenia. Za vhodnú považujem webovú aplikáciu hilite.me[10]. Ide o malú webovú aplikáciu, ktorá kód v prostom texte prevádza do pekne zvýrazneného *HTML* formátu [46]. Je k nej vytvorené *API*[11], ktoré zjednodušuje jej automatizované použitie.

Služba zvýrazňujúca časti formálneho textu pošle cez *API* do webovej aplikácie hilite.me text na zvýraznenie. Vrátený text služba prejde a upraví jeho formátovanie, aby bolo v súlade s témou systému. Následne text v *HTML* formáte vráti službe alebo ViewModelu, ktorý o zvýraznenie požiadal.

### Pravidlá EditorConfigu

Formálny text v *EditorConfig* formáte obsiahnutý v konvenciách obsahuje páry kľúča a hodnoty. Kľúč sa nazýva vlastnosťou (property) alebo pravidlom (rule). Tieto páry sú umiestnené pod sekciou, ktorá určuje, na aké súbory sa ich kontrola aplikuje. Pár určuje, aké pravidlo a s akou hodnotou sa aplikuje. Formálne texty a teda ich pravidlá bude možné

[9] https://www.nuget.org/packages/Diff.Match.Patch
[10] http://hilite.me
[11] http://hilite.me/api

v systéme kontrolovať na repozitári konvencie, ako aj importovať z repozitára. Pre viac informácií o fungovaní *EditorConfigu* viď 2.3.4.

*EditorConfig* v základe podporuje 7 pravidiel, ktoré budú implementované v logickej vrstve systému v *EditorConfig* službe. Špeciálna vlastnosť `root`, ktorá určuje koreňový *EditorConfig* súbor v súborovom systéme implementovaná nebude, keďže pri využívaní mimo súborový systém je bezvýznamná. Následne je vhodné definovať zoznam pravidiel s ich podporou v systéme, pričom podporou je myslená možnosť kontrolovať alebo importovať pravidlá, či už z časti alebo celkovo:

- `charset`: Určuje znakovú sadu (encoding) súboru. Plne podporovaná.
- `end_of_line`: Určuje reprezentáciu konca riadku. Plne podporovaná.
- `indent_size`: Určuje počet stĺpcov pre úroveň odsadenia a šírku tabulátorov. Šírka tabulátorov nie je a nemôže byť podporovaná. Je vlastnosťou *IDE*, nie súboru.
- `indent_style`: Určuje použitie medzier alebo tabov pri odsadení. Plná podpora.
- `insert_final_newline`: Určuje či je súbor ukončený novým riadkom. Plná podpora.
- `tab_width`: Určuje počet stĺpcov reprezentujúcich znak tabu. Vytvorená, avšak nepodporuje import ani kontrolu. Nemožno podporovať mimo *IDE*. Je vlastnosťou *IDE*, nie súboru.
- `trim_trailing_whitespace`: Určuje, či môžu byť prázdne znaky znakom nového riadka. Plná podpora.

Okrem základných pravidiel som sa rozhodol pridať podporu aj pre vlastnosti špecifického jazyka. Pridanie ďalších, rozširujúcich pravidiel má ukázať, ako sa do systému pridávajú ďalšie pravidlá. K čiastočnému cieleniu ukážok ďalších pravidiel som sa rozhodol, pretože väčšina automatizovaných nástrojov je cielených na určité jazyky, čím sa vymedzujú ku konvenciám štandardne používaným v danom jazyku. Systém budem vytvárať v *.NET* ekosystéme, preto som sa rozhodol vytvoriť ukážky vlastností používaných v *.NET* prostredí. Ďalším dôvodom zvolenia je existencia rozšírení pre .NET (vid rozšírenia *EditorConfigu* v 2.3.4), čo umožňuje implementovať už existujúce a používané pravidlá. Ďalšie pravidlá, ktoré navrhujem implementovať:

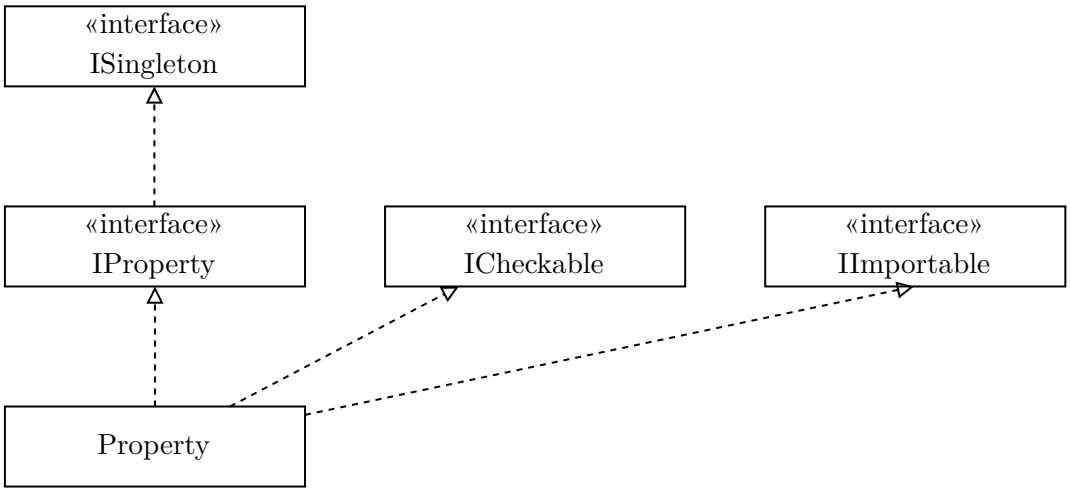
- `csharp_new_line_before_catch`: Určuje, či je výraz `catch` na novom riadku v kóde jazyka C#. Plná podpora.
- `csharp_new_line_before_else`: Určuje, či je výraz `else` na novom riadku v kóde jazyka C#. Plná podpora.
- `csharp_new_line_before_finally`: Určuje, či je výraz `finally` na novom riadku v kóde jazyka C#. Plná podpora.

Pravidlá kontrolujú v kóde vlastnosti, pre ktoré je nutné spraviť syntaktickú a lexikálnu analýzu. K tomu je možné použiť Roslyn. Roslyn je *open source* implementácia prekladača jazyka C# s *API* pre budovanie nástrojov na analýzu kódu [66]. Pomocou neho je možné z kódu vytvoriť syntaktický strom, nájsť hľadané výrazy a skontrolovať ich okolie.

### Pridávanie pravidiel

Základnou požiadavkou na systém je možnosť pridávať vlastné pravidlá. Tá je zabezpečená vytvorením rozhrania *IProperty*, ktorého implementovaním sa trieda stáva pravidlom

(vlastnosťou). K vytváraniu pravidiel bude vytvorená dokumentácia. Viac informácií k dokumentácii sa nachádza na konci časti o návrhu logickej vrstvy.




Obr. 3.5: Obrázok zobrazujúci vlasnosť (pravidlo), ktorá implementuje rozhrania pre ňu určené. Implementácia rozhrania *IProperty* je povinná, ňou sa začne trieda považovať za vlasnosť. Tým že *IProperty* implementuje *ISingleton* sa vlastnosť automatický zaregistruje aj do *IoC* kontajnera. Implementácia rozhraní *IImportable* a *ICheckable* je nepovinná. Rozhrania sa implementujú v prípade, ak má byť vlastnosť kontrolovateľná alebo importovateľná z repozitára. Viac informácií o rozhraniach pre kontrolu a import pravidiel sa nachádza v častiach o nich.

Z požiadaviek na systém vyplýva, že systém by mal obsahovať zoznam implementovaných pravidiel s ich popisom. Rozhranie *IProperty* preto núti triedy, aby obsahovali meno a popis pravidla. Zoznam pravidiel sa následne získa pomocou knižnice *Reflection* z tried implementujúcich rozhranie. V zozname implementovaných pravidiel je viditeľné, či je pravidlo kontrolovateľné podľa toho, či implementuje korešpondujúce rozhranie.

### Import pravidiel

Importovanie pravidiel (vlastností) z *GitHub* repozitárov je možné implementovaním *IImportable* rozhrania. Import sa prevádza na formálnom texte konvencie. V prvom kroku si užívateľ import pravidiel nastavuje. Najprv si zvolí, či chce formálny text rozšíriť o novo získané pravidlá, alebo ho chce nahradiť. Taktiež si vyberie vetvu v repozitári, z ktorej sa budú pravidlá importovať a zvolí pravidlá, ktoré si praje importovať. Pri každom pravidle si zvolí sekciu v repozitári v *EditorConfig* formáte. Prvotne sa užívateľovi zvolí implicitná sekcia, ktorú musí definovať každé pravidlo, ktoré implementovalo *IImportable* rozhranie. Sekcia určuje, z akých súborov v repozitári sa bude pravidlo importovať. Pri výbere importovaných pravidiel bude možné zvoliť výber všetkých naraz, pre uľahčenie práce pri nových repozitároch v systéme.

Po zvolení nastavení prebehne samotný import, ktorý má niekoľko fáz. Počas priebehu môže byť prerušený využitím tokenu prerušenia (cancelation token). Jednotlivé fázy importu sú:

1. Inicializácia, vytvorenie potrebných štruktúr, získanie služieb z *IoC* kontajnera.

2. Stiahnutie zvolenej vetvy repozitára na server.
3. Investigácia stiahnutých súborov. Jednotlivým pravidlám sa priradia cesty k súborom podľa zvolených sekcií.
4. Zavolanie `Import` metódy pre každé pravidlo na každom súbore z investigácie. `Import` metódu implementuje každé pravidlo implementovaním *IImportable* rozhrania.
5. Zjednotenie výsledkov `Import` metód pre každé pravidlo.
6. Zmazanie vetvy repozitára, ukončenie importu.

Zisťovanie, či súbor korešponduje so sekciou pravidla a teda, či sa má pravidlo zo súboru importovať bude riešené pomocou Roslyna. Po otvorení riešenia (solution) vo Visual Studiu, ktoré natívne *EditorConfig* podporuje, sa Roslyn pokúša nájsť *EditorConfig* súbory a dodržiavať ich. Roslyn teda obsahuje funkcionalitu na prácu s *EditorConfig* súbormi.

`Import` metóda ma jeden parameter a tým je súbor na ktorom je spustená. Ako výsledok vracia dvojice hodnoty pravidla a váhy, s akou je v súbore hodnota zastúpená. Napríklad pre pravidlo `csharp_new_line_before_else` na súbore, kde by bol výraz `else` dvakrát na novom riadku a raz nie, by mohla byť hodnota `true` s váhou dva a hodnota `false` s váhou jedna.

Po ukončení importu sa užívateľovi zobrazia všetky importované pravidlá s nájdenými hodnotami a ich percentuálnym zastúpením. Z hodnôt sa automaticky zvolí tá s najvyšším zastúpením, avšak užívateľ ju môže zmeniť, nastaviť na `unset` alebo vôbec nenaimportovať. U všetkých pravidiel hodnota `unset` ruší efekt daného pravidla, keď bolo nastavené už predtým [41].

### Kontrola pravidiel

Kontrolu bude možné spúšťať na pravidlách (vlastnostiach), ktoré implementujú *ICheckable* rozhranie. Pre implementovanie rozhrania je nutné vytvoriť texty, ktoré užívateľom objasnia, aké sú povolené hodnoty a typy súborov pri danej kontrole. Implementovať je nutné aj metódu `Check`, ktorá na súbore spustí dané pravidlo so zadanou hodnotou. Výsledkom metódy je úspešná alebo neúspešná kontrola s chybovou správou.

Kontrola pravidiel sa bude spúšťať v systéme v pohľade zvoleného repozitára. Užívateľ si zvolí kontrolované konvencie a vetvu repozitára, na ktorej kontrola prebehne. Následne sa spustí kontrola, skladajúca sa z viacerých fáz. Kontorla môže byť prerušená využitím tokenu prerušenia (cancelation token). Jednotlivé fázy kontroly sú:

1. Inicializácia, vytvorenie potrebných štruktúr, získanie služieb z *IoC* kontajnera.
2. Stiahnutie zvolenej vetvy repozitára na server.
3. Prepojenie stiahnutých súborov s pravidlami. Pravidlám v konvenciách sa priradia cesty k súborom podľa ich sekcií.
4. Zavolanie `Check` metódy pre každé pravidlo na každom súbore.
5. Zjednotenie a normalizovanie výsledkov kontroly.
6. Uloženie výsledkov do databázy.
7. Zmazanie vetvy repozitára, ukončenie kontroly.

Pri prepojení repozitárov s formálnymi textami konvencií je nutné formálne texty spracovať a získať z nich pravidlá s ich hodnotami v príslušných sekciách. K tomu využijem

rovnako ako pri importe pravidiel Roslyna, ktorý obsahuje funkcionalitu na prácu s *EditorConfig* súbormi. Ošetrenie zadania neexistujúceho pravidla je riešené na úrovni kontroly. Tá pravidlo detekuje a nastaví ho na neúspešné so zodpovedajúcou chybovou správou. Nesprávna hodnota pravidla sa detekuje v `Check` metóde, ktorá vráti príslušnú výnimku, ktorú odchytí kontrola a taktiež nastaví pravidlo na neúspešné s chybovou správou.

Výsledky sú užívateľovi prezentované ako formálne texty konvencií. Jednotlivé riadky sú sfarbené buď na zeleno v prípade úspechu alebo na červeno v prípade neúspechu. Riadok sa nezafarbí, ak neobsahuje pravidlo alebo pravidlo nebolo na žiadnom súbore spustené. Riadky sfarbené na červeno bude možné otvoriť a sledovať u nich správy k chybám na súboroch. Každá kontrola sa automaticky ukladá do databázy a je možné sa k jej výsledkom vrátiť.

#### Dokumentácia

Z požiadaviek na navrhovaný systém vystáva vytvorenie návodu na nasadenie systému a na implementáciu vlastných pravidiel vo formálnom texte. Pre dokumentáciu som sa rozhodol využiť DocFX. DocFX je generátor *API* dokumentácie pre *.NET*, ktorý generuje dokumentáciu z dokumentačných komentárov v kóde [60]. Podporuje taktiež využitie *Markdown* súborov na vytvorenie ďalších tém, ako sú návody a články [60].

DocFX som sa rozhodol využiť pre jeho napojenie na *.NET* ekosystém, ktorý je k riešeniu systému využitý. Návody vytvorím ako *Markdown* súbory a možnosť generovania dokumentácie z dokumentačných komentárov využijem pri návode na implementáciu vlastných pravidiel. K časti kódu, do ktorej sa budú pravidlá pridávať, vytvorím *API* dokumentáciu, na ktorú sa budem odkazovať z návodu. Dokumentáciu následne spoločne so systémom uverejním v službe *GitHub*, pričom z nej vytvorím stránku pomocou GitHub Pages. GitHub Pages je spôsob ako hostovať stránku priamo z GitHub repozitára [33].

### 3.2.4 Databázová vrstva

Databázová vrstva slúži ako prístupový bod k perzistentným dátam pre logickú vrstvu. Vytvorením databázovej vrstvy sa oddeľuje logika databázy od logiky aplikácie. Logická vrstva sa stáva nezávislá na spôsobe uloženia dát a druhu databázy. K práci s databázou je v návrhu systému použitý Entity Framework.

#### Entity Framework

Entity Framework je *open source ORM* rámec pre *.NET* aplikácie, podporovaný Microsoftom [16]. Umožňuje vývojárom pracovať s databázovými objektami a tabuľkami ako s .NET objektami, čo eliminuje väčšinu kódu napísaného k prístupu k databáze [16]. Konkrétne navrhujem využitie Entity Framework Core, ktorý je novšou odľahčenejšou verziou Entity Frameworku, s použitím *Code–First* prístupu. *Code–First* prístup sa zameriava na doménu aplikácie, vytvárajú sa pri ňom triedy nazývané entity reprezentujúce dizajn databázy, z ktorých sa následne vygeneruje databázová schéma [15].

Entity Framework Core sa do projektu inštaluje pomocou nuget balíčku[12]. Po jeho nainštalovaní bude možné vytvoriť kontext databázy vytvorením triedy dediacej `DbContext` triedu. `DbContext` je neoddeliteľnou súčasťou Entity Frameworku, jeho inštancia reprezentuje reláciu s databázou, ktorú je možné použiť na dotazovanie a ukladanie inštancií entít

[12] https://www.nuget.org/packages/Microsoft.EntityFrameworkCore

do databázy [13]. Do triedy reprezentujúcej kontext databázy sa vložia entity a prístupové údaje k databáze. Následne je možné vygenerovať a aplikovať databázovú migráciu.

#### Entity

Entita je v Entity Frameworku trieda, ktorá sa mapuje na databázovú tabuľku. Musí byť pridaná ako `DbSet<TEntity>` do `DbContext` triedy [14]. Entity Framework mapuje každú entitu na tabuľku a každú vlasnosť entity na stĺpec v tabuľke, čo umožňuje vytvárať dizajn databázy. Referencovaním medzi entitami je možné vytvárať referencie medzi tabuľkami.

V databáze budú uložené údaje o užívateľoch, ich repozitároch a ich konvenciách. Ku konvencii budú uložené informácie k jej komentárom, formálnemu a formátovanému textu. K textom bude uložená história v podobe zmien medzi ich verziami. K repozitárom budú uložené záznamy o vykonaných kontrolách konvencií na nich.

#### Návrhový vzor Repozitár a UnitOfWork

Na odtienenie použitej technológie na prácu s databázou som sa rozhodol využiť návrhové vzory Repozitár a UnitOfWork. Odtienením sa logická vrstva stáva nezávislou na použitom rámci a druhu databázy v databázovej vrstve.

Repozitár je iba trieda, ktorá implementuje logiku prístupu k dátam [67]. V systéme bude existovať generický repozitár obsahujúci základné operácie nad dátami, ako je vyhľadávanie podľa identifikátora, pridávanie, mazanie a aktualizovanie entít. Operácie špecifické pre danú entitu budú riešené pomocou rozširujúcich metód (extensions methods).

UnitOfWork sa správa ako obchodná transakcia, spája transakcie zo všetkých repozitárov do jednej, ktorá sa vykoná naraz [67]. UnitOfWork v navrhnutom systéme obsahuje repozitár pre každú entitu. Dovoľuje vytvoriť transakciu, vykonať čiastkové transakcie z repozitárov a vykonať ich nad databázou spolu, prípadne všetky vrátiť späť a nevykonať žiadnu, ak v priebehu nastala chyba. V logickej vrstve sa vytvorí databázová služba, ktorá bude obsahovať komplexné transakcie nad databázou podľa potreby vrstvy.

Do *IoC* kontajnera je ako jediničik zaregistrovaný UnitOfWork manažér, ktorého účelom je vytvárať novú inštanciu UnitOfWork pre každú transakciu. Na logickej vrstve požiada služba alebo ViewModel o novú inštanciu UnitOfWork. Následne v repozitároch inštancie vykoná požadované transakcie a pomocou UnitOfWork zmeny odošle.

## 3.3 Implementácia návrhu

Časť práce popisujúca implementáciu systému. Webovú aplikáciu a dokumentáciu som implementoval v súlade s navrhnutou architektúrou (viď 3.2). Snažil som sa o vytvorenie kvalitného, čistého a ľahko rozšíriteľného kódu. V prvej časti sa venujem prostrediu, použitého k implementácii (viď 3.3.1), následne popisujem problémové časti implementácie (viď 3.3.1) a jej nasadenie (viď 3.3.3).

### 3.3.1 Implementačné prostredie

Systém bol implementovaný na operačnom systéme Windows 10 v prostredí Visual Studio 2019. Visual Studio 2019[13] je plnohodnotné *IDE* pre operačný systém *Windows 10* vyvinuté firmou *Microsoft* [58]. Prostredie som zvolil, pretože je doporučeným *IDE* so všetkými

[13] https://visualstudio.microsoft.com

funkcionalitami pre vývoj *.NET* aplikácií. V prostredí som využil rozšírenie *ReSharper*[14], ktoré rozširuje *IDE* o ďalšie funkcionality zrýchľujúce vývoj a zvyšujúce kvalitu kódu. Taktiež som využil rozšírenie Markdown editor[15], ktoré do prostredia pridáva editor *Markdown* súborov. Výsledný systém obsahuje *Markdown* súbory v dokumentácii a *Markdown* `README` súbor v koreňovej zložke projektu.

Visual Studio som okrem samotného písania kódu, hľadania chýb a lokálneho nasadzovania systému využíval aj k udržiavaniu tajomstiev ako je prístupový reťazec do databázy, klientské tajomstvo a klientské ID ku *GitHub* OAuth aplikácii. K implementácii som využíval verzný systém Git priamo v prostredí Visual Studia, ktoré obsahuje vstavané nástroje na prácu s Git repozitárom. Zdrojové súbory systému som uchovával v privátnom projekte služby *GitLab*, kde som si zaznamenával aj informácie k aktuálneemu stavu diplomovej práce. Po dokončení vývoja som zdrojové súbory zverejnil v službe *GitHub* a nasadil v cloudovej platforme Azure (pre viac informácií viď 3.3.3).

### 3.3.2 Problémové časti implementácie

Vďaka navrhnutiu architektúry systému bolo už pred začiatkom implementácie jasné, aké funkcionality bude mať výsledný systém, na aké časti bude systém rozdelený a za čo budú mať časti zodpovednosť, poprípade aká technológia bude pre ich implementáciu využitá. Pri implementácii niektorých častí nastali problémy, s ktorými architektúra systému nepočítala. V tejto časti sa snažím ilustrovať najpodstatnejšie z týchto problémov a popísať ako boli vyriešené.

#### Model–View–ViewModel

Implementovanie Model–View–ViewModel bolo problematické. Vzor je využívaný prevažne s technológiou *WPF*, ku ktorej existujú *NuGet* balíčky na jej jednoduchšiu implementáciu. Tieto balíčky pri View počítajú s použitím xaml súborov pre užívateľské rozhranie, čo nie je kompatibilné s Razor Pages pri technológii Blazor. Tieto baličky sú prevažne pre .NET Framework, ktorý nie je plne kompatibilný s .NET Core.

Problematická bola časť rozdelenia logiky View a ViewModela. Nepodarilo sa mi implementovať View bez C# kódu na pozadí, ktorý by komunikoval s ViewModelmi. Problém je vyriešený tenkou vrstvou C# kódu na strane View, ktorý volá jednotlivé časti ViewModelov. Riešenie z časti narušuje čistotu kódu, avšak na používanie aplikácie nemá žiadny vplyv.

Taktiež sa mi nepodarilo implementovať návrhový vzor Command, tak ako je využívaný vo *WPF*. Command je návrhový vzor, ktorý zapuzdruje požiadavku do objektu [8]. Problémové bolo naviazanie Commandu na View. Namiesto Commandu je v systéme použité priame volanie metódy ViewModela.

Návrhový vzor Messenger bol implementovaný pomocou nuget balíčka MvvmLightLibs[16] pre .NET Standard. ViewModely medzi sebou nemajú závislosti a komunikujú cez Messenger. Práca s formálnymi a formátovanými textami konvencie je riešená pomocou dialógov volaných z pohľadu konvencie. ViewModel formálneho a formátovaného textu by mal veľkú závislosť na ViewModel konvencie, čo by zapríčiňovalo posielanie veľkého množstva správ medzi nimi. Preto som sa rozhodol tieto ViewModely zlúčiť a spravovať formálne a formátované texty konvencií vo ViewModeli konvencie.

[14] https://www.jetbrains.com/resharper
[15] https://github.com/madskristensen/MarkdownEditor
[16] https://www.nuget.org/packages/MvvmLightLibsStd10

#### História formátovaného textu

Tak ako je špecifikované, k vytváraniu zmien textov v systéme je použitý *NuGet* balíček diff–match–patch[17]. Ten je však určený na prácu s prostými textami bez formátovania. Pri použití formátovaného textu nemusí aplikácia zmien dopadnúť správne.

Preto sa pred vytvorením nových zmien z textu odstráni formátovanie a do databázy sa uložia iba prosté zmeny bez neho. V databáze sa nachádza aktuálny text s formátovaním a jeho história bez formátovania. Systém teda podporuje formátovaný text, avšak pri histórii si formátovanie nepamätá.

#### Charset pravidlo

Pravidlo `Charset` určuje znakovú sadu (encoding) súboru, tu nie je vždy možné presne rozoznať, keďže súbor nemusí obsahovať dostatočné informácie. Roslyn sa k problému stavia tak, že znakovú sadu prehlási za `utf-8`, ak sa mu nepodari zistiť iná.

K zisteniu znakovej sady súboru systém používa `StreamReader`. Pri jeho použití sa môže zameniť `latin1` a `utf-8` kódovanie za `utf-8bom`. Problém nie je riešiteľný a na danú skutočnosť systém upozorňuje v popise pravidla.

#### Roslyn

V návrhu je Roslyn použitý na na zisťovanie, či sekcia pravidla korešponduje s cestou k súboru a teda či má byť pravidlo na súbore kontrolované alebo z neho importované. Roslyn danú funkcionalitu obsahuje, avšak nie je verejná a je ju teda možné použiť iba zvnútra. Problém som vyriešil použitím knižnice *Reflection*, pomocou ktorej systém získa prístup k neverejnej metóde Roslyna a zavolá ju danými parametrami.

Podobný problém nastal pri využití Roslyna na získanie pravidiel s ich hodnotami a sekciami z formátovaného textu. Roslyn zadanú funkcionalitu obsahuje, avšak nie je verejne dostupná. V tomto prípade ani nebolo vhodné ju zavolať cez *Reflection*, pretože pracuje s inými typmi objektov a mapovanie by bolo komplikované. Časť potrebnú na získavanie pravidiel Roslyna som skopíroval do kódu systému, pozmenil, aby bolo možné ju použiť a označil jej pôvod.

### 3.3.3 Nasadenie

Systém bol po dokončení implementácie nasadený v cloudovej platforme Azure[18]. Azure je cloudová platforma s viac ako dvesto produktami a službami od spoločnosti *Microsoft* [64]. Zvolil som ju, pretože podporuje ASP.NET aplikácie a je natívne prepojená s Visual Studiom. Systém ostane nasadený počas testovania a následného hodnotenia.

Inštancia je nasadená ako aplikačná služba úrovne B1, ktorá poskuje 1.75 GB pamäte a 100 *ACU*. K aplikácii je vytvorený SQL server a Key vault, ktorý udržiava jej tajomstvá. Pri zvolenom výkone nasadenia je vhodná primárne na testovacie účely s využívaním importu a kontroly na repozitároch do 1000 súborov. K aplikácii je naviazaná vlastná doména a SSL certifikát. K nasadeniu systému je vytvorený článok v dokumentácii[19].

[17] https://www.nuget.org/packages/Diff.Match.Patch
[18] https://ccms.orlicek.net/
[19] https://orlicekm.github.io/CodingConventionsManagementSystem/articles/deployment.html

Okrem nasadenia inštancie systému som po dokončení implementácie zverejnil zdrojové súbory spolu s dokumentáciou v službe *GitHub*[20]. Následne som z dokumentácie z repozitára vytvoril stránku pomocou GitHub Pages[21].

[20] https://github.com/orlicekm/CodingConventionsManagementSystem
[21] https://orlicekm.github.io/CodingConventionsManagementSystem

# Kapitola 4

# Testovanie

Kapitola sa zaoberá procesom testovania vytvoreného systému potencionálnymi užívateľmi (viď 4.2). Dôraz bol kladený na výsledky testovania s ohľadom na silné a slabé stránky systému a vhodné oblasti použitia. K testovaniu bola vytvorená aj demonštrácia systému na open source projekte (viď 4.1).

## 4.1 Demonštrácia

Časť práce demonštrujúca použitie systému na vybranom *open source* projekte. Pre demonštráciu som vybral projekt Microsoft PowerToys z analýzy vybraných projektov (viď 2.2.2). Projekt som vybral, pretože bol v práci analyzovaný a z časti obsahuje aj kód v jazyku C#, čo umožní demonštrovanie využitia pravidiel pre jeho kontrolu.

Pre prácu s projektom v systéme som si vytvoril jeho kópiu pomocou funkcie Fork. Ak by som chcel, aby mal v systéme k repozitáru prístup aj iný užívateľ bolo by nutné ho pridať ako prispievateľa do projektu v službe *GitHub*.

Testovať aplikáciu na nasadenej inštancii neodporúčam s viac ako 1000 súbormi v repozitári. Pri zvolenom projekte, ktorý obsahuje viac ako 2800 súborov, trval import konvencií približne 5 minút a ich kontrola trvala približne 8 minút. Pre prácu s väčšími repozitármi je vhodné systém nasadiť na výkonnejšie riešenie.

### Prihlasovanie do systému

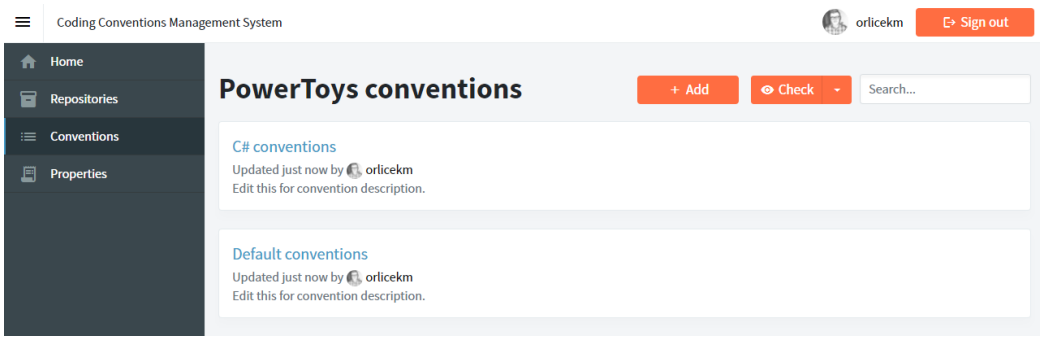
Užívateľ sa do systému prihlasuje v pravom hornom rohu aplikácie pomocou *GitHub* účtu. Pri prvom prihlásení je nutné systému povoliť prístup k informáciám o účte a repozitároch. Bez prihlásenia nie je možné systém používať.

### Výber repozitára

Užívateľ môže v systéme zvoliť repozitár, na ktorom bude spravovať programovacie konvencie. K repozitárom sa užívateľ dostane zvolením repozitárov (Repositories) v ľavom menu aplikácie. Následne si zvolí repozitár kliknutím naň. K repozitárom sú zobrazené užitočné údaje. ktorými sú popis, jazyk, licencia, počet konvencií, počet kontrol, počet forkov, počet hviezdičiek (*stargazers*) alebo počet otvorených problémov (open issues). Medzi repozitármi je možné vyhľadávať. Svoj aktuálne zvolený repozitár môže užívateľ vidieť v pravom spodnom rohu aplikácie. Ja som zvolil repozitár PowerToys, na ktorom je systém demonštrovaný.

### Správa konvencií

Po zvolení repozitára sa užívateľovi otvorí pohľad spravovania konvencií na danom repozitári. Do okna sa vie spätne dostať zvolením konvencií (Conventions) v ľavom menu aplikácie. Kliknutím na tlačítko pridať (Add) sa užívateľovi otvorí dialóg pridávania konvencie. Pridané konvencie sa následne zobrazujú užívateľovi spolu s užitočnými údajmi, ktorými sú popis konvencie, počet sekcií vo formálnom texte, počet pravidiel vo formálnom texte alebo počet komentárov. Medzi konvenciami je možné vyhľadávať.




Obr. 4.1: Do repozitára som pridal dve konvencie, jedna bude slúžiť na spravovanie predvolených konvencií ku všetkému, zatiaľ čo druhá sa bude sústrediť na jazyk C#.

Po kliknutí na pridanú konvenciu sa užívateľovi otvorí pohľad konvencie. V ňom môže konvenciu premenovať alebo zmazať.

### Komentovanie

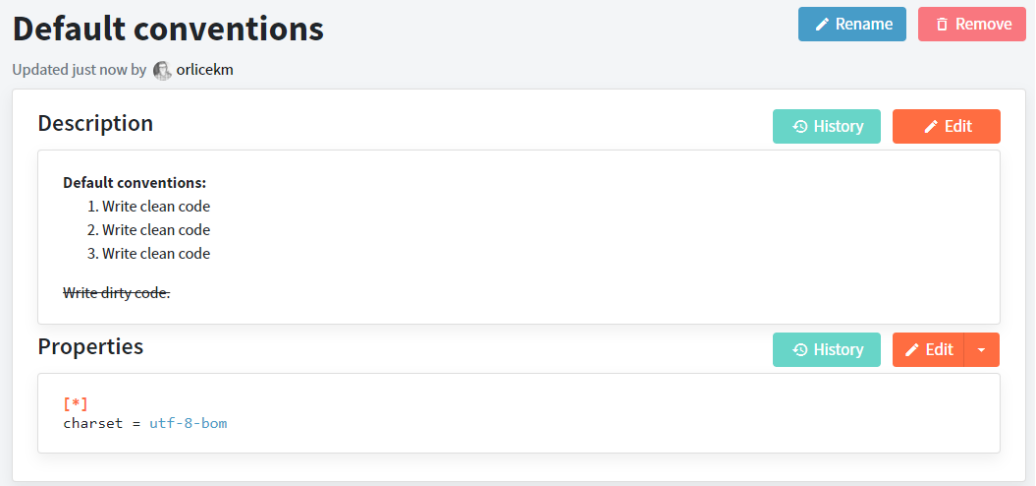
V spodnej časti pohľadu konvencie užívateľ vidí komentáre k danej konvencii. Napísaním textu a kliknutím na tlačítko pridať, môže ku konvencii pridať vlastný komentár.

### Podporované vlastnosti

Zvolením vlastnosti (Properties) v ľavom menu aplikácie sa užívateľovi zobrazia systémom podporované vlastnosti (alebo inak nazývané pravidlá). V nich si vie pozrieť popis, podporované súbory alebo hodnoty a zistiť, či je možné ich kontrolovať a importovať.

### Správa textov

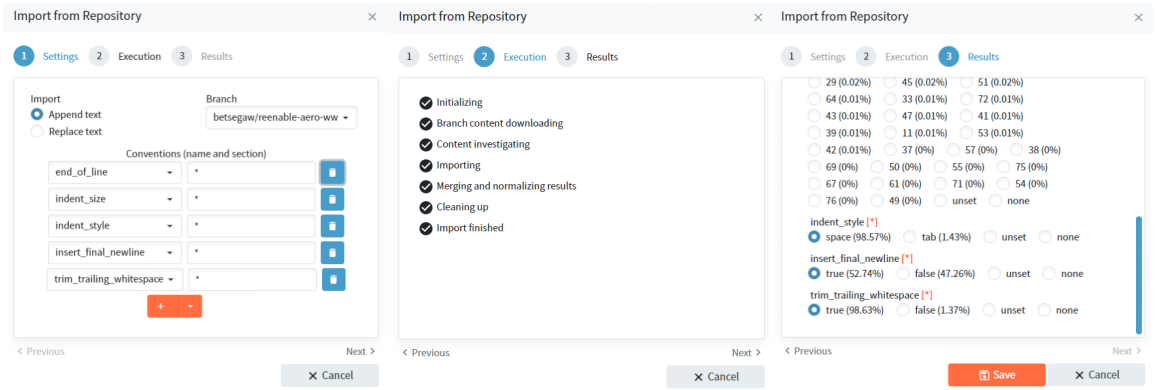
V pohľade konvencie môže užívateľ spravovať formátovaný (Description) a formálny (Properties) text konvencie. Kliknutím na tlačítko upraviť, ho môže meniť a kliknutím na históriu môže sledovať históriu zmien daného textu.




Obr. 4.2: Upravený formátovaný a formálny text konvencie, formálny text bude kontrolovať kódovanie všetkých súborov.

### Import vlastností

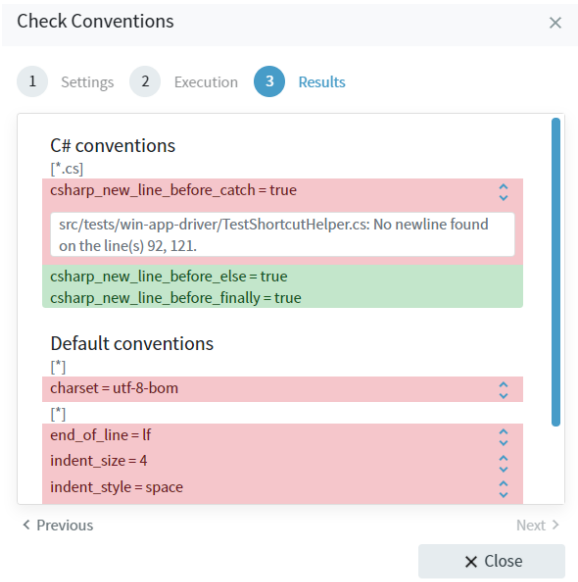
Vlastnosti je možné importovať kliknutím na šípku v tlačítku upraviť (Edit) pri formálnom texte konvencie. Následne si užívateľ zvolí, či chcel text pridať alebo nahradiť, z akej vetvy bude import prebiehať a aké vlastnosti (pravidlá) bude importovať v akých sekciách. Potom spustí samotný import. Po jeho skončení si užívateľ zvolí hodnoty, ktoré chce importovať.




Obr. 4.3: Do formálneho textu som dal importovať zvyšné predvolené pravidlá. Po dokončení som zvolil najčastejšie nájdené hodnoty.

### Kontrola vlastností

Kontrolovať vlastnosti je možné v pohľade repozitára kliknutím na tlačítko kontrola (Check). Následne si užívateľ zvolí vetvu repozitára a konvencie, ktoré chce na nej kontrolovať. Potom spustí samotnú kontrolu. Po jej skončení si užívateľ môže prezerať výsledky, ktoré sú automaticky uložené do databázy. K výsledkom predošlých kontrol sa môže vrátiť kliknutím na šípku tlačítka kontrola.




Obr. 4.4: Po importovaní pravidiel aj pre C# konvencie, som spustil na repozitári kontrolu. Na obrázku je možné vidieť jej výsledok. Zelené pravidlá prešli, zatiaľ čo pri červených nastala chyba. Tie je možné rozkliknúť, na zistenie podrobností.

## 4.2 Testovanie užívateľmi

Časť práce popisujúca priebeh testovania systému jeho potencionálnymi užívateľmi. V prvej časti vysvetľuje, ako boli účastníci testovania vybraný (viď 4.2.1), následne popisuje priebeh (viď 4.2.2) a v poslednej časti diskutuje silné a slabé stránky systému, určuje vhodné oblasti použitia s ohľadom na výsledky testovania. Pre porovnanie existujúcich technológii s funkcionalitami navrhované systému viď 3.1.1 a 3.1.2.

### 4.2.1 Výber účastníkov

Pri volení kritérií na účastníkov testovania som sa snažil zvoliť také, aby účasníci čo najlepšie reprezovali možných užívateľov systému. Preto som od účastníkov očakával:

- Aspoň jeden rok skúseností s programovaním.
- Poznanie pojmu *open source* a stretnutie sa *open source* projektom.
- Chápanie programovacích konvencií a ich významu.

Pod stretnutím sa s *open source* projektom myslím zoznámenie sa so zdrojovým kódom a spôsobom akým nejaký *open source* projekt funguje. Celkovo som oslovil trinásť ľudí spĺňajúcich dané kritéria.

### 4.2.2 Priebeh

Účastníci testovania dostali prístup k demonštrácii projektu, ktorá bola prepisom demonštrácie z tejto práce (viď 4.1). Následne dostali odkazy k repozitáru v službe *GitHub*, dokumentácii a nasadenej inštancii systému. Od účastníkov som očakával že si projekt naštudujú

a vrátia mi spätnú väzbu. Odpoveď mala formát volného textu, keďže som im poslal textový súbor s dvoma sadami otázok:

- Vedeli by ste projekt využiť? Ak áno, na akých projektoch a za akých podmienok?
- Čo Vám v systéme chýba? Co by ste chceli zmeniť? Čo Vám naopak systéme vyhovuje?

Z trinásť dotázaných ľudí som dostal deväť odpovedí. Jedna z nich neodpovedala na otázky, ale iba stručne hodnotila, že systém je v poriadku.

### 4.2.3 Výsledky

Výsledky od účastníkov som rozdelil do štyroch kategórii. V prvej sa venujem vhodným oblastiam použitia systému. V nasledujúcej zmenám, ktoré by bolo vhodné v systéme spraviť, aby bol viac užívateľsky prívetivý a vhodný na použitie. V ďalšej menujem rozšírenia systému a v poslednej aspekty, ktoré užívateľom na systéme vyhovovali.

#### Vhodne oblasti využitia

Väčšina, šesť z ôsmich účastníkov odpovedala, že by projekt využiť v aktuálnom stave nedokázala. Dvaja účastníci ktorý by projekt využiť vedeli a kontrolovali by vždy všetky konvencie na každom projekte. Ďalší traja účastníci si vedia predstaviť projekt využiť pri pridaní väčšieho množstva kontrol na špecifické jazyky.

Z môjho pohľadu sú vhodnými projektami na využitie také, ktoré vyžadujú vyššiu kvalitu kódu (pre viac informácii viď 2.1). Medzi oblasti v ktorých sa vyžaduje vyššia kvalita kódu patrí napríklad zdravotníctvo, letectvo, armáda alebo autonómne vozidlá.

#### Vhodné zmeny

Väčšina navrhovaných zmien sa týkala užívateľského rozhrania:

- Premiestnenie zobrazovania zvoleného repozitára.
- Osobitné tlačítka pre históriu kontroly a importovanie.
- Väčšie okná pre dialógy importu a kontroly, možnosť expandovať dialóg na celú obrazovku.
- Chyby v textoch.
- Prerobenie okna kontroly, zobrazovať namiesto čísla riadka priamo jeho text, poprípade celú časť a súboru a v nej chybu zvýrazniť. Nepraktické hľadanie chýb pri ich väčšom množstve.

Chyby v textoch som v aplikácii opravil.

#### Vhodné rozšírenia

Rozšírenia, ktoré užívatelia žiadali sa týkali prevažne rozsiahlejších zásahov do systému:

- Pridanie ďalších pravidiel na importovanie a kontrolovanie.
- Pridanie módu na čítanie (read-only mode) pre verejné projekty, ktorých nie je užívateľ účastníkom.

- Preloženie demonštrácie do angličtiny a vloženie do dokumentácie.
- Možnosť vytvoriť *pull request* na repozitár na základe kontroly.
- Spúšťanie kontroly na každom vytvorenom *pull requeste*, pridanie možnosti spustenia kontoroly pomocou GitHub Actions.
- Ku každému pravidlu pridať do ich zoznamu príklad použitia.

**Vyhovujúce aspekty**

Zoznam aspektov, ktorý užívateľom na systéme vyhovoval:

- Priama interakcia a napojenie na *GitHub*.
- Možnosť kontrolovať všetko v každom programovacom jazyku.
- Formát formálnych textov.
- Zdieľanie konvencii na rovnakom projekte medzi užívateľmi.

V zozname som neuviedol nešpecifikované odpovede ako „všetko“ alebo „práca s konvenciami“.

# Kapitola 5

# Záver

Cieľom práce je navrhnúť a implementovať systém na správu programovacích konvencií v projekte. Motiváciou využitia programovacích konvencií v projekte je zvýšenie čistoty kódu, ktorá zefektívňuje vývoj. Hlavným zámerom bolo vytvoriť systém, ktorý umožní uchovávať všetky druhy používaných programovacích konvencií a zároveň umožní užívateľovi dané konvencie automaticky kontrolovať a generovať. Systém je cielený primárne pre verejne dostupné *open source* projekty na platforme *GitHub*.

Pri vytváraní návrhu systému som uskutočnil niekoľko prieskumov. V prvom rade som zisťoval, aké benefity využitia programovacích konvencií v projektoch prinesie (viď 2.1). Následne som robil prieskum konvencií v *open source* projektoch (viď 2.2.1), z ktorých som vybral určité projekty pre podrobnejšiu analýzu (viď 2.2.2). Okrem toho som uskutočnil analýzu technológií spravujúcich programovacie konvencie, vďaka čomu som zistil, že väčšina projektov používa na uloženie konvencií iba nejakú formu formátovaného textu. Najčastejšie používaným automatizovaným nástrojom bol EditorConfig (viď 2.3.4), ktorý som sa rozhodol použiť pri návrhu a zapracovať do vytvoreného systému.

Pri prieskume konvencií v projektoch som skúmal päťdesiat projektov v službe *GitHub*, ktoré získali najviac hviezdičiek (*stargazers*) za rok 2020 a zároveň obsahujú *open source* licenciu. Zoznam projektov som získal v BigQuery dátovom sklade, uchovávajúcom informácie o službe *GitHub*. Z prieskumu som zistil, že veľké množstvo projektov konvencie vôbec neobsahuje a tie, ktoré ich obsahovali mali medzi sebou veľkú diverzitu.

Systém som sa rozhodol navrhnúť ako webovú službu s využitím technológie Blazor, napojenú na databázu a uloženú na Azure cloude, pričom som sa pokúsil o využitie najnovších technológií. Taktiež som sa snažil cieliť na *open source* projekty, preto je prihlasovanie navrhnuté cez protokol OAuth pomocou *GitHub* účtu.

Navrhnutý systém som následne implementoval, vytvoril k nemu dokumentáciu a nasadil ho do cloudovej platformy Azure. Pri implementácii som sa stretol s niekoľkými problematickými časťami (viď 3.3.2), v ktorých som nemohol postupovať podľa návrhu. Tie som následne ilustroval a popísal, akým spôsobom boli vyriešené.

Ako pri analýze, tak aj pri návrhu som sa nestretol so žiadnymi vážnejšími problémami. Najväčšou výzvou bolo pre mňa nájdenie vhodnej knižnej literatúry k programovacím konvenciám, keďže sa mi nedarilo nájsť knihy zamerané priamo na túto tematiku. Avšak existuje mnoho kníh zameraných na refaktorizáciu a čistotu kódu, ktoré túto tematiku riešia.

K systému bola vytvorená dokumentácia, nasadená ako GitHub Pages stránka, obsahujúca návod ako systém nasadiť a ako do systému pridať vlastné pravidlá na kontrolu konvencií, spolu s *API* dokumentáciou. Projekt bol demonštrovaný na *open source* projekte a testovaný možnými užívateľmi.

Na záver by som zmienil, že prácu by bolo možné ešte rozšíriť o ďalšie funkcionality, pričom inšpirácia by sa čerpala z dát získaných z testovania. Výsledný systém obsahuje iba základné funkcionality, získane z analýz na spravovanie programovacích konvencií. Pre lepšie využitie by bolo vhodné systém spraviť pohodlnejším na používanie. To by šlo docieliť najmä integrovaním systému do viacerých verzných služieb a pridaním rôznych druhov importov a exportov konvencií, ako aj pridaním viacerých pravidiel pre kontrolu. Verím, že implementovaný systém a vykonané analýzy v tejto práci pomôžu rozšíreniu výskumu a vývoju technológii v oblasti spravovania programovacích konvencií.

# Literatúra k práci


[1] Agile Alliance. What is Agile? *Agile 101* [online]. [cit. 2020-11-18]. Dostupné z: https://www.agilealliance.org/agile101.

[2] Alls, J. *Clean Code in C#: Refactor your legacy C# code base and improve application performance by applying best practices*. Packt Publishing, 2020. ISBN 978-1-8389-8297-3.

[3] Ankur. *C# Coding Standards* [online]. [cit. 2021-03-10]. Dostupné z: https://www.c-sharpcorner.com/UploadFile/ankurmalik123/C-Sharp-coding-standards.

[4] Bellairs, R. Coding best practices. *What is code quality? And how to improve code quality* [online]. [cit. 2020-11-18]. Dostupné z: https://www.perforce.com/blog/sca/what-code-quality-and-how-improve-code-quality.

[5] Cone, M. Advanced features that build on the basic Markdown syntax. *Extended Syntax* [online]. [cit. 2020-12-30]. Dostupné z: https://www.markdownguide.org/extended-syntax.

[6] Cordasco, I. S. *Flake8* [online]. [cit. 2021-03-17]. Dostupné z: https://flake8.pycqa.org/en/latest/manpage.html.

[7] Data & Object Factory, LLC.. *C# Coding Standards and Naming Conventions* [online]. [cit. 2020-11-23]. Dostupné z: https://www.dofactory.com/reference/csharp-coding-standards.

[8] Data & Object Factory, LLC. *C# Command* [online]. [cit. 2021-07-28]. Dostupné z: https://www.dofactory.com/net/command-design-pattern.

[9] Data & Object Factory, LLC.. *Dofactory* [online]. [cit. 2020-11-23]. Dostupné z: https://www.dofactory.com.

[10] DotNetPattern.com. *MVVM Light Messenger* [online]. [cit. 2021-07-27]. Dostupné z: http://dotnetpattern.com/mvvm-light-messenger.

[11] EditorConfig Team.. *EditorConfig Properties* [online]. [cit. 2021-01-04]. Dostupné z: https://github.com/editorconfig/editorconfig/wiki/EditorConfig-Properties.

[12] EditorConfig Team.. *EditorConfig Specification* [online]. [cit. 2021-01-04]. Dostupné z: https://editorconfig-specification.readthedocs.io.

[13] EntityFrameworkTutorial.net. *Entity Framework Core: DbContext* [online]. [cit. 2021-07-28]. Dostupné z: https://www.entityframeworktutorial.net/efcore/entity-framework-core-dbcontext.aspx.

[14] EntityFrameworkTutorial.net. *What is an Entity in Entity Framework?* [online]. [cit. 2021-07-28]. Dostupné z: https://www.entityframeworktutorial.net/basics/entity-in-entityframework.aspx.

[15] EntityFrameworkTutorial.net. *What is Code–First?* [online]. [cit. 2021-07-28]. Dostupné z: https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx.

[16] EntityFrameworkTutorial.net. *What is Entity Framework?* [online]. [cit. 2021-07-27]. Dostupné z: https://www.entityframeworktutorial.net/what-is-entityframework.aspx.

[17] Facebook. *React* [online]. [cit. 2021-01-07]. Dostupné z: https://github.com/facebook/react.

[18] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Druhé. Addison-Wesley, 2019. ISBN 978-1-5093-0698-5.

[19] García, I. S. The theory beyond the pattern. *Learn MVVM* [online]. [cit. 2021-07-27]. Dostupné z: https://www.learnmvvm.com/theory.html.

[20] GeeksforGeeks. *GeeksforGeeks* [online]. [cit. 2020-11-24]. Dostupné z: https://www.geeksforgeeks.org.

[21] Gefroh, J. Software engineering is a lot easier in consistent systems. *Why consistency is one of the top indicators of good code* [online]. [cit. 2020-11-19]. Dostupné z: https://medium.com/@jgefroh/why-consistency-is-one-of-the-top-indicators-of-good-code-352ba5d62020.

[22] GHTorrent. *GHTorrent on the Google cloud* [online]. [cit. 2020-12-20]. Dostupné z: https://ghtorrent.org/gcloud.html.

[23] GitHub. *About wikis* [online]. [cit. 2021-01-02]. Dostupné z: https://docs.github.com/en/free-pro-team@latest/github/building-a-strong-community/about-wikis.

[24] GitHub. *Adding or editing wiki pages* [online]. [cit. 2021-01-02]. Dostupné z: https://docs.github.com/en/free-pro-team@latest/github/building-a-strong-community/adding-or-editing-wiki-pages.

[25] GitHub. *Configuring issue templates for your repository* [online]. [cit. 2021-01-01]. Dostupné z: https://docs.github.com/en/free-pro-team@latest/github/building-a-strong-community/configuring-issue-templates-for-your-repository.

[26] GitHub. *Creating an issue* [online]. [cit. 2021-01-01]. Dostupné z: https://docs.github.com/en/free-pro-team@latest/github/managing-your-work-on-github/creating-an-issue.

[27] GitHub. *The largest open source community in the world* [online]. [cit. 2020-12-17]. Dostupné z: https://github.com/open-source.

[28] GitHub. *Mastering Issues* [online]. [cit. 2020-12-31]. Dostupné z: https://guides.github.com/features/issues.

[29] GitHub. *Mastering Markdown* [online]. [cit. 2020-12-30]. Dostupné z: https://guides.github.com/features/mastering-markdown.

[30] GitHub. *Octokit — GitHub API Client Library for .NET* [online]. [cit. 2021-06-29]. Dostupné z: https://github.com/octokit/octokit.net.

[31] GitHub. *Octoverse* [online]. [cit. 2020-12-22]. Dostupné z: https://octoverse.github.com.

[32] GitHub. *Using templates to encourage useful issues and pull requests* [online]. [cit. 2021-01-01]. Dostupné z: https://docs.github.com/en/free-pro-team@latest/github/building-a-strong-community/using-templates-to-encourage-useful-issues-and-pull-requests.

[33] GitHub, Inc.. *GitHub Pages* [online]. [cit. 2021-07-27]. Dostupné z: https://pages.github.com.

[34] Google. *C# at Google Style Guide* [online]. [cit. 2020-11-24]. Dostupné z: https://google.github.io/styleguide/csharp-style.html.

[35] Google. *Diff Match Patch* [online]. [cit. 2021-06-30]. Dostupné z: https://github.com/google/diff-match-patch.

[36] Google. *Google Style Guides* [online]. [cit. 2020-11-24]. Dostupné z: https://google.github.io/styleguide.

[37] Google, LLC. Waterfall model vs agile scrum. *Google Trends* [online]. [cit. 2020-11-15]. Dostupné z: https://trends.google.com/trends/explore?date=all&q=waterfall%20model,agile%20scrum.

[38] Grigorik, I. *GH Archive* [online]. [cit. 2020-12-22]. Dostupné z: https://www.gharchive.org.

[39] Grimes, R. A. a Fruhlinger, J. *What is OAuth? How the open authorization framework works* [online]. [cit. 2021-01-17]. Dostupné z: https://www.csoonline.com/article/3216404/what-is-oauth-how-the-open-authorization-framework-works.html.

[40] Hoffa, F. *Analyzing GitHub* [online]. [cit. 2020-12-22]. Dostupné z: https://github.com/fhoffa/analyzing_github.

[41] Hunner, T. a Xu, H. *EditorConfig* [online]. [cit. 2021-01-02]. Dostupné z: https://editorconfig.org.

[42] IBM Cloud Education. *Three–Tier Architecture* [online]. [cit. 2021-07-22]. Dostupné z: https://www.ibm.com/cloud/learn/three-tier-architecture.

[43] JetBrains s.r.o.. *IntelliJ IDEA overview* [online]. [cit. 2021-03-12]. Dostupné z: https://www.jetbrains.com/help/idea/discover-intellij-idea.html#editorconfig.

[44] JetBrains s.r.o.. *Use EditorConfig* [online]. [cit. 2021-03-12]. Dostupné z: https://www.jetbrains.com/help/resharper/Using_EditorConfig.html.

[45] Jimmy Bogard. *AutoMapper* [online]. [cit. 2021-07-27]. Dostupné z: https://automapper.org/.

[46] Kojevnikov, A. *Hilite.me* [online]. [cit. 2021-06-30]. Dostupné z: https://github.com/alexkay/hilite.me.

[47] Křena, B. a Kočí, R. Studijní opora. *Úvod do softwarového inženýrství* [online]. FIT VUT v Brně, 2010 [cit. 2021-05-22]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php/course/IUS-IT/texts/IUS_opora.pdf.

[48] Langa Łukasz. *Black* [online]. [cit. 2021-03-17]. Dostupné z: https://github.com/psf/black.

[49] Light, A. *Cpplint* [online]. [cit. 2021-03-12]. Dostupné z: https://github.com/google/styleguide/tree/gh-pages/cpplint.

[50] Martin, R. C. *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2008. ISBN 978-0-1323-5088-4.

[51] Martin, R. C. a Martin, M. *Agile principles, patterns, and practices in C#*. Prentice Hall, 2006. ISBN 978-0-1318-5725-4.

[52] Microsoft. *ASP.NET Core Blazor hosting models* [online]. [cit. 2021-01-17]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0.

[53] Microsoft. C# Programming Guide. *C# Coding Conventions* [online]. [cit. 2020-11-19]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions.

[54] Microsoft. C# Guide. *C# documentation* [online]. [cit. 2020-11-21]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp.

[55] Microsoft. *Code style rule options* [online]. [cit. 2021-03-12]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/code-style-rule-options.

[56] Microsoft. *Dependency injection in ASP.NET Core* [online]. [cit. 2021-07-27]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0.

[57] Microsoft. *Dependency injection in .NET* [online]. [cit. 2021-07-27]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection.

[58] Microsoft. Visual Studio 2019. *Downloads* [online]. [cit. 2021-01-17]. Dostupné z: https://visualstudio.microsoft.com/downloads.

[59] Microsoft. *Framework Design Guidelines* [online]. [cit. 2020-11-22]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines.

[60] Microsoft. *Getting Started with DocFX* [online]. [cit. 2021-07-27]. Dostupné z: https://dotnet.github.io/docfx/tutorial/docfx_getting_started.html.

[61] Microsoft. *Introduction to ASP.NET Core Blazor* [online]. [cit. 2021-07-27]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-5.0.

[62] Microsoft. *MICROSOFT DOCUMENTATION* [online]. [cit. 2020-11-21]. Dostupné z: https://docs.microsoft.com.

[63] Microsoft. *Microsoft PowerToys* [online]. [cit. 2021-01-07]. Dostupné z: https://github.com/microsoft/PowerToys.

[64] Microsoft. *What is Azure?* [online]. [cit. 2021-01-17]. Dostupné z: https://azure.microsoft.com/en-us/overview/what-is-azure.

[65] Morris, P. *What is Blazor?* [online]. [cit. 2021-01-17]. Dostupné z: https://blazor-university.com/overview/what-is-blazor.

[66] .NET Foundation. *Roslyn* [online]. [cit. 2021-07-26]. Dostupné z: https://github.com/dotnet/roslyn.

[67] Nguyen, L. *How to implement Repository & Unit of Work design patterns in .NET Core* [online]. [cit. 2021-07-28]. Dostupné z: https://enlabsoftware.com/development/how-to-implement-repository-unit-of-work-design-patterns-in-dot-net-core-practical-examples-part-one.html.

[68] OpenJS Foundation. *ESLint: About* [online]. [cit. 2021-03-12]. Dostupné z: https://eslint.org/docs/about.

[69] Prakash, S. C# Coding Standards. *GeeksforGeeks* [online]. [cit. 2020-11-24]. Dostupné z: https://www.geeksforgeeks.org/c-sharp-coding-standards.

[70] Prettier. *Prettier vs. Linters* [online]. [cit. 2021-01-05]. Dostupné z: https://prettier.io/docs/en/comparison.html.

[71] Prettier. *What is Prettier?* [online]. [cit. 2021-01-05]. Dostupné z: https://prettier.io/docs/en/index.html.

[72] Rabelo, J. *Three-Tier Architecture* [online]. [cit. 2021-07-22]. Dostupné z: https://www.techopedia.com/definition/24649/three-tier-architecture.

[73] Rasmusson, J. Agile's engine for getting things done. *Iterations* [online]. [cit. 2020-11-18]. Dostupné z: http://www.agilenutshell.com/iterations.

[74] Ritchie, D. a Kernighan, B. *The C Programming Language*. Druhé. New Jersey: Prentice Hall, 1988. ISBN 978-0-1311-0370-2.

[75] Stack Overflow. 2018. *Developer Survey Results* [online]. [cit. 2020-11-16]. Dostupné z: https://insights.stackoverflow.com/survey/2018.

[76] Stack Overflow. 2019. *Developer Survey Results* [online]. [cit. 2021-03-09]. Dostupné z: https://insights.stackoverflow.com/survey/2019.

[77] Testim. *What Is a Linte* [online]. [cit. 2021-01-05]. Dostupné z: https://www.testim.io/blog/what-is-a-linter-heres-a-definition-and-quick-start-guide.

[78] The Clang Team. *ClangFormat* [online]. [cit. 2021-03-18]. Dostupné z: https://clang.llvm.org/docs/ClangFormat.html.

[79] The Daring Fireball Company LLC.. *Markdown* [online]. [cit. 2020-12-30]. Dostupné z: https://daringfireball.net/projects/markdown.

[80] The Linux Information Project. *Source Code Definition* [online]. [cit. 2020-10-11]. Dostupné z: http://www.linfo.org/source_code.html.

[81] The mypy project. *Mypy* [online]. [cit. 2021-03-17]. Dostupné z: http://mypy-lang.org.

[82] The SublimeLinter Community. *About SublimeLinter* [online]. [cit. 2021-01-05]. Dostupné z: http://www.sublimelinter.com/en/v3.10.10/about.html.

[83] TheAlgorithms. *The Algorithms - Python* [online]. [cit. 2021-01-07]. Dostupné z: https://github.com/TheAlgorithms/Python.

[84] UnityCoin. *Clean Code — Uncle Bob / Lesson 1* [online]. [cit. 2020-11-19]. Dostupné z: https://www.youtube.com/watch?v=7EmboKQH8lM.

[85] Voříšek, L. *Analýza kvality zdrojových kódů* [online]. 2015. Bakalárska práca. České vysoké učení technické v Praze.

[86] Vuollet, P. *The 9 Coding Standards C# Developers Need to Get Started* [online]. [cit. 2021-03-10]. Dostupné z: https://blog.submain.com/coding-standards-c-developers-need.

[87] Zielczynski, P. *Requirements Management Using IBM Rational RequisitePro*. Prvé. IBM Press, 2007. ISBN 978-0-321-38300-6.

# Skratky

**ACU** Azure compute unit. 41

**API** Application Programming Interface Str. 4, 16, 17, 31, 33, 34, 35, 38, 49

**ASCII** American Standard Code for Information Interchange Str. 10

**CC** Creative Commons Str. 17

**CI** Continuous integration Str. 60

**CLA** Contributor License Agreement Str. 13

**CLR** Common Language Runtime Str. 59

**CSS** Cascading Style Sheets Str. 22, 29

**DI** Dependency injection. 29, 31, 32

**DoD** Definition of Done Str. 6, 9

**GUI** Graphical user interface. 30

**HTML** Hyper Text Markup Language Str. 17, 18, 22, 29, 34

**HTTP** Hypertext Transfer Protocol Str. 10

**IDE** Integrated development environment Str. 6, 9, 12, 20, 21, 22, 35, 39, 40, 59, 60

**INI** Initialization File Str. 21, 34

**IoC** Inversion of control. 31, 36, 37, 39

**JSON** JavaScript Object Notation Str. 22

**LINFO** The Linux Information Project Str. 6

**MIT** Massachusetts Institute of Technology Str. 3

**MVVM** Model-View-ViewModel. 30, 32

**ORM** Object-relational mapping Str. 28, 38

**OTI** Object Technology International Str. 8

**SoC** Separation of concerns. 30

**SQL** Structured Query Language Str. 17, 59

**TDD** Test-driven development Str. 9

**UML** Unified Modeling Language Str. 9

**VM** Virtual machine Str. 22

**WPF** Windows Presentation Foundation. 30, 40

**XHTML** Extensible HyperText Markup Language Str. 18

**XML** eXtensible Markup Language Str. 17

# Slovník

**.NET** je zastrešujúci názov pre súbor technológií v softvérových produktoch[1] od spoločnosti *Microsoft*, ktoré tvoria spoločnú platformu. Štandardizovanou špecifikáciou .NET jadra je *CLR* [18]. Str. 3, 16, 17, 21, 27, 29, 35, 38, 40

**Agilný vývoj softvéru** je vývoj založený na iteráciách. Dokáže reagovať na zmenu požiadaviek počas priebehu vývojového cyklu. Protikladom je lineárny *vodopádový model* [8]. Pre viac informácií z práce viď 2.1.1. Str. 3, 5, 6, 16, 60

**BigQuery** je plne spravovateľný serverless dátový sklad, ktorý umožnuje škálovaľnú analýzu na veľkom množstve dát a podporuje dopytovanie pomocou *SQL* [9]. Str. 10

**Clean code** je neredundantný kód, napísaný systematicky tak, aby ho iný programátor mohol jednoducho pochopiť a upraviť [24]. Pre viac informácií z práce viď 2.1.4. Str. 3, 5

**Eclipse** je *IDE* obsahujúce základný pracovný priestor a rozšíriteľný systém doplnkov na jeho prispôsobenie. Primárne využitie je pre vývoj *Java* aplikácií [11]. Str. 8

**Enterprise softvér** je počítačový softvér, ktorý sa používa skôr na uspokojenie potrieb organizácie ako jednotlivých používateľov [12]. Str. 16

**Git** je systém na kontrolu verzií, ktoré umožňujú sledovanie zmien v súboroch a koordináciu prác na týchto súboroch medzi viacerými programátormi [13]. Str. 11, 13, 14, 25, 59, 60

**GitHub** je hostingová služba[2] pre verzný systém *Git*, ktorá bola nedávno odkúpená spoločnosťou *Microsoft*. Služba sa používa predovšetkým na zdieľanie počítačového kódu a umožňuje vytvorenie repozitára zadarmo, vďaka čomu je bežne používaná na *open source* projekty [14]. Str. 3, 4, 5, 10, 11, 12, 13, 16, 17, 18, 19, 20, 21, 25, 26, 27, 31, 33, 36, 38, 40, 42, 43, 46, 48, 49, 59, 60, 64

**GitHub Issue** je spôsob ako sledovať úlohy, vylepšenia a chyby v projektoch [2]. Str. 11, 13, 14, 15, 18, 19, 24, 25

**Gitter** je platforma na chatovanie, ktorá pomáha spravovať a rozširovať komunity prostredníctvom správ. Výhodou je integrovatelnosť s viacerými službami, medzi ktoré patrí napríklad *GitHub* [6]. Str. 13

[1] https://dotnet.microsoft.com/
[2] https://github.com/

**JavaScript** multiplatformný, objektovo orientovaný skriptovací jazyk [15]. Str. 12, 14, 17, 21, 22, 29

**Markdown** je odľahčený značkovací jazyk, ktorý slúži na úpravu prostého textu a jeho následný prevod na formátovaný text publikovateľný na webe [16]. Pre viac o Markdown súboroch ako spôsobe ukladania programovacích konvencií viď 2.3.1. Str. 11, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 38, 40

**NuGet** je bezplatný a *open source* manažér pre správu balíčkov, určený pre vývojovú platformu spoločnosti *Microsoft* [7]. Str. 31, 33, 34, 40, 41

**Open source** produkty obsahujú oprávnenia na použitie zdrojových súborov, dizajnových dokumentov alebo ich obsahu. Často sa spája s open source modelom, ktorý je založený na otvorenej spolupráci a v ktorom sú produkty vydané pod open source licenciami. [19]. Str. 3, 4, 5, 8, 9, 10, 12, 13, 15, 16, 21, 23, 24, 25, 26, 29, 35, 38, 43, 46, 49, 59, 60, 64

**Pull request** je navrhovaná zmena do *Git* repozitára, odoslaná od užívateľa a akceptovaná alebo odmietaná spolupracovníkmi repozitára [3]. Str. 10, 11, 13, 14, 15, 18, 19, 48

**Pytest** je rozhranie pre *Python*, ktoré umožňuje ľahké písanie malých testov, ale podporuje komplexné funkcionálne testovanie aplikácií a knižníc [4]. Str. 13

**Python** je vysoko úrovňový skriptovací programovací jazyk, vyvíjaný ako *open source* projekt [20]. Str. 12, 13, 22, 60

**Refaktorizácia** je proces reštrukturalizácie existujúceho počítačového kódu bez zmeny jeho vonkajšieho správania. Refaktorizácia je určená na zlepšenie dizajnu, štruktúry a implementácie softvéru pri zachovaní jeho funkčnosti [10]. Pre viac informácií z práce viď 2.1.6. Str. 3, 6, 7, 9, 16

**ReSharper** je rozšírenie do *IDE* Visual Studio, ktore rozširuje o viac ako 2200 inšpekcií kódu za behu, na ktoré ponúka rýchle opravy [5]. Str. 9, 21, 40

**Scrum** je *agilný* rámec na vývoj, dodávanie a udržiavanie komplexných produktov. Je určený pre tímy s 10 a menej členmi, ktorí rozdeľujú svoju prácu na ciele, ktoré je možné splniť v rámci časovo obmedzených iterácií [21]. Str. 5, 6

**Stargazer (GitHub)** je človek, ktorý dal hviezdičku repozitáru v službe *GitHub* [1]. Str. 10, 43, 49

**Travis CI** je hostovaná *CI* služba, používaná na *CI* projektov v službách *GitHub* a *BitBucket* [6]. Str. 13

**Vodopádový model** je zloženie vývoja do lineárnych sekvenčných fáz, kde každá závisí na predchádzajúcej. Protikladom je *agilný vývoj softvéru* [22]. Str. 3, 5, 59

**WebAssembly** je otvorený štandard, ktorý definuje prenosný formát binárneho kódu pre spustiteľné programy, ako aj pre rozhrania na uľahčenie interakcií medzi týmito programami a ich hostiteľským prostredím [23]. Jeho hlavným cieľom je umožniť vytváranie vysoko výkonných aplikácií na webových stránkach, ale formát je navrhnutý na vykonávanie a integráciu aj v iných prostrediach [23]. Str. 29

**Windows** je séria niekoľkých rodín operačných systémov od spoločnosti *Microsoft* [17]. Str. 12, 13

# Literatúra k slovníku


[1] GitHub. *List stargazers* [online]. [cit. 2021-07-26]. Dostupné z: https://docs.github.com/en/rest/reference/activity#list-stargazers.

[2] GitHub. *Mastering Issues* [online]. [cit. 2020-12-31]. Dostupné z: https://guides.github.com/features/issues.

[3] GitHub Incorporation. *Pull request — GitHub Glossary* [online]. [cit. 2021-01-23]. Dostupné z: https://help.github.com/articles/github-glossary/#pull-request.

[4] Holger Krekel and pytest-dev team. *Pytest* [online]. [cit. 2021-07-26]. Dostupné z: https://docs.pytest.org/en/6.2.x.

[5] JetBrains s.r.o.. *Reshaper* [online]. [cit. 2021-07-25]. Dostupné z: https://www.jetbrains.com/resharper.

[6] New Vector Ltd. *Gitter* [online]. [cit. 2021-07-25]. Dostupné z: https://gitter.im.

[7] Wikipedia.

[8] Wikipedia. *Agile software development — Wikipedia, The Free Encyclopedia* [online]. [cit. 2020-09-29]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Agile%20software%20development&oldid=979979593.

[9] Wikipedia. *BigQuery — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-07-26]. Dostupné z: http://en.wikipedia.org/w/index.php?title=BigQuery&oldid=1017050899.

[10] Wikipedia. *Code refactoring — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-11-01]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Code%20refactoring&oldid=985307174.

[11] Wikipedia. *Eclipse (software) — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-11-02]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Eclipse%20(software)&oldid=980854952.

[12] Wikipedia. *Enterprise software — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-07-25]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Enterprise%20software&oldid=1028172826.

[13] Wikipedia. *Git — Wikipedia, The Free Encyclopedia* [online]. [cit. 2020-09-30]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Git&oldid=979088000.

[14] Wikipedia. *GitHub — Wikipedia, The Free Encyclopedia* [online]. [cit. 2020-09-30]. Dostupné z: http://en.wikipedia.org/w/index.php?title=GitHub&oldid=981138144.

[15] Wikipedia. *JavaScript — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-01-14]. Dostupné z: https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1000595969.

[16] Wikipedia. *Markdown — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-01-14]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Markdown&oldid=1000478104.

[17] Wikipedia. *Microsoft Windows — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-01-14]. Dostupné z: https://sk.wikipedia.org/w/index.php?title=Microsoft_Windows&oldid=6925930.

[18] Wikipedia. *.NET — Wikipedia, The Free Encyclopedia* [online]. [cit. 2020-09-30]. Dostupné z: http://sk.wikipedia.org/w/index.php?title=.NET&oldid=6465183.

[19] Wikipedia. *Open source — Wikipedia, The Free Encyclopedia* [online]. [cit. 2020-09-30]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Open%20source&oldid=979840260.

[20] Wikipedia. *Python — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-01-18]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=1000914465.

[21] Wikipedia. *Scrum (software development) — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-11-01]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Scrum%20(software%20development)&oldid=985930384.

[22] Wikipedia. *Waterfall model — Wikipedia, The Free Encyclopedia* [online]. [cit. 2020-09-29]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Waterfall%20model&oldid=979592091.

[23] Wikipedia. *WebAssembly — Wikipedia, The Free Encyclopedia* [online]. [cit. 2021-07-27]. Dostupné z: http://en.wikipedia.org/w/index.php?title=WebAssembly&oldid=1035244878.

[24] Wiktionary. *clean code — Wiktionary, The Free Dictionary* [online]. [cit. 2020-09-30]. Dostupné z: https://en.wiktionary.org/w/index.php?title=clean_code&oldid=59815596.

# Príloha A

# Analyzované projekty

*Open source* projekty v službe *GitHub* použité na analýzu v časti 2.2.1.

Tabuľka A.1: Analyzované projekty 1. časť

| | Repository |
|---|---|
| 1 | jwasham/coding-interview-university |
| 2 | donnemartin/system-design-primer |
| 3 | EbookFoundation/free-programming-books |
| 4 | danistefanovic/build-your-own-x |
| 5 | TheAlgorithms/Python |
| 6 | microsoft/PowerToys |
| 7 | trekhleb/javascript-algorithms |
| 8 | denoland/deno |
| 9 | flutter/flutter |
| 10 | sindresorhus/awesome |
| 11 | ytdl-org/youtube-dl |
| 12 | florinpop17/app-ideas |
| 13 | vuejs/vue |
| 14 | CSSEGISandData/COVID-19 |
| 15 | facebook/react |
| 16 | bradtraversy/design-resources-for-developers |
| 17 | ohmyzsh/ohmyzsh |
| 18 | microsoft/vscode |
| 19 | cli/cli |
| 20 | goldbergyoni/nodebestpractices |
| 21 | torvalds/linux |
| 22 | github/gitignore |
| 23 | ossu/computer-science |
| 24 | microsoft/playwright |
| 25 | huggingface/transformers |
| 26 | Genymobile/scrcpy |
| 27 | willmcgugan/rich |
| 28 | gothinkster/realworld |
| 29 | tiangolo/fastapi |
| 30 | PanJiaChen/vue-element-admin |
| 31 | jlevy/the-art-of-command-line |

Tabuľka A.2: Analyzované projekty 2. časť

| | Repository |
|---|---|
| 32 | microsoft/terminal |
| 33 | GitSquared/edex-ui |
| 34 | evanw/esbuild |
| 35 | tensorflow/tensorflow |
| 36 | anuraghazra/github-readme-stats |
| 37 | vinta/awesome-python |
| 38 | freeCodeCamp/freeCodeCamp |
| 39 | ryanmcdermott/clean-code-javascript |
| 40 | angular/angular |
| 41 | golang/go |
| 42 | tuvtran/project-based-learning |
| 43 | 30-seconds/30-seconds-of-code |
| 44 | electronicarts/CnC_Remastered_Collection |
| 45 | excalidraw/excalidraw |
| 46 | beurtschipper/Depix |
| 47 | 3b1b/manim |
| 48 | airbnb/javascript |
| 49 | tannerlinsley/react-query |
| 50 | lydiahallie/javascript-questions |