FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Technologie a postupy podporující vývoj udržitelného softwaru
Student:	Bc. Michal Pilař
Vedoucí:	Ing. Adam Vesecký
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

V rámci diplomové práce prozkoumejte a popište přístupy napomáhající vyvíjet udržitelný software – tedy software, od kterého očekáváme až desítky let produkčního provozu a jehož komplexita v průběhu let roste. Zaměřte se primárně na technické faktory – používané návrhové vzory, osvědčené postupy při psaní kódu a nástroje určené ke správě zdrojového kódu. Dále vytvořte prototyp aplikace, na kterém předvedete dříve popsané postupy a technologie.

- Popište pojem CI/CD pipeline. Následně prozkoumejte nástroje a platformy používané v rámci této pipeline. Zaměřte se na nástroje zajišťující kontrolu kvality, verzování a správu zdrojového kódu
- Popište technické faktory napomáhající vyvíjet udržitelný software – konkrétní návrhové vzory a základní principy při psaní kódu, jejichž cílem je minimalizovat vznik technického dluhu (SOLID, KISS, DRY, ...).
- Vytvořte prototyp aplikace, na kterém předvedete dříve popsané postupy a využití zmíněných nástrojů a technologií. Na prototypu ukažte praktické příklady použití zmíněných návrhových vzorů. Na základě předchozí analýzy zvolte vhodné nástroje a s jejich použitím předveďte sestavení kompletní CI/CD pipeline. - Vytvořený prototyp náležitě otestujte.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 18. listopadu 2023 v Praze.

Diplomová práce

TECHNOLOGIE A POSTUPY PODPORUJÍCÍ VÝVOJ UDRŽITELNÉHO SOFTWARU

Bc. Michal Pilař

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Adam Vesecký
8. května 2024

Odkaz na tuto práci: Pilař Michal. *Technologie a postupy podporující vývoj udržitelného softwaru*. Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat především své rodině, svým blízkým a své přítelkyni za podporu během studia. Dále bych rád poděkoval vedoucímu této práce Ing. Adamu Veseckému za velmi cenou zpětnou vazbu.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 8. května 2024

Abstrakt

Vývoj softwaru je disciplína, která je tu s námi již desítky let, a za tu dobu prošla velkým množstvím změn. Dnes již existují standardizované postupy napříč celým odvětvím, které nám napomáhají vyvíjet software co nejefektivněji. I tak se bohužel stává, že software během svého vývoje zastarává, hromadí se na něm technický dluh a časem se stává neudržitelným. Jaké tedy použít postupy a technologie při vývoji softwaru, aby byl jeho vývoj a provoz udržitelný klidně i desítky let? Tato práce je zaměřena právě na prozkoumání a popis takových postupů a technologií, které pomáhají softwarovým inženýrům minimalizovat zastarávání zdrojového kódu a vznik technického dluhu.

Klíčová slova software, vývoj, technický dluh, návrhové vzory, CI/CD pipeline, testování, udržitelnost, programování, softwarové inženýrství, principy, nástroje, kontejnerizace

Abstract

Software development is a discipline that has been around for decades and has evolved a lot in that time. Today, there are many standards across the industry that help us develop software as efficiently as possible, but it is often not enough. Unfortunately, it still happens that software becomes obsolete during its development, accumulating technical debt and becoming unsustainable over time. What practices and technologies should be used in software development to ensure that its development and maintenance will be sustainable for decades? This thesis focuses on exploring and describing such practices and technologies that help software engineers minimize source code obsolescence and the accumulation of technical debt.

Keywords software, development, technical debt, design patterns, CI/CD pipeline, testing, sustainability, programming, software engineering, principles, tools, containerization

Seznam zkratek

SRP	Single-Responsibility Principle
OCP	Open/Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Inversion Principle
KISS	Keep It Simple, Stupid
DRY	Don't Repeat Yourself
YAGNI	You Ain't Gonna Need It
DI	Dependency Injection
EDA	Event-Driven Architecture
VCS	Version Control System
CI	Continuous Integration
CD	Continuous Delivery
AWS	Amazon Wen Services
GCP	Google Cloud Platform
QA	Quality Assurance
BDD	Behavior Driven Development
TDD	Test Driven Development

Úvod

Vývoj softwaru je odvětví, které si za dobu své existence prošlo spoustou zásadních změn. Od dob, kdy bylo programování výsadou hrstky lidí, které si většinová společnost představovala jako podivíny sedící po tmě v garáži, až do dnešního dne, kdy je programování a vývoj softwaru obecně jedním z klíčových odvětví technologického pokroku. Začlenit do svého firemního prostředí nějaký informační systém se postupem času stalo nutností pro většinu existujících odvětví, jako je bankovnictví, telekomunikace, stavebnictví, zemědělství a mnoho dalších. Díky informačním systémům se vyřešila spousta problémů, usnadnily, či se plně automatizovaly jednoduché a repetitivní úkoly, usnadnila se správa, analýza a archivace dat a spoustu dalšího. S novou technologií ovšem vznikly i nové problémy a nástrahy, se kterými se nynější podniky musí umět vypořádat.

Jednou z variant, jak rozšířit svůj podnik o informační systém, je zadat jeho vývoj společnosti zaměřené přímo na vývoj softwaru (takzvaný *software house*) a nechat si doručit tento systém jako hotový produkt. Zároveň s vyvinutým softwarem může daný software house poskytnout i nutné množství vývojářů, kteří se zákaznickou firmou řeší jejich stížnosti a nové požadavky a kteří udržují daný systém v chodu. Druhou variantou je pak možnost vyvíjet a spravovat si informační systémy sami (*in-house*). Nevýhoda je ovšem například v tom, že zodpovědnost za chod informačního systému přechází na vlastníka daného softwaru, tedy v tomto případě na podnik, který daný systém i používá. V takové společnosti tudíž musí být zaměstnanci, kteří se o chod informačního systému sami starají a rozumí mu. U robustních systémů, které se často udržují v provozu jednotky až desítky let pak jistě nastane situace, kdy bude komplexnost tohoto systému růst a s tím i náklady na jeho údržbu. V kombinaci s technickým dluhem, který během let na systému vzniká nejen vlivem časté fluktuace vývojářů, ale také volbou technologií, které časem zastarávají, může být jak časová, tak finanční nákladnost údržby takového systému neúnosná.

Jediným řešením výše popsané situace bývá problémový systém nahradit systémem zcela novým. To může být ovšem extrémně nákladné a v některých situacích i nemožné, čímž se daná společnost dostane do patové situace. Efektivním řešením je tedy takové situaci předcházet a snažit se software od počátku vyvíjet tak, aby byl jeho vývoj udržitelný po celou dobu jeho provozu. Toho můžeme docílit zavedením a dodržováním nejrůznějších procesů, napomáhajících vyvíjet udržitelný software, jako je například zavedení efektivního sdílení znalostí, psaní kvalitní dokumentace a mnoho dalších. V rámci těchto procesů můžeme pak používat celou řadu nástrojů, například ke kontrole kvality kódu nebo k efektivní správě zdrojového kódu. Důležité pro zvýšení efektivity a předcházení zastarávání softwaru je také dodržování nejrůznějších standardů, pravidel a předpisů, na kterých se dané týmy dohodnou.

Technický dluh je téměř vždy nevyhnutelný, ale i tak by se měli softwaroví inženýři pracující na společném projektu snažit mu co nejvíce předcházet a ideálně ho s postupem času i snižovat. Tato práce popisuje jak jednotlivé procesy, tak nástroje, které jim k tomu mohou být nápomocny.

Cíl práce

Tato diplomová práce má za cíl prozkoumat technologie a postupy podporující vývoj udržitelného softwaru. Cílem je zaměřit se jak na technické faktory související s programováním, například na používané návrhové a architektonické vzory nebo principy dobrého návrhu, tak na osvědčené postupy a procesy používané při vývoji softwaru. Zároveň je cílem této práce popsat nástroje používané právě v rámci dříve zmíněných procesů, kterými mohou být například nástroje určené ke správě zdrojového kódu, ke kontrole kvality kódu nebo k automatizaci nejrůznějších činností. U jednotlivých nástrojů budou popsány jejich výhody a nevýhody a následně budou vybrány takové nástroje, které budou použity v rámci praktické části této práce – tedy v rámci vytvoření prototypu aplikace a automatizace procesu dodávání dané aplikace. Na vytvořeném prototypu budou zároveň předvedeny některé zmíněné návrhové či architektonické vzory a další popsané principy.

Cílem práce není vyjmenovat a popsat veškeré existující přístupy, procesy a nástroje nebo vytvořit vyčerpávající seznam návrhových vzorů, ale spíše poskytnout základní přehled, který bude sloužit jako úvod do problematiky softwarového inženýrství. Tato práce by měla sloužit jako návod primárně pro juniorní softwarové inženýry, které může čekat nástup do jejich prvního zaměstnání a mohli by se cítit ztraceni ve změti pojmů, procesů a nástrojů používaných při vývoji softwaru ve větších týmech.

Kapitola 1
Analýza

Tato kapitola se věnuje teoretické analýze a popisu základních postupů, procesů a nástrojů, které napomáhají týmům softwarových inženýrů vyvíjet udržitelný software a snížit riziko vzniku technického dluhu. Nejprve jsou zde popsány základní pojmy a myšlenky, které jsou klíčové k pochopení tohoto textu. Následuje popis aspektů vztahujících se primárně k programování, tedy konkrétně popis principů dobrého návrhu a návrhových a architektonických vzorů. Zbytek této kapitoly se již věnuje právě procesům a poté nástrojům softwarového inženýrství.

1.1 Základní pojmy a myšlenky

V této části jsou definovány základní pojmy a myšlenky týkající se vývoje softwaru obecně. Pravidlem bývá, že nové pojmy, myšlenky, procesy a celkově standardy týkající se tohoto řemesla, přicházejí od největších světových společností, zaměřených na vývoj softwaru. Jednou z takových společností je samozřejmě *Google* a o tom, jak se zde vyvíjí software, mluví kniha *Software Engineering at Google: Lessons Learned from Programming Over Time* [1].

1.1.1 Softwarové inženýrství

Úvodním pojmem této části je právě pojem „*Softwarové inženýrství*". Někdo ho chápe čistě jako ekvivalent pojmu „*Programování*"a představí si pod ním pouze psaní kódu, ovšem autoři zmíněné knihy *Software Engineering at Google* [1] chápou tento pojem mnohem komplexněji a definují ho následovně:

▶ **Definice 1.1** (Softwarové inženýrství)**.** *Softwarové inženýrství je programování v průběhu času. Nejedná se tedy pouze o činnost psaní kódu, ale o kompletní souhrn procesů a nástrojů používaných v organizaci k vytváření a udržování kódu v čase.*

Zatímco pojem programování tedy chápeme jako samotný proces psaní kódu, softwarové inženýrství je kompletní sada procesů, které s programováním úzce souvisí a které si musí vytvořit každá organizace, jejíž cílem je nějaký software vyvíjet. Tyto procesy a používané nástroje se napříč organizacemi mohou lišit, ovšem každý vývojář softwaru by měl mít alespoň obecný přehled o tom, s jakými prvky softwarového inženýrství se může v praxi setkat.

1.1.2 Udržitelnost softwaru

Dalším pojmem, klíčovým pro téma této práce, je „*Udržitelnost softwaru*". Očekávaná životnost vytvořeného softwaru se liší projekt od projektu. Existují projekty, jejichž životnost může být

v jednotkách týdnů – typickým příkladem jsou mobilní aplikace. Ty mají často velmi malou životnost a je běžné, že namísto údržby se daná aplikace celá od nuly přepíše. Dalším příkladem mohou být aplikace určené k propagaci nějaké události, například konference zaměřené na vývoj v určitém jazyce. Taková aplikace se navrhuje a vyvíjí pro konkrétní účel a jakmile je daná událost minulostí, aplikace ztrácí význam. V takových případech nemá samozřejmě smysl se zabývat udržitelností a je zbytečné podnikat kroky, abychom minimalizovali vznik technického dluhu. Na druhé straně ovšem existují projekty, které mohou mít očekávanou životnost klidně desítky let. Často se jedná o komplexní ekosystémy, jejichž komponenty jsou závislé jak samy na sobě, tak na externích vlivech, například knihovnách třetích stran. Příkladem projektů, u kterých máme očekávanou životnost dokonce takřka neomezenou, může být *Google Search* nebo *Linux kernel*. U takovýchto projektů se již udržitelností musíme nutně zabývat, abychom nedošli do bodu, kdy nebudeme schopni reagovat na jednotlivé změny. *Titus Winters*, jeden z autorů zmíněné knihy *Software Engineering at Google* [1], definuje pojem *Udržitelnost softwaru* takto:

▶ **Definice 1.2** (Udržitelnost softwaru)**.** *Software (projekt) je udržitelný, pokud jsme schopni po celou očekávanou dobu životnosti tohoto softwaru (projektu) reagovat na veškeré důležité změny, které v průběhu času přicházejí, ať už z technických či businessových důvodů.*

Pokud tedy pracujeme na projektu s nízkou životností, jakým může být například *python* script, který nám má usnadnit jednorázový úkol a který tedy spustíme pouze jednou, nemá smysl se v polovině psaní tohoto scriptu začít zabývat tím, že bychom měli aktualizovat na novou verzi nějaké konkrétní knihovny. Tento script po dokončení spustíme a následně zahodíme. Na druhou stranu, pokud vytváříme tak komplexní nástroj s neomezenou životností, jakým může být zmíněný *Google Search*, určitě by byl velký problém, kdybychom ho provozovali na operačním systému z roku 1990, popřípadě pokud bychom v rámci tohoto projektu používali nějaké zastaralé a dále nepodporované knihovny. *S rostoucí očekávanou živovtností systému tedy roste důležitost postupných aktualizací tohoto systému.*

1.1.3 Codebase

Dalším pojmem, který je důležité znát pro pochopení tohoto textu je pojem *codebase*, neboli kódová základna. *Robert Sheldon* definuje tento pojem ve svém článku *What is a codebase* [2] následovně:

▶ **Definice 1.3** (Codebase)**.** *Codebase, neboli kódová základna je kompletní sada všech zdrojových kódů softwarového projektu. Obsahuje veškeré zdrojové soubory, včetně konfiguračních souborů, potřebné ke kompilaci softwaru do strojového kódu.*

Codebase dále dělíme na monolitickou a distribuovanou. Kódová základna je monolitická, pokud je celá v jediném repozitáři (definice tohoto pojmu se nachází v následující části 1.1.4), který obsahuje veškeré softwarové komponenty. Tento přístup sice zajistí, že máme jediné místo, kde nalezneme vše potřebné, ale převažují spíše nevýhody, jako je nepřehlednost a časem rostoucí složitost. Distribuovaná codebase je oproti monolitické rozdělená do více repozitářů, podle jednotlivých komponent softwarového projektu. Díky tomu je codebase přehlednější a výsledný kód bývá snadnější dodávat na cílové platformy. Problémem u distribuované kódové základny může být složitější správa závislostí mezi jednotlivými komponentami.

1.1.4 Software repozitář

Často skloňovaným pojmem související se správou kódové základny je pojem *software repozitář* nebo jen *repozitář* (neplést s návrhovým vzorem *Repository*, který je popsán v sekci 1.3.3). *Vikram Gupta* definuje tento pojem ve svém článku *What IsRepository?* [3] takto:

▶ **Definice 1.4** (Software repozitář)**.** *Software repozitář je typ centrálně umístěného úložiště, kde můžeme uchovávat všechny soubory a zdroje svého projektu.*

Zjednodušeně je repozitář tedy místo, kam ukládáme naší codebase. Kterýkoliv ze zúčastněných subjektů nebo vývojářů projektu si pak z tohoto místa může kód (nebo jiný zdroj) stáhnout a následně navrhovat změny. Klíčové funkcionality a koncepty softwarových repozitářů jsou následující:

1. *Branch*. Jinak také nazývána *vývojová větev*, označuje *branch* samostatnou vývojovou linii, která má zpočátku stejnou kódovou základnu jako hlavní úložiště.
2. *Commit*. Jedná se o funkcionalitu, která uloží provedené změny v souborech projektu do samotného úložiště.
3. *Pull*. Tato funkcionalita reprezentuje stažení zdrojového kódu ze vzdáleného úložiště do počítače daného vývojáře.
4. *Push*. Funkcionalita *push* provádí nahrání lokálních změn do vzdáleného úložiště.
5. *Pull request*. Jedná se o mechanismus pro vytvoření požadavku na zanesení změn z jedné vývojové větve (tradičně té, ve které vývojář implementoval změny) do jiné větve (často větve hlavní, ve které je uložená hlavní codebase).

Repozitáře si můžeme vytvářet s použitím nástrojů pro verzování a správu zdrojového kódu, známých jako *VCS* nástroje, o kterých hovoří část 1.6.1. Každý takový nástroj by měl poskytovat výše popsané funkcionality a mechanismy, ovšem mohou zde být rozdíly v tom, jak jednotlivé nástroje fungují.

1.2 Principy dobrého návrhu

Ač se nám nejspíš nikdy nepodaří kompletně eliminovat veškeré hrozby, které na nás při vývoji softwaru čekají, můžeme se alespoň pokusit dopad těchto hrozeb minimalizovat. K tomu nám mimo jiné může pomoci, když budeme dodržovat několik základních principů, jak navrhovat a psát kvalitní kód. V této části samozřejmě nejsou obsaženy všechny existující principy dobrého návrhu, neboť jich existuje celá řada a všechny popsat by bylo nereálné. Jsou zde tedy popsány primárně ty, které by měl znát každý vývojář softwaru.

1.2.1 SOLID principy

Mezi často skloňované principy objektově orientovaného návrhu patří principy, které jsou souhrnně pojmenované zkratkou *SOLID*. Těmito principy jsou *Single responsibility* principle, *Open/Closed* principle, *Liskov substitution* principle, *Interface segregation* principle a *Dependency inversion* principle a všechny již byly popsány v mnoha zdrojích. Jedním z výrazných autorů, který zmíněné principy popisuje, je bezpochyby *Robert C. Martin*, který o těchto principech píše mimo jiné i v knize *Agile Software Development, Principles, Patterns, and Practices* [4]. Následuje podrobnější popis těchto principů.

Single-Responsibility principle

Abychom mohli popsat první ze SOLID principů, musíme definovat pojem *responsibility*, neboli odpovědnost. *Robert C. Martin* definuje tento pojem ve své knize následovně:

▶ **Definice 1.5** (Responsibility)**.** *Responsibility (odpovědnost) v kontextu Single responsibility principu definujeme jako „důvod ke změně“. Pokud existuje více než jeden motiv ke změně třídy, třída má více než jednu odpovědnost.*

Tento princip, nazvaný *Single-Responsibility principle*, zkráceně *SRP* nám říká, že každá třída by měla mít pouze jednu odpovědnost. V případě, kdy existuje třída, která má na starosti více věcí, máme více důvodů danou třídu měnit. Zároveň nám mezi dvěma nezávislými odpovědnostmi vzniká určitá svázanost. Když chceme pak změnit způsob, jakým třída obstarává jednu odpovědnost, můžeme nechtěně změnit chování při obstarávání odpovědnosti druhé – tento druh provázanosti vede tedy k návrhu, ve kterém může při jakékoliv změně vzniknout chyba v oblasti, kde bychom to vůbec nečekali.

Open/Closed principle

Další princip, který výrazně napomáhá při vývoji systému, od kterého chceme, aby vydržel déle než pouze s první verzí, je princip nazvaný *Open/Closed principle*, zkráceně *OCP*. Tento princip nám říká, že moduly, které ho dodržují, mají dvě klíčové vlastnosti:

1. *Otevřenost k rozšíření.* To znamená, že chování a vlastnosti daného modulu mohou být rozšířeny. Tak, jak se postupem času mění požadavky na aplikace, chceme, abychom byli schopni rozšířit modul o nové vlastnosti a uspokojili tak tyto požadavky.

2. *Uzavřenost k modifikacím.* Rozšíření modulu o nové vlastnosti by nemělo vyústit ve změnu vlastností stávajících. Rozšiřování vlastností modulu nesmí mít za následek změnu existujícího zdrojového nebo binárního kódu daného modulu.

Na první pohled se může zdát, že jsou výše zmíněné vlastnosti protichůdné. Tradiční způsob, jak rozšířit vlastnosti nějakého modulu, je právě změnou zdrojového kódu. Modul, který nemůže být změněn je většinou brán jako modul s fixními vlastnostmi. Jak tedy docílit toho, že rozšíříme modul o nové vlastnosti, aniž bychom zasáhli do zdrojového kódu? Odpovědí na tuto otázku je použití abstrakce.

Liskov substitution principle

Třetí ze SOLID principů je *Liskov substitution principle*, zkráceně *LSP*. Tento princip říká, že rodičovské typy musí být nahraditelné svými potomky. Pokud máme například dvě třídy, z nichž první, rodičovská třída *obdélník*, má jako potomka třídu *čtverec*, měli bychom být schopni nahradit všechny výskyty třídy *obdélník* třídou *čtverec*, aniž bychom rozbili výsledné chování aplikace, protože *čtverec* je *obdélník*, který má navíc další specifické vlastnosti.

Interface segregation principle

Interface segregation principle, zkráceně *ISP*, hovoří o nevýhodně složitých rozhraní. Třídy, které mají složité rozhraní, jsou třídy, u kterých hrozí, že jejich rozhraní nebude koherentní (soudržné). Tento princip sice připouští, že některé objekty vyžadují nekoherentní rozhraní, ovšem v takovém případě by se o nich klienti neměli dozvědět jako o jediné třídě, ale měli by znát základní abstraktní třídy, které koherentní rozhraní mají.

Dependency inversion principle

Poslední ze SOLID principů zvaný *Dependency inversion principle*, zkráceně *DIP*, hovoří o tom, že vysoko-úrovňové moduly by neměly záviset na nízko-úrovňových modulech. Uvažujme vysoko-úrovňové modely, které obsahují důležitou business logiku – v těchto modulech je obsažena identita samotné aplikace. Pokud jsou ovšem tyto moduly závislé na nízko-úrovňových modulech, změna v těchto nízko-úrovňových modulech může mít přímý dopad i do vysoko-úrovňových modulů a může tím pádem vynucovat změnu i zde, což je nežádoucí. Jsou to totiž právě vysoko-úrovňové moduly, které by měly ovlivňovat veškeré moduly nižší úrovně, ne obráceně.

1.2.2 Separation of concerns

Často zmiňovaným principem je bezpochyby princip *Separation of concerns*. *Tamerlan Gudabayev* ve svém článku *Separation of Concerns The Simple Way* [5] mluví o tomto principu jako o univerzálním principu, který používají téměř všichni napříč mnoha odvětvími – tedy nejen programátoři a softwaroví inženýři. Pro snadnější pochopení tohoto principu je nutné definovat pojmy *cohesion* neboli *soudržnost* a *coupling* neboli *provázanost*.

Cohesion (soudržnost)

Prvním definovaným pojmem je tedy *cohesion*, neboli soudržnost. *Tamerlan Gudabayev* definuje tento pojem následovně:

▶ **Definice 1.6** (Cohesion). *Cohesion (soudržnost) je v kontextu Separation of concerns principu měřítkem toho, jak spolu souvisí konkrétní skupina věcí. V informatice jde o to, jak silný je vztah mezi metodami a vlastnostmi třídy.*

Jak je řečeno v definici, *cohesion* je měřítkem toho, jak spolu souvisí konkrétní skupina věcí. *Tamerlan Gudabayev* vysvětluje tento pojem na příkladu s příborníkem. Příborník máme rozdělený do přihrádek a do každé přihrádky typicky dáváme jeden typ příboru – lžičky dáváme se lžičkami, vidličky s vidličkami atd. Nejvíc spolu tedy souvisí konkrétní typy příboru, ale všechen příbor má taky hodně společného – je vyskládán v jednom příborníku.

Výpis kódu 1.1 Cohesion příklad – Funkce pro vykreslování geometrických tvarů

```
def draw_circle():
    # draw a~circle code

def draw_rectangle():
    # draw a~rectangle code
```

Konkrétní příklad vztahující se k informatice je nastíněn v ukázce kódu 1.1. Zde máme dvě funkce, kdy jedna má na starosti vykreslit kruh, zatímco úkol druhé funkce je vykreslit obdélník. O těchto funkcích můžeme prohlásit, že mají vysokou *cohesion*, neboť jsou dost podobné na to, abychom je mohli umístit do stejného modulu nebo třídy, jejíž odpovědností je vykreslování. Příklad takové třídy je vyobrazen v ukázce kódu 1.2.

Výpis kódu 1.2 Cohesion příklad – Třída pro vykreslování geometrických tvarů

```
class Draw:
    def drawCircle(self):
        # draw a~circle code

    def drawRectangle(self):
        # draw a~rectangle code
```

Jedná se samozřejmě o jednoduchý příklad určený k pochopení daného konceptu. Při vývoji komplexního softwaru by mohla nastat situace, kdy bychom z určitých důvodů museli takto podobné funkce rozdělit do nezávislých tříd.

Coupling (provázanost)

Ač by mohl pojem *coupling*, neboli *provázanost* zdánlivě znamenat to samé, co pojem *cohesion*, jeho význam je v kontextu tohoto principu zcela odlišný. *Tamerlan Gudabayev* definuje význam tohoto pojmu takto:

▶ **Definice 1.7** (Coupling). *Coupling (provázanost) je v kontextu Separation of concerns principu míra závislosti mezi dvěma nebo více třídami, moduly nebo komponentami.*

Shrnutí principu Separation of concerns

Obecně pak platí, že bychom se měli snažit psát kód, který je vysoce soudržný (*highly cohesive*) a je málo provázaný (má *low coupling*). Dodržování tohoto pravidla a tím pádem i *Separation of concerns* principu nám přináší následující výhody:

1. *Lepší přehlednost kódu.* Je mnohem snazší pochopit, co se děje, když je vše přehledně uspořádáno.
2. *Lepší znovupoužitelnost kódu.* Při opakovaném použití kódu jsou náklady na jeho údržbu nižší. Zároveň je mnohem snazší kód rozšířit nebo opravit chybu, protože kód vztahující se k dané chybě nebo rozšíření se nachází na jediném místě.
3. *Lepší testovatelnost.* Když je vše přehledně izolováno, je mnohem jednodušší testovat naší aplikaci. Není potřeba složitě nastavovat celé testovací prostředí, ale stačí mockovat[1] sousední moduly s fiktivními daty.
4. *Rychlejší vývoj.* Izolování modulů pomáhá při vytváření a aktualizaci nových funkcích.
5. *Lepší organizace.* Pokud máme vše přehledně odděleno, mají vývojáři mnohem příjemnější zkušenost s vývojem – mohou se například dohodnout, na kterých modulech budou pracovat a díky tomu si nebudou navzájem překážet.

1.2.3 Další principy

Výše popsané principy samozřejmě nejsou jediné existující principy podporující dobrý návrh a psaní kvalitního kódu. V této části jsou shrnuty další, neméně důležité principy, které by měl znát každý softwarový inženýr.

The Boy Scout Rule

O tomto principu mluví *Robert C. Martin* ve své knize *Clean Code: A Handbook of Agile Software Craftsmanship* [6]. *The Boy Scout Rule* nám říká, že nestačí psát kvalitní kód. Kód musí být v čase udržován co nejčistší, a proto musíme podnikat aktivní kroky, abychom zabránili postupné degradaci kódu. Tento princip je inspirován americkými skauty, kteří mají jednoduché pravidlo, které se dá aplikovat i na profesi softwarových inženýrů:

„Zanechte tábořiště čistší, než jste ho našli.“

Pokud tedy náš kód lehce vyčistíme při každé změně, kterou v něm děláme, zabráníme tak postupnému zastarávání a degradaci kódu. Čistění kódu nemusí být ani nijak obsáhlé. Stačí jednoduché kroky, jako je přejmenování proměnné tak, aby byl její název více vypovídající, rozdělení obsáhle metody na dvě jednodušší nebo odmazání duplicitního kódu.

Keep It Simple, Stupid

O principu *Keep It Simple, Stupid*, zkráceně *KISS*, stejně jako o následujících principech, píše *Tarun Telang* ve svém článku *What are YAGNI, DRY and KISS principles in software development?* [7]. Tento princip, jak už jeho název napovídá, mluví o nutnosti nepřidávat zbytečnou komplexitu do našeho kódu. Vše bychom měli dělat co nejjednodušeji. Tento přístup pomáhá udržet kód přehledný a srozumitelný pro ostatní programátory. *KISS* může zároveň zlepšovat výkonnost našeho softwaru, neboť jednoduše napsaný kód bývá i pro počítač často snazší provést.

[1] Mockování je proces v softwarovém vývoji, kdy se vytváří fiktivní verze komponenty nebo modulu, aby se simulovalo jejich chování pro účely testování nebo prototypování.

Don't Repeat Yourself

Tento princip, zkráceně nazývaný *DRY*, říká, že bychom neměli mít napříč námi vytvářeným softwarem duplicitní kód nebo data. Duplicity mohou totiž vést k chybám a zároveň mohou dělat náš kód nepřehledným a složitým k údržbě a k případným rozšířením. Jak již bylo řečeno dříve, k postupnému zbavování se duplicit v kódu může pomáhat dodržování *The Boy Scout Rule*, kdy například v rámci implementace nové funkcionality smažeme duplicity, na které během implementace narazíme.

You Ain't Gonna Need It

Posledním zde zmíněným principem je princip *You Ain't Gonna Need It*, také nazývaný *You Aren't Gonna Need It*, zkráceně *YAGNI*. Tento, při vývoji softwaru často používaný princip, nám říká, že bychom neměli implementovat funkcionality, které aktuálně nepotřebujeme. Často totiž programátoři implementují nepotřebné funkcionality jen kvůli pocitu, že by se v budoucnu mohly použít, a ačkoliv tento pocit není vždy mylný, může toto chování opět dělat náš kód zbytečně složitým. Dodržování tohoto principu naopak pomáhá udržovat kód jednoduchý a více robustní. Díky tomu, že neplýtváme čas implementací zbytečných funkcionalit nám dodržování *YAGNI* principu pomáhá šetřit čas i peníze.

1.3 Návrhové vzory

Ve vývoji softwaru chápeme návrhové vzory jako přepoužitelná řešení běžných problémů, se kterými se vývojáři potkávají. Tyto vzory nám poskytují strukturu potřebnou k organizaci kódu tak, aby byl udržitelný, modulární a škálovatelný. Návrhové vzory mohou být aplikované na jakýkoliv software a nevztahují se k žádnému konkrétnímu programovacímu jazyku nebo platformě.

1.3.1 Tradiční návrhové vzory

Zdrojem, který slouží jako perfektní úvod do problematiky návrhových vzorů, je kniha *Design Patterns: Elements of Reusable Object-Oriented Software* [8]. V této knize jsou popsány veškeré základní návrhové vzory, o kterých by měl mít každý programátor alespoň základní povědomí. Tato část práce se ovšem nevěnuje těmto návrhovým vzorům do detailu, neboť se jedná o myšlenky nesčetněkrát popsané jak ve zmíněné knize, tak v nejrůznějších článcích. Podrobněji zde budou popsány modernější návrhové vzory používané primárně v backend vývoji.

Tyto tradiční návrhové vzory dělíme do tří základních skupin. V následujícím seznamu jsou popsány jednotlivé skupiny a uvedeny příklady návrhových vzorů z těchto skupin. Detailní informace o zmíněných návrhových vzorech lze dohledat ve zmíněné knize.

1. *Creational*. Návrhové vzory v této skupině abstrahují inicializační proces. Napomáhají vytvářet systém nezávislý na tom, jak jsou jednotlivé objekty vytvořeny, poskládány a reprezentovány. Návrhové vzory z této skupiny jsou například *Abstract Factory*, *Builder* nebo *Singleton*.
2. *Structural*. Strukturální návrhové vzory se zaměřují na to, jak jsou třídy a objekty skládány do větších struktur. Tyto návrhové vzory používají dědičnost ke skládání rozhraní nebo implementací do jedné třídy. Strukturální vzory jsou obzvláště užitečné pokud chceme zajistit kompatibilitu nezávisle vytvořených knihoven. Mezi návrhové vzory z této skupiny patří *Adapter*, *Decorator* nebo *Bridge*.
3. *Behavioral*. Behaviorální vzory jsou zaměřeny na algoritmy a přiřazování odpovědností mezi objekty. Nejedná se pouze o vzory objektů nebo tříd, ale také o vzory komunikace mezi objekty a třídami. Návrhové vzory z této skupiny jsou například *Command*, *Observer* nebo *Iterator*.

1.3.2 Návrhové vzory pro moderní backendový vývoj

V moderním backendovém vývoji jsou návrhové vzory klíčové pro vytváření robustních a flexibilních systémů, které se musí přizpůsobovat měnícím se požadavkům a potřebám uživatelů. Některé z těchto návrhových vzorů popisuje *Pacifique Linjanja* ve svém článku *Design Patterns for Modern Backend Development – with Example Use Cases* [9]. Jak tyto návrhové vzory, tak jejich výhody a nevýhody jsou zmíněny v následující části textu.

Návrhové vzory poskytují četné výhody při vývoji softwaru. Mohou zjednodušovat proces psaní kódu, zlepšit udržitelnost kódu a podpořit jeho opakované použití. Pomáhají zároveň vývojářům psát kód, který je více efektivní, škálovatelný a lépe přizpůsobitelný. Návrhové vzory jsou neuvěřitelně přínosné, pokud na projektu pracuje více vývojářů, neboť poskytují sdílený rámec osvědčených postupů, které mohou zajistit konzistenci napříč celou *codebase*.

Výhody používání návrhových vzorů v backendovém vývoji

Jak již bylo zmíněno, návrhové vzory jsou ozkoušená a časem prověřená řešení běžných problémů, se kterými se setkáváme při vývoji softwaru. V této části jsou popsány základní výhody využívání návrhových vzorů nejen v backendovém vývoji.

1. *Přepoužitelnost a konzistence.* Jednou z hlavních výhod používání návrhových vzorů je přepoužitelnost. Díky dodržování standardní struktury mohou vývojáři snadno opakovaně používat kód v různých částech aplikace nebo dokonce v různých aplikacích. Tím zároveň docílíme konzistence napříč kódovou základnou, protože stejný problém bude mít vždy stejné řešení.

2. *Škálovatelnost.* Návrhové vzory umožňují škálovatelnost aplikací, neboť poskytují strukturovaný přístup k psaní kódu. To usnadňuje přidávání nových funkcionalit nebo provádění změn v existujících funkcionalitách, aniž bychom narušili celkovou architekturu aplikace.

3. *Udržitelnost.* Další výhodou návrhových vzorů je, že jejich použití může zlepšit udržitelnost kódu. Návrhové vzory totiž poskytují standardizovaný přístup k řešení problémů, což vývojářům usnadňuje porozumění kódu napsanému ostatními a jeho údržbu v průběhu času.

4. *Snížení počtu chyb.* Návrhové vzory jsou vyzkoušená a otestovaná řešení běžných problémů. Použitím těchto vzorů se vývojáři mohou vyhnout běžným chybám a nástrahám, které často vznikají při psaní kódu od nuly.

5. *Zlepšení výkonu.* Použití návrhových vzorů může zlepšit výkon aplikace díky tomu, že poskytují strukturovaný přístup k řešení problémů. Výsledkem může být efektivnější kód, který se provádí rychleji a využívá méně zdrojů.

V následující části textu jsou již podrobněji popsány konkrétní návrhové vzory, které se využívají nejen v backedovém vývoji. Tyto návrhové vzory jsou mimo jiné popsány ve zmíněném článku *Design Patterns for Modern Backend Development – with Example Use Cases* [9].

1.3.3 Repository

Repository je návrhový vzor, který poskytuje abstraktní vrstvu mezi vrstvou přistupující k datům a zbytkem aplikace. Odděluje logiku, která má na starosti získávání dat, od vrstvy datového úložiště, čímž poskytuje modulárnější přístup k vývoji softwaru. V tomto návrhovém vzoru funguje *Repository* jako prostředník mezi vrstvou přistupující k datům a vrstvou aplikační logiky. Poskytuje jediný přístupový bod pro získání a manipulaci s daty, což umožňuje oddělit zbytek aplikace od konkrétní implementace datové vrstvy. Díky tomu můžeme snadno provádět změny v datové vrstvě, aniž bychom ovlivnili zbytek aplikace.

V základu funguje tedy návrhový vzor *Repository* jako most mezi obchodní logikou a úložištěm dat. Zapouzdřuje logiku přístupu k datům do takzvaných *Repositories*, což jsou třídy zodpovědné za dotazování a manipulaci s daty z podkladového zdroje dat, například databáze nebo webové služby. Tím chrání naší aplikaci před složitostmi datového úložiště a usnadňují přepínání mezi datovými zdroji nebo provádění změn v databázovém schématu bez vlivu na kód aplikace. *Anish Bilas Panta* ve svém článku *Understanding Repository Pattern with Implementation: A Step-by-Step Guide* [10] popisuje tyto základní prvky návrhového vzoru *Repossitory*:

1. *Entity.* Jedná se o jednotlivé datové objekty reprezentující základní datové struktury aplikace.

2. *Rozhraní Repository (IRepository).* Rozhraní definující kontrakt pro interakci s datovými entitami. Typicky obsahuje metody jako *GetById*, *GetAll*, *Add*, *Update* a *Delete* (v některých případech je běžné i použití metody pro nahrání více entit najednou, která se může jmenovat například *AddAll*). Rozhraní abstrahuje podrobnosti o datovém úložišti, čímž dělá výsledný kód nezávislý na tom, kde a jak jsou data uložena.

3. *Konkrétní Repositories.* Toto jsou třídy, které implementují *IRepository* rozhraní pro konkrétní entity.

4. *Datový zdroj.* Jak název napovídá, datový zdroj je místo, kde jsou uložena data. Může to být databáze, webová služba nebo jakékoliv jiné úložiště dat. *Repository* skrývá složitost práce se zdrojem dat.

5. *Klientský kód.* Jedná se o kód aplikace, který interaguje s danými *Repositories*. Volá metody definované v rozhraní *IRepository* aniž by se staral o to, jak jsou data získávána nebo ukládána.

Příklad implementace návrhového vzoru Repository

V této části následuje praktická ukázka implementace návrhového vzoru *Repository*. *Anish Bilas Panta* má ve svém článku [10] veškeré výpisy kódu v jazyce *C#*, ovšem v zájmu zachování konzistence s předchozími výpisy kódu byly veškeré ukázky přepsány do jazyka *Python*.

Ve výpisu kódu 1.3 máme definovanou entitu *User*. Jedná se o jednoduchý datový objekt, se kterým v rámci aplikace pracujeme a ke kterému chceme pomocí *Repository* přistupovat. Tato entita obsahuje základní atributy jako jsou identifikátor, uživatelské jméno a email, které jsou nezbytné pro manipulaci s uživatelskými daty.

Výpis kódu 1.3 Repository – Ukázka datové entity

```
class User:
    def __init__(self, id, username, email):
        self.id = id
        self.username = username
        self.email = email
        # Other properties
```

Dalším prvkem, který byl dříve popsán a jehož implementace je vyobrazena ve výpisu kódu 1.4 je rozhraní, respektive abstraktní třída *IRepository* (Jazyk *Python* neumožňuje vytvářet rozhraní, která známe například z jazyků *C#* nebo *Java*, tudíž je zde potřeba definovat abstraktní třídu). Tato třída definuje kontrakt pro interakci s datovými entitami – jsou zde metody typické pro rozhraní *Repository*, konkrétně *GetById* zajišťující přístup k entitě pomocí identifikátoru, *GetAll* vracející všechny entity, *Add* pro přidání entity, *Update* pro aktualizaci a *Delete* pro smazání entity. Následně je nutné vytvořit implementaci této abstraktní třídy pro každou datovou entitu, se kterou chceme pracovat.

Výpis kódu 1.4 Repository – Ukázka implementace rozhraní *IRepository*

```
from abc import ABC, abstractmethod

class IRepository(ABC):
    @abstractmethod
    def GetById(self, id):
        pass

    @abstractmethod
    def GetAll(self):
        pass

    @abstractmethod
    def Add(self, entity):
        pass

    @abstractmethod
    def Update(self, entity):
        pass

    @abstractmethod
    def Delete(self, entity):
        pass
```

Třída *UserRepository* z výpisu kódu 1.5 následně implementuje rozhraní, respektive abstraktní třídu *IRepository* pro datovou entitu *User*. Tato třída už musí obsahovat logiku, jak s danou datovou entitou zacházet, tedy jak provádět dotazy nad konkrétním datovým zdrojem.

Výpis kódu 1.5 Repository – Ukázka implementace konkrétní *Repository*

```
class UserRepository(IRepository):
    def __init__(self):
        self._users = []

    def GetById(self, id):
        return next((u for u~in self._users if u.Id == id), None)

    def GetAll(self):
        return self._users

    def Add(self, entity):
        self._users.append(entity)

    def Update(self, entity):
        # Implement update logic here
        pass

    def Delete(self, entity):
        self._users.remove(entity)
```

Ve finálním výpisu kódu 1.6 je ukázka toho, jak je možné *Repository* použít. Po vytvoření instance *UserRepository* můžeme provádět různé operace s uživateli, jako je přidání uživatele, aktualizace uživatele nebo jeho smazání. Dále můžeme získat všechny uživatele z databáze nebo vyhledat konkrétního uživatele pomocí jeho identifikátoru. Všechny tyto operace je možné provádět, aniž bychom se museli starat o to, jaké datové úložiště se používá nebo jak jsou implementované operace pro manipulaci s daty. Uvnitř třídy *UserRepository* bychom zároveň mohli vyměnit datové úložiště za jiné, aniž by bylo nutné zasahovat do kódu s business logikou.

Výpis kódu 1.6 Repository – Použití *UserRepository*

```
class Program:
    @staticmethod
    def Main():
        userRepository = UserRepository()

        # Create a new user
        newUser = User()
        newUser.Id = 1
        newUser.Username = "john_doe"
        newUser.Email = "john@example.com"
        userRepository.Add(newUser)

        # Retrieve a user by ID
        retrievedUser = userRepository.GetById(1)

        # Update the user
        retrievedUser.Username = "johndoe"
        userRepository.Update(retrievedUser)

        # Delete the user
        userRepository.Delete(retrievedUser)

        # Get all users
        allUsers = userRepository.GetAll()
```

Výhody a nevýhody používání návrhového vzoru Repository

Než se rozhodneme implementovat v naší aplikaci návrhový vzor *Repository*, je důležité uvědomit si jeho klady a zápory. Tento vzor totiž přináší výhody v podobě abstrahování připojení k datovému úložišti, ale v některých případech může učinit náš software zbytečně složitým. Konkrétní výhody a nevýhody tohoto návrhového vzoru jsou následující:

Výhody

1. *Abstrakce.* Tento návrhový vzor poskytuje abstraktní rozhraní pro přístup k datům, čímž umožňuje měnit datový zdroj, aniž bychom museli zasahovat do businessové logiky.

2. *Dodržení principu Separation of concerns.* Repository jasně odděluje kód, který zajišťuje přístup k datům, od aplikační logiky, díky čemuž je codebase mnohem udržitelnější.

3. *Testovatelnost.* Díky tomu, že používáme abstraktní rozhraní, můžeme snadno mockovat *Repositories* za účelem unit testování. K tomu se může výborně hodit návrhový vzor *dependency injection*, jehož popis následuje.

4. *Flexibilita.* Je možné vytvářet vlastní *Repositories* pro specifické datové entity a přizpůsobit jejich metody potřebám naší aplikace. Můžeme se tedy rozhodnout, zda uživatelům těchto *repositories* například umožníme mazat datové entity nebo je naopak přidávat.

5. *Konzistence.* Návrhový vzor Repository prosazuje konzistentní vzory a konvence přístupu k datům v celé aplikaci. Díky tomu je pak snazší pro inženýry angažované na projektu vytvořené *repositories* používat.

Nevýhody
Ač je Repository mocným návrhovým vzorem, který nám může výrazně usnadnit implementaci přístupu k datům v našich aplikacích, je potřeba mít povědomí i o nevýhodách tohoto vzoru. Díky tomu se pak můžeme vyvarovat jeho použití v případech, kdy by to mohlo být pro vývoj naší aplikace kontraproduktivní. *Pranaya Rout*, autor článku *Repository design pattern in C#* [11] popisuje následující nevýhody tohoto vzoru:

1. *Složitost.* Implementace vzoru *Repository* může do naší codebase vnést určitou složitost, zejména pokud je naše aplikace jednoduchá. Pro jednoduché projekty může být tento návrhový vzor přehnaně náročný.
2. *Výkonnostní zátěž.* Neefektivní použití vzoru (například načítání celých datových sad pro několik polí) může vést k výkonnostním potížím.
3. *Omezená flexibilita vytváření dotazů.* Obecný vzor *Repository* do jisté míry omezuje vytváření složitých dotazů, což může vést k nedostatečnému nebo nadměrnému načítání dat.
4. *Přílišná abstrakce.* V některých případech vede tento návrhový vzor k přílišné abstrakci, kdy je vrstva přístupu k datům natolik abstrahovaná, že je obtížné optimalizovat konkrétní dotazy nebo využívat určité vlastnosti databáze.

Shrnutí návrhového vzoru Repository

Návrhový vzor *Repository* je mocný nástroj pro správu přístupu k datům v aplikacích. Zapouzdřením logiky přístupu k datům do jednotlivých *Repositories* lze dosáhnout větší flexibility, udržitelnosti a testovatelnosti. Jedná se o jeden ze základních vzorů, které je potřeba brát v úvahu při návrhu softwaru.

1.3.4 Dependency Injection

Dalším návrhovým vzorem, používaným nejen v moderním backendovém vývoji, je vzor nazývaný *Dependency Injection*, zkráceně *DI*. Tento návrhový vzor umožňuje vytvářet volně vázané (*loosely coupled*) softwarové komponenty. Používá se ke snížení provázanosti (*coupling*) mezi komponentami a ke zlepšení flexibility, testovatelnosti a udržitelnosti kódu.

V případě vzoru Dependency Injection jsou závislosti do komponenty vkládány, nikoli vytvářeny v rámci komponenty. To umožňuje vytvářet komponenty nehledě na jejich závislosti, což usnadňuje nahrazení nebo úpravu závislostí bez vlivu na samotnou komponentu. Abychom mohli pochopit Dependency Injection, musíme nejdřív pochopit pojem *dependency*, neboli závislost. Pojem *dependency* můžeme definovat následovně:

▶ **Definice 1.8** (Dependency)**.** *Když třída A používá nějaké funkcionality třídy B, říkáme, že třída A je dependentní (závislá) na třídě B.*

Nyní může následovat samotné vysvětlení tohoto návrhového vzoru. *Bhavya Karia* ve svém článku *A quick intro to Dependency Injection: what it is, and when to use it* [12] dává příklad programování v jazyce *Java*, kde musíme před použitím metod jiných tříd nejprve vytvořit objekt této třídy (tj. třída *A* musí vytvořit instanci třídy *B*). *Dependency Injection* pak nazýváme situaci, kdy přeneseme úlohu vytvoření objektu na někoho jiného (třída *A* si tedy nevytváří instanci třídy *B*, ale požádá třídu, která má na starosti vytváření instancí, o dodání instance třídy *B*).

Proč použít Dependency Injectiton

Bhavya Karia [12] dává příklad na modelové situaci. Představme si, že máme třídu *Car* (automobil), která obsahuje několik různých objektů, jako jsou kola, motor, atd. Třída *Car* je zodpovědná

za vytváření veškerých objektů, na kterých je závislá. Co se ale stane, pokud chceme nahradit aktuální kola koly jiné značky?

Bez *DI* bychom museli předělat třídu *Car* a všude nahradit stávající značku kol koly novými. Pokud ale použijeme *DI*, můžeme měnit kola za běhu, protože závislosti mohou být vkládány za běhu programu, na rozdíl od varianty bez *DI*, kde musíme závislosti mít jasné už během kompilace programu. Můžeme tedy nad *DI* přemýšlet jako nad prostředníkem v našem kódu, který dělá veškerou práci při vytváření objektu preferovaných kol a poskytování těchto kol třídě *Car*. Díky tomu je třída *Car* nezávislá na vytváření objektů, které tato třída obsahuje.

Existující typy Dependency Injection

Existuje několik typů implementace návrhového vzoru Dependency Injection. Tyto typy, rozlišované podle toho, jakým způsobem jsou závislosti předávány, jsou následující:

1. *Constructor Injection*. Závislosti jsou poskytovány prostřednictvím konstruktoru třídy.
2. *Setter Injection*. Klient vystaví *setter* metodu, kterou injector použije k předání závislosti.
3. *Interface Injection*. Závislost vystavuje metodu zvanou *injector*, která předá závislost libovolnému klientovi, který jí byl předán. Klienti musí implementovat rozhraní, které vystavuje *setter* metodu. Ta potom přijímá potřebnou závislost.

Odpovědnosti návrhového vzoru Dependency Injection

Prostředník, jehož cílem je poskytovat požadované závislosti, má několik odpovědností. Ty jsou zpravidla následující:

1. Vytváření objektů.
2. Znát třídy, které tyto objekty vyžadují.
3. Poskytnutí všech potřebných objektů třídám, které je vyžadují.

Pokud musí nastat jakákoliv změna v objektech, jedná se o odpovědnost *Dependency Injection* se o tuto změnu postarat a cílové třídy, které se změna týká, by se to nemělo v žádném případě dotknout. Pokud se tedy objekty v budoucnu změní, pak je odpovědností *DI* poskytnout třídě příslušné objekty.

Výhody a nevýhody návrhového vzoru Dependency Injection

Stejně jako každý návrhový vzor má i Dependency Injection své pro a proti. Výhody a nevýhody tohoto návrhového vzoru jsou tyto:

Výhody

1. Dodržení *Dependency inversion* principu ze SOLID principů.
2. Zjednodušení unit testování.
3. Jednodušší rozšiřování aplikace a změny v aplikaci.
4. Podpoření volných vazeb mezi třídami.

Nevýhody

1. Je složitější na naučení, a pokud se používá příliš často, může vést k problémům.
2. Z chyb při kompilaci se stávají chyby za běhu programu.

Shrnutí návrhového vzoru Dependency Injection

Dependency Injection je návrhový vzor užitečný pro vytváření škálovatelných, udržitelných a efektivních systémů. Zmenšuje provázanost mezi komponentami a zlepšuje flexibilitu, testovatelnost a udržitelnost kódu. *DI* můžeme implementovat sami, ale dnes je standardem použít knihovny či frameworky třetích stran. Příkladem takových frameworků může být například *Spring* pro jazyk *Java* nebo framework *Autofac* pro *.NET*.

Zajímavostí tohoto návrhového vzoru je to, že se jeho implementace může výrazně lišit dle použitého programovacího jazyka. Například v jazyce *C++* bude vzhledem k absenci reflexe[2] vypadat implementace tohoto návrhového vzoru jinak (složitěji), než v jiných jazycích – implementaci tohoto návrhového vzoru právě v tomto jazyce se věnuje například *Aliaksei Radzevich* ve svém článku *Compile Time Dependency Injection in C++* [13].

1.3.5 Observer

Další popsaný návrhový vzor se nazývá *Observer*. Tento vzor umožňuje objektům (*pozorovaným*, nazýváme typicky *Subject*) upozornit jiné objekty (pozorovatele, nazýváme typicky *Observer*) na změnu svého stavu. Poskytuje objektům možnost vzájemné komunikace, aniž by o své existenci přímo věděli. V tomto návrhovém vzoru si objekt typu *Subject* udržuje seznam pozorovatelů (objektů typu *Observer*) a tyto pozorovatele upozorní v případě, že se změní jeho stav. Pozorovatelé mohou následně na tuto událost reagovat nějakou akcí. Toto chování podporuje volně vázaný vztah (*loose coupling*) mezi pozorovanými a pozorovateli, díky čemuž je následně mnohem jednodušší upravit nebo rozšířit naší aplikaci.

Příklad implementace návrhového vzoru Observer

Typickým příkladem pro představení tohoto vzoru je situace, kdy máme zpravodajskou agenturu, která vysílá zprávy svým předplatitelům. Zpravodajská agentura je v tomto případě pozorovaným subjektem a předplatitelé jsou pozorovatelé. Když zpravodajská agentura publikuje novou informaci, měli by o tom být informováni všichni předplatitelé. Na tomto příkladu představuje tento návrhový vzor i *Aashi Gangrade* ve svém článku *Observer Design Pattern* [14].

Výpis kódu 1.7 ukazuje definici abstraktní třídy *Observer*, která slouží jako rozhraní pro pozorovatele. Tato abstraktní třída obsahuje jednu abstraktní metodu *update*, která musí být implementována konkrétními pozorovateli a která slouží k aktualizaci pozorovatele s novými informacemi předanými v parametru *news*.

Výpis kódu 1.7 Definice abstraktní třídy *Observer*

```
from abc import ABC, abstractmethod

class Observer(ABC):
    @abstractmethod
    def update(self, news):
        pass
```

Ve výpisu kódu 1.8 vidíme implementaci konkrétního pozorovatele *NewsSubscriber*, který je odvozen od dříve definované abstraktní třídy *Observer* a představuje odběratele zpravodajské agentury. Tato třída implementuje dříve definovanou abstraktní metodu *update*, která slouží k tomu, aby jí volala právě zpravodajská agentura při publikování nových zpráv. Pro jednoduchost tohoto příkladu tato metoda pouze vytiskne přijatou zprávu na příkazový řádek, ale reakcí na přijatou zprávu by samozřejmě mohlo být cokoliv.

[2] Pojem reflexe chápeme jako schopnost programu zkoumat a modifikovat strukturu a chování svého vlastního kódu za běhu.

Výpis kódu 1.8 Implementace konkrétního pozorovatele *NewsSubscriber*

```
class NewsSubscriber(Observer):
    def __init__(self, name):
        self.name = name

    def update(self, news):
        print(f"{self.name} received news: {news}")
```

Na následujícím výpisu kódu 1.9 je zobrazena definice abstraktní třídy *Subject*, která slouží jako rozhraní pro pozorovaný subjekt. Tato abstraktní třída obsahuje následující abstraktní metody:

1. *register_observer* – Slouží k registraci, respektive k přidání nového pozorovatele.
2. *remove_observer* – Slouží k smazání existujícího pozorovatele.
3. *notify_observers* – Používá se k upozornění všech pozorovatelů o změně stavu.

Výpis kódu 1.9 Definice abstraktní třídy *Subject*

```
class Subject(ABC):
    @abstractmethod
    def register_observer(self, observer):
        pass

    @abstractmethod
    def remove_observer(self, observer):
        pass

    @abstractmethod
    def notify_observers(self, news):
        pass
```

V kódu 1.10 je představena implementace konkrétního pozorovaného subjektu, tedy třídy *NewsAgency*. Tato třída implementuje dříve zmíněné abstraktní metody *register_observer*, *remove_observer* a *notify_observers*. Navíc je zde metoda *publish_news*, která načítá zprávy předané v parametru a následně o nich informuje pozorovatele.

Výpis kódu 1.10 Implementace konkrétního pozorovaného subjektu *NewsAgency*

```
class NewsAgency(Subject):
    def __init__(self):
        self.observers = []
        self.latest_news = ""

    def register_observer(self, observer):
        self.observers.append(observer)

    def remove_observer(self, observer):
        self.observers.remove(observer)

    def notify_observers(self, news):
        for observer in self.observers:
            observer.update(news)

    def publish_news(self, news):
        self.latest_news = news
        self.notify_observers(news)
```

Na poslední ukázce kódu, vztahující se k návrhovému vzoru *Observer* 1.11, je demonstrován příklad použití dříve definovaných tříd *NewsSubscriber* a *NewsAgency*. Vytváří se zde instance těchto tříd, registrují se pozorovatelé a ti jsou následně informováni o nové zprávě. Poté je jeden pozorovatel smazán a opět je publikována nová zpráva.

Výpis kódu 1.11 Příklad použití návrhového vzoru *Observer*

```
def main():
    news_agency = NewsAgency()

    # Subscribers
    subscriber1 = NewsSubscriber("Subscriber 1")
    subscriber2 = NewsSubscriber("Subscriber 2")

    # Register subscribers
    news_agency.register_observer(subscriber1)
    news_agency.register_observer(subscriber2)

    # Publish news
    news_agency.publish_news("Breaking News: Observer Explained!")

    # Unregister a~subscriber
    news_agency.remove_observer(subscriber1)

    # Publish another news
    news_agency.publish_news("Python 3.9 Released!")

    # Output:
    # Subscriber 1~received news: Breaking News: Observer Explained!
    # Subscriber 2~received news: Breaking News: Observer Explained!
    # Subscriber 2~received news: Python 3.9 Released!

if __name__ == "__main__":
    main()
```

Výhody a nevýhody návrhového vzoru Observer

Výhody

1. *Loose coupling mezi objekty.* Tento návrhový vzor podporuje volné vazby mezi objekty, které na sebe vzájemně působí.

2. *Efektivita.* Návrhový vzor *Observer* umožňuje efektivně posílat data jiným objektům, aniž by muselo dojít ke změně ve třídách *Subject* nebo *Observer*.

3. *Flexibilita.* Můžeme kdykoliv přidat nebo odebrat pozorovatele.

Nevýhody

1. *Omezení typu One-to-Many.* Tento návrhový vzor je primárně navržen pro vztahy one-to-many mezi pozorovanými subjekty a pozorovateli. Pokud naše aplikace vyžaduje složitější vztahy nebo pozorování více subjektů, je potřeba implementovat vlastní řešení.

2. *Problémy s výkonem při nesprávné implementaci.* Pokud není návrhový vzor *Observer* korektně implementován, může zvýšit složitost a vést k problémům s výkonem.

Shrnutí návrhového vzoru Observer

Použitím návrhového vzoru *Observer* můžeme dosáhnout modulárnějšího, udržitelnějšího a snadněji rozšířitelného návrhu. Tento návrh podporuje *loose coupling* mezi jednotlivými komponentami. Zároveň usnadňuje oganizaci a zlepšuje strukturu našeho kódu tím, že umožňuje objektům pružně a nezávisle na sobě reagovat na změny. Existují samozřejmě knihovny, které tento návrhový vzor implementují a které můžeme využít, abychom ho nemuseli zbytečně implementovat sami. Pro jazyk *Python* existuje například jednoduchá knihovna *pattern-observer*, viz *www.pypi.org/project/pattern-observer*. Pro platformu Node.js pak existuje mimo jiné knihovna *events*, viz *www.nodejs.org/api/events.html*.

1.3.6 Decorator

Návrhový vzor *Decorator* je strukturální návrhový vzor, který umožňuje přidat novou funkcionalitu k existujícímu objektu, aniž by se změnila jeho původní třída. Tento vzor spočívá v zabalení původního objektu do třídy dekorátoru, která má stejné rozhraní jako objekt, který dekoruje. Třída dekorátoru pak přidává do objektu své chování za běhu, což zvyšuje flexibilitu a modularitu programování. Jednoduše řečeno, návrhový vzor *Decorator* umožňuje dynamické rozšíření chování nějakého objektu.

Princip návrhového vzoru Decorator

Popisu tohoto návrhového vzoru se věnuje *Eduard Ghergu* v článku *The Art of Decorating: Applying the Decorator Design Pattern in Real Life* [15], kde popisuje, že tento návrhový vzor se skládá ze čtyř hlavních prvků:

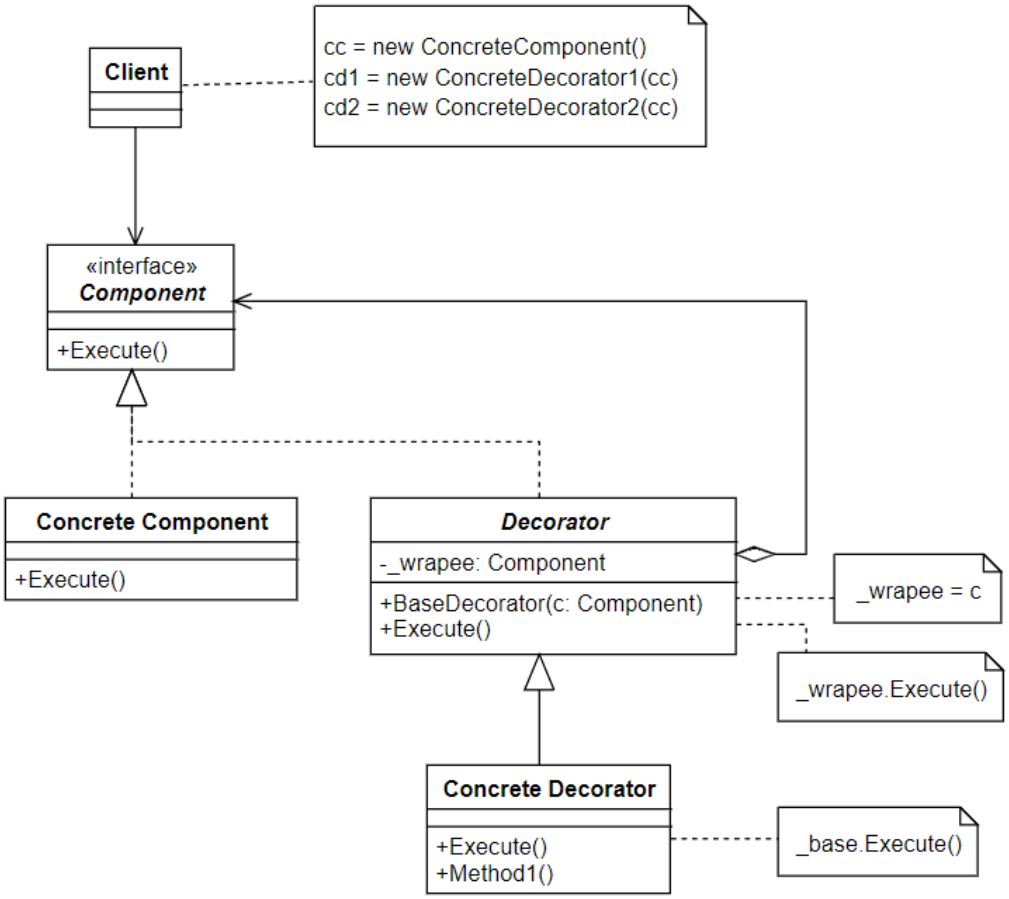
1. *Rozhraní Component*. Toto rozhraní definuje metody nebo vlastnosti, které bude implementovat jak cílový objekt (*Concrete Component*), tak jeho dekorátory (*Concrete Decorator*). Zajišťuje, aby dekorátory splňovaly stejné rozhraní jako objekt, který dekorují.

2. *Concrete Component – Konkrétní implementace rozhraní Component*. Jedná se o objekt, který chceme dekorovat (vylepšit nebo upravit). Implementuje rozhraní *Component* a zajišťuje základní funkčnost.

3. *Abstraktní třída Decorator*. Jedná se o abstraktní třídu nebo rozhraní, které je potomkem rozhraní *Component*. Funguje jako základní třída pro všechny konkrétní implementace třídy *Decorator* (*Concrete Decorator*). Obsahuje referenci na objekt *Component* a může přidávat nové nebo upravovat stávající chování delegováním volání na daný objekt třídy *Component*. Dekorátory můžeme vrstvit na sebe, což znamená, že k objektu můžeme přidat více dekorátorů.

4. *Concrete Decorator – Konkrétní implementace abstraktní třídy Decorator*. Každý konkrétní dekorátor přidává objektu specifickou funkci nebo chování. Přepisují implementaci metod z rozhraní *Component* a mohou volat odpovídající metodu komponenty, kterou obalují.

Návrhový vzor *Decorator* tedy umožňuje dynamicky přidávat nebo upravovat chování objektů pomocí dekorátorů, které implementují stejné rozhraní jako dekorované objekty. Dekorátory mohou být vrstveny na sebe, čímž poskytují flexibilitu a možnost kombinovat různé funkcionality. Výše popsané prvky tohoto návrhového vzoru a vazby mezi nimi jsou vyobrazeny na třídním diagramu na obrázku 1.1.



Obrázek 1.1 Třídní diagram návrhového vzoru *Decorator* z článku *The Art of Decorating: Applying the Decorator Design Pattern in Real Life* od *Eduarda Ghergu* [15].

Výhody a nevýhody návrhového vzoru Decorator

Výhody
Návrhový vzor *Decorator* nabízí několik zásadních výhod při vývoji softwaru, což z něj dělá cenný nástroj pro rozšíření funkcionality objektů. Dle *Eduarda Ghergu* [15] patří mezi klíčové výhody tohoto vzoru následující:

1. *Dodržení Open-Closed principu ze SOLID principů. Decorator* dodržuje a podporuje tento princip. Můžeme rozšiřovat funkčnost objektů, aniž bychom měnili jejich zdrojový kód. To podporuje stabilitu kódu a snižuje riziko zavedení nových chyb při přidávání nových funkcí.

2. *Dodržení Single Responsibility principu ze SOLID principů.* Každá třída dekorátoru má jednu odpovědnost. To umožňuje snadnější údržbu a lepší přehlednost kódu.

3. *Flexibilita a rozšiřitelnost.* Tento návrhový vzor poskytuje flexibilní způsob, jak dynamicky přidávat nebo odebírat funkcionality za běhu programu. Můžeme skládat více dekorátorů na sebe a tím dosáhnout různých kombinací chování.

4. *Znovupoužitelnost.* Dekorátory jsou opakovaně použitelné komponenty. Jakmile vytvoříme dekorátor pro určitou funkcionalitu, můžeme jej použít na různé objekty bez duplikace kódu. To podporuje znovupoužitelnost kódu a minimalizuje redundanci.

5. *Dodržení principu Separation of Concerns.* Rozdělením chování do menších, úzce zaměřených dekorátorů, můžeme oddělit různé úlohy nebo aspekty funkčností objektu. Díky tomuto oddělení je codebase přehlednější a lépe udržitelná.

Nevýhody
Ač tento návrhový vzor poskytuje velké množství výhod, je důležité mít na paměti i jeho nevýhody, které jsou dle *Eduarda Ghergu* [15] následující:

1. *Složitost.* Nadměrné používání dekorátorů může vyústit ve složitou hierarchii, která znesnadňuje pochopení celkové struktury a vztahů mezi dekorátory a základní komponentou.

2. *Zvýšený počet tříd.* Implementace dekorátorů pro více funkčností může vést k vytvoření mnoha tříd. To může mít za následek rozsáhlou a těžce spravovatelnou codebase, zejména pokud existuje mnoho kombinací dekorátorů.

Shrnutí návrhového vzoru Decorator

Návrhový vzor *Decorator* podporuje flexibilní a snadno udržitelný způsob vylepšování a rozšiřování chování objektů, aniž by bylo nutné měnit jejich základní třídy. Dodržuje Open-Closed princip, Single Responsibility princip a Separation of Concerns princip. Tento vzor je výhodný, když potřebujeme přidat funkcionality objektům opakovaně použitelným způsobem. Měl by však být používán s pečlivým zvážením potenciální složitosti a vysoké režie, kterou může do našeho softwarového návrhu vnést.

1.4 Architektonické vzory

Architektura je základním stavebním kamenem při vývoji jakéhokoliv softwaru. Architektonický vzor je osvědčená strukturální skladba různých komponent softwaru pro daný kontext. Architektonické vzory se staly nesmírně důležitými v odvětví vývoje softwaru, a to především díky jejich schopnosti zlepšit výkonnost a škálovatelnost. Zároveň umožňují vyhovět neustále se měnícím potřebám uživatelů.

Oleksandr Torbiievskyi popisuje v článku *Modern Software Architecture Patterns: The Main Things to Know* [16] softwarovou architekturu jako jeden z klíčových bodů při vývoji softwaru. Softwarová architektura totiž představuje návrhová rozhodnutí týkající se celkové struktury systému a jeho komponent, včetně jejich vzájemných vztahů a komunikace mezi nimi. Architektura slouží jako plán softwarových aplikací a funguje jako základ pro vývoj. Architektonický vzor je potom univerzální řešení běžně se vyskytujícího problému. Vzory ve své podstatě, stejně jako návrhové vzory, fungují jako šablony používané pro různé architektonické návrhy. Krom toho se často uplatňují v softwarové dokumentaci, kde pomáhají zúčastněným stranám komunikovat a vzájemně spolupracovat.

Dnes je běžně využíváno více architektonických vzorů, které mohou výrazně zlepšit a zefektivnit vývoj softwaru. Architektonické vzory tak mohou prospět životnímu cyklu vývoje, optimalizovat náklady na projekt, zlepšit zkušenosti uživatelů, usnadnit údržbu softwaru atd. Pomáhají tak dotvářet finální podobu produktu. V této části textu následuje příklad a popis pár vybraných architektonických vzorů, které se hojně používají v moderním vývoji softwaru.

1.4.1 Architektura mikroslužeb

První popsanou architekturou, respektive architektonickým vzorem, je *Microservices* architektura, také zvaná architektura mikroslužeb. Tato architektura využívá *Single-Responsibility* princip a rozšiřuje jej na volně provázané služby, které lze vyvíjet, nasazovat a udržovat nezávisle na sobě. Každá z těchto služeb je zodpovědná za samostatntý úkol a může komunikovat s ostatními službami přostřednictvím jednoduchých API rozhraní, čímž mohou tyto služby řešit komplexní businessové zadání. To, jak tato architektura vypadá a jak postupovat, pokud ji chceme využít pro naší aplikaci, popisuje *Jetinder Singh* v článku *The What, Why, and How of a Microservices Architecture* [17].

Vzhledem k tomu, že vyvíjené služby bývají relativně malé, mohou být od počátku vyvíjeny jedním nebo více malými týmy. Vývojové týmy můžeme rozdělit podle jednotlivých služeb, což usnadňuje případné škálování vývoje. Jakmile jsou služby vyvinuty, mohou být také nasazeny nezávisle na sobě. Díky tomu, že služby mohou být vyvíjeny i nasazovány nezávisle na sobě, získáváme spoustu výhod, mezi které patří možnost snadno identifikovat problémové služby nebo snazší výběr technologií.

Prvním důležitým krokem, rozhodneme-li se pro architekturu mikroslužeb, je dekompozice obchodních schopností právě na jednotlivé služby. Obchodní schopnost chápeme jako něco, co daný business dělá pro to, aby poskytl nějakou hodnotu svým koncovým uživatelům. Identifikace těchto obchodních schopností, a k ním odpovídajících služeb, vyžaduje detailní pochopení daného businessu.

Jakmile máme dekomponované jednotlivé obchodní schopnosti a zároveň jsme rozhodli o podobě výsledných služeb, můžeme přistoupit k vývoji. Jak již bylo zmíněno, dané služby může vyvíjet jeden nebo více malých týmů. Pro každou službu lze vybrat takové technologie, které jsou pro daný účel nejvhodnější. Poté, co je služba vyvinuta, můžeme pomocí zvolené platformy nastavit *CI/CD pipeline*[3] pro spouštění automatizovaných testů a nasazení této služby nezávisle na ostatních službách na zvolená prostředí.

Při návrhu dané služby je důležité ji správně definovat a promyslet si, co uvidí její uživatelé, jaké protokoly se budou používat při komunikaci s ostatními službami atd. Je velmi důležité, abychom uživatele odstínili od veškerých implementačních detailů a vystavili pouze to, co uživatelé potřebují. Pokud totiž vystavíme přebytečné detaily, může být složité službu v budoucnu měnit.

V případě, kdy máme vícero týmů nezávisle pracujících na vývoji různých služeb, bývá nejlepší zavést nějaké standardy a osvědčené postupy – například jak ošetřovat chyby. Pokud standardy nevytvoříme, pravděpodobně se stane, že každá služba bude mít vyřešené ošetření chyb jiným

[3] Pojmu *CI/CD pipline* se mimo jiné věnuje následující část této kapitoly, konkrétně část 1.5.6.

způsobem a zároveň vznikne spousta zbytečného kódu. Vytváření standardů je téměř nutností, stejně jako vytváření dokumentace jednotlivých služeb a jejich vystavených API. Zavádění těchto standardů se věnuje následující část této kapitoly.

Výhody a nevýhody architektury mikroslužeb

Výhody

Některé výhody architektury mikroslužeb, jako například dodržení principu *separation of concerns*, byly již zmíněný v rámci představení tohoto architektonického vzoru. *Jetinder Singh* [17] zmiňuje především následující benefity:

1. *Nezávislý vývoj i nasazení.* Jednotlivé služby nejsou pouze vyvíjeny nezávisle na sobě, ale mohou být nezávisle na sobě v průběhu času i nasazovány. Tato nezávislost zásadně usnadňuje práci vývojářů, neboť jednotlivé služby jsou rozděleny do více repozitářů a tak dochází mnohem méně ke konfliktům při zavádění nových změn do codebase.

2. *Snadná identifikace problémových služeb.* Vzhledem k nezávislému vývoji a nasazování můžeme v systému snadno identifikovat problémové služby, které lze následně škálovat nezávisle na celé aplikaci.

3. *Izolace chyb.* V případě chyby v jedné službě nemusí mít tato chyba nutně přesah i do ostatních služeb, tudíž nemusí vždy přestat fungovat celá aplikace.

4. *Usnadnění výběru technology stacku.* Díky oddělení jednotlivých služeb můžeme pro každou službu teoreticky používat jiné technologie (programovací jazyk, databáze, atd.). To nám umožňuje vždy vybrat ty technologie, které se pro danou službu nejvíce hodí.

Nevýhody

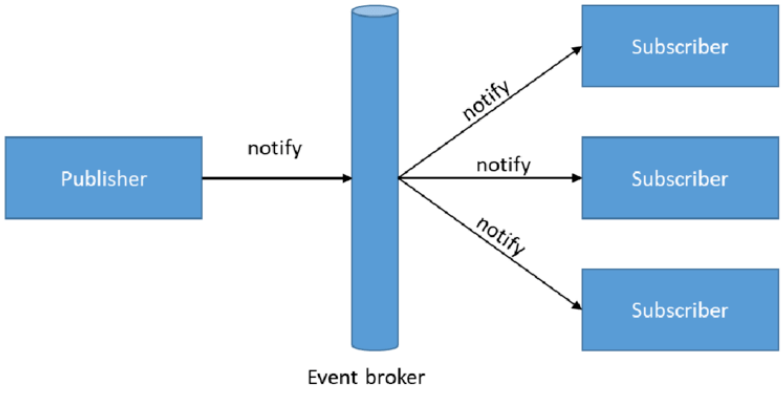
Architektura mikroslužeb má samozřejmě i své neduhy a nevýhody. Některé z těchto nevýhod nemusí při správném návrhu nikdy vyplout na povrch, ovšem o to důležitější je o nich vědět, abychom byli schopni se případně vyhnout jejich příčinám.

1. *Složité provádění změn při špatném návrhu.* Pokud službu špatně navrhneme a vystavíme zbytečné detaily, může dojít k tomu, že bude v budoucnu náročnější službu měnit. Roste totiž pravděpodobnost zásahu do něčeho, co někteří uživatelé využívají.

2. *Nespolehlivost komunikace po síti. Jacob Simon* ve svém článku *Microservices and distributed systems are unreliable. Here's what to do about it* [18] popisuje, že při implementaci architektury mikroslužeb musíme počítat s problémy, které se mohou vyskytnout při komunikaci mezi jednotlivými službami, jako hardwarové selhání, přetížení sítě a další. S tím musíme v naší implementaci počítat a tyto chyby ošetřit, například tím, že bude naše aplikace chyby tolerovat u volání nedůležitých služeb.

3. *Logování.* V této architektuře může být komplikované orientovat se ve výstupech logování – každá nezávisle běžící služba může mít svůj vlastní logger a rozsah těchto logů může být nereálné spravovat. Řešením je nějaká forma centralizovaného logování.

4. *Nutnost automatizace.* Mikroslužby bývají z principu mnohem náročnější na testování, nasazování a údržbu. Vykonávat tyto činnosti manuálně bývá s rostoucím počtem mikroslužeb nereálné a je nutné tyto činnosti automatizovat.



Obrázek 1.2 Diagram Event-driven architektury z článku *What Is Event-Driven Architecture? Everything You Need to Know* od *Stephena Roddewiga* [19].

1.4.2 Event-driven architektura

Další popsanou architekturou v tomto textu je takzvaná *Event-driven architektura*. Tato architektura se může na první pohled zdát jako velmi abstraktní koncept, ovšem její princip je velmi jednoduchý. Tuto architekturu můžeme přirovnat k příkladu řidiče stojícímu na křižovatce, zatímco na semaforu svítí červená. Jakmile se na semaforu rozsvítí zelená (nastane pozorovaná událost), řidič se rozjede vpřed (nastane reakce na pozorovanou událost). O této architektuře píše *Stephen Roddewig* v článku *What Is Event-Driven Architecture? Everything You Need to Know* [19]. Abychom mohli popsat *Event-driven architekturu*, neboli architekturu řízenou událostmi, potřebujeme si nejprve popsat, co je pro nás samotnou událostí.

Události jsou akce, ke kterým může dojít jak v rámci organizace, tak mimo ni – například nákupy zákazníků, aktualizace skladových zásob, narušení bezpečnosti nebo změna stavu aplikace. Tyto události jsou podnětem k reakcím v rámci organizace. V dnešní digitální době IT systémy zaznamenávají, zpracovávají a reagují na události na základě předem nastavené logiky. Všechny události sdílí následující charakteristiky:

- Jsou záznamem o tom, že se něco stalo.
- Jsou neměnné (immutable) – nemohou být změněny ani smazány.
- Lze je uchovávat po neomezenou dobu – události můžeme ukládat a zpřístupňovat navždy.
- Mohou být využity neomezeně mnohokrát – počet zpracování události službou není omezen.

Event-driven architektura (zkráceně *EDA*) je systém navržený tak, aby zaznamenával, přenášel a zpracovával tyto události prostřednictvím oddělené architektury. To znamená, že jednotlivé systémy o sobě nemusí vědět, aby mohly sdílet informace a plnit definované úkoly. V této architektuře figuruje takzvaný zprostředkovatel (*Event broker*), který distribuuje data v reálném čase těm službám, které je potřebují, aniž by o zaslání těchto dat musely dané služby žádat. To je zásadním rozdílem oproti tradičnímu *request-response* modelu.

Na obrázku 1.2 je k vidění jednoduchý diagram této architektury. Na tomto diagramu je vyobrazen *Publisher*, který upozorňuje *Event brokera* na to, že nastala nějaká událost (Publisher publikuje událost). Zmíněný Event broker má pak na starosti distribuci dat všem přidruženým odběratelům (*Subscriber*).

Event-driven architektura bývá běžně kombinována s architekturou mikroslužeb, čímž umožňuje efektivně sdílet informace mezi oddělenými systémy (službami) ve velkém měřítku. Na zmíněném diagramu na obrázku 1.2 mohou být jak entita *Publisher*, tak všechny entity *Subscriber*

samostatnými službami, které mezi sebou komunikují právě pomocí událostí. *Event broker* je pak určitý middleware, který má na starosti rozlišit které služby se zajímají o které události a o těchto událostech je informovat.

V *Event-driven architektuře* existují dva způsoby, jakým mohou spolupracovat producenti (*Publisher*), směrovače (*Event broker*) a konzumenti událostí (*Subscriber*). Prvním způsobem je vzor *Publisher/Subscriber* (zkráceně *pub/sub*). Druhou možností je *Event Streaming* vzor. Následuje podrobnější popis těchto dvou přístupů:

1. *Publisher/Subscriber architektura.* V této architektuře sleduje router událostí to, jaké události odebírá jaký odběratel. V případě, že je nějaká událost zveřejněná, router musí zajistit, aby jí obdrželi správní odběratelé. Události nelze zpětně přehrávat, tudíž pokud se odběratel přihlásí k odběru událostí po jejich vygenerování, nemůže k nim získat zpětný přístup. Daný odběratel bude od počátku odebírání přijímat pouze nové události. Jedná se o analogický příklad s příkladem na obrázku 1.2. Každý odběratel si může zvolit, který typ událostí ho zajímá a na routeru (Event brokeru) pak závisí, aby data týkající se této události daný odběratel obdržel.

2. *Event Streaming architektura.* V architektuře streamování událostí se události zapisují do logu a všichni konzumenti si mohou daný log přečíst a identifikovat příslušné události. Konzumenti mohou číst z jakéhokoliv místa v logu, tudíž se mohou dostat i k událostem z minulosti.

Výhody a nevýhody Event-driven architektury

Výhody

Event-driven architektura je moderní přístup k návrhu softwarových systémů, který nabízí řadu výhod v porovnání s tradičními přístupy. Tato architektura se zaměřuje na zpracování událostí a reakcí na ně v reálném čase. Zásadní výhody této architektury jsou následující:

1. *Oddělení jednotlivých systémů.* Architektura řízená událostmi nám umožňuje mít oddělené systémy, které navzájem neví o své existenci a i přesto mohou sdílet informace a plnit úkoly.

2. *Neměnnost (Immutability).* Jednou vytvořené události nelze měnit, takže je možné je sdílet s více systémy (službami), aniž by nastalo riziko, že jeden systém (služba) změní nebo odstraní informace, které by pak potřebovaly jiné systémy (služby).

3. *Perzistence.* V tradičních modelech, pokud je jedna ze dvou komunikujících služeb mimo provoz, požadavek funkční služby na službu mimo provoz způsobí chybu i u do té doby funkční služby. Díky událostem může zprostředkovatel danou událost uchovávat, dokud nebude služba mimo provoz opět dostupná a připravená událost přijmout.

Nevýhody

Přestože *Event-driven architektura* pomáhá týmům vyvíjet volně vázané, škálovatelné a udržitelné systémy, jako každá architektura má i své nevýhody. Mezi ty zásadní nevýhody a výzvy této architektury patří následující:

1. *Pokles výkonu.* Prostředník (router / event broker) mezi producenty a konzumenty může zhoršovat výkon systému, což může vést k prodloužení doby provádění jednotlivých akcí. Je potřeba najít kompromis mezi přenosem událostí se všemi daty a zasíláním stručnějších oznámení, která vyžadují, aby se konzumenti proaktivně dotazovali na další data. To má za následek potřebu další logiky, která zajistí, že konzumenti získají všechna potřebná data a zároveň nebudou zasílána data zbytečná.

2. *Případná konzistence.* Dvě služby mohou zpracovávat událost a reagovat na ni v různém čase z různých důvodů. To znamená, že ne všechny reakce na danou událost proběhnou ve stejnou dobu, ale ve finále všechny služby dokončí zpracování této události. Tento efekt nazýváme *případná konzistence.*

3. *Složitost.* Kompromisem za velmi nízkou provázanost (*coupling*) jednotlivých služeb je to, že sledování stavu událostí mezi více aplikacemi se stává náročnějším ve srovnání s architekturou, kde máme mezi službami přímé závislosti s jasně definovanými vztahy.

Shrnutí Event-driven architektury

Event-driven architektura je založená na přenášení a zpracovávání událostí. V této architektuře je klíčový zprostředkovatel, takzvaný *Event broker*, který distribuuje data v reálném čase těm službám, které je potřebují, aniž by o zaslání těchto dat musely tyto služby žádat. Komunikace prostřednictvím Event brokera podporuje volnou vazbu mezi službami a díky tomu nehrozí, že zatímco jedna služba nefunguje, přestane nám kvůli chybě v komunikaci fungovat i služba s ní propojená. Nevýhodou ovšem je rostoucí složitost celé architektury a možný pokles výkonů právě kvůli využití prostředníka. Mezi populární nástroje pro implementaci této architektury patří *Apache Kafka*, platforma pro streamování událostí (viz *www.kafka.apache.org*) a nebo například *RabbitMQ*, open source platforma pro implementaci Event brokera (viz *www.rabbitmq.com*).

1.5 Procesy softwarového inženýrství

Zatímco předchozí část textu se věnovala primárně aspektům softwarového inženýrství souvisejících čistě s programováním, tato část je věnována nejrůznějším procesům tohoto řemesla. Konkrétně se tato část věnuje pravidlům a předpisům při vývoji softwaru, sdílení znalostí, code review, psaní dokumentace a testování. V závěru této části je pak popsán proces zvaný CI/CD, který je klíčovým v moderním vývoji softwaru.

1.5.1 Pravidla a předpisy

Většina organizací vyvíjejících software si vytváří a udržuje seznam nejrůznějších pravidel týkajících se vývoje softwaru. Tato pravidla se mimo jiné týkají toho, kde bude uchováván kód, jak bude formátován, jaká bude použitá jmenná konvence, jaké se používají návrhové vzory, jak ošetřovat výjimky nebo pracovat s vlákny. Tato pravidla nejsou pouhými doporučeními, ale striktními předpisy, které musí dodržet každý softwarový inženýr v dané organizaci. Pravidlo může být například to, že veškeré názvy by měly být co nejvýstižnější.

Smyslem vytváření pravidel a předpisů je podpora *pozitivního* chování a odrazení od *nežádoucího* chování. Výklad pojmů *pozitivní* a *nežádoucí* se pak liší v každé organizaci v závislosti na tom, na čem dané organizaci záleží a na co klade důraz. Seznam těchto pravidel se zároveň odvíjí od kontextu dané organizace. V již citované knize *Software Engineering at Google* [1] její autoři představují pravidla v kontextu společnosti Google. Ta zaměstnává více než 30 000 softwarových inženýrů, kteří vykazují obrovské rozdíly v dovednostech a ve vzdělání. Tito inženýři společně každý den vytvoří přibližně 60 000 příspěvků do codebase, která čítá více než dvě miliardy řádků kódu a která existuje již desítky let. V takové organizaci je nutné vytvořit a udržet inženýrské prostředí, které bude odolné jak vůči rozsahu, tak vůči času. Vytváření a dodržování nejrůznějších předpisů a pravidel je prvním krokem, jak takové prostředí vytvořit.

Při vytváření předpisů a pravidel bychom si neměli klást otázku „Jaká pravidla bychom měli mít?“ Lepší je položit si otázku „Jakého cíle chceme dosáhnout?“ Pokud se totiž soustředíme na to, čeho chceme v rámci organizace při vývoji softwaru docílit, snáze určíme pravidla, která tento cíl podporují.

Každá organizace má samozřejmě jiné cíle. Pravidla a předpisy vznikají tím pádem rozdílně, podle kontextu každé společnosti. V organizaci, kde jsou jednotky softwarových inženýrů, kteří všichni spolupracují na denní bázi, bude přístup k vytváření pravidel rozdílný, než v organizacích, kde jsou inženýrů desetitisíce. Inženýři společnosti Google [1] přišli v průběhu času se svým

vlastním přístupem k vytváření pravidel – konkrétně kladou důraz na to, aby pravidla a předpisy dodržovaly následující principy:

1. *Vytváříme jen nutná pravidla.* Ne vše, co nás napadne, musí být ihned pravidlem. Při vysokém počtu pravidel vzniká hned několik problémů. Stávající softwaroví inženýři mohou mít problém si jednak pamatovat veškerá existující pravidla a dále pro ně může být náročné se učit každé další pravidlo. Pro nováčky může být zase příliš dlouhý seznam pravidel zahlcující a složitý na naučení.
2. *Optimalizujeme pro čtenáře.* Pravidla mají být optimalizována pro čtenáře a ne pro autora kódu. V průběhu času bude náš kód mnohem častěji čten, než psán. Pravidla by měla tedy podporovat psaní kódu, který je jednoduchý ke čtení, spíše než jednoduchý k napsání.
3. *Konzistence.* Pravidla mají vést k vytváření konzistentní codebase. Všichni programátoři by měli například ošetřovat chyby stejným způsobem.
4. *Vyhněte se překvapivým konstrukcím a konstrukcím náchylným k chybám.* Pravidla by měla omezovat využívání neobvyklých nebo složitých konstrukcí v jazycích, které při vývoji používáme. Tyto konstrukce mají často svá drobná úskalí, která nemusí být na první pohled zřejmá. To může později vést k vzniku chyb. I když je použitá konstrukce dobře chápána inženýry daného projektu, není zaručeno, že stejně dobře pochopí tuto konstrukci i budoucí členové projektu.
5. *Pokud je to nutné, ustupte praktickým požadavkům.* Ve snaze dodržovat pravidla a předpisy není dobré ignorovat vše ostatní. Někdy narazíme na případ, který bude vyžadovat výjimku z pravidla – a to je v pořádku. V případě potřeby jsou možné ústupky optimalizacím a praktickým záležitostem, které jsou jinak v rozporu s našimi pravidly a předpisy.

Ač se jedná o přístup jedné konkrétní společnosti, tyto principy mohou být při vytváření našich vlastních pravidel a předpisů jistou inspirací. Díky jejich flexibilitě a adaptabilitě je možné je přizpůsobit hodnotám každé organizace a díky tomu vytvořit efektivní sadu pravidel a předpisů, která usnadní vývoj softwaru napříč celou organizací.

Konkrétní příklady pravidel a předpisů

Jak již bylo zmíněno, každá organizace by si měla vytvořit vlastní seznam pravidel a předpisů v závislosti na tom, jaký je celkový kontext dané organizace. V této části jsou popsané jednotlivé oblasti, kterých se můžou pravidla a předpisy týkat. *Syed Ammar* dává ve svém článku *Coding Rules And Standards In Software Development* [20] za příklad tyto oblasti: *Jmenná konvence*, *Pojmenovávání souborů a adresářů*, *Formátování a odsazování*, *Komentáře a dokumentace*, *Importy*. Následuje podrobnější popis jednotlivých oblastí. V kontextu dané společnosti může být těchto oblastí a pravidel více či méně.

Jmenná konvence

Jmenná konvence specifikuje, jak při programování pojmenujeme naše třídy, metody, rozhraní atd. To, jakou konvenci vybereme, se většinou odvíjí od programovacího jazyka, který používáme. V programování jsou typicky používány následující jmenné konvence:

- *camelCase* – První písmeno názvu je malé, každé další slovo začínáme velkým písmenem, nepoužíváme žádný oddělovač slov.
- *lowercase* – Všechna slova začínáme malým písmenem, nepoužíváme žádný oddělovač slov.
- *PascalCase* – Všechna slova včetně prvního začínáme velkým písmenem, nepoužíváme žádný oddělovač slov.
- *snake_case* – Všechna slova začínáme malým písmenem, slova oddělujeme podtržítkem.

Pojmenovávání souborů a adresářů
Jednotná jmenná konvence je důležitá i v případě adresářů a souborů, neboť zpřehledňuje celkovou strukturu projektu. Správné pojmenování adresářů a souborů usnadňuje organizaci projektu a zlepšuje přehlednost. Příkladem takové jmenné konvence je pojmenovávání adresářů s použitím *lowercase* a pouze jediným slovem a pojmenovávání souborů s použitím *snake_case*.

Formátování a odsazování
Další důležitou oblastí, kde je nutné mít jednotné předpisy a pravidla napříč celým projektem, je formátování a odsazování. Tyto předpisy budou vždy úzce spjaté s tím, v jakém programovacím jazyce píšeme kód. Pravidla nám mohou určovat jak budeme psát *if-else* bloky, nebo jaké budeme používat odsazení pro jednotlivé bloky kódu. Důsledné dodržování formátovacích pravidel a předpisů usnadňuje čtení a údržbu kódu a zvyšuje konzistenci v celém projektu.

Komentáře a dokumentace
Jedním z nejlepších způsobů, jak zlepšit schopnost čtenáře porozumět kódu, je kód dokumentovat a komentovat tam, kde je to potřeba. Můžeme například zavést pravidlo psát malé komentáře vysvětlující funkčnost a podstatu každé funkce a smysl jejích parametrů, přidávat TODO komentáře pro označení plánovaných úprav kódu atd.

Importy
Pokud programovací jazyk, který pro vývoj naší aplikace používáme, využívá importy, je dobré zavést pravidla pro jejich používání. Ideální je rozdělit importy podle importů knihoven třetích stran a importů námi napsaných komponent. Dalším užitečným pravidlem je řadit importy podle abecedy, což zlepší čitelnost kódu. Při importování je dobré být konkrétní, tedy namísto importu „*import foo.*;*“, je lepší použít import konkrétní: „*import foo.A; import foo.B;*“. Tímto způsobem víme, které komponenty z daného balíčku chceme v našem kódu používat.

1.5.2 Sdílení znalostí

Sdílení znalostí je důležitým procesem v každém týmu softwarových inženýrů. Pomáhá zvýšit efektivitu celých týmů a usnadňuje práci jak stálým, tak nově příchozím členům. Jak píše *Omar Rabbolini* ve svém článku *Knowledge Sharing in Software Engineering Teams* [21], sdílení znalostí se nerovná sepsání dokumentace a její následné sdílení. Sdílení znalostí vyžaduje dva a více lidí, kteří komunikují nad určitou informací. Můžeme sdílení znalostí přirovnat k mentoringu či doučování, s výjimkou směru předávání znalostí. Zatímco u mentoringu či doučování instruktor předává znalosti a žák je získává, u sdílení znalostí by měly mít všechny zúčastněné strany šanci se něco naučit.

Výhoda sdílení znalostí navíc není jen o zlepšení technické zdatnosti jednotlivých členů týmu. Díky tomu, že se jedná o interakci mezi lidmi, tým jako celek se učí lépe spolupracovat. Roste důvěra a vzájemný respekt s tím, jak si jednotliví členové týmu vzájemně pomáhají a dozvídají se více o stylu práce, silných a slabých stránkách toho druhého. Aby bylo sdílení znalostí efektivní, musí se k němu správně přistupovat. V opačném případě může vyústit ve frustraci celého týmu. V následující části jsou popsány konkrétní metody sdílení znalostí, které *Omar Rabbolini* [21] popisuje v již zmíněném článku.

Schůzky vyhrazené pro sdílení informací

Jedním ze způsobů, jak sdílet znalosti s co nejvíce členy týmu, je uspořádat schůzku k tomu určenou, na které jednotliví účastníci mohou obohatit ostatní o to, co se za poslední dobu naučili. Tyto schůzky se mohou vztahovat ke konkrétním tématům, kterými mohou být projekty nebo různé aktivity. Mohou být i nezávislé na tématu, a vztahovat se k nějaké časové periodě –

například jeden týden nebo uplynulý sprint. Důležité u těchto schůzek je, aby se jednalo o aktivitu, na kterou se těší celý tým, jinak nebudou tyto schůzky zdaleka tak efektivní, jak by mohly být. Následující kroky mohou pomoci k tomu, aby byly schůzky vyhrazené pro sdílení informací zajímavé a zážívné pro celý tým:

1. *Plánujte schůzky mimo „hlavní dobu vývoje“.* Jako softwarový inženýr nechcete být rušen ve chvíli, kdy jste ponořen do práce. Z toho důvodu je lepší naplánovat tyto schůzky ve chvíli, kdy vývojáři nejsou ponoření do páce. Netříštíme tím jejich pozornost a zvyšujeme intelektuální angažovanost týmu. „Hlavní doba vývoje“ se může lišit programátor od programátora, proto je vždy důležité znát kontext daného týmu a dle toho tento čas určit.

2. *Zbytečně se sdílením znalostí neotálejte.* Pokud plánujete tyto schůzky dle zvolené časové periody, například každé dva týdny, může se stát, že členové týmu pozapomenou něco z toho, co se naučili. Bývá tedy lepší plánovat schůzky co nejdříve po dokončení určité části kódu (většinou pevně ohraničeného úkolu), kdy mají vývojáři vše čerstvě v paměti.

3. *Udělejte schůzky interaktivní.* Sdílení získaných zkušeností je pouze polovinou úspěchu. Druhou polovinou je řešení otázek či obav týmu. Sdílení znalostí funguje nejlépe formou diskuze s otázkami a odpověďmi. Stále je ovšem nutné schůzku moderovat, aby se debata neodchýlila od tématu.

Obecně platí, že sdílení znalostí formou k tomu určené schůzky je ideální, pokud chceme sdílet mezi větším počtem lidí, ideálně celým týmem. Tato forma je nejlepší ke sdílení a zapamatování si získaných zkušeností. Zároveň nám sdílení znalostí touto formou může pomoci neopakovat dokola stejné chyby.

Brainstorming

Zatímco předchozí forma schůzky se zaměřovala na sdílení znalostí získaných při implementaci již hotových úkolů, sdílení formou brainstormingu probíhá paralelně s implementací konkrétního úkolu. Typický výstup brainstormingu je řešení daného problému, ovšem tato sezení jsou i skvělým způsobem sdílení znalostí. Pro lepší sdílení znalostí během brainstormingu je nutné si uvědomit, že veškeré nápady jsou si rovny. Každý inženýr musí být připraven obhájit své řešení, ale zároveň by měl být ochoten uznat výhody navržených alternativ. Ten, kdo schůzku plánuje, by měl dát ostatním dostatek času na přípravu, a ideálně i umožnit účast jen jako pasivní pozorovatel.

Brainstorming je ideální forma schůzky, pokud chceme najít řešení nějakého problému, zatímco se jeden od druhého učí novým věcem. Brainstormingu by se měla účastnit menší skupina maximálně deseti kolegů. Brainstorming může být ovšem užitečný i pro pasivní pozorovatele, kteří mohou být případně i nad rámec doporučené kapacity.

Párové programování

Někteří softwaroví inženýři vidí párové programování jako ztrátu času. Obecně tento dojem nemusí být úplně mylný – dva až tři lidé se dívají na stejný problém, zatímco by každý mohl řešit jiný úkol. Párové programování může být ovšem velmi užitečné, hlavně v následujících případech:

1. *Pomoc novým členům týmu seznámit se s codebase.* Novým členům týmu softwarových inženýrů vždy nějakou dobu trvá, než se naučí základy daného systému a poznají jeho architekturu, ať už k němu existuje kvalitní dokumentace či nikoliv. V takovém případě je efektivní spárovat nového člena s někým zkušenějším a přimět je, aby zadaný problém řešili společně. Nový člen se tedy mnohem rychleji dostane do vývojového procesu a může téměř od začátku naplno využívat své dovednosti. Zároveň se tento člen mnohem rychleji osamostatní.

2. *Snížení rizika vzniku chyb při implementaci kritických změn kódu.* Se složitostí úkolu, na kterém inženýr pracuje, se zároveň zvyšuje riziko vzniku chyb. V případě zásahu do kritických částí kódu může párové programování snížit riziko zavedení nových chyb kvůli složitosti příslušného kódu. Je samozřejmě důležité vybrat správnou dvojici inženýrů, aby mezi nimi panoval vzájemný respekt a uznání schopností toho druhého. Důsledkem tohoto přístupu je zvýšení znalosti o systému u obou inženýrů, neboť se od sebe zúčastnění inženýři vzájemně učí.

Párové programování je ideální přístup, pokud chceme snáze začlenit nového člena týmu a nebo je naším cílem snížit riziko vzniku chyb při zásahu do kritických částí codebase. V obou případech dochází k předávání znalostí mezi zúčastněnými stranami. Jak z názvu vyplývá, párového programování by se měli účastnit dva, výjimečně tři členové týmu.

Shrnutí sdílení znalostí

Všechny výše zmíněné techniky sdílení znalostí jsou založeny na interakci mezi členy týmu. Interakcí nejen předáváme znalosti a dovednosti, ale zároveň budujeme důvěru mezi zúčastněnými kolegy. To jsou klíčové výsledky, kterých chceme dosáhnout při závádění strategie sdílení znalostí.

Ať uspořádáme schůzku vyhrazenou pro sdílení znalostí, schůzku zaměřenou na brainstorming nebo naplánujeme párové programování, získáváme nejen nové znalosti o naší codebase, ale také znalosti o našem týmu a jeho jednotlivých členech. Vzájemná pomoc napomáhá budování důvěry a vzájemného respektu, díky čemuž je budoucí sdílení znalostí o to snazší. Noví inženýři se rychleji začlení do týmu a naučí se tomuto přístupu do budoucna.

Snížení efektivity, ke kterému může zdánlivě dojít zavedením těchto činností, je obvykle kompenzováno zkrácením doby zaučování nových členů týmu a snížením počtu nově vznikajících chyb. Správná strategie sdílení znalostí nakonec pomáhá každému členovi týmu se zlepšovat a tím se zlepšuje i tým jako celek.

1.5.3 Code Review

Jak již název tohoto procesu napovídá, *code review* je proces, kdy nově napsaný kód reviduje někdo jiný než sám autor, většinou ještě předtím, než kód přidáme k existující codebase. Přístup k tomu, jak zavést tento proces, se v jednotlivých organizacích liší. Někde existuje skupina vybraných inženýrů napříč celou organizací, kteří mají tuto revizi na starosti. Jinde je revize kódu delegována na menší týmy, čímž je umožněno, aby rozdílné týmy měly rozdílné požadavky na revizi.

Typické kroky Code Review

Pro snadnější pochopení procesu code review následuje seznam kroků, ze kterých se tento proces skládá. Následující seznam je popsán v již několikrát citované knize *Software Engineering at Google: Lessons Learned from Programming Over Time* [1] a popisuje kroky definované inženýry zaměstnanými právě ve společnosti Google. Tento seznam se ovšem nemusí vztahovat pouze ke zmíněné společnosti a pravděpodobně by byl podobný, nebo dokonce stejný, i kdyby ho vytvářeli zaměstnanci zcela jiné organizace. Popsané kroky jsou následující:

1. Uživatel provede změnu codebase ve svém pracovním prostoru. Následně vytvoří snapshot této změny ke které přidá odpovídající popis. Tato změna je nahrána do používaného nástroje pro revizi kódu[4]. Následně se vytvoří porovnání s aktuální codebase, které slouží k vyhodnocení, na jakých místech a jakým způsobem byl kód změněn.

[4]Společnost Google například používá svůj vlastní nástroj *Critique*. Často jsou k tomuto účelu ale používány VCS nástroje použité k hostování repozitáře s naší codebase – některé tyto nástroje jsou popsány v části 1.6.1.

2. Autor změny může následně změnu komentovat a tím přidat svůj pohled na danou změnu. Jakmile považuje autor vytvořenou změnu za kompletní, zašle ji jednomu nebo více recenzentům (proces zaslání se liší dle použitého nástroje). Recenzenti jsou touto formou upozorněni na nově vzniklou změnu a požádání o prohlédnutí a okomentování vytvořeného snapshotu.
3. Recenzenti kontrolují nově vytvořené změny a přidávají komentáře. Některé komentáře mohou být požadavkem na změnu, jiné komentáře mohou být pouze informativní.
4. Autor změny provádí odpovídající opravy na základě komentářů recenzentů a znovu je žádá o kontrolu. Tento a předchozí krok (kroky 3 a 4) se mohou několikrát opakovat.
5. Jakmile jsou recenzenti spokojení se stavem vytvořené změny, změnu schvalují (proces schvalování se opět liší dle použitého nástroje). Každý tým může požadovat různý počet schválení k tomu, aby bylo změnu možné zavést do codebase.
6. Jakmile je změna schválena odpovídajícím počtem recenzentů, autor změnu zavádí do existující codebase.

Výhody a nevýhody Code Review

Správné nastavení procesu code review a jeho dodržování napříč organizací s sebou nese velké množství benefitů, pramenících z toho, že každou změnu, která se zavádí do codebase, kontroluje i někdo, kdo není autorem dané změny. *Alexandra Mendes* a *Rodrigo Ferreira* popisují ve svém článku *What is Code Review and when should you do it?* [22] následující výhody tohoto procesu:

1. *Sdílení znalostí.* Proces code review umožňuje recenzentům předat autorovi své znalosti a postřehy z dané oblasti.
2. *Konzistence napříč codebase.* Každý programátor má svůj osobitý styl psaní kódu. Pokud by všichni psali kód svým stylem, vedlo by to k nekonzistentnímu kódu, zdržovalo vývoj a znesnadňovalo spolupráci. Během code review se inženýři navzájem kontrolují, zda dodržují definované pravidla a předpisy, a tím napomáhají vytváření konzistentní codebase.
3. *Optimalizace kódu pro lepší výkon a snížení rizika vzniku chyb.* I ti nejzkušenější programátoři mohou snadno přehlédnout chybu v nově napsaném kódu. Tím, že změnu zkontroluje více členů týmu, se snižuje riziko vzniku nových chyb a riziko zavedení neefektivních konstrukcí do již existující codebase.

Zásadní a možná i jediná nevýhoda, pokud proces code review nastaví organizace správně, je, že kontrola každé změny zabere čas nejen tomu členovi týmu, který je jejím autorem, ale navíc ještě dalším inženýrům. Tento čas je ovšem vynahrazen časem ušetřeným díky tomu, že se nemusí řešit chyby, které by v codebase bez tohoto procesu mnohem častěji vznikaly. Zároveň tento proces v dlouhodobém měřítku zlepšuje kvalitu a schopnost spolupráce celého týmu, což má za následek efektivnější práci a tím i další ušetření času.

1.5.4 Dokumentace

Jednou z nejčastějších stížností, kterou si můžeme vyslechnout v týmu softwarových inženýrů, bývá absence kvalitní dokumentace. Není proto divu, že dalším důležitým procesem softwarového inženýrství, popsaným v tomto textu, je právě tvorba a udržování projektové dokumentace.

Techničtí analytici a projektoví manažeři mohou být při tvorbě dokumentace nápomocni, ale většinu dokumentace stejně musí vždy psát sami softwaroví inženýři. Potřebují proto vhodné nástroje a hlavně správnou motivaci. Klíčem k usnadnění tvorby kvalitní dokumentace je zavedení procesů a nástrojů, které bude možné v případě potřeby škálovat spolu s organizací a které budou propojeny s jejich stávajícími pracovními postupy.

Co lze považovat za dokumentaci? Dokumentací chápeme všechny doplňující texty, které inženýr potřebuje ke své práci napsat – tedy nejen samostatné dokumenty, ale i komentáře ke kódu. Ve společnosti Google jsou komentáře ke kódu dokonce největší podíl z dokumentace, kterou inženýr napíše.

Typy softwarové dokumentace

Článek *Software Documentation Best Practices* od *Davida Oragui* [23] definuje pojem *softwarová dokumentace* jako typ dokumentace, který poskytuje informace o softwarových produktech a systémech. Obvykle tato dokumentace zahrnuje celou škálu dokumentů a materiálů, které popisují funkce, možnosti a použití softwaru. Autor tohoto článku dále hovoří o tom, že softwarovou dokumentaci můžeme rozdělit do různých kategorií, v závislosti na cílové skupině a účelu dokumentace. Tyto kategorie jsou následující:

1. *Vývojářská.* Dokumentace pro vývojáře a další technické subjekty. Jedná se o dokumentaci ve které se nachází podrobné technické informace o softwaru. Již zmíněným příkladem tohoto typu dokumentace jsou komentáře, které píší inženýři přímo do kódu. Další takovou dokumentací může být dokumentace o rozhraní API, popřípadě dokumentace popisující datové struktury a použité algoritmy.

2. *Provozní.* Dokumentace pro správce systémů a další IT profesionály. Pro tuto cílovou skupinu je často užitečné, pokud existuje instalační příručka, která obsahuje pokyny pro instalaci a nastavení softwaru na různých typech systémů a na různých prostředích.

3. *Uživatelská.* Tento typ dokumentace je většinou nějaký typ uživatelské příručky určený koncovým uživatelům, která popisuje pokyny pro běžné úkoly a zároveň popisuje funkcionality a možnosti softwaru. Častá je také uživatelská dokumentace formou výukových programů nebo jiné typy školících materiálů.

Vzhledem k tomu, že každý ze zmíněných typů dokumentace je určen jiné cílové skupině, je potřeba k vytváření těchto dokumentací přistupovat odlišně, aby byla dokumentace pro danou cílovou skupinu srozumitelná. Koncoví uživatelé například nepotřebují znát detailní informace o hardwarových a softwarových specifikacích daného systému a vývojářům by naopak mohly nějaké technické detaily chybět, pokud bychom je vynechali.

Klíčovým prvkem vývojářské dokumentace bývá soubor, ve kterém je popsán proces zprovoznění a spuštění aplikace, včetně testů, na lokálním prostředí. Takový soubor se většinou nachází přímo v kořenovém adresáři repozitáře daného projektu a zpravidla nese název *README*. Důvody, proč tento soubor vytvářet a rady, jak by měl vypadat, popisuje *Payam Saderi* ve svém článku *Why we should write a good readme?* [24]. V momentě, kdy nejsme na projektu jediným vývojářem a je pravděpodobné, že v budoucnu se na daném projektu vystřídá více kolegů, může být dobře vytvořený soubor *README* ideální vstupní branou pro všechny nováčky.

Osvědčené postupy psaní softwarové dokumentace

Zásadní problém se softwarovou dokumentací je ten, že ač se jedná o extrémně přínosnou součást vývoje softwaru, mnoho softwarových inženýrů má tendenci softwarovou dokumentaci ke svým projektům zanedbávat, ať už z důvodu nedostatku času, odborných znalostí, schopností psát nebo nedostatku motivace. V takových případech je dokumentace buď neúplná nebo vůbec neexistuje. Úvodním krokem k vytvoření kvalitní softwarové dokumentace je tedy věnovat jí čas. Jakmile je však překonána úvodní bariéra a s tvorbou dokumentace se začne, existuje několik kroků, jak tvorbu této dokumentace zefektivnit. Následující kroky popisuje *David Oragui* v již zmíněném článku [23]:

1. *Prioritizace tvorby dokumentace v procesu vývoje.* Ač tento krok může znít jako samozřejmost, jak bylo zmíněno dříve, velké množství softwarových inženýrů tvorbu dokumentace z nejrůznějších důvodů zanedbává. Dalším důvodem může být, že organizace, ve které se software vyvíjí, nezavedla standardizované procesy a předpisy pro psaní a údržbu softwarové dokumentace. Tvorba dokumentace by ovšem měla být vždy prioritou a softwaroví inženýři by neměli doručovat nové funkcionality, pokud nejsou řádně zdokumentované. Důležité je, aby všichni zúčastnění chápali důležitost a výhody softwarové dokumentace, neboť pochopením hodnoty softwarové dokumentace mohou inženýři činit informovaná rozhodnutí o tom, jak jí v procesu vývoje prioritizovat.

2. *Identifikace cílové skupiny.* Jak již víme, softwarovou dokumentaci dělíme do několika skupin, podle toho, které cílové skupině je daná dokumentace určena. Je proto důležité jednotlivé cílové skupiny poznat a přizpůsobit dokumentaci jejich potřebám a očekáváním.

3. *Definice rozsahu a cílů.* Po definování cílové skupiny následuje definice rozsahu a cílů dané dokumentace. To nám pomůže soustředit se na ty nejpodstatnější informace a zajistit, že dokumentace bude relevantní a užitečná.

4. *Vytvoření obsahové strategie.* Dalším důležitým krokem je naplánovat, jak bude tvorba softwarové dokumentace probíhat a kdo bude za její tvorbu zodpovědný. K tomu nám může pomoci definování určitých pravidel a dohodnutí se na používaných nástrojích. Do tohoto plánu můžeme také zahrnout proces revize dokumentace.

5. *Definice jednotného stylu.* Stejně jako vytváříme pravidla a předpisy pro samotný vývoj softwaru, měli bychom vytvořit i pravidla a předpisy pro tvorbu dokumentace. Napříč dokumentací by měl být používán jednotný styl, standardizovaná terminologie, formátování stránek atd.

Výhody softwarové dokumentace

Jak již vyplývá z předchozího textu, softwarová dokumentace je při vývoji softwaru klíčovým prvkem. Bohužel v mnoha organizacích softwaroví inženýři tvorbu dokumentace zanedbávají. Jedním ze způsobu, jak inženýry motivovat ji vytvářet průběžně s vývojem softwaru, je představit jim jednotlivé benefity a vysvětlit, jak softwarová dokumentace v budoucnu usnadní práci jak jim samotným, tak jejich kolegům. Zásadní benefity softwarové dokumentace jsou dle *Davida Oragui* [23] následující:

1. *Lepší zkušenost uživatele.* Softwarová dokumentace pomáhá uživatelům lépe pochopit, jak software používat a poskytuje důležité informace o tom, dosáhnout jejich cílů. Díky tomu zlepšíme celkovou zkušenost uživatele se softwarem a zpříjemníme jeho zážitek.

2. *Zlepšení spolupráce.* Dokumentace umožňuje softwarovým inženýrům pochopit technické aspekty daného softwaru a obsahuje informace, které k vývoji softwaru potřebují. To zlepšuje spolupráci napříč celým týmem.

3. *Zvýšení efektivity.* V dokumentaci by měly být jasné, konzistentní a aktuální informace o daném softwaru, což může pomoci vývojářům pracovat efektivněji. Vývojáři mohou například v dokumentaci rychle najít informaci, kterou potřebují, a díky tomu nemusí trávit čas reverzním inženýringem daného kódu, aby zjistili, jak daný software funguje.

4. *Zlepšení kvality.* Softwarová dokumentace napomáhá zajistit, že proces vývoje softwaru bude konzistentní. Mapuje průběh nejrůznějších rozhodnutí a akcí, které se děly během vývoje, což nám může pomoci zvýšit celkovou kvalitu vyvíjeného softwaru a snížit riziko vzniku chyb.

Psaní dokumentace má tedy velké množství benefitů. V dnešní době existuje nespočet nástrojů, které nám umožní vygenerovat souborovou nebo dokonce webovou dokumentaci právě z komentářů napsaných v kódu. Pro jazyk *Python* je například populární knihovna *Sphinx* (viz *www.sphinx-doc.org*).

1.5.5 Testování

Testování je odjakživa nedílnou součástí programování. Testování softwaru bylo po mnoho let rigidním procesem, většinou manuálním a velmi náchylným na chyby. Od roku 2000 se ovšem přístup celého průmyslu dramaticky vyvinul, aby bylo možné se vypořádat s velikostí a komplexitou moderních softwarových systémů. Hlavním aspektem této evoluce bylo používání automatizovaného testování řízeného samotnými vývojáři.

Automatizované testování může zabránit tomu, aby neodhalené chyby unikly do světa a ovlivnily uživatele. Čím později ve vývojovém cyklu je chyba zachycena, tím je dražší (v mnoha případech i exponenciálně) ji odstranit. Odhalování chyb je pouze částečná motivace, proč zavádět automatizované testy. Stejně důležitým důvodem je, že automatizované testování podporuje naší schopnost provádět změny v softwaru. Ať už se jedná o přidávání nových funkcionalit, refactoring kódu nebo podstoupení většího redesignu aplikace, automatizované testy mohou rychle odhalit vzniklé chyby a zvyšují naše sebevědomí v momentě, kdy provádíme tyto změny.

Psaní testů zároveň zlepšuje celkový návrh našeho systému. Test nám může prozradit spoustu informací o našich rozhodnutích týkajících se návrhu. Je náš systém příliš těsně svázán s databází? Podporuje API všechny požadavky? Vypořádá se náš systém bez problému s krajními případy? Psaní automatizovaných testů nás nutí vypořádat se s takovými problémy již zkraje vývojového cyklu, což obecně ústí ve více modulární a flexibilní software.

Typy testů

Testy dělíme do dvou základních skupin – testy funkční a nefunkční. Jednotlivé typy a jejich subtypy popisuje *Archana Choudary* v článku *Get Started With The Different Types Of Software Testing* [25].

Funkční testy
Funkční testy jsou definované jako ten typ testů, který ověřuje každou funkcionalitu softwaru vůči odpovídající specifikaci požadavků. Funkční testy mohou být jak manuální, tak automatizované. Mezi subtypy funkčních testů patří například následující testy:

1. *Unit testy.* Unit testy spočívají v izolovaném testování jednotlivých komponent nebo jednotek kódu s cílem ověřit jejich správnou funkci. Tyto testy se zaměřují na ověřování nejmenších jednotek softwaru, jako jsou funkce, metody nebo třídy, aby se zajistilo, že při daném vstupu poskytují očekávaný výstup. Unit testy zvyšují naše sebevědomí pokud jde o změny nebo udržení kódu. Je mnohem méně nákladné řešit chyby, které jsou odhaleny v průběhu unit testování, než když jsou odhaleny v pozdějších fázích životního cyklu softwaru.

2. *Integrační testy.* Jedná se o úroveň testování softwaru, kdy jsou jednotlivé jednotky spojeny a testovány jako skupina. Účelem těchto testů je odhalit chyby v interakci mezi integrovanými jednotkami.

3. *Regresní testy.* Regresní testování je pro softwarový produkt klíčovou fází a je velmi užitečné, protože pomáhá zajistit stabilitu produktu s měnícími se požadavky. Regresní testy jsou testy, které se provádí za účelem ověření, zda změna kódu v softwaru neovlivní stávající funkčnost produktu.

Nejedná se o kompletní výčet funkčních testů, dále existují například *systémové testy*, *interface testy*, *user acceptance testy* a další. Podrobnější informace o jednotlivých typech testů jsou popsané buď ve zmíněném článku *Get Started With The Different Types Of Software Testing* [25] a nebo v článku *A Guide to Different Types of Software Testing* od společnosti *Leed Software Development* [26].

Jak bylo zmíněno při popisu návrhového vzoru *Dependency Injection* v části 1.3.4, tento návrhový vzor velmi zjednodušuje unit testování. Poskytuje totiž volnou vazbu mezi komponentami a jejich závislostmi a tím pádem je mnohem jednodušší při testování použít namockované závislosti.

Nefunkční testy

Nefunkční testy na rozdíl od funkčních netestují chování systému vůči existující funkční specifikaci, ale spíše testují nějakou vlastnost. Touto vlastností může být mimo jiné výkon nebo spolehlivost. Mezi subtypy nefunkčních testů patří například následující testy:

1. *Performance testy.* Testování výkonu hodnotí odezvu, škálovatelnost a stabilitu softwaru za různých podmínek pracovního zatížení. Měří se zde faktory jako je doba odezvy, propustnost a využití zdrojů, aby bylo možné posoudit výkonnost softwaru a identifikovat potenciální problémová místa nebo problémy s výkonem.

2. *Reliability testy.* Spolehlivost testujeme s cílem zajistit, že výsledný produkt je bezporuchový a spolehlivý pro svůj zamýšlený účel. Jde o testování aplikace tak, aby byly poruchy odhaleny ještě před nasazením systému. Netestujeme ovšem, zda dané jednotky softwaru vrátí správný výsledek, ale pouze to, že dané jednotky proběhnou, aniž by běh aplikace skončil s chybou.

3. *Security testy.* Testování bezpečnosti hodnotí schopnost softwaru chránit data, zabránit neoprávněnému přístupu a odolat bezpečnostním hrozbám nebo útokům. Identifikuje zranitelnosti, jako je například SQL injection, cross-site scripting (XSS) nebo chyby v ověřování uživatelů a zajišťuje, aby byla implementována vhodná bezpečnostní opatření ke zmírnění rizik.

1.5.6 CI/CD – Průběžná integrace a dodávání kódu

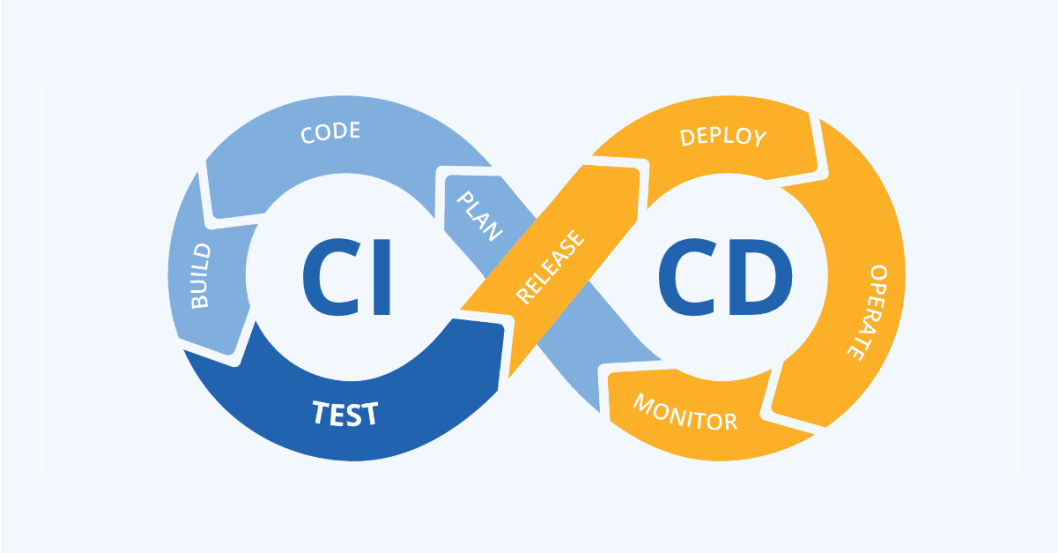
Závěrečným procesem popsaným v této části je průběžná integrace a dodávání kódu. Pro tento proces se obecně používá zkratka *CI/CD*, neboli *Continuous Integration / Continuous Delivery*. Tyto pojmy definuje *Isaac Sacolick* ve svém článku *What is CI/CD? Continuous integration and continuous delivery explained* [27]. Definice pojmu *Continuous Integration* zní takto:

▶ **Definice 1.9** (Continuous Integration)**.** *Continuous Integration je filozofie psaní kódu a soubor postupů, které vedou vývojové týmy k častému provádění malých změn do codebase a jejich následovné kontrole. CI zavádí automatizovaný způsob pro build a testování aplikace při každém zavádění změny.*

Jiné texty [1] definují *CI* více obecně, jako průběžný build a testování celého komplexního a rychle se vyvíjejícího ekosystému. Pojem *Continuous Delivery* je definován následovně:

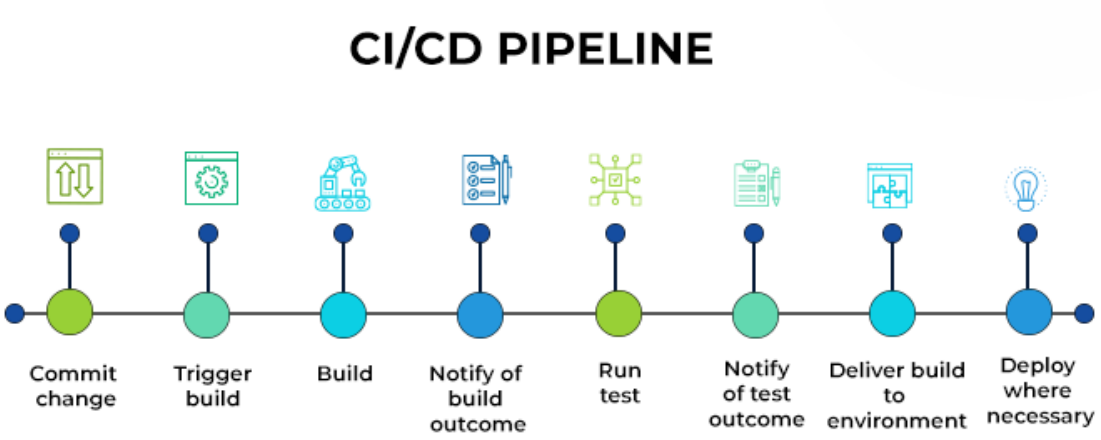
▶ **Definice 1.10** (Continuous Delivery)**.** *Continuous Delivery navazuje na Continuous Integration a automatizuje dodávání aplikací do vybraných prostředí. CD je automatizovaný způsob, jak do těchto prostředí zavádět změny kódu.*

Kombinaci těchto dvou procesů pak známe pod pojmem *CI/CD pipeline*. Celý tento proces je vyobrazen na obrázku 1.3. Na tomto obrázku můžeme vidět, že v rámci procesu *CI* probíhá build a test našeho softwaru. Jak build, tak testování, musí probíhat automatizovaně a vývojář by měl být upozorněn, pokud v jednom z těchto kroků nastane chyba. V rámci procesu *CD* pak dochází k automatizované dodávce výsledné aplikace, služby nebo jiného systému na zvolené prostředí.



Obrázek 1.3 Znázornění *CI/CD pipeline* z článku *An Introduction To CI/CD* od *Izzyho Azeri* [28].

Remya Mohanan mluví ve svém článku *CI/CD definition, process, benefits, and best practices* [29] o *CI/CD* jako o souboru provozních principů a funkcí, které umožňují rychlé, opakovatelné a bezpečné doručení změn softwaru uživatelům, zavedením automatizace do procesů vývoje softwaru. Z tohoto článku také pochází obrázek 1.4, na kterém jsou podrobněji rozděleny jednotlivé kroky *CI/CD pipeline*.



Obrázek 1.4 Znázornění *CI/CD pipeline* z článku *CI/CD definition, process, benefits, and best practices* od *Remyi Mohanani* [29].

Osvědčené postupy CI/CD

Abychom z naší *CI/CD pipeline* vytěžili co nejvíce, je dobré držet se známých a osvědčených postupů. *Remya Mohanan* ve svém článku zmiňuje následující kroky, díky kterým můžeme zefektivnit celý proces a tím zrychlit jednotlivé dodávky:

1. *Automatizace co největšího množství testů.* Abychom minimalizovali riziko lidské chyby, je dobré mít co největší množství testů automatizované. Automatizace testů dále umožňuje inženýrům věnovat se úkolům s vyšší prioritou.

2. *Volba správných nástrojů.* Pro správné fungování a začlenění *CI/CD pipeline* do vývojového procesu je nutné zvolit odpovídající nástroje, ať už se jedná o samotnou platformu pro verzování a správu zdrojového kódu nebo o nástroj určený pro kontrolu kvality kódu. Výběru těchto nástrojů se věnuje následující část této kapitoly.

3. *Co nejčastější slučování změn.* Pro nejefektivnější průběh *CI/CD pipeline* je dobré co nejčastěji (ideálně každý den) slučovat změny z lokálních větví do hlavní větve. Častým slučováním změn snížíme počet konfliktů a minimalizujeme nutnost tyto konflikty řešit.

Výhody používání CI/CD

Smyslem *CI/CD* je umožnit týmu vyvíjejícímu software dodávat nové funkcionality co nejrychleji a nejefektivněji. Správně nastavená pipeline má ovšem celou řadu výhod. Tyto výhody jsou dle *Remya Mohanan* následující:

1. *Zvýšení efektivity.* Vyšší produktivita je jednou z hlavních výhod kvalitní *CI/CD pipeline*. Díky s ní související automatizaci jsme schopni mnohem rychleji dodávat nové verze softwaru.

2. *Snížení rizika výskytu chyb.* Jak již víme, hledání a řešení chyb v pozdější fázi vývoje může být velmi nákladné jak časově, tak finančně. Díky *CI/CD pipeline* a automatizovaným testům jsme schopni velké množství chyb odhalit zkraje vývojového procesu a tím snížit riziko výskytu těchto chyb v pozdějších fázích.

3. *Rychlejší dodávání produktu.* Pokud náš *CI/CD* proces nastavíme správně, můžeme mnohem rychleji nasazovat nové verze na jednotlivá prostředí, navíc s minimem manuálních zásahů do tohoto procesu. Inženýři mají díky tomu více prostoru na úkoly týkající se samotného vývoje a nemusí trávit čas procesem dodávky.

Shrnutí CI/CD

Vytvoření vlastní *CI/CD pipeline* je dnes již standardem pro všechny organizace, které chtějí frekventovaně provádět změny a vyžadují spolehlivý proces dodávky. Jakmile je *CI/CD pipeline* úspěšně začleněna do vývojového procesu, umožňuje celému týmu soustředit se více na samotné změny v softwaru a méně na detaily týkající se dodávání softwaru.

K úspěšnému zavedení *CI/CD pipeline* do vývojového procesu je nutné správně zvolit sadu technologií, které k tomu budeme využívat. Nejen technologiím určeným k sestavení této pipeline se věnuje následující část této kapitoly.

1.6 Nástroje softwarového inženýrství

V poslední části této kapitoly jsou představeny konkrétní nástroje související s dříve popsanými procesy softwarového inženýrství. V této části jsou vybrané skupiny nástrojů, každá skupina je představena a poté jsou z dané skupiny uvedeny příklady konkrétních nástrojů.

1.6.1 Verzování a správa zdrojového kódu

Klíčovým nástrojem, který musí používat každá organizace, jejíž cílem je vyvíjet svůj vlastní software, je nástroj určený k verzování a správě zdrojového kódu. Takové nástroje jsou obecně nazývané *Version Control Systems*, zkráceně *VCS*. Existuje celá řada těchto systémů. *Anjana Rao* v článku *What is version control? Definition, types, systems and tools* [30] dává za příklad tyto běžně používané systémy:

1. *Git*. Jedná se o distribuovaný systém pro správu verzí repozitářů, podobný systému Mercurial (viz dále). Jedná se o vůbec nejpopulárnější VCS nástroj. Podporuje například větvení, nelineární vývoj a hlavně je podporován velkým množstvím platforem třetích stran.

2. *SVN*. Subversion, zkráceně SVN, je centralizovaný nástroj pro správu verzí. Jedná se sice o pokročilejší nástroj než lokální správa verzí, ale vzhledem k jeho centralizaci hrozí, že při poškození centrálního úložiště o všechny změny přijdeme.

3. *TFS*. Team Foundation Server od společnosti Microsoft je stejně jako SVN centralizovaný systém pro správu verzí, který s sebou nese i stejná omezení. Ve spojení s Azure DevOps však může spolupracovat se systémem Git a poskytovat distribuovaný systém správy verzí.

4. *Mercurial*. Jedná se o distribuovaný systém správy verzí s otevřeným zdrojovým kódem (open source). Obecně je populárnější než TFS a SVN, protože zrcadlí centrální úložiště a celou historii na každém lokálním úložišti, čímž zabraňuje úplné ztrátě informací.

Tato část je dále zaměřená na verzovací systém Git. Jedná se o bezesporu nejpopulárnější VCS nástroj. *James Miller* jako důvod této popularity ve svém článku *Which Version Control System Should Your Developers Use?* [31] uvádí například fakt, že Git nabízí nejširší sadu funkcí pro vývojáře, nebo to, že je podporován většinou platforem a služeb třetích stran na trhu. Následuje popis některých z nich, konkrétně platforem *GitHub*, *GitLab* a *Azure DevOps*. *Shanika Wickramasinghe* je srovnává v článku *GitHub, GitLab, Bitbucket & Azure Devops: What's the difference?* [32].

GitHub

GitHub je platforma určená pro hosting kódu, která usnadňuje správu verzí. Zároveň umožňuje bezproblémovou spolupráci na projektech bez ohledu na jejich umístění. Vlastníkem této platformy je od roku 2018 společnost Microsoft. GitHub nabízí grafické uživatelské rozhraní, které umožňuje využívat systém Git. To, jak GitHub funguje, a jeho zásadní výhody popisuje *Maria Kharlantseva* v článku *What Is GitHub? Everything You Need to Know* [33].

Softwaroví inženýři používají GitHub k vytváření vzdálených, jak veřejných, tak soukromých repozitářů v cloudu. Repozitář, jak už bylo popsáno, představuje veškeré soubory daného projektu a jejich celou historii. Po vytvoření repozitáře na GitHubu jej vývojáři mohou zkopírovat do svého zařízení, poté přidávat a upravovat soubory lokálně a následně odesílat změny zpět do úložiště. Tento princip je stejný i pro následující popsané nástroje, tedy i pro *GitLab* a pro *Azure DevOps*. Každý nástroj má pak své specifické výhody. Mezi základní funkcionality a benefity GitHubu patří následující:

1. *Hladké řízení projektu*. GitHub nabízí celou řadu funkcionalit pro řízení a správu projektu, což usnadňuje spolupráci mezi projektovými manažery a vývojáři.

2. *Code Review*. GitHub dává členům týmu možnost revidovat změny jejich kolegů ještě předtím, než jsou zavedeny do hlavní codebase, čímž podporuje proces Code Review. Tento proces je více popsán v sekci 1.5.3.

3. *Github Actions*. GitHub Actions je nástroj v rámci GitHubu, který umožňuje automatizaci nejrůznějších úkolů. Tomuto nástroji se více věnuje následující část zaměřená na nástroje pro vytváření *CI/CD pipeline*.

4. *Nástroje třetích stran*. GitHub umí využívat nástroje třetích stran, díky kterým můžeme například v rámci Github Actions provádět statickou analýzu kódu při každé změně.

5. *Rozsáhlá dokumentace*. GitHub poskytuje rozsáhlou dokumentaci v sekci nápovědy, která usnadňuje orientaci v jeho funkcionalitách.

GitHub nabízí své služby jak zdarma, pro individuální uživatele a malé organizace, kterým stačí omezené služby, a nebo za poplatek pro větší organizace. Cena placené varianty se odvíjí od počtu uživatelů dané organizace a nabízí například lepší týmovou spolupráci nebo větší bezpečnost.

GitLab

GitLab je podobně jako GitHub webová platforma umožňující pracovat s Git repozitáři. Jedná se o DevOps[5] platformu, která umožňuje inženýrům provádět nejrůznější úkoly související se softwarovým projektem. GitLab existuje ve dvou variantách – *GitLab Community Edition*, která je určená pro veřejnost a *GitLab Enterprise Edition*, která je určená pro organizace. Verze určená organizacím navíc nabízí jak placenou, tak neplacenou variantu.

Karin Kelley ve svém článku *What is GitLab and How to Use It?* [35] popisuje, že hlavní výhodou používání GitLabu je, že umožňuje všem členům týmu spolupracovat v každé fázi projektu. GitLab umí projekt monitorovat od plánování až po samotnou implementaci a pomáhá vývojářům automatizovat celý životní cyklus projektu. Mezi hlavní funkcionality a výhody GitLabu patří následující:

1. *GitLab CI/CD*. V rámci GitLabu je možné vytvářet kompletní *CI/CD pipeline*, díky čemuž můžeme mít většinu zásadních nástrojů potřebných k vývoji softwaru na jedné platformě.

2. *Neomezené množství repozitářů*. GitLab neklade žádné omezení na počet repozitářů, nehledě na to zda používáme placenou variantu či nikoliv.

3. *GitLab ChatOps*. Díky této funkci můžeme integrovat *CI/CD* do nejrůznějších služeb, jako je například Slack. Následně je možné spravovat *CI/CD pipeline* skrze tyto služby.

Azure DevOps

Azure DevOps je další DevOps platformou, tentokrát od společnosti Microsoft. Azure DevOps je platforma založená na cloudové platformě Microsoft Azure, přičemž samotná správa verzí spadá pod službu Azure Repos, která existuje v rámci této platformy.

Nástroj Azure DevOps je s výjimkou testovacích plánů dostupný zcela zdarma pro omezený počet uživatelů. Předplatitelé nástroje Visual Studio mají v rámci svého předplatného zároveň plný přístup ke službám Azure DevOps. Tato platforma je nejlepší volbou, pokud pracujeme v organizaci, která ve větším měřítku používá další služby společnosti Microsoft. Zásadní funkcionality a výhody této platformy jsou následující:

1. Přímá integrace s cloudovou platformou Azure a dalšími službami od společnosti Microsoft.

2. Plně funkční platforma zdarma pro uživatele (vyjma testovacích plánů), která zahrnuje neomezený počet soukromých úložišť.

[5] Slovo DevOps je kombinací pojmů *Development* a *Operations*. Tento pojem má představovat společný nebo sdílený přístup k úkolům prováděným týmy vývoje aplikaci (Dev) a provozu v IT společnosti (Ops) [34].

3. Podpora populárních Git klientů.

4. Nástroje určené pro podporu celého životního cyklu vývoje softwaru – konkrétně Azure Boards (nástěnka), Pipelines (CI/CD), Repos (VCS), Test Plans.

1.6.2 Kontejnerizace aplikací

Důležitou součástí moderního vývoje softwaru je takzvaná kontejnerizace aplikací. Tento proces popisuje společnost *RedHat* v článku *What is containerization?* [36]. Jedná se o zabalení softwarového kódu se všemi jeho nezbytnými součástmi a závislostmi, jako jsou knihovny, frameworky a další tak, aby byly izolovány ve vlastním „kontejneru“. To proto, aby bylo možné software nebo aplikaci v tomto kontejneru přesouvat a konzistentně spouštět v jakémkoliv prostředí a na jakékoliv infrastruktuře nezávisle na operačním systému daného prostředí nebo infrastruktury. Kontejner funguje jako jakási bublina, která obklopuje aplikaci a udržuje ji nezávislou na okolí. Je to v podstatě plně funkční a přenosné výpočetní prostředí.

Myšlenka izolace procesů je známá již řadu let, ale až společnost *Docker* stanovila standard pro používání kontejnerů, když představila v roce 2013 nástroj *Docker Engine*. Tento nástroj zároveň přinesl univerzální přístup ke kontejnerizaci softwaru, což následně urychlilo přijetí kontejnerové technologie. Dnes již existuje řada kontejnerových platforem a nástrojů, mezi které patří například nástroje *Podman*, *Buildah* nebo *Skopeo*. V této části následuje popis průkopníka této technologie, kterým je již zmíněný *Docker*.

Docker

Platformu *Docker*, popisuje například *Anshul Ganvir* ve svém článku *Introduction to Docker* [37]. Jedná se o open-source platformu, která umožňuje vývojářům vytvářet, nasazovat a spravovat aplikace v nejrůznějších prostředích. Poskytuje virtualizační systém založený právě na kontejnerech, který vývojářům umožňuje zabalit aplikace do izolovaných prostředí. Ty lze následně nasadit na libovolný operační systém nebo cloudovou platformu. Následuje popis zásadních komponent tohoto nástroje:

1. *Docker Image*. Docker vytváří takzvaný *Docker Image*, což je samostatně spustitelný balíček, který obsahuje vše potřebné ke spuštění aplikace, včetně kódu, provozního prostředí, knihoven a systémových nástrojů.

2. *Docker Container*. Běžící instance konkrétního image se nazývá *Docker Container* – je to izolované prostředí určené pro běh daného image. Toto prostředí nemá žádnou znalost o operačním systému na kterém Docker běží, ani o souborech existujících na dané platformě.

3. *Dockerfile*. Jedná se o konfigurační soubor, na základě kterého Docker automaticky sestavuje image. Tento soubor obsahuje všechny příkazy, které jsou potřebné k sestavení daného image.

Mezi zásadní výhody nejen Dockeru, ale kontejnerizace obecně, patří nezávislost na platformě a na infrastruktuře. Tato vlastnost vychází ze schopnosti kontejnerů sdílet jádro operačního systému hostitelského počítače, což eliminuje potřebu samostatného operačního systému pro každý kontejner a umožňuje, aby aplikace běžela stejně na jakékoliv infrastruktuře, ať už na vlastním hardwaru, v cloudu nebo dokonce i v rámci virtuálních počítačů.

1.6.3 CI/CD pipeline

Proces CI/CD a s ním související CI/CD pipeline byl představen v předchozí části této kapitoly. Nástroje CI/CD jsou pak specializované softwarové aplikace, které jsou navrženy tak, aby usnadňovaly zavedení tohoto procesu do životního cyklu vývoje softwaru. Tyto nástroje jsou

zodpovědné za správu úkolů spojených s automatizovanou integrací kódu, buildem, testováním, kontejnerizací kódu a nasazováním do různých prostředí. *Flavius Dinua* a *Sumeet Ninawe* ve svém článku *13 Most Useful CI/CD Tools for DevOps in 2024* [38] uvádějí celou plejádu nástrojů určených k vytvoření CI/CD pipeline. V této části budou popsány následující nástroje: *Github Actions*, *GitLab CI/CD* a *Azure Pipelines*.

GitHub Actions

GitHub Actions je platforma integrovaná do služby GitHub, která automatizuje převážně build, testování a nasazování softwaru. Tato platforma byla představena v roce 2018 a od té doby se těší vysoké popularitě mezi softwarovými inženýry.

Vzhledem k otevřené povaze platformy GitHub se nejedná o jediný nástroj, který je možné použít k implementaci CI/CD pipeline pro projekty hostované právě na platformě GitHub. V některých případech mohou být vhodnější jiné nástroje, jako například Azure Pipelenes (součást Azure DevOps) nebo Jenkins. Protože jsou však GitHub Actions integrované s platformou GitHub jako celkem, uživatelé se při jejich použití nemusí starat o zajištění komunikace mezi GitHubem a externím nástrojem. Celková funkčnost GitHub Actions je navržena tak, aby usnadnila práci s projekty právě na platformě GitHub, díky čemuž se použití této platformy stává pro mnohé atraktivní volbou. *Davide Benvegnù* v článku *5 Top Reasons to Use GitHub Actions for Your Next Project* [39] zmiňuje následující benefity používání GitHub Actions:

1. *Automatizace.* GitHub Actions umožňují automatizovat pracovní postup. Můžeme s jejich pomocí automatizovaně provádět build, testování či nasazování kódu přímo z GitHubu na cílovou platformu. Tento nástroj je tedy ideálním pro implementaci CI/CD pipeline, pokud používáme GitHub i jako VCS nástroj a tudíž zde hostujeme náš kód.

2. *Přizpůsobitelnost.* GitHub Actions jsou velmi dobře přizpůsobitelné. Můžeme si vytvořit vlastní actions nebo použít actions dostupné na GitHub tržišti a s jejich použitím automatizovat náš pracovní postup.

3. *Integrace.* Další výhodou GitHub Actions je perfektní integrace s dalšími funkcemi GitHubu, jako jsou *pull requesty* nebo *issues*. To umožňuje snadno spravovat celý pracovní postup na jednom místě. Lze například provést automatický build a kód otestovat vždy, když je vytvořen nový *pull request*. Díky tomu zajistíme, že kód je otestován předtím, než je zaveden do hlavní codebase. Kromě toho Actions umožňují integraci se zásadními poskytovateli cloudových služeb a s většinou běžných DevOps nástrojů. GitHub Actions je možné používat i když svůj kód nehostujeme přímo na GitHubu.

4. *Komunita.* O nic méně zásadní výhodou GitHub Actions, je velká aktivní komunita. Jak již bylo zmíněno, na GitHub tržišti je dostupná spousta již předpřipravených Github actions a je možné zde s komunitou sdílet i své vlastní. Často se totiž stane, že někdo před námi již vytvořil Github action, kterou potřebujeme, a tudíž ji nemusíme vytvářet od nuly.

5. *Cena.* Poslední zmíněnou výhodou je cena, respektive to, že jsou GitHub Actions do určité míry zdarma. Nejen proto jsou GitHub Actions velmi populární i mezi jednotlivci a malými týmy.

GitHub Actions mají tedy veliké množství výhod, což z nich dělá atraktivní nástroj pro implementaci nejen CI/CD pipeline, jak pro jednotlivce či menší týmy, tak pro větší organizace. Použít GitHub actions zároveň můžeme i tehdy, pokud používáme jinou platformu než GitHub jako naší VCS platformu.

GitLab CI/CD

GitLab CI/CD je dalším nástrojem, který umožňuje vývojářům automatizovat kroky související s buildem, testováním a nasazováním kódu. Jak už název napovídá, jedná se o nástroj integrovaný v platformě GitLab.

GitLab CI/CD je nástroj podobný GitHub Actions, ovšem má své specifické výhody a funkcionality – některé z těchto funkcionalit popisuje *Enrique Corrales* v článku *GitLab CI/CD Tool Review* [40]. Funkcionalitou, která odlišuje tuto platformu od platforem konkurenčních je například nástroj *ChatOps*. Tento nástroj umožňuje vývojářům interagovat s jejich CI/CD pipeline za použití různých chatovacích služeb. Stejně jako mnoho konkurenčních platforem, nabízí i GitLab CI/CD řadu již hotových funkcionalit pro CI/CD, které mohou vývojáři jednoduše použít. *Michael Levan* pak ve svém článku *5 advantages of GitLab CI/CD pipelines* [41] popisuje tyto výhody nástroje GitLab CI/CD:

1. *Snadná konfigurace.* Nástroj GitLab CI/CD je možné nainstalovat kamkoliv – ať už provozuje organizace GitLab na svých vlastních serverech a nebo v cloudu.
2. *Zabezpečení zdrojového kódu.* GitLab poskytuje plnou kontrolu nad řízením přístupu a fyzickým místem uložení kódu. Organizace si mohou nasadit GitLab na své vlastní servery a tím si veškeré repozitáře a přístupy k nim spravovat sami.
3. *Automatizace.* GitLab obsahuje funkci nazvanou Auto DevOps, která umožňuje automatizovat build, testování, nasazování a monitorování naší aplikace, díky čemuž můžeme implementovat celou CI/CD pipeline.
4. *Zpětná vazba k maturitě DevOps.* GitLab umožňuje uživatelům sledovat skóre vytvářené na základě toho, jak dobře implementují nástroj CI/CD pipeline. Toto skóre a s ním související návrhy na zlepšení pomáhají týmům zhodnotit své DevOps procesy a nebo posoudit, zda vývojáři vhodně využívají dané funkce GitLabu. Poskytované návrhy zahrnují i oblasti mimo GitLab CI/CD pipeline. Uživatelé GitLabu mohou následně porovnávat své aktivity s jinými organizacemi a přistupovat k výukovým materiálům určeným ke zlepšení implementace daných DevOps procesů.
5. *Plánování nasazení.* CI/CD pipeline většinou funguje tak, že po vytvoření změny a po proběhnutí CI procesu se automaticky spouští CD proces, v rámci kterého se kód nasadí. GitLab CI/CD nám umožňuje nenasazovat kód hned, ale určit čas nasazení konkrétní větve na konkrétní prostředí.

Azure Pipelines

Azure Pipelines je další nástroj určený nejen k implementaci CI/CD pipelines, který je integrovaný v rámci rozsáhlejší platformy, v tomto případě Azure DevOps. Tento nástroj může být atraktivní převážně pro organizace, které nějakým způsobem využívají jiné nástroje od společnosti Microsoft a hlavně které již využívají Azure DevOps pro hostování kódu svých projektů. Tento nástroj funguje na stejném principu, jako dříve popsané nástroje GitHub Actions a GitLab CI/CD. Azure Pipelines popisuje podrobně *Jaydeep Patadiya* v článku *Get Started with Azure DevOps Pipelines and Transform Software Delivery* [42]. Mezi zásadní výhody této platformy patří následující:

1. *Rozšiřitelnost a přizpůsobitelnost.* Stejně jako předchozí platformy, i Azure Pipelines nabízí vysokou míru přizpůsobitelnosti a rozšiřitelnosti, zároveň s přednastavenými nástroji, které mohou DevOps týmy použít.
2. *Podpora více cloudových platforem.* Azure Pipelines podporují integraci s všemi zásadními cloudovými službami, tedy mimo cloudové služby Microsoft Azure můžeme integrovat například i AWS, GCP a mnoho dalších.

3. *Integrace s dalšími nástroji.* V rámci Azure Pipelines lze implementovat i nejrůznější testy a QA řešení. Týmy mohou definovat testovací případy, spouštět unit testy nebo provádět další typy automatizovaného testování.

4. *Monitorování a zpětná vazba.* Azure Pipelines poskytují vestavěné mechanismy monitorování a zpětné vazby. Můžou být integrovány s používanými monitorovacími nástroji a díky tomu je možné sledovat stav a výkon aplikací a být informováni, pokud nastanou kritické problémy.

Shrnutí CI/CD nástrojů

Proces CI/CD a s ním související CI/CD pipeline jsou pojmy, které byly představeny v předchozí části této kapitoly. Nástroje CI/CD jsou pak specializované platformy, které jsou navrženy tak, aby usnadňovaly zavedení tohoto procesu do životního cyklu vývoje softwaru. Jsou zodpovědné za správu úkolů spojených s automatizovanou integrací kódu, buildem, testováním, packagingem kódu a nasazováním do různých prostředí. GitHub Actions, GitLab CI/CD a Azure Pipelines patří mezi tyto nástroje pro vytváření CI/CD pipeline a proto byly podrobněji popsány v této části. Každý z těchto nástrojů má své vlastní výhody a specifické vlastnosti, které by měly být zohledněny při výběru nástroje pro konkrétní vývojový tým.

1.6.4 Testování

O důležitosti testování našeho softwaru pomocí automatizovaných testů hovoří předchozí část této kapitoly. *Viktoriia Ivanenko* v článku *Best 7 automated software testing tools in 2024* [43] uvádí a porovnává několik nástrojů určených k automatizovanému testování.

Selenium

Selenium se řadí mezi nejpopulárnější nástroje pro automatizované testování webových aplikací. Jedná se o výkonný nástroj pro testování s použitím nejrůznějších prohlížečů a lze jej použít pro mnoho typů testů. Je kompatibilní se všemi populárními prohlížeči a platformami, takže nabízí možnost testování napříč těmito platformami, což je v prostředí různých zařízení klíčové.

Selenium podporuje vytváření podrobných testovacích skriptů, čímž zlepšuje pracovní postupy a pomáhá vytvářet lépe organizovaný proces testování. Jedná se sice o open source nástroj, ovšem jeho nastavení a údržba může vyžadovat značnou pracovní sílu. Výhody tohoto nástroje jsou následující:

1. Velmi snadné pro testování grafického uživatelského rozhraní.
2. Selenium je open source nástroj.
3. Podporuje většinu zásadních a populárních programovacích jazyků.
4. Možnost paralelního testování, kdy současně zatěžujeme různé části aplikace.

Jedná se tedy o open source nástroj který má velmi přívětivé uživatelské rozhraní. Zároveň umožňuje efektivní integraci díky tomu, že podporuje velké množství platforem a prohlížečů. Tento nástroj má ale i své nevýhody, mezi které patří tyto:

1. Selenium vyžaduje od testera větší technické znalosti, mimo jiné ohledně používání nástrojů třetích stran.
2. Nejedná se o soběstačný nástroj – je nutní podpora frameworků třetích stran, například Sikuli k testování obrázků.
3. Jsou potřebné neustále aktualizace, vylepšování a údržbu celého ekosystému.

Cucumber

Cucumber je open source nástroj zaměřený na testování s použitím BDD[6] přístupu. V rámci BDD se testovací scénáře vytvářejí ještě před napsáním samotného kódu. Cucumber používá svůj vlastní jazyk, který se nazývá Gherkin. Tento jazyk používá syntaxi, jejíž smyslem je být snadno pochopitelná i pro čtenáře bez technických znalostí. Díky této syntaxi pak mohou testy sloužit zároveň jako specifikace vytvořeného softwaru. Výhody nástroje Cucumber jsou následující:

1. Používá jazyk (Gherkin), který je lépe srozumitelný i pro méně technicky zdatné čtenáře.
2. Jedná se o open source nástroj.
3. Umožňuje rychlé a snadné nastavení.
4. Cucumber je nástroj zaměřený primárně na zážitek koncových uživatelů.

Tento nástroj může být tedy velkým přínosem, pokud z nějakého důvodu chceme nebo potřebujeme aplikovat BDD přístup. To může být ale i určitou nevýhodou – konkrétně mezi nevýhody tohoto nástroje patří následující:

1. Jednotlivé kroky v napsaných scénářích jsou těžko přepoužitelné.
2. Vyžaduje alespoň minimální zkušenosti testerů s TDD[7].

1.6.5 Statická analýza kódu

V rámci každé CI/CD pipeline by měla probíhat statická analýza kódu. Nástroje pro analýzu kódu jsou aplikace, které analyzují zdrojový kód a hledají v něm potenciální chyby, aniž by jej spouštěly. Jsou používány k identifikaci a opravě chyb nebo k odhalení bezpečnostního rizika. Tyto nástroje jsou automatizovaně spouštěny právě v rámci CI/CD pipeline. Vývojáři díky nim získávají zpětnou vazbu v reálném čase během práce, což jim umožňuje problémy řešit co nejdříve a dodávat kvalitnější kód. *Paulo Gardini Miguel* v článku *12 Best Code Analysis Tools in 2024* [44] popisuje několik takových nástrojů a uvádí jejich výhody a nevýhody.

SonarQube

SonarQube je open source platforma, která dokáže identifikovat chyby a bezpečnostní zranitelnosti. Zároveň může vynucovat určité standardy kódování pro zajištění konzistentních postupů. Tento nástroj můžeme hostovat samostatně a nebo jej nasadit do cloudu.

SonarQube vyniká zejména svým vestavěným analyzátorem, který upozorňuje na problémy v průběhu kódování. Tyto problémy následně rozděluje do kategorii dle závažnosti a uvádí odhad, jak dlouho bude trvat oprava daného problému.

Tento nástroj umožňuje vytvářet takzvané *quality gates* pro softwarové projekty. Jedná se o určitou sadu pravidel, která mohou být nastavena tak, aby nebylo možné zanést kód do hlavní codebase, pokud nejsou tato pravidla splněna. Mezi další výhody tohoto nástroje patří následující:

1. Podporuje velké množství programovacích jazyků (konkrétní výčet viz dokumentace na stránce *docs.sonarsource.com/sonarqube/latest/analyzing-source-code/languages/overview*).
2. Nabízí integraci s populárními DevOps platformami, jako například s platformou GitHub, GitLab nebo Azure DevOps.

[6] Behavior Driven Development – přístup zaměřený na propojování technických týmů a business subjektů.
[7] Test Driven Development – přístup, kdy jsou nejdříve vytvořeny testovací scénáře, až pak samotný kód.

SonarQube je tedy statický analyzátor kódu, který umožňuje analýzu projektů vytvořených v nejrůznějších programovacích jazycích a který nabízí integraci s většinou populárních DevOps platforem. Jako každý nástroj má ale i SonarQube svá negativa, mezi která patří například tato:

1. Může vést k falešně pozitivním výsledkům – nemusí odhalit všechny chyby a vývojáři by na tento nástroj neměli kompletně spoléhat.
2. Verze tohoto nástroje, která je zdarma, poskytuje pouze omezené funkcionality.

Code Climate

Dalším popsaným nástrojem pro statickou analýzu kódu je Code Climate, který funguje na podobném principu jako SonarQube. Mezi jazyky, pro které poskytuje Code Climate statickou analýzu patří například PHP, Java, JavaScript, Python nebo Ruby (viz dokumentace na stránce *docs.codeclimate.com/docs/supported-languages-for-maintainability*).

Code Climate má nativní integraci s platformou GitHub, díky čemuž je správnou volbou hlavně pro uživatele této platformy. Umožňuje ovšem integraci i s platformou GitLab. Mezi funkce, které odlišují tento nástroj od nástrojů podobných, patří například hodnocení technického dluhu. Code Climate přiděluje kódu známky od A do F na základě jeho udržitelnosti a pokrytí kódu testy. Zároveň odhaduje, jak dlouho by trvalo vyřešení daných problémů. Mezi výhody Code Climate patří následující:

1. *Vizuální přehledy o pokroku.* Tento nástroj poskytuje grafické znázornění pokroku, tedy toho, jak se daří řešit nalezené problémy v aplikaci.
2. *Systém známkování.* Jak již bylo zmíněno, Code Climate známkuje náš kód známkami od A do F, na základě jeho udržitelnosti a pokrytí testy.
3. *Automatické prosazování stylů a standardů kódování.*

Nevýhody Code Climate jsou stejné jako u nástroje SonarQube – tedy výsledky, které nám tento nástroj poskytne můžou být falešně pozitivní. Zároveň, ač tento nástroj má bezplatný plán, tento plán poskytuje pouze omezené funkcionality.

Codacy

Dalším nástrojem pro statickou analýzu kódu, který mohou softwaroví inženýři používat je nástroj Codacy. Tento nástroj podporuje velké množství programovacích jazyků a frameworků již v základním nastavení (viz dokumentace na stránce *docs.codacy.com/getting-started/supported-languages-and-tools*).

Codacy vyniká primárně jednoduchou integrací s hlavními platformami využívanými k implementaci CI/CD pipeline. Mezi platformy, které Codacy nativně podporuje patří GitHub, GitLab a Bitbucket.

Tento nástroj umožňuje nastavit si vlastní sadu pravidel, které musí kód splnit, aby mohl být zaveden do hlavní codebase. Zároveň tento nástroj nabízí stovky pravidel již přednastavených.

K provozování tohoto nástroje pro statickou analýzu kódu zároveň není nutné si sám nástroj provozovat na vlastní infrastruktuře. K používání Codacy stačí vytvoření účtu a následně zvolení předplatného – samotný nástroj je pak ihned k dispozici. Mezi další výhody a funkcionality tohoto nástroje patří následující:

1. *Vizuální přehled.* Codacy poskytuje grafický přehled vývoje kvality kódu naší aplikace. Zároveň má velmi jednoduché a snadno pochopitelné webové uživatelské rozhraní.
2. *Systém známkování.* Stejně jako Code Climate má i Codacy zavedený systém známkování, který hodnotí kvalitu kódu.

Kapitola 2
Návrh

Tato kapitola se věnuje návrhu softwaru – tedy prototypu aplikace, který má být vytvořen v rámci praktické části práce a na kterém mají být předvedeny praktické ukázky principů, procesů a nástrojů popsaných v předchozí kapitole. Krom návrhu samotné aplikace, tedy primárně její architektury, jsou zde popsány i zvolené technologie.

2.1 Představení vyvíjené aplikace

V rámci praktické části práce bude vytvořen prototyp čistě backendového systému, který má imitovat bankovní systém určený k ohodnocení obcí České republiky. Toto ohodnocení bude prezentováno formou skóre, které je poskytováno dále do banky a použito jako jeden z mnoha faktorů při procesu odhadování rizikovosti daného klienta, respektive dané obce.

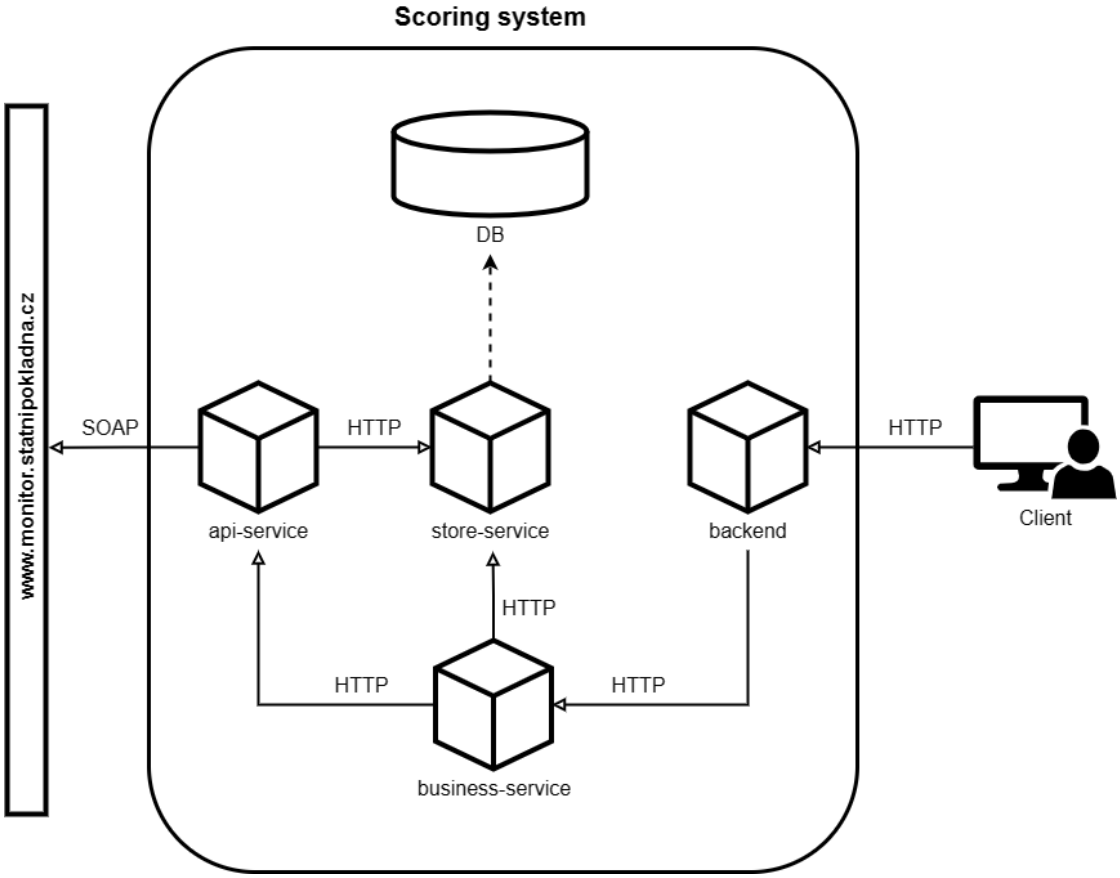
Vytvořená aplikace bude pouze imitací – výpočty prováděné pro získání skóre budou čistě fiktivní. Výpočet bude ovšem založen na reálných datech, které poskytuje Ministerstvo financí České republiky prostřednictvím portálu *www.monitor.statnipokladna.cz*. Na tomto portálu jsou popsány veřejné služby, které mohou klienti využít k získání nejrůznějších dat týkajících se právě obcí České republiky.

2.2 Návrh architektury

Pro implementaci dříve popsané aplikace byla zvolena architektura mikroslužeb. Myšlenka, výhody a nevýhody této architektury jsou popsány v předchozí kapitole, konkrétně v části 1.4.1. Na obrázku 2.1 je diagram navržené architektury. Na tomto diagramu je znázorněna vyvíjená aplikace pod názvem *Scoring system*. Mimo tento systém se na obrázku vyskytuje již zmíněný portál *www.monitor.statnipokladna.cz*, ze kterého bude aplikace čerpat data a poté klient (Client) – ten představuje externí entitu, která může využívat veřejně vystavenou HTTP službu poskytující skóre. Aplikace bude složena celkem ze 4 mikroslužeb, které jsou následující:

1. *api-service* – Jedná se o službu, jejíž úkolem je získávat data právě z portálu Ministerstva financí. Poté co jsou data získána, posílá tato služba data službě *store-service* pomocí HTTP endpointu.
2. *store-service* – Zodpovědností této služby je přístup do databáze a práce s daty – tedy jak vkládání nových dat, tak čtení dat stávajících. Vkládání dat probíhá pouze tehdy, pokud služba *api-service* nějaká data této službě posílá, popřípadě pokud *business-service* vkládá nově napočítané skóre. O čtení dat typicky žádá *business-service*, aby získala potřebná data k vypočítání skóre, popřípadě již dříve napočítané skóre.

3. *business-service* – Tato služba má na starosti počítání skóre konkrétní obce. Požadavek na vypočítání skóre zasílá služba *backend* formou HTTP endpointu. *Business-service* následně posílá požadavek na *store-service* o získání již vypočítaného skóre, pokud je uložené v databázi. Pokud skóre ještě vypočítané nebylo, požaduje tato služba od *store-service* data potřebná k výpočtu. Může samozřejmě nastat i situace, kdy nejsou v databázi ani tato data – v ten moment je zaslán požadavek na *api-service*, aby data načetla. Závěrem zasílá vypočítané skóre jako odpověď na požadavek od *backend* služby a zároveň zasílá požadavek na zápis vypočítaného skóre do databáze.
4. *backend* – Služba vystavující jediný veřejně dostupný HTTP endpoint z celého systému ven do prostředí banky. Odtud si ho mohou volat jednotliví klienti – například systém sloužící k vyhodnocení, zda je bezpečné, aby banka poskytla klientovi úvěr. Po provolání klientem volá tato služba *business-service* službu a následuje proces popsaný právě u této služby.



Obrázek 2.1 Návrh architektury

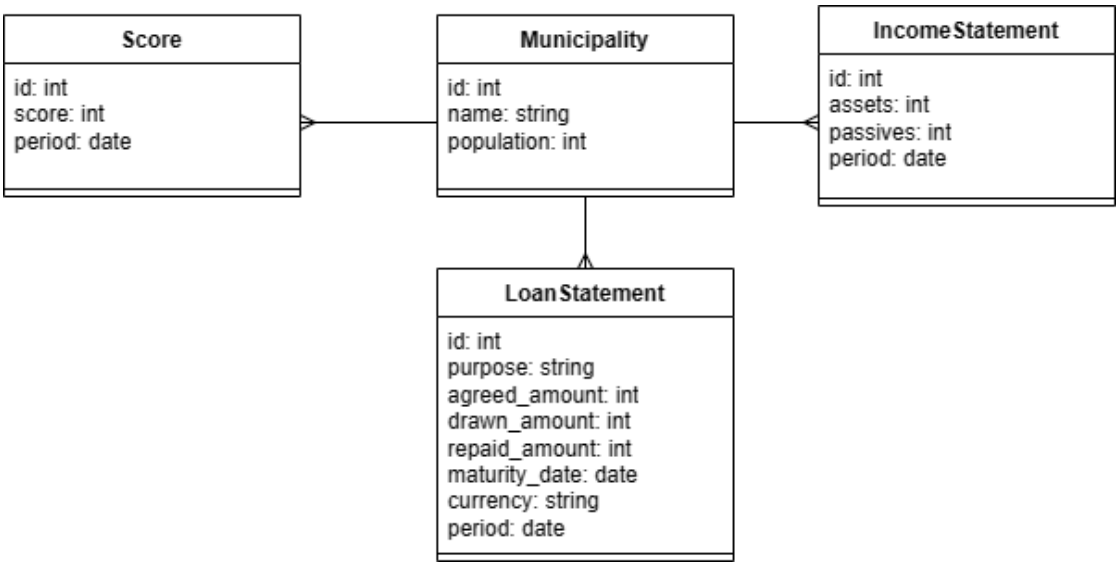
Zásadní výhodou této architektury je dodržení principu *separation of concerns*. Pokud bychom například měnili databázovou technologii, při správné implementaci stačí následný zásah pouze do *store-service* a všechny ostatní služby mohou zůstat nedotčené. Zároveň můžeme po této změně *store-service* nezávisle na ostatních nasadit pomocí naší CI/CD pipeline.

Tato architektura v budoucnu nemusí vůbec sloužit pro výše popsaný případ užití. V bankách (a nejen v bankách) by se mohl dát celý systém, popřípadě jednotlivé služby, přepoužít pro jiné účely – ať už pro počítání jiného druhu skóre, pokud chceme přepoužít celý systém a nebo o načítání dat z nějakého zdroje, kde bychom mohli využít pouze *api-service* a *store-service*.

2.3 Návrh služeb

Tato část se věnuje konkrétnímu návrhu jednotlivých služeb. Nejprve jsou představeny businessové datové entity, které souvisí s výpočtem finálního skóre. Následně jsou jednotlivé služby podrobněji popsány a pro každou službu je zde uveden seznam HTTP endpointů, které služba vystavuje, ať už svému okolí v rámci systému, tak ven do ekosystému banky.

V systému musí existovat sada datových entit, které mohou sloužit k výpočtu skóre a které budou uloženy v databázi. Pro samotný výpočet skóre byly z toho důvodu navrženy tyto datové entity: *Municipalita* (*Municipality*), *Výkaz zisků a ztráty* (*IncomeStatement*), *Výkaz úvěrů a zápůjček* (*LoanStatement*). Výsledkem a hlavně datovou entitou, která bude zajímat klienty této služby je pak entita *Skóre* (*Score*). Diagram návrhu těchto datových entit je na obrázku 2.2. Následuje popis API endpointů jednotlivých služeb, aby mohly dříve popsané služby komunikovat a předávat si mezi sebou právě tyto datové entity.



Obrázek 2.2 Návrh businessovýchc datových entit

2.3.1 Backend

Služba nesoucí název *backend* je tou službou, která bude vystavovat jediný veřejný API endpoint, pomocí kterého mohou klienti získávat napočítané skóre daných obcí vztahujících se k danému období. Tato služba tedy vystavuje jediný endpoint *GET /municipalities/{municipallity_id}/score* – u tohoto endpointu bude dále požadován povinný parametr *období* (*period*), ke kterému chceme skóre vypočítat. Souhrn API rozhraní této služby je vypsán v tabulce 2.1.

Backend

Endpoint	Konzumenti
GET /municipalities/{municipallity_id}/score	Klient (např. frontend)

Tabulka 2.1 Tabulka vystavených endpointů služby backend

2.3.2 Store-service

Smyslem *store-service* je abstrahovat připojení k databázi pro celý zbytek systému. Jedná se o jedinou službu, která bude k databázi přistupovat na přímo a která bude poskytovat ostatním službám potřebná data, popřípadě data vkládat. V databázi budou ukládány dříve popsané businessové datové entity. Veškeré vystavené HTTP endpointy, včetně konzumentů, jsou shrnuty v tabulce 2.2.

Store-service

Endpoint	Konzumenti
GET /municipalities	*Endpoint primárně pro testování*
GET /municipalities/{municipallity_id}	business-service, backend
GET /municipalities/{municipallity_id}/incomeStatement	business-service
GET /municipalities/{municipallity_id}/loansStatement	business-service
GET /municipalities/{municipallity_id}/score	business-service
POST /municipalities/{municipallity_id}	api-service
POST /municipalities/{municipallity_id}/incomeStatement	api-service
POST /municipalities/{municipallity_id}/loansStatement	api-service
POST /municipalities/{municipallity_id}/score	business-service

Tabulka 2.2 Tabulka vystavených endpointů služby store-service

S cílem předvést alespoň nějaké z dříve popsaných návrhových vzorů bude v rámci implementace *store-service* implementován i návrhový vzor *Repository*, který je popsán v předchozí kapitole v části 1.3.3. S použitím tohoto návrhového vzoru ještě více abstrahujeme datovou vrstvu.

2.3.3 Api-service

Api-service je služba, která má za úkol získávat data z portálu Ministerstva financí. Tato služba bude mít vystavené celkem tři HTTP endpointy pro získávání potřebných dat (viz tabulka 2.3), ale musí umět komunikovat i přes SOAP protokol, neboť tímto protokolem komunikuje právě portál Ministerstva financí. Díky dodržení *separation of concerns* v podobě architektury mikroslužeb ovšem změna komunikačního protokolu ze strany Ministerstva financí v budoucnu nezasáhne celý systém, ale pouze tuto službu, kterou můžeme nezávisle na ostatních změnit, otestovat a nasadit.

Api-service

Endpoint	Konzumenti
GET /municipalities/{municipallity_id}	business-service
GET /municipalities/{municipallity_id}/incomeStatement	business-service
GET /municipalities/{municipallity_id}/loansStatement	business-service

Tabulka 2.3 Tabulka vystavených endpointů služby api-service

2.3.4 Business-service

Služba *business-service* bude stejně jako *backend* vystavovat jediný API endpoint, viz tabulka 2.4. Konzumentem tohoto endpointu je právě služba *backend*, která ho využívá v případě, kdy obdrží požadavek o vypočítání skóre dané obce pro dané období.

Business-service	
Endpoint	**Konzumenti**
GET /municipalities/{municipallity_id}/score	backend

Tabulka 2.4 Tabulka vystavených endpointů služby business-service

Funkčnost této služby, která má na starosti to nejdůležitější, tedy počítání skóre, byla shrnuta při představení navržené architektury. Zjednodušeně má tato služba na starost buď získat dříve vypočítané skóre skrze *store-service* a nebo data potřebná k provedení nového výpočtu. Mohou nastat tyto situace:

1. *Busisness-service* požaduje po *store-service* dříve vypočítané skóre. *Store-service* skóre nachází v databázi a vrací ho v odpovědi zpět na *business-service*. Následně je možné skóre vrátit tazateli, tedy *backendu*.

2. *Business-service* požaduje po *store-service* dříve vypočítané skóre, ovšem to není nalezeno v databázi a tudíž vrací *store-service* prázdnou odpověď. *Business-service* zasílá nový požadavek na *store-service* o zaslání dat potřebných k výpočtu skóre. Následně mohou nastat dvě situace:

   a. Potřebná data existují, *store-service* je vrací v odpovědi, *business-service* počítá skóre které následně požaduje uložit do databáze (opět požadavkem na odpovídající endpoint služby *store-service*) a ve finále vrací skóre v odpovědi na *backend*.

   b. Potřebná data neexistují a vrací se prázdná odpověď. *Business-service* zasílá požadavek na *api-service* o načtení potřebných dat přímo z portálu Ministerstva financí. *Api-service* po načtení dat zašle jednak požadavek na uložení těchto dat do databáze (odpovědnost *store-service*), ale vrací je i v odpovědi zpět na *business-service*, aby nebylo potřeba zasílat nový dotaz na získání těchto dat z databáze.

Všechny popsané služby jsou navrženy tak, aby byly co nejvíce univerzální a aby je tím pádem bylo možné v budoucnu znovu použít i za jiným účelem. Zároveň je kladen velký důraz na dodržení principu *separation of concerns* popsaného v předchozí kapitole v sekci 1.2.2. Tento princip je u samotné architektury mikroslužeb klíčovým, umožňuje nám totiž oddělit jednotlivé odpovědnosti našeho systému do samostatných služeb, které pak můžeme nezávisle na sobě měnit a nasazovat na námi zvolené prostředí. Dojde-li tedy například ke změně databázové technologie, dotkne se tato změna pouze služby *store-service*. Pokud by se Ministerstvo financí rozhodlo, že se zbaví komunikace přes SOAP protokol, změna se dotkne pouze *api-service*.

Následující část textu se věnuje technologiím zvoleným k implementaci popsaného systému. Důraz je kladen primárně na volbu technologií usnadňujících právě implementaci a provoz systému postavenému na architektuře mikroslužeb, kde je automatizace testů a nasazování kódu stěžejní.

2.4 Použité technologie

Tato část se věnuje výběru použitých technologií. Technologiemi rozumíme jak technologie nutné k implementaci samotné aplikace, tak technologie nutné k předvedení praktických ukázek procesů popsaných v předchozí kapitole. V rámci praktické části bude nejen představena implementace aplikace, ale i těchto procesů, konkrétně sestavení CI/CD pipeline na vybrané platformě a vytvoření automatizovaných testů. Tato část se tedy následně dělí na dvě části – jedna věnující se implementaci aplikace a druhá věnující se implementaci procesů.

2.4.1 Implementace aplikace

Pro implementaci aplikace je nutné zvolit primárně programovací jazyk a následně již k tomuto jazyku vztahující se knihovny potřebné k implementaci jednotlivých služeb. Klíčová pro jednotlivé služby je knihovna pro komunikaci pomocí HTTP protokolu a vzhledem k povaze portálu Ministerstva financí i knihovna umožňující komunikovat pomocí SOAP protokolu. Zvolené technologie jsou následující:

1. *Programovací jazyk: Python 3.9.*
2. *Knihovna pro vytvoření HTTP APIs: Fastapi.*
3. *Knihovna pro komunikaci přes HTTP protokol a přes SOAP protokol: Requests*

2.4.2 Implementace procesů

K procesům, které budou předvedeny v praktické části práce patří sestavení CI/CD pipeline a vytvoření automatizovaných testů, které se budou nejen v rámci této pipeline, konkrétně jako součást CI procesu spouštět. Je tedy potřeba vybrat odpovídající VCS platformu, ideálně tak, aby na této platformě bylo možné sestavit kompletní CI/CD pipeline. Zároveň je potřeba zvolit platformu, na které bude hostována výsledná aplikace, respektive jednotlivé služby. Neméně důležitý je ovšem i nástroj pro vytváření automatizovaných testů. Nástroje, které budou použity pro implementaci jednotlivých procesů jsou následující:

1. *VCS platforma: GitHub.*
2. *CI/CD platforma: GitHub Actions.*
3. *Automatizované testy: Pytest*[8].
4. *Platforma pro nasazení aplikace: Microsoft Azure.*
5. *Statická analýza kódu: Codacy.*

[8] Vzhledem k tomu, že se jedná o jednoduchý prototyp backend systému, byl k testování zvolen nástroj *pytest*. Tento nástroj a jeho konkrétní použití v rámci projektu se nachází v kapitole 4 (*Testování*)

Kapitola 3
Implementace

Tato kapitola je věnována implementaci systému, na kterém mají být předvedeny některé dříve popsané přístupy a technologie. Je rozdělena do několika částí – nejprve je zde popsána implementace služeb navržených v předchozí kapitole, poté je představena implementace procesů, primárně CI/CD pipeline. Testování výsledného softwaru popisuje následující kapitola (v následující kapitole je popsán i proces statické analýzy kódu, který stejně jako testování patří do procesu CI).

3.1 Implementace služeb

Jako VCS nástroj byla zvolena platforma *GitHub*. Každá služba má na této platformě vytvořený svůj vlastní repozitář, v rámci kterého jsou vytvořeny i jednotlivé GitHub Actions, pomocí kterých je implementována CI/CD pipeline. Veškeré repozitáře vztahující se k této práci jsou pod organizací *pilarmi2* (FIT ČVUT uživatelské jméno autora), dostupné na odkazu *www.github.com/pilarmi2*. Následuje popis jednotlivých služeb – nejsou zde popsány veškeré zdrojové kódy, ale primárně zajímavosti a důležité elementy jednotlivých služeb. Zbylé zdrojové kódy jsou včetně popisujících komentářů k vidění na zmíněné adrese.

3.1.1 Store-service

Jak bylo zmíněno při návrhu systému, v rámci implementace služby *store-service* bylo cílem implementovat i dříve popsaný návrhový vzor *Repository*, který nám poskytuje abstraktní vrstvu nad vrstvou přistupující k databázi. Ukázka implementace abstraktní třídy *Repository*, která slouží jako rozhraní pro konkrétní implementace jednotlivých tříd přistupujících k datům, je k vidění ve výpisu kódu 3.1. Je zde vidět, že tato třída nenabízí abstraktní metodu pro mazání jednotlivých záznamů – to koresponduje s tím, že neexistují ani vystavené HTTP endpointy pro mazání záznamů. To, jak je systém navržen, umožňuje na úrovni jednotlivých služeb data pouze vkládat, maximálně aktualizovat. Mazání dat není povoleno, neboť ani jedna služba nemá motivaci data mazat. Rozšíření tohoto rozhraní o mazání by mohlo dávat smysl, pokud by byl systém rozšířen o frontend a uživatelé z nějakého důvodu tuto funkcionalitu potřebovali. Momentálně ale jediným smyslem výsledného softwaru zůstává počítat a poskytovat skóre vztahující se k jednotlivým obcím.

Ukázka kódu 3.2 pak obsahuje implementaci abstraktní třídy *Repository* zaměřenou na práci s datovou entitou *Municipality*. Tato třída se jmenuje *MunicipalityRepository* a jako datové úložiště používá pouze datový typ *List* a data za běhu programu uchovává v paměti. Jakmile ovšem dojde k výpadku, o všechna data přijdeme – k tomuto ústupku došlo kvůli tomu, že systém je

Výpis kódu 3.1 Implementace abstraktní třídy *Repository* v rámci služby *store-service*

```
from abc import ABC, abstractmethod


class Repository(ABC):
    @abstractmethod
    def get_all(self):
        pass

    @abstractmethod
    def get_by_id(self, object_id: int):
        pass

    @abstractmethod
    def get_by_id_and_period(self, object_id: int, period: str):
        pass

    @abstractmethod
    def add_or_update(self, data: dict):
        pass
```

pouze prototyp určený k demonstraci vybraných postupů a nástrojů a tím pádem není použití tradiční databáze klíčové. Třída neimplementuje metodu *get_by_id_and_period*, protože entita typu *Municipality* neobsahuje atribut *period*.

Výpis kódu 3.2 Implementace *MunicipalityRepository* v rámci služby *store-service*

```
class MunicipalityRepository(Repository):
    def __init__(self):
        self.__municipalities: List[Municipality] = []

    def get_all(self) -> List[Municipality]:
        return self.__municipalities

    def get_by_id(self, object_id: int) -> Municipality:
        return next(
            (m for m in self.__municipalities if m.municipality_id
            == object_id), None)

    def add_or_update(self, entity: Municipality):
        municipality: Municipality = next((m for m in self.__municipalities
            if m.municipality_id == entity.municipality_id), None)

        if municipality is None:
            self.__municipalities.append(entity)
            return StandardResponse(status=200, message="Created")
        else:
            municipality.municipality_id = entity.municipality_id
            municipality.name = entity.name
            municipality.citizens = entity.citizens
            return StandardResponse(status=200, message=f"Updated")
```

Na ukázce kódu 3.3 je vidět použití vytvořených tříd *Repository* a *MunicipalityRepository*. Je zde k vidění, že program pracuje s proměnnou *repository* pouze se znalostí rozhraní abstraktní třídy *Repository*. Je to až inicializace třídy *MunicipalityRepository*, která proměnné poskytuje

implementaci konkrétních metod, ovšem klient, který chce přistupovat k datům, by měl znát pouze abstraktní definici jednotlivých metod a neměla by ho zajímat konkrétní implementace. Na všech zmíněných výpisech kódu vidíme, že v momentě, kdy budeme chtít přejít z jedné databázové technologie na jinou, stačí nám změnit pouze konkrétní implementaci, tedy třídu *MunicipalityRepository*.

Výpis kódu 3.3 Použití implementované *MunicipalityRepository* v rámci služby *store-service*

```
repository: Repository = MunicipalityRepository()


@router.get("")
async def search_municipality() -> List[Municipality]:
    return repository.get_all()


@router.get("/{municipality_id}")
async def search_municipality_by_id(
        municipality_id: int
) -> Optional[Municipality]:
    return repository.get_by_id(municipality_id)


@router.post("")
async def create_municipality(
        municipality: Annotated[Municipality, Body()],
) -> StandardResponse:
    return repository.add_or_update(municipality)
```

Jak již bylo zmíněno, v momentě, kdy bychom chtěli přejít z naivního *in-memory* ukládání dat na solidnější řešení, například v podobě *Azure Cosmos DB*, nemuseli bychom měnit nic jiného, než konkrétní implementace abstraktní třídy *Repository* (tedy v tomto případě *MunicipalityRepository*). Stejným způsobem jsou v rámci služby *store-service* implementovány i přístupy k zbylým datovým entitám.

3.1.2 Api-service

Specifikem služby *api-service* je schopnost komunikovat s portálem Ministerstva financí pomocí protokolu SOAP. Za tímto účelem byla využita knihovna *requests*, která nám umožňuje posílat s použitím HTTP protokolu předdefinované zprávy ve formátu XML. Tento formát je právě typický pro komunikaci přes SOAP protokol. Zmíněná knihovna je napříč celým systémem použita i ke komunikaci mezi jednotlivými službami pomocí HTTP požadavků. Zároveň je specifikem této služby, že se jedná o jednu ze dvou služeb, která používá POST endpointy služby *store-service*.

Co se týče obsluhování komunikace s portálem Ministerstva financí, to má v rámci této služby na starosti třída *FinanceMonitor*. Tato třída obstarává komunikaci, následné zpracování obdržených dat včetně transformace do formátu *json* a navrácení jen potřebných dat tazateli. Zpracování každého požadavku se skládá ze dvou kroků. Prvním krokem je sestavení požadavku v odpovídajícím XML formátu, aby bylo možné jej použít pro komunikaci přes protokol SOAP – to obstarává metoda nazvaná *___build_soap_request*. Druhým krokem je zaslání tohoto požadavku a zpracování odpovědi. Celou třídu je možné si prohlédnout ve veřejném repozitáři na GitHubu na již zmíněném odkazu *www.github.com/pilarmi2/api-service*.

Oproti návrhu přibyla ještě integrace s dalším portálem Ministerstva financí, s názvem *Ares*. Tento portál vystavuje veřejnou službu, která na základě identifikačního čísla dané obce poskytne její název, což portál Monitor neumožňuje. Komunikace s portálem Ares již probíhá přes HTTP

API pomocí zmíněné knihovny requests, tudíž byla samotná implementace jednodušší – tuto komunikaci zajišťuje třída *AresPortal*. Popis a technická dokumentace tohoto portálu a jednotlivých služeb je k nalezení na stránce *www.mfcr.cz/cs/ministerstvo/informacni-systemy/ares*.

Potenciálním zlepšením této služby by bylo provádět komunikaci se službou *store-service* asynchronně, tedy zapojit prvky *event driven architektury* v podobě *message brokera* (například platformy Apache Kafka), požadavky na vkládání nových záznamů pouze předávat tomuto brokerovi a nečekat zbytečně na odpověď. Pokud totiž při vkládání dat do databáze v rámci *store-service* nastane chyba, bude chybná i odpověď *api-service* na obsluhovaný požadavek. Službu *business-service* ale nemusí trápit, že se požadovaná data nepodařilo vložit do databáze a proto by u *POST HTTP endpointů* služby *store-service* dávala asynchronní komunikace větší smysl. Stejně jako u implementace databáze se ovšem ani zde nejedná o klíčový element pro předvedení dříve popsaných principů a nástrojů a proto byl celkový návrh architektury zjednodušen.

3.1.3 Business-service

Službou, která obstarává výpočet požadovaného skóre je třída business-service. Tato služba komunikuje se *store-service* s cílem získat požadovaná data nebo zapsat nově vypočítané skóre. Dále komunikuje se službou *api-service* pokud od *store-service* neobdrží požadovaná data. Komunikace mezi jednotlivými službami probíhá stejně jako u *api-service* pomocí knihovny *requests*.

Zásadním prvkem této služby je samozřejmě počítání samotného skóre. V reálném bankovním systému by byl tento výpočet mnohem složitější a vypočítaný z více datových entit, nejspíše i napříč několika časovými obdobími. Pro účely tohoto textu stačí jednoduchý výpočet. Vzorec pro výpočet skóre byl zvolen následující:

$$max(1, \frac{pasiva + soucet\_pujcek - aktiva}{pocet\_obyvatel})$$

Tímto vzorcem dosáhneme dvou vlastností. První vlastnost je, že čím rizikovější klient, tím horší skóre – to totiž roste, pokud jsou pasiva a agregované půjčky poměrově vyšší, než aktiva dané obce. Druhou vlastností je, že nejmenší možné skóre je 1, čehož je docíleno funkci *max*. Horní součet je zároveň vydělen počtem obyvatel. Tento výpočet je zcela smyšlený a reálně neposkytuje žádnou informaci o rizikovosti klienta. Cílem je primárně předvést architekturu mikroslužeb a konkrétní vypočítané skóre je vzhledem k tomu, že se jedná o prototyp, vedlejší. Třída obstarávající výpočet skóre je k vidění ve výpisu kódu 3.4.

Výpis kódu 3.4 Třída *ScoreCalculator* počítající skóre

```
class ScoreCalculator:
    def calculate_score(self, municipality: Municipality,
                        income_statement: IncomeStatement,
                        loans_statements: List[LoanStatement]) -> Score:
        citizens: int = municipality.citizens
        assets: float = income_statement.assets
        passives: float = income_statement.passives
        aggregated_loans: float = 0.0

        for l in loans_statements:
            aggregated_loans += (loan.drawn_amount - loan.repaid_amount)

        score: float = round(max(1, (passives + aggregated_loans - assets) /
                             citizens), 2)

        return Score(municipality_id=municipality.municipality_id,
                     score=score,
                     period=income_statement.period)
```

3.1.4 Backend

Nejjednodušší službou je služba *backend*. Ta má na starosti pouze přeposlání požadavku od klienta službě *business-service* a po obdržení skóre ho vrátit tazateli. Třída zajišťující komunikaci se službou *business-service* je k vidění ve výpisu kódu 3.5. V této třídě je použita několikrát zmíněná knihovna *requests*, která zajišťuje právě komunikaci přes HTTP protokol.

Výpis kódu 3.5 Třída zajišťující komunikaci mezi službami *backend* a *business-service*

```
class BusinessService:
    def __init__(self):
        self.__url = os.environ["BUSINESS_SERVICE"]

    def get_score(
        self,
        municipality_id: str,
        period: str
    ) -> Union[Score, StandardResponse]:
        response = requests.get(f"{self.__url}"
                                f"/municipalities/{municipality_id}"
                                f"/score?period={period}")

        if response.text != "null"
        and response.status_code == 200:
            return Score(**response.json())
        else:
            return None
```

3.2 Implementace procesů

Tato část se věnuje implementaci a ukázkám jednotlivých procesů. Následuje v ní popis implementace CI/CD pipeline, ale zároveň je zde nastíněno například psaní dokumentace nebo to, jak díky platformě GitHub provádět *code review*.

3.2.1 Implementace CI/CD pipeline

Vzhledem k tomu, že VCS platformou pro tento projekt je platforma GitHub, dává smysl použít pro CI/CD pipeline GitHub Actions. Jak již bylo popsáno v předchozí kapitole tohoto textu, konkrétně v sekci 1.5.6, CI/CD pipeline se skládá z průběžného testování a buildu kódu (CI) a z průběžného nasazování kódu (CD) na cílovou platformu (Azure). Z toho důvodu dává i největší smysl vytvořit pro každou část tohoto procesu jednu GitHub Action – první se bude spouštět vždy při vytvoření *pull requestu*, aby proběhly veškeré kontroly včetně testů, druhá při zanesení změn do hlavní větve (*master*) a bude nasazovat výsledný kód na platformu Azure.

Implementace CI procesu

GitHub Actions jsou definovány pomocí souborů typu *YAML*, které jsou v GitHub repozitáři umístěny ve složce *.github/workflows*. Každý soubor nám reprezentuje jednu action (respektive workflow). Tuto workflow je potřeba vytvořit pro každý repozitář, tudíž pokud máme v našem systému čtyři služby a tím pádem čtyři repozitáře, každá služba bude mít své specifické GitHub Actions, i když mohou být zpravidla velmi podobné.

V rámci CI procesu chceme vytvořit GitHub Action, která nám při každém novém pull requestu kód sestaví (provede build) a otestuje. Tato action je vzhledem ke své povaze stejná pro všechny implementované služby. Definice této workflow je k vidění ve výpisu kódu 3.6.

Výpis kódu 3.6 YAML kód definující GitHub Action pro *build* kódu

```
- name: Build

on:
  pull_request:
    branches: [ "master" ]

permissions:
  contents: read

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python 3.9
      uses: actions/setup-python@v3
      with:
        python-version: "3.9"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
            wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --
            statistics
    - name: Test with pytest
      run: |
        pytest
```

Jak je ve zmíněném výpisu kódu vidět, workflow se skládá z několika úrovní. Nejvyšší úrovní jsou *jobs* – v případě této workflow je zde pouze jeden job s názvem *build*. V rámci jobu je definováno, na jakém operačním systému chceme, aby job běžel (*ubuntu-latest*) a následně jsou definované jednotlivé kroky tohoto jobu. Prvním krokem je nastavení python prostředí a v rámci tohoto kroku je specifikovaná verze pythonu, kterou chceme spouštět. V následujícím kroku je uveden příkaz, pomocí kterého se nainstalují veškeré závislosti projektu ze souboru *requirements.txt*, zároveň je v rámci tohoto kroku nainstalován nástroj *pytest*. Další krok spouští nástroj, který odhaluje syntaktické chyby v kódu naší aplikace a v posledním kroku jsou pomocí nástroje *pytest* spouštěny testy. Ještě před definicí samotných *jobs* je také potřeba uvést, kdy a s jakými oprávněními se bude daná workflow spouštět – tato workflow se například spouští při každém pull requestu do větve *master* a má práva pouze pro čtení.

Implementace CD procesu

Jednotlivé služby jsou nasazovány na platformu Microsoft Azure v takzvaných kontejnerech. K tomu nám pomáhá softwarová platforma s názvem *Docker*. Jak tato platforma, tak samotný proces kontejnerizace jsou popsány v první kapitole v části 1.6.2.

Docker umožňuje automaticky sestavit takzvaný Docker Image pouze na základě konfiguračního souboru, který se nazývá Dockerfile. Jedná se o textový dokument, který obsahuje všechny

příkazy potřebné k tomu, aby byl sestaven daný image. Příklad tohoto souboru je k vidění ve výpisu kódu 3.7, konkrétně se jedná o Dockerfile pro službu *api-service*. V tomto souboru se mimo jiné nastavují veškeré proměnné prostředí, které aplikace potřebuje k správnému běhu.

Výpis kódu 3.7 Dockerfile vztahující se ke službě business-service

```
# Build the base image from redhat ubi8 python-39
FROM registry.access.redhat.com/ubi8/python-39:1

# Install requirements
# Copy files to the /app folder in the container
COPY ./ /app/
COPY ./requirements.txt /app/requirements.txt

# Set the working directory in the container to be /app
WORKDIR /app

# Upgrade pip to latest version
RUN pip3 install --upgrade pip

# Install the packages from requirements.txt in the container
RUN pip install --no-cache-dir -r requirements.txt

# Expose port 8000
ENV PORT=8000
EXPOSE 8000

ENV HOST="0.0.0.0"
ENV MONITOR="https://monitor.statnipokladna.cz/api/monitorws"
ENV STORE_SERVICE="https://scoring-store-service.azurewebsites.net"
ENV ARES="https://ares.gov.cz/ekonomicke-subjekty-v-be/rest/ekonomicke-
    subjekty"

# Start service
CMD ["python", "main.py"]
```

Jakmile máme v našem projektu vytvořený správný Dockerfile, můžeme přejít k implementaci automatizovaného nasazování na cílovou platformu. Na rozdíl od CI procesu má GitHub Action provádějící automatizované nasazování kódu svá specifika, lišící se pro každou službu. Vytvořený job se skládá z celkem šesti kroků:

1. Prvním krokem je stejně jako u *build* workflow checkout kódu do větve *master*.
2. Druhým krokem je již přihlášení na Azure pomocí přihlašovacích údajů, které jsou uložené v GitHubu v takzvaných *secrets* – zjednodušeně se jedná o jakési šifrované proměnné, které se nejčastěji používají právě k autentizaci.
3. Třetí krok je také přihlášení, ovšem tentokrát do entity zvané *Azure Container Registry* – jedná se o repozitář v rámci platformy Azure, do kterého se nahraje vytvořený Docker Image. Zde je potřeba specifikovat správný *Azure Container Registry* – všechny služby mohou být nahrány v jednom, ovšem pod rozdílnými názvy.
4. Čtvrtý krok je build kódu do *Docker Image* a následný deploy tohoto image právě do *Azure Container Registry*.
5. Pátý krok nasazuje již samotnou aplikaci – bere jí z *Azure Container Registry* a nasazuje na platformu *Web App*, která musí být vytvořena v rámci Azure předplatného a jejíž jméno je specifikováno v tomto kroku pod parametrem *app-name*.

6. Finálním krokem je odhlášení z Azure.

Popsaná workflow je k vidění ve výpisu kódu 3.8. Proces s přihlášením a nasazením samotné aplikace do Azure bohužel není možný se studentským předplatným, tudíž je skutečná workflow u jednotlivých služeb zjednodušená – nasazuje kontejner pouze do *Azure Container Registry*, neboť se GitHub dále nedostane a pak je potřeba nasadit buď ručně a nebo nasazovat z GitHubu stále se stejným tagem, aby došlo k automatickému nasazení. S klasickým předplatným ovšem toto není problém.

3.2.2 Code Review

GitHub je nejen výbornou VCS platformou a platformou umožňující automatizovat nejrůznější procesy, ale v rámci *pull requestů* umožňuje i zavést do naší organizace efektivní proces *code review*. To, čeho se tento proces týká, je popsáno v kapitole Analýza v sekci 1.5.3.

Code review tradičně probíhá v rámci procesu CI, konkrétně ještě předtím, než zavedeme nový kód do hlavní codebase (v terminologii GitHubu provádíme merge větve vedlejší do větve hlavní). GitHub krom pravidel určujících, že není možné provést merge, pokud úspěšně neproběhnou nejrůznější automatizované kontroly, umožňuje i vynutit proces schvalování od ostatních vývojářů. Pokud tedy naše změna není schválená od zvoleného počtu kolegů, nemůžeme tuto změnu zanést do hlavní codebase. Tím motivujeme jednotlivé inženýry k tomu, aby se na kód, který jejich kolega vytvořil, alespoň podívali, neboť schválením tohoto kódu přebírají určitou část zodpovědnosti za jeho správnost. Na obrázku 3.1 je k vidění, jak GitHub blokuje proces, dokud nedojde alespoň k jednomu schválení nové změny.

Kolegové mohou během procesu code review psát komentáře k novému kódu, vyžadovat změny, doptávat se na nejasnosti atd. Poté, co někdo z kolegů změnu schválí, možnost provést merge se opět odemkne. Na obrázku 3.2 je uživatelské rozhraní GitHubu, respektive stav daného pull requestu poté, co je změna schválena daným počtem kolegů.

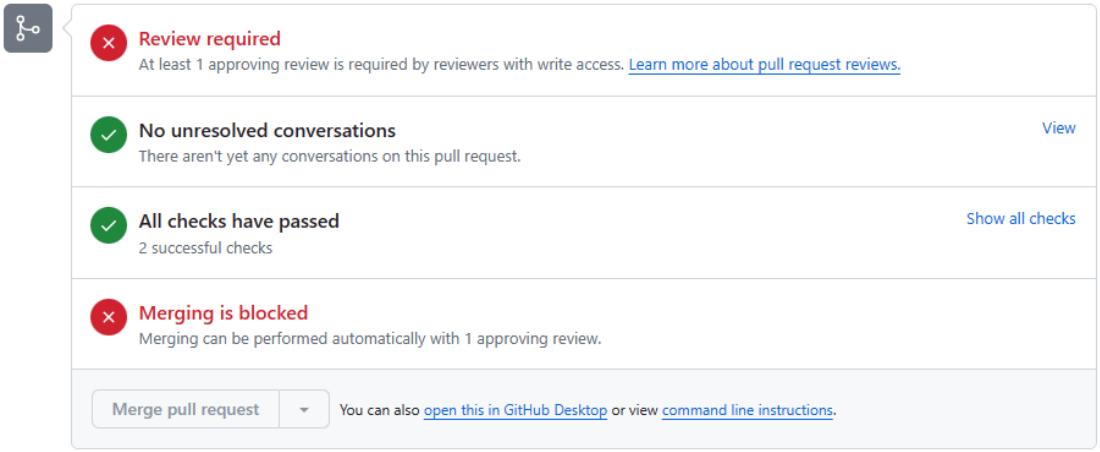
3.2.3 Dokumentace

V rámci vývoje byla vytvářena vývojářská dokumentace v podobě komentářů jednotlivých tříd a metod. Důležité metody a třídy jsou okomentovány pomocí *docstrings*, což jsou speciální typy komentářů uzavřené v trojitých uvozovkách, určených právě k dokumentaci kódu. Existuje řada nástrojů (například dříve zmíněný nástroj *Sphinx*), které z takto okomentovaného kódu mohou vytvářet dokumentaci v nejrůznějších formátech, ať už jde o html, pdf či obyčejný textový soubor. Příklad takto okomentovaného kódu je ve výpisu kódu 3.9.

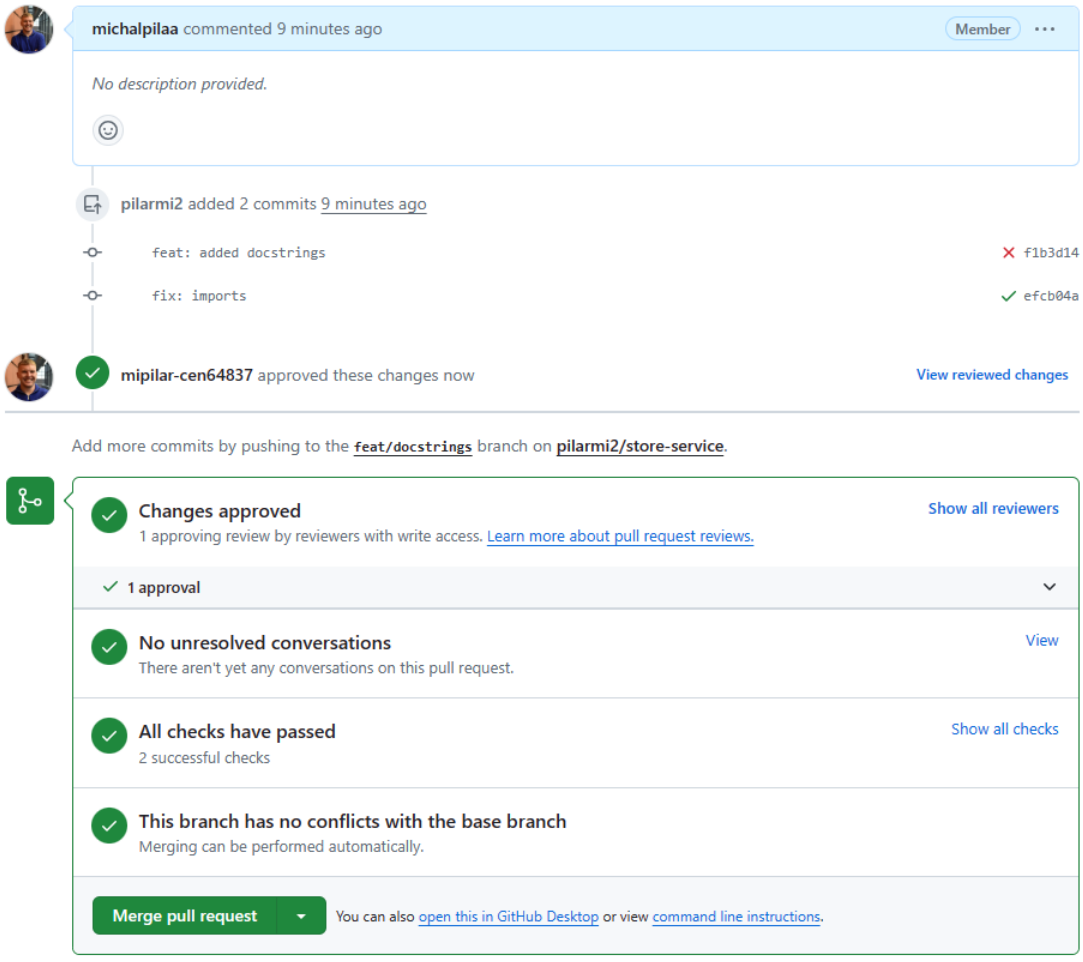
Další vytvořenou dokumentací, která je také určená vývojářům, je soubor *README.md*, který je umístěn v kořenovém adresáři každého repozitáře. Tento soubor popisuje základní požadavky a postup nutný k tomu, aby šlo kód zprovoznit na lokálním prostředí každého vývojáře. Tento typ dokumentace usnadní novým vývojářům se rychleji začlenit do projektu a začít pracovat na jeho rozšiřování.

Výpis kódu 3.8 YAML kód definující GitHub Action pro nasazení kódu do Azure

```
- name: Deploy to Azure

on:
  push:
    branches:
      - master

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:

    - name: Checkout GitHub Actions
      uses: actions/checkout@master


    - name: Login via Azure CLI
      uses: azure/login@v1
      with:
        client-id: ${{ secrets.AZURE_CLIENT_ID }}
        tenant-id: ${{ secrets.AZURE_TENANT_ID }}
        subscription-id: ${{ secrets.AZURE_SUB_ID }}


    - name: Login to Azure Container Registry
      uses: azure/docker-login@v1
      with:
        login-server: pilarmi2.azurecr.io
        username: ${{ secrets.REGISTRY_USERNAME }}
        password: ${{ secrets.REGISTRY_PASSWORD }}


    - name: Build and push container image to registry
      run: |
        docker build . -t storeservice.azurecr.io/store-service:${{ github.sha }}
        docker push storeservice.azurecr.io/store-service:${{ github.sha }}


    - name: Deploy to App Service
      uses: azure/webapps-deploy@v2
      with:
        app-name: 'pilarmi2-store-service'
        images: 'storeservice.azurecr.io/store-service:${{ github.sha }}'
        slot-name: 'staging'


    - name: Azure logout
      run: |
        az logout
```



Obrázek 3.1 Ukázka zablokování akce *merge*, dokud nedojde alespoň k jedné revizi



Obrázek 3.2 Ukázka pull requestu po požadovaném počtu schválení

Výpis kódu 3.9 Ukázka psaní kódu včetně *docstring* komentářů

```
class Repository(ABC):
    """
    Abstract base class for repositories, defining the interface
    for working with data.

    Implementations of repositories must provide methods for:
    - retrieving all items
    - retrieving an item by ID
    - retrieving an item by ID and period
    - adding or updating an item.

    This class inherits from ABC (Abstract Base Class) from the abc module
    and contains abstract methods that must be implemented by subclasses.
    """


    @abstractmethod
    def get_by_id(self, object_id: int):
        """
        Abstract method to retrieve an item from the repository by its ID.

        Args:
            object_id (int): The ID of the object to retrieve.

        Returns:
            object: The object corresponding to the given ID.
        """
        pass
```

Kapitola 4

Testování

Pokud používáme architekturu mikroslužeb, nestačí jen správně celý systém testovat, ale i mít co největší množství testů automatizované. Tato kapitola se věnuje právě testování – nejdříve jsou popsány automatizované testy jednotlivých služeb, následně testy jednotlivých HTTP endpointů a ve finále je zde popsán proces statické analýzy kódu a k tomuto účelu použitý nástroj.

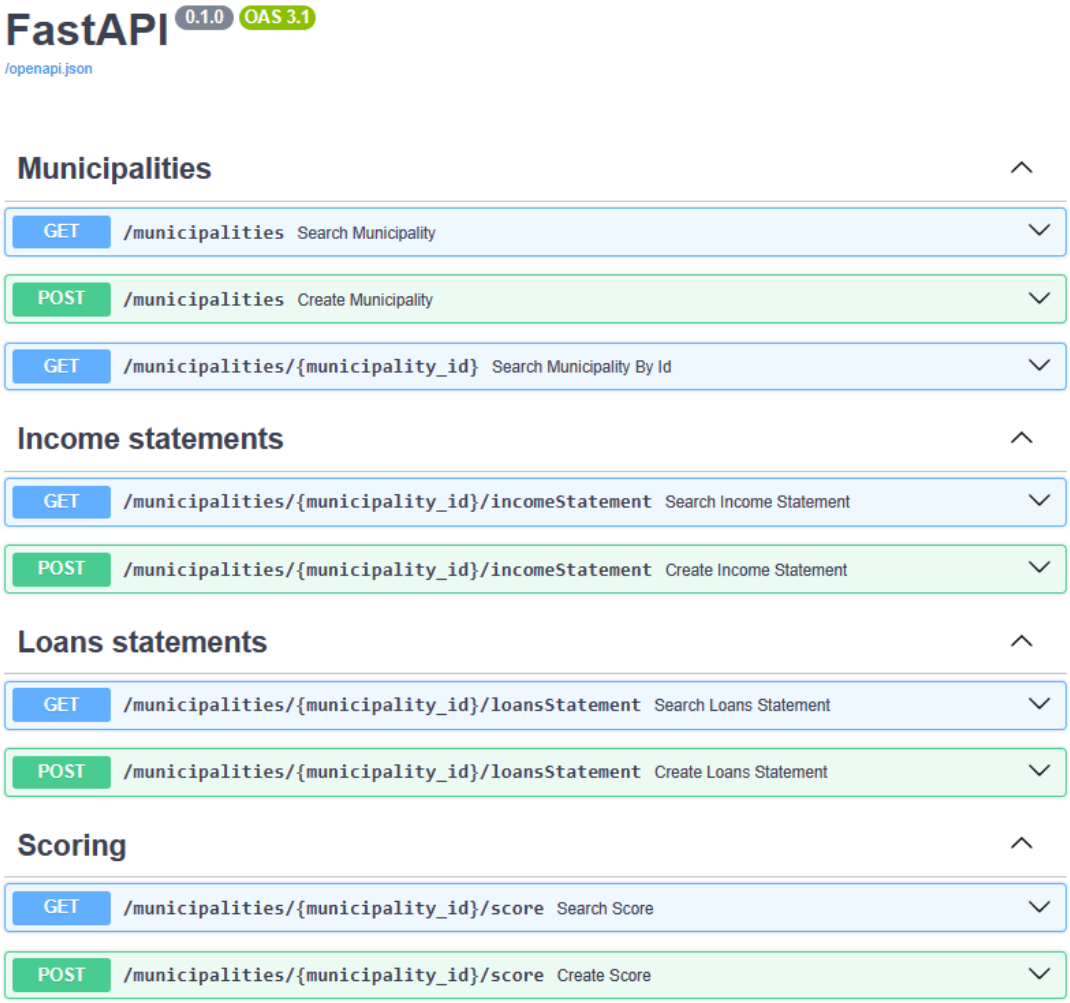
4.1 Unit testy

Jak bylo popsáno v kapitole *Analýza* v části 1.5.5, základní jednotkou funkčních testů jsou unit testy, které spočívají v izolovaném testování jednotlivých komponent nebo jednotek kódu s cílem ověřit jejich správnou funkci. Každá vyvinutá služba je tedy pokryta automatizovanými unit testy, které si může vývojář spouštět na svém lokálním prostředí během vývoje, aby si ověřil, zda změny, které v kódu provádí, nerozbijí stávající funkcionalitu. Tyto testy se ale primárně spouští v rámci CI/CD pipeline, konkrétně při každém vytvoření pull requestu na platformě GitHub. Pokud úspěšně neproběhnou všechny testy, vývojář nemůže sloučit změny z jeho nově vytvořené větve do větve hlavní.

Vytvořené testy jsou v každém repozitáři v adresáři *tests/unit* a poté jsou rozděleny do podadresářů podle toho, jakou funkcionalitu testují. Tyto testy jsou následně automatizovaně spouštěny pomocí nástroje *pytest* v rámci vytvořené CI/CD pipeline. Ve výpisu kódu 4.1 je k vidění příklad těchto unit testů – konkrétně dvou testů, které ověřují správnou funkčnost třídy *MunicipalityRepository*.

Výpis kódu 4.1 Unit testy testující třídu *MunicipalityRepository*

```
1 class MunicipalityRepositoryTest(unittest.TestCase):
2     def test_update_municipality(self):
3         repository: Repository = MunicipalityRepository()
4         repository.add_or_update(Municipality(municipality_id="1", name="
    Sample Municipality", citizens=1000))
5         assert repository.get_by_id("1").name == "Sample Municipality"
6         repository.add_or_update(Municipality(municipality_id="1", name="
    Prague", citizens=1000))
7         assert repository.get_by_id("1").name == "Prague"
```



Obrázek 4.1 OpenAPI specifikace pro store-service

4.2 HTTP API testy

Pro zjednodušení testování HTTP API enpointů vytvořených služeb vznikl repozitář na adrese . Obsahem tohoto repozitáře je konfigurační soubor pro nástroj *Docker Compose* s názvem *docker-compose.yml*. Ten slouží k vytvoření *Docker Images* všech služeb a k jejich následnému spuštění v *Docker* kontejnerech na lokálním prostředí. V tomto repozitáři se také nachází soubor *README.md*, ve kterém je popsáno, jak *Docker Compose* spustit.

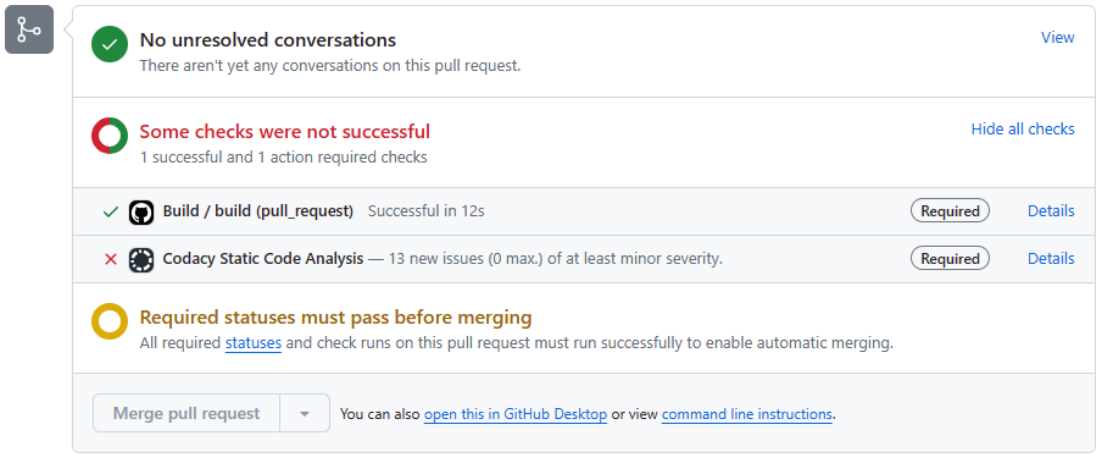
Knihovna *FastAPI* vytváří při spuštění aplikace OpenAPI specifikaci, kterou následně zpřístupňuje na adrese */docs* (například pro vytvořenou službu store-service hostovanou na platformě Azure najdeme specifikaci na adrese *scoring-store-service.azurewebsites.net/docs*). Díky tomu můžeme jednoduše testovat funkčnost našich API rozhraní přímo z prohlížeče. Zmíněná specifikace, konkrétně pro službu *store-service* je k vidění na obrázku 4.1.

K provedení kompletního *end-to-end* testu je možné si provolat HTTP endpoint služby backend: *GET /municipalities/{municipality_id}/score*. Při správné funkčnosti a použití správného *municipality_id*, kterým by mělo být IČO dané obce (například pro obec Ostrava je to *00845451*), aplikace vrátí objekt *Score* ve formátu JSON. Tento test je možné provést buď na lokálním prostředí po spuštění všech služeb a nebo po nasazení do Microsoft Azure na odpovídající adrese.

4.3 Statická analýza kódu

Ke statické analýze kódu byl použit dříve představený nástroj *Codacy*, který umožňuje provádět tuto analýzu zdarma pro open source projekty, bez nutnosti sám si danou platformu hostovat. Stačí si na stránkách *www.codacy.com* vytvořit uživatelský účet, následně propojit aplikaci se svou organizací na platformě GitHub, čímž proběhne i samotná instalace na této platformě (lze vybrat buď konkrétní repozitáře, kde chceme aplikaci používat, nebo celou organizaci). Jakmile aplikaci do své GitHub organizace nainstalujeme, při každém novém pull requestu nám proběhne statická analýza kódu a vytvoří se report, který se zobrazí v uživatelském rozhraní *Codacy*.

Statická analýza kódu je další pojistkou, kterou můžeme použít při zavádění nových změn do codebase. Ať už se jedná o *Codacy* či o jinou aplikaci, lze si nastavit takzvané „Brány kvality“ (běžně je používaný anglický termín *quality gates*), přes které když náš kód neprojde, neumožníme vývojářům zanést změny do hlavní codebase. Na obrázku 4.2 je vidět, jak vypadá pull request, pokud tato kontrola hlásí chybu – dokud vývojář danou chybu nevyřeší, není možné provést *merge* do hlavní větve. Na nástroje pro kontrolu kvality ovšem není dobré spoléhat, jako na jediný zdroj pravdy. I ve chvíli, kdy žádné chyby neodhalí, neznamená to, že v kódu žádné chyby nejsou. Proto je důležité aplikaci i kvalitně testovat a dbát na dodržování procesu *code review*.



Obrázek 4.2 Ukázka neúspěšné statické analýzy kódu

4.4 Shrnutí

Testování je klíčovým elementem zdravého životního cyklu každé aplikace. Kdyby se nejednalo o prototyp, ale o aplikaci v reálném bankovním systému, aplikace by byla testována ještě důkladněji – přibyly by například integrační testy mezi naším systémem a ostatními systémy, které v rámci své činnosti konzumují námi vypočítané skóre. Vzhledem k povaze systému by také mohly být prováděny *performance testy* aby se ověřilo, zda klientské aplikace nečekají na výpočet skóre příliš dlouho.

Závěr

Tato práce měla za cíl představit nejrůznější procesy a nástroje napomáhající vyvíjet udržitelný software, tedy software, který bude co nejlépe odolávat působení času a na kterém nebude vznikat velký technický dluh.

Práce byla primárně zaměřena na teoretickou část. V ní jsou popsány základní pojmy a myšlenky nutné k pochopení celého textu. Následně se věnuje tématům souvisejícím především s programováním – tedy principům dobrého návrhu a návrhovým a architektonickým vzorům. Poté jsou již popsány klíčové procesy softwarového inženýrství, do kterých mimo jiné spadá například testování nebo CI/CD pipeline. V závěru této části jsou popsány nástroje, které je možné použít při implementaci dříve popsaných procesů.

Praktická část je zaměřena na tvorbu softwaru, na kterém mají být předvedeny některé z popsaných procesů, technologií a nástrojů. Jako vytvářený software byla zvolena imitace bankovního systému, která má na starosti hodnocení rizikovosti obcí, respektive municipalit. Tento systém byl nejprve navržen a následně implementován.

Z popsaných návrhových a architektonických vzorů byla předvedena praktická ukázka architektury mikroslužeb a v rámci jedné z těchto služeb byl implementován návrhový vzor Repository, popsaný v části 1.3.3. Co se týče procesů, hlavním cílem bylo pro jednotlivé služby předvést implementaci kompletní CI/CD pipeline, která umožní při každé změně spustit automatizované testy a další kontroly, aby mohlo následně dojít k automatickému nasazení nového kódu na platformu Microsoft Azure. Tato pipeline byla pro všechny služby implementována a popsána. Zároveň byl představen proces code review prováděn s použitím platformy GitHub. K výslednému kódu také vznikla vývojářská dokumentace v podobě docstring komentářů a souboru README.md, který je obsažen v repozitáři každé služby.

Vzhledem k tomu, že v rámci této práce vznikl pouze prototyp aplikace, který byl určen primárně pro demonstraci popsaných postupů, principů a nástrojů, má výsledný software určitě prostor ke zlepšení. Mezi možná budoucí vylepšení rozhodně patří přidání autentizace do HTTP komunikace, aby nemohl jednotlivé HTTP endpointy používat kdokoliv a aby endpointy, které nemají být vystaveny ven ze systému, nebylo možné použít, pokud nejsme určený konzument. Dalším rozšířením je samozřejmě přidání klasické databáze. Vzhledem k tomu, že pro nasazení je použita platforma Microsoft Azure se nabízí například využití databázové technologie Azure Cosmos DB. Zároveň by do budoucna dávalo smysl rozšířit komunikaci mezi službou komunikující s portálem Ministerstva financí a službou napojenou na databázi o Event brokera (například Apache Kafka) s cílem učinit tuto komunikaci asynchronní, ovšem pro účely této práce nebylo ani jedno z těchto rozšíření stěžejní. Jednotlivé služby jsou implementovány tak, aby se jednalo o funkční, snadno rozšiřitelné a snadno přepoužitelné řešení daných problémů. Díky tomu mohou do budoucna posloužit jako základní stavební kámen při vývoji služeb s podobným účelem, ať už se bude jednat o čtení a manipulaci dat z externí webové služby a nebo o práci s databází.

Hlavním cílem bylo tedy představit nástroje a procesy softwarového inženýrství, které na-

pomáhají vyvíjet kvalitní a udržitelný software. Práce má být návodem primárně pro ty, pro které je řemeslo softwarového inženýrství nové a kteří by se mohli cítit ve změti pojmů, procesů a technologií ztraceni. Softwarové inženýrství je totiž širokým pojmem, pod kterým chápeme velké množství procesů a technologií. Cíl této práce, tedy popsat alespoň základy těchto procesů a technologií, které by měl znát každý softwarový inženýr, byl splněn.

Bibliografie

1. WINTERS, Titus; MANSHRECK, Tom; WRIGHT, Hyrum. *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media, 2020.
2. SHELDON, Robert. *What is a codebase (code base)? – TechTarget definition*. TechTarget, 2023. Dostupné také z: `https://www.techtarget.com/whatis/definition/codebase-code-base`.
3. GUPTA, Vikram. *What is a repository?* 2023. Dostupné také z: `https://builtin.com/software-engineering-perspectives/repository`.
4. MARTIN, Robert Cecil. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
5. GUDABAYEV, Tamerlan. *Separation of concerns the simple way*. DEV Community, 2021. Dostupné také z: `https://dev.to/tamerlang/separation-of-concerns-the-simple-way-4jp2`.
6. MARTIN, Robert C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
7. TELANG, Tarun. What are YAGNI, DRY and KISS principles in software development? *Educative*. [B.r.]. Dostupné také z: `https://www.educative.io/answers/what-are-yagni-dry-and-kiss-principles-in-software-development`.
8. HELM, Richard; JOHNSON, Ralph E; GAMMA, Erich; VLISSIDES, John. *Design patterns: Elements of reusable object-oriented software*. Braille Jymico Incorporated Quebec, 2000.
9. LINJANJA, Pacifique. *Design patterns for modern backend development – with example use cases*. freeCodeCamp.org, 2023. Dostupné také z: `https://www.freecodecamp.org/news/design-pattern-for-modern-backend-development-and-use-cases/#importance-of-design-patterns-in-software-development`.
10. PANTA, Anish Bilas. *Understanding repository pattern with implementation: A step-by-step guide*. Medium, 2023. Dostupné také z: `https://medium.com/@pantaanish/understanding-repository-pattern-with-implementation-a-step-by-step-guide-ca1bf36be3b4`.
11. ROUT, Pranaya. *Repository design pattern in C#*. 2023. Dostupné také z: `https://dotnettutorials.net/lesson/repository-design-pattern-csharp/?utm_content=cmp-true`.
12. KARIA, Bhavya. *A quick intro to dependency injection: What it is, and when to use it*. freeCodeCamp.org, 2020. Dostupné také z: `https://www.freecodecamp.org/news/a-quick-intro-to-dependency-injection-what-it-is-and-when-to-use-it-7578c84fa88f/`.

13. RADZEVICH, Aliaksei. *Compile time dependency injection in C++: Managing dependencies without late binding*. Medium, 2023. Dostupné také z: https://medium.com/@aliaksei.radzevich/compile-time-dependency-injection-in-c-managing-dependencies-without-late-binding-4338e0afcc44.
14. GANGRADE, Aashi. *Observer design pattern*. Medium, 2024. Dostupné také z: https://medium.com/@aashigangrade06/observer-design-pattern-0ee836b1b0d8.
15. GHERGU, Eduard. *The Art of Decorating: Applying the Decorator Design Pattern in Real Life*. 2023. Dostupné také z: https://www.pentalog.com/blog/design-patterns/decorator-design-pattern/.
16. TORBIIEVSKYI, Oleksandr. *Modern Software Architecture Patterns: The Main Things to Know*. 2023. Dostupné také z: https://ideasoft.io/blog/modern-software-architecture-patterns/.
17. SINGH, Jetinder. *The what, why, and how of a microservices architecture*. Hashmap, an NTT DATA Company, 2018. Dostupné také z: https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9.
18. SIMON, Jacob. *Microservices and distributed systems are unreliable. here's what to do about it*. 2022. Dostupné také z: https://www.tryexponent.com/blog/microservices-and-distributed-systems-are-unreliable-heres-what-to-do-about-it.
19. RODDEWIG, Stephen. *What is event-driven architecture? everything you need to know*. HubSpot, 2023. Dostupné také z: https://blog.hubspot.com/marketing/event-driven-architecture.
20. AMMAR, Syed. *Coding rules and standards in software development*. Bazaar Engineering, 2022. Dostupné také z: https://medium.com/bazaar-tech/coding-rules-and-standards-in-software-development-503c4ae6d39c.
21. RABBOLINI, Omar. *Knowledge sharing in software engineering teams*. The Startup, 2019. Dostupné také z: https://medium.com/swlh/knowledge-sharing-in-software-engineering-teams-efc69cf1d04a.
22. MENDES, Alexandra; FERREIRA, Rodrigo. *What is code review and when should you do it?* Blog | Imaginary Cloud, 2023. Dostupné také z: https://www.imaginarycloud.com/blog/what-is-code-review-and-when-should-you-do-it/.
23. ORAGUI, David. *Software documentation best practices*. 2024. Dostupné také z: https://helpjuice.com/blog/software-documentation.
24. SADERI, Payam. *Why we should write a good readme ?* Medium, 2023. Dostupné také z: https://medium.com/@saderi/why-we-should-write-a-good-readme-409ca0ae4d3.
25. CHOUDARY, Archana. *Get started with the different types of software testing*. Edureka, 2020. Dostupné také z: https://medium.com/edureka/types-of-software-testing-d7aa29090b5b.
26. LEED SOFTWARE DEVELOPMENT. *A guide to different types of software testing*. Medium, 2024. Dostupné také z: https://leeddev.medium.com/a-guide-to-different-types-of-software-testing-beb30e351755.
27. SACOLICK, Isaac. *What is Ci/CD? continuous integration and continuous delivery explained*. InfoWorld, 2024. Dostupné také z: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html.
28. AZERI, Izzy. *What is Ci/CD?* mabl, 2021. Dostupné také z: https://www.mabl.com/blog/what-is-cicd.
29. MOHANAN, Remya. *CI/CD definition, process, benefits, and best practices*. 2022. Dostupné také z: https://www.spiceworks.com/tech/devops/articles/what-is-ci-cd/.

30. RAO, Anjana. *What is version control? definition, types, systems and Tools*. 2023. Dostupné také z: https://blog.logrocket.com/product-management/version-control-systems-definition-types/.
31. MILLER, James. *Which version control should developers use?* 2021. Dostupné také z: https://www.bairesdev.com/blog/which-version-control-system-developers-use/.
32. WICKRAMASINGHE, Shanika. *GitHub, GitLab, Bitbucket & Azure Devops: What's the difference?* 2021. Dostupné také z: https://www.bmc.com/blogs/github-vs-gitlab-vs-bitbucket/.
33. KHARLANTSEVA, Maria. *What Is GitHub? Everything You Need to Know*. 2024. Dostupné také z: https://everhour.com/blog/what-is-github/.
34. BIGELOW, Stephen J.; COURTEMANCHE, Meredith; GILLIS, Alexander S. *What is DevOps? the ultimate guide*. TechTarget, 2024. Dostupné také z: https://www.techtarget.com/searchitoperations/definition/DevOps.
35. KELLEY, Karin. *What is GitLab and how to use it? 2024: Simplilearn*. Simplilearn, 2024. Dostupné také z: https://www.simplilearn.com/tutorials/git-tutorial/what-is-gitlab.
36. REDHAT. *What is containerization?* 2021. Dostupné také z: https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization.
37. GANVIR, Anshul. *Introduction to docker*. Medium, 2023. Dostupné také z: https://medium.com/@anshulganvir/introduction-to-docker-337b9d09a079.
38. DINU, Flavius; NINAWE, Sumeet. *13 most useful CI/CD tools for DevOps Engineers in 2024*. 2024. Dostupné také z: https://spacelift.io/blog/ci-cd-tools.
39. BENVEGNÙ, Davide. *5 top reasons to use github actions for your next project*. DEV Community, 2023. Dostupné také z: https://dev.to/n3wt0n/5-top-reasons-to-use-github-actions-for-your-next-project-cga.
40. CORRALES, Enrique. *Gitlab CI/CD Tool Review*. 2024. Dostupné také z: https://www.techrepublic.com/article/gitlab-review/.
41. LEVAN, Michael. *5 advantages of GitLab CI/CD pipelines*. 2020. Dostupné také z: https://www.techtarget.com/searchsoftwarequality/video/5-advantages-of-GitLab-CI-CD-pipelines.
42. PATADIYA, Jaydeep. *Azure Devops Pipelines: An in-depth Introduction & Analysis*. Radixweb, 2024. Dostupné také z: https://radixweb.com/blog/introduction-to-azure-devops-pipelines.
43. IVANENKO, Viktoriia. *Best 7 automated software testing tools in 2024*. 2024. Dostupné také z: https://mailtrap.io/blog/best-automation-testing-tools/.
44. MIGUEL, Paulo Gardini. *12 best code analysis tools in 2024*. 2024. Dostupné také z: https://thectoclub.com/tools/best-code-analysis-tools/.

Obsah příloh