Umělá inteligence v softwarovém inženýrství

Jakub Vlček

Bakalářská práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jakub Vlček**
Osobní číslo:	**A19785**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Umělá inteligence v softwarovém inženýrství**
Téma práce anglicky:	**Artificial Intelligence in Software Engineering**

Zásady pro vypracování

1. Vypracujte literární rešerši na dané téma.
2. Popište softwarové procesy a klíčové fáze procesů.
3. Identifikujte klíčové oblasti využití umělé inteligence v softwarovém inženýrství.
4. Proveďte analýzu přínosů metod umělé inteligence, ve vybraných oblastech využití.
5. Navrhněte softwarový proces, který bude využívat zjištění z provedené analýzy.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. DAVIS, Ernest, Douglas D. EDWARDS, Davis FORSYTH, et al. Artificial Intelligence: A Modern Approach. Third Edition. Upper Saddle River, New Jersey 07458: Pearson Education, 2010. ISBN 978-0-13-604259-4.
2. BATARSEH, Feras A. a Ruixin YANG. Data Democracy: At the Nexus of Artificial Intelligence, Software Development, and Knowledge Engineering. London: Academic Press, 2020. ISBN 978-0-12-818366-3.
3. SOMMERVILLE, Ian a Jakub GONER. Softwarové inženýrství. Brno: Computer Press, 2013. ISBN 978-80-251-3826-7.
4. BARENKAMP, Marco, Jonas REBSTADT a Oliver THOMAS. Applications of AI in classical software engineering [online]. 26 July 2020, , 1-15 [cit. 2021-11-26]. Dostupné z: doi:https://doi.org/10.1186/s42467-020-00005-4
5. KALECH, Meir, Rui ABREU a Mark LAST. Artificial Intelligence Methods for Software Engineering. Singapore: World Scientific Publishing Company, 2021. ISBN 978-981-123-991-5.

Vedoucí bakalářské práce: **doc. Ing. Radek Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání bakalářské práce: **3. prosince 2021**
Termín odevzdání bakalářské práce: **23. května 2022**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 9. 5. 2022 Jakub Vlček, v. r.

ABSTRAKT

Tato práce se zabývá propojením oblastí softwarového inženýrství a umělé inteligence. Zabývá se klíčovými fázemi softwarového inženýrství počínaje plánováním, přes analýzu, návrh, implementaci až po testování a údržbu. Cílem je zanalyzovat případy využití umělé inteligence v jednotlivých oblastech softwarového inženýrství a zhodnotit jejich přínos pro tvorbu softwarového produktu. Dále se práce věnuje návrhu softwarového procesu s využitím nástrojů založených na technikách umělé inteligence.

Klíčová slova: umělá inteligence, softwarové inženýrství, softwarový proces, životní cyklus softwaru

ABSTRACT

This thesis examines the interconnection of software engineering and artificial intelligence. It deals with the crucial phases of software engineering from planning, through analysis, design, implementation to testing and maintenance. The aim is a use-case analysis in areas of software engineering and evaluation of its contribution to the production of a software product. Furthermore, the thesis presents the design of a software process using tools based on artificial intelligence techniques.

Keywords: artificial intelligence, software engineering, software process, software development life cycle

Rád bych poděkoval vedoucímu mé bakalářské práce panu doc. Ing. Radku Šilhavému, Ph.D. za odborné vedení a cenné rady, které mi poskytl.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD

S rychlým vývojem vědy a nových technologií roste také poptávka po nových nástrojích, které by lidem usnadnily práci. Jednou z oblastí, která se stále více rozšiřuje, je vývoj softwaru. Se softwarem se dnes setkáváme každý den v nejrůznějších podobách a s různým způsobem jeho využití – od aplikací určených pro zábavu až po zařízení, na kterých jsou závislé lidské životy. Proto je nutné klást velký důraz na vhodné metody, které vedou k vytvoření kvalitního softwaru.

Softwaroví inženýři se musejí potýkat s řadou problémů, které musejí zohledňovat ve své práci a nalézat vhodné kompromisy. Tlak je nejen ze strany kvality, ale také výsledné ceny, rychlosti doručení, možností údržby atd. Proto se do jednotlivých fází životního cyklu vývoje softwaru stále více zapojují nové technologie, které mají lidem pomoci s řešením těchto problémů.

Mezi techniky, které se stále více rozšiřují do oblasti softwarového inženýrství, řadíme i ty, které označujeme souhrnným názvem jako umělá inteligence. Jedná se o počítačové přístupy, které napodobují lidskou inteligenci. Tato práce ukazuje propojení oblastí softwarového inženýrství a umělé inteligence.

Cílem práce je přiblížit klíčové fáze životního cyklu vývoje softwaru a poukázat na některé možnosti, které do této oblasti přináší techniky umělé inteligence. Práce postupně prochází šest fází softwarového procesu – plánování, analýzu, návrh, implementaci, testování a údržbu. Následně je navržen přístup pro využití některých aktuálně dostupných nástrojů pro softwarové inženýrství, které jsou založeny na principech umělé inteligence. Na závěr je popsán celkový pohled na provedenou analýzu a jsou zhodnoceny zjištěné výsledky.

Tato práce se nezaměřuje na jednu oblast softwarového inženýrství ani na konkrétní techniku z oboru umělé inteligence, ale mapuje obecný přehled na základě aktuálně dostupné odborné literatury. Použitá klasifikace různých technik či přístupů se může v různých případech lišit podle daného zaměření či autora publikace. Jednotné a obecně platné dělení není vždy možné nalézt. Podobně je tomu i s použitou terminologií. Ne všechny výrazy, které se v práci vyskytují, mají v češtině standardně používaný ekvivalent. Z tohoto důvodu jsou některé termíny ponechány v angličtině, případně je uvedena vysvětlivka.

1 SOFTWAROVÉ INŽENÝRSTVÍ

V druhé polovině 19. století sestrojil Charles Babbage [1] mechanický programovatelný stroj, který umožňoval matematické výpočty. Jeho cílem bylo omezit počet chyb ve výpočtech vzniklých lidským faktorem. Tento počin bývá označován jako první softwarový program. Cílem softwaru bylo pomoci lidem automatizovat procesy. Proces tvorby softwaru však zůstal převážně lidskou činností. Pojem softwarové inženýrství [2] se poprvé objevuje v roce 1968. Tehdy se uskutečnila konference podporovaná NATO věnovaná programátorské profesi. Byly zde diskutovány problémy tehdejšího vývoje softwaru a také stanoven termín softwarové inženýrství (software engineering).

Softwarové inženýrství [3] je technická disciplína, která se zabývá aspekty produkce softwaru. Tzn. že zahrnuje několik fází. Tyto fáze bývají označovány také jako procesy softwarového inženýrství[1]. Jedná se o posloupnost činností, které vedou k produkci výsledného softwarového produktu. Pro všechny tyto procesy jsou společné čtyři základní aktivity. Specifikace, vývoj, validace a evoluce softwaru. Jednotlivé vývojové procesy se liší podle typu systému. Na základě softwaru a zvoleném způsobu vývoje se rozlišuje různá úroveň podrobnosti zmíněných čtyř základních aktivit.

U technických disciplín by měl být kladen důraz na vhodné metody a nástroje. Mezi základní aspekty, které by měly být uvažovány, patří možnosti provádění údržby. Software by mělo být možné dále vyvíjet s ohledem na požadavky zákazníků. Spolehlivost a bezpečnost by měly být na takové úrovni, aby v případě selhání systému nedošlo k fyzickým nebo ekonomickým škodám. Také by mělo být znemožněno získat přístup k softwaru neoprávněným uživatelům či osobám, které by jej mohly poškodit. Další vlastností vhodně vyvíjeného softwaru je efektivita. Software by měl mít dostatečnou rychlost reakce a vhodně nakládat s přidělenými zdroji, jako je paměť nebo využití procesoru. V neposlední řadě by měl být software přijatelný pro uživatele, pro které je určen. Měl by být srozumitelný a použitelný.

V softwarovém inženýrství je důležitý systematický a organizovaný přístup, aby bylo dosaženo kvalitního produktu. Tento přístup bývá označován jako softwarový proces. Ten se vyznačuje základními aktivitami, mezi které patří: specifikace softwaru, návrh

[1] v anglické literatuře se používá termín Software Development Process

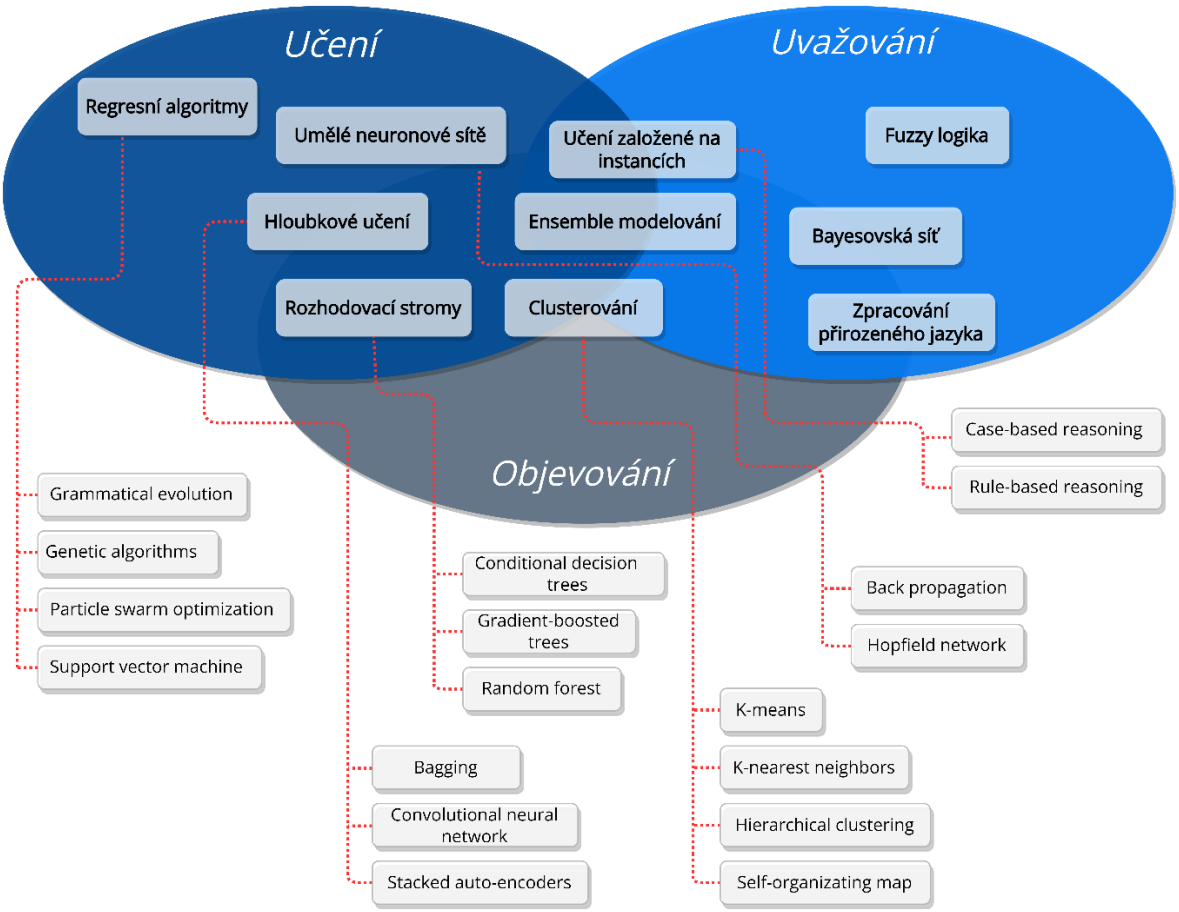
a implementace softwaru, validace softwaru a evoluce softwaru. Můžeme však nalézt i další podrobnější způsoby dělení těchto základních aktivit.

Vývoj softwaru probíhá v definovaných procesech [3]. Tyto procesy jsou označovány jako životní cyklus vývoje softwaru[2]. Existují obecné modely, někdy také označované jako paradigmata procesů. Model softwarového procesu podává abstraktní reprezentaci popisu procesu vývoje z určitého hlediska. Tuto strukturu lze rozšířit podle konkrétních procesů softwarového inženýrství. Existuje celá řada modelů vývoje softwaru, např. vodopádový model, inkrementální model, spirálový model aj.

[2] v anglické literatuře se používá termín Software Development Life Cycle (SDLC)

2 UMĚLÁ INTELIGENCE

Nalézt jedinou odpověď na otázku, co je umělá inteligence, není možné [4]. Podle Dictionary of Computer Science Engineering and Technology [5] zní definice pojmu umělá inteligence následovně: „Studie počítačových technik, které napodobují aspekty lidské inteligence, jako je rozpoznávání řeči, logické vyvozování a schopnost uvažovat z dílčích informací.“ Název umělá inteligence[3] byl tomuto oboru přiřazen teprve v roce 1956 [4]. Je tedy jedním z nejmladších odvětví vědy a inženýrství. V současnosti existuje obrovská škála disciplín, od obecných, jako je např. učení, až po konkrétní, kam patří např. řízení automobilů. Artificial Intelligence (AI) nabízí univerzální pole pro jakýkoliv intelektuální úkol.



Obrázek 1. Přehled vybraných metod umělé inteligence [6]

[3] v anglické literatuře Artificial Intelligence (AI)

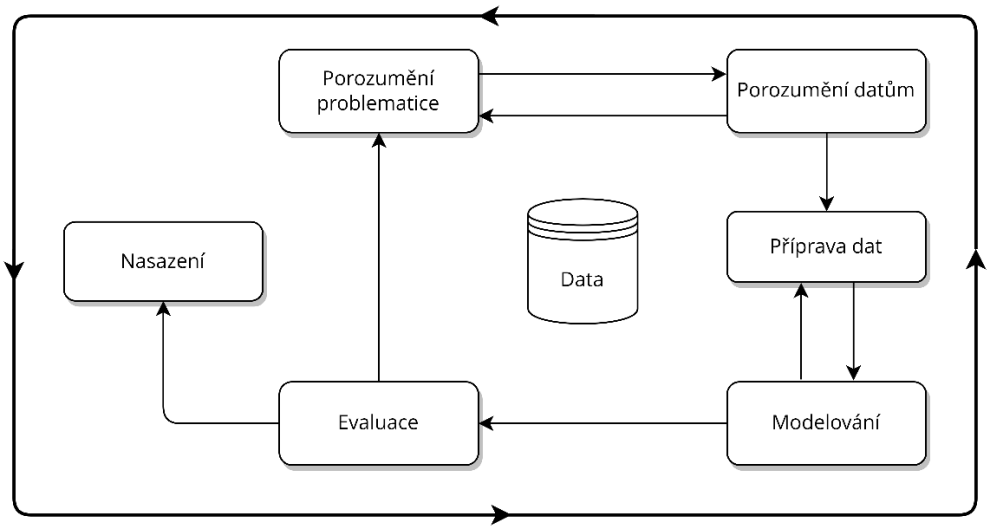
2.1 Učení

Získávání znalostí je typickým požadavkem pro inteligentní chování [6]. Efektivním způsobem, jak nabývat znalostí, je učení. Proto se studie umělé inteligence zabývají technikami učení. Cílem je umožnit počítačům automatické učení bez nutnosti zásahu člověka. Existuje řada metod, které se liší a jsou více či méně vhodné pro různé konkrétní scénáře.

Výkony jednotlivých metod souvisejí s množstvím a kvalitou dat, na základě kterých probíhá učení. Obecně lze říci, že čím více máme informací a čím méně obsahují šumu, tím lepších výsledků můžeme dosáhnout. Mezi skupiny technik, které se zabývají učením, můžeme zahrnout umělé neuronové sítě (artificial neural networks), evoluční algoritmy (evolutionary algorithms), hloubkové učení (deep learning), rozhodovací stromy (decision trees), aj.

2.2 Průzkum a získávání znalostí

Algoritmy pro získávání znalostí vyhledávají potenciální informace z databáze. Bývají označovány jako dobývání znalostí z databází (knowledge discovery in databases) [6]. Cílem těchto algoritmů je rozpoznat platné a potenciálně užitečné informace. Zahrnují vzorce, podle kterých se rozhoduje o tom, která data jsou užitečná a která nepoužitelná v rámci kontextu zájmu.



Obrázek 2. CRISP-DM model dobývání znalostí [6]

Proces dobývání znalostí lze rozdělit do šesti kroků znázorněných pomocí Cross-Industry Standard Process for Data Mining (CRISP-DM). Porozumění problematice, porozumění datům, příprava dat, modelování, vyhodnocení výsledků a využití výsledků. Mezi příklady

technik můžeme zařadit algoritmus k-nejbližších sousedů (k-nearest neighbor) nebo hierarchické shlukování (hierarchical clustering). Kroky dobývání znalostí jsou znázorněny na obrázku 2.

2.3 Rozhodování

Rozhodování na základě získaných znalostí vede k vytvoření závěru [6]. Zahrnuje použití logických technik, mezi něž patří indukce a dedukce. Cílem systémů, které zahrnují mechanismy rozhodování, je provádět úkoly na stejné úrovni, jako by je řešil člověk. Tyto systémy využívají heuristiku, tedy jen přibližné řešení založené na odhadu, aby zmenšily prostor pro hledání možného řešení.

Mezi tři hlavní části, na kterých bývají tyto systémy založeny, patří: získávání znalostí, vědomostní báze a inferenční (rozhodovací) engine. Systém získávání znalostí shromažďuje závěry a nová pravidla, které se používají pro další rozvoj. Znalostní báze obsahuje pravidla a informace. Patří sem vztahy, podmínky atd. Inferenční engine propojuje znalosti báze se shromážděnými znalostmi. Tento proces umožňuje rozhodování. Systém umožňuje ukládat znalosti do vědomostní báze. Díky tomu jsou nová řešení hledána se zohledněním předchozích případů. Mezi zástupce těchto technik patří rozhodování na základě pravidel (rule-based reasoning), případové rozhodování (case-based reasoning) nebo fuzzy logika (fuzzy logic).

3 KLÍČOVÉ FÁZE SOFTWAROVÉHO PROCESU

Životní cyklus vývoje softwaru můžeme rozdělit na šest fází [7], tedy šest softwarových procesů, aby zahrnovaly všechny kroky od studie proveditelnosti až po nasazení a údržbu. Jedná se o tyto fáze:

1. Plánování
2. Analýza
3. Návrh
4. Implementace
5. Testování a integrace
6. Údržba

3.1 Plánování

Na začátku softwarového projektu je tvorba jeho plánu [3]. Projektový plán zahrnuje přiřazení práce jednotlivým členům týmu. Díky plánu je možné sledovat postup prací a informovat zákazníka. V době návrhu se rozhoduje o prostředcích na dokončení práce a o určení ceny produktu.

Plánování se však prolíná i do dalších fází vývojového životního cyklu. Vyskytuje se v celém průběhu práce na projektu. Na základě postupu prací je možné získat přesnější informace, a upravit tak odhady stanovené na začátku vývoje. V průběhu prací se plán neustále vyvíjí a je potřeba jej udržovat stále aktuální. Navíc se průběžně mohou měnit softwarové požadavky. V případě agilních přístupů tvorby softwaru probíhá plánování častěji v kratších časových výhledech a je typické, že se často mění.

Přestože se obvykle jedná o první fázi vývoje softwaru, je plánování úzce spjato i s analýzou a některá literatura tyto dva procesy spojuje do jednoho. Dochází k diskuzi vývojářů a zákazníka za účelem stanovení cílů a požadavků. Tato jednání jsou důležitá pro určení technické proveditelnosti, ale také ekonomické efektivity celého projektu. V této fázi dochází k optimalizaci cílů projektu, aby bylo vyhověno všem omezením v závislosti na kvalitě produktu, době vývoje a také nákladech [8]. Tyto nároky jsou protichůdného charakteru. Vhodné plánování by mělo uvažovat všechny okolnosti a navrhnout optimální řešení. Odhad času potřebného k dokončení projektu je prováděn jen s určitou mírou účinnosti. Pro zpřesnění odhadů se nabízí využití umělé inteligence.

Mezi faktory ovlivňující správný odhad doby trvání patří zkušenosti odhadce. K větším nepřesnostem odhadů dochází v případech, kdy projekt obsahuje nejistoty [9]. Autoři Morgenshtern a kol. [10] uvádějí čtyři kategorie technik odhadů.

Odhad analogií – Jde o porovnání již známého řešení problému a jeho časové náročnosti s úkolem, jehož časovou náročnost chceme odhadnout. Nevýhodou této metody je problém nalezení vhodné analogické úlohy. V případě srovnatelných činností jde o rychlý proces.

Parametrické modely – Analýzou známých dat jsou vyhledány parametry, které ovlivňují odhad času a potřebného úsilí. Mezi parametry se mohou řadit např. velikost systému, zkušenosti týmu, použité nástroje aj. Může vzniknout nepřesnost v nevhodné definici parametrů. Patří sem metody jako COCOMO (Constructive Cost Model) k odhadování na základě počtu řádků zdrojového kódu nebo metoda funkčních bodů, která uvažuje počet entit a jejich složitost.

Odborný odhad – Jedná se o nejpoužívanější metodu, kdy odhadce používá osobní zkušenost. Nelze přesně definovat faktory, které takový odhad ovlivňují, protože vychází ze subjektivního názoru odhadce a závisí na jeho zkušenostech. Problém však bývá v často příliš optimistickém odhadu potřebného času.

Využití technik umělé inteligence – Počítač kombinuje odhad určený analogicky s parametrickými modely.

Umělá inteligence je přínosná k podpoře řízení softwarových projektů, a to zejména v přidělování úkolů v závislosti na dostupných lidských zdrojích. Konvenční plánovací modely musí přistupovat k zjednodušení vstupních faktorů a scénářů, aby přinesly uspokojivé výsledky. Algoritmy umělé inteligence, které jsou založeny na nelineárních a samooptimalizačních algoritmech, dokáží tyto problémy úspěšně řešit snižováním složitosti rozhodování [11].

Jedním z přístupů k optimalizaci nákladů a dosažení kvalitních výsledků mohou být algoritmy využívající Bayesovské sítě [11, 12]. Takové modely umožňují oproti jiným způsobům optimalizace zahrnout velké množství dalších faktorů a koeficientů, což umožňuje vyrovnat se s nedostatečnými, chybějícími daty a subjektivními úsudky. Vyžadují však přesnou matematickou formulaci sady problémů, proto jsou závislé na lidském plánování a analýze problému.

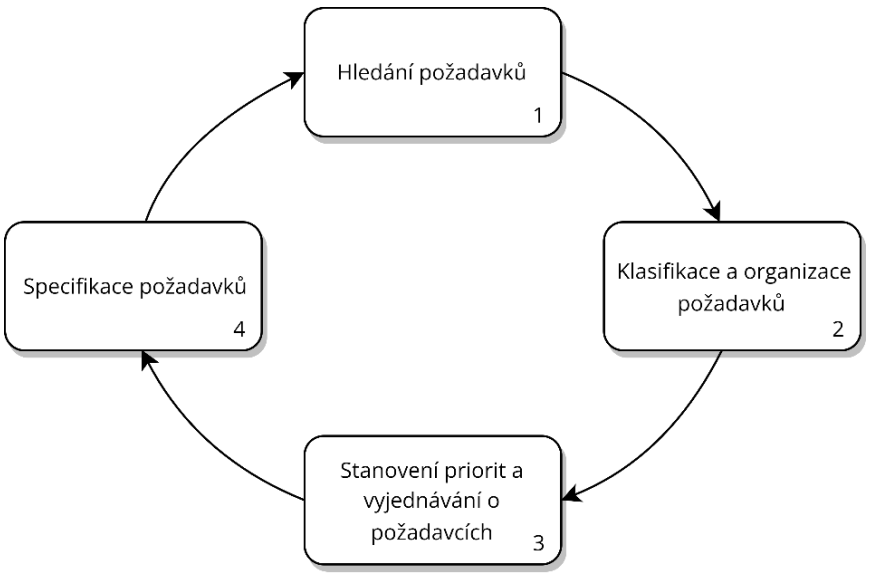
Jiné modely [8] spoléhají na samooptimalizační iterativní algoritmy. Mezi takové patří algoritmus optimalizace mravenčích kolonií, který je také možné využít k minimalizaci nákladů a trvání softwarových projektů. Mahdikův Algoritmus [11] optimalizace mravenčích kolonií iterativně aproximuje ideální pracovní zadání a poskytuje adaptivní časový a funkční plán ve formě seznamu úkolů a matice přidělování úkolů zaměstnancům. Tato matice průběžně optimalizuje přidělování zaměstnanců a plánování úkolů.

Jednou z preferovaných technik mezi softwarovými inženýry je odhad pomocí analogie. Umělou inteligenci lze využít k napodobení tohoto lidského přístupu. Azzeh a kol. [13] navrhli model, který využívá teorie fuzzy množin v kombinaci s Grey Relational Analysis (GRA). Predikce úsilí probíhá na základě známých projektů. GRA slouží k posouzení podobnosti mezi daty a teorie fuzzy množin ke snížení nejistoty v měření. [14].

Technik a přístupů k využití umělé inteligence v plánování softwarových projektů je velké množství [8] a jsou založeny na různých principech. K výběru vhodného plánovacího algoritmu pro konkrétní případ je vždy nutný lidský přístup.

3.2 Analýza

Další fází vývoje softwaru je proces analýzy požadavků. Tato disciplína bývá označována také jako inženýrství požadavků a probíhá v iteracích. Schéma níže (obrázek 3) od Sommerville [3] zobrazuje možnou podobu těchto kroků.



Obrázek 3. Schéma procesu zjišťování a analýzy požadavků [3]

Hledání požadavků – V tomto kroku probíhá interakce se zainteresovanými osobami za účelem hledání požadavků. Pro hledání požadavků je možné využít různé techniky, mezi které patří např. rozhovory, případy užití aj.

Klasifikace a organizace požadavků – Dochází k roztřídění a uspořádání nestrukturované sady požadavků do skupin. Organizace požadavků úzce souvisí s návrhem architektury.

Stanovení priorit a vyjednávání o požadavcích – Požadavkům jsou stanoveny priority. V případě nalezení konfliktů mezi požadavky je nutné tyto konflikty odstranit. Dochází k dohodě zainteresovaných osob pro nalezení kompromisu.

Specifikace požadavků – Probíhá dokumentace nalezených požadavků. Dokumentace může být vedena v různých podobách. Můžeme zde zařadit přirozený jazyk, strukturovaný přirozený jazyk, grafické znázornění (obvykle UML) či matematické specifikace.

Analýza požadavků je inženýrská disciplína [14], která klade důraz na používání systematických a opakovatelných technik. Musí zajišťovat úplné, konzistentní a relevantní výsledky. Je považována za jednu z klíčových fází životního cyklu vývoje softwaru, jelikož nejednoznačná a nekvalitní specifikace požadavků může vést k nasazení nežádoucího produktu, tedy zvýšit náklady na vývoj. Z tohoto důvodu se spolu s rostoucí velikostí a složitostí softwarových systémů zvyšuje i poptávka po využití umělé inteligence v analýze požadavků.

3.2.1 Zjišťování a analýza požadavků

Institute of Electrical and Electronics Engineers (IEEE) [14] definuje požadavek jako podmínku nebo schopnost potřebnou pro uživatele k řešení problému nebo dosažení cíle, který musí systém splňovat, aby splnil smlouvu nebo standard. Požadavky definují rámec pro proces vývoje softwaru. Specifikují, co musí systém dělat, jaké vlastnosti musí vykazovat, jaká musí splňovat omezení apod. Již od konce 70. let 20. století se výzkumné práce zabývají počítačovými nástroji, které pomáhají návrhářům. Ty by měly pomáhat lidem s formulací formálních a procesně orientovaných požadavků.

Blazer a kol. [15] prozkoumali řadu softwarových specifikací psaných neformální formou. Uvádí, že hlavní rozdíl mezi těmito specifikacemi a formálním ekvivalentem je používání neúplných popisů. Tyto částečné popisy je možné pochopit a doplnit díky okolnímu kontextu. Dílčí popisy směřují pisatelovu i čtenářovu pozornost na podstatu věci a zhušťují specifikaci. Mezi výhody využití neformálních specifikací patří mimo jiné jejich přesnost. Hlavní přínos počítačového nástroje pro podporu analýzy požadavků vidí v použití

počítačového nástroje, který by dokázal využívat kontext k dokončení těchto neúplných popisů. Autoři [14] tedy navrhli, aby jakékoliv potřebné změny byly prováděny právě v neformální specifikaci a o úpravy se staral systém.

Serena a kol. [16] uvádí tři přístupy pro řízení znalostí v inženýrství požadavků. Jeden z přístupů odpovídá sociální interakci (rozhovory, dotazníky, user stories, use cases). Může však docházet k nejasnostem kvůli nesprávné identifikaci požadavků a k vynechání některých nezbytných funkcí a naopak k implementaci funkcí, které se později jeví jako zbytečné. Tento přístup často není dostatečný pro identifikaci všech systémových požadavků.

Jiným přístupem je podpora dynamických herních technik, které se používají v agilních metodách vývoje softwaru. Tyto metody poskytují potřebnou zpětnou vazbu a aktivní účast aktérů. Dochází k lepší specifikaci požadavků a zvyšuje se následná kvalita a přijetí softwaru.

Přístup založený na metodách umělé inteligence nabízí podle Serena a kol. dva základní přínosné proudy.

1. Systémy doporučení zaměřené na identifikaci osob, které mohou poskytnout popis nezbytných požadavků, a také na doporučení požadavků dostupných k opětovnému použití.
2. Princip fuzzy logiky, jehož cílem je lepší reprezentace informací. Tento princip je založen na dřívějších zkušenostech. Fuzzy logika je využita pro interpretaci a rozhodování z dat získaných při vývoji, na základě kterých jsou generovány konceptuální mapy[4].

3.2.1.1 Systémy doporučení

INTELLIREQ je název prostředí, které představili Ninaus a kol. [18] ke snížení rizika nekvalitních požadavků. Prostředí je založeno na přístupech doporučení. Existují čtyři základní typy přístupů.

1. kolaborativní filtrování
2. filtrování založené na obsahu (pomocí klíčových slov)
3. doporučení založené na znalostech

[4] konceptuální mapy jsou grafický nástroj pro organizaci a reprezentaci znalostí [17]

4. doporučení pro skupiny

INTELLIREQ podporuje rané fáze inženýrství požadavků [14], kdy je hlavním cílem prioritizace požadavků. Systém zároveň rozpoznává možné závislosti mezi požadavky. Výsledkem je soubor požadavků s odhady úsilí a plán pro jejich implementaci.

3.2.1.2 Systémy pro reprezentaci informací

Jedno z prvních využití umělé inteligence [14] pochází z roku 1994. Byl představen systém, který pomáhal při správě požadavků – Systems Information Resource/Requirements Extractor (SIR/REX). První část – System Information Resource slouží k vytváření vazeb mezi fragmenty informací, čímž zjednodušuje čitelnost dokumentu psaného v přirozeném jazyce. Druhá část – Requirements Extractor využívá umělou inteligenci k analýze přirozeného jazyka. Řeší tím obvykle zdlouhavý úkol získání požadavků ze specifikace. SIR/REX nejprve analyzuje dokument psaný přirozeným jazykem a poté vybírá konkrétní věty, u kterých je vysoká pravděpodobnost, že by mohly být požadavky. Při porovnání výsledků extrakce požadavků tímto programem a člověkem bylo zjištěno, že algoritmus byl lepší jak v účinnosti, tak i v rychlosti oproti lidské práci.

Potenciál využití umělé inteligence v analýze požadavků nabízí disciplína označována jako Natural Language Processing (NLP). Z pohledu stroje je text pouze souborem písmen, slov a vět uspořádaných do určitých vzorů, které mají specifický význam. Vemuri a kol. [19] představili techniku, která dokáže identifikovat aktéry a případy užití. Využívají k tomu strojové učení. Algoritmus předvídá případy užití v kombinaci předmět-sloveso-objekt. Na základě toho jsou tvořeny use case diagramy.

3.2.2 Prioritizace požadavků

Další aspekt inženýrství požadavků je výběr priorit mezi požadavky [14]. Jedná se o rozhodování, které požadavky budou zahrnuty v dalším vydání softwarového systému. V literatuře bývá tato problematika označována jako Next Release Problem (NRP). Musí být optimalizován vliv cílů – úsilí vývoje a uspokojení zákazníka.

Chaves-Gonzáles a kol. [20] využívají k řešení tohoto problému algoritmus optimalizace založený na učení – MO-TLBO (Multiobjective Version of Teaching-Learning Based Optimization). Tento algoritmus využívá technik umělé inteligence založených na inteligenci hejna. Výsledky experimentů provedených na osmi různých projektech potvrzují účinnost tohoto algoritmu ve srovnání s jinými.

Problémy softwarového inženýrství lze přeformulovat na optimalizační problémy [8]. Strukturování problémů však zatím musí provádět lidé a stroje jim dokáží pomoci s jejich řešením. Stroje zatím nejsou schopny samy rozložit složité problémy reálného světa na jednotlivé prvky. K rozvoji analytických schopností strojů je stále nutný další pokrok v oblasti analýzy softwarových problémů.

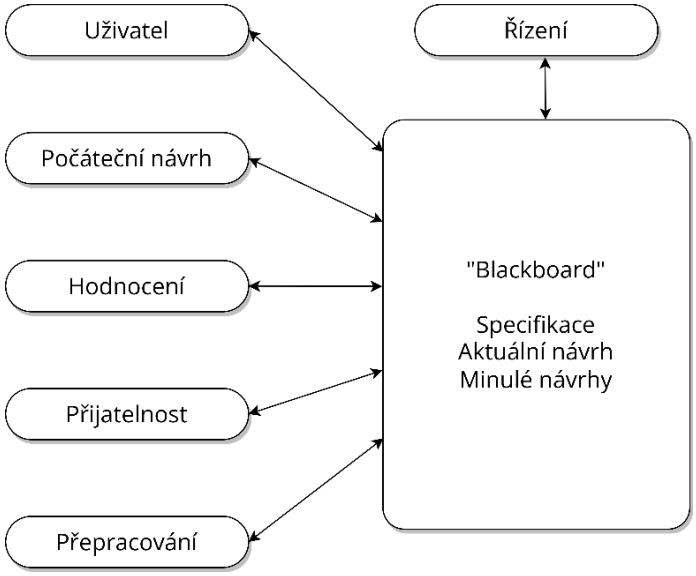
3.3 Návrh

Ve fázi návrhu softwaru dochází k rozdělení projektu na jednotlivé kroky [8]. Je vytvářena inženýrská reprezentace vyvíjeného softwarového produktu [14]. Dokument požadavků na software je převeden do návrhových modelů. Jsou definovány datové struktury, systémová architektura, jednotlivá rozhraní (interfaces) a další komponenty. Zatímco požadavky říkají, co má produkt dělat, návrh se zabývá tím, jak toho dosáhnout. V průběhu klasického návrhu [21] je určena vnitřní struktura produktu. Návrháři rozloží produkt do jednotlivých modulů. Moduly tvoří nezávislé části kódu. Tyto části by měly mít pečlivě navrženy rozhraní, aby byla zajištěna kompatibilita se zbytkem systému. Jakmile je dokončen architektonický návrh, přistupuje se k detailnímu návrhu jednotlivých modulů.

Pro efektivní tvorbu návrhu můžeme do tohoto procesu zapojit umělou inteligenci. Oproti analýze je menší množství metod [14], které ji při tvorbě návrhu využívají.

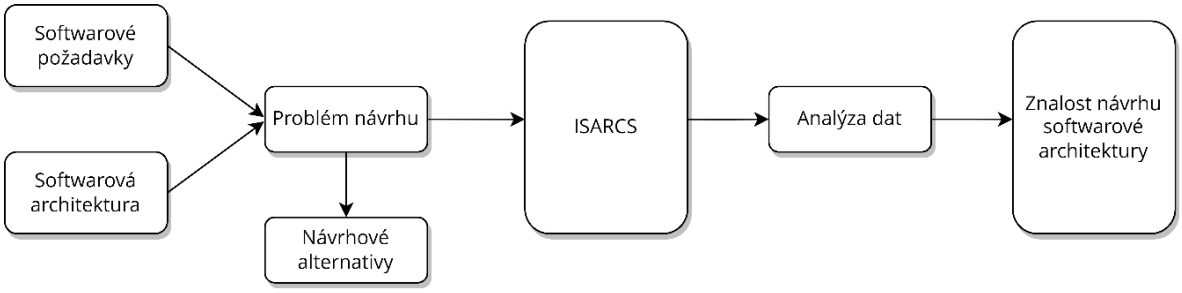
3.3.1 Volba vhodného návrhu

Jedním z možných využití umělé inteligence jsou systémy založené na znalostech nebo také jinak zvané jako expertní systémy – Knowledge-Based Systems (KBS). Dixon a kol. [22] ve své práci poukazují na řadu výhod a nevýhod, které vycházejí z různých návrhů. Z možných návrhů je potřeba zvolit nejlepší řešení, které bude akceptovatelné. Návrh vyžaduje velké zkušenosti a je zatížen subjektivitou. Navíc tvorba návrhu probíhá pod časovým tlakem. Rozlišuje mezi dvěma rozdílnými případy návrhu – tvorbou počátečního návrhu a přepracováním stávajícího návrhu (redesign). Navrhovaná architektura obsahuje čtyři počáteční funkce: počáteční návrh, hodnocení, přijatelnost a přepracování. Každá z těchto funkcí by měla čerpat ze samostatné vědomostní báze. Navíc do systému vstupují funkce uživatele (návrhář) a funkce řízení. Řízení probíhá iterativně a nezávisle na jiných funkcích na základě vlastního expertního systému.



Obrázek 4. Schéma Redesign architektury [22]

Proces návrhu softwarové architektury je složitý proces, který musí splňovat všechny stanovené požadavky. Je třeba hledat shody mezi různými pohledy na návrh, aby došlo ke zvolení vhodného návrhu. Odůvodnění pro zvolení architektury nejsou vždy přesně zachycena, což se později projevuje ve fázi údržby systému. Chanda a Liu [23] se ve své práci zabývají zdůvodněním volby architektury, aby pomohli s rozhodováním zúčastněným stranám. Zjišťují obecné názory skupin na různá hlediska a věnují pozornost převažujícím názorům. Dále navrhují metodu k vytvoření matice sledovatelnosti, která spojuje požadavky s prvky softwarové architektury. Sledovatelnost požadavků pomáhá udržovat softwarové systémy a řešit řízení změn. Také analyzují názory zúčastněných osob, aby určily prioritní témata diskuze. Autoři navrhli systém ISARCS, který zachycuje strukturované zdůvodnění návrhu a zachovává vazby na softwarové požadavky a prvky architektury. Při vyhodnocování argumentů v analýze názorů se využívá fuzzy logiky a klastrování. Zapojení systému ISARCS (Intelligent Software Architecture Rationale Capture System) do průběhu volby návrhu je znázorněno na obrázku (obrázek 5).



Obrázek 5. Zapojení systému ISARCS do průběhu návrhu [23]

Entitně-relační (ER) datový model zachycuje informace o entitách, atributech a jejich vztazích. ER model je konceptuální model pro usnadnění návrhu databáze. Ghosh a kol. [24] navrhli nástroj AGER (Automated ER Diagram Generation), který od uživatele přebírá textový popis v přirozeném jazyce a následně na jeho základě generuje ER diagram. Jednotlivé věty jsou rozděleny na slova. U každého slova se rozhoduje, o jaký slovní druh se jedná. Detekce entit, atributů a vztahů je prováděna s využitím pomocné databáze a techniky strojového učení – metody podpůrných vektorů[5]. Návrhář má možnost upravovat model, aby upřesnil nejasnosti. Tím je snížena obvyklá časová náročnost pro tvorbu celého diagramu manuálně.

3.3.2 Posuzování kvality návrhu

Atributy kvality, mezi které můžeme zařadit bezpečnost, výkonnost, kompatibilitu, udržovatelnost apod., ovlivňují rozhodnutí o návrhu architektury. Architektura se vyvíjí s přibývajícími funkcemi. V případě agilních metod vývoje jsou často požadavky specifikovány pomocí tzv. user stories. Gilson a kol. [25] se zaměřili na způsob jak automaticky identifikovat atributy kvality z user stories. Identifikace atributů kvality by mohla pomoci lépe porozumět v rozhodování o rané architektuře, a vyvarovat se tak problémům s lidskou zaujatostí. Autoři využili knihovnu spaCy založenou na konvolučních neuronových sítích k prozkoumání téměř 1 700 user stories z reálně realizovaných projektů. V těchto projektech se však objevilo také množství nejednoznačností, což negativně ovlivnilo kvalitu trénování modelu strojového učení. Podařilo se jim úspěšné nalezení user stories souvisejících s výkonem, kompatibilitou a zabezpečením. Menší úspěšnost je pro atributy související se spolehlivostí, údržbou nebo přenositelností. Do budoucna chtějí rozšířit trénovací data, která jsou nyní hlavním limitem pro dosažení lepších výsledků.

Vývoj a také údržbu negativně ovlivňují tzv. Architectural Smells (architektonické pachy). Jedná se o rozhodnutí, která negativně ovlivňují kvalitu softwaru. Nelze však přesně definovat části kódu, které již spadají mezi architektonické pachy a které ne. Může to záviset na řadě okolností, jako je určení softwaru, rozsáhlost apod. Takové označení je ovlivněno subjektivním pohledem vývojáře. V tomto ohledu má strojové učení výhodu, že dokáže odhlédnout od zaujatosti. Cunha a kol. [26] přišli s nástrojem InSet, který dokáže odhalit Unstable

[5] v anglické literatuře Support Vector Machines

Dependency a God Component. God Component je druh komponenty, který přebírá velké množství kontroly, může např. zahrnovat více tříd, metod, rozhraní apod., což přináší nesrozumitelnost a složitou údržbu kódu. Unstable Dependency nastává ve vztahu komponenty s dalšími komponentami, které jsou méně stabilní než ona sama. Stabilní komponenta zůstává v provozu a nešíří vedlejší efekty do dalších komponent, které jsou na ní závislé, přestože přijme několik změn. Komponenty ovlivněné Unstable Dependency mohou způsobit vlnový efekt změn v systému.

InSet analyzuje kód pomocí nástrojů třetích stran. Po identifikaci komponent a závislostí analyzovaného systému představí uživateli detekované pachy. Uživatel má možnost označit, zda souhlasí s nalezenými návrhy, či nikoliv. Tím může poskytnout systému zpětnou vazbu, která slouží ke zlepšování nástroje. Návrh vykazuje vysokou úspěšnost okolo 88 % pro Unstable Dependency a God Component. Do budoucna mají autoři v plánu systém rozšířit, aby dokázal identifikovat více typů pachů.

3.3.3 Software 2.0

V klasickém návrhu softwaru je projekt strukturován na definované úkoly. Karpathy [27] používá pojem software 2.0 jako synonymum pro software, který využívá umělou inteligenci. Kód by měl vznikat zpracováním neuronové sítě. Jedná se o abstraktní jazyk, který není pro člověka snadno srozumitelný. Na druhou stranu by ale nemusel být kontrolován lidmi. Neuronová síť by měla vytvářet programové kódy na základě souboru vstupů. Pro množství problémů reálného světa je jednodušší shromáždit data a identifikovat jejich chování než psát přímo program.

Zatímco klasicky je kód psaný člověkem zkompilován do binárního souboru, tak v softwaru 2.0 obsahuje zdrojový kód datovou sadu, která definuje chování a architekturu neuronové sítě, která určuje hrubou kostru kódu. Programovací týmy jsou rozděleny na dvě části. Programátoři 2.0, kteří upravují a rozšiřují datové sady, a několik programátorů 1.0, kteří udržují a rozvíjí tréninkový kód. Trénováním neuronové sítě je zkompilována datová sada do binární podoby.

Neuronové sítě se neustále vylepšují [8]. Zatím jsou však ve fázi návrhu softwaru závislé na činnostech stanovených člověkem – pracovní strategii a vlastní návrh softwaru musí realizovat lidský inženýr. Využití umělé inteligence vyžaduje jasně zadané úkoly a podporu člověka. Kroky, které jsou prováděny automatizovaně, musí být definovány a integrovány do

balíku automatizovaného vývojového prostředí, které pak může tyto funkce provádět samostatně.

Podle Lake a kol. [28] by v budoucnu mohly systémy za pomoci umělé inteligence samostatně rozpoznávat jevy reálného světa, a nejen předem naprogramované vzorce. Tudíž by se samy dokázaly učit a přizpůsobovat, nejen optimalizovat známé dovednosti.

3.4 Implementace

Ve fázi implementace dochází k tvorbě spustitelné verze softwaru [3]. Implementace může probíhat ve vysokoúrovňových či nízkoúrovňových programovacích jazycích. Případem implementace je i adaptace standardně dostupných systémů pro konkrétní účely zákazníka. Jednotlivé typické přístupy k implementaci jsou závislé na konkrétní volbě programovacího jazyka. Mezi klíčové aspekty, které jsou obecně platné pro různé případy programovacích jazyků, patří mimo jiné možnost opakovaného použití.

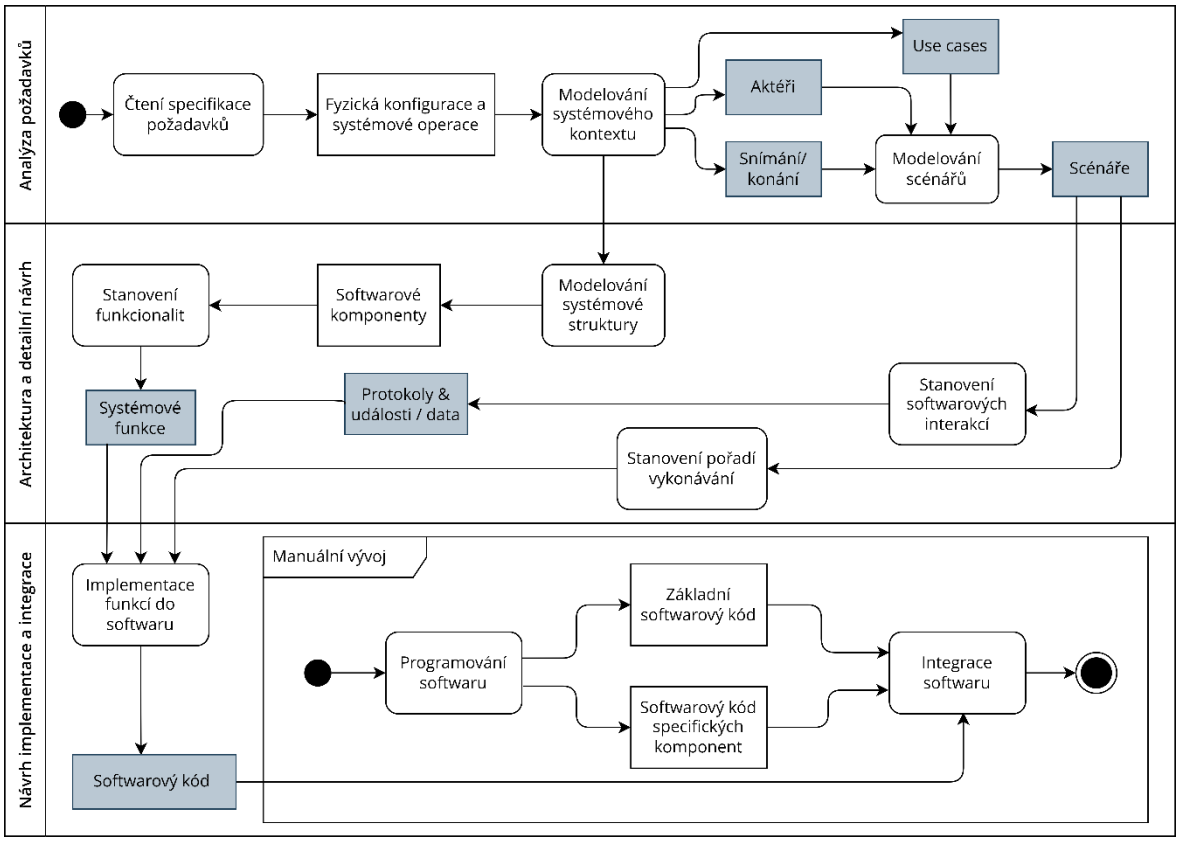
Opakované použití – Tento přístup je dnes rozšířen, jelikož umožňuje ušetřit čas a náklady na vývoj. Další výhodou je i pravděpodobnost vyšší spolehlivosti oproti nově vyvíjenému softwaru, jelikož byl systém již testován. Dříve, ve druhé polovině 20. století, se jednalo především o využívání funkcí a objektů z knihoven programovacích jazyků. Dnes se s opakovaným použitím setkáváme na různých úrovních – počínaje abstrakcí (návrhové a architektonické vzory) nebo již zmíněnými knihovními funkcemi. Další je úroveň komponent. Jedná se o kolekce objektů a tříd, které společně poskytují funkce (např. obsluhu událostí). Opakovaně můžeme používat ale i celé systémy. Opakované využití již existujících systémů může zahrnovat úpravu kódu softwarového produktu nebo integraci několika různých systémů do jednoho.

3.4.1 Generování kódu

Fáze implementace může probíhat dlouho a v jejím průběhu dochází ke změnám požadavků. Navrhované řešení je automatizované programovací prostředí, které by sloužilo ke generování kódu, jeho opětovnému využívání a úpravám. Umělá inteligence by mohla pomoci vývojářům v generování funkcí nebo datových struktur.

Insaurralde [29] navrhl Autonomous Development Process. Obvyklá automatizace vývoje softwaru představuje syntézu návrhových modelů a předdefinovaných pravidel. Autor představil samostatně řízený proces, který sám rozhoduje o vývoji softwaru. Autonomus Software Code Generation (ASCG) je agentově orientovaný přístup pro automatické generování

kódu. V tomto případě se software development agent (SDA) stává „lidským“ vývojářem, který provádí vývojové činnosti samostatně. Agent se zabývá čtením specifikace požadavků. Jsou specifikovány cíle a úkoly. Je schopen tyto informace zachytit a dotazovat se na své interní vědomosti, na základě kterých si zdůvodní rozhodnutí. Tímto způsobem se rozhoduje o návrhu systému, který realizuje systémovou logiku. Logika je tvořena z propojených bloků, jež si spolu mohou vzájemně předávat informace. Informace je reprezentována jako text, ale může mít i jinou reprezentaci. Informacemi mohou být mezivýsledky nebo grafická data. Na základě těchto informací je agent schopen vygenerovat softwarový kód, který mu byl specifikován.



Obrázek 6. Autonomní softwarové generování kódu [29]

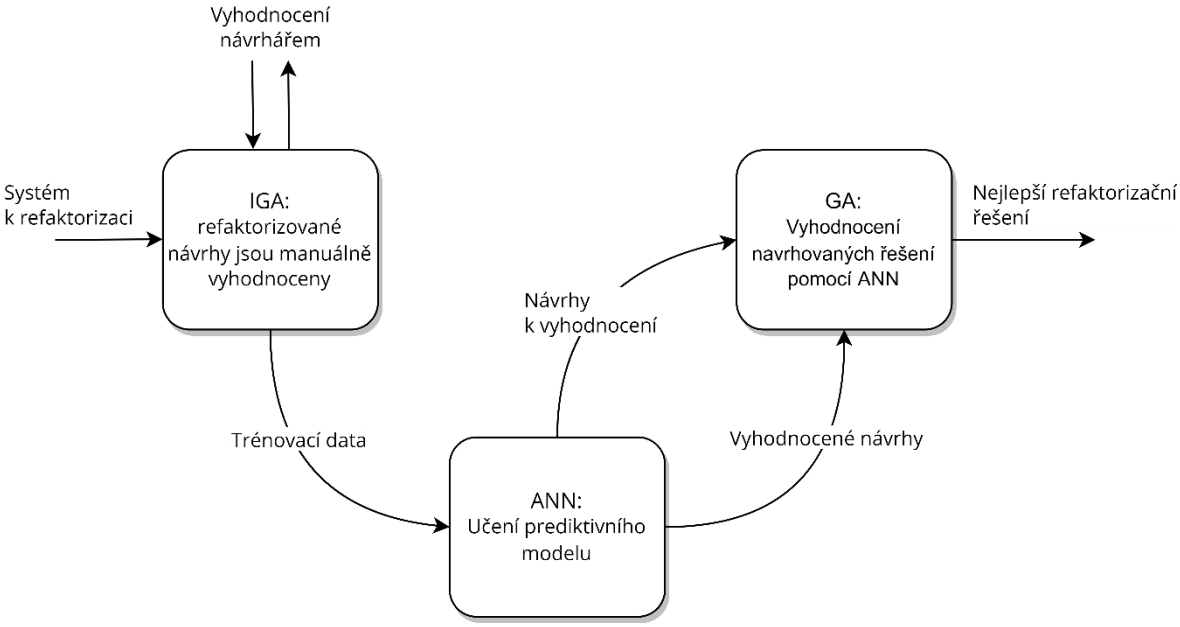
Oproti dřívějším poloautonomním systémům se jedná o čistě automatizovanou metodu. Schéma na obrázku 6 znázorňuje postup.

Husain a kol. [30] se ve své práci zabývají myšlenkou nalezení relevantního kódu jako odpověď na dotaz podaný v přirozeném jazyce. Přirozený jazyk je na rozdíl od programovacího jazyka vhodný k popisu nepřesných myšlenek. Vytvořili proto CodeSearchNet Corpus, který je vytvořen z 99 dotazů v přirozeném jazyce a k nim náleží asi 4 000 pravděpodobných anotací. Data byla získána z open-source repozitářů. Anotace

přirozeného jazyka vznikly spárováním jednotlivých funkcí s jejich dokumentací. Projekt zahrnuje anotace pro 6 programovacích jazyků, mezi které patří Go, Java, JavaScript, PHP, Python a Ruby. Návrh využívá strojové učení a techniky zpracování přirozeného jazyka. Pro vyhledávání kódu využívají techniky Neural Search System a vyhledávací engine ElasticSearch. Autoři porovnávají výkonnost a možnosti jednotlivých návrhů vyhledávacích technik.

3.4.2 Refaktorizace systému

Jednou z nejnákladnějších součástí procesu implementace je refaktoring. Amal a kol. [31] přestavili využití neuronových sítí a genetických algoritmů pro refaktoring softwaru. Schéma na obrázku 7 zobrazuje navrhovaný algoritmus.



Obrázek 7. Refaktorizace systému podle Amal a kol. [31]

Vstupem je systém určený k refaktoringu a výstupem jsou návrhy refaktoringu, které by mohly přispět ke zlepšení systému. Je využito interaktivního genetického algoritmu (IGA), se kterým interaguje návrhář, který hodnotí navrhovaná řešení. Vyhodnocuje proveditelnost a účinnost každého navrhovaného řešení. Na základě svých preferencí ohodnotí návrhy jako vhodné, nebo nevhodné. Tento přístup nevyžaduje definici fitness funkce. Všechna tato ohodnocená řešení jsou trénovací sadou pro učící genetický algoritmus (LGA). LGA za využití neuronové sítě generuje prediktivní model, který se přibližuje hledanému řešení. Ve srovnání s jinými refaktorizačními přístupy bylo toto řešení využívající neuronové sítě

vyhodnoceno jako nejlepší, a to z důvodu menšího úsilí a počtu interakcí návrhářů se systémem pro dosažení výsledků.

Činnosti prováděné ve fázi implementace jsou doposud převážnou částí tvůrčí oblastí závislou na lidské kreativitě. V budoucnu by umělá inteligence mohla nalézt uplatnění v systémech, které by dokázaly vytvořit celistvé kódy bez nutnosti lidského zásahu [8]. To však s sebou přináší riziko, že by automatická implementace mohla způsobit nesrozumitelnost kódu pro člověka.

3.5 Testování a integrace

Testování softwaru je proces spouštění programu se záměrem nalézt softwarové vady nebo chyby [14]. Někdy bývá softwarové testování označeno jako proces validace a verifikace [3]. Validace se zabývá obecnou otázkou, zda vyvíjíme „správný systém“, který odpovídá požadavkům. Tzn., že přesahuje pouhou kontrolu dodržení specifikace, ale snaží se i demonstrovat, že software dělá to, co od něj zákazník očekává. Naproti tomu verifikace se zabývá tím, jestli je software „vyvíjen správně“, tzn., zda odpovídá požadavkům.

Cílem testování je dokázat, že software splňuje specifikované požadavky, a zároveň odhalit situace, kdy se software nechová správně. Účelem testování může být i zajištění kvality a odhad spolehlivosti. Při testování se také simulují interakce mezi softwarem a jeho prostředím. Testovací scénáře by měly pokrývat kompletně zdrojový kód.

Firmy zabývající se vývojem softwaru vynakládají 30 % – 40 % úsilí právě na testování [30], přičemž náklady na testování se pohybují okolo 50 % celkových výdajů na vývoj. Testování se prolíná s celým životním cyklem vývoje softwaru. V průběhu implementace se setkáváme s unit testováním základních částí, např. metod nebo tříd. Následně se provádí integrační testování. Testují se celé komponenty, aby byla zajištěna integrita a ověřilo se, zda vše odpovídá specifikaci. Vzhledem k používání kódů třetích stran nebo v případě více vývojových týmů pracujících na stejném kódu současně se provádí další úroveň testování, tzv. systémové testování. To ověřuje, zda funguje systém tvořený jednotlivými komponentami jako celek správně.

Vzhledem ke změnám požadavků nebo kvůli údržbě softwaru se provádí změny v kódu. To vede k možnému zanesení chyb do systému, a proto se provádí tzv. regresní testování, které ověřuje správnou funkčnost po provedení jakýchkoliv změn v kódu. Před vydáním softwaru

se ověřuje, zda byly implementovány všechny uživatelské požadavky při akceptačním testování.

Přístupy k testování jsou závislé také na způsobu vývoje softwaru. Existuje celá řada dalších přístupů k testování [3]. Obecně však lze říct, že testování může ukázat pouze přítomnost chyby, nikoliv její nepřítomnost. Nikdy tedy nemůžeme spoléhat na to, že je systém dostatečně otestovaný a že neobsahuje žádné chyby. Průběh testování lze charakterizovat životním cyklem testování softwaru[6]. Významný přínos technik umělé inteligence můžeme nalézt hned v několika fázích životního cyklu. Khaliq a kol. [32] uvedli stěžejní oblasti, ve kterých má umělá inteligence přínos.

3.5.1 Specifikace testovacích případů

Na začátku životního cyklu testování softwaru jsou určeny testovací případy[7] [32]. Ty jsou vytvářeny na základě požadavků a funkcí softwaru, aby bylo dosaženo otestování všech požadavků. Testovací případ je charakterizován vstupními daty a očekávaným výsledkem. Obsahuje mimo jiné také postup krok za krokem, jak mají být jednotlivé kroky provedeny, a kritéria pro splnění, či nesplnění testu. Soubor testovacích případů bývá označován jako testovací sada[8].

Briand a kol. [33] přišli s návrhem jak vylepšit testovací sady. Ve své metodologii navrhují využití strategie Category-Partition (CP), tedy rozdělení podle kategorií. Jedná se o přístup, který se využívá pro black-box testovaní (testování černé skříňky). Testovací případy jsou vytvořeny na základě specifikace a pomocí CP je redukováno potenciální množství testovacích případů na počet, který lze reálně implementovat. Na základě CP jsou testovací případy automatizovaně převedeny do abstraktních testovacích případů, které mají vstupní vlastnosti (kategorii a volbu) a ekvivalentní výstupy.

Návrh využívá strojové učení, které slouží ke generování klasifikačních pravidel. Konkrétně jde o využití rozhodovacího stromu C4.5, kde cestu od kořenového uzlu stromu k libovolnému listu považujeme za pravidlo. V tomto případě je za pravidlo považována kombinace vstupních vlastností propojených s výstupní třídou ekvivalence. Tato pravidla jsou

[6] v anglické literatuře Software Testing Life Cycle (STLC)
[7] v anglické literatuře Test Cases (TC)
[8] v anglické literatuře Test Suite (TS)

analyzována a jsou určena možná vylepšení testovací sady. Zde záleží na zkušenostech testera. Mezi problémy, které je možné odhalit, patří nadbytečné testovací případy či naopak jejich nedostatek, ale také úpravy testovací specifikace, např. přidání kategorií nebo volby pro Category-Partition.

3.5.2 Upřesnění testovacího případu

Anglicky je tato činnost označována jako Test Case Refinement. Jedná se o aktivitu, kdy testeři vybírají nejúčinnější testovací případy a upravují omezení testu, aby se zúžil prostor pro řešení [32]. Tím se snaží docílit snížení nákladů na testování a zvýšit efektivitu celého procesu. Mezi techniky umělé inteligence, které našly uplatnění v této oblasti, můžeme zařadit informační fuzzy sítě[9].

Last a kol. [34] představili přístup, který automatizovaně redukuje kombinace black-box testů. Tento princip je založený na identifikaci vstupně-výstupních vztahů dat programu. Vstupem do IFN algoritmu je náhodně vygenerovaná trénovací sada testovacích případů. Generování náhodných testů probíhá na základě systémových vstupů a výstupů. Nejsou potřeba žádné informace o funkčních požadavcích, protože algoritmus IFN automaticky odhaluje vstupně-výstupní vztahy z náhodně generovaných tréninkových případů. Opakovaným spouštěním nalezne algoritmus podmnožinu vstupních proměnných relevantních pro každý výstup a k nim sadu potenciálních testovacích případů. Skutečné testovací případy jsou generovány z automaticky zjištěných tříd ekvivalence pomocí existujících zásad. To může být například požadavek, že se použije pouze jeden test pro jednu třídu ekvivalence apod.

3.5.3 Generování testovacích případů

Kompletní ruční tvorba testovacích sad je omezena rozsáhlostí testovacího systému. Pro většinu složitých systémů se používají techniky automatického generování testovacích případů [32]. Za posledních dvacet let roste zájem o automatizaci generování testovacích případů. I tato oblast byla ovlivněna možnostmi, které přicházejí s využitím umělé inteligence.

V této oblasti bylo představeno mnoho přístupů. Můžeme zmínit např. využití genetického algoritmu v práci Khan a Amjad [35]. Navrhují novou techniku, která je založena na využití mutačního testování a genetického algoritmu. Mutační testování spočívá v záměrném

[9] v anglické literatuře Information Fuzzy Networks (IFN)

vložení chyby do programu. Tyto chyby jsou označovány jako mutace. Poté jsou vygenerovány testovací případy za využití genetického algoritmu a křížení. Vznikají tak nové testovací případy.

Rosenfeld [36] a kol. se zaměřili na testování aplikací pro Android. Funkční testování je úkol, který lze zatím provádět pouze vytvářením jednotlivých testů pro každou novou aplikaci, nebo případně ručním testováním. Vyžaduje totiž vyšší úroveň uvažování než vývoj nástrojů pro automatizovanou dynamickou analýzu. Ve svém návrhu nástroje ACAT (Activities Classification for Application Testing) se zaměřují na strukturu grafického uživatelského rozhraní (GUI). Aplikují předdefinovanou sadu obecných testovacích případů pro konkrétní skupinu založenou na podobnostech GUI. Použití strojového učení umožňuje klasifikovat každou z aplikačních činností do konkrétního typu, což dovoluje testovat různé očekávané funkční chování jednotlivých obrazovek.

Ansari a kol. [37] navrhují systém, který se zabývá automatickým generováním testovacích případů z funkčních požadavků pomocí zpracování přirozeného jazyka[10] (NLP). Navrhovaný systém dokáže automaticky analyzovat funkční požadavek ze specifikace požadavků a na základě požadavku generuje testovací případ. Pro zpracování pomocí NLP je nezbytné, aby byly požadavky definovány v konjunktivní normální formě. Systém by bylo možné rozšířit tak, aby bylo možné generovat testovací případy z požadavků v libovolném formátu. V současnosti systém také nedokáže určovat prioritu testovacího případu.

Testovací případy je možné generovat na základě UML diagramů. S přístupem využívajícím UML a optimalizaci mravenčích kolonií přišli Li a Lam [38]. Automatické generování testovacích kroků probíhá na základě UML stavových diagramů. Stavový diagram je automaticky převeden na orientovaný graf. K nalezení optimální cesty grafem je využita technika optimalizace mravenčích kolonií.

[10] v anglické literatuře Natural Language Processing

3.5.4 Generování testovacích dat

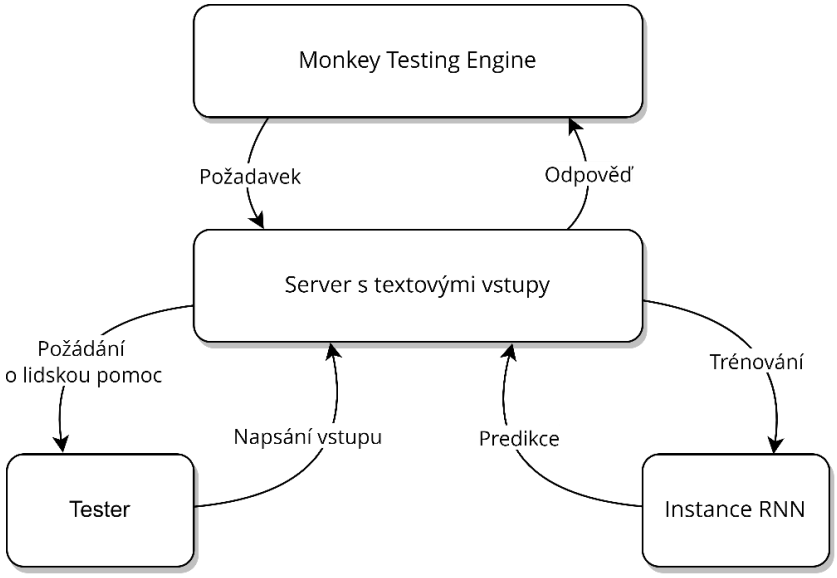
Generování testovacích dat je činnost, při které dochází k vytvoření testovacích vstupů pro testovaný systém. Testovací data jsou vytvářena na základě testovacích případů a scénářů. Od kvality testovacích dat se odvíjí rozsah pokrytí testů pro testovaný software.

Paduraru a kol. [39] představili open-source nástroj, který umožňuje generovat testovací data. Jako vstup mu slouží soubor vzorových testů. Nástroj využívá clusterovou (shlukovou) analýzu pro seskupení testů do clusterů. Následně se učí generativní model pro každý cluster. Díky tomu lze poté generovat nové testy s vysokou mírou správnosti pro každý cluster. Modely mají strukturu rekurentní neuronové sítě. Nástroj nevyžaduje žádné ruční úsilí uživatelů, kromě přizpůsobení parametrů jednotlivých clusterů.

Na generování textů pro testování mobilních aplikací se ve své práci zaměřuje Liu a kol. [40]. Jejich výzvou bylo vytvořit automatické generování co nejrelevantnějšího textu v kontextu případu užití. Tyto texty jsou využívány jako vstupy pro Monkey Testing. Tento typ testování je založen na náhodné sekvenci akcí interagujících s aplikací. Pro relevantní výsledky je však nezbytné zadat také relevantní textové vstupy, což s využitím navrhovaného nástroje eliminuje tento problém, a tudíž i nutnost vynakládat lidské úsilí pro jejich tvorbu. Autoři rozdělují řešení na dvě části – trénink z kontextu a předpověď vstupního slova. Využívají rekurentní neuronové sítě[11] (RNN).

Když testovací engine objeví textbox, požádá o relevantní vstup ze serveru. Server vyřeší požadavek tím, že jej odešle lidskému testerovi nebo RNN. V manuálním režimu může tester zadat relevantní vstup nebo jej ignorovat. Tyto vstupy jsou pak zaznamenány do databáze. V režimu AI může natrénovaná RNN automaticky předvídat textový vstup. Protože se však aplikace, pro které předpovídáme vstupy, liší od aplikací, na kterých model trénujeme, nemusí model RNN správně rozpoznat kontext. K vyřešení tohoto problému byla přidána technika Word2Vec z oblasti zpracování přirozeného jazyka, která dokáže rozpoznat synonyma a odlišnosti v kontextu. Názorně je princip fungování znázorněn na obrázku (obrázek 8).

[11] v anglické literatuře Recurrent Neural Network

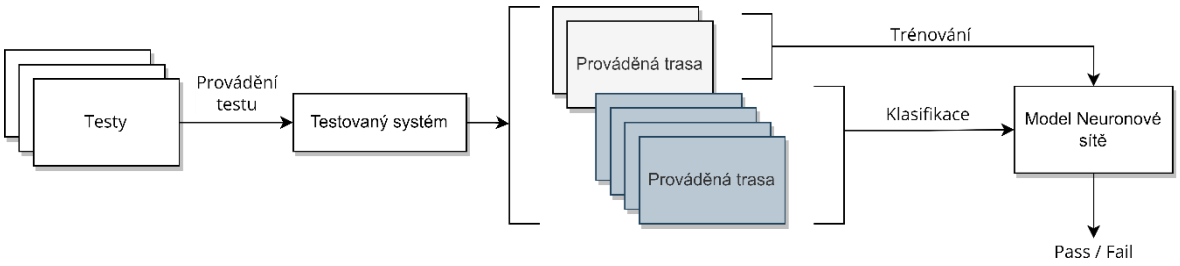


Obrázek 8. Princip fungování Monkey Testing Engine [40]

3.5.5 Testovací oracle problém

Při testování se snažíme ověřit správné chování systému podle požadavků. Je potřeba však umět rozlišit mezi správným a nesprávným chováním systému s konkrétně zadaným vstupem. Zodpovězení této otázky bývá označováno jako oracle problém a je klíčovým problémem pro automatizované testování.

Mezi nejaktuálnější řešení zabývající se tímto problémem patří návrh od Tsimpourlas a kol. [41]. Implementují framework pro automatizaci jednotlivých prováděných kroků. Hlavní myšlenka fungování je zobrazena na obrázku.



Obrázek 9. Princip fungování automatického řešení oracle problému [41]

Autoři využívají učení neuronové sítě s učitelem. Určitá část vzorků je ručně označena a využita pro trénování neuronové sítě. Nejedná se tedy o plně automatizovaný proces. Bez klasifikátoru by však vývojář musel specifikovat očekávaný výstup pro všechny testy, což je časově výrazně náročnější než malý podíl testů, které je potřeba ručně označit.

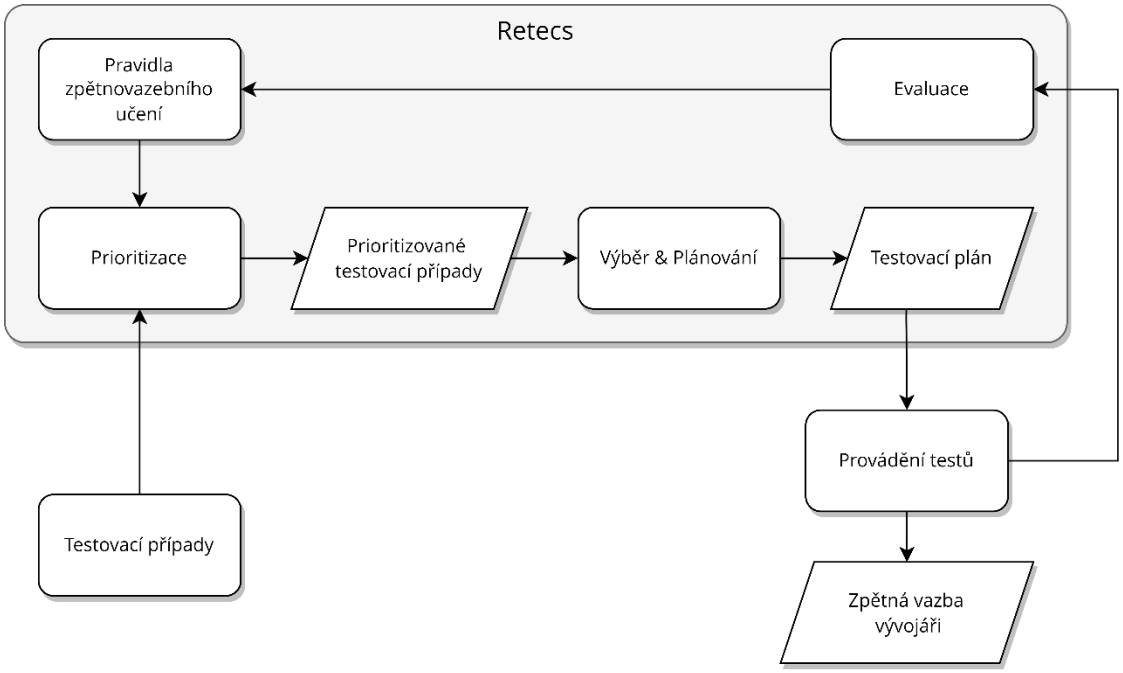
Pomocí trénovacích dat se model naučí rozlišovat běhové vzory pro úspěšné (pass) a neúspěšné (fail) provedení testu. Úspěšnost navrhovaného řešení se na základě vyhodnocení autorů pohybuje kolem 90 %.

3.5.6 Stanovení priority testovacích případů

Stanovení pořadí provádění testovacích případů bývá určeno na základě jejich priority [32]. Testovací případy, které s největší pravděpodobností odhalí chyby, by měly být prováděny v procesu testování dříve. Upřednostnění může být také na základě rizika, tj. závažnosti dopadu rizika, pokud by k němu došlo. Mohou být ale zohledňovány další faktory ovlivňující pořadí testů podle konkrétního testovaného systému. Dokud nebyla zapojena do této problematiky umělá inteligence, neexistovaly nástroje, které by umožňovaly tuto oblast automatizovat.

Kontinuální nebo také průběžná integrace, častěji známá pod anglickým spojením Continuous Integration (CI), je jeden z přístupů používaných při vývoji softwaru, zvlášť ve chvíli, kdy jednotliví vývojáři často integrují svou práci [42]. CI zahrnuje správu verzí, správu konfigurace, automatické sestavování (build) a také automatické regresní testování. Cílem testování je odhalit chyby co nejdříve. Proto je nutná volba relevantních testovacích případů. CI vyžaduje přísnou kontrolu nad stanovením priorit testovacích případů. Testovací případy, které selhaly v minulosti, mají větší pravděpodobnost, že selžou v budoucnu. V dalších cyklech jsou tedy plánovány jako první. Roli hraje také doba k provedení jednoho cyklu. Doba pro provedení testovacího případu se může výrazně lišit. Kvůli tomu není možné provádět všechny testy a je nutný pouze určitý výběr testovacích případů.

Autoři Spieker a kol. [42] představili metodu nazvanou Retecs (Reinforced Test Case Selection), která slouží pro výběr a upřednostnění testovacích případů na základě jejich doby trvání, předchozího posledního provedení a historie selhání. Metoda využívá techniku zpětnovazebního učení. Systém je adaptivní a učí se rozpoznat indikátory neúspěšných testovacích případů během svého běhu. Sleduje testovací případy, výsledky provedených testů a jejich dopad. Není přitom vyžadován žádný přístup ke zdrojovému kódu testovaného programu. Navrhovaná metoda vyžaduje pouze konkrétní historické výsledky testů, doby jejich trvání a doby posledního provedení testu.



Obrázek 10. CI s využitím prioritního plánování Retecs [42]

Každý testovací případ je vyhodnocen jednotlivě a poté je vytvořen plán ze všech vybraných testovacích případů. Ve zpětnovazebním učení interaguje agent se svým okolím pomocí tzv. stavů a volbou vhodné akce, která na něj reaguje. Akce je zvolena buď na základě naučených pravidel, nebo náhodným zkoumáním možných akcí. Testovací plán následně agent získává zpětnou vazbou, kterou zhodnotí výkon jeho předchozí akce. Stav představuje data o jednom testovacím případu. Jako akce je vrácena priorita testovacího případu pro aktuální cyklus CI. Poté, co jsou upřednostněny všechny testovací případy v testovací sadě, je naplánována prioritní testovací sada, včetně výběru nejdůležitějších testovacích případů, a odeslána k provedení. Na základě výsledků provedených testů je určena odměna. Podle odměny agent upraví své rozhodování pro další akce. V případě pozitivních odměn je zvolené chování podporováno a v případě negativních odměn je od zvoleného chování upuštěno.

3.5.7 Odhady nákladů na testování

Testování softwaru je proces náročný na vynaložený čas i zdroje [43]. Je proto klíčové co nejdříve předvídat úsilí potřebné k testování softwaru, aby bylo možné plánovat činnosti a optimálně alokovat zdroje. Odhad testovacího úsilí je jedním z důležitých aspektů pro úspěšné řízení testování. Celkové úsilí vynaložené na testování závisí na mnoha různých okolnostech – včetně lidských faktorů, testovacích technik, použitých nástrojů apod. Při odhadování se v rané fázi vychází obvykle z prvních dostupných informací popisujících vytvářený systém, což jsou obvykle funkční požadavky, které bývají popsány pomocí případů

užití (Use Cases). Z přístupu založeném na požadavcích a případech užití vychází celá řada odhadovacích technik. Problémem, který komplikuje přesnost predikce, jsou technické faktory, které je velmi těžké jakkoliv objektivně měřit.

Jedním ze znaků používaných k identifikaci velikosti testování projektu je tzv. Test Lines Of Code (TLOC), tj. počet řádků kódu potřebných pro otestování systému. Badri a kol. [43] zkoumali možnosti odhadování velikosti TLOC podle metrik založených na případech užití. Své výsledky porovnávali s metodou Use Case Points (UCP), která také vychází z případů užití. Zabývali se několika metrikami založenými na případech užití a jejich vztahu k TLOC. Porovnávali různé techniky strojového učení s různě nastavenými kritérii. Mezi použité techniky strojového učení, které ověřovali, zařadili algoritmus k-nejbližších sousedů, lineární regrese, naivní bayesův klasifikátor učení, C4.5, náhodný les a vícevrstvý perceptron.

Ve výsledku došli k závěru, že přístup založený na metrikách případů užití poskytuje lepší výsledky z hlediska predikce velikosti testovacího kódu než přístup pomocí Use Case Points. Zároveň však uvádí, že je potřeba doplnit řadu dalších zkoumání a porovnání s jinými systémy, aby bylo možné vyvodit obecně platné závěry pro různé typy projektů.

Podle výsledků studie z roku 2022 [32], která se zabývala možnostmi využití technik umělé inteligence v testování softwaru a jejich přínosem, vyplývá, že jednotlivé fáze testování softwaru byly výrazně zlepšeny s využitím těchto technik. Největší přínos byl zaznamenán zejména při generování a návrhu testovacích případů. Výše zmíněné oblasti testování se týkají nejdůležitějších aktivit spojených s procesem testování softwaru. Byly vynechány některé další aktivity, např. automatický výběr testovacích technik nebo automatické opravování testů, pro které zatím nebyly provedeny dostatečné studie mapující možnosti a přínos umělé inteligence.

Techniky umělé inteligence byly nejčastěji využity pro řešení optimalizace napříč různými aktivitami spojenými s testováním. Nejčastěji se jedná o genetické algoritmy, umělé neuronové sítě a zpětnovazební učení. Podle Barenkamp a kol. [8] je i přes neustálé zlepšování automatizovaného testování založeného na umělé inteligenci stále nutný lidský přístup k řízení testovacího procesu. Implementační část mohou vykonávat stroje. Podle jejich studie přibližně 35 % z dotázaných 328 expertů zastává názor, že úplné nahrazení lidských programátorů stroji v procesu testování nebude možné. Umělá inteligence však urychluje proces testování a šetří pracovní sílu a celkové náklady.

3.6 Údržba

Organizace IEEE definuje údržbu softwaru jako úpravu softwarového produktu po dodání za účelem odstranění závad, zlepšení výkonu nebo jiných atributů nebo přizpůsobení produktu upravenému prostředí [14]. Nelze přesně vymezit hranice činností, které spadají do údržby softwaru. Obecně se tak označují aktivity, které jsou prováděny ve chvíli, kdy je software doručen zákazníkovi [3]. Mezi hlavní typy údržby softwaru můžeme zařadit:

Oprava vad – Do této skupiny spadá oprava chyb kódu nebo chyb návrhu. V případě opravy chyb požadavků se jedná o nákladný proces, protože může vyžadovat rozsáhlé změny v systému.

Přizpůsobení prostředí – Tento typ údržby (někdy označován jako adaptivní údržba) se provádí ve chvíli, kdy dojde ke změnám v prostředí systému. Může se jednat o hardwarové změny, ale i softwarové, například změna operačního systému apod.

Přidání nových funkcí – Obvykle se jedná o změny většího rozsahu. Přidání nových funkcí může být reakcí na změny ve firmě. Provádění změn v existujícím softwaru je obvykle náročnější než nový vývoj. Také časová náročnost a finanční náklady jsou obvykle větší v případě přidávání nových funkcí než při opravě chyb.

3.6.1 Vyhledávání informací

Údržba by měla být mimo jiné reakcí na to, co uživatelé požadují od softwaru a co je třeba změnit, aby byli zákazníci spokojeni. Jednou z podob zpětné vazby od uživatelů jsou recenze. Ty však mohou nabývat neobjektivnosti kvůli přehnaně pozitivním, či naopak přehnaně negativním sdělením. Touto problematikou se zabývají Catal a Guldan [44]. Zaměřili se na vývoj modelu využívajícího vícenásobné klasifikační systémy[12] (MCS) pro detekci falešných negativních recenzí. Kombinací vícenásobných klasifikačních systémů se strojovým učením se podařilo dosáhnout lepších výsledků než u doposud navrhovaných způsobů řešení tohoto problému. Jejich model využívá pět individuálních klasifikátorů a vyhodnocuje výsledky každého zvlášť, což je hlavním rozdílem oproti doposud známým návrhům. Model trénovali na testovacích datech vytvořených z recenzí hotelů, která byla využita již u dřívějších návrhů jiných autorů. Díky tomu mohli srovnat dosažené výsledky s jinými

[12] v anglické literatuře Multiple Classifier Systems

systémy. Přesnost navrhovaného řešení dosahuje více než 88 %. Výsledky předchozích návrhů jiných autorů dosahovaly přesnosti 86 %. V budoucí práci by bylo možné zkoumání jiných klasifikátorů s jinými parametry k dosažení ještě vyšší přesnosti.

Systémy pro správu verzí uchovávají všechny změny údržby provedené u zdrojového kódu během vývoje softwarového systému. Dokumentovaná data historie verzí však neobsahují tagy, které by identifikovaly účel příslušné provedené změny odevzdání. Práce Meqdadi a kol. [45] zkoumá historie verzí systémů s open source kódy, aby automaticky klasifikovala odevzdání verzí do jedné ze dvou kategorií – adaptivní commit[13] a neadaptivní commit. Jako příklad adaptivní údržby, který se dotýká velkého množství organizací, je například přechod na novější verzi vývojové platformy. Identifikace změn v průběhu adaptivní údržby umožňuje nahlédnout do předchozího úsilí adaptivní údržby a může zlepšit odhad úsilí při provádění budoucích úkolů. Autoři se ve své práci snažili rozlišit faktory, které se objevují zejména v commitech spojených s adaptivní údržbou. Pomocí technik strojového učení vytvořili systém, který dokáže tyto commity rozpoznat na základě změn ve zdrojovém kódu, nikoliv pouze na základě komentářů u commitů. Výzkum ukazuje přesnost ve vyhodnocování commitů od 75 % až po 84 %.

3.6.2 Detekce problémů

Softwarové návrhové vzory jsou abstraktní popisy, které ukazují vhodný postup řešení opakujících se problémů. Rozpoznání návrhového vzoru v implementaci konkrétního projektu je přínosné pro další rozvoj nebo údržbu kódu. Rozpoznat návrhový vzor může být obtížné kvůli nedostatečné nebo zcela chybějící dokumentaci. Vzhledem k volnosti využití návrhových vzorů je nemusí být jednoduché automaticky rozpoznat. K tomuto účelu může být využito strojové učení, jak ukazují Alhusain a kol. [46]. Ve své práci navrhují postup jak identifikovat návrhové vzory existujících projektů. Navrhovaný přístup nevyužívá žádná předem definovaná pravidla, ale znalosti získává z reálných implementovaných aplikací. Nejprve je identifikována sada kandidátních tříd pro každou roli v každém návrhovém vzoru. Následně jsou zkoumány možné kombinace souvisejících rolí, zda představují instanci návrhového vzoru, či nikoliv. Pro každý návrhový vzor je trénována neuronová síť s odlišnými vstupními vlastnostmi. Trénovací data jsou vytvořena na základě existujících nástrojů pro

[13] commit je operace, která odesílá nejnovější změny zdrojového kódu do repozitáře

rozpoznávání návrhových vzorů. Díky rozpoznání instancí návrhových vzorů je usnadněna údržba systému. Jsou tak vysvětleny vztahy a interakce mezi třídami a je pochopitelné zdůvodnění zvoleného návrhu.

Jedním z problémů, který se projeví zejména ve fázi údržby, jsou tzv. Code smells, což by mohlo být volně přeloženo do češtiny jako pachy v kódu. Jedná se o nevhodné programovací návyky, které nemusejí mít vliv na samotnou funkčnost aplikace, ale produkují další problémy a celkově snižují kvalitu kódu. Das a kol. [47] se zaměřili na dva případy těchto nevhodných přístupů – Brain class a Brain method. Brain class je označení pro složitou třídu, která centralizuje funkci systému v této třídě. V případě Brain method se problém týká příliš dlouhých metod, které jsou těžko pochopitelné a je prakticky nemožné je znovu použít. Vhodně napsaná metoda by měla mít přiměřenou složitost, která je v souladu s účelem metody. K hledání Brain class a Brain method v kódu využili model konvolučních neuronových sítí, který byl trénován na deseti Java projektech. Experimentální výsledky ukazují přesnost kolem 95 % pro detekci Brain method a nejméně 97 % pro Brain class.

Umělá inteligence nabízí řešení pro dílčí činnosti spojené s údržbou systému. Setkáme se s ní v oblasti zpětné vazby od uživatelů, která je klíčová k pochopení toho, jakým způsobem s konkrétní aplikací její uživatelé pracují a s jakými potížemi se setkávají. Své místo má i v oblasti detekce problémů a využívání tzv. best practices. Může pomoci odhalit nevhodné přístupy, a pomoci tak s pochopením kódu pro provedení jeho údržby. Do budoucna by mohly najít uplatnění systémy s využitím umělé inteligence ve správě rozsáhlé softwarové infrastruktury, aby se dokázala přizpůsobovat změnám prostředí. Zatím však nebyl představen systém, který by to nyní dokázal samostatně, bez lidské kontroly [8].

4 NÁVRH SOFTWAROVÉHO PROCESU

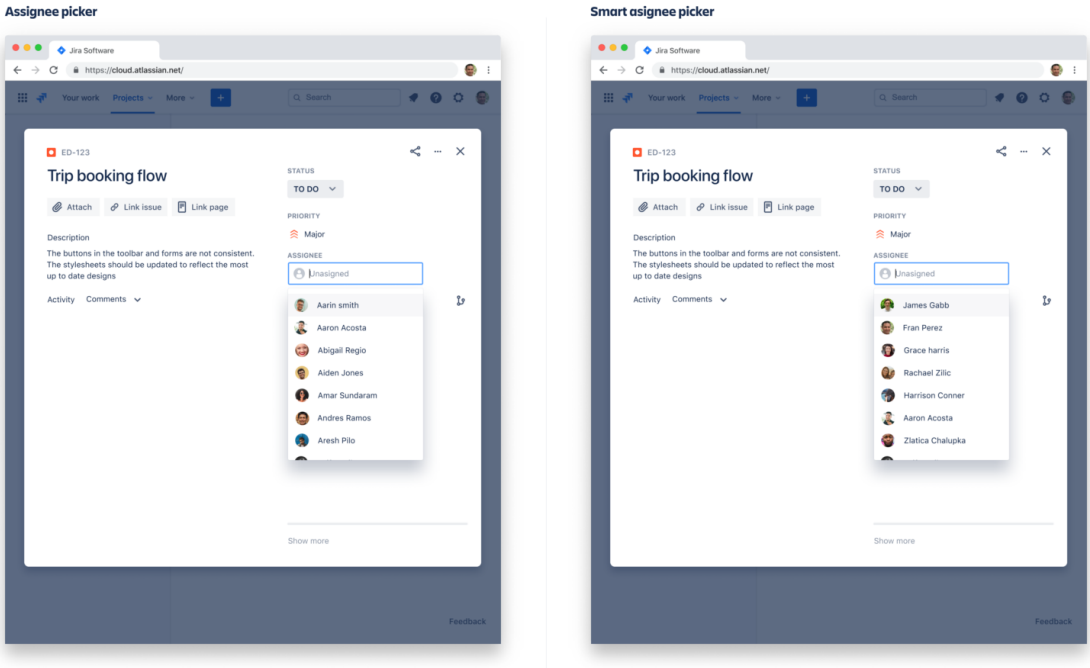
V současné chvíli je dostupný nespočet nástrojů pro vývoj a správu softwaru. Jedná se o nejrůznější nástroje, které mohou být dostupné buď samostatně, nebo mohou být seskupeny do balíků obsahujících sadu funkcionalit. V tomto případě pak hovoříme o tzv. integrovaných vývojových prostředích[14] (IDE). Ty obvykle obsahují editor zdrojového kódu, kompilátor a většinou i debugger pro ladění programu. Dále mohou obsahovat celou řadu dalších funkcionalit podle konkrétního zaměření, např. pro objektově orientovaný vývoj apod. IDE tak tvoří plnohodnotný celek použitelný pro vývoj softwaru. Editory zdrojového kódu na rozdíl od integrovaných vývojových prostředí pouze usnadňují jeho editaci, ale mohou být propojeny s dalšími rozšiřujícími nástroji [48]. Následující kapitola se zaměřuje na představení několika samostatných nástrojů, ale i rozšíření, která lze využít v průběhu vývoje softwaru. Jedná se o aktuálně dostupný software, který využívá technik umělé inteligence k podpoře jednotlivých činností vývoje.

V prvotních fázích softwarového procesu se setkáváme s plánováním. Fáze plánování zahrnuje celou řadu činností a je nedílnou součástí každého projektu. Proto doporučuji využití nástroje, který by pomohl s plánováním, a usnadnil tak tuto fázi. Na většině rozsáhlých projektů spolupracují velké týmy, které spolu musí komunikovat. Jednotliví pracovníci se mohou nacházet na různých místech, proto je nutné zajistit kvalitní komunikaci pro spolupráci všech členů vývojových týmů. Společnost Atlassian [49] se zabývá tvorbou nástrojů pro plánování a týmovou spolupráci. Jedním z nich je Jira Software [50]. Jedná se o nástroj, který pomáhá řešit projektový management. Jednotliví členové vývojového týmu mohou sledovat a plánovat svou práci. Je možné provádět prioritizaci a diskutovat nebo využívat možností vizualizace dat pro přehled prací prováděných mezi týmy.

S ohledem na pracovní týmy a postup prací lze provádět efektivně plánování a udržovat stále aktuální organizaci týmů. Společnost Atlassian využívá strojové učení pro sledování práce jednotlivých týmů a snaží se zmapovat, jakým způsobem spolu jednotliví vývojáři a vývojové týmy interagují v rámci cloudové platformy. Aplikace pak na základě získaných dat pomáhá k dosažení maximální produktivity práce. Konkrétně se jedná například o predikci při vyhledávání nebo o inteligentní filtrování dat na základě informací, které software získal

[14] v anglické literatuře Integrated Development Environment

o konkrétním vývojáři. Dále jde třeba o přehled dokumentů, který je personalizovaný a usnadňuje tím čas potřebný pro jejich vyhledávání. Kromě dokumentů je software schopen také predikovat osoby podle toho, s kým spolupracují a kdo by mohl být přínosem v konkrétních oblastech vývoje. Přesnost predikce se pohybuje v rozmezí 75–79 %.

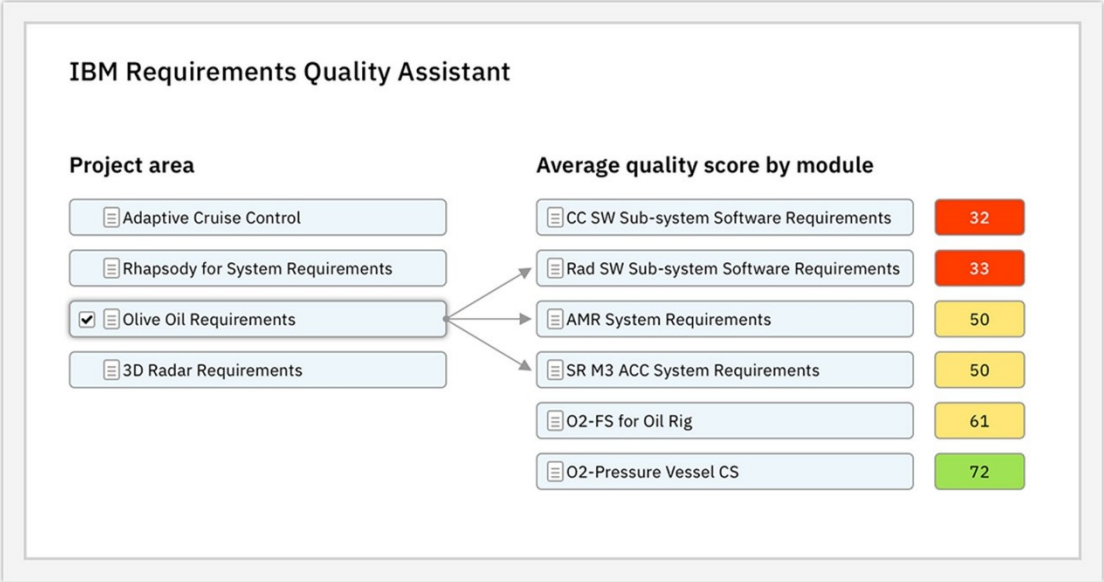


Obrázek 11. Jira Software – smart predikce [51]

Díky strojovému učení je vyhodnoceno velké množství dat. Zjištěný závěr je pak nabídnut v pravý okamžik. Díky tomu se zvyšuje efektivita celé plánovací fáze a snáze se dokáže přizpůsobovat neobvyklým situacím, kdy je potřeba provádět neočekávané změny.

Nedílnou součástí softwarového procesu je analýza. V jejím průběhu se utvářejí požadavky na software a probíhá utváření koncepce, jak by měl vypadat výsledný produkt. Pro vývojáře je v této fázi výzvou vytvořit organizovaný přehled požadavků na základě přání zákazníka. V této oblasti se angažuje společnost IBM se svým nástrojem IBM Engineering Requirements Management DOORS Next [52]. Jedná se o komplexní nástroj umožňující optimalizaci komunikace mezi všemi zúčastněnými stranami. Umožňuje zachytit a analyzovat požadavky na rozsáhlé produkty, sledování priorit v průběhu životního cyklu vývoje a uchovává historii všech provedených změn v reálném čase mezi více uživateli bez konfliktu přístupů. Umožňuje vizualizaci změn a řadu dalších funkcí, které mají za cíl zjednodušit lidskou činnost a snížit riziko chyb, případných duplicit či nepochopení požadavků. Pro optimalizaci všech činností systém nabízí doporučení, která provádí na základě technik umělé inteligence.

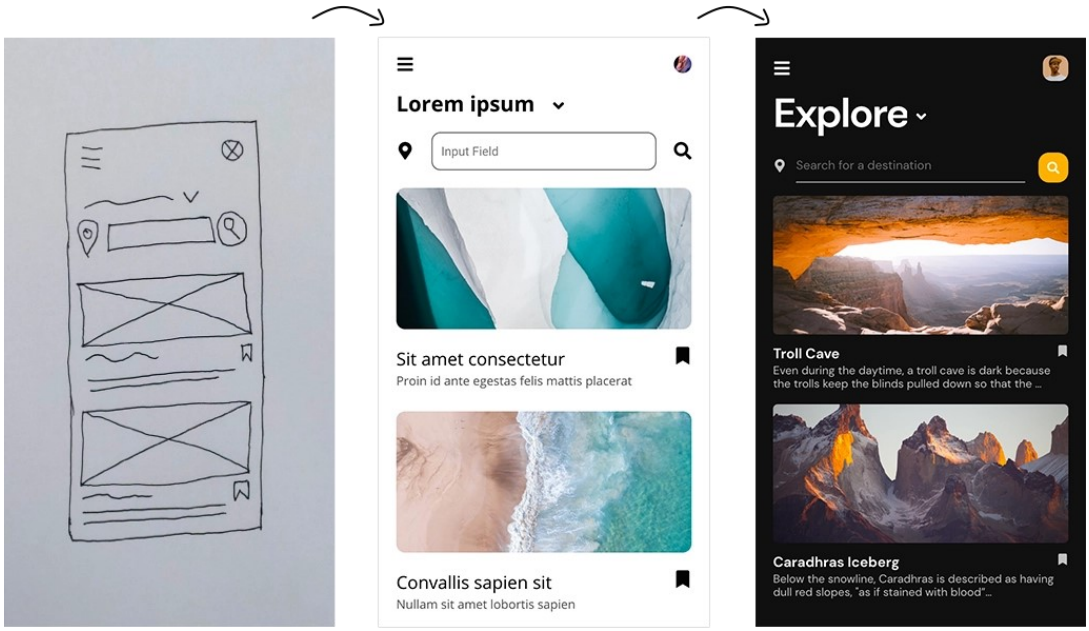
Doplněk Requirements Quality Assistant (RQA) vyhodnocuje kvalitu požadavků a pomocí strojového učení jsou nabízeny návrhy pro jejich zlepšení, aby bylo dosaženo minimálních rizik, potažmo co nejnižších nákladů na vývoj softwaru.



Obrázek 12. IBM Requirements Quality Assistant [53]

Zpracování přirozeného jazyka umožňuje analyzovat požadavky a odhalit případné problémy [53]. RQA pomáhá se zlepšením srozumitelnosti a jednoznačnosti požadavků.

Dalším z úkolů, se kterým se inženýři v průběhu vývoje setkávají, je volba a tvorba návrhu. Jedním z návrhů, se kterým se ve většině aplikací setkáváme, je ten popisující uživatelské grafické rozhraní. Grafické rozhraní musí splňovat všechny požadavky na funkce systému a zároveň musí být pro uživatele srozumitelné a přehledné, protože právě GUI je část, se kterou přicházejí uživatelé nejčastěji do kontaktu a má zásadní vliv na přijetí, či nepřijetí vytvořeného produktu. Pro tvorbu návrhu prototypu grafického rozhraní doporučuji nástroj, který získal ocenění Best AI Product of the Year [54]. Jde o nástroj Uizard, který slouží pro tvorbu designového návrhu pro mobilní, webové i desktopové aplikace. Nabízí spoustu možností jak vytvořit návrh a má implementovány některé techniky založené na umělé inteligenci. Jednou z nich je možnost počítačového vidění [55]. Návrhy, například drátěné modely, vytvořené na papíře je možné pomocí počítačového vidění převést do podoby prototypu. Při tom je možné zohlednit barvy, typografii, design jednotlivých komponent apod.



Obrázek 13. Uizard – tvorba prototypu [56]

Uizard využívá také technik zpracování přirozeného jazyka. K jednotlivým grafickým komponentám přistupuje, jako by to byla slova, k obrazovkám aplikace, jako by to byly věty atd. Na tomto principu je založeno automatické dokončování rozpracovaných návrhů. Oblastí zapojení umělé inteligence je ale více a neustále se vylepšují. Přestože nástroj výrazně usnadňuje práci na vzhledu aplikace a nabízí množství variant a snadnou modifikaci, kreativní přínos člověka zůstává nadále velmi důležitý.

V další části životního cyklu vývoje softwaru dochází k psaní zdrojového kódu, tedy k tvorbě samotného produktu. Jedná se o činnost, která je velmi závislá na lidských vývojářích, a není zatím obvyklá její úplná automatizace. Nicméně i tato oblast je značně ovlivněna novými trendy a dochází zde k usnadnění lidské práce díky umělé inteligenci. Jednou z technologií, která je aktuálně dostupná v náhledové verzi, je GitHub Copilot představený v červenci 2021. Autoři [57] nazývají tuto službu jako párového programátora, který pomáhá s psaním kódu. Má za cíl umožnit programování s menším úsilím a ušetřit čas. Tato technologie využívá systém Open AI Codex, který vytvořila společnost OpenAI a je založený na strojovém učení a zpracování přirozeného jazyka. Systém byl trénován na veřejně dostupných datech, především zdrojových kódech na GitHub.

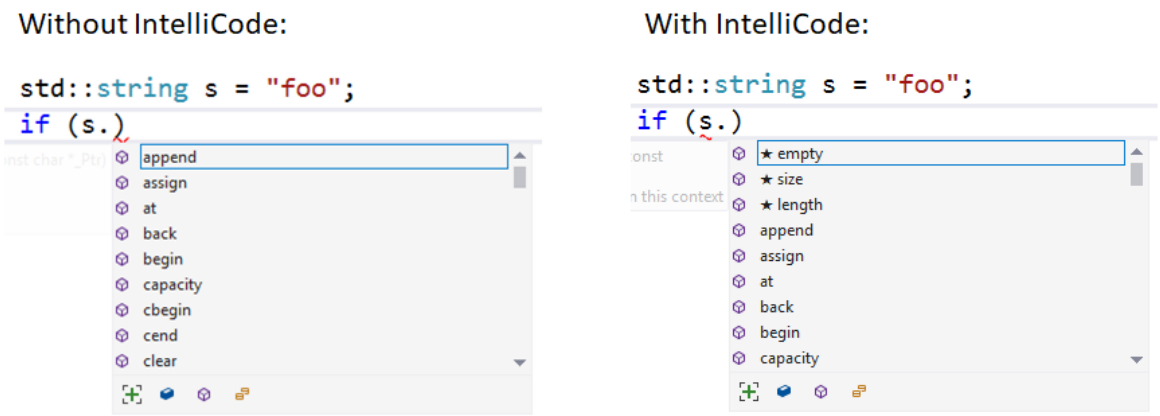
Obrázek 14. Princip fungování GitHub Copilot [57]

GitHub Copilot čerpá kontext z komentářů již napsaného kódu, ale i dalších souvisejících souborů. Na tomto základě navrhuje další řádky zdrojového kódu a celé nové funkce. Již v úvodu této kapitoly byl vysvětlen rozdíl mezi integrovaným vývojovým prostředím a editorem zdrojového kódu. V současné chvíli je služba k dispozici jako rozšíření pro některé editory, včetně Visual Studio Code, Neovim nebo nástrojů od JetBrains. Na vylepšování systému se stále pracuje. Autoři uvádí jako příklad funkčnosti nástroje sadu funkcí napsanou v Pythonu, které se vyskytují v open source repozitářích. Vymazali těla těchto funkcí a nechali GitHub Copilot doplnit chybějící kód. Ve 43 % případů se to systému podařilo na první pokus. Za výsledný kód je však plně zodpovědný vývojář, který tento systém využívá, a je nutné kontrolovat jeho výstup a provádět testování.

Obrázek 15. GitHub Copilot – automatické doplňování kódu [57]

Nástroje pro implementaci úzce souvisí s nástroji pro údržbu kódu a v mnoha případech jsou tyto nástroje totožné. Údržba může sestávat z provádění úprav kódu, a proto se mnohdy opakuje softwarový proces opět od plánování až po testování.

Dalším nástrojem, který může být přínosný pro usnadnění procesu implementace, potažmo pro údržbu kódu, je Intelli Code od Microsoftu [58]. Jeho cílem je maximálně usnadnit práci při psaní a úpravách kódu za pomoci technik umělé inteligence. Jedná se o rozšíření pro vývojové prostředí Visual Studio s podporou několika programovacích jazyků, mezi něž patří C++ nebo C#. Rozšíření je možné také využít ve Visual Studio Code s podporou jazyků jako Java, Python, JavaScript, TypeScript a SQL. Podobně jako GitHub Copilot i Intelli Code vychází z dat získaných z open source projektů na GitHub. Díky těmto datům je umožněno navrhovat dokončení kódu tak, aby odpovídalo s co největší pravděpodobností běžné praxi s ohledem na kontext psaného kódu.

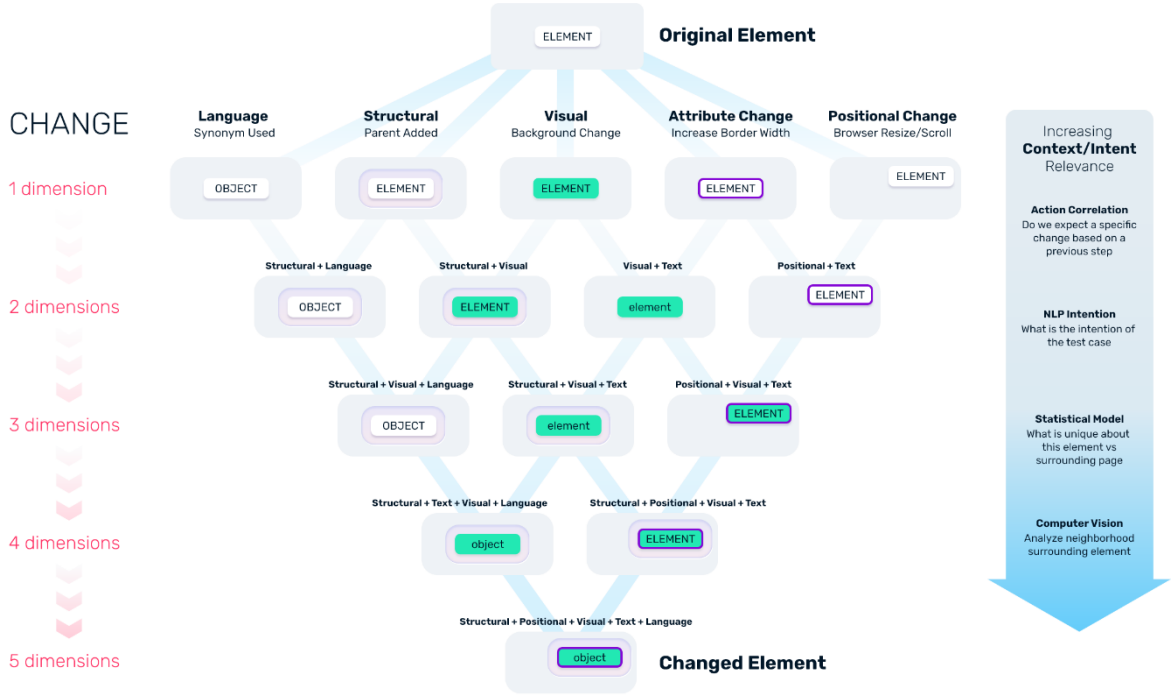


Obrázek 16. IntelliCode – návrh dokončení kódu [59]

Dokončit umí nejen známé API, ale také argumenty funkcí. Kromě návrhů dokončování kódu a doplňování celých řádků kódu se můžeme setkat ještě s další funkcí, a to při provádění úprav. Intelli Code sleduje prováděné úpravy v kódu a rozpozná, že se jedná o opakující se akci na více místech, proto vyhledá doporučené části kódu, kde by mohla být požadována stejná úprava. Intelli Code má svá omezení a některé funkce zatím fungují jen pro jazyk C#.

Nezbytnou součástí každého vývoje nové aplikace, a také po jakémkoliv zásahu do jejího zdrojového kódu, je testování. K otestování správné funkčnosti je třeba zvolit adekvátní druh testování podle prováděných úprav. Jednotlivé přístupy k testování byly podrobněji rozebrány v kapitole 3.5 Testování a integrace. Nyní se zaměříme na aktuálně dostupné nástroje, které umožňují podporu testování za využití umělé inteligence.

Zapojení strojového učení a počítačového vidění do testování umožňuje nástroj Functionize [60] určený pro testování webových stránek. Ten umožňuje tvorbu testů dvěma způsoby. Jedním z nich je psaní popisů testů v podobě přirozeného jazyka (v angličtině) a tento text je následně pomocí NLP převeden do podoby automatizovaných testů. Druhý přístup je založen na strojovém učení, kdy tester prochází aplikací a posloupnost jednotlivých kroků je zaznamenávána. Poté, co jsou takto vytvořeny testovací případy, lze testy již spouštět automatizovaně. Pomocí nástroje je například možné automaticky vyplňovat data formulářů adekvátními daty nebo ověřovat odeslání e-mailu na náhodnou e-mailovou adresu vygenerovanou pro provádění testu. V případě, že dojde k úpravám grafického rozhraní, provádění testů neselže a je přizpůsobeno novým úpravám. To platí nejen v případě upravení pozice elementů na stránce, ale i v případě vizuálních nebo jazykových modifikací stránky apod. Na následujícím obrázku (obrázek 17) je znázorněno, jakým způsobem je pohlíženo na element na webové stránce. Změnu prvku lze sledovat ve více dimenzích – jazyk, struktura, vizuální, typ atributu a pozice.

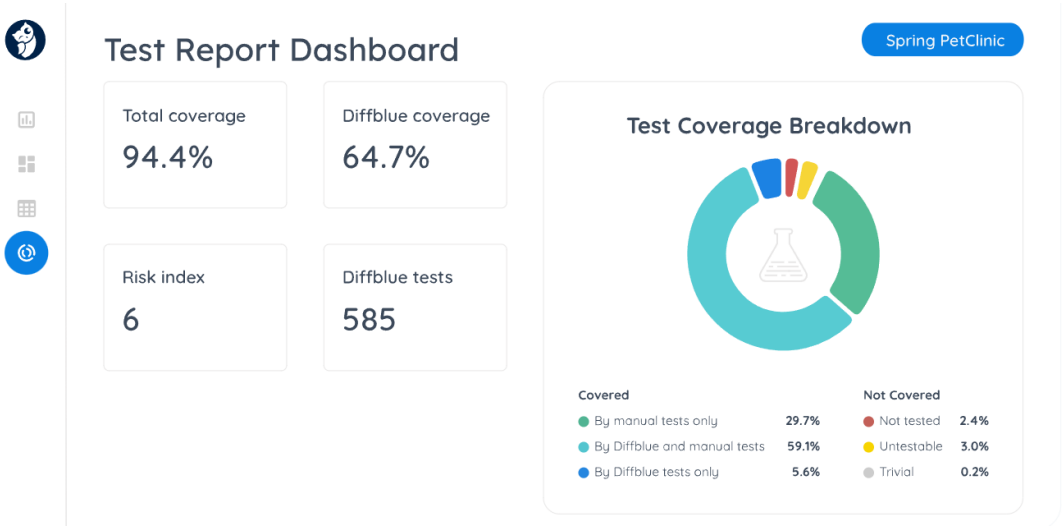


Obrázek 17. Functionize – sledování elementu webové stránky [61]

V okamžiku, kdy jsme do kódu zanesli změny, je nutné provádět regresní testování. Ujistíme se tak, že provedené změny negativně neovlivnily již fungující části programu. Nástroj Diffblue Cover [62] je určený pro psaní regresních testů pro jazyk Java. Využívá při tom zpětnovazebního učení. Spuštěním tohoto nástroje dojde k analýze kódu a jsou vygenerovány

testy pro jednotlivé metody nebo třídy. Ty lze využít při continuous integration, kdy dochází k regresnímu testování mezi jednotlivými verzemi kódu. Testování je nedílnou součástí všech úprav zdrojového kódu, proto by měl být kladen důraz na vhodný přístup k provádění testů a měla by být snaha maximálně pokrýt zdrojový kód testy. Tyto testy by měly být vždy aktualizovány podle změn, které byly v projektu provedeny. Tím je zajištěno správné otestování.

V tomto je hlavní přínos nástroje Diffblue Cover [62]. Nástroj dokáže vytvořit kompletní sadu testů pro celý projekt. Tu udržuje stále aktuální i přes prováděné změny v kódu. Na základě analýzy je zobrazeno pokrytí kódu testy a případná rizika.



Obrázek 18. Diffblue Cover – report pokrytí testů [62]

Z tohoto přehledu vybraných nástrojů vyplývá, že se již nyní techniky umělé inteligence uplatňují v praxi a můžeme nalézt nástroje, které jsou dostupné pro vývojáře. V každé z fází softwarového procesu se máme možnost setkat s přínosy umělé inteligence. Jednotlivé společnosti zabývající se jejich vývojem si kladou různé podmínky pro jejich využívání, ale ve většině případů je možné seznámit se s produktem alespoň v nějaké omezené náhledové verzi. Nástrojů, které by mohly zefektivnit softwarový proces za pomoci umělé inteligence, je dnes již možné dohledat celou řadu. Další specializované nástroje bychom mohli doporučit podle toho, na kterou oblast vývoje bychom se zaměřili a jaké bychom měli specifické nároky.

5 ZHODNOCENÍ PŘÍNOSŮ UMĚLÉ INTELIGENCE V SOFTWAROVÉM INŽENÝRSTVÍ

Oblast softwarového inženýrství se v čase neustále proměňuje a vyvíjí. S příchodem nových zařízení, technologií a s komplexností počítačových systémů se proměňují také požadavky na software, který má na těchto zařízeních běžet. V této práci jsem se zaměřil na techniky umělé inteligence, které nabízí spoustu řešení pro mnohé z výzev souvisejících s vývojem a údržbou softwaru. Některé odborné články navrhují pouze možná teoretická řešení, ale nezabývají se podrobně postupem, jak je implementovat. Ve třetí kapitole jsem představil konkrétní navrhovaná řešení pro některé problémy softwarového inženýrství.

V rámci studie prováděné Barenkam a kol. [8] byly provedeny rozhovory se softwarovými inženýry s různými zkušenostmi k tomuto tématu. Na základě názorů odborníků ze zmíněné studie a zjištění získaných z analýzy odborných publikací zmíněných v rámci této práce, uvádím svůj pohled na probíranou problematiku. Rozhodnutí využít názory respondentů ze zmíněné studie jsem učinil proto, abych do zhodnocení výsledků zahrnul i pohled odborníků z praxe, kteří se aktivně setkávají s těmito přístupy a mnohou lépe posoudit jejich přínos.

Jako hlavní přínos umělé inteligence do softwarového inženýrství hodnotím přenechání opakujících se činností na automatizovaných nástrojích. To umožňuje lidem věnovat se kreativním činnostem a inovacím. Tím je dosaženo zvýšení efektivity celého softwarového procesu. Na základě mých zjištění nelze říci, že by se některá konkrétní technologie z oblasti umělé inteligence vztahovala ke konkrétním činnostem softwarového inženýrství nebo ke konkrétní fázi softwarového procesu. Každý z autorů navrhuje originální přístupy k řešení jednotlivých problémů. Zjištěné poznatky pro každou z fází vývoje jsem shrnul v následujícím přehledu.

Plánování

V oblasti plánování dochází ke zlepšením oproti využití pouze tradičních metod. Důvody, díky kterým dochází ke zlepšení, vychází z omezení lidského faktoru a subjektivity, která je zanášena nebo jsou na ní některé tradiční metody odhadování přímo závislé. Umělá inteligence dokáže díky prozkoumání množství dat z dřívějších projektů předcházet možným selháním. V současné chvíli na trhu již existují nástroje pro plánování využívající umělou inteligenci. Mezi omezení AI v této oblasti můžeme zařadit neschopnost kreativní činnosti.

Dle jednoho z respondentů [8] je již nyní dosaženo lepšího přehledu o nákladech a časové náročnosti projektu.

Analýza

Analýza je jednou ze zásadních fází softwarového procesu. Vytvoření správných požadavků a jejich srozumitelná dokumentace je zásadní pro všechny další prováděné úkony. Techniky umělé inteligence lze využít nejen pro zjišťování požadavků, ale také pro jejich prioritizaci. Hlavním omezením je v současné chvíli to, že AI může sice napomáhat se zjišťováním či tříděním požadavků, nelze však zatím z této činnosti odstranit lidský faktor. Aktuální přínos můžeme spatřit v napomáhání zlepšit získané požadavky, aby byly úplné a správně definované a byly zohledněny rizikové faktory.

Návrh

Tvorba návrhu byla a zatím stále je činností závislou na lidském přístupu. Samotné rozhodnutí o vhodném způsobu návrhu stále závisí na zkušenostech a subjektivitě. Umělá inteligence zatím nenabízí způsob jak převést problémy reálného světa do rozhodnutí pro volbu návrhu. Umožňuje však kontrolu a vyhodnocení člověkem navrhovaných řešení. Ověřuje logiku programu a umožňuje vyřešit rutinní činnosti bez lidského zásahu, což významně ulehčuje návrhářům práci. Na tomto přínosu se shodují i respondenti studie [8] a uvádí jej jako největší přínos AI pro tvorbu návrhu.

Implementace

Přestože se objevila myšlenka Softwaru 2.0, implementace probíhá aktuálně nejčastěji formou programovacího jazyka. Ve 4. kapitole jsou zmíněny nástroje, které mají za cíl spolupracovat s programátorem, například formou predikce vhodných metod, případně poskytují příklady kódu pro konkrétní problém. Samotná implementační činnost je zatím ponechána člověku, ale je mu poskytována významná podpora. V případě teoretického ponechání kódování čistě na umělé inteligenci se vyskytují obavy o zachování srozumitelnosti a čitelnosti kódu pro člověka.

Testování

Provádění testování je nezbytnou součástí každého vývoje. Testování zahrnuje velké množství technik a přístupů. Největší přínos technik umělé inteligence byl zaznamenán zejména při generování a návrhu testovacích případů [32]. Kromě toho pomáhají chybám již předcházet. Mezi nejčastěji se vyskytující techniky patří genetické algoritmy, zpětnovazební učení a umělé neuronové sítě. Studie, kterou provedl Bataresh a kol., [14] poznamenává, že se problematice testování věnovalo nejvíce výzkumných prací zkoumajících aplikace metod umělé inteligence do softwarového procesu.

Údržba

Pro udržení funkčního systému je nutné věnovat úsilí jeho údržbě. Ta se může skládat z různých činností. Hlavním přínosem umělé inteligence v této oblasti je podpora právě dílčích činností řízených člověkem. Jedná se mj. o nástroje, které dokáží analyzovat kód a upozorňovat na problematická místa. Díky tomu je snazší přizpůsobit dříve napsaný kód novým úpravám. V současnosti není možné svěřit údržbu jedinému nástroji, který by ji dokázal nezávisle vykonávat.

Všechny navrhované přístupy, které jsou zahrnuty v této práci, usilují o dosažení větší efektivity softwarového inženýrství. Především se jedná o snížení nákladů na vývoj. To úzce souvisí s optimalizací jednotlivých procesů. Nesetkal jsem se s návrhem, který by se snažil obecně vyřešit celý softwarový proces, a nahradit tak tradiční postupy a lidské řízení. Barenkamp a kol. [8] také poznamenává, že na základě jejich zjištění jsou přístupy využívající technik umělé inteligence nejčastěji vytvářeny na základě pokusů a omylů a hledáním řešení, která fungují nejlépe pro daný účel, nebo jinými neformálními způsoby.

V současnosti jsou návrhy zaměřeny na zlepšení dílčích činností a podporu inženýrů. Z mého pohledu nyní není aktuální předpoklad, že by měla umělá inteligence v blízké době plně nahradit lidskou činnost. K tomuto názoru se přiklánějí také oslovení respondenti [8]. Hlavní roli ve vývoji a kreativních inovacích budou mít podle nich na starosti dále lidé. Přesto připouští, že hodnotit přesné budoucí směřování vývoje nových technologií je nemožné. I v současnosti je počet dostupných nástrojů pro softwarové inženýrství založených

na technikách umělé inteligence stále omezený a nedokážeme předvídat, jakými možnostmi budou v budoucnu disponovat.

V průběhu analýzy jsem se nesetkal s tím, že by některý z autorů kritizoval přístupy umělé inteligence a hodnotil je jako nevyhovující pro oblast softwarového inženýrství. Osobně jsem přesvědčen, že nástroje, které nebudou využívat potenciál, který jim nabízí moderní technologie, včetně umělé inteligence, budou mít problém udržet konkurenceschopnost a postupně bude zavádění zmíněných moderních technik nezbytnou součástí softwarového inženýrství.

ZÁVĚR

V rámci této práce jsem se zaměřil na prozkoumání klíčových oblastí softwarového inženýrství. Na základě odborné literatury jsem analyzoval, jaké jsou možnosti využití technik umělé inteligence v životním cyklu vývoje softwaru. Na základě takto provedené analýzy jsem navrhl softwarový proces. Následně jsem zhodnotil získané informace a vysvětlil, kde se nachází možnosti pro aplikaci technik umělé inteligence v softwarovém inženýrství.

V první a druhé kapitole je uveden stručný úvod do probírané problematiky. První kapitola je úvodem do softwarového inženýrství, vysvětluje historický vývoj a jsou popsány základní pojmy. Druhá kapitola se zaměřuje na vysvětlení pojmu umělá inteligence. Vzhledem k množství přístupů a různých úhlů, jak lze na tuto oblast nahlížet, bylo upuštěno od přesné klasifikace jednotlivých metod. Pojem umělá inteligence zde chápu jako zastřešující pojem pro všechny techniky, které je možné dohledat v různých odborných publikacích. Jelikož nebylo cílem dopodrobna rozebrat principy, na kterých tyto techniky staví, věnuji se převážně jejich využití.

Ve třetí kapitole jsem prošel klíčové fáze softwarového procesu. Cílem bylo vystihnout stěžejní činnosti, které se v těchto fázích vykonávají. Zaměřoval jsem se obecně na všechny oblasti, aby vznikl celkový přehled. Na základě prostudované odborné literatury jsem zanalyzoval přístupy a přínos umělé inteligence v každé z šesti fází. Jsou zmíněny výhody i rizika, která přinášejí nové technologie oproti tradičním přístupům.

Čtvrtá kapitola se věnuje návrhu softwarového procesu. S ohledem na rychlost, s jakou vznikají nové poznatky ohledně umělé inteligence a s množstvím článků, které se tomuto tématu věnuje, nelze stanovit nějaký pevný navrhovaný rámec pro dlouhodobě platná doporučení vyplývající ze získaných poznatků. Proto jsem přistoupil k formě doporučení aktuálních nástrojů, které jsou dostupné na trhu. Jedná se o komerční nástroje, proto k nim jsou omezené veřejně dostupné informace z hlediska přesného fungování. Proto jsem vyzdvihl zejména výhody, které do oblasti softwarového inženýrství přináší. Nicméně informace o nástrojích se od doby psaní tohoto textu mohly změnit a mohly být doplněny nebo upraveny některé jejich funkce.

Pátá kapitola je věnována zhodnocení poznatků získaných prostudováním dostupné literatury a provedených studií na toto téma. Věnuji se zde každé z klíčových fází a provádím zhodnocení na základě zjištěných informací a také na základě dotazovaných vývojářů ze studie Barenkamp a kol. [8], abych zohlednil také názor lidí z praxe.

Umělá inteligence má velký potenciál pro rozvoj softwarového inženýrství. Samozřejmě naráží na různé limity a je velmi obtížné prorokovat, jakým způsobem se bude tato oblast rozvíjet v budoucnu. V současné chvíli není možné vyloučit ze softwarového cyklu lidský přístup. Z provedených studií, ale i z dostupných nástrojů vyplývá, že umělá inteligence nabízí podporu a optimalizuje jednotlivé činnosti softwarového projektu. Nelze jí však zatím svěřit řízení těchto činností, aby se stala nezávislá na člověku. To zároveň vidím jako obrovské pole možností pro další výzkum.

SEZNAM POUŽITÉ LITERATURY

[1] TURING, Alan. Computing machinery and intelligence. Mind. Oxford University Press on behalf of the Mind Association, 1950, 1950(236), 433-460.

[2] WIRTH, Niklaus. A Brief History of Software Engineering. IEEE Annals of the History of Computing. IEEE Computer Society, 2008, s. 33. 1058-6180.

[3] SOMMERVILLE, Ian. Softwarové inženýrství. Brno: Computer Press, 2013. ISBN 978-80-251-3826-7.

[4] RUSSELL, Stuart a Peter NORVIG. Artificial intelligence: a modern approach. 3rd ed. Upper Saddle River: Prentice Hall, 2009. ISBN 978-0-13-604259-4.

[5] LAPLANTE, Phillip A. Dictonary of computer science, engineering, and technology. Florida: CRC Press, 2001. ISBN 0-8493-2691-5.

[6] CONTRERAS, Iván a Josep VEHÍ. Artificial Intelligence for Diabetes Management and Decision Support: Literature Review. Journal of Medical Internet Research. 2018, 20, 1–7. Dostupné z: doi:10.2196/10775

[7] KISSELV, Alexej. The software Devleopment life Cyle. Alexej Kisselev [online]. 2022 [cit. 2022-01-22]. Dostupné z: https://www.kisselev.de/

[8] BARENKAMP, Marco, Jonas REBSTADT a Oliver THOMAS. Applications of AI in classical software engineering. AI Perspectives. 2020, , 1-15. Dostupné z: doi:https://doi.org/10.1186/s42467-020-00005-4

[9] VAN DER LINDEN, Jos. Revisiting the problems that led to Brooks's Law. Gent, 2020. Disertační práce. Ghent University. Vedoucí práce Prof. Dr. Geert Poels.

[10] MORGENSHTERN, Ofer, Tzvi RAZ a Dov DVIR. Factors affecting duration and effort estimation errors in software development projects. Information and Software Technology. 2007, (49), 827-837. Dostupné z: doi:https://doi.org/10.1016/j.infsof.2006.09.006

[11] MAHADIK, Avinash. An Improved Ant Colony Optimization Algorithm for Software Project Planning and Scheduling. International Journal of Advanced Engineering and Global Technology. 2014, 2(1), 400-405. ISSN 2309-4893.

[12] FENTON, Norman, Peter HEARTY, Martin NEIL a Łukasz RADLIŃSKI. Software Project and Quality Modelling Using Bayesian Networks. 1-25. Dostupné z: doi:10.4018/978-1-60566-758-4.ch001

[13] AZZEH, Mohammad, Daniel NEAGU a Peter I. COWLING. Fuzzy grey relational analysis for software effort estimation. Empirical Software Engineering. 2010, 15, 60-90. Dostupné z: doi:10.1007/s10664-009-9113-0

[14] BATARSEH, Feras A., Rasika MOHOD, Abhinav KUMAR a Justin BUI. The application of artificial intelligence in software engineering: a review challenging conventional wisdom. BATARSEH, Feras A. a Ruixin YANG. Data Democracy: At the Nexus of Artificial Intelligence, Software Development, and Knowledge Engineering. 1. Elsevier, s. 179-232. ISBN 978-0-12-818366-3.

[15] BALZER, Robert, Neil GOLDMAN a David WILE. Informality in Program Specifications. IEEE Transactions on Software Engineering. 4(2), 94-103.

[16] SERNA, Edgar M., Oscar S. BACHILLER a Alexei A. SERNA. Knowledge meaning and management in requirements engineering. International Journal of Information Management. 37(3), 155-161. ISSN 0268-4012. Dostupné z: doi:10.1016/j.ijinfomgt.2017.01.005

[17] WANG, W. M., C. F. CHEUNG, W. B. LEE a S. K. KWOK. Mining knowledge from natural language texts using fuzzy associated concept mapping. Information Processing and Management. Pergamon Press, 44(5), 1707–1719. ISSN 0306-4573. Dostupné z: doi:10.1016/j.ipm.2008.05.002

[18] NINAUS, Gerald, Alexander FELFERNIG, Martin STETTINGER, Stefan REITERER, Gerhard LEITNER, Leopold WENINGER a Walter SCHANIL. INTELLIREQ: Intelligent techniques for software requirements engineering. Frontiers in Artificial Intelligence and Applications. 2014, (263), 1-6. Dostupné z: doi:10.3233/978-1-61499-419-0-1161

[19] VEMURI, Sandeep, Sisay CHALA a Madjid FATHI. Automated use case diagram generation from textual user requirement documents. In: 2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE). 2017. Dostupné z: doi:10.1109/CCECE.2017.7946792

[20] CHAVES-GONZÁLEZ, José, Miguel PÉREZ-TOLEDANO a Amparo NAVASA. Teaching learning based optimization with Pareto tournament for the multiobjective software requirements selection. Engineering Applications of Artificial Intelligence. 2015, (43), 89-101. ISSN 0952-1976. Dostupné z: doi:https://doi.org/10.1016/j.engappai.2015.04.002

[21] SCHACH, Stephen R. Object-Oriented and Classical Software Engineering. Eighth Edition. New York: McGraw-Hill, 2011. ISBN 978-0-07-337618-9.

[22] DIXON, J. R., M. K. SIMMONS a P. R. COHEN. An Architecture for Application of Artificial Intelligence to Design. 21st Design Automation Conference Proceedings. 1984, 634-640. Dostupné z: doi:10.1109/DAC.1984.1585866

[23] CHANDA, NagaPrashanth a Xiaoqing LIU. Intelligent analysis of software architecture rationale for collaborative software design. In: *2015 International Conference on Collaboration Technologies and Systems (CTS)*. 2015, s. 287-294. Dostupné z: doi:10.1109/CTS.2015.7210436

[24] GHOSH, Sutirtha, Prasenjit MUKHERJEE a Baisakhi CHAKRABORTY. Automated Generation of E-R Diagram from a Given Text in Natural Language. In: *2018 International Conference on Machine Learning and Data Engineering (iCMLDE)*. 2018. Dostupné z: doi:10.1109/iCMLDE.2018.00026

[25] GILSON, Fabian, Matthias GALSTER a Francois GEORIS. Extracting Quality Attributes from User Stories for Early Architecture Decision Making. In: *International Conference on Software Architecture Workshops (ICSAW)*. 2019. Dostupné z: doi:10.1109/ICSA-C.2019.00031

[26] CUNHA, Warteruzannan Soyer, Guisella Angulo ARMIJO a Valter Vieira de CAMARGO. InSet: A Tool to Identify Architecture Smells Using Machine Learning. *In 34th Brazilian Symposium on Software Engineering (SBES '20)*. 2020, 760-765. Dostupné z: doi:10.1145/3422392.3422507

[27] KARPATHY, Andrej. Software 2.0. Karpathy.medium.com [online]. 2017 [cit. 2022-02-18]. Dostupné z: https://karpathy.medium.com/software-2-0-a64152b37c35

[28] LAKE, Brenden M., Tomer D. ULLMAN, Joshua B. TENENBAUM a Samuel J. GERSHMAN. Building machines that learn and think like people. Behavioral and Brain Sciences. 2017, 40(253). Dostupné z: doi:10.1017/S0140525X16001837

[29] INSAURRALDE, Carlos C. Software Programmed by Artificial Agents toward an Autonomous Development Process for Code Generation. 2013 IEEE International Conference on Systems, Man, and Cybernetics. 2013, 3294-3299. Dostupné z: doi:10.1109/SMC.2013.561

[30] HUSAIN, Hamel, Ho-Hsiang WU, Tiferet GAZIT, Miltiadis ALLAMANIS a Marc BROCKSCHMIDT. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. ArXiv. 2019.

[31] AMAL, Boukhdhir, Kessentini MAROUANE, Bechikh SLIM, Josselin a Ben Said LAMJED. On the Use of Machine Learning and Search-Based Software Engineering for Ill-Defined Fitness Function: A Case Study on Software Refactoring. Proceedings of the 6th International Symposium on Search-Based Software Engineering. 2014, 31-45. Dostupné z: doi:10.1007/978-3-319-09940-8_3

[32] KHALIQ, Zubair, Sheikh Umar FAROOQ a Dawood Ashraf KHAN. Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect. Elsevier. 2022. Dostupné z: doi:arxiv-2201.05371

[33] BRIAND, Lionel, Yvan LABICHE a Zaheer BAWAR. Using Machine Learning to Refine Black-Box Test Specifications and Test Suites. Proceedings - International Conference on Quality Software. 2008, 135-144. Dostupné z: doi:10.1109/QSIC.2008.5

[34] LAST, Mark, Menahem FRIEDMAN a A. KANDEL. Using Data Mining for Automated Software Testing. International Journal of Software Engineering and Knowledge Engineering. 2011, 14. Dostupné z: doi:10.1142/S0218194004001737

[35] KHAN, Rijwan a Mohd. AMJAD. Automatic test case generation for unit software testing using genetic algorithm and mutation analysis. In: 2015 IEEE UP Section Conference on Electrical Computer and Electronics. UPCON, 2015, s. 1-5. Dostupné z: doi:10.1109/UPCON.2015.7456734

[36] ROSENFELD, Ariel, Odaya KARDASHOV a Orel ZANG. Automation of Android Applications Functional Testing Using Machine Learning Activities Classification. In: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems. MOBILESoft, 2018, s. 122-132.

[37] ANSARI, Ahlam, Mirza Baig SHAGUFTA, Ansari Sadaf FATIMA a Shaikh TEHREEM. Constructing Test cases using Natural Language Processing. In: 2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics. AEEICB, 2017, s. 95-99. ISBN 978-1-5090-5434-3. Dostupné z: doi:10.1109/AEEICB.2017.7972390

[38] LI, Huaizhong a Chiou LAM. An ant colony optimization approach to test sequence generation for state based software testing. In: Fifth International Conference on Quality Software. Melbourne: QSIC'05, 2005, s. 255-262. ISBN 0-7695-2472-9. Dostupné z: doi:10.1109/QSIC.2005.12

[39] PADURARU, Ciprian a Marius-Constantin MELEMCIUC. An Automatic Test Data Generation Tool using Machine Learning. In: *Proceedings of the 13th International Conference on Software Technologies*. SCITEPRESS – Science and Technology Publications, 2018, s. 472-481. ISBN 978-989-758-320-9. Dostupné z: doi:10.5220/0006836604720481

[40] LIU, Peng, Xiangyu ZHANG, Marco PISTOIA, Yunhui ZHENG, Manoel MARQUES a Lingfei ZENG. Automatic Text Input Generation for Mobile Testing. In: *2017 IEEE/ACM 39th International Conference on Software Engineering*. 2017, s. 643-653. Dostupné z: doi:10.1109/ICSE.2017.65

[41] TSIMPOURLAS, Foivos, Ajitha RAJAN a Miltiadis ALLAMANIS. Supervised learning over test executions as a test oracle. In: *The 36th ACM/SIGAPP Symposium on Applied Computing*. 2021, 1521-1531. Dostupné z: doi:10.1145/3412841.3442027

[42] SPIEKER, Helge, Arnaud GOTLIEB, Dusica MARIJAN a Morten MOSSIGE. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In: *Proceedings of 26th International Symposium on Software Testing and Analysis*. 2017. Dostupné z: doi:10.1145/3092703.3092709

[43] BADRI, Mourad, Linda BADRI a William FLAGEOL. Investigating the Accuracy of Test Code Size Prediction using Use Case Metrics and Machine Learning Algorithms: An Empirical Study. In: *International Conference on Machine Learning and Soft Computing*. 2017, s. 25-33. ISSN 978-1-4503-4828-7. Dostupné z: doi:10.1145/3036290.3036323

[44] CATAL, Cagatay a Suat GULDAN. Product review management software based on multiple classifiers. *IET Softw*. 2017, (11), 89-92. Dostupné z: doi:10.1049/iet-sen.2016.0137

[45] MEQDADI, Omar, Nouh ALHINDAWI, Jamal ALSAKRAN, Ahmad SAIFAN a Hatim MIGDADI. Mining Software Repositories for Adaptive Change Commits

Using Machine Learning Techniques. *Information and Software Technology*. 2019, (109). Dostupné z: doi:10.1016/j.infsof.2019.01.008

[46] ALHUSAIN, Sultan, Simon COUPLAND, Robert JOHN a Maria KAVANAGH. Towards machine learning based design pattern recognition. In: *2013 13th UK Workshop on Computational Intelligence (UKCI)*. 2013, s. 244-251. Dostupné z: doi:10.1109/UKCI.2013.6651312

[47] DAS, Ananta Kumar, Shikhar YADAV a Subhasish DHAL. Detecting Code Smells using Deep Learning. In: *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*. 2019, s. 2081-2086. Dostupné z: doi:10.1109/TENCON.2019.8929628

[48] *Visual Studio: Integrované vývojové prostředí (IDE) a editor kódu pro vývojáře softwaru a týmy* [online]. Microsoft, 2022 [cit. 2022-04-13]. Dostupné z: https://visualstudio.microsoft.com/cs/

[49] *Jira | Issue & Project Tracking Software | Atlassian* [online]. Atlassian, 2022 [cit. 2022-04-13]. Dostupné z: https://www.atlassian.com/software/jira

[50] *Roadmap features in Jira Software* [online]. Atlassian, 2022 [cit. 2022-04-13]. Dostupné z: https://www.atlassian.com/software/jira/features/roadmaps

[51] *Introducing machine learning-powered "smarts" - Work Life by Atlassian* [online]. Atlassian, 2022 [cit. 2022-05-06]. Dostupné z: https://www.atlassian.com/blog/platform/atlassian-smarts-cloud-launch

[52] *IBM Engineering Requirements Management DOORS Next* [online]. IBM, 2022 [cit. 2022-04-17]. Dostupné z: https://www.ibm.com/products/requirements-management-doors-next/details

[53] *Watson AI can help you write clear, consistent requirements* [online]. IBM, 2022 [cit. 2022-05-06]. Dostupné z: https://www.ibm.com/products/requirements-management-doors-next/demos/ai/watson-ai-can-help-you-write-clear-consistent-requirements

[54] *Uizard AI* [online]. Uizard Technologies, 2022 [cit. 2022-04-17]. Dostupné z: https://uizard.io/ai/

[55] FUENTES, Javier. *How deep learning is transforming design: NLP and CV applications* [online]. Towards Data Science, 2021, 30. 9. 2021 [cit. 2022-04-17]. Dostupné z: https://towardsdatascience.com/how-deep-learning-is-transforming-design-cv-and-nlp-applications-4518c50690e6

[56] *Welcome your new team member: AI* [online]. Uizard Technologies, 2022 [cit. 2022-05-06]. Dostupné z: https://uizard.io/design-assistant/

[57] GitHub Copilot · Your AI pair programmer [online]. GitHub, 2021 [cit. 2022-03-01]. Dostupné z: copilot.github.com

[58] *IntelliCode pro Visual Studio | Microsoft Docs* [online]. Microsoft, 2022 [cit. 2022-04-20]. Dostupné z: https://docs.microsoft.com/cs-cz/visualstudio/intellicode/intellicode-visual-studio

[59] U, Nick. AI-Assisted Code Completion Suggestions Come to C++ via IntelliCode. *Https://devblogs.microsoft.com/* [online]. Microsoft, 2018 [cit. 2022-05-06]. Dostupné z: https://devblogs.microsoft.com/cppblog/cppintellicode/

[60] *Test Creation - Intelligent End-to-End Web Application Testing - Functionize* [online]. Functionize, 2022 [cit. 2022-04-20]. Dostupné z: https://www.functionize.com/test-creation

[61] *Functionize Architect* [online]. Functionize, 2022 [cit. 2022-05-06]. Dostupné z: https://www.functionize.com/architect

[62] *Diffblue Accelerate Java Shift Left | Diffblue* [online]. Diffblue, 2022 [cit. 2022-04-20]. Dostupné z: https://www.diffblue.com/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ACAT	Activities Classification for Application Testing
AGER	Automated ER Diagram Generation
AI	Artificial Intelligence
ANN	Artificial Neural Networks
API	Application Programming Interface
ASCG	Autonomus Software Code Generation
CI	Continuous Integration
COCOMO	Constructive Cost Model
CP	Category-Partition
CRISP-DM	Cross-Industry Standard Process for Data Mining
ER	Entitně-relační
GRA	Grey Relational Analysis
GUI	Graphic User Interface
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IFN	Information Fuzzy Networks
IGA	Interactive Genetic Algorithm
ISARCS	Intelligent Software Architecture Rationale Capture Systém
KBS	Knowledge-Based Systems
LGA	Learning Genetic Algorithm
MCS	Multiple Classifier Systems
MO-TLBO	Multiobjective Version of Teaching-Learning Based Optimization
NATO	North Atlantic Treaty Organization
NLP	Natural Language Processing

NRP	Next Release Problem
Retecs	Reinforced Test Case Selection
RNN	Recurrent Neural Network
RQA	Requirements Quality Assistant
SDA	Software Developer Agent
SDLC	Software Development Life Cycle
SIR/REX	Systems Information Resource/Requirements Extractor
SQL	Structured Query Language
STLC	Software Testing Life Cycle
TC	Test Case
TLOC	Test Lines Of Code
TS	Test Suite
UCP	Use Case Points
UML	Unified Modeling Language

SEZNAM OBRÁZKŮ