

Univerzita Jana Evangelisty Purkyně  
v Ústí nad Labem  
Přírodovědecká fakulta



Využití generativní AI pro detekci a opravu  
zápachů v kódu

BAKALÁŘSKÁ PRÁCE

**Vypracoval:** Lukáš Priban

**Vedoucí práce:** Ing. Mgr. Pavel Beránek

**Studijní program:** Aplikovaná informatika

**Studijní obor:** Informační systémy

ÚSTÍ NAD LABEM 2024



# UNIVERZITA JANA EVANGELISTY PURKYNĚ V ÚSTÍ NAD LABEM

Přírodovědecká fakulta

Akademický rok: 2023/2024

## ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jméno a příjmení: Lukáš PRIBAN  
Osobní číslo: F21270  
Studijní program: B0613P140005 Aplikovaná informatika  
Téma práce: Využití generativní AI pro detekci a opravu zápachů v kódu  
Zadávající katedra: Katedra informatiky

### Zásady pro vypracování

Evoluce softwaru je jedna z nejdražších fází životního cyklu vývoje softwaru. Automatizační techniky, které ušetří vývojářskému týmu lidský čas, jsou přínosem pro mnohé zainteresované osoby. Možnost automatizace evolučního procesu se otevírá díky neustále se rozvíjejícím velkým jazykovým modelům. Stále však není jisté, jak přínosné v procesu evoluce velké jazykové modely jsou. Cílem práce je posoudit refaktorizační schopnosti velkých jazykových modelů v pravidelné údržbě kódu ve fázi evoluce životního cyklu procesu vývoje softwaru.

Osnova:

1. Úvod a cíle závěrečné práce
2. Přehled současného stavu problematiky
3. Teoretická východiska
  - 3.1. Životní cyklus procesu vývoje softwaru
  - 3.2. Fáze evoluce
  - 3.3. Zápachy v kódu
  - 3.4. Velké jazykové modely
  - 3.5. Inženýrství výzev (*prompt engineering*)
4. Praktická část
  - 4.1. Metodika výzkumu
  - 4.2. Tvorba korpusu kódů se zápachy
  - 4.3. Optimalizace výzev a profilů
  - 4.4. Posouzení refaktorizačních schopností
5. Výsledky výzkumu
6. Diskuse
7. Závěr

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

Seznam doporučené literatury:

1. CÁMARA, Javier, Javier TROYA, Lola BURGUEÑO a Antonio VALLECILLO. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modelling*. 2023, 22(3), 781-793. ISSN 1619-1366.
2. FOWLER, Martin a Kent BECK. *Refactoring: Improving the Design of Existing Code*. London: Pearson Education, 2019. ISBN 978-0134757599.
3. VASWANI, Ashish, Noam SHAZEER, Niki PARMAR, Jakob USZKOREIT, Llion JONES, Aidan N. GOMEZ, Lukasz KAISER a Illia POLOSUKHIN. Attention Is All You Need. *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, 2017, 6000-6010.

Vedoucí bakalářské práce: **Ing. Mgr. Pavel Beránek**  
Katedra informatiky

Datum zadání bakalářské práce: **2. listopadu 2023**  
Termín odevzdání bakalářské práce: **15. července 2024**

L.S.

---

**doc. RNDr. Michal Varady, Ph.D.**  
děkan

---

**RNDr. Jiří Škvor, Ph.D.**  
vedoucí katedry

## **Prohlášení**

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a použil jen pramenů, které cituji a uvádím v přiloženém seznamu literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., ve znění zákona č. 81/2005 Sb., autorský zákon, zejména se skutečností, že Univerzita Jana Evangelisty Purkyně v Ústí nad Labem má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Jana Evangelisty Purkyně v Ústí nad Labem oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladu, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

V Ústí nad Labem dne 16. července 2024

Podpis: .....



Děkuji vedoucímu práce Ing. Mgr. Pavlu Beránkovi  
za neocenitelné rady a pomoc při tvorbě bakalářské práce.  
Dále děkuji všem svým nejbližším, kteří mě po dobu celého studia podporovali.





**Abstrakt:**

Tato bakalářská práce se zabývá možnostmi velkých jazykových modelů (LLM) v refaktORIZACI kódu obsahujícího zápachy, které způsobují zvýšení nákladů na další vývoj. Cílem práce je posoudit refaktORIZAČNÍ schopnosti velkých jazykových modelů tyto zápachy detekovat a opravit. Výstupy této práce tvoří korpus kódů se zápachy v jazyce JavaScript, optimalizovaná výzva pro refaktORIZACI kódu pomocí LLM a vyhodnocení úspěšnosti modelu GPT-4o při identifikaci a refaktORIZACI zápachů na základě této výzvy. Tato práce má přínos pro softwarové vývojáře, kteří mohou využít výzvu nebo poznatky z této práce pro zlepšení kvality a udržitelnosti svého kódu.

**Klíčová slova:** softwarové inženýrství, LLM – velké jazykové modely, refaktORIZACE, zápachy v kódu

USING GENERATIVE AI TO DETECT AND REPAIR CODE SMELLS

**Abstract:**

This bachelor's thesis explores the capabilities of large language models (LLMs) in refactoring code containing smells, which cause increased development costs. The aim of the thesis is to assess the refactoring abilities of large language models to detect and fix these smells. The outputs of this work include a dataset of code with smells in JavaScript, an optimized prompt for refactoring code using LLMs, and an evaluation of the success of the GPT-4o model in identifying and refactoring smells based on this prompt. This thesis is beneficial for software developers who can use the prompt or insights from this work to improve the quality and maintainability of their code.

**Keywords:** software engineering, LLM – large language models, refactoring, code smells



# Obsah

<b>Úvod</b>	<b>17</b>
<b>1. Teoretická východiska</b>	<b>19</b>
1.1. Životní cyklus vývoje softwaru (SDLC)	19
1.2. Softwarová evoluce	20
1.3. Zápachy v kódu	24
1.4. Velké jazykové modely	27
1.5. Inženýrství výzev (prompt engineering)	32
<b>2. Praktická část</b>	<b>41</b>
2.1. Metodika výzkumu	41
2.2. Tvorba korpusu kódů se zápachy	42
2.3. Explorační fáze	43
2.4. Optimalizace výzvy	48
2.5. Výsledky refaktORIZACE	54
<b>3. Diskuse</b>	<b>61</b>
<b>4. Závěr</b>	<b>63</b>
<b>A. Externí přílohy</b>	<b>69</b>



# Seznam obrázků

1.1. Příklad pozic slov v prostoru. Slova označující zvířata se nachází ve shluku v porovnání se slovem „domy“. Patrný je i stejný rozdíl mezi sémanticky podobnými slovy ale s opačným pohlavím. [14]	30
2.1. Graf četnosti hodnocení refaktORIZACE	58
3.1. Graf porovnání četnosti hodnocení refaktORIZACE mezi problémovými doménami	62



# Seznam tabulek

2.1. Výsledky refaktORIZACE kódu . . . . .	58
--	----





# Úvod

S příchodem a rychlým rozvojem velkých jazykových modelů v posledních letech se otevírá možnost automatizace mnoha fází vývoje softwaru. Tato bakalářská práce zkoumá možnosti velkých jazykových modelů ve fázi, která nastává po nasazení softwaru do provozu. Tuto fázi, nazývanou evoluční, provází jev známý jako technický dluh, který způsobuje zvyšování nákladů na další vývoj. Cílem práce je posoudit schopnosti velkých jazykových modelů eliminovat druh tohoto technického dluhu, známý jako zápachy v kódu.

S pokračující digitalizací a automatizací procesů kolem nás se stává softwarové inženýrství, jehož jsou fáze vývoje softwaru součástí, stále důležitějším. Schopnosti a rychlost generování aktuálně nejpokročilejších veřejně dostupných velkých jazykových modelů, jako je například GPT-4o, nepochybně představují příležitost pro výzkum jejich uplatnění jako nástroje pro automatizaci vývoje softwaru. Ačkoliv má cíl této práce daleko od vytvoření plně autonomního „AI inženýra“, může být považován za malý krok tímto směrem, kterým se možná v následujících letech softwarové inženýrství vydá.

Tato práce se snaží dosáhnout svého cíle pomocí několika kroků. Prvním je vytvoření korpusu kódů, které obsahují zápachy. Dalším krokem je optimalizace výzvy, neboli způsobu, kterým je jazykovému modelu kód předložen. Posledním krokem je vyhodnocení úspěšnosti modelu zápachy identifikovat a eliminovat, na jehož základě jsou formulovány závěry.



# 1. Teoretická východiska

## 1.1. Životní cyklus vývoje softwaru (SDLC)

Životní cyklus vývoje softwaru (SDLC) je proces, který zahrnuje fáze vývoje softwarového produktu od jeho vzniku až po jeho vyřazení. SDLC je strukturovaným přístupem k vývoji softwaru, jehož cílem je zlepšit kvalitu výsledného produktu a zároveň zvládnout složitost procesu vývoje. Dodržováním SDLC mohou vývojové týmy projekt systematicky řídit, včas identifikovat potenciální rizika a zajistit efektivní využití zdrojů [1]. Tato sekce slouží jako přehled SDLC a popisuje jeho jednotlivé fáze.

Ačkoli existují různé druhy SDLC, včetně vodopádového, agilního a spirálového modelu, všechny obecně zahrnují nějakou formu následujících fází:

1. **Analýza požadavků:** Tato počáteční fáze zahrnuje sběr a analýzu požadavků od zainteresovaných stran, aby bylo jasné, co má softwarový produkt dělat a jak má fungovat [1].
2. **Návrh:** Na základě shromážděných požadavků se ve fázi návrhu nastíní, jak bude software strukturován. Zahrnuje vytváření diagramů a dokumentace, která software blíže specifikuje. [1]
3. **Implementace:** V této fázi vývojáři převedou návrh do spustitelného kódu. Dochází k samotné konstrukci softwarového produktu s využitím programovacích jazyků a vývojových nástrojů. [1]
4. **Testování:** Jakmile je software vyvinut, prochází důkladným testováním, aby se zjistily a opravily případné chyby nebo problémy. Testy jsou navrženy na míru podle požadavků zjištěných v první fázi, čímž se zajišťuje, že software původní požadavky splňuje. [1]
5. **Nasazení:** Po testování je software nasazen do produkčního prostředí, kde je k dispozici koncovým uživatelům. Tato fáze může zahrnovat také školení uživatelů a přípravu dokumentace k nasazení. [1]
6. **Údržba:** V poslední fázi SDLC se provádí aktualizace softwaru za účelem přidání nových funkcí, opravy chyb nebo zlepšení výkonu. Údržba je průběžný proces, který prodlužuje životnost softwarového produktu [2]. Právě tato fáze je předmětem zájmu této práce.

### Údržba softwaru

Po nasazení softwaru do provozu se obvykle projeví problémy, které během předchozích fází vývoje nebyly zjištěny. Může se jednat o chyby v kódu nebo funkčnosti, ale také o změny v požadavcích uživatelů/zadavatele nebo změny prostředí (technologií), ve kterém je software nasazen. Proto je nutné na konec životního cyklu softwaru zahrnout fázi údržby, která se těmito problémy zabývá. Jejím cílem je zajistit, že software bude kontinuálně poskytovat své služby. [2]

Údržba softwaru je často výraznou částí nákladů na vývoj softwaru [2]. Z toho důvodu je vhodným kandidátem pro automatizaci pomocí generativní umělé inteligence.

### 1.2. Softwarová evoluce

Koncept softwarové evoluce je v oblasti vývoje softwaru důležitým tématem již od jeho počátků. Tento termín, který použil Mark I. Halpern již kolem roku 1965, neměl v průběhu let jasné danou definici a mnozí jej zaměňovali se softwarovou údržbou [3]. V knize [3] je rozdíl mezi těmito pojmy vysvětlen tak, že zatím co údržba se zaměřuje na prevenci selhání nebo úpadku softwaru, evoluce je zaměřena na neustálé zlepšování softwaru z méně optimálního směrem k perfektnímu stavu. Jiní autoři zase za údržbu považují všechny úpravy provedené po nasazení softwaru do provozu a evoluci za úpravy provedené na základě změn v požadavcích uživatelů/zadavatelů nebo v prostředí, ve kterém je software nasazen. [3]

Obecně mají všechny zainteresované strany při softwarové evoluci požadavky na co nejrychlejší a nejlevnější dodání nových změn. Dalšími požadavky jsou ale také zachování spolehlivosti a udržitelnosti softwaru po provedených změnách [3]. Ignorování těchto požadavků může vést k tzv. technickému dluhu, který může způsobit zvýšení nákladů na budoucí změny [4]. Téma technického dluhu je v této práci více popsáno před částí o refaktoringu, který s ním úzce souvisí.

### Typy programů v softwarové evoluci

V softwarové evoluci se klasifikují evolvované programy do tří skupin – SPE (Specified, Problem, Evolving), navržených Meirem M. Lehmanem kolem roku 1980. Tato klasifikace pomáhá rozlišit způsoby, jimiž softwarové systémy odolávají změnám v průběhu času, a které jsou založené na interakci programů s vlastním prostředím a povaze problémů, které řeší. [3]

**S-typové programy** (specifikované) jsou specifické jejich úplnými a definitivními specifikacemi, které neobsahují žádné nejasnosti pro implementaci. Takové programy jsou navrženy pro řešení problémů, které jsou přesně definovány a mají známá řešení. Jakákoliv změna v problému, který program řeší, vyžaduje program zcela nový. Příkladem jsou základní aritmetické operace nebo manipulace s maticemi – problémy, které mají přesná a známá řešení, a pokud by se změnily, vyžadovaly by nový program. [3]

**P-typové programy** (problémové) naopak řeší problémy, u kterých není třeba znát přesný výstup, ale spíše je třeba najít nejlepší možné výstup za daných okolností. Řešení jsou tedy z praktických důvodů pouze odhadována. Příkladem může být program pro hraní šachu, který musí vyhodnotit nespočet potenciálních tahů v omezeném čase, což činí přesné řešení nepraktickým. Tyto programy se přizpůsobují a vyvíjejí prostřednictvím postupných změn a předefinování problému pro dosažení co nejpraktičtějšího řešení. [3]

**E-typové programy** (evolující) figurují v neustále měnících se reálných prostředích. Odlišují se potřebou přizpůsobovat se změnám ve své oblasti, provozním prostředím a rozvíjejícím se očekáváním zainteresovaných stran. Správnost těchto programů je definována spokojeností zainteresovaných stran, nikoliv specifikacemi. Evoluce E-typových programů je vnímána jako smyčka, kdy program mění prostředí ve kterém figuruje, a toto prostředí zase ovlivňuje, jak má program evolvovat. Příkladem takového programu může být internetový obchod. [3]

## Lehmanovy zákony softwarové evoluce

Lehmanovy zákony slouží k pochopení dynamiky softwarové evoluce a popisují co softwarovou evoluci ovlivňuje. Zde je shrnutí těchto zákonů s vysvětlením:

### 1. Zákon neustálých změn:

*„Program typu E, který je používán, je třeba neustále upravovat, jinak se postupně stává méně uspokojivým.“* [5] (překlad autora)

Program typu E vyžaduje neustálé přizpůsobování, aby si zachoval svou použitelnost, která z důvodu měnících se požadavků, prostředí a technologií postupně klesá.

### 2. Zákon nárůstu složitosti:

*„Při evoluci programu se zvyšuje jeho složitost, pokud není vyvíjeno úsilí ji udržovat nebo snižovat.“* [5] (překlad autora)

Složitost programu se zvyšuje v důsledku mnoha po sobě následujících změn, které jsou implementovány na základě zákonů 1 a 7. Tyto změny snižují kvalitu struktury programu a závislosti mezi jednotlivými částmi systému. Bez úsilí o snižování rychlosti nárůstu složitosti programu se práce potřebná k udržení uspokojivosti programu zvyšuje, což může v důsledku zpomalit další evoluci systému. [5]

### 3. Zákon samoregulace:

*„Proces evoluce programu je samoregulační s téměř normálním rozložením měření atributů produktu a procesu.“* [5] (překlad autora)

Proces vývoje softwaru ve velkých organizacích je samoregulační. To znamená, že v průběhu času mají různé vlastnosti softwaru, jako je velikost a složitost, tendenci následovat normální rozdělení. Vývojové týmy pracují strukturovaně s kontrolními mechanismy nastavenými vedením, které zajišťují dodržování pravidel a dosahování organizačních cílů. Tyto kontrolní

mechanismy vytvářejí zpětnou vazbu, která pomáhá usměrňovat růst softwaru, což vede k předvídatelnému vývoji a charakteristikám. Toto chování je způsobeno řadou nezávislých manažerských a technických rozhodnutí, která vedou k disciplinovanému a regulovanému procesu vývoje softwaru. [5]

#### 4. **Zákon zachování organizační stability:**

*„Průměrná míra celkové práce na evolutivním systému je během života produktu neměnná.“* [5] (překlad autora)

Navzdory různým manažerským strategiím a vnějším tlakům má celková míra práce na vývoji softwaru tendenci zůstat v čase stabilní. Toto neměnné tempo práce naznačuje, že finální výstup je méně ovlivňován samotnými rozhodnutími managementu a více souhrou okolností, jako je dostupnost kvalifikovaných vývojarů, vnější zpětná vazba, komplexnost softwaru atd., které společně řídí produktivitu vývojového týmu. [5]

#### 5. **Zákon zachování obeznámenosti:**

*„Během aktivního života evolutivního programu je obsah postupných vydání statisticky neměnný.“* [5] (překlad autora)

Vydáním v tomto případě rozumíme vydanou verzi. Obeznámení s cíly softwaru je mezi členy vývojového týmu důležité. Významné změny nebo doplňky v jedné verzi mohou tým zahltit ztížením pochopitelnosti požadavků. Pokrok a kvalitu vývoje tak ovlivňuje i to, jak rychle a důkladně se tým dokáže učit a přizpůsobovat novým informacím. Vztah mezi velikostí změn a schopností týmu je zvládnout není lineární. Překročení určitého prahu velikosti změn může vyvolat změny v chování týmu. Proto je obsah vydání konzistentní – záleží na schopnosti týmu se učit a přizpůsobovat. [5]

#### 6. **Zákon neustálého růstu:**

*„Funkční obsah programu musí být neustále zvyšován, aby byla udržena uživatelská spokojenost po celou dobu jeho životnosti.“* [5] (překlad autora)

Pro udržení spokojenosti uživatelů se musí funkční obsah softwarového programu neustále rozšiřovat. Tento růst není pouze reakcí na zpětnou vazbu od uživatelů, ale také základním požadavkem na přizpůsobení se změnám v provozním prostředí a základním předpokladům, které mohou časem pozbýt platnosti. [5]

#### 7. **Zákon klesající kvality:**

*„Programy typu E budou vnímány jako s klesající kvalitou, pokud nebudou důsledně udržovány a přizpůsobeny měnícímu se provoznímu prostředí.“* [5] (překlad autora)

Vnímaná kvalita softwaru typu E klesá, pokud není důsledně udržována a přizpůsobována novým výzvám v provozním prostředí. Tento pokles způsobuje zvětšující se mezera mezi neustále se vyvíjejícím komplexním prostředím a softwarem, který do něj byl v nějakém časovém bodě zasazen. S postupem času mohou některé skutečnosti, na kterých byl software navržen, pozbýt platnosti. [5]

## 8. Zákon systému zpětné vazby:

*„Procesy programování typu E tvoří víceúrovňové systémy s více smyčkami zpětné vazby a musí být jako takové zacházeno, aby mohly být úspěšně modifikovány nebo vylepšeny.“ [5] (překlad autora)*

Vývoj softwaru typu E je složitý proces řízený zpětnou vazbou, která zahrnuje více smyček a úrovní. Pochopení a řízení zpětné vazby je pro efektivní vývoj softwarových systémů klíčové, a tím je rozpoznání a začlenění těchto mechanismů zpětné vazby do procesu vývoje nutné. [5]

## Technický dluh

Pojem technický dluh označuje kompromisy, které se dělají během vývoje softwaru za účelem urychlení vývoje nebo úspory nákladů mimo osvědčené postupy, což vede k vytvoření kódu, který je sice rychlejší napsat, ale z dlouhodobého hlediska může být problematický. Technický dluh je podobný finančnímu dluhu v tom, že se na něj vztahují „úroky“ v podobě zvýšeného úsilí při následné údržbě. Na rozdíl od finančního dluhu však technický dluh nemusí vždy vyžadovat úplné vyřešení. Důležité je, že rozhodnutí o vyřešení technického dluhu musí být strategické, aby umožnilo rychlejší vývoj nebo úsporu nákladů a ne naopak. Technický dluh totiž může být v určitých případech přehlížen, například když je software vyvíjen pouze pro krátkodobé použití a nevyžaduje dlouhodobou údržbu nebo další vývoj. [4]

## Refaktoring

V kontextu této práce je pochopení problematiky refaktoringu důležité. Jedná se totiž o hlavní způsob eliminace technického dluhu [6]. Pojmy refaktoring a refaktORIZOVAT se ve vývoji softwaru často používají příliš obecně a proto je pro jejich přesné použití nutné popsat jejich přesnou definici. V této práci je přijata definice refaktoringu, jak ji popsal Martin Fowler ve své knize [7]. Tato kniha zavádí definice dvě: pojmu refaktoring, jako podstatného jména, a pojmu refaktORIZOVAT, jako slovesa:

**Refaktoring** označuje úpravu vnitřní struktury softwaru, jejíž cílem je zvýšit srozumitelnost a škálovatelnost, aniž by se změnila vnější funkčnost softwaru.

**RefaktORIZOVAT** znamená provádět metodickou restrukturalizaci softwaru prostřednictvím refaktoringů, se zachováním jeho stávající funkčnosti.

RefaktORIZOVÁNÍ je podobné optimalizaci výkonu, protože v obou případech se jedná o úpravy kódu, aniž by se měnila celková funkčnost programu. Jejich cíle se však liší. Refaktoring má za cíl zlepšit čitelnost kódu a snížit náklady na úpravy, což může mít pozitivní nebo negativní vliv na výkon, zatímco optimalizace se zaměřuje výhradně na zvýšení výkonu, někdy na úkor přehlednosti a udržitelnosti kódu. [7]

### Zachování funkčnosti

Nutnou vlastností refaktoringu je zachování vnější pozorovatelné funkčnosti. To znamená, že si software po refaktORIZACI zachovává svůj výstup, i když se jeho vnitřní funkčnost může změnit. Může dojít ke změnám vnitřních mechanismů, ale tyto změny nesmí mít dopad na uživatelské prostředí nebo základní funkčnost softwaru. [7]

### Zlepšení návrhu softwaru

Bez pravidelného refaktORIZOVÁNÍ je návrh softwaru ve fázi údržby náchylný ke zvyšování complexity a snižování škálovatelnosti, protože se hromadí změny, které jsou občas prováděné bez ohledu na architekturu systému. Takové zanedbávání vede ke vzniku nepřehledného kódu, ve kterém je stále obtížnější se orientovat a aktualizovat jej. Příkladem úspory nákladů na budoucí údržbu lepším návrhem softwaru je eliminace nadbytečného kódu, který zapříčiňuje komplikace, jako je nutnost opakovat stejnou úpravu na několika místech. [7]

## 1.3. Zápachy v kódu

Tato kapitola se zabývá zápachy v kódu a jejich podobami, identifikací a významem v kontextu využití generativní umělé inteligence pro údržbu kódu. Pojem „zápach v kódu“ zavedl nejprve Kent Beck a později popularizoval Martin Fowler ve své knize „Refactoring“ [7]. Označuje určité struktury v kódu, které sice nejsou vyloženě chybné, ale naznačují, že v kódu se vyskytují hlubší problémy, které mohou bránit udržitelnosti, čitelnosti a škálovatelnosti [7]. Zápachy v kódu proto přispívají k technickému dluhu [6].

Zápachy v kódu hrají v softwarové evoluci důležitou roli, protože s jejich pomocí lze redukovat technický dluh nebo vylepšovat s ním spojené vlastnosti softwaru, mezi něž patří srozumitelnost, měnitelnost, rozšiřitelnost, znovupoužitelnost, testovatelnost a spolehlivost. Akumulace technického dluhu v podobě zápachů v kódu tyto vlastnosti zhoršuje. [6]

Srozumitelnost kódu je důležitá například proto, že snižuje nutné vývojové úsilí a ulehčuje adaptaci nových členů týmu, kteří se musí v kódu zorientovat. Měnitelnost a rozšiřitelnost zase v případě zhoršení způsobují čím dál tím větší nutné úsilí pro stejně velké změny. Špatná znovupoužitelnost způsobuje, že je čím dál tím obtížnější využít již existující komponenty v jiných částech systému a je třeba je refaktORIZOVAT, jinak dojde k dalšímu zvyšování technického dluhu. Spolehlivost a testovatelnost jsou nezbytné pro zajištění konzistentního výkonu softwaru za očekávaných podmínek a pro efektivní ověření jeho funkčnosti. Snížení spolehlivosti a testovatelnosti činí stále obtížnějším pro týmy zajišťující kvalitu psát testy, což může vést k situacím, kdy chyby nejsou zachyceny již u vývojového týmu, a jejich odhalení může poškodit důvěryhodnost softwaru a samotného týmu. [6]

Rozpoznání a náprava zápachů může výrazně snížit technický dluh a připravit základy pro robustnější a udržitelnější softwarový produkt [6]. Pokud se provede v rané fázi vývojového



procesu, může ušetřit značné množství času a zdrojů, avšak často není možné vzniku zápahů zcela zabránit, ať už z důvodu nedostatku času nebo znalostí [8].

## Typy zápahů v kódu

V knize [7] je popsáno celkem 24 zápahů v kódu a s nimi spojené refaktorizační techniky. V této sekci je stručně popsáno 13 těchto zápahů pro představu o tom, jaké problémy mohou v kódu vzniknout a být považovány za zápachy:

1. **Záhadná jména:** Pokud jsou prvky kódu, jako funkce, proměnné nebo třídy, pojmenovány nejasně, nutí to vývojáře neseznámené s kódem vynakládat zbytečné úsilí na rozluštění účelu kódu, což je při vývoji samozřejmě nežádoucí. Výstižné pojmenování usnadňuje pochopení a používání komponent kódu, čímž se zvyšuje udržitelnost. Pokud máme při vývoji problém s pojmenováním nějaké komponenty, naznačuje to, že náš návrh je pravděpodobně nedomyšlený nebo nejasný. [7]
2. **Duplicitní kód:** Duplicitní kód se týká situace, kdy se stejná struktura kódu vyskytuje na více místech programu. Taková redundance komplikuje údržbu, protože když potřebujeme udělat nad strukturou nějakou úpravu, každou z nich je třeba jednotlivě upravit a ověřit. [7]
3. **Dlouhá funkce:** Tento zápach se pozná především podle délky funkce, která je taková, že komplikuje pochopení účelu funkce a je možné ji zkrátit pomocí extrakce jejích částí do samostatných funkcí. Oprava tohoto záпахu spočívá v dekompozici funkcí – transformaci velkých bloků kódu do menších funkcí pojmenovaných podle jejich účelu. [7]
4. **Dlouhý seznam parametrů:** Tento zápach je v kódu přítomen, pokud má funkce nadměrný počet parametrů, které jsou tím pádem matoucí a nepřehledné. Řešením může být například shluknutí parametrů do objektu.
5. **Globální data:** Globální data jsou data, která jsou přístupná z kterékoli části softwarového programu bez omezení jejich použití nebo úprav. Tento neomezený přístup znamená, že ke změnám globálních dat může dojít v jakémkoli segmentu kódu bez dohledatelného záznamu o tom, co a kým bylo změněno, což vede k nepředvídatelným chybám. Mezi běžné formy globálních dat patří globální proměnné, proměnné tříd a třídy jedináčci. Pro zmírnění rizik spojených s globálními daty je nejlepším postupem zapouzdřit proměnnou a omezit její rozsah na oblast třídy nebo modulu, aby se omezila její viditelnost a modifikace na kontrolovanější prostředí.
6. **Mutabilní data:** Mutabilní data znamenají volnost v programování, kdy lze proměnné měnit a upravovat jak je libo. To ale může způsobit nepředvídatelné chování a chyby, které se mohou vyskytnout vzácně, což ztěžuje jejich detekci. S těmito riziky se vypořádala podmnožina programování zvaná funkcionální programování, která prosazuje neměnnost, kdy se datové struktury neaktualizují, ale místo toho se vytvářejí nové kopie s případnými změnami. Tím pádem zůstávají původní data zachována. I když pracujeme s programova-

cím jazykem, který funkcionální není, můžeme využít výhody principů imutability a tím minimalizovat vedlejší efekty úprav proměnných.

7. **Divergentní změny:** Divergentní změna označuje situaci, kdy modul vyžaduje úpravy z různých důvodů, které s ním až tolik nesouvisí [7]. Pokud si například říkáte: „*Tyto tři funkce budu muset změnit pokaždé, když budu mít novou databázi. Tyto čtyři funkce budu muset změnit pokaždé, když se objeví nový finanční nástroj.*“ [7] (překlad autora), je pravděpodobné, že kód obsahuje zápach divergentní změny. V ideálním případě by se pro zjednodušení vývoje a údržby měly tyto oddělené záležitosti izolovat do samostatných modulů. Díky takové organizaci kódu se pak mohou vývojáři soustředit na relevantní změny specifické pro jeden kontext bez nutných zásahů do nesouvisejících funkcí. [7]
8. **Operace brokovnicí:** Tento zápach kódu se objevuje, když jedna změna vyžaduje více malých úprav (jako vystřelené broky z brokovnice) v různých třídách nebo částích kódu, což způsobuje že je místa nutná upravit snadné přehlédnout. Řešením tohoto problému je centralizace související funkčnosti na jedno místo. [7]
9. **Závistivá funkce:** Situace, kdy funkce v jedné části softwaru nadměrně interaguje s daty nebo funkcemi jiného modulu, nikoli s daty, ke kterým přirozeně patří. Pro tento zápach je typické, že funkce často využívá k provádění svých operací metody přístupu k datům externího modulu, což naznačuje, že její logika je těsněji spjata s daty právě toho modulu než s jejím vlastním modulem. Vhodná refaktorizační strategie zahrnuje přemístění funkce do modulu, kam logicky nejvíce zapadá. [7]
10. **Shluky dat:** Shluky dat jsou skupiny několika datových položek, které se často vyskytují společně ve více částech softwarového systému, například v několika polích tříd nebo parametrech metod. Tento společný výskyt naznačuje že existuje potenciál na vylepšení v zapouzdření těchto dat do samostatného objektu. [7]
11. **Posedlost jednoduchostí:** Tento zápach se týká nadměrného používání primitivních datových typů, jako jsou celá čísla, čísla s pohyblivou řádovou čárkou a řetězce, namísto vytváření vlastních typů, které jsou v kontextu domény smysluplnější. Tento zápach vzniká, když vývojáři zpracovávají data specifická pro nějaký problém – například peněžní hodnoty, fyzikální měření, souřadnice nebo telefonní čísla pouze pomocí těchto základních typů. Například fyzikální výpočty mohou být prováděny s obyčejnými celými čísly nebo plovoucími body, čímž se ignoruje práce s jednotkami (například záměna milimetrů za palce). Podobně jsou řetězce často nesprávně používány k reprezentaci strukturovaných dat, jako jsou telefonní čísla, kterým může prospět existence vyhrazeného typu, který zapouzdřuje logiku ověřování a zobrazování. [7]
12. **Opakované switche:** Tento zápach je typický pro objektově orientované programování, kdy se stejná logika prostřednictvím příkazů switch nebo if/else používá nadbytečně. Pokud jsou totiž tyto switch bloky stejné na více místech, vyžaduje jakákoli změna nebo přidání možnosti aktualizaci v každé instanci, kde switch existuje. Proti tomuto neefektivnímu opakování slouží použití polymorfismu jako elegantnějšího a udržitelnějšího řešení. [7]

13. **Cykly:** Zápach kódu související s používáním cyklů přišel s posunem v programovacích paradigmatech směrem k modernějším a funkčním stylům. Historicky byly cykly ve většině programovacích jazyků nezbytné kvůli nedostatku alternativ. Dnes lze však tradiční cyklové struktury často nahradit stručnějšími a jasnějšími operacemi jako je filter nebo map. Tyto operace jasně zobrazují zpracovávaná data a co se s nimi děje. [7]

## Identifikace zápachů v kódu

Zápachy v kódu se nemusí okamžitě projevit jako chyby, ale mohou způsobit překážky v dalším vývoji. Beck a Fowler přirovnávají identifikaci zápachů v kódu k intuitivnímu procesu podobnému rozpoznávání nepříjemných pachů a vysvětlují, že jejich identifikace je subjektivní činností, která často závisí na intuici a zkušenostech vývojáře. Identifikace zápachů v kódu se nazakládá na přesných metrikách nebo pevných kritériích. Jedná se spíše o úsudek vývojáře, který dokáže rozpoznat vzory, jež se odchylují od osvědčených postupů. [7]

## 1.4. Velké jazykové modely

Velké jazykové modely (Large Language Models, LLM) jsou rozsáhlé neuronové sítě, učené bez učitele (tzn. ke vstupním datům není předurčen výstup), určené k vytváření souvislého a kontextově relevantního textu. Tyto modely vynikají v řadě jazykových úloh, jako je porozumění čtenému textu, strojový překlad, zodpovídání otázek nebo sumarizace textu, aniž by vyžadovaly trénink specifický pro danou úlohu. Příkladem, kterým se tato práce zabývá nejvíce, jsou modely GPT (Generative Pre-trained Transformer) od společnosti OpenAI, založené na transformátorech s miliardami parametrů naučených na obrovské sadě dat webových stránek. Modely LLM se učí z nezpracovaných textových dat, díky čemuž jsou schopné provádět různé úlohy zpracování přirozeného jazyka odvozením vzorů a příkladů přímo ze vstupních dat, čímž odpadá potřeba specifického trénování pro danou oblast. [9]

## Historie

Motivací k vytvoření jazykových modelů byla potřeba vyvinout obecnější systémy schopné provádět širokou škálu úloh různého charakteru bez pracného procesu ručního vytváření a anotování trénovacích datových sad pro každou z takových úloh. Donedávna převládající přístup spočívající v trénování modelů na jeden typ úlohy pomocí datových sad specifických pro tuto úlohu byl nedostatečně škálovatelný. Pokroky ve vývoji poté ukázaly, že jazykové modely jsou schopné plnit úlohy v zero-shot nastavení (tedy bez nutnosti trénovat model na konkrétní úloze), což naznačovalo jejich potenciál pro bezproblémové zvládnutí široké škály úloh. [10]

### Technický základ

K alespoň základnímu pochopení fungování velkých jazykových modelů je často popisováno několik vrstev AI [11]. A to konkrétně strojové učení, hluboké učení a nakonec samotné jazykové modely.

#### Strojové učení

Strojové učení se zaměřuje na využití dat a algoritmů k napodobování způsobu, kterým se učí lidé. Fungování algoritmu strojového učení se popisuje třemi částmi. První z nich je rozhodovací proces, v němž algoritmus provádí předpověď nebo klasifikaci na základě vstupních dat, která mohou být buď označená nebo neoznačená. Pojmy data označená nebo neoznačená jsou vysvětleny v dalším odstavci. Výsledkem tohoto procesu je odhad, který nalézá vzory v datech. Další je chybová funkce, která vyhodnocuje předpovědi modelu porovnáním se známými příklady, čímž vyhodnocuje přesnost modelu. Poslední je proces optimalizace modelu, který model upravuje tak, aby lépe odpovídal datovým bodům v trénovací množině. Tento proces pokračuje dokud algoritmus nedosáhne předem požadované přesnosti. [12]

Označená a neoznačená data jsou pro strojové učení nepostradatelná a liší se především svým použitím a vlastnostmi. Označená data obsahují značky a je obtížnější je získat kvůli pracnému procesu označování, který je většinou prováděn lidmi. Trénování modelu pomocí označených dat se nazývá učení pod dohledem. Naproti tomu neoznačená data, která se používají při učení bez dohledu žádné označování neobsahují. Označená data vedou modely při učení specifických vstupně-výstupních vztahů (značka souvisí s výstupem nebo ho určuje), zatímco neoznačená data pomáhají odhalit skryté vzory a struktury. [13]

#### Hluboké učení

Hluboké učení a strojové učení se zásadně liší ve svých přístupech k procesu učení. Hluboké učení může využívat označené datové soubory k trénování algoritmů, ale také dokáže efektivně pracovat s nestrukturovanými neoznačenými daty, jako jsou texty nebo obrázky, v jejich surové podobě. Toto umožňuje hlubokému učení samostatně identifikovat a extrahovat relevantní rysy z dat, což výrazně snižuje potřebu lidského zásahu a zvyšuje schopnost zpracovávat velké objemy dat. [12]

Hluboké učení, podmnožina neuronových sítí, se snaží napodobit funkce lidského mozku. Neuronové sítě se skládají ze vzájemně propojených vrstev „neuronů“. Každý vstupní signál prochází všemi vrstvami, což připomínajících několik vrstev lineární regrese, které jsou spojením vrstev rozšířené o nelinearity, které umožňují modelovat vysoce nelineární vztahy. [11][12]

## Jazykové modely

Základní koncept v jazykovém modelování je generování následujícího slova na základě předcházející posloupnosti slov. Tato úloha se dá transformovat do úlohy strojového učení, kde vstupem neuronové sítě je posloupnost slov a jejím výstupem je následující slovo. To z úlohy dělá klasifikační problém s počtem tříd rovným počtu možných slov. Modely tuto úlohu zvládají využitím rozsáhlých textových dat z různých zdrojů, jako jsou knihy, články a online obsah. Takový typ textu činí proces učení učním samokontrolované, protože všechna slova slouží jako označení dat pro posloupnost předchozích slov. Dostatečně rozsáhlá neuronová síť, trénovaná na takových datech, se naučí předpovídat kontextově vhodných slov. Vygenerovaná textová sekvence se pak iterativně rozšiřuje předpovídaním jednoho slova po druhém, což umožňuje vytvářet souvislý a kontextově relevantní text. [11]

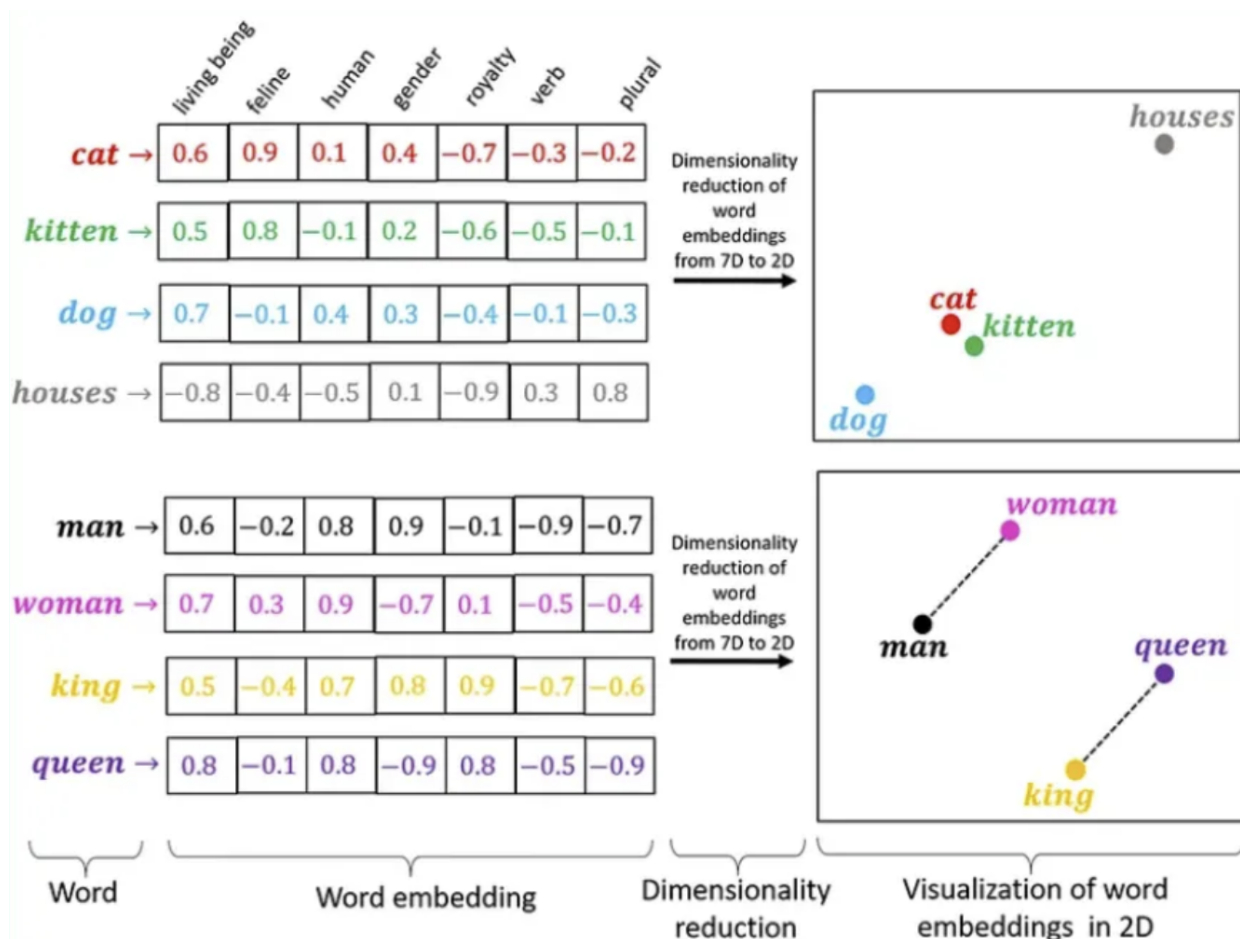
## Vnoření slov (word embeddings)

K tomu, abychom mohli pracovat se slovy strojově, je nutné je nějakým způsobem převést na čísla. V jazykovém modelování ale nestačí pouhé přiřazení číselných hodnot slovům nebo písmenům, jelikož potřebujeme znát i vztahy mezi slovy. K tomu slouží vnoření slov – způsob převádění slov na vektory reálných čísel. Hlavní výhodou vnoření slov je, že významově podobná slova se v prostoru nacházejí blízko sebe. Zároveň jsou i zachovány typy vztahů mezi slovy, jako je například vztah pohlaví mezi slovy král a královna nebo muž a žena, jak je zjednodušeně znázorněno na obrázku 1.1. [14]

Tradiční vnoření slov, jako jako například model Word2Vec, jsou statické. Každé slovo má pevnou reprezentaci. To může být omezující v případech, kdy se význam slova mění v závislosti na kontextu. Například anglické slovo „bank“ může označovat banku ale také břeh řeky v závislosti na kontextu. Aby se tento problém vyřešil, bylo vyvinuto kontextualizované vnoření slov, prováděné například modelem ELMo. Toto kontextualizované vnoření slov se v modelech GPT realizuje prostřednictvím modelu Transformer. [15]

## Trénování velkých jazykových modelů

Důležitou prací, zabývající se novým přístupem k trénování jazykových modelů se schopnostmi takovými, jaké dnes známe, je práce [16], která představila model Transformer. Narozdíl od tradičních modelů, které se spoléhají na rekurentní nebo konvoluční neuronové sítě, je Transformer založen výhradně na mechanismech pozornosti. Tento posun umožnil výrazně lepší paralelizaci, zkrátil dobu potřebnou k trénování a zvýšil schopnost modelu naučit se určité druhy spojitostí v datech. Dá se říci, že umožňuje modelu dávat větší pozornost důležitým částem vstupních dat – podobně jako lidé. [16][11]



Obrázek 1.1.: Příklad pozic slov v prostoru. Slova označující zvířata se nachází ve shluku v porovnání se slovem „domy“. Patrný je i stejný rozdíl mezi sémanticky podobnými slovy ale s opačným pohlavím. [14]

## Předtrénování

Předtrénování, základní fáze trénování, spočívá v tom, že se do modelu vloží obrovské množství textových dat, aby bylo možné předpovědět další slovo v posloupnosti slov. Tato fáze umožňuje modelu uchopit koncepty gramatiky, syntaxe a rozsáhlou škálu znalostí o světě. Toto počáteční trénování však model převážně učí generovat souvislý text, nikoli přesně reagovat na konkrétní dotazy nebo instrukce. [11]

„Pokud se například předtrénovaného LLM zeptáte „Jaké je tvé křestní jméno?“, může odpovědět „Jaké je tvé příjmení?“ jednoduše proto, že taková data viděl během předtrénování, jako například v mnoha prázdných formulářích“ [11].

## Doladění instrukcí

K řešení uvedeného problému se provádí doladění instrukcí. V této fázi prochází předtrénovaný model dalším tréninkem pomocí dvojic instrukcí a odpovědí, čímž se naučí rozumět instrukcím a poskytovat smysluplné odpovědi. Použitá datová sada je mnohem menší ale pečlivě propracovaná. Často zahrnuje instrukce a odpovídající odpovědi vytvořené člověkem, což je náročnější na

vytvoření. Tento proces dolaďování přeúčí model od pouhého doplňování textových sekvencí k chování užitečného asistenta.

### **Učení lidskou zpětnou vazbou**

U pokročilých modelů se používá třetí fáze známá jako Reinforcement Learning from Human Feedback (RLHF). Tato fáze dále dolaďuje reakce modelu prostřednictvím opakované zpětné vazby. Dalším záměrem je penalizovat model za chyby nebo potenciálně škodlivé odpovědi [17]. RLHF má zásadní význam pro zdokonalování modelu a jeho posunutí k dovednostem na úrovni člověka nebo nad ně. [11]

### **Tokens a jejich pravděpodobnost**

Token představuje jednotku textu, která může mít délku jednoho znaku až jednoho slova. Rozhraní modelu umožňuje získat i pravděpodobnost právě vygenerovaného tokenu zvanou „logprob“. Tato pravděpodobnost udává, jak pravděpodobný je výskyt tokenu v dané sekvenci na základě předchozího textu. Tato metrika je obzvláště užitečná pro pochopení jistoty modelu v jeho předpovědi a pro vytváření aplikací, které vyžadují vysokou přesnost, jako jsou klasifikační úlohy, systémy otázek a odpovědí založené na vyhledávání a funkce automatického doplňování. [9][18]

### **Omezení velkých jazykových modelů**

Při práci s velkými jazykovými modely je třeba počítat s několika omezeními, která ovlivňují jejich funkčnost a spolehlivost. Chybí jim trvalá paměť, což v určitých případech vyžaduje externí uchovávání kontextu. Jejich stochastická povaha způsobuje lišící se výstupy i pro neměnný vstup, což vede k nekonzistenci. Modely jsou také trénované na datech získaných k nějakému datu, čímž jsou omezeny na neaktuální informace. Kromě těchto omezení mohou modely vytvářet i faktually nesprávný, ale věrohodně znějící obsah – jev známý jako „halucinace“. A v neposlední řadě, ačkoli LLM ze své podstaty fungují obecně, občas potřebují data specifická pro danou oblast, aby mohly dobře plnit specializované úkoly. Například generování návrhu informačního systému bude kvalitnější při specifickém pojmenování entit a jejich vztahů, než nahrazením názvy „entita1“, „entita2“ apod. [19]

### **Využití jazykových modelů v oblasti softwarového inženýrství**

Nástup velkých jazykových modelů, jako jsou Copilot a ChatGPT, vyvolal v komunitě softwarových inženýrů značný zájem a diskusi. Tyto výkonné generativní modely prokázaly pozoruhodné schopnosti při generování kódu a poskytly vývojářům nové nástroje pro zefektivnění a zlepšení procesu vývoje softwaru.

### Výzkum využití jazykových modelů v oblasti modelování softwaru

Zatímco potenciál LLM pro generování kódu byl široce zkoumán a diskutován, jejich použití v oblasti modelování softwaru zůstává relativně málo prozkoumán [20]. Práce [20] zkoumá možnosti a omezení ChatGPT právě v oblasti modelování softwaru. Tato studie se snaží tuto mezeru zaplnit tím, že zkoumá, jak dobře může ChatGPT modelovat a pomáhat při modelování úloh, identifikuje klíčové nedostatky a navrhuje zlepšení. Výzkum se odehrával v době před vydáním modelu GPT-4. Je důležité na tento fakt poukázat, protože vývoj v oblasti jazykových modelů je velmi rychlý a nové modely mohou, a pravděpodobně překonají, výsledky této studie. Pro účely této práce budou v této části uvedeny výsledky studií relevantní pro využití jazykových modelů v oblasti refaktorizace kódu.

Výsledky obsahují několik zjištění týkajících se možností a omezení ChatGPT v úlohách modelování softwaru. Jedním z významných zjištění je, že zatímco při generování kódu si ChatGPT vede obdivuhodně, jeho použitelnost při modelování softwaru je více omezená. Návrhy generované ChatGPT často obsahují syntaktické i sémantické chyby týkající se agregace, asociace a použití kompozic. Tyto problémy ukazují, že existuje rozdíl mezi schopnostmi ChatGPT při generování kódu a generování návrhových modelů a poukazují na problémy při využití LLM v abstraktní oblasti vývoje softwaru. [20]

Dalším důležitým zjištěním je citlivost ChatGPT na kontext a problémovou doménu úlohy. Studie zjistila, že ChatGPT je přesnější v dobře známých doménách, kde existují rozsáhlá trénovací data, jako jsou např. bankovní systémy, a vytváří sémanticky přesnější modely. V méně známých doménách tak bývají generované modely chybné. Tato vlastnost závislosti na problémové doméně omezuje použitelnost ChatGPT v různých modelovacích scénářích. Škálovatelnost je další oblastí, kde ChatGPT čelí značným výzvám. ChatGPT má potíže s efektivním generováním a úpravou větších modelů a obvykle zvládá pouze malé modely o 10-12 třídách, než se jeho výkonnost sníží. Výsledky vytvořené ChatGPT navíc vykazují vysokou variabilitu a nekonzistenci, a to i v případě, že jsou generovány stejnými výzvami. Tato skutečnost ztěžuje dosažení spolehlivých a reprodukovatelných výsledků. [20]

### 1.5. Inženýrství výzev (prompt engineering)

Tato sekce se zabývá základními i pokročilými technikami inženýrství výzev a popisuje postup při vytváření efektivních výzev pro modely GPT. Všechny příklady výstupů byly vygenerovány modelem GPT-4o, pokud není uvedeno jinak.

Inženýrství výzev je nově vznikající disciplína v oblasti generativní umělé inteligence, která se zaměřuje na vytváření textových vstupů, podle nichž se řídí výstupy generativních modelů. Zabývá se interakcí s různými druhy modelů, od generátorů obrázků až po komplexní jazykové modely. Postup při vývoji výzev je iterativní a zahrnuje prvky podobné tradičnímu softwarovému inženýrství, jako je kontrola verzí a regresní testování. [19]



## Výzva (prompt)

Výzva slouží jako textový vstup ovládající generované odpovědi modelu. Výzvy mohou být velmi různé, od jednoduchých otázek po podrobné popisy konkrétních úkolů. Obsahují instrukce, otázky, vstupní data a příklady. Instrukce a otázky jsou nezbytné pro provedení požadovaného úkolu a další prvky, jako jsou vstupní data a příklady, mohou zvýšit přesnost odpovědi. Pokročilejší výzvy mohou model nabádat i např. k „přemýšlení“ o určitém problému a jeho následném řešení. [19]

Protože jsou modely GPT, využívané v této práci, trénovány na obrovském množství textových dat, ve kterých je nejvíce zastoupen anglický jazyk, je důležité výzvy psát právě v anglickém jazyce. Zejména v těžších úlohách, vyžadujících komplexní uvažování a zdůvodňování, je rozdíl mezi anglickým jazykem a ostatními méně zastoupenými jazyky výrazný. [21]

Příklad jednoduché výzvy pro model GPT-4o může vypadat následovně (instrukce a vstupní data):

```
Refactor the following code:
function add(a, b) {
  return a + b;
}
```

## Základy tvorby výzev

Tvorba výzvy by měla být založena na několika osvědčených principech, které zlepšují a ztvárňují výstupy modelu do požadované podoby. Několik takových principů je popsáno v průvodcích inženýrstvím výzev [22] a [23]. Mezi tyto základní principy patří:

1. **Jasnost a jednoznačnost:** Čím jasnější a jednoznačnější výzva je, tím lepší výsledky lze očekávat. Instrukce by měly být hned na začátku výzvy a různá vstupní data by měla být jasně oddělena pomocí oddělovačů „###“ nebo „“““. [22]

Příklad:

```
Compare these two texts and identify the differences.

[Text 1]

###

[Text 2]
```

2. **Specifičnost a preciznost:** Výzva by měla být co nejvíce specifická, aby modelu poskytla dostatečné informace k vytvoření požadovaného výstupu. Pokud je součástí požadavků kontext, délka, formát, styl atd., měly by být tyto požadavky patřičně uvedeny. Jakékoliv

počty či popisy by měly být popsány precizně: „Napiš 5 vět“ namísto „Napiš několik vět“ [22]

3. **Instrukce co dělat, ne co nedělat:** Instrukce by měly být napsány takovým způsobem, který vyjadřuje co má model udělat, nikoli co nemá. Např. „Napiš nějaký kód. Všechnu potřebnou funkcionalitu naprogramuj samostatně.“ namísto „Napiš nějaký kód. Nepoužívej externí moduly.“ [23]

## Základní techniky tvorby výzev

Existuje několik základních technik, které lze použít při tvorbě výzev pro velké jazykové modely. Mezi ty základní patří Zero-shot a Few-shot výzvy. Jejich podrobný popis a vysvětlení je uvedeno v průvodci inženýrstvím výzev [24].

### Zero-shot výzvy

Zero-shot výzvy jsou jednou z hlavních technik používaných při práci s velkými jazykovými modely. Jejich podstata je v absenci jakýchkoliv příkladů nebo demonstrací možného výstupu. Model je přímo instruován k provedení určitého úkolu bez dalších ukázek, které by jej usměrňovaly. [24]

Příklad zero-shot výzvy může vypadat následovně:

Write a function in Python that returns the sum of two numbers.

Výstup:

```
Certainly! Here is a simple Python function that returns the sum of two
numbers:
```python
def sum_of_two_numbers(a, b):
    return a + b

# Example usage:
result = sum_of_two_numbers(5, 3)
print(result)  # Output: 8
```

You can call this function with any two numbers to get their sum.
```

V uvedené výzvě modelu nebyly poskytnuty žádné příklady kódu, který by ukazoval, jak funkce, sčítající dvě čísla, vypadá.

Pokud zero-shot přístup nepřináší požadované výstupy, je doporučeno využít jiné techniky, např. technika few-shot [24].

## Few-shot výzvy

Few-shot výzva je technika, která zlepšuje výstupy při řešení složitých úkolů tím, že je ve výzvě uvedeno několik příkladů nebo ukázek možného výstupu. Tato technika model navádí k vytváření odpovědí podobných poskytnutým příkladům. [24]

Použitá technika few-shot je pojmenovávána podle počtu příkladů, které jsou ve výzvě uvedené. V případě, že je ve výzvě uveden jeden příklad, mluvíme o one-shot výzvě, při třech příkladech 3-shot výzvě atd. [24]

Příklad 2-shot výzvy pro vytvoření Python funkce, která vypisuje a vrací součin dvou čísel:

```
A function that returns the sum of two numbers:
def sum(a, b) {
    sum = a + b
    print(sum)
    return sum;
}

A function that returns the difference of two numbers:
def difference(a, b) {
    difference = a - b
    print(difference)
    return difference;
}

Write a function that returns the product of two numbers:
```

Ve výzvě není uvedeno nic o tom, v jakém jazyce má být kód napsán, nebo že má funkce výsledek vypsat do konzole. Výstup modelu však tyto vlastnosti díky poskytnutým příkladům obsahuje:

```
Here's a function that returns the product of two numbers, following the
same format as the provided functions for sum and difference:
```python
def product(a, b):
    product = a * b
    print(product)
    return product
```

This function multiplies two numbers a and b, prints the result, and then
returns it.
```

Few-shot výzvy však mají svá omezení, zejména u složitých úloh, při kterých je potřeba více kroků nebo komplexnějšího uvažování. Pokud na úkol nestačí využití technik zero-shot ani few-shot, je

zapotřebí použít pokročilejších technik, jako jsou například myšlenkové řetězce. [24]

### Myšlenkové řetězce (Chain-of-Thought, CoT)

Výzvy podněcující myšlenkové řetězce zlepšují výstupy komplexních úloh tím, že obsahují mezikroky potřebné pro uvažování o vyřešení problému. Tuto techniku lze kombinovat s few-shot přístupem uvedením několika příkladů myšlenkových pochodů. Hlavním prvkem výzvy typu myšlenkového řetězce je nějaká forma instrukce „přemýšlej“. Příkladem může být věta „Let’s think step by step“. [24]

Rozdíl výstupu s výzvami typu myšlenkový řetězec se dá demonstrovat na aritmetických příkladech. Příklad výzvy:

```
How much is 13! - 123456789
Answer immediately without any other text than the result.
Result:
```

Výstup modelu:

```
62270208000
```

Výsledek výpočtu není správný. Přidáním myšlenkového řetězce se výstup modelu zlepší:

```
Let's think step by step:
How much is 13! - 123456789

Result:
```

Výstup modelu:

```
To solve 13! - 123456789, we first need to calculate 13!.

The factorial of 13, denoted as 13!, is the product of all positive
integers up to 13:

13! = 13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1

Calculating this step-by-step:

13! = 13 * 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
    = 13 * 12 = 156
    = 156 * 11 = 1716
    = 1716 * 10 = 17160
```

```

= 17160 * 9 = 154440
= 154440 * 8 = 1235520
= 1235520 * 7 = 8648640
= 8648640 * 6 = 51891840
= 51891840 * 5 = 259459200
= 259459200 * 4 = 1037836800
= 1037836800 * 3 = 3113510400
= 3113510400 * 2 = 6227020800
= 6227020800 * 1 = 6227020800

```

So,  $13! = 6227020800$ .

Next, we subtract 123456789 from 6227020800:

```
6227020800 - 123456789
```

Performing the subtraction:

```

6227020800
-123456789
6103564011

```

Thus, the result of  $13! - 123456789$  is:

```
6103564011
```

Výsledek, ke kterému model došel, je v tomto případě správný.

## Pokročilé techniky tvorby výzev

Kromě základních technik tvorby výzev existují i pokročilejší techniky, využívající nebo zaměřující se na vlastnosti modelů GPT, které často zahrnují více kroků nebo výběr z více výstupů. [24]

Mezi pokročilejší techniky tvorby výzev patří např. technika Self-Consistency, která využívá few-shot myšlenkových řetězců (tzn. vytvoření několika možných myšlenkových pochodů, vedoucích k řešení problému) a z více výstupů modelu vybere ten nejkonzistentnější. [24]

Další technikou je generování vědomostí (Generated Knowledge Prompting), která model využívá nejprve k vytvoření seznamu znalostí o daném problému, který je následně použit jako vstupní data výzvy k vyřešení problému. [24]

Úpravou techniky myšlenkových řetězců vzniká technika stromu myšlenek (Tree of Thoughts), která model vede k postupnému rozvíjení myšlenek a následnému uvažování o jejich správnosti. Pokud model myšlenku zavrhne, pokračuje v uvažování nad dalšími možnostmi [24]. Tuto techniku lze využít i v jedné výzvě, jak je popsáno v práci [25].

### Optimalizace výzev

Nevýhodou relativně nové disciplíny inženýrství výzev je nedostatek standardů a osvědčených postupů optimalizace výzev. Počet možných výzev je, s ohledem na maximální délku vstupu modelu, a tím pádem všech možných kombinací slov, obrovský [26]. Tato sekce se zabývá rešerší existujících metodik a postupů pro optimalizaci výzev.

#### Možné přístupy k optimalizaci výzev

Článek [27] představuje metodiku vývoje výzev nazývanou GPEI (Cíl, Výzva, Hodnocení, Iterace), strukturovanou do čtyř hlavních kroků: definování cíle, navržení výzvy, vyhodnocení odpovědi a iterace na základě výsledků. Tento systematický přístup je navržen tak, aby zvýšil přesnost a užitečnost výzev a řešil problémy jako jsou halucinace a nekvalitní odpovědi. Prvním krokem je jasné definování cíle, kterého má model dosáhnout. Tento cíl určuje strukturu výzvy a stanovuje kritéria pro vyhodnocení odpovědi. Krok navržení výzvy zahrnuje výběr a použití vhodných technik pro vytvoření výzvy, která bude modelu poskytovat potřebné informace k dosažení cíle. Krok vyhodnocení zahrnuje ruční ověření odpovědi modelu na základě kritérií stanovených v cíli. Tento krok identifikuje problémy jako je neúplnost, nepřesnosti, nerelevantnost a halucinace. Proces hodnocení může zahrnovat další výzvy v návaznosti na původní výzvu, vyžadující vysvětlení rozhodnutí modelu. Pokud odpověď nesplňuje kritéria daná cílem, je výzva upravena a proces iterován. Zdokonalování pokračuje, dokud není dosaženo adekvátní odpovědi. [27]

Článek [28] představuje sadu technik pro vylepšení interakce s modely. Zavádí koncept vzorů výzev, podobný softwarovým vzorům, které představují opakovaně použitelná řešení běžných problémů, s nimiž se při práci s modely setkáváme. Článek zdůrazňuje, že tyto vzory výzev významně obohacují možnosti modelů a jsou přizpůsobitelné v různých oblastech i mimo vývoj softwaru.

Zajímavým přístupem k optimalizaci výzev je algoritmus popsáný v práci [26], která k optimalizaci využívá samotný model v roli inženýra výzev. Podstata algoritmu je popsána v následující sekci.

#### Automatický inženýr výzev

Automatický inženýr výzev (Automatic Prompt Engineer, APE) je algoritmus, který je schopen generovat a vybírat výzvy pro velké jazykové modely. Algoritmus je založen na procesu, ve kterém model generuje výzvy, které jsou následně vyhodnoceny. V oblasti zero-shot výzev je APE schopen generovat výzvy stejně kvalitně jako člověk. [26]

Cílem algoritmu je najít instrukci, která po předložení modelu spolu se vstupními daty přiměje model k vytvoření požadovaného výstupu. Vstupem algoritmu je tedy sada dvojic, tvořených vstupem a patřičným požadovaným výstupem. Algoritmus nejprve na základě dvojic vstupů/výstupů odvozuje sadu kandidátních instrukcí, které jsou následně vyhodnoceny vybranou hodnotící funkcí, a poté vybere instrukci s nejvyšším skóre. [26]

Pro hodnocení kvality generovaných instrukcí je navrženo několik hodnoticích funkcí, které měří shodu mezi vstupními daty algoritmu a generovaným výstupem kandidátní instrukce. Jednou z hlavních je funkce logaritmické pravděpodobnosti, která měří průměr všech logaritmických pravděpodobností tokenů z výstupu modelu po předložení kandidátní instrukce. [26]





## 2. Praktická část

Praktická část se zaměřuje na vyhodnocení schopností modelu GPT-4o při identifikaci a refaktORIZACI zápachů v kódu, popisuje metodiku výzkumu, tvorbu korpusu kódů se zápachy, exploraci detekce a refaktORIZACE zápachů, optimalizaci výzev použitých pro získání výstupů modelu a nakonec získané výsledky.

### 2.1. Metodika výzkumu

V této sekci je popsána metodika výzkumu, jehož cílem je odpovědět na následující výzkumné otázky:

1. Jak je model GPT-4o přesný při detekci a identifikaci zápachů v kódu?
2. Jaký vliv má inženýrství výzev? Existuje metoda, která má lepší výsledky než ostatní?
3. Jaká je úspěšnost opravy zápachů v kódu pomocí modelu GPT-4o?

Pro realizaci tohoto výzkumu byla zvolena následující metodika:

1. Tvorba korpusu kódů se zápachy
2. Explorační fáze
3. Optimalizace výzvy
4. Posouzení refaktORIZAČNÍCH schopností

### Hodnocení refaktORIZACE

Pro účely optimalizace výzvy a samotného posouzení refaktORIZAČNÍCH schopností je zapotřebí způsob hodnocení výsledků refaktORIZACE. Protože nebyla nalezena vhodná metoda, která by dokázala posoudit přítomnost zápachů v kódu na základě měřitelných hodnot, byla zavedena hodnotící škála úspěšnosti refaktORIZACE od 1 do 5, která je výsledkem ručního hledání zápachu v kódu na základě jeho definice:

1. **Bez přítomnosti:** Zápach byl zcela odstraněn.
2. **Minimální přítomnost:** Zápach je nepatrný.
3. **Mírná přítomnost:** Zápach je přítomen, ale má malý dopad na udržitelnost.

4. **Střední přítomnost:** Zápach je patrný a má střední dopad na udržovatelnost.

5. **Silná přítomnost:** Zápach výrazně zhoršuje kvalitu a udržovatelnost kódu.

Protože je v kódech v korpusu složité izolovat jediný zápach, hodnotící škála je zaměřena na hlavní zápach, pro který byl kód vytvořen. Pokud tento zápach nelze podle své definice v kódu nalézt, je hodnocení refaktorizace 1. Zbýlé hodnoty škály poté připadají na situace, kdy byl zápach odstraněn pouze částečně, a nebo byl při refaktorizaci vytvořen jiný, dříve nepřítomný zápach.

## 2.2. Tvorba korpusu kódů se zápachy

Vytvoření korpusu kódů se zápachy je hlavním krokem při posuzování refaktorizačních schopností velkých jazykových modelů. Tato sekce popisuje kroky provedené při vytváření tohoto korpusu kódů v jazyce JavaScript, z nichž každý vykazuje alespoň jeden specifický zápach kódu. Korpus v této práci slouží jako primární soubor dat pro vyhodnocení účinnosti modelu GPT-4o při identifikaci a refaktorizaci zápachů.

### Popis korpusu

Korpus kódů se zápachy v této práci je soubor kódů v jazyce JavaScript, z nichž každý vykazuje alespoň jeden specifický zápach kódu, podle kterého je pojmenován. Každý z těchto kódů implementuje sadu funkcí typickou pro e-shopy nebo bankovníctví a obsahuje třídu API, která definuje hlavní případy užití kódu a slouží jako rozhraní pro testování. Dále se v korpusu kódů nachází pro každý kód soubor s testy, které ověřují jeho vnější chování, právě skrze třídu API. Korpus je rozdělen na složky, pojmenované podle zápachů, které obsahují soubory kódu a testů.

### Postup při tvorbě korpusu

Pro tvorbu korpusu byla zvolena problémová doména e-shop, zejména kvůli svému rozšíření na internetu a vhodnosti pro demonstraci různých zápachů v kódu, protože obsahuje mnoho různých funkcí a vztahů mezi entitami (např. produkty, objednávky, zákazníci, platby), a problémová doména bankovníctví, kvůli své zajímavosti v oblasti modelování softwaru za pomoci jazykových modelů.

Pro vytvoření korpusu zápachů v kódu byly provedeny následující kroky:

1. **Generování kódů pomocí GPT-4:** Počáteční verze kódů byly vygenerovány pomocí modelu GPT-4. Každý skript je pojmenován podle zápachu, který obsahuje. Při tomto kroku nastává riziko, že GPT-4 vygeneruje kód, který je modelu dobře známý, a tím by mohl zkreslit výsledky identifikace zápachů. Pro minimalizaci tohoto rizika byl proveden následující krok.
2. **Ruční úprava:** Každý vygenerovaný skript byl ručně upraven a zkontrolován. Tento krok měl za cíl zajistit, že skripty nejsou identické s výstupy generovanými GPT-4.

3. **Ověření správnosti:** U každého skriptu byla ověřena přítomnost požadovaného zápachu, a v případě potřeby byly provedeny další úpravy.
4. **Vytvoření rozhraní:** Pro každý skript bylo vytvořeno rozhraní ve formě třídy API, které definuje jeho hlavní případy užití. Tento krok je nezbytný pro některé typy zápachů, jako je např. „Alternativní třídy s různými rozhraními“, protože refaktORIZACE takového zápachu změnila rozhraní alespoň jedné ze tříd, a tím pádem selže testování, které očekává původní rozhraní. Nová třída API proto dává možnost při refaktORIZACI upravit i způsob, jakým jsou objekty nebo funkce používány, bez zásahu do testů.
5. **Vytvoření testů:** Pro každý skript byly vytvořeny testy, které ověřují jeho vnější chování v podobě hlavních případů užití. Tyto testy jsou nástrojem pro rychlé a spolehlivé ověření hlavního znaku refaktORIZACE, uvedeného i v definici, a tedy že proces refaktORIZACE nemění externí chování kódu.

## 2.3. Explorační fáze

V této sekci je popsána druhá fáze výzkumu, která se zaměřuje na průzkum možností modelu GPT-4o při detekci a opravě zápachů v kódu. Jejím cílem je získání představy o silných a slabých stránkách modelu, které budou následně využity k optimalizaci výzev. V této fázi není použit žádný systematický postup ale spíše zkoušení a ověřování úvah autora, založených na jeho předchozích zkušenostech a znalostech v oblasti využívání jazykových modelů v softwarovém inženýrství. Výsledkem této fáze je soubor poznatků, které jsou využity k dalšímu výzkumu.

Pro účely této fáze byly vybrány takové tři zápachy kódu, které se týkají různých konstrukcí v kódu, a tedy pokrývají různé typy refaktORIZACE. Pro třídy je to zápach „Alternativní třídy s různými rozhraními“, pro funkce zápach „Dlouhá funkce“ a pro proměnné nebo vlastnosti zápach „Shluky dat“.

### Explorace detekce zápachů

Prvním krokem v této fázi bylo ověření schopnosti modelu GPT-4o detekovat zápachy v kódu. Pro každý zápach byl modelu předložen kód, který obsahuje daný zápach, a bylo zjištěno, zda ho model správně detekuje a identifikuje. Pro lepší výsledky byly všechny výzvy provedeny několikrát.

Zde je uveden příklad výzvy a výstupu detekovaných zápachů v kódu obsahujícím zápach „Dlouhá funkce“. Výzva je typu zero-shot pro zjištění k jakému chování má model tendenci.

#### Výzva:

Analyze the following JavaScript code and identify any code smells it contains (it might contain none).

JavaScript Code:

[vložený kód]

Output: A list of code smells present in the code.

### Výstup (jen důležité části):

1. Magic Strings and Numbers:  
Using hardcoded values for `orderDetails.paymentMethod` and `orderDetails.shipmentMethod`:
  - `'CreditCard', 'PayPal', 'Standard', 'Express'`
  - Better to use constants or an enumeration for these values.
2. Inconsistent Error Handling
3. Primitive Obsession
4. Long Method
  - The `processOrder` method is doing too many things. It can be broken down into smaller methods for better readability and maintainability.
5. Boolean Trap
6. Global Dependency
7. Duplicated Code
8. Hidden Dependencies
9. Side Effects
10. No Validation of `orderDetails` Structure
11. Inconsistent API Design
12. Callback Pattern

**Poznatek 1:** Model je schopen identifikovat zápachy v kódu správně i špatně.

V případě zápalu „Dlouhá funkce“ výstup pokaždé obsahoval zápal „Long Method“ neboli „Dlouhá funkce“. V případě zápalu „Shluky dat“ výstup požadovaný zápal neobsahoval ani jednou.

**Poznatek 2:** Model detekuje i odlišné typy zápalů, než jsou popsány v knize [7].

Z výsledku je také patrné, že model detekoval různé typy zápalů. Ke každému z nich vygeneroval i krátký popis, dle kterého byly naprostá většina detekována správně. Omezení výběru zápalů ve výzvě na případy popsané v knize [7] by v případě detekce zápalů mohlo být výhodné, protože by se model mohl zaměřit na konkrétní typy zápalů, ale v případě refaktorisace by mohl být tento přístup nevýhodný, protože by modelu mohl zúžit všechny možné způsoby, jak daný zápal v kódu odstranit.

**Poznatek 3:** Model detekuje zápachy, které nejsou v kódu přítomné, ale v navrženém refaktorisovaném kódu je refaktorisuje správně.

Pro zápachy „Alternativní třídy s různými rozhraními“ a „Shluky dat“ byl výsledek detekce odlišný. Model znovu správně detekoval několik zápalů, ale tentokrát neidentifikoval požadovaný zápal.

Místo něj byl detekován zápach „Duplicitní kód“, který je sice podobný, ale neodpovídá požadovanému zápachu. Nicméně model na konci výstupu navrhnul podobu refaktorizovaného kódu, který zápach „Alternativní třídy s různými rozhraními“ opravuje. To naznačuje, že model může být schopnější zápach opravit, než ho správně identifikovat. Zároveň je na základě tohoto poznatku možné, v případě tvoření výzev metodou myšlenkových řetězců, doporučit omezování snahy modelu zápachy v myšlenkovém řetězci identifikovat, protože by mohlo dojít k vadné identifikaci a následné refaktorizaci zápachu, který v kódu není přítomen, nebo vynechání refaktorizace zápachu, který v kódu je.

## Explorace refaktorizace zápachů

**Poznatek 4:** Model může v případě využití myšlenkových řetězců ignorovat zápach.

Prvním pokusem byla výzva pro refaktorizaci s technikou myšlenkových řetězců. Zde je uveden příklad výzvy a výstupu refaktorizace zápachu „Dlouhá třída“:

### Výzva:

You are provided with JavaScript code that includes functionality with potential code smells, an API class, and a test file. Your task is to refactor the code to eliminate any code smells while ensuring the code still passes all provided tests. Follow these steps to complete the refactoring:

1. Identify and address code smells
2. Maintain functionality: Ensure that the refactored code retains the original functionality. The refactoring should keep the expected outputs of the API class.
3. Ensure test coverage: Ensure that the refactored code passes all the tests provided in the test file.

Here is the original code:

[vložený kód]

Here is the test file:

[vložený soubor testů]

Tento přístup (myšlenkové řetězce) přispěl k předchozí úvaze v poznatku 3, a tedy možnost, že model bude schopnější zápach opravit, než ho správně identifikovat. Výstup refaktorizace s technikou myšlenkových řetězců neobsahoval v žádné z „myšlenek“ požadovaný zápach, a tím nedošlo k jeho opravě. Naopak při použití výzvy bez myšlenkových řetězců byl zápach správným způsobem odstraněn. Zde je uveden příklad tohoto jevu.

### Důležitá část výstupu s myšlenkovými řetězci:

To refactor the provided code and eliminate potential code smells while ensuring the functionality is maintained and all tests pass, follow these steps:

Identify and Address Code Smells:

Long methods: Break down long methods into smaller, more focused methods.

Magic strings: Replace hardcoded strings with constants.

Duplicated code: Consolidate duplicate code.

Inefficient searches: Optimize search operations within methods.

Maintain Functionality:

Ensure that all methods still perform their expected tasks.

The refactored code should pass all provided tests without any modifications.

Here's the refactored code:

Refactored EShop Class

```
class EShop {
    constructor() {
        this.products = [
            { id: 1, name: 'Laptop', price: 1000, stock: 10 },
            { id: 2, name: 'Smartphone', price: 700, stock: 15 },
            { id: 3, name: 'Tablet', price: 500, stock: 20 },
        ];
        this.cart = [];
        this.user = { loggedIn: false, name: '', email: '' };
    }

    loginUser(email, password) {
        // Simulate user login with static credentials
        if (this.isValidLogin(email, password)) {
            this.user = { loggedIn: true, name: 'John Doe', email: email };
        } else {
            throw new Error('Login failed. Please check your credentials.');
```

```

    this.user = { loggedIn: false, name: '', email: '' };
}

```

Oproti tomuto výstupu měl výstup bez myšlenkových řetězců správně opravený zápach definováním nové třídy User, která obsahuje metody pro přihlášení a odhlášení.

#### Důležitá část výstupu bez myšlenkových řetězců:

Let's refactor the `EShop` and `API` classes to eliminate code smells and maintain the functionality.

Here's the refactored code:

```

class User {
    constructor() {
        this.loggedIn = false;
        this.name = '';
        this.email = '';
    }

    login(email, password) {
        if (email === 'user@example.com' && password === 'password123') {
            this.loggedIn = true;
            this.name = 'John Doe';
            this.email = email;
        } else {
            throw new Error('Login failed. Please check your
            credentials.');
```

```
        }
    }

    logout() {
        this.loggedIn = false;
        this.name = '';
        this.email = '';
    }
}

class EShop {
    constructor() {
        this.products = [
            new Product(1, 'Laptop', 1000, 10),
            new Product(2, 'Smartphone', 700, 15),

```

```
        new Product(3, 'Tablet', 500, 20),  
    ];  
    this.cart = [];  
    this.user = new User();  
}
```

**Poznatek 5:** Generovaný kód splní většinu předem definovaných testů, ale občas nezachová vnější chování kódu.

Výsledky testů byly v naprosté většině pozitivní, což znamená, že základní znak refaktorisace, tedy zachování vnějšího chování kódu, byl splněn. V několika případech byly výsledky testů negativní i přes správnou refaktorisaci zápachu tím, že proces změnil vnější chování kódu, což je pro automatizovanou refaktorisaci nepřijatelné.

### Shrnutí poznatků z explorace

Z výsledků explorace plyne několik důležitých poznatků. Nelze je však považovat za potvrzené, protože byly získány na malém vzorku zápachů a pokusů. Poznátky jsou následující:

1. Model je schopen identifikovat zápachy v kódu správně i špatně.
2. Model detekuje i odlišné typy zápachů, než jsou popsány v knize [7].
3. Model detekuje zápachy, které nejsou v kódu přítomné, ale v navrženém refaktorovaném kódu je refaktorisuje správně.
4. Model může v případě využití myšlenkových řetězců ignorovat zápach.
5. Generovaný kód splní většinu předem definovaných testů, ale občas nezachová vnější chování kódu.

## 2.4. Optimalizace výzvy

V této sekci je popsána třetí fáze výzkumu, jejímž cílem je vytvoření výzvy, která bude použita k posouzení refaktorisacích schopností modelu GPT-4o. Prvním krokem je získání nejdůležitější části výzvy, a tedy instrukce, která povede model k vytvoření požadovaného výstupu. Dále je na základě poznatků z explorační fáze třeba zvolit vhodnou techniku tvorby výzvy a vstupní data a výslednou výzvu sestavit.

### Zvolená instrukce

Instrukce pro vytvoření výzvy byla zvolena s pomocí algoritmu APE (Automatic Prompt Engineer). Zdrojový kód algoritmu musel být nejprve upraven tak, aby vyhovoval novému rozhraní modelů od



OpenAI. Algoritmu byly předloženy tři dvojice vstupů/výstupů (kódy se zápachy a testy/opravené kódy). Z důvodu komplexnějšího úkolu, který refaktORIZACE kódu představuje, oproti úkolům, na kterých je algoritmus měřený, byla instrukce zvolena na základě pozorování opakovaných vlastností instrukcí nejvýše hodnocených. Téměř všechny tyto instrukce začínaly slovy „Refactor the code“ a obsahovaly fráze „improve readability“ a „improve maintainability“. Na základě těchto pozorování byla zvolena instrukce:

Refactor the code to improve readability and maintainability.

## Iterační vývoj výzvy

Pro vývoj výzvy byla využita metoda GPEI (Cíl, Výzva, Hodnocení, Iterace). Výzva byla vždy složena ze zvolené instrukce s obměnami použité techniky vytváření výzev a použitých vstupních dat (kódu se zápachem a jeho testů). Pro stručnost je zavedeno základní hodnocení, které je součástí cíle každé iterace. Skládá se z hodnocení refaktORIZACE a počtu splněných testů. Základní hodnocení je považováno za pozitivní, pokud je hodnocení refaktORIZACE 1 nebo 2, a všechny testy jsou splněny. Každá iterace má pak své vlastní rozšiřující hodnocení, které závisí na cíli dané iterace.

### První iterace:

1. **Cíl:** Požadavkem je, že kód musí být ohraničen znaky, které očekává automatizační skript, a tedy „“javascript“ na začátku a „““ na konci.
2. **Výzva:**

Refactor the code to improve readability and maintainability. Ensure the refactored code still passes all tests and the test file remains the same.

Code:

```
```javascript
[kód]
```
```

Tests:

```
```javascript
[soubor testů]
```
```

### 3. Hodnocení:

- a) Alternativní třídy s různými rozhraními: Základní hodnocení je negativní. Kód sice je refaktORIZOVÁN správně, ale nebyl splněn 1 ze 4 testů. Rozšiřující hodnocení je pozitivní. Tento zápach je tedy použit v další iteraci.

- b) Dlouhá funkce: Základní i rozšiřující hodnocení je pozitivní.
- c) Řetězce zpráv: Základní i rozšiřující hodnocení je pozitivní.

Ohraničení kódu bylo splněno ve všech případech.

- 4. **Iterace:** V další iteraci je třeba se zaměřit na vnější chování kódu.

### Druhá iterace:

- 1. **Cíl:** Požadavkem je zachování vnějšího chování kódu.
- 2. **Výzva:**

Refactor the code to improve readability and maintainability. Ensure the refactored code still passes all tests and the test file remains the same. Ensure the external behavior of the code remains the same.

Code:

```
```javascript
[kód]
```
```

Tests:

```
```javascript
[soubor testů]
```
```

### 3. Hodnocení:

- a) Alternativní třídy s různými rozhraními: Základní hodnocení je pozitivní.
- b) Líný prvek: Základní hodnocení je negativní. Kód byl refaktorizován pouze částečně. Testy, a tím pádem i vnější chování kódu, byly splněny.
- c) Globální data: Základní i rozšiřující hodnocení je negativní. Kód byl sice refaktorizován s hodnocením 1, ale testy nebyly splněny. V případě tohoto zápachu je ale splnění testů po refaktorizaci složitějším úkolem, protože přesunutím globálních dat do třídy se změní způsob instancování a používání těchto dat. Jedná se o ojedinělý takový případ, a proto bude pro účely optimalizace výzvy hodnocení zápachu Globální data ignorováno.

Vnější chování kódu bylo v případě dříve problematického zápachu zachováno.

- 4. **Iterace:** V další iteraci je třeba se zaměřit na úplnou refaktorizaci kódu.

### Třetí iterace:

- 1. **Cíl:** Požadavkem je úplná refaktorizace kódu.

2. **Výzva:** Pro tento nový požadavek by mohlo být řešením přidáním seznamu zápachů do vstupních dat a podněcování modelu k myšlenkovému řetězci, kdy model vybere ze seznamu zápachů ty, které kód obsahuje, vygeneruje jejich definice a následně kód refaktorizuje. Pro přidání tohoto seznamu do vstupních dat je použita systémová výzva:

You are a software engineer proficient in refactoring code. Here is a list of code smells you know:

- Alternative Classes with Different Interfaces
- Comments
- Data Class
- Data Clumps
- Divergent Change
- Duplicated Code
- Feature Envy
- Global Data
- Insider Trading
- Large Class
- Lazy Element
- Long Function
- Long Parameter List
- Loops
- Message Chains
- Middle Man
- Mutable Data
- Mysterious Name
- Primitive Obsession
- Refused Bequest
- Repeated Switches
- Shotgun Surgery
- Speculative Generality
- Temporary Field

Následuje uživatelská výzva:

Refactor the code to improve readability and maintainability. Ensure the refactored code still passes all tests and the test file remains the same. Ensure the external behavior of the code remains the same. Let's solve this step by step to be sure you don't miss anything:

Step 1: Identify the code smell in the code.

Step 2: Write down the definition of the code smell you identified.

Step 3: Refactor the code.

```
Code:
```javascript
[kód]
```

Tests:
```javascript
[soubor testů]
```
```

### 3. Hodnocení:

- Líný prvek:** Základní hodnocení je negativní. Zápach nebyl identifikován ani refaktori-zován. Testy byly splněny.
- Řetězce zpráv:** Základní hodnocení je pozitivní. Tento zápach byl vyzkoušen znovu pro ověření, zda je model schopen zápach identifikovat. Výsledný kód je však až na pár detailů identický s kódem z první iterace.
- Opakované switche:** Základní hodnocení je pozitivní. Zápach byl správně identifikován a refaktORIZOVÁN.
- Odmítnuté dědictví:** Výzva byla vyzkoušena na dvou kódech s tímto zápachem. Oproti explorační fázi, kde model tento zápach s použitím zero-shot výzvy nebo myšlen-kových řetězců bez seznamu zápachů neidentifikoval, byl tentokrát zápach správně identifikován i refaktORIZOVÁN. U jednoho kódu ale došlo k výrazné změně vnějšího chování kódu přepsáním API třídy, a u obou kódů byl ponechán nadbytečný kód. I tak je ale výsledek této výzvy lepší než použitím pouze zero-shot výzvy nebo vynecháním seznamu zápachů.

4. **Iterace:** V další iteraci je třeba se zaměřit na zachování API třídy.

### Čtvrtá iterace:

- Cíl:** Požadavkem je zachování API třídy.
- Výzva:** Systémová výzva je stejná jako v předchozí iteraci.

Uživatelská výzva:

```
Refactor the code to improve readability and maintainability. Ensure
the refactored code still passes all tests and the test file remains
the same. Ensure the external behavior of the code remains the same
by maintaining the structure of the API class. Let's solve this step
by step to be sure you don't miss anything:
Step 1: Identify the code smell in the code.
Step 2: Write down the definition of the code smell you identified.
Step 3: Refactor the code.
```

```
Code:
```javascript
[kód]
```

Tests:
```javascript
[soubor testů]
```
```

### 3. Hodnocení:

- a) Odmítnuté dědictví: Výzva byla znovu vyzkoušena na dvou kódech s tímto zápachem. U jednoho kódu bylo základní hodnocení pozitivní a u druhého, v minulé iteraci problematického kódu, bylo základní hodnocení negativní – nebyl splněn 1 z 5 testů. Tento test ale ověřuje právě funkcionalitu odmítnutého dědictví, a tak by se za určitých okolností dal přehlédnout. Oproti minulé iteraci bylo tedy zachování API třídy výrazně zlepšeno.
- b) Cykly: Základní i rozšiřující hodnocení je pozitivní.
- c) Shluky dat: Základní i rozšiřující hodnocení je pozitivní.

4. **Iterace:** Tato iterace byla poslední. Výsledky byly až na výjimku pozitivní. Výzva je tedy považována pro účely této práce za dostatečně optimalizovanou.

## Výsledky optimalizace výzvy

Výsledkem optimalizace výzvy jsou tedy systémová a uživatelská výzva ze čtvrté iterace, které budou použity k posouzení refaktorizačních schopností.

## 2.5. Výsledky refaktORIZACE

V této sekci je popsána čtvrtá fáze výzkumu, jejímž cílem je získat výsledky pro posouzení refaktORIZAČNÍCH schopností jazykových modelů. Prvním krokem je získání výsledků refaktORIZACE modelem pomocí výzvy vybrané optimalizací a automatizačních skriptů pro generování a extrakci výsledků.

### Automatizační skripty

Pro automatizaci procesu generování a testování refaktORIZOVANÉHO kódu byly vytvořeny tři skripty.

#### Generování refaktORIZOVANÉHO kódu

Tento skript (`main.py` v přílohách) je navržen tak, aby automatizoval proces refaktORIZACE JavaScriptových vzorků kódu pomocí OpenAI API. Skript iteračně nad všemi složkami západů vloží soubor kódu a jeho odpovídající testovací soubor do výzvy, kterou posílá modelu GPT-4o, aby získal refaktORIZOVANOU verzi kódu. Posléze ukládá odpověď s refaktORIZOVANÝM kódem do nového souboru. Skript také zajišťuje správné pojmenování výstupních souborů pro uložení více generací refaktORIZOVANÝCH výsledků.

#### Extrakce kódu z výstupu modelu

Extrakční skript (`extract.py` v přílohách) je navržen k automatizaci procesu extrakce JavaScriptového kódu z výstupů generujícího skriptu. Skript prochází adresáře se zápachy kódu, konkrétně se zaměřuje na podadresáře „refactored“, kde čte obsah textových souborů a vyjímá z nich bloky kódu uzavřené v patřičných sekvencích znaků do souborů s příponou „.js“.

#### Spuštění testů

Třetí skript (`main.test.js` v přílohách) automatizuje testovací proces – najde všechny JavaScriptové kódy ve složkách „refactored“ a podle čísla na začátku názvu souboru pro ně spustí příslušné testy. Testy jsou pro každý vzorek kódu definovány ve funkci „runDescribe“, do které vstupuje API třída refaktORIZOVANÉHO kódu, a tím je pomocí definování API třídy v každém vzorku kódu možné testy spustit pro libovolný kód, který takovou třídu exportuje. Toto řešení výrazně zjednodušuje testování refaktORIZOVANÉHO kódu.

### Výsledky refaktORIZACE

V dalším kroku byl každý výsledek refaktORIZACE subjektivně ohodnocen a ve zvláštních případech byla uvedena i krátká poznámka. Výsledky byly zapsány do tabulky 2.1, která obsahuje název

zápachu, hodnocení refaktORIZACE dříve uvedenou škálou, počet splněných testů, poznámku a sloupec identifikace zápachu, který informuje o tom, zda byl zápach identifikován v myšlenkovém řetězci výsledku:

| <b>Zápach – kontext</b>                        | <b>Hod.</b> | <b>Testy</b> | <b>Id.</b> | <b>Poznámka</b>  |
|--|-------------|--------------|------------|--|
| <b>Alternativní třídy s různými rozhraními</b> |             |              |            |  |
| E-shop   | 1           | 3/4          | Ano        |  |
| Bankovníctví                                   | 1           | 3/4          | Ano        |  |
| <b>Komentáře</b>                               |             |              |            |  |
| E-shop   | 4           | 4/4          | Ne         |  |
| Bankovníctví                                   | 3           | 1/1          | Ano        | Tento zápach byl opraven, ale kód stále obsahuje nejasné číselné hodnoty.  |
| <b>Datová třída</b>                            |             |              |            |  |
| E-shop   | 5           | 5/5          | Ano        | Zápach byl opraven přidáním nepožadované funkcionality do problematické třídy.   |
| Bankovníctví                                   | 1           | 5/5          | Ano        |  |
| <b>Shluky dat</b>                              |             |              |            |  |
| E-shop   | 1           | 4/4          | Ano        |  |
| Bankovníctví                                   | 3           | 9/9          | Ano        | Mohla by být vytvořena i třída Account, ale v kontextu tohoto kódu by to nebylo přínosné, protože se číslo účtu a jméno nepoužívají společně v parametrech funkcí. |
| <b>Divergentní změny</b>                       |             |              |            |  |
| E-shop   | 1           | 4/5          | Ne         |  |
| Bankovníctví                                   | 5           | 7/7          | Ne         |  |
| <b>Duplicitní kód</b>                          |             |              |            |  |
| E-shop   | 1           | 5/5          | Ano        |  |
| Bankovníctví                                   | 3           | 7/7          | Ano        | Byla vytvořena třída, která by se dala považovat za datovou, ale model jí dal jednu nevyžádanou funkci, aby to zápach nebyl.                                       |
| <b>Závistivá funkce</b>                        |             |              |            |  |

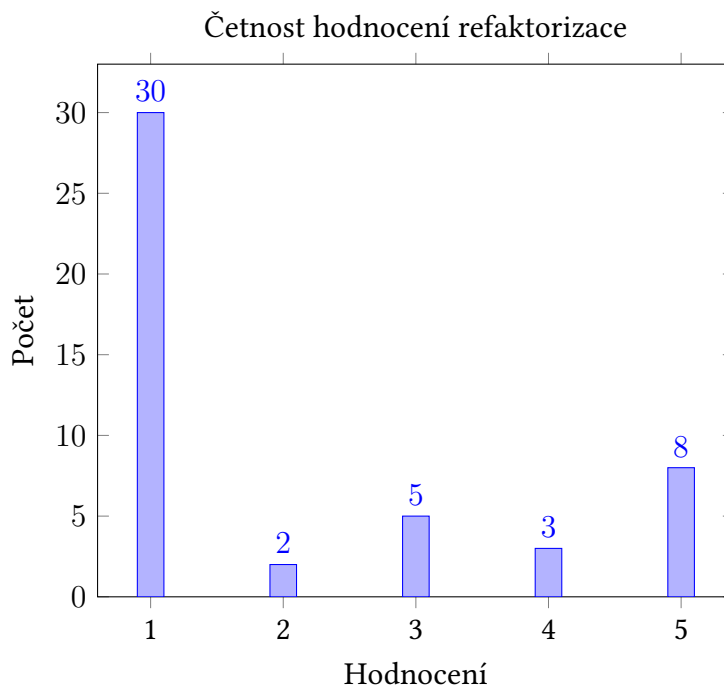
|                                     |   |     |     |   |
|-------------------------------------|---|-----|-----|---|
| E-shop                              | 1 | 5/5 | Ano |   |
| Bankovníctví                        | 1 | 3/3 | Ano |   |
| <b>Globální data</b>                |   |     |     |   |
| E-shop                              | 2 | 5/5 | Ano | Byl přidán potenciální zápach datové třídy.   |
| Bankovníctví                        | 1 | 4/4 | Ano |   |
| <b>Obchodování zasvěcených osob</b> |   |     |     |   |
| E-shop                              | 5 | 8/8 | Ne  |   |
| Bankovníctví                        | 5 | 7/7 | Ne  |   |
| <b>Velká třída</b>                  |   |     |     |   |
| E-shop                              | 1 | 7/7 | Ano |   |
| Bankovníctví                        | 5 | 5/5 | Ano |   |
| <b>Líný prvek</b>                   |   |     |     |   |
| E-shop                              | 1 | 4/4 | Ne  | Ve větší velikosti kódu by zde zápach nebyl, v tomto případě by ale mohl být odstraněn. |
| Bankovníctví                        | 4 | 6/6 | Ne  |   |
| <b>Dlouhá funkce</b>                |   |     |     |   |
| E-shop                              | 1 | 4/4 | Ano | Ověřování účtů mohlo být extrahováno do dalších funkcí.                                 |
| Bankovníctví                        | 3 | 7/7 | Ano |   |
| <b>Dlouhý seznam parametrů</b>      |   |     |     |   |
| E-shop                              | 1 | 6/6 | Ano |   |
| Bankovníctví                        | 1 | 4/5 | Ano |   |
| <b>Cykly</b>                        |   |     |     |   |
| E-shop                              | 1 | 7/7 | Ano |   |
| Bankovníctví                        | 1 | 5/5 | Ano |   |
| <b>Řetězce zpráv</b>                |   |     |     |   |
| E-shop                              | 1 | 4/4 | Ne  |   |
| Bankovníctví                        | 1 | 5/5 | Ano |   |
| <b>Middle man</b>                   |   |     |     |   |
| E-shop                              | 1 | 4/4 | Ano |   |



|                                |      |             |       |   |
|--------------------------------|------|-------------|-------|---|
| Bankovníctví                   | 1    | 6/8         | Ano   |   |
| <b>Mutabilní data</b>          |      |             |       |   |
| E-shop                         | 1    | 7/7         | Ne    |   |
| Bankovníctví                   | 1    | 5/5         | Ne    |   |
| <b>Záhadná jména</b>           |      |             |       |   |
| E-shop                         | 1    | 6/6         | Ano   |   |
| Bankovníctví                   | 1    | 4/4         | Ano   |   |
| <b>Posedlost jednoduchostí</b> |      |             |       |   |
| E-shop                         | 3    | 5/5         | Ano   | Telefonní číslo je také příkladem posedlosti jednoduchostí. |
| Bankovníctví                   | 1    | 4/4         | Ano   |   |
| <b>Odmítnuté dědictví</b>      |      |             |       |   |
| E-shop                         | 1    | 6/6         | Ano   |   |
| Bankovníctví                   | 4    | 0/5         | Ano   | RefaktORIZACE zanechala zbytečný kód.                       |
| <b>Opakované switche</b>       |      |             |       |   |
| E-shop                         | 5    | 4/4         | Ano   | Byl zaveden počínající zápach divergentní změny.            |
| Bankovníctví                   | 1    | 6/6         | Ano   |   |
| <b>Operace brokovnicí</b>      |      |             |       |   |
| E-shop                         | 1    | 7/7         | Ne    |   |
| Bankovníctví                   | 5    | 5/6         | Ne    |   |
| <b>Spekulativní obecnost</b>   |      |             |       |   |
| E-shop                         | 1    | 4/4         | Ano   |   |
| Bankovníctví                   | 5    | 4/4         | Ne    |   |
| <b>Dočasný atribut</b>         |      |             |       |   |
| E-shop                         | 2    | 4/4         | Ano   |   |
| Bankovníctví                   | 1    | 3/3         | Ano   |   |
| <b>Celkové výsledky</b>        | 2.10 | 237/<br>249 | 35/48 |   |

Tabulka 2.1.: Výsledky refaktORIZACE kódu

Výsledky hodnocení refaktORIZACE byly zaneseny do grafu 2.1.



Obrázek 2.1.: Graf četnosti hodnocení refaktORIZACE

## Zodpovězení výzkumných otázek

### 1. Jak je model GPT-4o přesný při detekci a identifikaci zápachů v kódu?

Model GPT-4o byl schopen správně identifikovat zápachy v kódu v 72,92 % případů.

### 2. Jaký vliv má inženýrství výzev? Existuje metoda, která má lepší výsledky než ostatní?

Vliv inženýrství výzev je zásadní pro úspěšnost refaktORIZACE, což je zjevné z procesu optimalizace výzev, kde bylo dosaženo výrazného zlepšení výsledků po několika iteracích. Nejlepší výsledky byly dosaženy výzvou, která modelu poskytla seznam zápachů v kódu a následně ho vedla k myšlenkovému řetězci, kde model vybral z tohoto seznamu zápachy, které kód obsahuje, a refaktORIZOVAL je. Tato metoda dosáhla z vyzkoušených metod nejlepších výsledků, ale je třeba poznamenat, že je pravděpodobné, že existuje ještě lepší kombinace vstupních dat, technik a instrukcí, které by mohly vést k ještě lepším výsledkům.

### 3. Jaká je úspěšnost opravy zápachů v kódu pomocí modelu GPT-4o?

Za zcela úspěšnou refaktORIZACI je považována refaktORIZACE s hodnocením 1 a všemi testy splněnými. Zcela úspěšná refaktORIZACE lze sledovat v 25 případech, což představuje 52 % z celkového počtu. V 9 případech (18,75 %) byl zápach odstraněn částečně a kód prošel všemi

testy, což je situace, která sice není tak ideální jako zcela úspěšná refaktORIZACE, ale i tak vede ke kvalitnějšímu kódu. Celkově tedy ke zlepšení kvality kódu došlo v 70,75 % případů. Zbylé případy tvoří 29,15 % a jsou považovány za neúspěšnou refaktORIZACI a mají tedy hodnocení 5 nebo nesplněné testy. Takových případů je celkem 14, z čehož v 5 případech byl zápal správně odstraněn, ale nebyly splněny všechny testy. Testů bylo celkově splněno 95,18 % a 41 (85,42 %) refaktORIZOVANÝCH kódů splnilo všechny testy.



## 3. Diskuse

Tato kapitola se zaměřuje na posouzení refaktorizačních schopností modelu GPT-4o, diskusi nad výsledky výzkumu, jeho omezeními a možnými zlepšeními.

### Posouzení refaktorizačních schopností

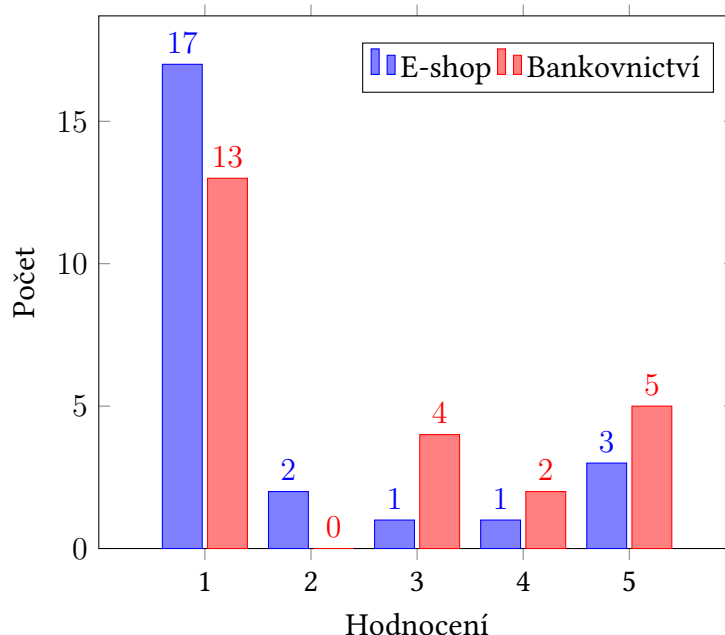
Celkově model GPT-4o vykazuje schopnost detekovat, identifikovat a refaktorizovat zápachy v kódu způsobem, který vede ke zlepšení kvality kódu ve většině případů.

Model GPT-4o byl schopen identifikovat většinu zápachů v kódu s přesností 72,92 %. Výsledky refaktorisace ukázaly, že v 70,75 % případů došlo ke zlepšení kvality kódu více či méně úspěšnou refaktorisací, která zároveň zachovala vnější chování kódu, což je základní předpoklad refaktorisace. 52 % případů bylo zcela úspěšných, což znamená, že byl zápach vhodně a zcela odstraněn. V některých případech byla refaktorisace také více či méně úspěšná, ale nebyly splněny všechny testy. V praxi by takové případy vyžadovaly další manuální úpravy kódu, aby bylo možné zajistit správnou funkčnost, ale i tak by pravděpodobně proces refaktorisace urychlily a zjednodušily. Celkově z výsledků plyne, že model GPT-4o je velmi užitečným nástrojem pro detekci a refaktorisaci zápachů, zejména proto, že je schopen většinu zápachů refaktorizovat oproti klasické lidské refaktorisaci za velmi krátkou dobu. Nicméně je stále nutné jeho výstupy manuálně ověřovat a upravovat, což vyžaduje čas a znalosti problematiky zápachů v kódu. Není tedy možné doporučit využití modelu k tomuto účelu vývojářům, kteří nemají dostatečné znalosti v oblasti refaktorisace.

### Omezení výzkumu

Je nutné podotknout, že dosažené výsledky se týkají pouze kódu v JavaScriptu a problémových domén e-shop a bankovníctví a nemusí být tedy zobecnitelné na všechny domény a programovací jazyky. Z grafu 3.1 sice kvůli malému počtu vzorků a složité izolaci vlastností kódu nelze vyvodit obecné závěry, ale naznačuje, že by mezi problémovými doménami mohly být rozdíly v úspěšnosti refaktorisace. Senzitivita modelu na problémovou doménu už je nicméně známým faktem.

Porovnání četnosti hodnocení refaktORIZACE mezi problémovými doménami



Obrázek 3.1.: Graf porovnání četnosti hodnocení refaktORIZACE mezi problémovými doménami

Dalším potenciálním omezením je, že ke konzistentním výsledkům byla při získávání odpovědí modelu zvolena teplota s hodnotou 0 v předpokladu, že tím bude dosaženo deterministického chování modelu. V pozdní fázi výzkumu ale bylo zjištěno, že nastavení teploty modelu na hodnotu 0 neznamena naprosto deterministické chování modelu, jen se k němu blíží. To znamená, že při opakovaném spuštění modelu s tímto nastavením mohou být výsledky mírně odlišné. Tato skutečnost představuje hrozbu pro replikovatelnost výsledků. Bylo proto na několika vzorcích kódu provedeno opakované spuštění modelu s nastavením teploty na 0 a výsledky byly porovnány. Text který kód provázal se občas lišil, ale výsledný kód byl sémanticky identický s občasnými drobnými změnami např. v pojmenování proměnných (rozdíly lze vidět na vzorcích odpovědí z první iterace optimalizace výzvy, tedy soubory obsahující „opt-1“). I tak ale tato skutečnost může znamenat, že výsledky mohou být zkreslené a není možné zaručit, že by bylo možné získat stejné výsledky i při opakovaném spuštění modelu na stejných vstupech.

Lepších výsledků by bylo možné dosáhnout také s větším korpusem kódů, který by byl rozmanitější a obsahoval kódy s různými vlastnostmi. Takový korpus by ale také byl časově významně náročnější na vytvoření a hodnocení s ohledem na nutné vytvoření testů pro jeho výstupy a ověření správnosti refaktORIZACE. V rámci práce nebyla nalezena metoda automatického hodnocení výsledků refaktORIZACE pro všechny zápachy, což by mohlo zjednodušit a zrychlit proces hodnocení výsledků.

## 4. Závěr

Cílem této práce je posoudit refaktorizační schopnosti velkých jazykových modelů, a proto se zabývá výzkumem schopností modelu GPT-4o detekovat, identifikovat a refaktorizovat zápachy v kódu. Hlavním cílem je odpovědět na tři klíčové výzkumné otázky týkající se přesnosti modelu při detekci zápachů, vlivu inženýrství výzev na úspěšnost refaktorisace a úspěšnosti opravy zápachů v kódu pomocí modelu GPT-4o.

V rámci metodiky výzkumu byl vytvořen korpus kódů obsahující různé zápachy v jazyce JavaScript. Proces tvorby tohoto korpusu zahrnoval i tvorbu testů pro každý jednotlivý kód pro efektivní ověřování vnějšího chování. Následně byla provedena explorační fáze, která se zaměřila na zkoušení detekce a refaktorisace vybraných zápachů. Výsledky této fáze poskytly různé poznatky, které byly využity při optimalizaci výzvy.

Optimalizace výzvy probíhala v několika iteracích, během nichž byly testovány různé výzvy a techniky jejich tvorby. Finální výzva byla vytvořena tak, aby model vedla k požadovanému způsobu identifikace a refaktorisace zápachů, přičemž zahrnovala instrukci získanou algoritmem APE, seznam zápachů, definované testy a samotný kód, a zároveň podněcovala model k myšlenkovým řetězcům. Tato optimalizovaná výzva byla následně použita k posouzení refaktorizačních schopností modelu.

Dále byly vytvořeny automatizační skripty, které výrazně šetří čas při generování refaktorisovaného kódu a jeho testování. Tyto skripty zahrnují procesy pro generování kódu, extrakci kódu z výstupů modelu a následné spouštění testů, čímž se celý proces hodnocení zjednodušil. Po získání odpovědi modelu byl každý refaktorisovaný kód ručně vyhodnocen pro zjištění úspěšnosti refaktorisace.

Výsledky ukázaly, že model GPT-4o je schopen ve většině případů správně identifikovat a refaktorisovat zápachy v kódu, čímž zlepšuje jeho kvalitu a udržitelnost. Je nutné podotknout, že tyto výsledky jsou silně závislé na použité výzvě a dalších faktorech, jako je kvalita testů a zvolené hodnocení refaktorisace. Celkově se ukázalo, že model GPT-4o je užitečným nástrojem pro detekci a refaktorisaci zápachů v kódu, ale je třeba brát v úvahu jeho omezení a závislost na vlastnostech vstupních dat.

Tato práce poskytuje poznatky o schopnostech modelu GPT-4o v oblasti údržby a refaktorisace kódu, čímž se snaží přispět k rozšíření znalostí v této velmi aktuální oblasti softwarového inženýrství. Vhodným rozšířením této práce by mohlo být testování na větším korpusu kódů, který by obsahoval širší spektrum problémů a složitostí. Dalším navázáním by také mohlo být vytvoření

automatizovaného systému pro refaktORIZACI kódu, který by prováděl refaktORIZACI kódu bez nutnosti lidské intervence.



# Seznam použitých zdrojů

1. PETERSEN, Kai; WOHLIN, Claes; BACA, Dejan. The Waterfall Model in Large-Scale Development. In: *Product-Focused Software Process Improvement* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 386–400 [cit. 2024-07-04]. ISBN 978-3-642-02151-0. Dostupné z DOI: 10.1007/978-3-642-02152-7\_29.
2. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION [ISO]; INTERNATIONAL ELECTROTECHNICAL COMMISSION [IEC]; INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS [IEEE]. *Software engineering — Software life cycle processes — Maintenance*. Third ed. 2022.
3. TRIPATHY, Priyadarshi; NAIK, Kshirasagar. *Software evolution and maintenance: a practitioner's approach*. Hoboken: Wiley, [b.r.]. ISBN 978-0-470-60341-3.
4. ALLMAN, Eric. Managing Technical Debt: Shortcuts that save money and time today can cost you down the road. *Queue* [online]. 2012, 10(3), 10–17 [cit. 2024-03-29]. ISSN 1542-7730. Dostupné z DOI: 10.1145/2168796.2168798.
5. LEHMAN, M. M. Laws of software evolution revisited. In: *Software Process Technology* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, s. 108–124 [cit. 2024-04-09]. ISBN 978-3-540-61771-6. Dostupné z DOI: 10.1007/BFb0017737.
6. SURYANARAYANA, Girish; SAMARTHYAM, Ganesh; SHARMA, Tushar. *Refactoring for Software Design Smells: Managing technical debt*. Oxford, England: Morgan Kaufmann, 2014. ISBN 978-0128013977.
7. FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. 2nd ed. Pearson Education, [b.r.]. ISBN 0-13-475759-9.
8. TUFANO, Michele; PALOMBA, Fabio; BAVOTA, Gabriele; OLIVETO, Rocco; PENTA, Massimiliano Di; LUCIA, Andrea De; POSHYVANYK, Denys. When and Why Your Code Starts to Smell Bad. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* [online]. Florence: IEEE, 2015, s. 403–414 [cit. 2024-03-29]. ISBN 978-1-4799-1934-5. ISSN 1558-1225. Dostupné z DOI: 10.1109/ICSE.2015.59.
9. OPENAI. *Better language models and their implications* [online]. [cit. 2024-06-09]. Dostupné z: <https://openai.com/index/better-language-models/>.
10. RADFORD, Alec; WU, Jeff; CHILD, Rewon; LUAN, David; AMODEI, Dario; SUTSKEVER, Ilya. Language Models are Unsupervised Multitask Learners. In: 2019. Dostupné také z: <https://api.semanticscholar.org/CorpusID:160025533>.

11. STÖFFELBAUER, Andreas. *How Large Language Models work* [online]. 2023. [cit. 2024-06-28]. Dostupné z: <https://medium.com/data-science-at-microsoft/how-large-language-models-work-91c362f5b78f>.
12. IBM. *What is machine learning (ML)?* [online]. [cit. 2024-07-06]. Dostupné z: <https://www.ibm.com/topics/machine-learning>.
13. KUDAN, Natalie. *The difference between labeled and unlabeled data* [online]. [cit. 2024-07-06]. Dostupné z: <https://toloka.ai/blog/labelled-data-vs-unlabelled-data/>.
14. AYDIN, Aysel. *Understanding Word Embeddings in NLP* [online]. 2024-06-30. [cit. 2024-07-15]. Dostupné z: <https://ayselaydin.medium.com/9-understanding-word-embeddings-in-nlp-1c86a46f7942>.
15. SEDINKINA, Marina. *History of GPT-4: In the beginning was a Word Embedding, the path to the human-level performance*. [online]. 2023-04-04. [cit. 2024-07-15]. Dostupné z: <https://medium.com/datareply/history-of-gpt-4-a71d4705b3d2>.
16. VASWANI, Ashish; SHAZEER, Noam M.; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. Attention is All you Need. In: *Neural Information Processing Systems*. 2017. Dostupné také z: <https://api.semanticscholar.org/CorpusID:13756489>.
17. ZHAN, Qiusi; FANG, Richard; BINDU, Rohan; GUPTA, Akul; HASHIMOTO, Tatsunori; KANG, Daniel. *Removing RLHF Protections in GPT-4 via Fine-Tuning*. 2024. Dostupné z arXiv: 2311.05553 [cs.CL].
18. JAMES HILLS, Shyamal Anadkat. *Using logprobs* [online]. [cit. 2024-07-04]. Dostupné z: [https://cookbook.openai.com/examples/using\\_logprobs](https://cookbook.openai.com/examples/using_logprobs).
19. AMATRIAIN, Xavier. *Prompt Design and Engineering: Introduction and Advanced Methods*. 2024. Dostupné z arXiv: 2401.14423 [cs.SE].
20. CÁMARA, Javier; TROYA, Javier; BURGUEÑO, Lola; VALLECILLO, Antonio. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* [online]. 2023, 22(3), 781–793 [cit. 2024-06-09]. ISSN 1619-1366. Dostupné z DOI: 10.1007/s10270-023-01105-5.
21. LAI, Viet Dac; NGO, Nghia Trung; VEYSEH, Amir Pouran Ben; MAN, Hieu; DERNONCOURT, Franck; BUI, Trung; NGUYEN, Thien Huu. *ChatGPT Beyond English: Towards a Comprehensive Evaluation of Large Language Models in Multilingual Learning*. 2023. Dostupné z arXiv: 2304.05613.
22. OPENAI. *Prompt engineering* [online]. [cit. 2024-06-26]. Dostupné z: <https://platform.openai.com/docs/guides/prompt-engineering>.
23. DAIR.AI. *General Tips for Designing Prompts: Prompt Engineering Guide* [online]. [cit. 2024-06-26]. Dostupné z: <https://www.promptingguide.ai/introduction/tips>.
24. DAIR.AI. *Prompting Techniques: Prompt Engineering Guide* [online]. [cit. 2024-06-26]. Dostupné z: <https://www.promptingguide.ai/techniques>.

25. HULBERT, Dave. *Using Tree-of-Thought Prompting to boost ChatGPT's reasoning* [<https://github.com/dave1010/tree-of-thought-prompting>]. Zenodo, 2023. Dostupné z DOI: 10.5281/ZENODO.10323452.
26. ZHOU, Yongchao; MURESANU, Andrei Ioan; HAN, Ziwen; PASTER, Keiran; PITIS, Silviu; CHAN, Harris; BA, Jimmy. *Large Language Models Are Human-Level Prompt Engineers*. 2023. Dostupné z arXiv: 2211.01910 [cs.LG].
27. VELÁSQUEZ-HENAO, Juan David; FRANCO-CARDONA, Carlos Jaime; CADAVID-HIGUITA, Lorena. Prompt Engineering: a methodology for optimizing interactions with AI-Language Models in the field of engineering. *DYNA* [online]. 2023, 90(230), 9–17 [cit. 2024-07-01]. ISSN 2346-2183. Dostupné z DOI: 10.15446/dyna.v90n230.111700.
28. WHITE, Jules; FU, Quchen; HAYS, Sam; SANDBORN, Michael; OLEA, Carlos; GILBERT, Henry; ELNASHAR, Ashraf; SPENCER-SMITH, Jesse; SCHMIDT, Douglas C. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. 2023. Dostupné z arXiv: 2302.11382 [cs.SE].



## A. Externí přílohy

Externí přílohy této bakalářské práce jsou umístěny na adrese:

<https://github.com/Priban/using-LLMs-to-detect-and-repair-code-smells>

Struktura úložiště:

---

|   |  |
|---|--|
| <code>code-dataset/</code>                | složka s materiály praktické části               |
| <code>code_smells/</code>                 | korpus zápachů v kódu                            |
| <code>(název_zápachu)/</code>             | složka s kódem a testy pro daný zápach           |
| <code>(1/2).js</code>                     | kód se zápachem (1 – e-shop, 2 – bankovníctví)   |
| <code>(1/2).tests.js</code>               | testy kódu                                       |
| <code>refactored/</code>                  | složka s refaktorizovaným kódem                  |
| <code>(1/2)_1.(js/txt)</code>             | refaktorizovaný kód/odpověď modelu               |
| <code>(1/2)_opt-(iterace).(js/txt)</code> | refaktorizovaný kód/odpověď z optimalizace výzev |
| <code>extract.py</code>                   | skript pro extrakci kódu z odpovědí modelu       |
| <code>main.py</code>                      | skript pro generování refaktorizovaného kódu     |
| <code>main.test.js</code>                 | skript spouštějící testy                         |
| <code>thesis.pdf</code>                   | text práce v PDF                                 |

---