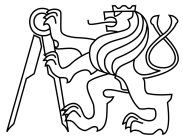
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Automatické pojmenovávání skupin slov
Student:	Bc. Jan Effenberger
Vedoucí:	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2018/19

Pokyny pro vypracování

1) Proveďte rešerši postupů a softwarových nástrojů, jakými lze lingvisticky zatřiďovat slova do skupin významově podobných slov.
2) Na základě předchozí rešerše vyberte vhodné postupy, kterými lze přiřadit skupině slov společný název.
3) Vybrané postupy diskutujte s vedoucím práce a navržená řešení implementujte ve vhodném programovacím jazyku.
4) Implementované postupy ověřte na reálných datech. Výběr testovacích dat konzultujte s vedoucím práce.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 28. února 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Automatické pojmenovávání skupin slov

Bc. Jan Effenberger

Vedoucí práce: doc. RNDr. Ing. Marcel Jiřina, Ph.D

8. května 2017

Poděkování

Rád bych na tomto místě poděkoval za trpělivé vedení a cenné rady svému vedoucímu doc. RNDr. Ing. Marcelu Jiřinovi, Ph.D. Dále pak JUDr. Janě Effenbergerové, nejlepší manželce na světě, za neutuchající podporu, trpělivost při korektuře a veškerou další pomoc, které se mi dostalo. Rád bych také poděkoval i rodině za neuvěřitelnou víru v dokončení práce. V neposlední řadě bych chtěl poděkovat osobnostem vystupujícím ve světě pod jmény Batu, Sissi, Matýsek a Džemík, kteří si tuto práci přečíst nemohou, ale neustále při jejím psaní dohlíželi na mou bdělost.

Zároveň bych tuto práci rád věnoval všem příbuzným, kteří mají zajisté na práci svůj díl, ale již bohužel nejsou mezi námi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2017 .....

Odkaz na tuto práci

Effenberger, Jan. *Automatické pojmenovávání skupin slov.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce řeší vývoj aplikace pro automatické pojmenovávání skupin slov za pomoci externích databází znalostí. V práci analyzuji možnosti popisování a běžně používané metody pro popisování. Pro návrh aplikace jsem využil praktické zkušenosti se zaměřením na další rozšiřitelnost a rozvoj. Jako externí databáze jsem v práci využil WordNet a Microsoft Concept Graph a jako programovací jazyk jsem použil jazyk Java. Podařilo se mi vytvořit takovou aplikaci, která umožňuje snadnou změnu popisovacího algoritmu a jiných částí, což je hlavním přínosem práce, a zároveň jsem sestavil dva popisovací algoritmy – každý pro jednu databázi znalostí. Výsledky popisování se v rámci funkčního testování vzhledem ke kvalitě vstupních dat ukázaly jako adekvátní a uspokojivé.

Klíčová slova slovo, popis, skupina slov, automatické popisování, popisování skupin slov, WordNet, Microsoft Concept Graph

Abstract

This work solves the development of application for automatic cluster labeling, using external databases of knowledge. In my thesis I analyze the possibilities of description and commonly used methods for the cluster labeling. I used the practical experience with the focus on further expandability and development. As an external database, I used WordNet and Microsoft Concept Graph at work, and I used Java as a programming language. I managed to create an application that makes it easy to change the labeling algorithm and other parts, which is the main benefit of my work, and I also put together two labeling algorithms – each for one knowledge base. The results of the labeling in the functional testing proved to be adequate and satisfactory with respect to the quality of the input data.

Keywords word, description, cluster of words, automatically description, cluster labeling, labeling, WordNet, Microsoft Concept Graph

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Doba, ve které dnes žijeme, přináší kromě velkých možností také velké množství dostupných informací, ať už v podobě multimediální, virtuální nebo textové. Všechny tyto dostupné informace je pak pro jejich rozsah nutné strojově zpracovávat. Existuje mnoho oblastí, které se zabývají získáváním informací z různých typů dat, ať už se jedná o obrázky, video nebo prostý text. Text mining je jednou z takových oblastí, která se zabývá získáváním znalostí z prostého textu a je možné ji rozdělit na několik různých podoblastí. Jedna z nich je pak tématem mé práce – automatické pojmenovávání skupin slov. Tato oblast text miningu patří k těm, bez kterých bychom se při zpracovávání velkého množství informací, které textové dokumenty obsahují, neobešli. Pojmenovávání skupin slov najde své využití mimo jiné při hierarchickém kategorizování vědeckých článku, webových stránek či jiných libovolných textových dokumentů.

Jedno slovo, které obecně, ale přitom dostatečně výstižně popíše skupinu jiných slov dokáže člověk díky znalosti jejich významu v běžném životě určit zcela bez potíží. Stroji takové porozumění významu, které je nezbytné k určení správného popisného slova chybí, nicméně v současné době již existují způsoby, jak stroj může za pomocí dalších vlastností a znalostí určit takové slovo s dostatečnou přesností na to, aby bylo možné jej použít automatizovaně ve zpracování různých dokumentů. Implementací některých těchto způsobů porozumění pro popsání skupin slov jiným slovem se budu zabývat ve své práci.

Cílem mé práce je implementovat nástroj, který pro zadanou skupinu českých slov najde dostatečně obecné, ale stále velmi přesné slovo, které skupinu zadaných slov popíše z hlediska jejich významu. Naopak cílem mé práce není získání samotné skupiny slov pro pojmenování, a tak se jím v textu nijak nezabývám a předpokládám na vstupu vždy slova v jejich základním tvaru.

Popisování skupin slov je možné implementovat mnoha různými technikami, a proto jsem se rozhodl zaměřit na popisování pomocí samotného shluku slov se zvláštním důrazem na získávání znalostí z externího zdroje, jakým

může být například Wikipedia, Linked Data, WordNet nebo Microsoft Concept Graph. Jedná se tedy o databáze, které obsahují určitým způsobem vytvořenou ontologii jednotlivých pojmů. Ve své práci tudíž nebudu řešit rozdílové popisování shluků (Differential cluster labeling), kde se používají techniky založené na popisu dle maximálních rozdílů v jednotlivých shlucích.

V první kapitole práce se zabývám teoretickým úvodem do samotné problematiky, popisem různých principů popisování shluků slov, nahlédnutím do výběru skupin slov z dokumentů a rešerší souvisejících prací. Druhá kapitola obsahuje návrh samotné aplikace pro popisování shluků slov a návrh jednotlivých algoritmů. Třetí kapitole je věnována implementaci a je následována čtvrtou kapitolou, obsahující popis testování a zhodnocení výsledků jednotlivých algoritmů.

Kapitola 1

Teoretický základ

1.1 Pojmenovávání skupin slov

Pojmenováváním skupin slov můžeme označit proces, kde se pro vstupní množinu slov hledá množina takových slov, které ji dostatečně přesně vystihují a popisují a zároveň je výrazně menší, než vstupní množina a v nejlepším případě obsahuje jen jedno slovo. Zcela záměrně jsem v předchozí větě použil, a dále v textu budu používat, označení „slovo“ pro libovolné i víceslovné pojmenování, jelikož v naprosté většině případů je nalezené pojmenování jednoslovné. Formálněji řečeno, je to takové zobrazení z množiny množin slov do sebe sama, které splňuje tři vlastnosti: velikost, popisnost a výstižnost. Tyto vlastnosti můžeme definovat jen velice obtížně už ze samotného principu tak rozmanitého, komplexního a unikátního lidského jazyka, přesto je můžeme částečně popsat.

1.1.1 Velikost množiny pojmenování

Problém velikosti výsledné množiny můžeme vyřešit tak, že zavedeme pravidlo, že výsledná množina musí obsahovat nanejvýš stejný počet prvků jako množina vstupní. Navíc, množiny obsahující méně slov jsou zvýhodněny oproti těm množinám, které obsahují slov více, jelikož lépe plní účel takového pojmenovávání.

1.1.2 Popisnost množiny pojmenování

Problém popisnosti spočívá v tom, že každé slovo ze vstupní množiny musí být popsatelné pomocí jednoho slova z výsledné množiny na základě jasně definovaného vztahu. Pokud bychom použili hierarchické uspořádání slov, pak jistě můžeme říct, že „dub“ ze vstupní množiny je možné popsat jako „strom“ a využít tak vztahu hyperonymum (nadřazený pojem) – hyponymum (podřazený pojem). V případě vztahu synonym pak můžeme říct, že „chlapec“

je zároveň „hoch“. Kromě synonym a hyperonym je možné pro pojmenovávání využít například vztahu meronyma k holonymu, tedy vztah „část celku – celek“, jako například u slov „střecha“ a „budova“.

1.1.3 Výstižnost množiny pojmenování

Výstižnost množiny slov, která popisuje vstupní množinu slov je problémem, který lze jen velmi obtížně definovat. V důsledku toho je pak také obtížné sledovat úspěšnost jeho splnění. Nicméně každé slovo z množiny pojmenování musí být natolik výstižné, aby popisovalo co nejmenší počet slov neuvedených ve vstupní množině, a to na základě vybraného vztahu definovaného v předchozím odstavci. To si můžeme ukázat na příkladu „dub“ – „strom“. Pokud by ve vstupní množině byly stromy listnaté, jehličnaté apod., pak by s největší pravděpodobností bylo správným pojmenováním pro takovou skupinu uvedené slovo „strom“. Pokud by vstupní množina obsahovala naopak pouze listnaté stromy, pak by zajisté existovalo lepší pojmenování pro takovou skupinu – „listnatý strom“. Pakliže by ale množina obsahovala slova jako „dub“, „strom“, „růže“ a „mech“, pak by nejlepší možné pojmenování těchto slov bylo zřejmě slovo „rostlina“.

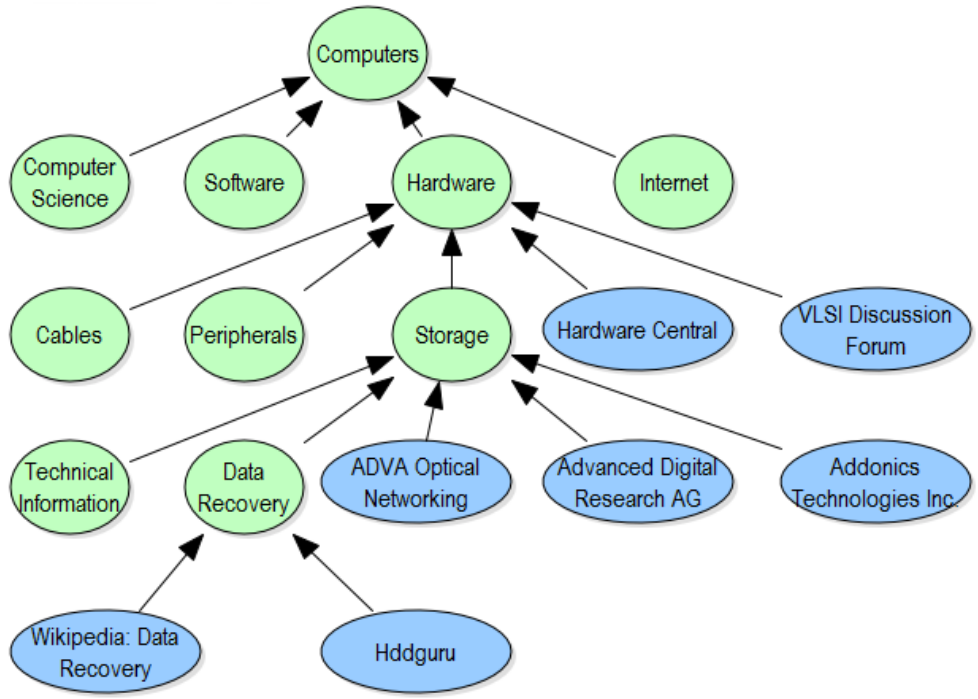
1.2 Účel

Automatické pojmenovávání skupin slov najde své uplatnění v mnoha různých úlohách text miningu, respektive skoro na každém místě, kde se zpracovává velké množství textových dat a je potřeba je kategorizovat či katalogizovat.

Například internetové katalogy webových stránek mohou být tvořeny právě na základě pojmenovávání skupin slov. Jakmile stroj nebo člověk vytvoří hierarchii dokumentů do určité stromové struktury a extrahuje z každého dokumentu charakteristická slova, mohou být tato slova použita pro pojmenování jednotlivých vnitřních uzlů stromu. Může tak vzniknout hierarchie jako na obrázku 1.1, která představuje vybranou část internetového katalogu webových stránek Open Directory Project[21]. Na tomto obrázku představují zeleně podbarvené prvky jednotlivé kategorie katalogu a modře podbarvené prvky představují odkazy umístěné v dané kategorii.

Automatické pojmenovávání skupin slov najde využití například u internetových vyhledávačů, které na základě vyhledávaného výrazu mohou určit, o jakou kategorii výsledků jeví uživatel zájem a následně mu předložit i výsledky, které na základě fulltextového vyhledávání neodpovídají.

Na základě procházení výsledků uživatelem a charakteristik jeho procházení, jako je například čas strávený u jednotlivých výsledků, může internetový vyhledávač vybrat charakteristická slova vybraných výsledků a pro ně najít společné pojmenování. Jakmile takové pojmenování vyhledávač získá, může jej použít například jako klíčové slovo pro personalizaci reklamy.



Obrázek 1.1: Ukázka hierarchie internetového katalogu Open Directory Project

V případě vyhledávačů a jim podobných internetových nástrojů bychom mohli najít celou řadu dalších podobných využití, které pracují na výše uvedeném principu.

1.3 Počáteční skupina slov

Pakliže chceme popisovat skupinu slov, vyvstává velice důležitá otázka a sice, kde tuto skupinu slov získat a jakým způsobem. V části Účel jsem uvedl, že slova se vyextrahují z každého dokumentu hierarchie a jsou pak vstupem pro pojmenovávání. Získávání skupiny slov pro pojmenování a jejich samotné pojmenovávání jsou dva problémy, které jsou relativně dobře oddělené a přitom velmi obsáhlé, výběrem slov se proto ve své práci zabývat nebudu. Nicméně se domnívám, že je nezbytné se o této problematice zmínit, a to z jednoduchého důvodu, neboť čím „lepší“ slova pro popisování získáme, tím můžeme vybrat lepší a nejjednodušší popisek.

První způsob, jak získat charakteristickou skupinu slov z dokumentu, spočívá ve výběru nejčetnějších n slov. Zde je problémem fakt, že nejčastější slova, která se v textech vyskytují, jsou velmi často spojky a/nebo slova, která nesou velmi málo informací. Což si můžeme ukázat na příkladech slov, která jsou

v první stovce nejčastějších, používaných v českých textech dle Českého národního korpusu [16]. Najdeme zde například slova „být“, „ale“, „rok“, „co“, „na“, „já“ nebo „něco“. Taková slova o dokumentu vůbec nic nevypovídají a jsou nám sice při popisování k ničemu, ale je relativně snadné je odfiltrovat.

Další snadnou možností jak získat skupinu slov jsou klíčová slova, která sice většina dokumentů neobsahuje, ale pokud ano, jsou velmi vhodnými kandidáty. Vystihují dokument vcelku přesně, je jich o několik řádů méně, než slov v samotném dokumentu a jsou sepsána přímo autorem dokumentu.

Pokročilejší metody jsou pak založené nejen na zkoumání samotného textu, ale například i na jeho členění (zcela logicky nadpis může mít větší váhu než samotný odstavec) a na dalších vlastnostech dokumentu. Nicméně výsledkem všech metod je jedna skupina slov, která navíc může být určitým způsobem ohodnocena, například počtem výskytu daného slova.

1.4 Principy pojmenovávání skupin slov

Pojmenovávání skupin slov můžeme rozdělit do dvou základních druhů:

- interní popisování skupin slov, neboli cluster–internal labeling
- rozdílové popisovaní skupin slov, neboli differential cluster labeling

Jak už názvy napovídají, zásadní rozdíl mezi těmito přístupy spočívá v tom, jaké informace pro popisování použijeme. V případě interního popisování skupin slov můžeme mluvit o využití informací, které nám poskytuje samotná skupina slov bez ohledu na jiné skupiny slov. Metody patřící do druhé skupiny naopak spočívají ve využívání informací, které nám poskytují jiné skupiny slov, zejména jejich hierarchie. Tyto metody jsou založené na maximalizaci rozdílu v popisu, tedy aby popis co nejvíce vystihoval zadanou skupinu slov a naopak co nejméně všechny ostatní skupiny.

V následující části se velmi často místo slov popisují dokumenty, které jsou uspořádány do určité skupiny. Slova bychom z takových dokumentů získali různými metodami, které jsem částečně popsal v části Počáteční skupina slov.

1.4.1 Rozdílové popisování skupin slov

U tohoto typu popisování závisí výsledek buď na relativní entropii, nebo na χ^2 výběru. V případě entropie se využívá rozdíl mezi sdruženou distribuční funkcí a marginálními distribučními funkcemi dvou diskrétních proměnných, kde jedna proměnná určuje, zda je daný dokument součástí popisované skupiny slov a druhá určuje, zda je vybrané slovo součástí vybraného dokumentu. Výsledkem je pak slovo, které je součástí největšího počtu dokumentů z dané skupiny slov.

V druhém případě výsledek závisí na členství slova v množině, kterou vybrané slovo popisuje a na přítomnosti vybraného slova ve skupině slov, které chceme popsat.

Tento typ popisování využívají autoři v pracích [4] a [27], kde například v práci [27] využívají právě χ^2 výběru pro určení rozdílů mezi slovy, nicméně ani s druhým použitým algoritmem, kterým bylo využití entropie popsané výše, nedosáhli autoři uspokojivých výsledků.

Naproti tomu v práci [4] přišli se zajímavou myšlenkou, ve které sice využili rozdílové popisování skupin slov, nicméně přidali i rozdělení dle oblastí zájmu, čímž dokázali zvýšit úspěšnost popisování na použitelnou míru. K určení možných oblastí zájmů pak využívají externí databáze znalostí, díky čemuž by bylo možné zařadit práci i do druhé kategorie.

1.4.2 Interní popisování skupin slov

Tento typ popisování je možné, stejně jako v předchozím případě, rozdělit na dvě části, kdy první z nich závisí na těžišti skupiny slov a druhý způsob závisí na externích databázích znalostí. Metody založené na principu těžiště reprezentují dokumenty jako binární vektory, kde jednotlivé složky reprezentují přítomnost vybraného slova v dokumentu. Vektory však nemusí být pouze binární, ale mohou kromě přítomnosti slova v dokumentu reprezentova i četnost výskytu slova v dokumentu. Následně se v množině vektorů, reprezentujících jednotlivé dokumenty hledá těžiště a výsledkem popisování je pak slovo nejblíže těžišti, případně nadpis dokumentu, které je nejblíže těžišti.

Druhou možností, kterou můžeme řadit do kategorie interního popisování skupin slov je popisování na základě externí databáze znalostí. Použití externí databáze znalostí spočívá v nalezení konkrétního předka popisovaných slov na základě hierarchie, která je v databázi popsána a definována například pomocí vztahu hyponymum–hyperonymum. Dle mého názoru se jedná o nejlepší množnost popisování skupin slov, jelikož představují určitý popis reálného světa, na rozdíl od ostatních metod, které mohou pracovat jen s velmi omezenou a mnohdy pokroucenou podmnožinou slov či dokumentů.

Interní popisování skupin slov je využito například v pracích [1], [3] a [34]. První práce [1] je ukázkovým případem interního popisování skupin slov, kde extrahují z dokumentů kandidáty na popisné slovo a následně je porovnávají s podobnými kandidáty, získanými z online encyklopedie Wikipedia[36]. Využívají tedy obě části – jak těžiště skupiny slov, tak externí databázi znalostí.

Velmi zajímavou prací je pak jediný český zástupce [3], kde autoři využili jako externí databázi znalostí DBpedia[2] a hledali spojitosti mezi zadanými slovy nejen na základě vztahu hyperonyma k hyponymu, ale i na základě synonym a četnosti použití slov společně v rámci uvedené databáze. Nicméně práce je dle mého názoru průzkumem, zda má smysl zabývat se touto myšlenkou i nadále, neboť ověření proběhlo pouze na velmi malé skupině dat.

Další práce [34] nepojednává pouze o popisování skupin slov, ale spíše o třídění dokumentů. Nicméně k třídění využívá databázi WordNet a vztahu hyperonyma k hyponymu. Výsledkem je pak strom, kde je dokument podřazený pod příslušným synsetem v databázi a dal by se tak považovat za popis dané množiny slov, které jsou v dokumentu významné.

1.5 Externí databáze znalostí

Jak bylo uvedeno výše, pro popisování skupin slov pomocí interní metody, se jeví jako nejvýhodnější využít externí databáze znalostí, které nám při popisování poskytnou možné kontexty a významy jednotlivých slov. Znalostních databází, respektive databází, ze kterých je možné čerpat potřebná data pro popisování je jistě více, než které zde uvádím. Do popisu níže jsem vybral takové databáze, které jsou velmi rozsáhlé (např. Microsoft Concept Graph), případně ty, které se užívají v podobných aplikacích relativně často (např. WordNet).

1.5.1 WordNet

1.5.1.1 Vlastnosti

WordNet[5][19] je velmi rozsáhlá lexikální databáze pro anglický jazyk, která vznikla přibližně v roce 1985 na univerzitě Princeton a je nadále rozvíjena. Databáze obsahuje podstatná jména, slovesa, přídavná jména a příslovce seskupené do skupin synonym. Tyto skupiny jsou nazvány jako synsety a jsou uspořádány hierarchicky v rámci vztahů hyponymum–hyperonymum a holonymum–meronymum. Pakliže má jedno slov více možných významů, je uvedeno ve více synsetech, což nám umožní nahradit ho jednoznačným identifikátorem a nemůže tak dojít k pozměnění významu jako v případě homonym. V rámci databáze tedy pracujeme s významem slova, nikoliv s jeho vyjádřením v psaném textu, což je velmi důležitý detail.

Anglický WordNet je aktuálně dostupný v poslední verzi 3.1 z roku 2012 a například verze 3.0, která se liší od nejnovější verze pouze o jednotky záznamů, obsahuje více než 155 tisíc unikátních slov, uspořádaných do více než 117 tisíc synsetů.

Kromě anglického WordNetu existují klony pro jiné, převážně evropské, jazyky a jedním z nich je i český jazyk. Český WordNet[23], jak je databáze nazvána, se začal vyvíjet roku 1998 na Masarykově univerzitě v Brně a jeho první verze v roce 1999 obsahovala přibližně 14 tisíc synsetů. Volně k dispozici pro nekomerční účely je Český WordNet ve verzi 1.9, z roku 2011 a obsahuje přibližně 23 tisíc synsetů. Další verze 2.0, která odpovídá anglickému WordNetu ve verzi 2.0, obsahuje 28 tisíc synsetů ale bohužel již není volně dostupná, ani pro nekomerční účely. Zároveň byla také vytvořena diplomová práce, jejíž téma spočívalo ve slovníkovém překladu anglického WordNetu a výsledkem

bylo 12 tisíc vytvořených kandidátů na synsety v českém WordNetu. Nicméně kvůli nutnosti ručního ověřování výsledku k rozšíření databáze doposud nedošlo. K podobným účelům je pak v databázích WordNet využit identifikátor s názvem „Inter–Lingual Index“, zkráceně ILI, který umožňuje navázat libovolný sysnet z libovolného WordNetu na anglický synset s odpovídajícím významem.

Pokud si porovnáme velikosti nejnovějšího anglického WordNetu a jediné volně dostupné verze Českého WordNetu, vidíme, že český obsahuje přibližně jednu pětinu synsetů oproti anglickému. Proto myslím, že mohu říct, že český Wordnet zaostává za anglickou verzí velmi výrazně, důsledkem čehož jsem se rozhodl, že ve své práci využiji spíše právě anglickou verzi WordNetu a pokusím se najít metodu překladu i s vědomím, že tím utrpí kvalita výsledků. Nicméně neutrpí tím kvalita samotných algoritmů, jelikož při získání Českého WordNetu v jeho poslední verzi (s markantním doplněním alespoň na úroveň anglického), ho můžeme využít pro popisování. V takovém případě bychom nemohli získat horší výsledky než při použití anglického WordNetu se strojovým překladem.

1.5.1.2 Možné využití při popisování skupin slov

Základním kamenem pro popisování pomocí lexikální databáze WordNet, je synset, ve kterém může být více slov a zároveň jedno slovo může být ve více synsetech. Všechny synsety kromě jediného, kterým je v anglickém WordNetu slovo „entity“ neboli entita, mají uveden nadřazený synset z hlediska významu. Entita je „nejvyšším“ prvkem databáze a vede do něj minimálně jedna cesta z každého synsetu. Těchto vlastností pak můžeme využít při popisování skupiny slov. Slovo může existovat ve více synsetech v závislosti na počtu významů daného slova a zároveň každý synset může mít více cest ke kořeni. Pokud bychom nalezli takový synset, který je společný pro určité významy všech slov, můžeme takový synset prohlásit za nadřazený prvek a tedy popis zadaných slov.

1.5.1.3 Možnosti připojení

Pro potřeby zpracovávání přirozeného jazyka vzniklo mnoho možností, jak propojit vlastní aplikaci s lexikální databází WordNet. Například více než desítka API přímo pro jazyk Java, která povětšinou načítají databázi přímo ze souborů a pak jí zpracovávají. Nicméně z hlediska výkonu by zajisté bylo výhodnější nechat načítání na nějakém druhu relační databáze. Mnoho projektů, které se zabývaly poskytováním dat z WordNetu ukončilo činnost úplně, nebo se dále nerozvíjejí, i když jejich potenciál mohl být velký. Nicméně projekt WordNetSQL[37] poskytuje data i z poslední verze WordNet 3.1 ve formě snímku MySQL, a tak postačí pouze tento snímek naimportovat a uživatel má k dispozici všechna důležitá data včetně vytvořených databázových pohledů.

Navíc je distribuován pod stejnou licencí jako samotný WordNet a je tedy možné ho využít v rámci této práce.

1.5.2 Microsoft Concept Graph

1.5.2.1 Vlastnosti

Společnost Microsoft vydala k volnému použití pro soukromé a nekomerční účely, tedy i pro akademické aplikace, orientovaný graf slov s názvem Concept Graph[30][32][33][11][31][28]. Tento graf je orientovaný pomocí relace nadřazený–podřazený prvek, kde nadřazený prvek je obecným označením pro podřazený prvek a všechny hrany jsou ohodnoceny pravděpodobností. Například pro vstup „python“ uvádí jako nadřazené prvky (seřazené dle pravděpodobnosti) „language“, „animal“, „snake“, „reptile“. Jak je vidět, homonyma jsou zde reprezentována jako nadřazené prvky a je zde uvedena i pravděpodobnost s jakou slovo patří právě pod tento obecný pojem. Tato databáze aktuálně obsahuje přibližně 12 miliónů slov a slovních spojení a přes 85 miliónů hran. Co do rozsahu je tedy mnohonásobně větší než WordNet, nicméně hierarchie zde naopak není tak jasná, neboť není jasný přímý předek. Je to vidět v uvedeném příkladu, neboť krajta je zcela jistě had, had je zcela jistě plaz a plaz je zcela jistě zvíře. Ve WordNetu by jako nadřazený prvek byl právě had, jeho nadřazený prvek plaz atd. Navíc oproti WordNetu, neexistuje žádná verze v jiném jazyce, než v originálním anglickém a v případě použití pro česká slova by bylo nutné využít služeb některého z překladačů.

1.5.2.2 Možné využití při popisování skupin slov

Vzhledem k tomu, že máme pro všechna slova uveden nadřazený prvek, včetně míry příslušnosti pod daný význam, můžeme tento orientovaný graf využít při popisování například tím způsobem, že si pro všechna zadaná slova najdeme nadřazené prvky a následně najdeme taková slova, která jsou ve všech množinách. Pakliže je těchto slov víc, můžeme využít i pravděpodobnost k určení nejlepšího možného výsledku. Vyvstává zde pouze otázka, zda v takovém případě vzít v potaz slovo s největší pravděpodobností, které bude ovšem velmi obecné, nebo naopak slovo, které bude mít nejmenší pravděpodobnost a bude velmi konkrétní, avšak nemusí odpovídat popisovaným dokumentům právě kvůli nízké pravděpodobnosti.

1.5.2.3 Možnosti připojení

Tuto databázi je možné si stáhnout a přistupovat k ní lokálně, nebo je možné využít internetové API, které na HTTP dotaz vrací výsledná data v podobě JSON. Z tohoto důvodu je možné využít veškeré programovací jazyky, které umožňují komunikaci přes HTTP, tedy téměř všechny, včetně jazyku Java.

1.5.3 Ostatní databáze znalostí

První z nich (WordNet) je příkladem přísně strukturovaných a zaručeně pravdivých informací, ale o menším rozsahu. Mezi podobné databáze bychom mohli zařadit například projekty DBpedia, či Open Directory Project. V obou případech se jedná o vztahy, kde jsou vazby jasně dané a data v nich uložená jsou člověkem minimálně ověřena.

Druhý (Microsoft Concept Graph) je představitelem různorodých vazeb o daleko větším rozsahu, ale s méně konkrétními informacemi. Mezi podobné databáze vazeb je možné zařadit i Google Knowledge Graph[8], který využíváme v podstatě, když vyhledáváme pomocí vyhledáváče Google. Struktura jednotlivých položek je sice podobná jako DBpedia, nicméně data jsou provázana jako u Microsoft Concept Graph.

Kapitola 2

Analýza a návrh

Problém popisování skupin slov můžeme členit na několik logických částí, a to z hlediska architektury výsledného software. Nejběžnějším architektonickým vzorem je třívrstvý model, který se skládá z datové, logické a prezentační vrstvy[35]. Datová vrstva má na starosti komunikaci mezi daty, uloženými v různých formátech, a touto aplikací, přičemž tato data předává logické vrstvě. Logická vrstva získaná data zpracovává a předává je prezentační vrstvě, která se stará o jejich zobrazení uživateli. Hlavním cílem této práce je vytvořit všechny tři vrstvy jakožto celistvou aplikaci, nicméně důraz je zcela oprávněně kladen na logickou vrstvu. Avšak ani datová vrstva nemůže být opomenuta, neboť data jsou k dispozici v rozličných formátech, jako například ve formátu textového souboru, dostupného přes protokol HTTP, nebo tabulky v MySQL databázi, a je tedy nutné poskytovat jednotný přístup tak, aby logická vrstva „nerozeznala“ rozdíl ve zdrojích. Nad jiným typem architektonického návrhu aplikace, než jakým je zmíněný třívrstvý model, nemá smysl uvažovat, neboť bychom vytvořili buď obtížně udržovatelnou aplikaci, a to v případě jednovrstvé aplikace, nebo naopak zbytečně předimenzovanou aplikaci v případě například servisně orientované architektury.

Z funkčního hlediska můžeme aplikaci rozdělit na určitý počet modulů, kde každý z nich samostatně vykonává svou práci. Tyto moduly je pak možné „spojovat“ do větších celků, dokud nevznikne samotná aplikace. Z tohoto důvodu můžeme jednotlivé vrstvy třívrstvého modelu chápat jako samostatné moduly pro:

1. komunikaci s externí databází znalostí,
2. popisování skupin slov a
3. komunikaci s uživatelem.

Tyto moduly můžeme následně rozdělit na menší podmoduly, například dle typů externích databází znalostí nebo dle způsobů popisování skupin slov.

Rozdělení na menší podmoduly není možné specifikovat, bez toho, aby tomu předcházela specifikace samotné aplikace[35].

2.1 Specifikace aplikace

Aby bylo možné vytvořit návrh aplikace, musíme znát její specifikace, ať již z tzv. „high–level“ pohledu, který bych nazval také jako manažerské shrnutí funkčnosti v několika málo větách, tak i z hlediska funkčních požadavků jako například které zdroje mají být využity, a zároveň z hlediska nefunkčních požadavků, které definují například odezvu a rychlost zpracování.

2.1.1 High–level popis

Aplikace, která bude výsledkem této práce, bude popisovat skupinu českých slov na základě algoritmů založených na externích databázích znalostí. Vstup bude kromě zadaných slov obsahovat i jejich váhu v rámci skupiny. Uživatel bude s aplikací komunikovat pomocí příkazového řádku případně konfiguračních, vstupních a výstupních soborů. Výstupem aplikace pak bude seznam možných pojmenování s vypočtenou mírou popisnosti a případně seznamem slov, která dané pojmenování popisují (v případě popisování neúplných skupin slov).

2.1.2 Funkční požadavky

Z předchozí kapitoly a přechozího hrubého pohledu na aplikaci jsem sestavil funkční i nefunkční požadavky na aplikaci tak, aby splňovala zadání celé práce, a zároveň aby splňovala osvědčené postupy na vývoj software, jakými jsou například požadavky na snadnou udržovatelnost, rozšiřitelnost apod.

Mezi těmito požadavky je, že:

1. Aplikace bude k popisování využívat externí databáze znalostí. Konkrétně bude využívat:
   - anglickou verzi WordNet a
   - Microsoft Concept Graph.
2. Aplikace bude využívat automatický překlad pomocí:
   - vlastního slovníku nebo
   - online nástrojů.

   Jelikož jsou externí databáze znalostí v anglickém jazyce, je pro popisování skupin českých slov nutné vytvořit či využít určitý typ automatického překladu, a to minimálně v rozsahu překladu z českého jazyka do

anglického a zpět apod. Pro tyto potřeby je možné použít vlastního slovníkového překladu nebo využít některý z online překladačů, které jsou pro nekomerční účely poskytovány zdarma, byť v omezeném rozsahu, a jejich dostupného API, jako je například Microsoft Bing. Překladače třetích stran vynikají svou přesností a rozsáhlostí, nejen z hlediska počtu slov přeložitelných z jednoho jazyka do druhého, ale i z hlediska počtu dostupných jazyků. Oproti tomu vlastní slovníkový překlad umožní rychlejší práci a v některých specializovaných případech lepší výsledky.

3. Aplikace bude s uživatelem komunikovat pomocí příkazového řádku.

   Vstup bude možné předat přímo jako parametr v příkazovém řádku, nebo pomocí souboru. Stejně tak výstup bude možné zobrazit na konzoli nebo vypsat do definovaného souboru.

2.1.3 Nefunkční požadavky

V rámci nefunkčních požadavků na aplikaci jsou uvedeny převážně požadavky, které plynou z ověřených postupů při vývoji software. Tyto požadavky vychází z mé vlastní praxe při vývoji software, neboť postupy, které jsou uvedeny v mnoha publikacích, jsou v praxi velmi často nepotřebné či neproveditelné. Jako nefunkční požadavky lze uvést následující:

1. Primárním cílem optimalizace popisovacích algoritmů není rychlost zpracování, ale dosažení lepší popisnosti.

2. Je nutné omezit počet dotazů na minimum v případě online překladu, například za pomocí cache.

3. Veškeré použité zdroje, knihovny a aplikace třetích stran musí být vydány pod licencí, umožňující volné použití pro nekomerční nebo akademické využití.

4. V aplikaci musí existovat mechanismus na snadnou výměnu algoritmů pro popisování skupin slov.

5. V aplikaci musí existovat mechanismus na snadnou výměnu implementace překladače.

2.2 API překladače Microsoft Bing

Společnost Microsoft poskytuje[18] pro nekomerční použití webové API svého překladače Bing Translator. V případě neplaceného použití je zde ovšem limit na přeložení maximálně 2 miliónu znaků za měsíc, což v případě průměrných 8 znaků na jedno slovo poskytuje možnosti na překlad 250 000 slov během jednoho měsíce. Během testování a ladění programu jsem se však nepřiblížil ani

hranici 100 000 slov za měsíc, díky čemuž považuji tuto hranici pro akademické použití za dostatečnou.

Pro využití překladače Microsoft Bing je nutná pouze registrace uživatele i aplikace a vygenerování tzv. „api-key“, které slouží pro autentizaci uživatele. Pro potřeby aplikace jsem vytvořil účet a zaregistroval aplikaci.

Ostatně jak již bylo řečeno v kapitole 1.5.1, použití anglického WordNetu a překladače namísto českého WordNetu je závislé pouze na rozsahu a dostupnosti české verze a pouze pro lepší ověření správnosti popisovacích algoritmů, které jsou od jednotlivých jazyků abstrahovány. Navíc v rámci specifikace i v rámci architektonického návrhu je definována možnost snadné výměny jednotlivých částí, například překladače. Všechny tyto faktory pak způsobí, že popisovací algoritmy jsou dostatečně abstrahovány od problémů s překladem externích databází znalostí a můžeme je tak testovat i vyhodnocovat bez ohledu na různá omezení překladačů.

Bohužel v rámci realizace se ukázal Microsoft Bing Translator jako nešťastně zvolený, neboť v jejím průběhu došlo k migraci poskytované služby a ke změně autentizace způsobu volání. Knihovna[17], která toto napojení původně zprostředkovávala z tohoto důvodu není v současné době použitelná, dokud nedojde k její úpravě. Vzhledem k tomu, že značná část vývoje včetně testování proběhla na tomto překladači a může dojít k aktualizaci knihovny pro napojení, analýzu zde ponechám.

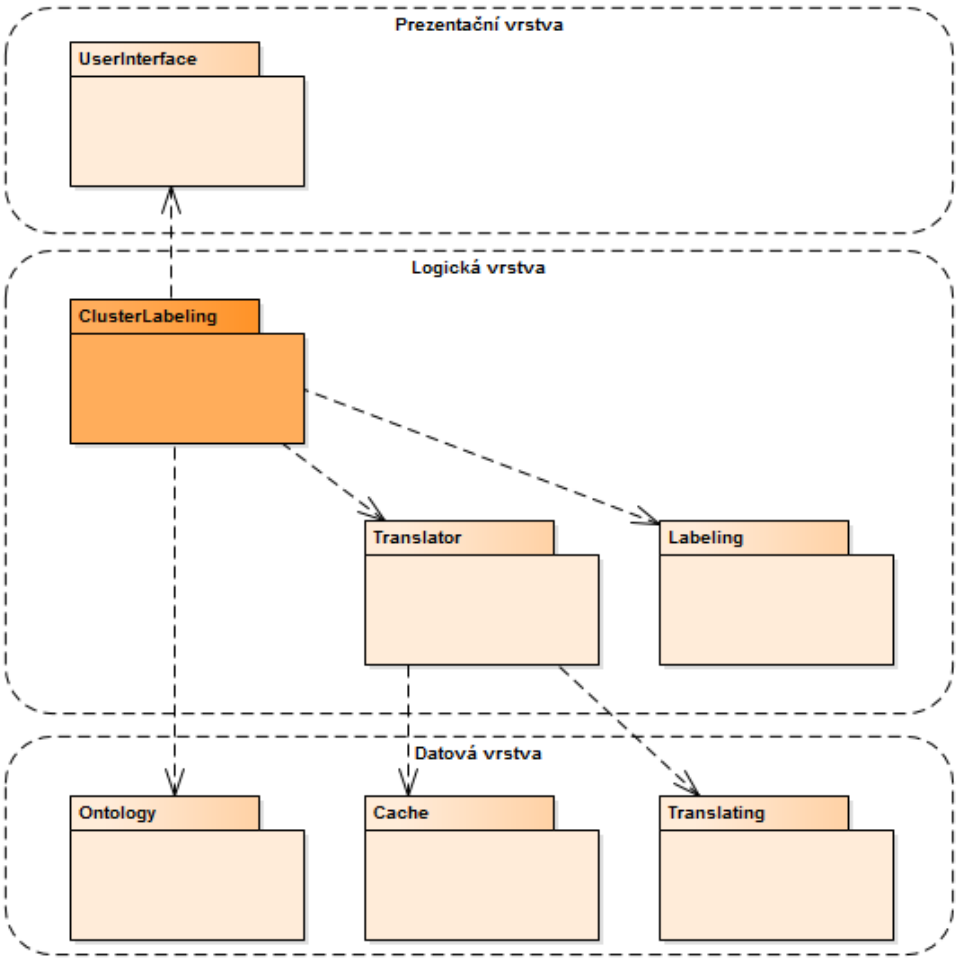
2.3 Google Translator

Stejně jako v předchozím případě poskytuje společnost Google v rámci svých cloudových služeb i rozhraní pro automatizovaný překlad. Není však možné jej využívat neomezeně a bezplatně. Základní sazba pro stejné využití jako u Microsoft Bing Translator, dosahuje 20 dolarů za měsíc. Nicméně zároveň Google poskytuje v rámci vyzkoušení služeb kredit ve výši 300 dolarů pro volné použití a tak nám dovoluje po registraci používat API překladače po dobu 15 měsíců, pokud zároveň dodržíme i druhou podmínku, a sice maximálně 2 milióny znaků za měsíc.

Pro využití služby je potřeba pouze „apiKey“, který můžeme získat vygenerováním v rámci administrace vlastního účtu na Google Cloud Platfotm. Vygenerovaný klíč, který jsem použil v rámci vývoje a testování, jsem ponechal v konfiguračním souboru aplikace, a je tak možné jej využít. Případně je možné si vytvořit vlastní účet a vygenerovat jiný klíč.

2.4 Návrh aplikace

Z výše uvedených požadavků plyne rozdělení na jednotlivé moduly či balíčky, které je znázorněno na obrázku 2.1. Balíček *ClusterLabeling* představuje v aplikaci sdílené prvky a základní logické funkční části. Balíček *Labeling*



Obrázek 2.1: Rozdělení aplikace na jednotlivé moduly

pak obsahuje popisovací algoritmy a ve výsledku může využívat služeb balíčku *Ontology*, který představuje jednotlivé databáze externích znalostí. *Translator* je balíček zajišťující překlad a vždy využívá služeb balíčků *Translating*, což jsou samotné překladače. *Cache* je pak v samostatném, stejnojmenném balíčku. Posledním balíčkem na obrázku je zde jediný zástupce z prezentační vrstvy – $UserInterface$, který obsahuje části aplikace nutné pro komunikaci s uživatelem.

Jak je vidět z obrázku, jednotlivé balíčky nepřesahují příslušné vrstvy z třívrstvého modelu, a nadále tak můžeme mluvit o prezentační, logické a datové vrstvě, stejně jako o jednotlivých balíčcích, přičemž se pouze snižuje míra abstrakce návrhu.

Vzhledem k tomu, že mezi nefunkčními požadavky je uvedena snadná výměna popisovacích algoritmů, a zároveň možnost využití různých překladačů,

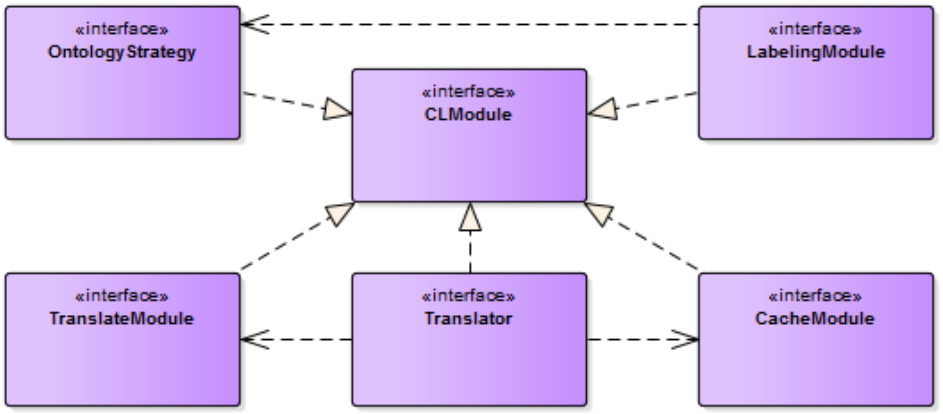
máme dvě možnosti, jak k tomuto problému přistupovat, a to buď:

- pokaždé přidat balíček s algoritmem nebo s překladačem a poskytnout ho balíčku *ClusterLabeling* tak, aby ho mohl uživatel využít, nebo
- definovat balíčky jako jednotlivá rozhraní, která by přidávané algoritmy implementovala.

S první možností bychom do aplikace zavedli vysokou provázanost mezi jednotlivými balíčky a přišli bychom tak o jednoduchou udržovatelnost, což je v přímém rozporu s osvědčeným přístupem k vývoji z praxe. Naproti tomu je druhá možnost ukázkou jednoho z návrhových vzorů [24], konkrétně se jedná o návrhový vzor strategie, což je v tomto případě jeden z vhodných přístupů.

Využijme-li návrhový vzor strategie pro vhodné balíčky (*Labeling*, *Ontology*, *Cache*, *Translator* a *Translating*), můžeme tyto balíčky nahradit rozhraním. Je velmi pravděpodobné, že minimálně v jednom případě (*Translator*) by bylo vhodné kromě signatur metod definovat i jejich tělo. A tak, oproti doporučení v [24], si dovolím vyměnit v návrhovém vzoru strategie interface za abstraktní třídu. Nicméně pro lepší čitelnost následujícího obrázku 2.2, zobrazujícího strukturu aplikace však předpokládejme, že bude diagram obsahovat pouze rozhraní. Stejné záměny rozhraní za abstraktní třídu se dopustím dále v následujícím textu, nicméně pouze v rámci této kapitoly. V rámci kapitoly Realizace je uveden diagram s abstraktními třídami, a zároveň zde již budu s pojmy rozhraní a abstraktní třída zacházet dle obecných zvyklostí.

Dále je ze zkušeností velmi vhodné, aby všechna aktuálně navržená rozhraní implementovala společné, nadřazené rozhraní, byť by neobsahovalo žádnou signaturu metody a bylo tak vlastně prázdné. To nám minimálně umožní říct, co je balíčkem na úrovni celé aplikace a co je naopak „nepodstatným“ rozhraním uvnitř balíčku. V našem případě bychom toho mohli využít pro registraci různých modulů a jednoduše tak uživateli dynamicky předávat informaci o dostupných modulech. Pokud bychom například chtěli přidat modul pro zobrazování obrázků štěňátek nebo modul pro extrahování skupiny slov z dokumentů, stačí aby takový modul implementoval společné rozhraní, na základě kterého je možné tento modul pomocí reflexe dynamicky vyhledat. Následně ho uživatel bude moci využít, aniž by to bylo překážkou pro využití dalšího návrhového vzoru, kterým je řetěz odpovědnosti. To nám umožní využít aplikaci s GUI, kde si uživatel postupně poskládá „krabičky“, reprezentující jednotlivé moduly, sestaví si tak frontu, kterou musí slova projít a určí jakým způsobem budou zpracována. Například si může uživatel sestavit postupně řetěz Vstup–Translator–Labeling–Translator–Výstup a u Translatoru kromě metody pro překlad a cachování vždy definovat vstupní a výstupní jazyk. Následně by například mohl před Translator vložit „Zpracovat dokumenty“, a tak zpracování změnit, bez jakékoliv znalosti programování. V případě, že by chtěl uživatel přidat modul, který ještě nikdo jiný nevytvořil, stačí, když bude implementovat dané rozhraní a může modul využít.



Obrázek 2.2: Definovaná rozhraní v rámci aplikace

Z tohoto důvodu je na diagramu 2.2 vytvořeno další rozhraní $CLModule$, které všechna ostatní rozhraní implementují. Rozhraní $CLModule$ je součástí balíčku $ClusterLabeling$, ostatní je pak možné zařadit na základě podobnosti názvu.

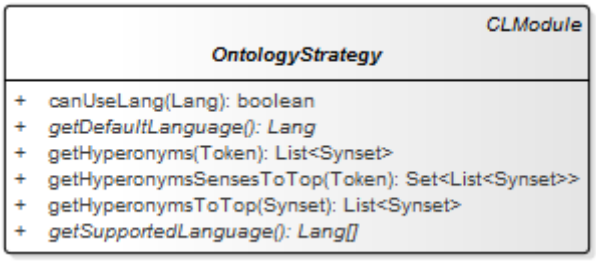
2.5 Rozhraní OntologyStrategy

Jedním z prvních funkčních požadavků ve specifikaci aplikace je využití dvou externích databází znalostí – anglický WordNet a Microsoft Concept Graph. Obě databáze jsou určitou reprezentací ontologie a z tohoto hlediska mají mnoho společného. Proto můžeme navrhnout rozhraní takovým způsobem, aby poskytovalo potřebné metody pro popisování skupin slov. Abychom mohli popisovat skupinu slov, musíme v rámci ontologie mít přístup k nadřazeným prvkům (metody $getHyperonyms(\ldots)$), podřazeným prvkům (metody $getHyponyms(\ldots)$) a případně k synonymům (metody $getSynonyms(\ldots)$).

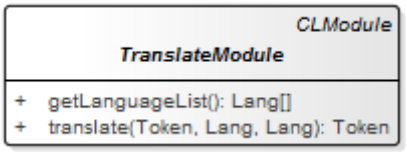
Jak již bylo řečeno v kapitole 1.5.1, každé slovo může mít více různých významů a i když se metoda $getHyperonymsSensesSet(\ldots)$, která vrací seznam nadřazených prvků pro všechny významy slova, využije pouze v případě databází typu WordNet, dovolím si ji zařadit do definice rozhraní, stejně jako následující dvě metody.

V případě kompletní stromové struktury, jakou je právě WordNet, může být vhodná i metoda $getHyperonymsToTop(\ldots)$, která by poskytovala seznam hyperonym od zadaného slova až ke kořenu stromu, a to pouze pro jeden význam slova. Oproti tomu metoda $getHyperonymsSensesToTop(\ldots)$ je kombinací obou předchozích metod, která vrací seznam listů hyperonym až ke kořenu, a to pro všechny významy slova.

Významy tříd $Synset$ a $Token$, které jsou předávány definovaným me-



Obrázek 2.3: Návrh rozhraní *OntologyStrategy*



Obrázek 2.4: Návrh rozhraní *TranslateModule*

todám, jsou popsány dále v podkapitole Přepravky vstupních a výstupních hodnot.

Kromě metod, které úzce souvisí s ontologií a jsou popsány výše, obsahuje rozhraní metody pro definici jazyka. Konkrétně se jedná o metody pro zjištění, které jazyky ontologie podporuje, nebo lépe řečeno, ve kterém jazyce se daná ontologie nachází.

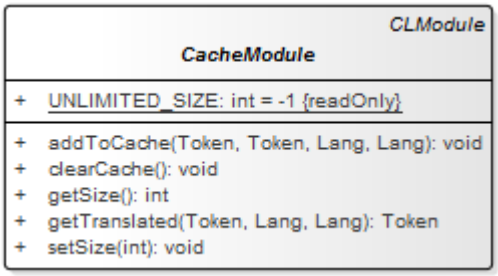
2.6 Rozhraní TranslateModule

Dalším požadavkem ve specifikaci je využití služeb automatického překladu, pro který je určeno rozhraní *TranslateModule*, obsahující pouze dvě metody. První metodou je *getLanguageList*(), která vrací seznam všech jazyků, které je možné využít pro překlad. Ve všech případech by implementace měla podporovat český a anglický jazyk.

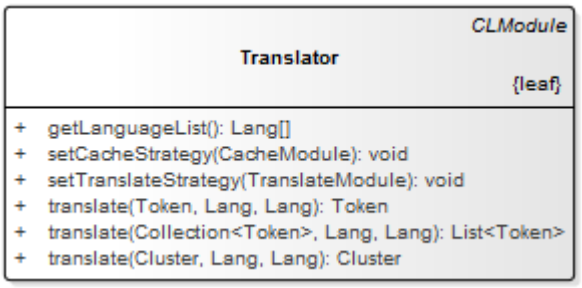
Druhou metodou je *translate*(...), která zajišťuje samotný překlad slova obsaženého v parametru třídy *Token*. Následují parametry jazyků, kde prvním parametrem je zdrojový jazyk a druhým parametrem je cílový jazyk překladu.

2.7 Rozhraní CacheModule

Požadavek na omezení počtu přístupů k překladači je splnitelný vcelku jednoduše. Jak již bylo naznačeno, je možné využít formu cache. Pro práci s cache jsou pak potřeba standardní metody (*addToCache*(...) a *getTranslated*(...))



Obrázek 2.5: Návrh rozhraní *CacheModule*



Obrázek 2.6: Návrh rozhraní *Translator*

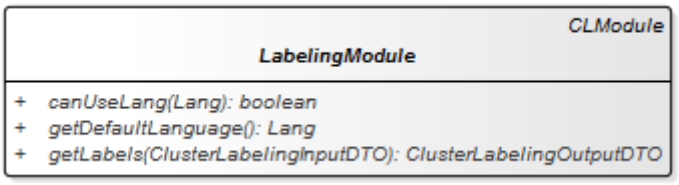
pro vložení a získání objektu do cache a z ní. Jelikož je více než pravděpodobné, že cache bude uložena přímo v paměti, je dobré mít možnost omezit její velikost. Pro takové případy slouží metody *getSize*() a *setSize*(...) a konstanta *UNLIMITED_SIZE* určující, že cache není nijak omezena.

2.8 Rozhraní Translator

Rozhraní *Translator* by teoreticky nemuselo být ani rozhraním, ale konkrétní třídou, nicméně z hlediska správného OOP návrhu je vhodné ho vytvořit. Toto rozhraní využívá další dvě předchozí rozhraní pro konkrétní překlad a cachování výsledků, což je jeho hlavním účelem – poskytovat překlad nejprve z cache, pokud je dostupná, a v případě nenalezení v cache z překladače. Právě kvůli možnosti využít více různých překladačů je vhodné definovat v tomto místě rozhraní namísto konkrétní třídy.

2.9 Rozhraní LabelingModule

Posledním rozhraním, které je třeba navrhnout, je *LabelingModule*, které bude obstarávat samotné popisování skupin slov. Toto rozhraní obsahuje metodu *getLabels*(...), která vstupní množinu slov popisuje a je tak asi nejdů-



Obrázek 2.7: Návrh rozhraní *LabelingModule*

ležitější metodou v celé aplikaci. Kromě popisovací metody obsahuje rozhraní ještě metody pro předání typu ontologie, pomocí které má algoritmus slova popsat.

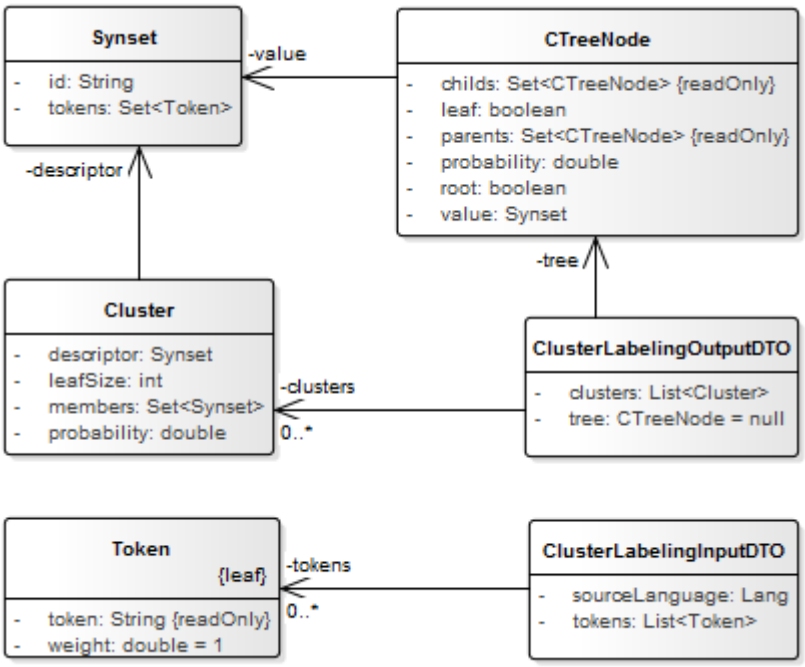
Z předchozího odstavce a popisu rozhraní *OntologyStrategy* je patrné, že správný návrh rozhraní by měl obsahovat ještě jedno rozhraní, abstrahující právě *OntologyStrategy*, které může implementovat jen externí databáze, jež není ontologií. Takové abstraktnější rozhraní by pak bylo předáváno rozhraní LabelingModule a mohlo by tedy využít i jiné typy znalostních databází, než kterými jsou ontologie. Nicméně tato vlastnost návrhu je dle mého názoru namístě, neboť zvyšuje přehlednost kódu i návrhu, a zároveň přidání takového „mezirozhraní“ není při refaktoringu ničím zásadním. Z tohoto důvodu jsem se rozhodl tuto vlastnost v návrhu ponechat a pouze na ni upozornit možné budoucí rozšiřovatele aplikace.

2.10 Sdílené prvky modulů

Již v tuto chvíli je jasné, že bude nutné, aby některé moduly měly možnost získat parametry z konfiguračního souboru, případně ze vstupních parametrů. Dále je jisté, že vstupní skupina slov a některé další atributy se budou předávat mezi jednotlivými moduly. Z těchto důvodu je nutné se zamyslet nad možnostmi, jak situaci vyřešit již nyní v rámci návrhu aplikace tak, aby se předešlo zbytečnému refaktoringu pokaždé, když dojde k jakékoliv, byť i drobné změně.

2.10.1 Přepravky vstupních a výstupních hodnot

Jednodušší řešení má v tuto chvíli problém předávání skupin slov mezi moduly. Pro takovou situaci je návrhový vzor přepravka [24] asi nejvhodnějším řešením. Nicméně vstupní přepravka by měla být odlišná od výstupní, a to kvůli různým atributům, které udané přepravky mají předávat. Respektive data se ze vstupní do výstupní přepravky transformují přesně v modulu pro pojmenovávání slov. Na diagramu tříd zobrazeném na obrázku 2.8 jsou uvedeny nejen přepravky (*ClusterLabelingInputDTO* a *ClusterLabelingOutputDTO*), ale



Obrázek 2.8: Návrh tříd (DTO) pro předávání atributů

i pomocné třídy. Diagram naopak neobsahuje metody, které dané třídy obsahují.

Vstupní přepravka *ClusterLabelingInputDTO* obsahuje *List* objektů třídy *Token*, která reprezentuje jedno vstupní slovo včetně jeho váhy. V případě třídy *Token* by bylo možné využít i standardní implementaci řetězce v Java, a sice třídu *String*, nicméně osvědčená praxe v tomto případě velí „obalit“ slovo do vlastní třídy, aby bylo možné případně přidávat další parametry. Bylo by možné uvažovat například o jazyku, ve kterém je dané slovo, případně by třída *Token* mohla obsahovat překlady do jednotlivých jazyků. Nicméně v tuto chvíli předpokládáme, že všechna vstupní slova jsou v jednom konkrétním jazyce, a tak je jazyk uveden již ve vstupní přepravce.

Výstupní přepravka *ClusterLabelingOutputDTO* obsahuje list *clusters*. Třída *Cluster* obsahuje podmnožinu vstupních slov v parametru *members* a jejich popisné slovo. Jak vstupní slova, tak popisná slova jsou objekty třídy *Synset*, která odpovídá názvem i významem synsetu v databázi WordNet. Druhým parametrem, který třída *ClusterLabelingOutputDTO* obsahuje, je *tree*, což je kořen stromu popisné ontologie a implementuje jej třída *CTreeNode*, o které pojednává samostatná podkapitola Strom synsetů.

2.10.2 Konfigurační parametry

Relativně složitějším problémem, než jsou přepravky vstupních a výstupních hodnot, jsou konfigurační parametry. Můžeme předpokládat, že každý modul může ke své práci potřebovat určité konfigurační parametry, kterým může být například v kapitole Google Translator zmiňovaný „api-key“. Takové konfigurační parametry není vhodné předávat pomocí přepravky vstupních hodnot, ať už z hlediska logického rozdělení, nebo z hlediska principu zapouzdření.

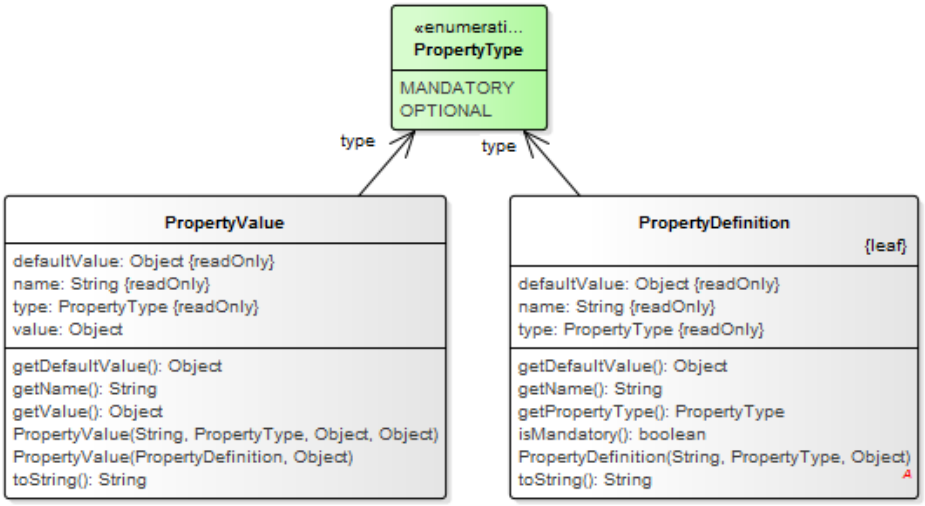
Je tedy nutné kromě přepravek předávat modulům i tyto konfigurační parametry. Jedním ze způsobů, jak takové parametry zpřístupnit, je použít konfigurační soubor, který bude globálně dostupný v celé aplikaci. Ovšem obtížně se v takovém případě řeší nejen to, které parametry jsou povinné a které volitelné, ale případně i jejich výchozí hodnota. Zároveň je opět problematické řešit princip zapouzdření, což v tomto případě není kritická záležitost, ale obecně praxe radí neposkytovat zbytečně citlivá data, a to ani v rámci kódu jedné aplikace.

Z tohoto důvodu jsem navrhl způsob předávání parametrů, které všechny tyto popsané problémy řeší a navíc umožňuje v případě GUI dynamicky uživateli sdělovat, co z povinných atributů zbývá doplnit ke správné funkčnosti. Nicméně vynechat samotný konfigurační soubor je velmi nevhodné. Vhodnější variantou, kterou jsem zvolil je omezit jeho dostupnost pouze na jádro aplikace, které bude parametry předávat jednotlivým modulům. Každá z implementací jednotlivých modulů si může specifikovat, které parametry jsou pro spuštění povinné a které nepovinné. Na diagramu 2.9 jsou uvedeny dvě třídy, kde první z nich ($PropertyDefinition$) je definicí parametru a druhá ($PropertyValue$) pak představuje zadanou hodnotu.

Každá implementace libovolného rozhraní, které je uvedeno na obrázku 2.2, pak definuje seznam parametrů a jejich přítomnost. Zde se ukazuje, že volba abstraktních tříd namísto klasického rozhraní, byla vhodná, neboť každá abstraktní třída před samotným voláním definované metody kontroluje, zda jsou všechny parametry již definovány, což navíc umožní kontrolovat pouze ty parametry, které jsou potřeba pouze pro volání dané metody. Jestliže není dostupná hodnota definována uživatelem, použije se výchozí hodnota v definici parametru. Pakliže tato hodnota není dostupná a zároveň se jedná o povinný parametr, je vyvolána výjimka $CLException$.

2.10.3 Strom synsetů

Jelikož budeme v rámci aplikace pracovat převážně s popisovacími algoritmy založenými na ontologii, bude zcela jistě nutné vytvořit jistou reprezentaci stromové struktury, podobnou struktuře WordNetu, po které bychom se mohli pohybovat směrem k hyperonymům i hyponymům. Pro tyto účely jsem vytvořil třídu uvedenou na diagramu 2.10 s názvem $CTreeNode$, což je poněkud zavádějící označení, neboť tato třída reprezentuje třídu orientovaného grafu.



Obrázek 2.9: Diagram tříd pro konfigurační parametry

Ten ale nutně nemusí být stromem, jelikož jeden uzel může obsahovat více potomků i rodičů, nicméně všechny směřují k jedinému uzlu – kořeni „stromu“. Kromě klasických metod pro přidání potomků i rodičů obsahuje třída samozřejmě atribut typu Synset, což představuje hodnotu uzlu. Dále pak obsahuje dva atributy – $leaf$ a *root* a přístupové metody pro ně. Atribut $leaf$ značí, že se jedná o jedno z popisovaných slov, což opět může být v některých případech zavádějící, jelikož popisované slovo může mít jako rodiče jiné popisované slovo a listem stromu je pouze jedno z nich. Atribut *root* pak určuje, zda se jedná o kořen stromu, či nikoliv.

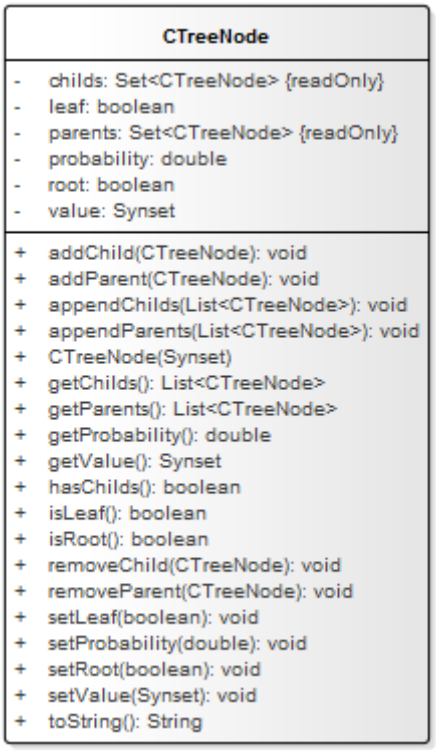
Zcela jistě bude nutné pro potřeby práce s třídou vytvořit ještě utilitní třídu, nicméně to se ukáže až v rámci realizace.

2.10.4 Systém výjimek

Dalším prvkem, převzatým z praxe je definice vlastního systému výjimek, které nám umožní lépe reagovat na chybové stavy způsobené během programu. Navíc v případě zapojení do většího programu umožní uživateli lépe identifikovat, kde nastal problém. Z těchto důvodu jsem navrhl vytvoření třídy $CLException$, která umožní zabalení všech ostatních výjimek, a to včetně doplnění vlastní zprávy k chybě.

2.11 Diagram návrhu

Kompletní diagram návrhu aplikace včetně všech veřejných metod, je vyobrazen v příloze na obrázcích C.1 a C.2.



Obrázek 2.10: Diagram třídy *CTreeNode*

Kapitola 3

Realizace

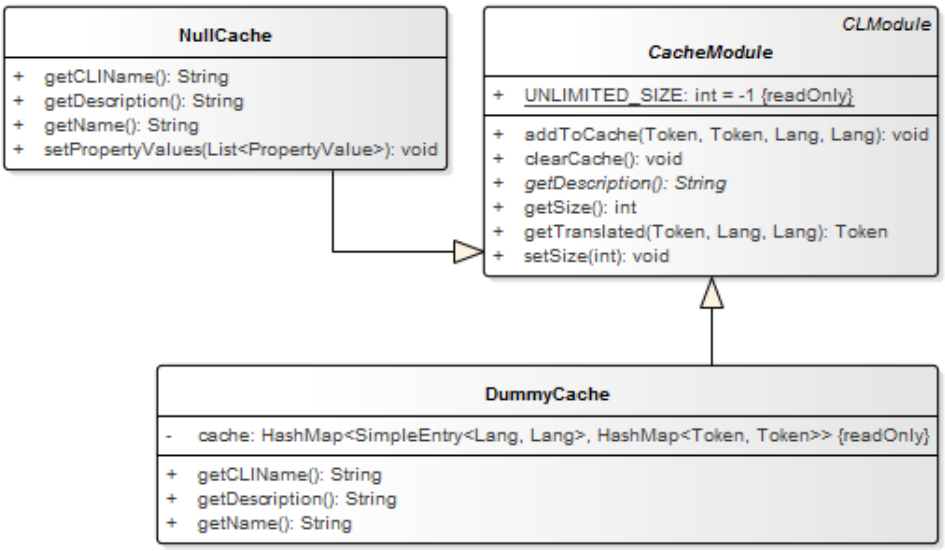
V této kapitole popisuji realizaci celé aplikace, jejíž návrh jsem popsal v předchozí kapitole, a zároveň již zde budu dodržovat rozlišení mezi rozhraním a abstraktní třídou. V řadě případů, například při využití návrhových vzorů, se návrh ukázal jako vhodný, a nebylo tak nutné do něj zasahovat. Pouze v několika případech došlo k rozšíření počtu poskytovaných metod v rámci abstraktní třídy tak, aby komunikace a vykonávání byly efektivnější.

3.1 Maven

Pro sestavování aplikace jsem ve své práci použil nástroj Maven[20], který mi v průběhu vytváření často umožňoval urychlení vývoje. Díky tomuto nástroji vytvářím nejenom spustitelný soubor JAR, ale také dokumentaci v podobě standardního nástroje JavaDoc a výstupy z dvou testovacích frameworků pro statickou analýzu kódu, které popisuji v kapitole Testování.

V neposlední řadě jsem mohl díky Mavenu velice jednoduše spravovat potřebné knihovny pro sestavení celé aplikace, proto jsem je v řadě případů využil. Avšak řada z těchto knihoven je v práci využita jen z části, přestože nabízí mnohem více možností. I přes tento fakt, jsem se rozhodl pro jejich použití namísto vytváření funkčnosti ve vlastním kódu, neboť to zcela odporuje zavedenému a ověřenému pravidlu z praxe, které říká, že opakování kódu (v tomto případě stejného jako v uvedených knihovnách) je cesta, jejímž výsledkem je neudržitelnost aplikace. Naproti tomu je však nutné při aktualizaci knihovny vždy otestovat celou aplikaci, což v řadě případů může znamenat velké náklady na udržování aplikace. I tato situace je ovšem řešitelná a sice za pomoci automatizovaného testování.

Z tohoto důvodu jsem se zároveň s rozhodnutím využít Maven, rozhodl využití automatizovaného testování a statické analýzy kódu tak, aby každý, kdo bude pokračovat v rozšiřování a udržování aplikace nemusel tyto náklady, ať už v podobě času, nebo v podobě peněz, řešit.



Obrázek 3.1: Diagram potomků abstraktní třídy *CacheModule*

3.2 Implementace cache

Při vývoji aplikace jsem se rozhodl pro implementaci dvou potomků abstraktní třídy *CacheModul*. První z nich je tzv. „Null objekt“, neboli prázdný objekt a druhý je typickou implementací cache s prvky v paměti. Na diagramu 3.1 je vyobrazena část implementovaného řešení, kde jsou záměrně vypsány nejen veřejné metody, ale i metody s modifikátorem přístupu *protected*. Abstraktní třída *CacheModule* implementuje veřejné metody tak, že volá abstraktní *protected* metody se stejným názvem a přidaným sufixem *Inner*. Tyto veřejné metody pak kontrolují, zda jsou všechny potřebné konfigurační parametry pro běh konkrétní implementace nastaveny.

3.2.1 NullCache

Návrhový vzor „Null objekt“ je založen na myšlence, že je vždy lepší poskytnout alespoň něco místo ničeho. Takový objekt je náhradou za hodnotu null proto, abychom se vyhnuli zbytečné chybě při vykonávání, a jedná se tedy o jakési „šidítko“. Na takovém objektu je sice možné volat všechny metody definované v rozhraní, ale ve skutečnosti se nic nestane. Můžeme tedy mít takovou cache kdekoliv v kódu a volat vložení do cache a vybrání z cache. První metoda skončí ihned a nevykoná žádné instrukce, druhá způsobí standardní stav, kterým je nenalezení záznamu v cache.

Dodržíme-li pak v třídě *Translatoru* kontrakt tím, že místo null hodnoty vždy doplníme tento objekt, není nutné při volání libovolné cache řešit, zda je cache k dispozici a můžeme ji přímo volat bez obav, že nastane chyba.

3.2.2 DummyCache

Druhou implementací cache je třída *DummyCache*, která ukládá výsledky překladu do paměti, a to bez omezení velikosti. Jak již bylo určeno v návrhu, metoda *addToCache*(...) má na vstupu slovo ve dvou různých jazycích a určené oba tyto jazyky. Z tohoto důvodu jsem se rozhodl použít obě slova a udělat oboustrannou cache, kdy při dotazu na překlad z jednoho či druhého jazyka vrátí metoda *getTranslated*(...) příslušný výsledek. Navíc *DummyCache* nemá v konstruktoru, ani v žádné metodě určeno, že se používá pro exkluzivní dva jazyky, a tak při násobném volání metody *addToCache*(...) s různými jazyky nezpůsobí chybu, ale všechny tyto překlady se do cache uloží.

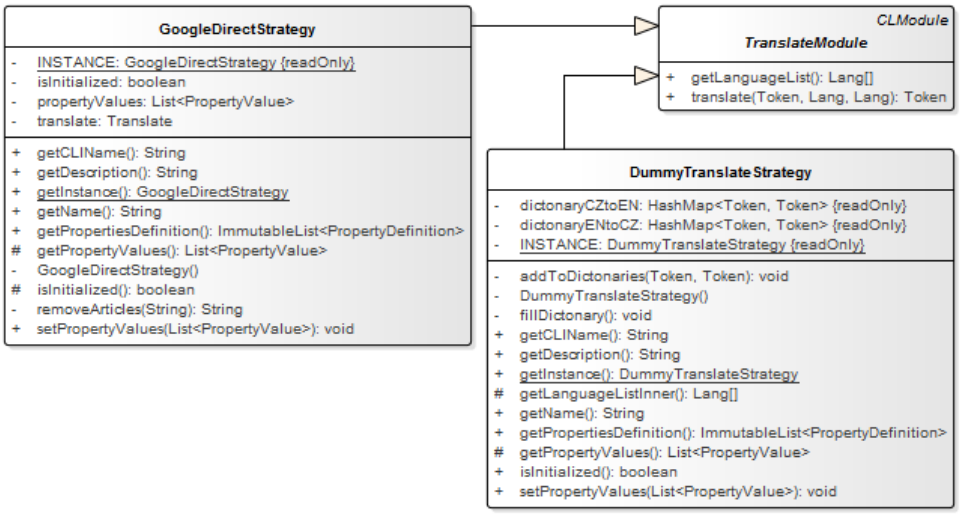
Jedná se však o velmi jednoduchou cache, tudíž obsahuje možnost přeložit slovo pouze pro dva různé jazyky. Pakliže by cache obsahovala záznamy:

- car (EN) – auto (CZ)
- auto (CZ) – wagen (GER)

bylo by teoreticky možné na dotaz *getTranslated*("*car*", "*EN*", "*GER*") vrátit token „wagen“. Nicméně toto nebyl účel třídy *DummyCache*, a proto toto není implementováno, mimo jiné také z důvodu, že v drtivé většině případů bude nutné použít překlad z jednoho jazyka do druhého, a tak dotazy, a to včetně vkládání do cache, budou pouze dvoujazyčné.

Jako další možné rozšíření této cache bych viděl možnost uchovávání hodnot i po ukončení běhu aplikace, ať už pomocí perzistence objektu na disk v podobě souboru, nebo pomocí ukládání do databáze. To by snížilo počet dotazů na online překladače, a to hlavně v případě vývoje a testování různých algoritmů, kdy se nebude měnit vstupní skupina slov, a zároveň bychom nepřišli ani o rychlost vykonání dotazu. Museli bychom ovšem využít i omezení počtu položek uchovávaných v paměti, a zároveň vytvořit mechanismus, kdy dojde k odstranění položky z paměti. Jde například o metodu nejméně používaného nebo nejstaršího záznamu.

Zároveň si dokáži představit určitý „mód učení“, kde by si cache uchovávala vložená data, ale nevydávala by žádné hodnoty. Pokud by taková cache byla použita v rámci vícero překladačů, mohla by si tak vytvořit určitý druh statistiky, kde by si vytvářela vlastní shluky slov a k nim určitý překlad. Následně by pak na dotazy odpovídala z jednoho shluku překladů a na jinou sadu dotazů zase z jiného shluku, čímž by si kromě samotného překladu uchovávala i kontext. To by v případě velkého množství dat mohlo vést k vylepšení překladu, nicméně se domnívám, že k takovému chování by bylo potřeba velkého množství učících dat, a zároveň je diskutabilní, zda se ještě stále jedná o cache, nebo již o samotný překladač.



Obrázek 3.2: Diagram potomků abstraktní třídy *TranslateModule*

3.3 Implementace překladačů

Stejně jako v případě cache jsem se u překladačů rozhodl implementovat dva překladače, kde první z nich je velmi zjednodušený a druhý, využívající online překladu za pomoci Google Translator, je již komplexní.

Kromě dvou níže uvedených možností by bylo možné využít i jiné překladače, ať už v podobě online dotazů, nebo slovníku uloženém na disku. Nicméně přínos pro zlepšení kvality překladu dle mého názoru spočívá ve znalosti kontextu, jak jsem již zmínil v předchozí podkapitole DummyCache. Základní možností jak využít kontext v případě překladu, je předložit překladači všechna překládaná slova v jednom dotazu. Tím překladač získá alespoň nějakou možnost si kontext z těchto slov odvodit, což povede k lepšímu překladu. Toto je však použitelné pouze v případě vstupní skupiny slov. Pokud bychom chtěli přeložit část podstromu WordNetu, ztráceli bychom kontext s každým slovem, které je nadřazené předchozímu, až by se s přidáním nejvyššího slova „element“ kontext vytratil úplně.

Řešením by teoreticky mohlo být předložit překladači vždy pouze několik slov, která jsou nadřazená a kontext ještě částečně udržují. Nicméně domnívám se, že tato problematika přesahuje rámec této práce a je to spíše problémem celého vědního odvětví lingvistiky.

3.3.1 DummyTranslateStrategy

Implementace velmi zjednodušeného překladače obsahuje dva slovníky v podobě *HashMap*, které obsahují stejná data, pouze s obrácenými klíči a hodno-

tami z důvodu rychlosti. Na dotaz o překlad slova nejprve ověří, že zdrojový i cílový jazyk je podporovaný tímto slovníkem a následně se jej pokusí vyhledat. Pakliže jej v příslušném slovníku nenajde, vrací hodnotu *null*.

Kvůli redundantním datům v podobě dvou stejných slovníků a zároveň faktu, že změna slovníků je možná jen v případě překódování, je tento překladač implementován jako návrhový vzor jedináček (Singleton).

3.3.2 GoogleDirectStrategy

V případě implementace překladče Google Translator je situace poněkud složitější z toho důvodu, že komunikace probíhá online pomocí dotazů na URL obsahující kromě jiného i překládané slovo a JSON odpovědi obsahující přeložené slovo. Navíc je nutné se autorizovat za pomocí zmíněného „apikey“, jelikož se jedná v placenou službu.

Vzhledem ke složitosti, která není zcela jistě v rozsahu této práce, jsem využil knihovnu[10], poskytovanou přímo od společnosti Google, která je vydána pod licencí Apache v2.0. Tato knihovna řeší veškerou komunikaci se servery společnosti Google a z hlediska kódování tak bylo nutné vytvořit pouze obalovací třídu. Dalo by se říci, že se jedná o návrhový vzor adaptér, který má vlastní implementaci tak, aby splnil rozhraní abstraktní třídy $TranslateModule$.

Kvůli tomu, že by bylo vhodné mít komunikaci přes tuto knihovnu pod kontrolou, a zároveň knihovna nemění svůj vnitřní stav při dotazování na překlad různých slov, je zcela jistě na místě implementovat tuto třídu jako jedináčka (Singleton).

3.4 Napojení na externí databáze znalostí

Napojením na externí databáze znalostí je v obou případech implementace abstraktní třídy *OntologyStrategy*, o které jsem se zmiňoval v předchozí kapitole. Tentokrát jsou obě dvě implementace již komplexnější, nicméně třída $MicrosoftConceptOntology$ by se dala z hlediska rozsahu považovat za jednodušší, nikoliv však za méně funkční či bezvýznamnou.

3.4.1 MicrosoftConceptOntology

Napojení na Microsoft Concept Graph je možné za pomocí dvou různých přístupů, stejně jako v případě WordNetu. Prvním je stažení databáze a její použití lokálně, nebo využití online API, které je dostupné po registraci. V tomto případě jsem se rozhodl pro využití online API, jelikož se databáze neustále rozrůstá, na rozdíl od WordNetu, který již jistou dobu nezaznamenal žádný výrazný rozvoj. Druhým důvodem je pak relativně jednoduchá implementace komunikace s API, neboť na základě metody GET na URL adresu, která obsahuje i dotazované slovo, případně jiné parametry, vrací API ve standardním

použití JSON zprávu. Tuto JSON zprávu následně program zpracovává pomocí velmi rozšířené knihovny JSON-JAVA[14] a práci s komunikací tak velmi usnadňuje.

Příchozí odpověď na dotaz obsahuje 10 nejpravděpodobnějších nadřazených slov i s uvedenou mírou pravděpodobnosti. Ovšem pokud se mezi slovy, která jsou uvedena v odpovědi vyskytuje určitý typ vazby, není ve zprávě nijak uveden. To vede k tomu, že na volání metody $getHyperonyms(\ldots)$ je vrácen pouze seznam slov uvedených v odpovědi. Společně s faktem, že se jedná o orientovaný graf, který rozhodně není stromem, nás tyto vlastnosti nutí vytvořit si syntetický kořen. Ten jsem pro účely popisování nazval stejně jako kořen WordNetu, tedy „element“. Při volání metody $getHyperonymsToTop(\ldots)$ je výsledkem množina deseti seznamů o dvou slovech. Prvním z nich je slovo navrácené z provolání API a druhým pak zmiňovaný kořen „element“.

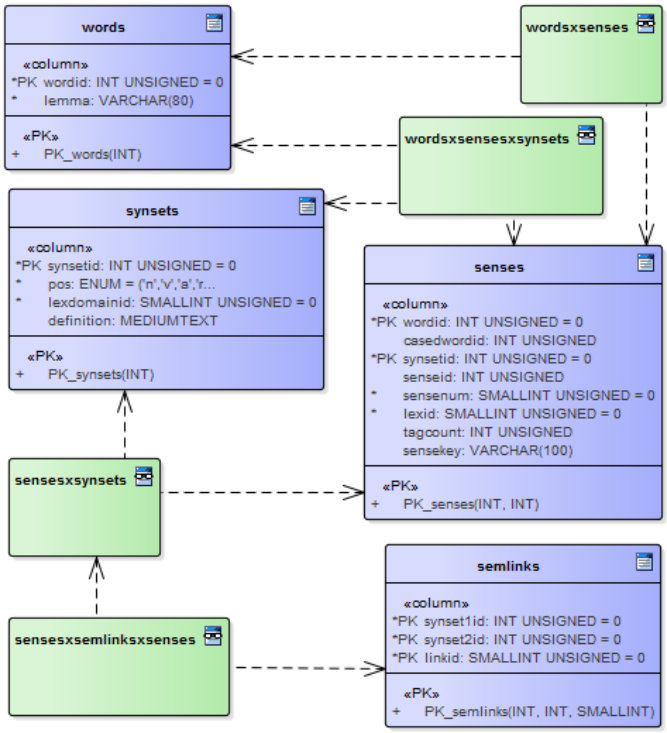
3.4.2 WordNetOntology

Jak již bylo v předchozím textu zmíněno, nejvhodnějším způsobem, jak využít databázi WordNet v rámci mé aplikace je využití lokální databáze. Místo parsování a nahrávání souborů WordNet, které si uživatel může stáhnout z oficiálních stránek, existuje možnost získat snímek MySQL databáze připravený k použití. Jedním z takových snímků, dostupných na internetu je WordNet-SQL, který obsahuje aktuálně WordNet ve verzi 3.1, tedy poslední, která je zároveň vydána pod stejnou licencí jako WordNet.

Napojení na tuto databázi je pak možné pomocí standardního MySQL konektoru a v rámci implementace této aplikace je nutné řešit pouze správnost SQL dotazů. Snímek databáze, jehož část je vyobrazena na obrázku 3.3, obsahuje nejenom tabulky s daty, ale také již vytvořené pohledy, které propojují jednotlivé tabulky. Kromě jiných tabulek, obsahuje tento snímek databáze pro naše potřeby tyto čtyři nejdůležitější tabulky:

- *words* – obsahuje jednotlivá slova a identifikátor jednoznačný v rámci celé databáze
- *synsets* – obsahuje synsety a jejich slovní definici např.: „fruit with red or yellow or green skin and sweet to tart crisp whitish flesh“ pro slovo „apple“
- *senses* – propojení mezi předchozími dvěma tabulkami, které spojuje synset s konkrétnim slovem na základě společného významu
- *semlinks* – tabulka obsahující vazby mezi synsety

Import snímku do lokální MySQL databáze je popsán v části Použití aplikace.



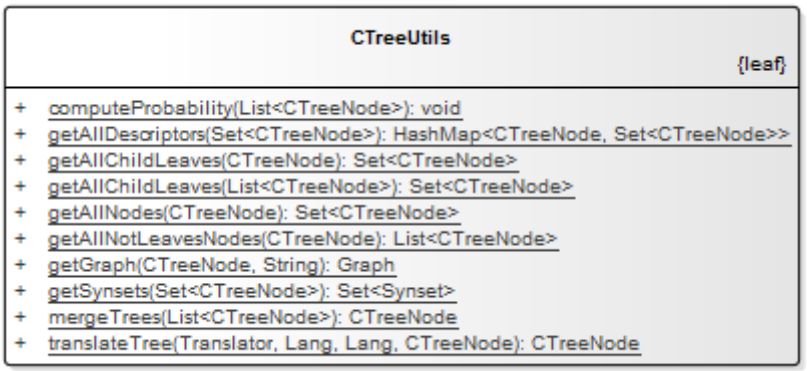
Obrázek 3.3: Část struktury snímku MySQL databáze WordNet

3.5 Utilitní třída CTreeUtils

Pro potřeby práce s objekty typu *CTreeNode* jsem vytvořil utilitní třídu *CTreeUtils*, která obsahuje veřejné a statické metody a není možné jí instancovat, jedná se tedy o návrhový vzor „Knihovní třída“[24]. Všechny poskytované metody je možné vyčíst z diagramu 3.4, takže lze zde uvést pouze dvě z nich, které si dle mého názoru zasluhují pozornost pro lepší pochopení popisovacích algoritmů.

3.5.1 Sjednocování stromů

Rozhraní *OntologyStrategy* bylo navrženo tak, aby pracovalo po jednotlivých slovech a nikoliv s množinou slov, která je na vstupu programu. Algoritmy pro popisování však většinou potřebují znát nejen cestu k vrcholu, ale v ideálním případě podgraf celkového stromu, který je možné sestavit ze získaných cest k vrcholu. Pro tento případ jsem vytvořil metodu $mergeTrees(\ldots)$, která postupně slučuje jednotlivé cesty dohromady a tvoří z nich strom s jediným vrcholem, kterým je v obou případech ontologií slovo element.



Obrázek 3.4: Diagram utilitní třídy *CTreeUtils*

3.5.2 Grafické znázornění stromu

Jelikož bylo při vývoji značně obtížné ladit jednotlivé metody obou popisovacích algoritmů, zvláště pokud jsme chtěli vidět celý strom, který vytvořila metoda *mergeTrees*(...), rozhodl jsem se vytvořit jednoduchou utilitku, která by výsledný strom zobrazila ve formě obrázku. K tomuto účelu velmi dobře posloužil nástroj Graphviz a knihovna *graphviz − java*[9] dostupná opět pod licencí Apache v2.0.

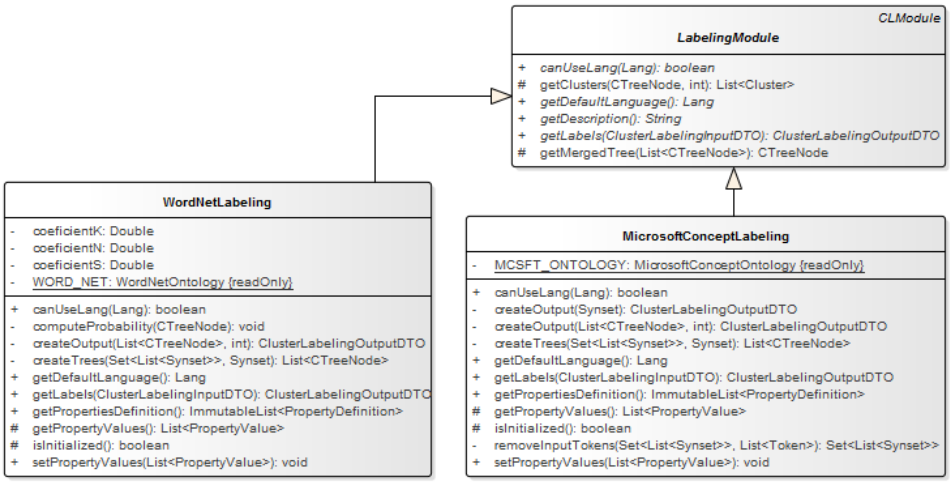
Pomocí této knihovny se mi podařilo transformovat strom, tvořený objekty *CTreeNode* na obrázek, ve kterém je patrná struktura a snadněji se tak ověřuje správnost výsledných algoritmů. Výsledkem takové transformace je pak například graf na obrázku 3.12, kde šipky vedou vždy od hyponyma k hyperonymu a kde zeleně podbarvené prvky jsou vstupní slova. V tomto ukázkovém případu uzly představují synsety z WordNetu, a tak některé z nich obsahují více synonym, ze kterých je zobrazený pouze první z nich, i když nemusí být zrovna nejvýstižnější.

3.6 Popisovací algoritmy

Tato podkapitola pojednává o algoritmech pro popisování skupin slov, tedy o implementacích rozhraní *LabelingModule*, které jsou vyobrazeny na diagramu 3.5.

3.6.1 MicrosoftConceptLabeling

MicrosoftConceptLabeling je potomkem abstraktní třídy *LabelingModule*, tím pádem je představitelem prvního z popisovacích algoritmů. Ke své činnosti využívá služeb poskytovaných třídou *MicrosoftConceptOntology* pro získávání nadřazených prvků.



Obrázek 3.5: Diagram potomků abstraktní třídy LabelingModule

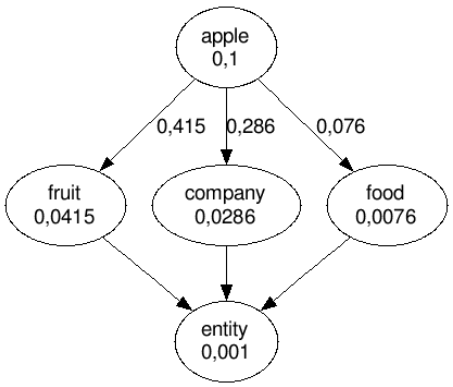
Níže popisovaný algoritmus jsem navrhl a implementoval na základě znalosti struktury externí databáze Microsoft Concept Graph a na základě předpokladu, že hyperonymum, které je společné pro podmnožinu vstupních slov a které je nejvíce používané v různých souvislostech může být aplikovatelným popisem pro zadaná slova.

Vstupem pro popisovací algoritmus jsou slova zadaná uživatelem, včetně jejich váhy. Výstupem je seznam možných popisů, kde každý popis obsahuje:

- algoritmem vybrané popisné slovo,
- podmnožinu vstupních slov, pro které je popisné slovo z předchozího bodu vybráno,
- algoritmem vypočtenou pravděpodobnost, že vybrané slovo z prvního bodu je hledané slovo a
- procentní vyjádření, kolik slov ze vstupní množiny je obsaženo v podmnožině uvedené v druhém bodě.

Popisovací algoritmus postupuje ve třech základních krocích, kde prvním z nich je získání nadřazených prvků pro každé slovo ze vstupní množiny, druhým krokem je výpočet pravděpodobností v rámci získaných nadřazených prvků v prvním kroku a posledním krokem je sloučení výsledků v podobě stromů.

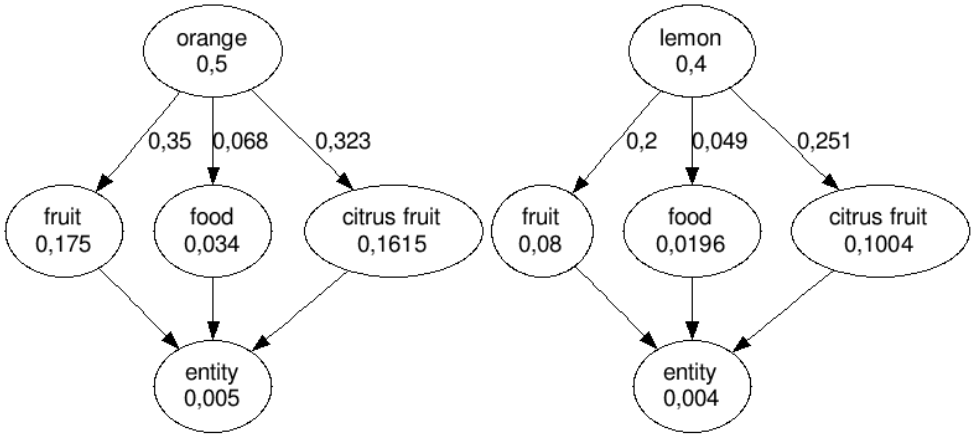
V rámci prvního kroku se pro všechna slova z vstupní množiny volá metoda $getHyperonymsSensesToTop()$ z třídy $MicrosoftConceptOntology$ čímž získáme seznam 10 možných hyperonym daného slova z databáze Microsoft Concept Graph včetně pravděpodobnosti. Pravděpodobnost uvedená u hypero-



Obrázek 3.6: Diagram nalezených hyperonym pro slovo „apple“

nyma je mírou používání zadaného podřazeného slova ve významu nalezeného hyperonyma. Kupříkladu slovo „apple“ má v angličtině minimálně dva možné významy – ovoce a technologická společnost. Předpokládejme, že slovo „apple“ je ve významu ovoce používáno v textech mnohem častěji. Tím pádem bude u hyperonyma „ovoce“ vyšší pravděpodobnost, než u hyperonyma „technologická společnost“. Abychom, v následujícím kroku umožnili jednodušší sloučení stromů, určíme pro všechna nalezená hyperonyma jedno společné nadřazené slovo, kterým je *element*. Výsledný strom pro jedno slovo ze vstupní množiny je zobrazen na obrázku 3.6. Pro přehlednost jsem ponechal pouze tři nalezená hyperonyma místo deseti. Nalezenými hyperonymy pro slovo „apple“ se zadanou váhou $0,1$ jsou slova „fruit“ s pravděpodobností $0,415$, „company“ s pravděpodobností $0,286$ a „food“ s pravděpodobností $0,076$. Váha slova je uvedena přímo u názvu uzlu a pravděpodobnosti jsou uvedeny u hran grafu. Hodnota, která je uvedena u hyperonym je vypočtena dle vzorce, uvedeném v následujících odstavcích.

V druhém kroku vypočteme u každého jednotlivého vytvořeného stromu pravděpodobnost hyperonym na základě váhy podřazených prvků. Tedy pravděpodobnost, že dané hyperonymum popisuje vstupní skupinu. Ta se vypočítá jako pouhý součin váhy vstupního slova a pravděpodobnosti udané v ontologii. Tedy $prob = prob_{MCO} * weight(input)$, kde $prob$ je výsledná pravděpodobnost, $prob_{MCO}$ je pravděpodobnost získaná z ontologie a $weight(input)$ váha slova, zadaná uživatelem. Na obrázku 3.6 je tedy u hyperonyma „fruit“ vypočtena pravděpodobnost jako součin váhy slova „apple“ a pravděpodobnosti uvedené u hrany k danému přechodu: $0,1 * 0,415 = 0,0415$. Pro uměle vložené hyperonymum *element* jsem experimentálně stanovil jako vhodnou pravděpodobnost přechodu $0,01$ tedy 1%. Takováto pravděpodobnost nám zaručí, že se v popisu použije slovo *element* pouze v případech, kdy pro vstupní slova nebylo nalezeno žádné jiné společné pojmenování. Hyperonymum *element* má vypočtenou pravděpodobnost $0,1 * 0,01 = 0,001$, tedy součin váhy slova ze vstupní



Obrázek 3.7: Diagram nalezených hyperonym pro slova „lemon“ a „orange“

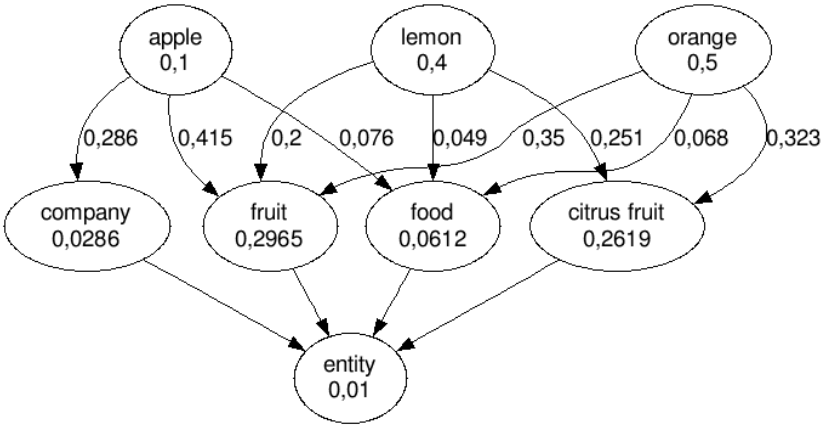
množiny, které popisuje a vlastní pravděpodobnosti.

Ve třetím kroku dojde ke sloučení všech jednotlivých stromů hyperonym do jednoho stromu, u kterého bude kořenem prvek *element*. Sloučení je možné provést pomocí popisované metody $mergeTrees()$, která vypočtené pravděpodobnosti pro sloučená slova sčítá. Po sloučení tedy pravděpodobnost uvedená u hyperonyma odpovídá vzorci $\sum_{k} k_{prob} * weight(k_{child})$, kde k je množina sloučených hyperonym z různých stromů, k_{prob} je pravděpodobnost daného hyperonyma před sloučením (získaná z ontologie) a $weight(k_{child})$ je váha vstupního slova, které je hyponymem pro příslušné slučované hyperonymum k.

Sloučením následujících dvou stromů uvedených na obrázku 3.7 se stromem na obrázku 3.6 vznikne jeden strom, který je uveden na obrázku 3.8. Výpočet pravděpodobnosti pak probíhá dle algoritmu popsaného výše a například u slova „food“ je následující: $0,1 * 0,076 + 0,4 * 0,049 + 0,5 * 0,068 = 0,0076 + 0,0196 + 0,034 = 0,0612$. Pravděpodobnost, že zadaná slova by se dala označit slovem „food“, tedy jídlo, je relativně malá.

Jakmile máme všechna slova ohodnocena, přichází poslední krok algoritmu a sice vybrání výsledků. Každý uzel takto sestaveného stromu je kandidátem na popisné slovo, a proto pro všechny sestavíme strukturu popisu, která je uvedena na začátku této podkapitoly. Ze všech těchto vygenerovaných popisů následně vybereme ty s nejlepšími výsledky, tedy ty, které mají největší podmnožinu popisovaných vstupních slov, a zároveň mají nejvyšší pravděpodobnost, vypočtenou podle metody z předchozího odstavce.

Vynecháme-li popis, který je generovaný samotnými vstupními slovy a obsahuje vždy jen jedno slovo, bude výstup stejný jako v tabulce 3.1. Můžeme zde vidět, že dvě popisná slova na konci tabulky mají vyšší pravděpodobnost, než například „entity“, což je způsobeno tím, že nepokrývají všechna vstupní slova. Předpokladem je, že se snažíme najít takové popisné slovo, které ob-



Obrázek 3.8: Diagram sloučených stromů hyperonym

Tabulka 3.1: Ukázka výstupu popisování z algoritmu Microsoft Concept

Popisné slovo:	Podmnožina vstupních slov	Vypočtená pravděpodobnost	Pokrytí
fruit	apple, lemon, orange	0,2965	100%
food	apple, lemon, orange	0,0612	100%
entity	apple, lemon, orange	0,01	100%
citrus fruit	lemon, orange	0,2619	66,6%
company	apple	0,0286	33,3%

sáhne všechna vstupní slova. Nicméně pracujeme i s tou variantou, že vstupní množina slov obsahuje i jistý druh „šumu“, tedy slova která buď nejsou v externí databázi znalostí k dispozici, nebo jsou zde uvedena kvůli nedokonalosti algoritmu výběru.

Záleží již na uživateli aplikace, zda je pro něj důležitá vypočtená pravděpodobnost, nebo procentuální pokrytí vstupní množiny slov. Pokud bychom řadili výsledky čistě dle pravděpodobnosti, bez ohledu na pokrytí, výsledkem by bylo opět slovo „fruit“ (ovoce), s 29%, následované slovem „citrus fruit“ (citrusové ovoce) s 26%. Rozdíl tří procentních bodů se zdá jako velmi malý, ale je způsoben hlavně zvolenou váhou jednotlivých slov, neboť slova „orange“ a „lemon“, která daný popis obsahuje, mají dohromady 90% ze součtu vah všech slov. Pakliže bychom zvýšili váhu slova „apple“, rozdíl by se razantně zvětšil.

3.6.2 WordNetLabeling

Dalším algoritmem pro popisování vstupní skupiny slov je popisování pomocí externí databáze WordNet, který je založen na hledání společného nadřaze-

ného slova stojícího nad všemi vstupními slovy. K vytvoření takovéto podoby algoritmu jsem dospěl na základě grafického zobrazení stromů ze vstupních testovacích dat a porovnáním, kde se nachází hledané popisné slovo.

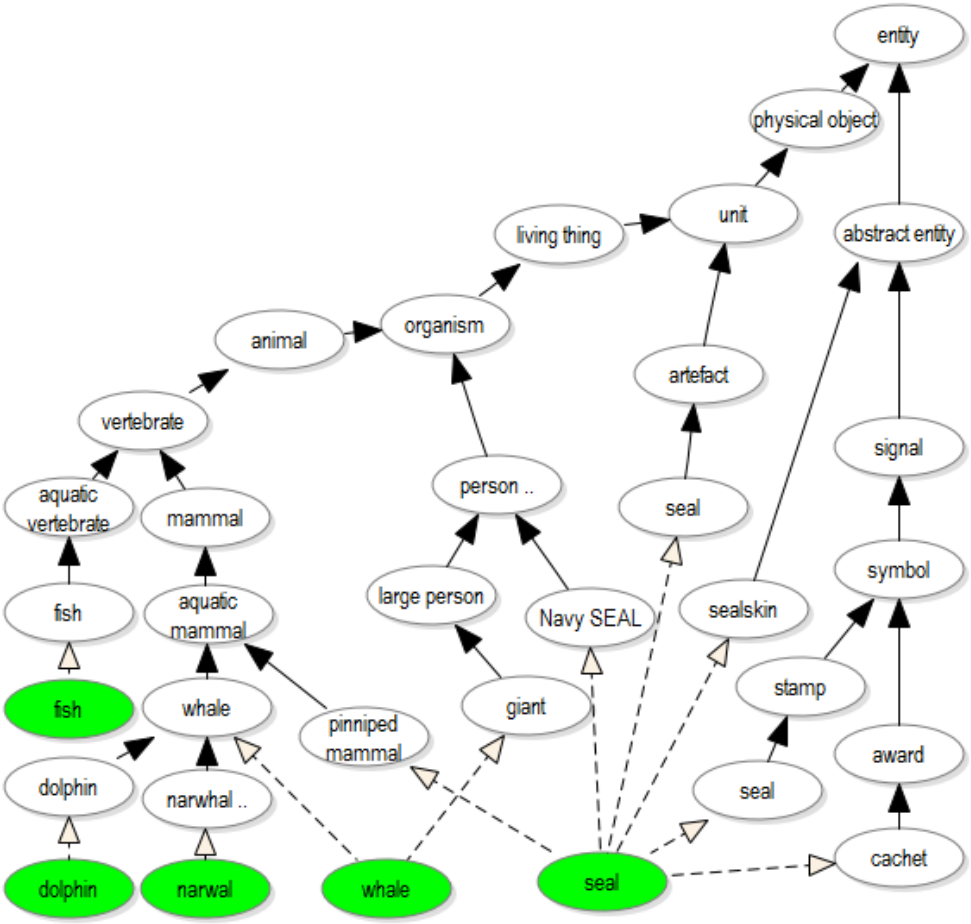
Tyto stromy jsem vytvořil pomocí podobných metod jako u předchozího popisovacího algoritmu. Pro každé vstupní slovo jsem nejprve nalezl všechny možné významy, které současně databáze obsahuje, a následně jsem získal jejich ID synsetu. Pro všechny tyto synsety jsem sestavil seznam hyperonym až ke kořeni a následně je sloučil do jednoho stromu, kde kořenem je slovo „element“.

Například pro vstupní slova „dolphin“, „narwal“, „whale“, „fish“ a „seal“ jsem na základě WordNetu vytvořil strom, který je zobrazen na diagramu 3.9. Původní diagram, který vznikl na základě těchto slov, jsem pro přehlednost upravil tak, že jsem odebral nadbytečné uzly (byť je možné je považovat za popis) a u zbylých uzlů v diagramu jsem ponechal pouze jedno slovo ze synsetu. Zeleně podbarvené uzly představují vstupní slova a nepodbarvené uzly představují jednotlivé synsety z databáze WordNet. Černé šipky pak představují vztah hyponyma k hyperonymu. Světlé přerušované šipky představují různé významy pro jednotlivá zadaná slova.

Podíváme-li se u tohoto stromu na celkovou strukturu, můžeme nalézt (v případě pokrytí všech vstupních slov) sedm možných popisů – „vertebrate“, „animal“, „organism“, „living thing“, „unit“, „physical object“ a „entity“. Každé z těchto slov má určitě vlastnosti, které určují míru toho, jak moc pravděpodobným popisem by mohlo být. Například slova „entity“ a „physical object“ jsou nejvzdálenější od všech vstupních slov, logicky tedy můžou popisovat mnoho slov a popis by byl velmi obecný. Ovšem mohou nastat případy, kdy by tato slova byla natolik nesourodá, že daný popis by byl nejlepším dostupným řešením, a proto by se jednalo o zcela legitimní popis i pro zadaná vstupní slova. Druhým extrémem jsou slova „vertebrate“ a „animal“, která jsou naopak nejblíže vstupním slovům a jsou tak nejpřesnějším dostupným popisem, avšak pouze pro jeden možný význam většiny vstupních slov. Například bychom mohli slovo „seal“ myslet ve významu „Navy SEAL“, tedy jako příslušníka útvaru zvláštních sil námořnictva Spojených států amerických, namísto zvířete, zatímco ostatní slova by byla myšlena ve významech uvedených mořských zvířat. V takovém případě je pojmenování „organism“ zcela jistě správnějším popisem, než „animal“.

Porovnáním stromu můžeme v tomto případě nalézt tři druhy uzlů, které jsou zároveň popisem, a to:

- nejbližší popis pro jeden vybraný význam každého slova, jinými slovy žádné z hyponym nepokrývá všechna vstupní slova – v uvedeném příkladu se jedná o slovo „vertebrate“
- popis, který oproti předchozímu (myšleno nižšímu) přidává další význam minimálně jednoho ze vstupních slov – v uvedeném případě se jedná o slova „organism“, „unit“ a „entity“



Obrázek 3.9: Diagram upraveného stromu WordNet

- popis, který k předchozím dvěma nepřidává další význam žádného ze vstupních slov, ale jeho hyperonymem – v uvedeném případě se jedná o slova „animal“, „living thing“ a „physical object“

Uvedený příklad je velmi podobný grafům, vygenerovaným pro všechny testovací množiny vstupních slov, které jsem ve své práci použil. Principiálně je ve vstupní skupině slov je vždy pouze malé množství slov, které mají více významů, které jsou ovšem navzájem diametrálně odlišné. Na základě těchto skutečností je dle mého názoru možné vytvořit takový popisovací algoritmus, který popisy vyhodnotí a přiřadí jim jistou míru pravděpodobnosti.

Z porovnání takto vytvořených grafů a očekávaných výsledků vyplynulo, že hledaný popis je buď prvním druhem uzlu, popsaným výše, nebo se minimálně nachází v jeho blízkosti (třetí druh uzlu), případně se nachází na místě, které je propojením stávajícího stromu a jiného významu vstupního slova (druhý

druh uzlu). Proto můžeme ohodnocení sestavit tak, že nejpravděpodobnějším popisem je slovo, které je nejblíže jednomu z významů všech vstupních slov. Dalším pravděpodobným popisem jsou slova, která jsou hyperonymem k prvnímu slovu a zároveň připojují jiný význam vstupního slova. V pořadí třetím pravděpodobným popisem jsou slova, která leží mezi prvním a druhým slovem, tedy ta, která jsou hyperonymem prvního slova, hyponymem druhého slova a přitom nepřidávají žádný z významů kteréhokoliv vstupního slova. Druhý a třetí krok se pak opakuje do té doby, než dojdeme ke kořeni stromu.

Tento popis nám na současném příkladu ukazuje, že popisy jsou seřazeny dle pravděpodobnosti, a to od nejvyšší po nejnižší, následovně: „vertebrate“, „organism“, „animal“, „unit“, „living thing“, „entity“ a „physical object“.

Jelikož máme stanoveno, která slova ze stromu se snažíme získat a jaká je jejich pravděpodobnost, je nutné vytvořit způsob, jak celý strom ohodnotíme tak, abychom získali požadované výsledky. K tomuto účelu jsem navrhl následující algoritmus.

Pravděpodobnost každého uzlu se počítá jako k-násobek pravděpodobnosti předka, pakliže nedošlo k odebrání jednoho z možných významů vstupních slov. a to za podmínky, že došlo k odebrání jednoho z možných významů vstupního slova, pravděpodobnost uzlu se počítá jako s-násobek pravděpobodobnosti předka. Koeficienty k a s jsou vstupními parametry předanými uživatelem. Navíc, pokud došlo k „odebrání“ vstupního slova, tedy k tomu, že potomek již nepopisuje ani jeden význam určitého vstupního slova, násobí se vypočtená pravděpodobnost ještě jedním koeficentem n.

Algoritmus začíná v kořeni stromu a prochází jej do šířky s tím, že kořen má určenou pravděpodobnost 1. Výsledkem popisování je pak ohodnocený strom, ze kterého můžeme vybrat jednotlivá popisná slova a určit u nich stejné parametry jako v předchozím případě. Při volbě koeficientů $k = 5$, $s = 0,6$ a $n = 0,1$ získáme výsledky uvedené v tabulce 3.2. Výsledky jsou opět seřazeny nejprve podle pokrytí podmnožiny vstupních slov a následně dle vypočtené pravděpodobnosti, kterou je nutné dále normalizovat tak, aby součet výsledků vždy dosahoval 1 či 100%. Nicméně pro lepší porozumění algoritmu jsem ponechal data nenormalizovaná.

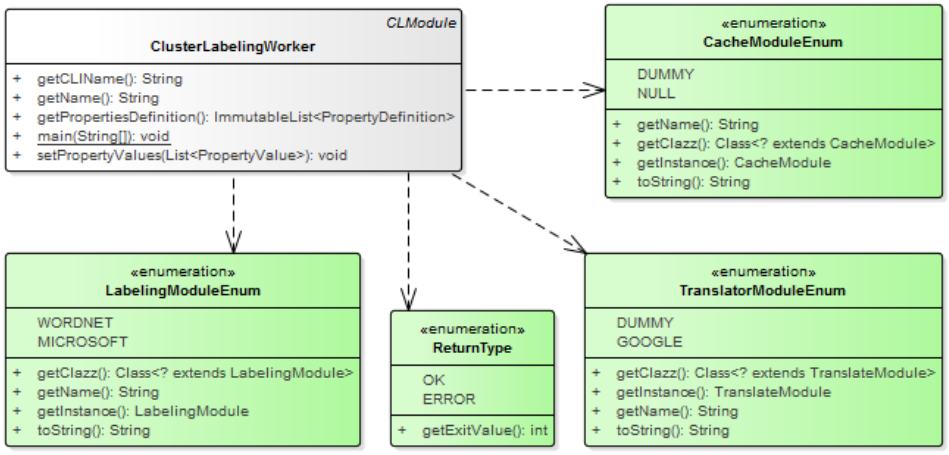
3.7 Společné prvky aplikace

3.7.1 Třída main

Jak již bylo řečeno, pro komunikaci uživatele s aplikací jsem zvolil formu příkazového řádku, neboť grafické rozhraní je v tomto případě nadbytečné. Respektive by neumožnilo zapojení do řetězu zpracovávaných dat, kdy na počátku je například sada dokumentů, jiný nástroj z nich vyextrahuje klíčová slova, tato aplikace je pak pojmenuje a výsledek využije zase úplně jiný nástroj, a to například pro zveřejnění dokumentů pod daným pojmenováním. Pro zjedno-

Tabulka 3.2: Ukázka výstupu popisování z algoritmu WordNet Labeling

Popisné slovo:	Podmnožina vstupních slov	Vypočtená pravděpodobnost	Pokrytí
vertebrate	fish, dolphin, narwal, whale, seal	27	100%
organism	fish, dolphin, narwal, whale, seal	9	100%
animal	fish, dolphin, narwal, whale, seal	5,4	100%
unit	fish, dolphin, narwal, whale, seal	3	100%
living thing	fish, dolphin, narwal, whale, seal	1,8	100%
entity	fish, dolphin, narwal, whale, seal	1	100%
physical object	fish, dolphin, narwal, whale, seal	0,6	100%
aquatic mammal	dolphin, narwal, whale, seal	8,1	80%
mammal	dolphin, narwal, whale, seal	1,62	80%
person	whale, seal	0,54	40%

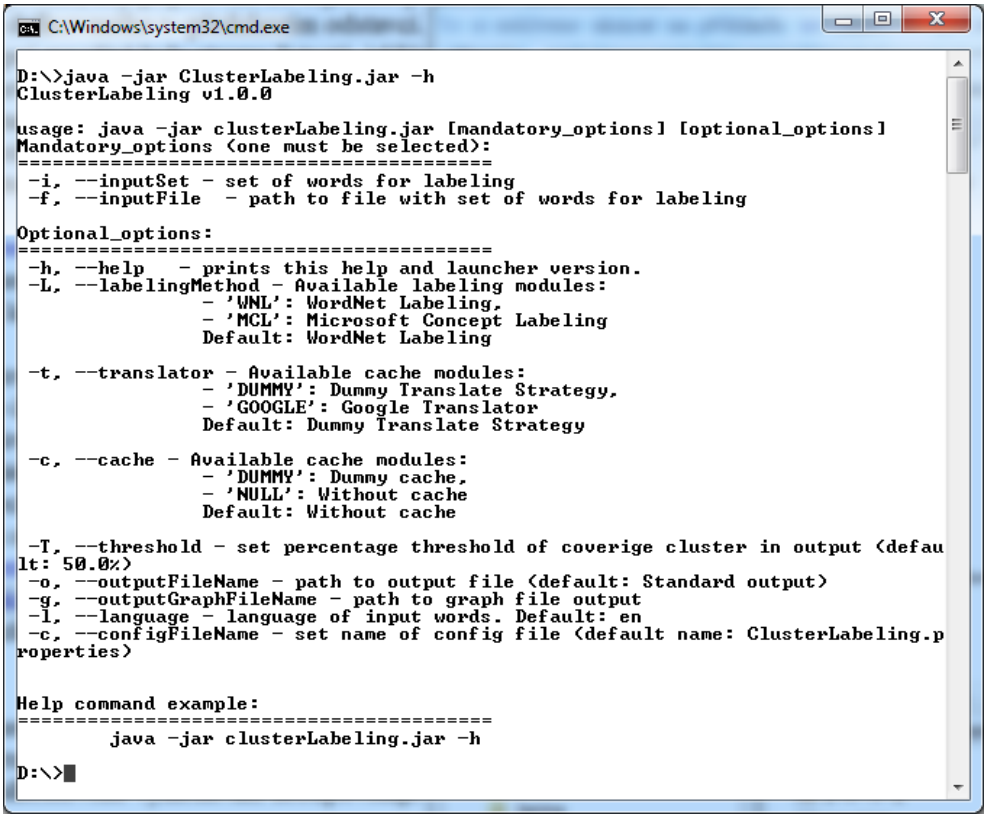


Obrázek 3.10: Diagram třídy *ClusterLabelingWorker*

dušení umožňuje aplikace předání vstupních nebo výstupních hodnot pomocí souborů.

Komunikaci zprostředkovává třída *ClusterLabelingWorker*, která je zároveň jediným vstupním bodem do aplikace, společně s výčtovými typy implementací jednotlivých rozhraní. Kromě třídy *ClusterLabelingWorker* jsou diagramu 3.10 uvedeny i výčtové typy včetně nejdůležitějších metod.

Třída *ClusterLabelingWorker* na základě vstupních parametrů sestavuje objekt třídy *ClusterLabeling*, který se stará o samotný překlad a výsledek vypisuje na zadaný výstup. Při sestavování je nejzajímavějším prvkem výběr jednotlivých implementací abstraktních tříd za pomoci metod poskytovaných výčtovými typy. Všechny výčtové typy obsahují metody *getHelpString*(...),



Obrázek 3.11: Ukázka sestavené nápovědy použití

které sestavují řetězec dostupných implementací a využití najdou při sestavování nápovědy použití. Výstup je vidět na obrázku 3.11. Druhou důležitou metodou ve výčtových typech je $getCacheModuleByName(\ldots)$ (analogicky pro ostatní), která poskytuje konkrétní implementaci pro zadaný název. Například pro vstup $NULL$ navrátí metoda instanci „NullCache“.

3.7.2 Příkazový řádek

Použití v rámci příkazového řádku samozřejmě kromě jiného obsahuje i parametr pro vypsání nápovědy, jejíž výstup je vidět na obrázku 3.11 a jsou na něm jasně patrné všechny vstupní parametry. Celý text nápovědy je rozdělen do tří částí: povinné parametry, nepovinné parametry a nápověda. Poslední z jmenovaných obsahuje typické příklady použití. Z definice nepovinných parametrů je jasné, že všechny mají svou výchozí hodnotu, a tak jediným povinným parametrem je vstupní množina slov v podobě textu nebo cesty k souboru. Formát vstupního souboru obsahujícího množinu slov a formáty dalších souborů jsou uvedeny v následující podkapitole.

3.7.3 Formát vstupních a výstupních souborů

Vstupní soubor, který je možné aplikaci předat při spuštění s přepínačem $-i$, musí mít následující strukturu, kde [*slovo*] je jedno slovo, které chceme pojmenovat a [*vaha*] je váha daného slova v rámci popisovacího algoritmu:

```
[slovo]:[vaha];
[druhe_slovo]:[vaha];
[posledni_slovo]:[vaha]
```

Pro skupinu tří slov, by pak takový konfigurační soubor mohl vypadat následovně:

```
auto:0.9;
letadlo:0.1;
plavidlo:0.005
```

Soubor tedy musí obsahovat maximálně jedno slovo na řádku, vždy následované dvojtečkou a váhou daného slova ve formátu s desetinnou tečkou. Všechny řádky, kromě posledního, jsou pak ukončeny středníkem.

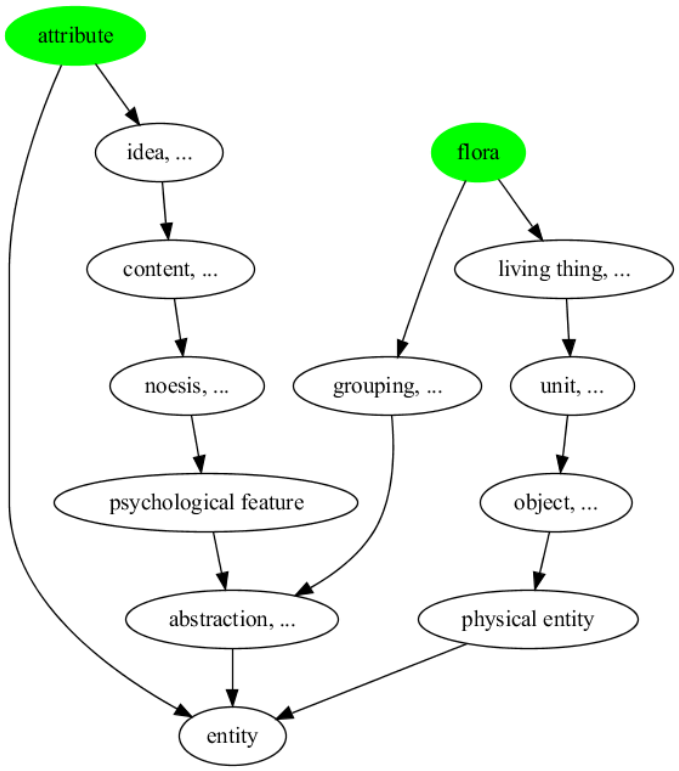
Ať již v případě výstupu do konzole, tak i při výstupu do soboru, je struktura výpisu shodná a logicky jí můžeme členit na tři části:

- zvolená popisná slova,
- popisovaná slova a
- parametry popisu.

První část může obsahovat více slov, pakliže se jedná o synonyma, která jsou v takovém případě uvedena v hranatých závorkách a oddělena čárkou. Druhá část obsahuje výpis popisovaných slov, pro která byla nalezena popisná slova v první části. V parametrech popisu pak můžeme nalézt dva údaje o vlastnostech nalezeného popisu. Prvním z nich je procentuální vyjádření pravděpodobnosti v rámci vypsaného výstupu, že právě uvedené slovo odpovídá požadovanému popisu. Konkrétní výpočet této hodnoty je uveden v popisech algoritmů WordNetLabeling a MicrosoftConceptLabeling. Nicméně dochází zde k normalizaci vypsaných pravděpodobností v případě, že máme jedním z vstupních parametrů omezen výstup na popisy obsahující více než $x\%$, a tak je jejich součet v takovém případě vždy roven 1.

Druhý údaj pak procentuálně vyjadřuje, kolik slov ze vstupní množiny (která jsou uvedena v druhé části výpisu) popisuje zvolené popisné slovo. Ukázkový výpis níže je seřazen nejprve podle počtu slov ze vstupní množiny, které popisuje a dále podle pravděpodobnosti daného popisu.

```
animal: [dog, rat, cat], prob.:37,30%, cover.:100,00%
mammal: [dog, rat, cat], prob.:6,97%, cover.:100,00%
specie: [dog, rat, cat], prob.:6,47%, cover.:100,00%
```



Obrázek 3.12: Ukázka grafu vytvořeného pro pojmenovávání dvou slov „attribute“ a „flora“

```
predator: [dog, rat, cat], prob.:5,77%, cover.:100,00%
element: [dog, rat, cat], prob.:1,04%, cover.:100,00%
pet: [dog, cat], prob.:20,86%, cover.:66,67%
domestic animal: [dog, cat], prob.:8,45%, cover.:66,67%
                              ⋮
```

Tento výpis je možné uložit do souboru pomocí přepínače $-o$, za kterým je uvedena cesta k souboru a pomocí přepínače $-g$, za kterým je opět uvedena cesta, získáme graf reprezentující ontologii, pomocí které je pojmenování nalezeno. Výstupem je graf podobný uvedenému na obrázku 3.12.

3.7.4 Konfigurace

Konfigurace aplikace je možná na základě konfiguračního souboru, v základním nastavení pojmenovaném *ClusterLabeling.properties* a umístěném ve stejném adresáři jako *ClusterLabeling.jar*, tedy spustitelném souboru aplikace. Případně je možné použít libovolný jiný soubor pomocí přepínače $-c$, za kterým následuje cesta k danému souboru. To nám umožní definovat různé

konfigurační soubory v případě použití na různých systémech, či pro různé uživatele.

Tento konfigurační soubor obsahuje vždy dvojce *klic* = *hodnota*, pro které jsem zavedl následující konvenci klíčů. Každý klíč je prefixován názvem příslušného modulu, ke kterému náleží. V následujícím příkladu jsou všechny klíče a jejich hodnoty určeny pro modul *WordNetOntology*, kterému jsem přiřadil prefix *wordnet*.

```
#WordNet
#=======================
#Mandatory:
wordnet.password=
wordnet.dbName=

#Optional (with default values):
#wordnet.host=localhost
#wordnet.port=3306
#wordnet.user=root
                    ⋮
```

3.8 Použití aplikace

3.8.1 Import databáze WordNet

Abychom mohli v aplikaci použít databázi WordNet, je nutné mít k dispozici instanci MySQL, do které je možné naimportovat snímek, dostupný na přiloženém médiu (konkrétní umístění je uvedeno v příloze B). Složka se snímkem databáze obsahuje kromě samotného snímku také dokumentaci, příklady použití a dále skripty pro import snímku do vybrané databáze. Pro systémy Microsoft Windows je určen skript *restore − mysql.bat*. Pro import snímku do lokální MySQL databáze postačuje skript spustit, zadat heslo k uživateli „root“ a název databáze, do které se má snímek obnovit.

3.8.2 Spuštění aplikace

Jelikož se jedná o aplikaci napsanou v jazyku Java, je pro správný běh nutné mít nainstalováno JRE ve verzi 1.8. Po úpravě konfiguračního souboru dle vlastních systémů je možné aplikaci spustit na systémech Microsoft Windows příkazem:

```
java -jar ClusterLabeling.jar -i input.txt -o output.txt
```

Spustitelný soubor *ClusterLabeling.jar* i *ClusterLabeling.properties* jakožto konfigurační soubor, je uložen na přiloženém médiu a konkrétní umístění je uvedeno v příloze B.

3.9 Úpravy aplikace

Aplikaci jsem vyvíjel v prostředí Eclipse, nicméně pro úpravy je možné využít libovolné IDE, které je použitelné pro jazyk Java. Pro sestavení aplikace je pak vhodné ve složce se zdrojovými kódy (kde je uložen soubor *pom.xml*) použít příkaz pro Maven:

```
mvn clean package site
```

Dojde tak ke smazání složky *target*, stažení potřebných knihoven do lokálního Maven repozitáře, sestavení aplikace, spuštění všech jednotkových testů, sestavení spustitelného a konfiguračního souboru a vytvoření dokumentace aplikace.

Kapitola 4

Testování

V této kapitole se zabývám nejen testováním v rámci vývoje, ať už se jedná o statickou analýzu kódu, nebo o jednotkové testy, ale i testováním navržených a realizovaných popisovacích algoritmů. S ohledem na to jsem kapitolu rozdělil na tři základní části, kde popisuji a případně diskutuji jednotlivé fáze testování a jejich výsledky. K prvním dvěma částem velmi přispělo využití nástroje Maven, jelikož sestavení testování mohlo probíhat automaticky v kterékoliv fázi vývoje, včetně návrhu. Z vlastní zkušenosti jsem se nevydal cestou tzv. „vývoje řízeného testy“, kdy jsou nejprve navrhnuty a vytvořeny testy, a teprve následně probíhá vývoj samotného kódu. V této aplikaci totiž není takový druh vývoje možný, neboť ověření správnosti výsledků je v případě funkčního testování velmi obtížné, jak popisuji v podkapitole Funkční testování. Statická analýza kódu navíc má své největší uplatnění právě v průběhu samotného vývoje a nikoliv před jeho začátkem.

Druhým extrémem, oproti „vývoji řízeného testy“, je klasický model vývoje s názvem „vodopád“, kdy teprve po skončení vývoje je na řadě vytvoření testů a na jejich základě případné opakování vývoje. Nevýhody takového přístupu jsou očividné a všeobecně známé.

Proto jsem se rozhodl pro vlastní styl vývoje, kdy vytvoření testů proběhlo až po návrhu i vytvoření menší části kódu, u kterého jsem si ručně ověřil základní funkčnost a integritu této části kódu za pomoci již vytvořených testů. Takový vývoj by šel stěží podřadit pod konkrétní metodiku, neboť z praxe mám ověřeno, že stejný styl vývoje probíhal jako podoba agilních metodik nebo naopak striktně procesních a vodopádovitých metodik.

4.1 Statická analýza kódu

Jak již bylo řečeno, statická analýza kódu byla ve vývojové fázi velkým pomocníkem. Statická analýza je v mnoha případech podceňována a označována za zbytečnou ztrátu času, v praxi se však vždy ukáže, že minimálně může zrychlit vykonávání, nebo dokonce předejít chybám. Navíc pokud vykonání

této analýzy vložíme do automatického sestavení, které vytváří Maven, jediný čas, který tím ztratíme, je ten strojový. Člověk pak už jen zapracuje nalezené chyby, případně označí nalezené chyby jako správné řešení, pokud chybou ve skutečností nejsou.

4.1.1 FindBugs

Nástroj FindBugs[6] pracuje na úrovni kompilovaného bytekódu a pomáhá odhalovat programátorské „chyby“, a to na základě více než 400 pravidel, rozdělených do 9 kategorií. Jedná se například o bezpečnost, výkon, „špatnou praxi“, zranitelnost apod. Kód, který neobsahuje příkazy, které by odhalilo jedno z uvedených pravidel, není možné považovat za bezchybný, ale je možné říct, že neobsahuje některá „přehlédnutí“, která je možné automatizovaně odhalit. Nicméně v určitých případech se může stát, že FindBugs označí za „chybu“ i kód, který je takto napsán záměrně.

V mé aplikaci nástroj nalezl chybu „Neefektivní použití iterátoru klíčů místo iterátoru hodnot“ při procházení kolekce v metodě $getClusters(\ldots)$ ve třídě $LabelingModule$. Aplikace skutečně prochází kolekce pomocí klíčů, nicméně důvodem je, že klíče jsou následně použity jako hodnoty pro popisování a efektivnější cesta, jak získat zároveň klíč, i hodnotu z objektu $HashMap$, neexistuje. Příslušné pravidlo $WMI\_WRONG\_MAP\_ITERATOR$ je nutné z tohoto důvodu deaktivovat, a to minimálně na úrovni dané metody za pomoci přidání následující anotace k metodě.

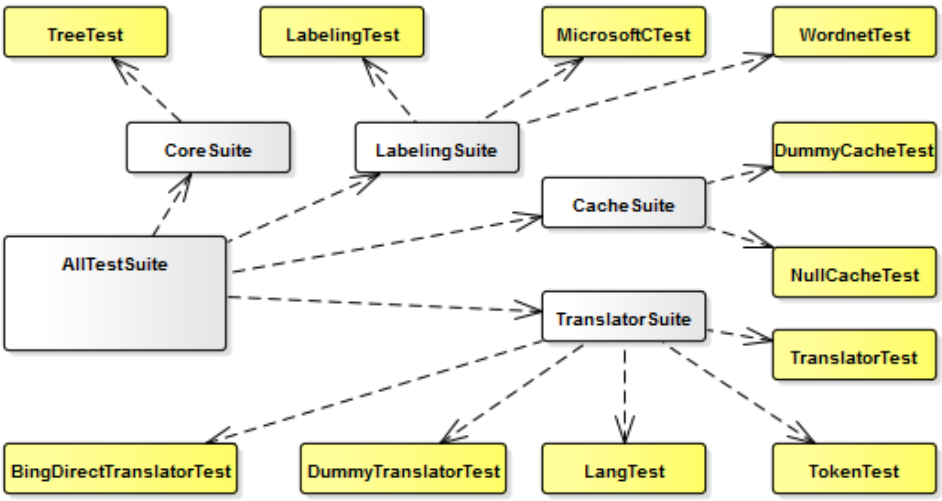
```
@SuppressFBWarnings("WMI_WRONG_MAP_ITERATOR")
```

Ostatní nešvary, které se v kódu vyskytly, jsem za pomocí FindBugs odstranil. Což dokládá i vygenerovaný report v dokumentaci na přiloženém médiu.

4.1.2 JaCoCo

Nástroj JaCoCo[12] je tzv. „Java Code Coverage Library“, neboli nástroj pro kontrolu kódu pokrytého testy. Společně s testy z jednotkového testování v následující podkapitole vytváří přehled o tom, které příkazy jsou v rámci těchto testů procházeny a jsou tím pádem otestovány. Z toho plyne, že i když jsou jednotlivé příkazy vykonány, není zaručeno, že testy postihnou všechny možnosti, nicméně jsou velmi dobrým pomocníkem v případě nacházení neotestovaných metod, či větví v kódu. Stejně jako v předchozím případě vytvoří nástroj dokumentaci v podobě HTML stránek, které jsou dostupné na přiloženém médiu a mají podobu procentuálního vyjádření pokrytí kódu testy pro jednotlivé metody v třídách. Tyto testy a jejich vyhodnocení se provádí při každém sestavení programu a samotné sestavování nástrojem Maven prodlužují, ale opět ubírají pouze strojový čas a nikoliv čas programátora.

Zavedená praxe říká, že není možné pokrýt 100 % kódu testy. Z logického hlediska to ani není efektivní, neboť velmi často zůstávají neotestovány na-



Obrázek 4.1: Diagram jednotkových testů

příklad více parametrové konstruktory, které kromě vytvoření instance ještě nastaví hodnoty. Taková akce může vyvolat výjimku v mizivém počtu případů, navíc náklady na vytvoření takového testu jsou přitom velmi vysoké v porovnání s efektivitou takového testu. Všeobecně se udává, že efektivní poměr pokrytí kódu testy se pohybuje kolem 80 %, což se mi v této práci s výsledkem 83 % podařilo dodržet.

4.2 Jednotkové testování

Nedílnou součástí každé aplikace by měla být sada jednotkových testů k ověření správnosti implementace. Rozdělíme-li aplikaci na menší jednotky, můžeme tyto jednotky samostatně testovat tak, abychom měli jistotu, že při společné práci nedojde k chybě. Tyto jednotky můžeme dále dělit na stále menší, až je jednotkou samotná metoda v rámci určité třídy. V rámci této aplikace můžeme považovat za testovanou jednotku jednotlivé implementace abstraktních tříd, uvedených v kapitole Analýza a návrh na diagramu 2.2 a dále samotné implementované metody.

Pro jednotkové testování se v Javě využívají převážně dva testovací frameworky, kde jedním z nich je JUnit. Tento nástroj poskytuje různé možnosti jak kód testovat a rozdělovat do menších částí, a zároveň je možné ho využít i v rámci nástroje Maven při sestavování aplikace pro ověření funkčnosti, které předchází samotnému sestavení aplikace. V poslední řadě je možné ho využít pro ověření pokrytí kódu testy, a tak jsou zároveň výsledky pokrytí i testů uvedeny ve vygenerované dokumentaci na přiloženém médiu. Rozdělení na menší části, tzv. soubory testů, mi umožnilo vytvořit strukturu testů

podobnou struktuře implementace, což vede k jednoduššímu ověřování funkčnosti a vytváření dalších testů. Struktura testů je vyobrazena na diagramu 4.1. Na tomto diagramu je hlavním souborem testů třída *AllTestSuite*, která při spuštění dále provolává soubory testů *CoreSuite*, *LabelingSuite*, *CacheSuite* a *TranslatorSuite*. Tyto menší soubory testů pak již obsahují jednotlivé testy funkčnosti.

4.3 Funkční testování

Funkční testování je zcela jistě nejdůležitější částí celého testovacího procesu, kdy si můžeme ověřit, že naprogramované algoritmy poskytují požadované výsledky na zadané vstupy. V případě automatického pojmenovávání skupin slov ovšem narážíme na problém, jak ověřit, že výstupy z algoritmů jsou správné, případně, jak moc jsou správné, neboť výsledků může být mnoho a je prakticky nemožné vyloučit kterýkoliv z nich. Ve většině případů totiž jako pojmenování můžeme považovat téměř každé slovo, které má na vstupní slova určitou vazbu.

Druhým problémem je pak výběr vhodných testovacích dat, na kterých můžeme správnost algoritmu ověřit, neboť ty jsou velmi často výsledkem mnohem delšího procesu zpracovávání, než je jejich samotné popisování. Proto jsem využil primárně dva různé zdroje vstupních dat, které se navzájem liší způsobem výběru.

4.3.1 Způsob výběru testovacích dat

Pro vývoj a otestování funkčnosti základních algoritmů včetně překladů, jsem jako jeden zdroj využil vlastní a menší soubor skupin slov, u kterých jsem zároveň určil očekávaný popis. Pro zhodnocení úspěšnosti popisovacích algoritmů jsem naopak využil větší soubor skupin slov, který vznikl jako součást bakalářské práce [29]. Tento větší soubor skupin slov má také určen očekávaný popis, nicméně v rámci testování se ukázalo, že určení hledaného názvu je nedostatečné a „šum“ v souborech je příliš velký. Soubor vznikl na základě článků ve Wikipedii, kde z každého článku bylo vybráno 20 významných slov a jako pojmenování byl použit název článku.

Kvůli těmto nedostatkům jsem vytvořil třetí soubor skupin slov, který vznikl náhodným výběrem 100 skupin slov z předchozího souboru a následně jsem každou skupinu ručně upravoval tak, aby neobsahovala zbytečná či neurčitá slova a více odpovídala samotnému článku, ze kterého skupina slov vznikla. V druhém kroku jsem každé této skupině určil více možných hledaných výsledků tak, aby odpovídal zadané skupině.

Tabulka 4.1: Výsledky popisování 14510 skupin slov dle pokrytí vstupní skupiny slov pro oba popisovací algoritmy

Pokrytí [%]	Microsoft Concept	WordNet
<10	13994	13002
11-20	336	890
21-30	101	331
31-40	37	157
41-50	25	77
51-60	9	33
61-70	2	13
71-80	0	6
81-90	0	0
>90	6	1
Celkem	14510	14510

4.3.2 Způsob zhodnocení výsledků popisování

Jelikož výstupem z aplikace je vždy seznam možných popisů, u kterých je určena pravděpodobnost a pokrytí vstupní skupiny, můžeme na vyhodnocování výsledků nahlížet ze hlediska dvou různých kritérií, kde prvním z nich je pokrytí co největšího počtu vstupních slov a druhým je získání výsledků s nejvyšší možnou pravděpodobností.

Pro ověření výsledků práce popisovacího algoritmu *WordNetLabeling* dle pravděpodobnosti je nutné nejprve získat vhodné hodnoty koeficientů, pomocí kterých algoritmus pravděpodobnost zvoleného popisu vypočítává. Nicméně pro základní prověření kvality vstupních a výstupních dat je možné použít druhé kritérium. Výsledky zpracování souboru o více než 14 tisících množin slov, každá o 20 slovech, jsou uvedeny v tabulce 4.1.

Pro každý popisovací algoritmus tabulka ukazuje, pro kolik vstupních množin slov se podařilo najít definovaný výstup s procentuálním pokrytím vstupní množiny. Je vidět, že úspěšnost je velmi malá, neboť pouze pro 6 vstupních množin slov (v případě popisování pomocí algoritmu *MicrosoftConceptLabeling*), respektive pro 1 vstupní množinu (v případě popisování pomocí algoritmu *WordNetLabeling*), popisoval definovaný výstup více než 90 % vstupních slov. Nízká úspěšnost je způsobena hlavně „šumem“ ve vstupních datech a až přílišnou konkrétností požadovaného výstupu. Například u hudebních skupin je požadován jako výsledek popisu jejich konkrétní název, přičemž vstupní skupina slov obsahuje slova, které označují hudební skupinu obecně.

Z tohoto důvodu jsem náhodně vybral 100 různých množin vstupních slov a ty následně editoval tak, aby neobsahovaly zmíněný šum. V dalším kroku jsem pro tyto skupiny slov vybral vhodnější hledaný popis jako požadovaný

Tabulka 4.2: Výsledky popisování vybraných 100 vstupních skupin pomocí obou popisovacích algoritmů

	Neupravená data		Upravený vstup		Upravený vstup i výstup	
Pokrytí [%]	Microsoft Concept	WordNet	Microsoft Concept	WordNet	Microsoft Concept	WordNet
<10	90	80	63	70	32	21
11-20	8	15	32	21	24	17
21-30	1	2	3	5	11	17
31-40	1	2	2	2	17	22
41-50	0	0	0	0	9	13
51-60	0	0	0	1	4	3
61-70	0	0	0	0	1	4
71-80	0	0	0	0	1	2
81-90	0	0	0	0	1	0
>90	0	1	0	1	0	1
Celkem	100	100	100	100	100	100

výsledek. Výsledek popisování je uveden v tabulce 4.2, kde můžeme vidět porovnání obou popisovacích algoritmů z hlediska pokrytí vstupních slov ve třech případech, a to z:

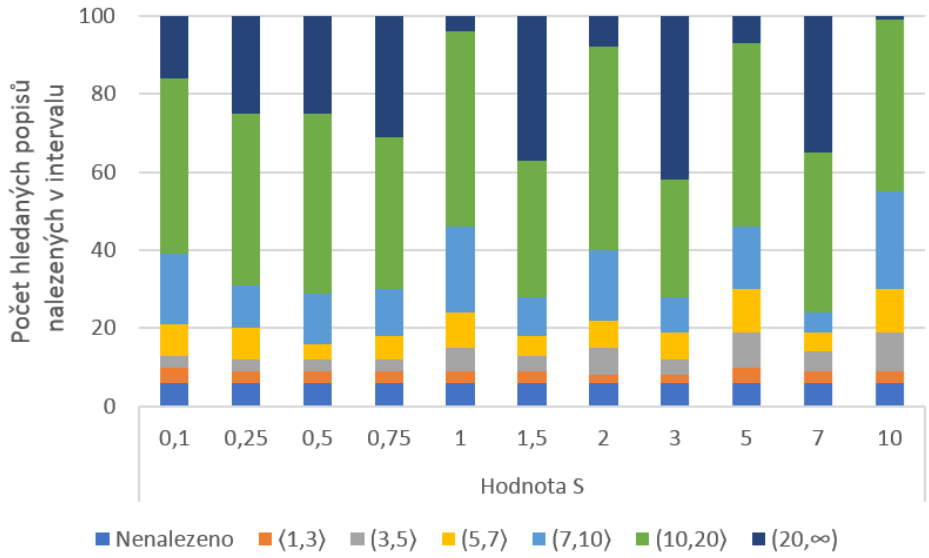
- neupravených vstupních dat,
- upravených vstupních dat a
- upravených vstupních i výstupních dat.

Je očividné, že nejlepších výsledků dosahují algoritmy v případě upravených vstupních i výstupních množin slov, díky čemuž jsem tento upravený soubor použil při hledání vhodných koeficientů pro algoritmus využívající WordNet.

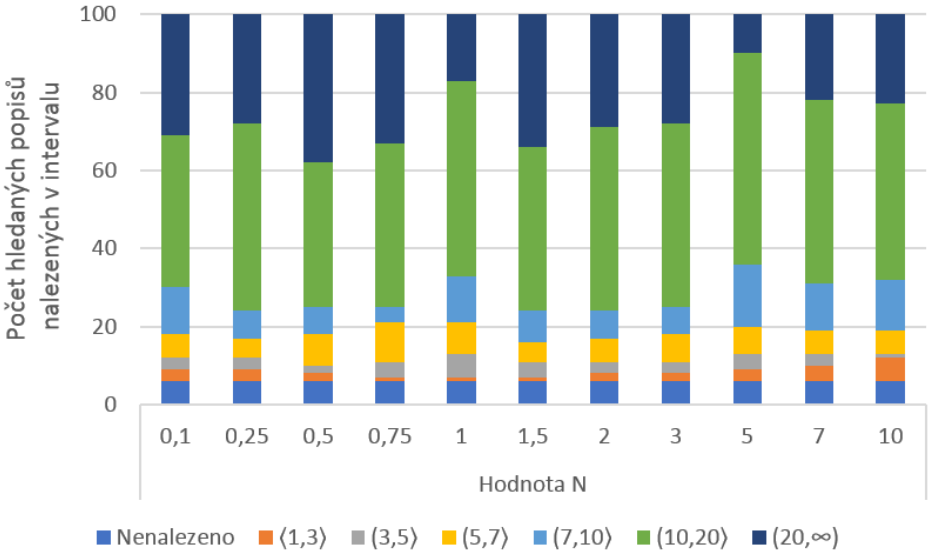
Toto hledání jsem začal zvolením počátečních hodnot ($s = 0,75$, $k = 5$, $n = 0,1$), které jsem získal z příkladu uvedeného v popisu algoritmu $WordNetLabeling$. Následně jsem pro všechny koeficienty zvolil možné hodnoty v rozmezí $0,1 - 10$ na základě pozorování na menším souboru testovacích dat a následném určení, kde již dochází k ustálení výsledků a změna koeficientů nemá vliv na pravděpodobnost popisu.

Při hledání jsem vždy zafixoval dva ze tří koeficientů a vyhodnocoval výsledky s různými hodnotami třetího koeficientu. Výsledky jsou znázorněny na grafech 4.2, 4.3 a 4.4.

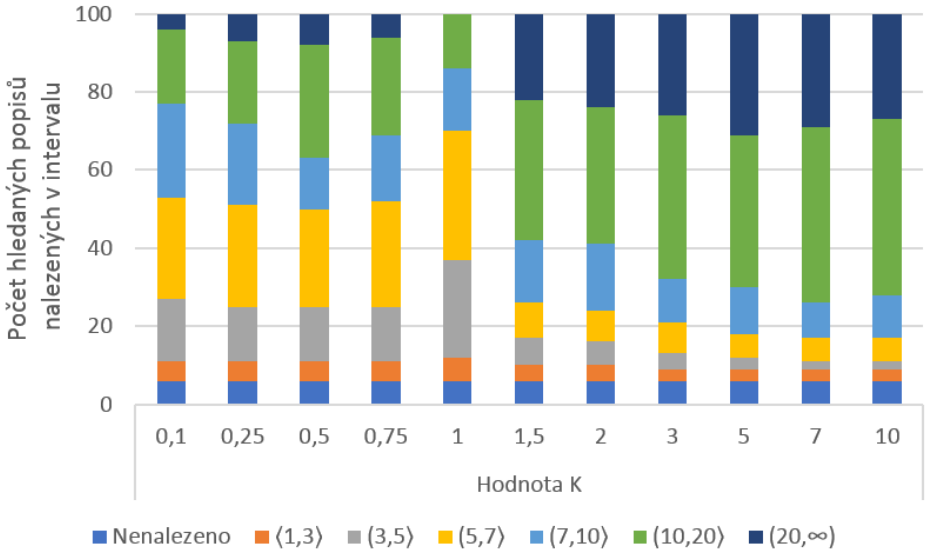
Všechny uvedené grafy jsou histogramem, kde je znázorněno, kolik výsledků bylo nalezeno v možných popisech na pozici v intervalu $\langle i, j \rangle$, seřadíme-li výstup dle vypočtené pravděpodobnosti. Například v grafu 4.2 je nejlepším



Obrázek 4.2: Závislost výsledků popisování dle vypočtené pravděpodobnosti algoritmem *WordNetLabeling* na koeficientu s



Obrázek 4.3: Závislost výsledků popisování dle vypočtené pravděpodobnosti algoritmem *WordNetLabeling* na koeficientu n

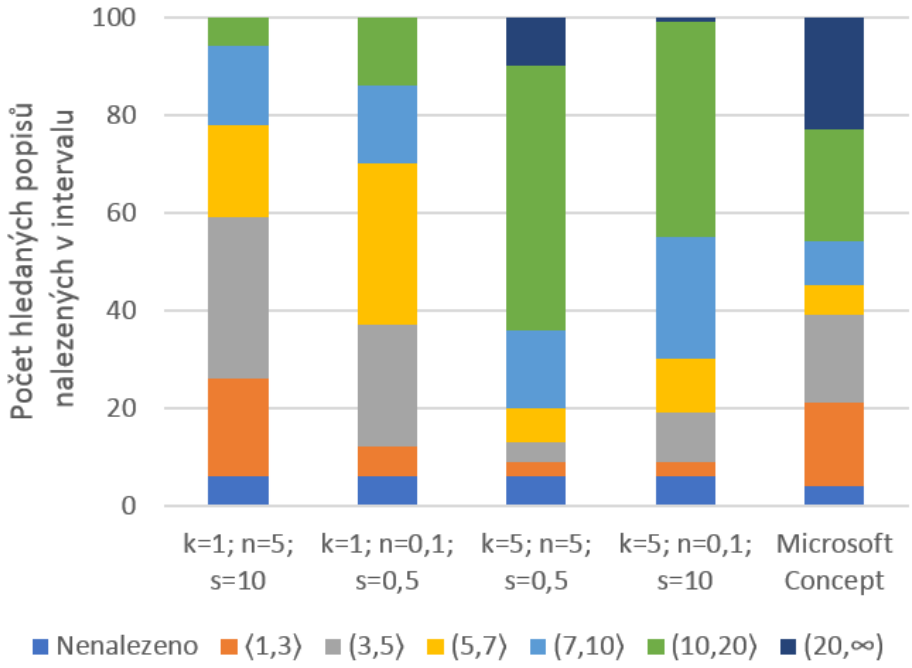


Obrázek 4.4: Závislost výsledků popisování dle vypočtené pravděpodobnosti algoritmem *WordNetLabeling* na koeficientu k

výsledkem hodnota $s = 10$. Jelikož nejvíce výsledků se nachází na nejvyšších příčkách – zhruba 50% vstupních skupin slov bylo popsáno definovaným výstupem, který se nacházel v prvních 10 výsledcích seřazených dle pravděpodobnosti.

Z uvedených grafů je jasně patrné, že nejlepších výsledků z hlediska vypočtené pravděpodobnosti, dosahujeme při použití koeficientů ($s = 10$, $k = 1$, $n = 5$), neboť pro tyto koeficienty nacházíme požadované výsledky velice blízko první pozici.

Následující graf 4.5 je pak porovnáním nejlepších výsledků dle pravděpodobnosti při změně pouze jednoho z koeficientů a výsledků získaných kombinací nejlepších koeficientů. Zároveň ukazuje nejlepší výsledky z popisování pomocí algoritmu *MicrosoftConceptLabeling* na stejných (upravených) vstupních i výstupních datech. Můžeme tak vidět, že při použití koeficientů ($s = 10$, $k = 1$, $n = 5$) pro algoritmus WordNetLabeling nacházíme více než 50 % výsledků na jedné z prvních pěti pozic. Naproti tomu algoritmus *MicrosoftConceptLabeling* dosahuje stejných výsledků pouze v 35 %. I přes tento fakt může být algoritmus *MicrosoftConceptLabeling* dle mého názoru využit pro popisování skupin slov.



Obrázek 4.5: Porovnání popisování algoritmem *WordNetLabeling* s různými koeficienty a popisování algoritmem *MicrosoftConceptLabeling*

Závěr

Cílem mé práce bylo vytvoření aplikace pro automatické pojmenovávání skupin slov, což se mi podle mého názoru podařilo. Aplikace tak nyní může být použita v rámci většího celku pro zpracovávání dokumentů. V první kapitole své práce jsem analyzoval způsoby, jakými je možné skupinu slov popsat, dále jaké algoritmy pro popisování již existují a jaké mají vlastnosti. Ze všech uvedených jsem se rozhodl pro popisování na základě externích databází znalostí, neboť dle mého názoru nabízejí největší možnosti pro popisování. Jako první z externích databází znalostí jsem si zvolil WordNet, který je velmi rozsáhlý a nabízí možnosti využití i na jiných místech, než jen popisování. Jako druhou externí databázi znalostí jsem se rozhodl využít Microsoft Concept Graph, neboť se neustále rozšiřuje a v podobné aplikaci ještě nebyl použit.

V druhé kapitole se pak zabývám návrhem samotné aplikace a snažím se klást důraz hlavně na rozšiřitelnost a udržovatelnost, což vychází jednak z mé dosavadní praxe vývojáře a jednak ze zaměření aplikace. Prvním bodem návrhu byla specifikace aplikace, jakožto nejdůležitější krok v rámci návrhu, ve které jsem si stanovil cíle, jichž by mělo být dosaženo. Praxe vývojáře mi umožnila relativně snadno identifikovat místa, která jsou kritická pro další možné rozšíření. Ve všech těchto místech jsem se snažil použít takové návrhové vzory, které rozšíření umožňují.

Třetí kapitola popisuje proces realizace navržených rozhraní z předchozí kapitoly. Dále bylo nutné řešit napojení na externí databáze znalostí, u kterých jsem zvolil odlišný přístup. WordNet jsem ponechal jako lokální MySQL databázi, ke které poskytoval přístup aplikaci standardní MySQL ovladač. Pro získávání informací z Microsoft Concept Graph jsem zvolil online metodu, hlavně kvůli neustálému rozšiřování. Kromě napojení na externí databáze znalostí, implementace jádra aplikace a implementací rozhraní, jsem nad rámec požadavků implementoval i nástroj pro grafické zobrazení grafů ontologie, což se ukázalo jako velmi výhodné zvláště při vývoji a testování popisovacích algoritmů. Velmi významnou částí třetí kapitoly jsou pak samotné popisovací algoritmy a možnost jejich konfigurace. Popisování pomocí Microsoft Concept

Graph je založeno na vztazích mezi zadanými slovy a jejich pravděpodobností užití v daném kontextu. Oproti tomu je popisovací algoritmus s využitím databáze WordNet založen na hledání hyperonyma pro podmnožinu zadaných slov a jejich pravděpodobnostním ohodnocení za pomocí tří různých koeficientů. Zároveň jsem v této kapitole uvedl, jak je možné aplikaci použít a jak ji případně rozšiřovat, či sestavit z dostupných zdrojových kódů za pomocí nástroje Maven.

Poslední kapitola se zabývá testováním výsledné aplikace, ať již z hlediska statické analýzy kódu, tak z hlediska jednotkových testů. Zásadním prvkem testování je však důkladné funkční testování, které jsem provedl na datech získaných z jiné práce a dále upravených tak, aby lépe vyjadřovala realitu. Domnívám se, že výsledné popisovací algoritmy pracují správně a s relativně vysokou přesností, což považuji za úspěšné splnění hlavního cíle této práce.

Problematika popisování skupin slov je velice rozsáhlá a domnívám se, že jsem vytvořil dobrý základ pro další, kteří by chtěli pokračovat a popisovací algoritmy buď vylepšovat, nebo vytvářet zcela nové. Zároveň věřím, že výsledná práce najde využití v rámci automatického popisování dokumentů a bude dále rozvíjena, ať již v akademické, nebo i v komerční sféře.

Literatura

[1] Carmel, D.; Roitman, H.; Zwerdling, N.: Enhancing cluster labeling using wikipedia. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2009, pp. 139–146.

[2] *DBPedia* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `http://wiki.dbpedia.org`.

[3] Dostál, M.; Nykl, M.; Ježek, K.: Cluster labeling with Linked Data. *Journal of Theoretical & Applied Information Technology*, volume 53, no. 3, 2013.

[4] Eikvil, L.; Jenssen, T.-K.; Holden, M.: Multi-focus cluster labeling. *Journal of biomedical informatics*, , no. 55, 2015: pp. 116–123.

[5] Fellbaum, C.: *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[6] FindBugs [online]. [Cit. 7.5.2017]. Dostupné z WWW: `http://findbugs.sourceforge.net/`.

[7] Fowler, M.: *Destilované UML*. Knihovna programátora, Grada Publishing a.s., 2009, ISBN 978-80-247-2062-3.

[8] *Google Knowledge Graph* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://www.google.com/intl/bn/insidesearch/features/search/knowledge.html`.

[9] graphviz-java [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://github.com/nidi3/graphviz-java`.

[10] *google-cloud-java* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://github.com/GoogleCloudPlatform/google-cloud-java/tree/master/google-cloud-translate`.

[11] Hua, W.; Wang, Z.; Wang, H.; etc.: Short text understanding through lexical-semantic analysis. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, IEEE, 2015, pp. 495–506.

[12] JaCoCo Java Code Coverage Library [online]. [Cit. 7.5.2017]. Dostupné z WWW: `http://www.eclemma.org/jacoco/`.

[13] *JDBC Driver for MySQL* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://www.mysql.com/products/connector/`.

[14] *JSON in Java* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://github.com/stleary/JSON-java`.

[15] Keogh, J.: *Java*. Bez předchozích znalostí, Computer Press, a.s., 2005, ISBN 978-80-251-0839-0.

[16] Český národní korpus: *Abecední a retrográdní slovníky*. Ústav Českého národního korpusu FF UK: Praha, 2016, dostupné z WWW: `http://www.korpus.cz`.

[17] *microsoft-translator-java-api* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://github.com/boatmeme/microsoft-translator-java-api`.

[18] Getting Started with Translator: Subscribe and get Credentials [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://blogs.msdn.microsoft.com/translation/gettingstarted1/`.

[19] Miller, G. A.: WordNet: a lexical database for English. *Communications of the ACM*, volume 38, no. 11, 1995: pp. 39–41.

[20] *Maven* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `https://maven.apache.org`.

[21] *Open Directory Project* [online]. [cit. 7.5.2017]. Dostupné z WWW: `http://dmoztools.net`.

[22] Oppel, A.: *SQL*. Bez předchozích znalostí, Computer Press, a.s., 2008, ISBN 978-80-251-1707-1.

[23] Pala, K.; Ševeček, P.: *The Czech WordNet, final report*. Brno: Masarykova univerzita, 1999, 21 s., technická zpráva.

[24] Pecinovský, R.: *Návrhové vzory*. Computer Press, a.s., 2007, ISBN 978-80-251-1582-4.

[25] Pecinovský, R.: *Java 7*. Knihovna programátora, Grada Publishing a.s., 2012, ISBN 978-80-247-3665-5.

[26] Pecinovský, R.: *Java 8*. Knihovna programátora, Grada Publishing a.s., 2014, ISBN 978-80-247-4638-8.

[27] Popescul, A.; Ungar, L. H.: Automatic labeling of document clusters. 2000, dostupné z WWW: `http://www.cis.upenn.edu/~ungar/Datamining/Publications/labels.pdf`.

[28] Song, Y.; Wang, H.; Wang, Z.; etc.: Short text conceptualization using a probabilistic knowledgebase. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, AAAI Press, 2011, pp. 2330–2336.

[29] Štefančík, M.: *Automatické pojmenovávání skupin slov*. Bakalárská praca, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2017.

[30] Wang, Z.; Wang, H.: Understanding Short Texts. 2016.

[31] Wang, Z.; Wang, H.; Hu, Z.: Head, modifier, and constraint detection in short texts. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, IEEE, 2014, pp. 280–291.

[32] Wang, Z.; Wang, H.; Wen, J.-R.; etc.: An Inference Approach to Basic Level of Categorization. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ACM, 2015, pp. 653–662.

[33] Wang, Z.; Zhao, K.; Wang, H.; etc.: Query understanding through knowledge-based conceptualization. 2015.

[34] Wei, T.; Lu, Y.; Chang, H.; etc.: A semantic approach for text clustering using WordNet and lexial chains. *Expert Systems with Applications*, volume 42, no. 4, 2015: pp. 2264–2275.

[35] Wiegers, K. E.: *Požadavky na software*. Computer Press, a.s., 2008, ISBN 978-80-251-1877-1.

[36] *Wikipedia* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `http://en.wikipedia.org/`.

[37] *WordNet SQL* [online]. [Cit. 7.5.2017]. Dostupné z WWW: `http://wnsql.sourceforge.net`.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

CZ Český jazyk

DTO Data Transfer Object

EN Anglický jazyk

GER Německý jazyk

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol)

IDE ntegrated Development Environment

ILI Inter–Lingual Index

JAR Java Archive

JRE Java Runtime Environment

JSON JavaScript Object Notation

ODP Open Directory Project

OOP Objektově orientované programování

SQL Structured Query Language

URL Uniform Resource Locator

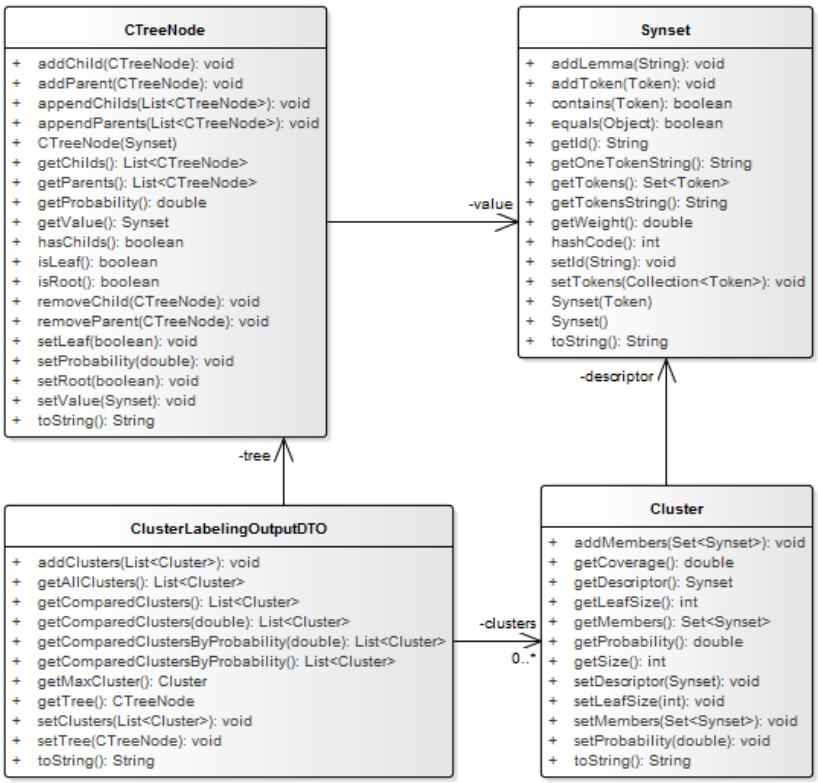
Příloha **B**

Obsah přiloženého CD

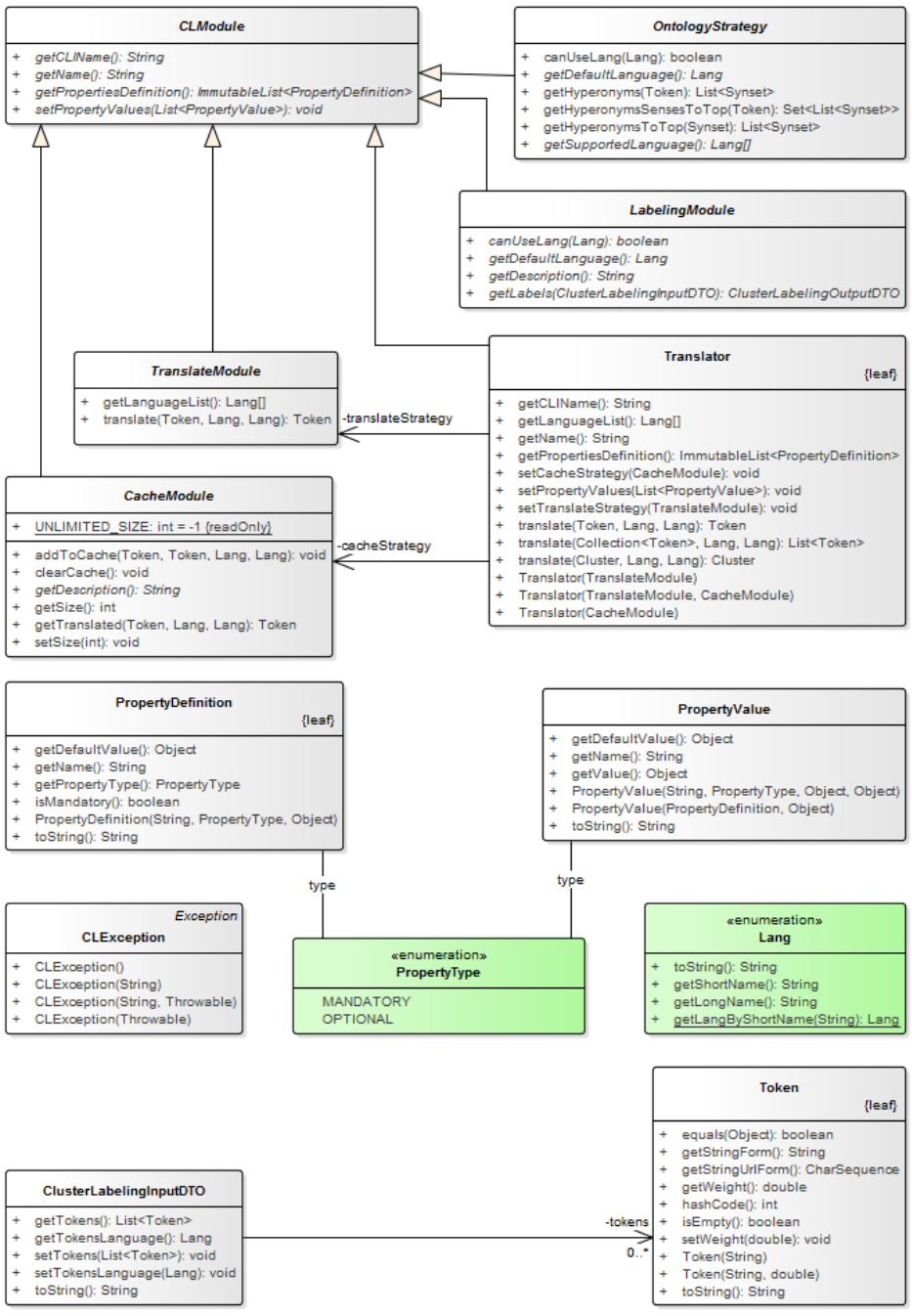
- `readme.txt` ..... stručný popis obsahu CD
- `jar` ..... spustitelná forma implementace celého nástroje
  - `ClusterLabeling.jar` ..... spustitelný JAR soubor aplikace
  - `ClusterLabeling.properties` ..... konfigurační soubor aplikace
- `license` ..... licence použitých knihoven a nástrojů
- `src` ..... zdrojové kódy
  - `application` ..... zdrojové kódy implementace celého nástroje
    - `src` .....
      - `main` ..... zdrojové kódy aplikace
      - `test` ..... zdrojové kódy testů
    - `CTIME.txt` ..... Zjednodušený postup pro vývoj nástroje
    - `pom.xml` ..... konfigurační soubor pro nástroj Maven
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `tests` ..... testovací soubory
  - `large` ..... soubor o 14510 vstupních množinách
  - `reduced` ..... redukovaný soubor o 100 vstupních množinách
    - `100` ..... neupravený testovací soubor
    - `100-upraveny_vstup` ..... upravený testovací soubor
    - `hledane_vysledky.txt` ... hledané výsledky v upraveném souboru
- `text` ..... text práce
  - `generated_documentation` ..... generovaná dokumentace aplikace
  - `Effenberger_DP.pdf` ..... text práce ve formátu PDF
- `WordNetSQL` ..... snímek MySQL databáze WordNet

Příloha C

Diagram návrhu aplikace



Obrázek C.1: Diagram tříd návrhu aplikace



Obrázek C.2: Pokračování diagramu tříd navrhu aplikace