VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GENEROVÁNÍ PROGRAMOVÉHO KÓDU Z DEFINICE ONTOLOGIE

GENERATING PROGRAM CODE FROM ONTOLOGY DEFINITION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. TOMÁŠ SVETLÍK**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání diplomové práce

Student:	**Svetlík Tomáš, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Softwarové inženýrství
Název:	**Generování programového kódu z definice ontologie** **Generating Program Code from Ontology Definition**
Kategorie:	Softwarové inženýrství

Zadání:

1. Seznamte se s technologiemi sémantického webu se zaměřením na jazyky pro definici ontologií, zejména jazykem OWL.
2. Prostudujte softwarové existující nástroje a knihovny pro zpracování RDF a OWL definic a pro generování programového kódu pomocí šablon.
3. Po dohodě s vedoucím navrhněte způsob generování programového kódu v různých cílových jazycích na základě definic objektů a vlastností ve zdrojové ontologii.
4. Implementujte navržený generátor pomocí vhodných technologií.
5. Proveďte testování vytvořeného řešení na různých vstupních datech.
6. Zhodnoťte dosažené výsledky.

Literatura:

- OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation 11 December 2012, http://www.w3.org/TR/owl-overview

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

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

Vedoucí práce:	**Burget Radek, doc. Ing., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	18. května 2022
Datum schválení:	11. října 2021

Abstrakt

Táto diplomová práca sa venuje procesu generovania programového kódu z definície ontológie. Pri vývoji ontologických aplikácií je výhodné mať definíciu ontológie vo forme zdrojového kódu. Takýto kód vyjadruje ontologické triedy a vlastnosti. Hlavným cieľom práce je vytvoriť implementáciu nástroj, ktorý využíva tento proces. Práca popisuje návrh, implementáciu a testovanie tohto nástroja. Výsledkom práce je plne funkčný generátor zdrojového kódu. Je schopný generovať výstupný kód v programovacích jazykoch Java a Python. Funkčnosť a spoľahlivosť bola vyhodnotená na základe testovania so sadou reálne využívaných ontológií.

Abstract

This master thesis deals with the process of generating source code from ontology definition. It is advantageous to have ontologies in the source code representation, when developing ontological applications. This source code expresses ontology classes and properties. The main goal of this thesis is to develop an implementation of the tool that uses this process. The thesis describes the design, implementation and testing of this tool. The result is a fully functional source code generator. It is able to generate Java or Python source code. Functionality and reliability were evaluated according to testing with set of actually used ontologies.

Kľúčové slová

Sémantický web, Ontológia, RDF, RDFS, OWL, Generátor, Generovanie zdrojového kódu, Java, RDF4J

Keywords

Semantic web, Ontology, RDF, RDFS, OWL, Generator, Source code generation, Java, RDF4J

Citácia

SVETLÍK, Tomáš. *Generování programového kódu z definice ontologie*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Radek Burget, Ph.D.

Generování programového kódu z definice ontologie

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána doc Ing. Radka Burgeta Ph.D.. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Tomáš Svetlík
16. mája 2022

Poďakovanie

Chcel by som poďakovať vedúcemu práce doc Ing. Radkovi Burgetovi Ph.D. za čas a cenné rady, ktoré mi venoval počas priebehu tvorby tejto diplomovej práce.

Obsah

Kapitola 1

Úvod

Sémantický web je stále pre väčšinu ľudí pomerne neznámym pojmom. Postupne sa však stáva viac a viac rozvinutejším a jeho technológie sú rozšírenejšie.

Jednou z kľúčových technológií sémantického webu je ontológia. Je možné ju popísať ako formalizovaný popis určitej domény informácií. Popisované sú pojmy a ich vzájomné vzťahy. Menej rozvinutou oblasťou sémantického webu sú aplikácie, ktoré pracujú s dátami popísanými pomocou ontológie. Rozvoj tejto oblasti je do značnej miery limitovaný dostupnosťou vývojárov schopných vytvárať takéto aplikácie. Týchto vývojárov je ale nedostatok. Zvyčajne sú vývojári zvyknutý na objektovo orientované programovanie. Vývoj ontologických aplikácií potrebuje rozdielny prístup. Pre prácu s dátami z ontológie je potrebná manipulacia s RDF dátovým modelom, ktorý uchováva ontológiu. Táto manipulácia sa vykonáva pomocou rôznych aplikačných rozhraní určených pre manipuláciu s modelom. Mnohí vývojári nemajú žiadnu skúsenosť s takýmto prístupom. Učenie sa niekoľkých nových technológií a postupov súčasne je často až príliš komplikované. To je dôvodom, ktorý vývojárov odrádza od tvorby ontologických aplikácií.

Výhodou pre vývoj by bolo mať ontologické prvky namapované ako triedy v zdrojovóm kóde a pracovať s nimi na spôsob objektovo orientovaného programovania. Prinieslo by to zjednodušenie práce s ontológiami a hlavne by sa rozšírila skupina vývojárov schopných vytvárať aplikácie založené na sémantickom webe. Manuálne mapovanie ontológie do programových tried ale zaberá určitý čas. Táto práca práve preto prináša spôsob, ako jednoducho, rýchlo a spoľahlivo vytvárať zdrojový kód z ontológie.

Cieľom tejto práce je navrhnúť a vytvoriť nástroj, pomocou ktorého je z definície ontológie vygenerovaný zdrojový kód. Vygenerovaný zdrojový kód cieľového jazyka definuje triedy a ich vlastnosti. Takýto nástroj poskytuje podporu programátorom pri mapovaní ontológie do zdrojového kódu, šetrí čas a znižuje riziko chýb.

Táto práca pozostáva z ôsmich kapitol. Po tomto úvode nasleduje kapitola 2 venovaná všeobecnému popisu základných sémantických technológií. Jej úvod bližšie popisuje pojem sémantický web a predstavuje jeho koncept. Kapitola sa ďalej zaoberá jednotlivými vybranými technológiami podstatnými pre túto prácu. Týmito technológiami sú Resource Description Framework (RDF) a ontológia. Nasledujúca kapitola 3 pojednáva o najrelevantnejších ontologických jazykoch. Prvým predstaveným jazykom je RDF Schéma. Detailnejšie je tiež opísaný jazyk OWL, ktorý je jedným z najvyužívanejších jazykov pre definovanie ontológií. Kapitola 4 poukazuje na existujúce softvérové nástroje. V jej prvej časti sú popísané nástroje na prácu a spracovanie RDF a OWL. Druhá časť predstavuje nástroje pre generovanie programového kódu pomocou šablón. Kapitola 5 je vstupom do praktickejšej časti tejto práce. Kapitola začína analýzou požiadaviek a analýzou už existujúcich podobných riešení

nástrojov pre generovanie kódu z ontológie. Súčasťou kapitoly sú návrh architektúry vytváraného nástroja a návrh mapovania ontologických entít do cieľových jazykov. Posledné, čo kapitola obsahuje, je návrh štruktúry generovaného kódu. V kapitole 6 sa predstavuje proces implementácie. Sú v nej uvedené použité implementačné nástroje, štruktúra implementácie a popisy implementácie jednotlivých dôležitých častí generátora. Popis procesu testovania a vyhodnotenie funkčnosti výsledného nástroja sa uvádza v kapitole 7. Poslednou kapitolou 8 je záver, ktorý zhrňuje dosiahnuté výsledky.

Kapitola 2

Sémantický web a Ontológia

Pojem „Sémantický web“ bol prvý krát spomenutí Timom Berners-Leem, tvorcom súčasného internetu, v roku 1999 pri predstavovaní vízie pre web. Vízia znela nasledovne:

„*I have a dream for the Web in which computers become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A "Semantic Web", which makes this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The ïntelligent agents"people have touted for ages will finally materialize.*“ [3]

V roku 2001 Berners-Lee, spolu so svojím tímom združeným v konzorciu W3C[1], prezentoval ideu sémantického webu [4]. Idea bola predstavená ako riešenie vtedajšieho stavu internetu. Ten bol podľa nich len spleť informácií, v ktorých bolo stále zložitejšie nájsť relevantné údaje. Väčšina obsahu webu bola určená len pre ľudské čítanie a nie na spracovávanie softvérmi. Predstavenou ideou bola vízia o webe prepojených údajov. Vízia hovorí, že sémantický web prináša štruktúru do zmysluplného obsahu webových stránok. Cieľom tejto štruktúry je zjednodušenie zdieľania informácií na webe a umožnenie spracovania informácií na webe pomocou softvérových nástrojov. Koncept sémantického webu sa postupne rokmi lepšie špecifikoval a rozvíjal. Vzniklo množstvo sémantických technológií a nástrojov. Tieto technológie sa však ešte nerozšírili dostatočne, takže Berners-Leeho konštatovanie stavu internetu je do veľkej miery aktuálne aj dnes.

Sémantický web sa často nazýva aj ako web dát. Takéto pomenovanie vyplýva z toho, že základnou jednotkou sú dáta a nie stránky ako pri klasickom webe. Pod názvom sémantický web nie je myslený nejaký nový web. Jedná sa len o rozšírenie už existujúceho webu. Toto rozšírenie predstavuje určitú formu špecifikácie relevantných dát z webu.

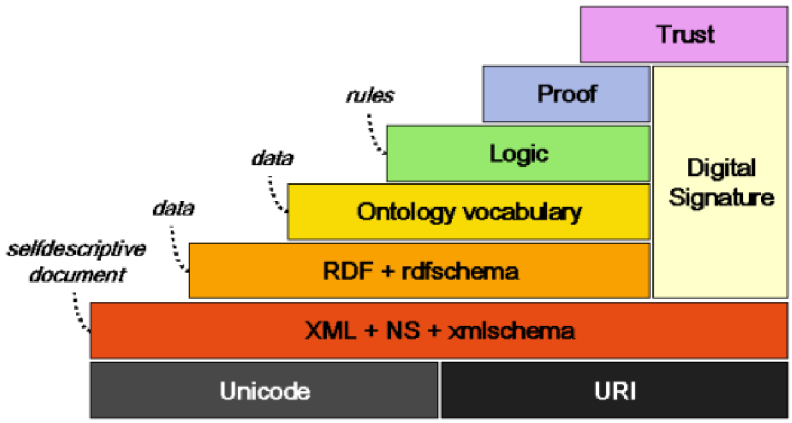
Súčasťou sémantického webu sú rozličné technológie. Ich predstavenie a usporiadanie do vrstiev je uvedené v nasledujúcej sekcií 2.1. Podrobnejšie sa táto kapitola zameriava na dvojicu technológií, ktoré sú podstatné v ďalších fázach tejto práce. Hlavnou technológiou, ktorá tvorí základ sémantického webu, je *Resource Description Framework* (RDF) [27]. RDF je venovaná sekcia 2.2. V záverečnej fáze tejto kapitoly 2.3 je podrobnejšie predstavená ďalšia dôležitá technológia, ktorou je ontológia.

[1] https://www.w3.org/

2.1 Vrstvy sémantického webu

Princíp sémantického webu pozostáva z viacerých technológií, ktoré sú usporiadané do niekoľkých vrstiev. Vrstvy vytvárajú architektúru, ktorá sa nazýva *The Semantic Web Stack* a je zobrazená na obrázku 2.1. Jednotlivé vrstvy medzi sebou dodržiavajú dva princípy [14]:

- **Zostupná kompatibilita** – technológie na určitej vrstve vedia interpretovať a používať informácie z nižších úrovní
- **Čiastočné porozumenie smerom nahor** – technológie na určitej vrstve získavajú aspoň čiastočné výhody z informácií na vyššej vrstve, pri plnej vedomosti o danej vrstve



Obr. 2.1: The Semantic Web Stack [14].

Najspodnejšou časťou architektúry sú vrstvy Unicode a URI. Unicode zabezbečuje, že používame medzinárodnú znakovú sadu a vďaka prostriedkom z URI je možné jednoznačne identifikovať objekty z reálne sveta v sémantickom webe [14].

Nasledujúcou časťou je XML. Je to štandardizovaný značkovací jazyk, ktorý je vhodný na odosielanie dát cez internet. Tento jazyk umožňuje písať štrukturované webové dokumenty so slovnou zásobou definovanou užívateľom [19]. Táto schopnosť sa využíva v sémantickom webe.

Ďalšiu vrstvu tvorí technológia RDF spolu s RDF Schema. Sú to technológie poskytujúce základné pravidlá pre definovanie výrokov o objektoch. RDF často využíva syntax na základe XML.

Vrstva ontológie poskytuje podporu pre vývoj slovnej zásoby, ktorá slúži k definícií výrokov o objektoch. Napomáha k vzniku nových vzťahov z už existujúcich vzťahov.

Dôležitou súčasťou sémantického webu je tiež digitálny podpis (Digital Signature), ktorý zaručuje dôveryhodnosť a autenticitu dokumentov na webe [14]. Digitálny podpis môže tiež slúžiť k identifikácií zmien v dokumente.

Najvyššie položené vrstvy, Logika, Dôkaz a Dôvera sú stále v procese vývoja pracovníkmi konzorcia W3C. Vrstva Logika sa používa na automatické odvodenie dát z ontológie, čo vedie k vzniku deklaratívnych špecifických aplikácií vedomostí. Dôkaz by mal následne prinášať deduktívny proces a overenie či sú údaje správne. Dôvera sa dosiahne po vyriešení všetkých vrstiev, keď používatelia budú dôverovať ich operáciam a kvalite poskytovaných informácií. [1]

2.2 RDF

RDF (*Resource Description Framework*) je špecifikácia rámcu pre vyjadrenie informácií o zdrojoch z reálneho sveta. Jeho úlohou je popis, znovupoužitie a výmena metadát. Konzorcium W3C predstavuje štandard RDF ako technologický základ sémantického webu. Táto sekcia vychádza z jednej časti oficiálnej špecifikácie RDF [21].

RDF žiadnym spôsobom neovplyvňuje informácie, ktoré sa zobrazujú ľudom. Jeho zmysel je naplnený pri situácií, kedy je potrebné, aby sa informácie na webe spracovávali pomocou aplikácií. Vďaka špecifikovaniu rámca získa aplikácia schopnosť porozumieť dátam. Tento rámec na spracovanie dát je všeobecný, čo umožňuje vymieňanie informácií medzi aplikáciami bez straty údajov.

Využitie RDF môžeme nájsť vo viacerých oblastiach. Jednou z možností môže byť rozšírenie súboru údajov jeho prepojením o súbory z tretích strán. Týmto krokom sa užívateľ vie dostať k väčšiemu množstvu informácií. Aplikovaním strojovo čitateľných informácií na webové stránky sa dá zlepšiť proces vyhľadávania alebo rozlíšiť obsah, čo umožňuje vylepšiť zobrazovaciu štruktúru dát. Informácie sa tiež razom stávajú dostupné pre automatické spracovávanie aplikáciami tretích strán.

2.2.1 Datový model RDF

Podstatou RDF je vytváranie výrokov. Tieto výroky sú štruktúrované do jednoduchého formátu tzv. RDF trojice. RDF trojica je základným prvkom RDF. Skladá sa z nasledujúcich troch položiek:

$$<subjekt>\ <predikát>\ <objekt>$$

RDF výrok vyjadruje vzťah medzi dvoma zdrojmi, nazývanými subjekt a objekt. Tento vzťah je vždy definovaný smerovo, pričom smer každého vzťahu vedie od subjektu k objektu. V rámci RDF sa tento vzťah nazýva vlastnoť. Predikát je nositeľom danej vlastnosti subjektu a objekt definuje hodnotu tejto vlastnosti. Rovnaké zdroje sa môžu vyskytovať v rôznych trojiciach. Vďaka tomu je možné nachádzať spojitosti medzi trojicami.

Na ukážke 2.1 sú zobrazené príklady RDF trojíc. Je tam vidieť, že v štyroch prípadoch je subjektom zdroj „Bob“. Ďalším viacnásobne použitým zdrojom je „the Mona Lisa“. Tento zdroj sa nachádza v jednej trojici ako subjekt a v dvoch trojiciach ako objekt.

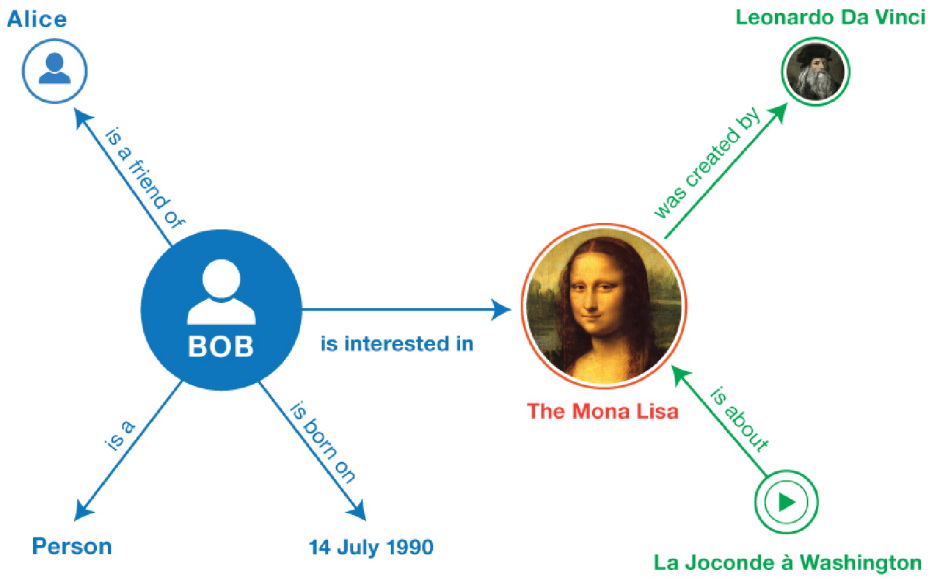
```
1 <Bob> <is a> <person>
2 <Bob> <is a friend of> <Alice>
3 <Bob> <is born on> <the 4th of July 1990>
4 <Bob> <is interested in> <the Mona Lisa>
5 <the Mona Lisa> <was created by> <Leonardo da Vinci>
6 <the video La Joconde a Washington> <is about> <the Mona Lisa>
```

Výpis 2.1: Ukážka jednoduchých príkladov RDF trojíc zapísaných v pseudokóde [21]

2.2.2 RDF grafy

Grafická reprezentácia usporiadanej trojice hodnôt subjekt, predikát a objekt sa nazýva RDF graf. RDF graf sa zobrazuje pomocou uzlov prepojených orientovanými hranami. Uzly sú reprezentáciou subjektov a objektov. Hrany symbolizujú predikáty. RDF trojica je tvorená pomocou dvoch uzlov a orientovanej hrany. Orientácia hrany je vždy určená smerom od subjektu k objektu. Jednotlivé trojice je možné spájať bez toho, aby sa zmenil

ich význam. Spájaním jednotlivých trojíc sa zvyšuje zložitosť grafu. Ako taký graf vyzerá je možné vidieť na obrázku 2.2.



Obr. 2.2: RDF graf zobrazujúci trojice z ukážky 2.1 [21]

2.2.3 Typy RDF údajov

Hodnoty, ktoré sa vyskytujú v trojiciach, môžu mať tri rôzne podoby:

- **IRI** – IRI je skratka pre *International Resource Identifier*, čo v preklade znamená medzinárodný identifikátor zdroja. Je to kompaktný reťazec znakov, ktorý identifikuje zdroj. Pojem IRI je zovšeobecnením identifikátora URI (*Uniform Resource Identifier*), ktorý rozširuje použiteľnú znakovú sadu v reťazci znakov IRI aj o znaky iné ako ASCII znaky. IRI sa používajú na identifikáciu zdrojov, ako sú dokumenty, ľudia, fyzické predmety a abstraktné pojmy. V RDF trojici sa IRI využíva vo všetkých troch pozíciach. Predikát musí byť vždy vo forme IRI.
- **Literály** – Literály vyjadrujú základné hodnoty. Tieto hodnoty sú spojené s dátovými typmi, ktoré ich umožňujú správne analyzovať a interpretovať. Koncept RDF poskytuje rôzne dátové typy ako napr. *string*, *int*, *date* či *boolean*. V rámci RDF trojíc sa literály môžu objaviť iba v pozícií objektu.
- **Prázdne uzly** – V niektorých prípadoch je užitočné uvažovať o zdrojoch bez obťažovania sa s použitím globálneho identifikátora (IRI). Zdroj bez globálneho identifikátora, môže byť reprezentovaný prázdnym uzlom. Prázdne uzly predstavujú nejakú vec bez toho aby vyjadrovali hodnotu. Toto vyjadrenie sa môže objaviť v pozícii subjektu alebo objektu trojice.

2.2.4 Syntax RDF

Pre syntax RDF nie je priamo definovaný určitý formát. Doteraz bol v tejto kapitole RDF dátový model zobrazovaný vo forme abstraktnej syntaxe, t.j. dátového modelu ne-

závislého od konkrétnej syntaxe. Rôzne konkrétne syntaxe môžu vytvoriť presne rovnaký graf z pohľadu abstraktnej syntaxe. Existuje množstvo rôznych formátov na zapisovanie grafov RDF. Rôzne formáty zápisu toho istého grafu sú však ekvivalentné, pretože vedú k úplne rovnakým trojiciam. V súčastnosti existuje niekoľko konkrétnych syntaxí používaných k zápisu RDF výrokov. Bežnými formátmi sú RDF/XML, Turtle, N-Triples, N-Quads, JSON-LD, TriG a TriX. Podrobnejšie sú popísané tri najpoužívanejšie z nich.

N-Triples

N-Triples poskytuje jednoduchý riadkový spôsob zápisu RDF trojíc [8]. Každý riadok obsahuje jednu takúto trojicu. Formát N-Triples je zobrazený na ukážke 2.2.

Identifikátory zdrojov sú v tomto formáte uzavreté v lomených zátvorkách. Bodka na konci riadku symbolizuje ukončenie trojice. Literáry sa miesto zátvoriek zapisujú do úvodzoviek, viď zápis hodnoty „Mona Lisa“ v ukážke na riadku 5. Dátový typ sa k literáru pripája pomocou oddeľovača `^^`, viď riadok 3.

N-Triples sa často využíva na výmenu veľkého množstva RDF výrokov a na spracovanie rozsiahlich RDF grafov pomocou nástrojov pre spracovanie textu. Nevýhodou môže byť menšia prehľadnosť pre užívateľa.

```
1 <http://example.org/bob#me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
      xmlns.com/foaf/0.1/Person> .
2 <http://example.org/bob#me> <http://xmlns.com/foaf/0.1/knows> <http://example.org/alice
      #me> .
3 <http://example.org/bob#me> <http://schema.org/birthDate> "1990-07-04"^^<http://www.w3.
      org/2001/XMLSchema#date> .
4 <http://example.org/bob#me> <http://xmlns.com/foaf/0.1/topic_interest> <http://www.
      wikidata.org/entity/Q12418> .
5 <http://www.wikidata.org/entity/Q12418> <http://purl.org/dc/terms/title> "Mona Lisa" .
6 <http://www.wikidata.org/entity/Q12418> <http://purl.org/dc/terms/creator> <http://
      dbpedia.org/resource/Leonardo_da_Vinci> .
7 <http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619> <http://
      purl.org/dc/terms/subject> <http://www.wikidata.org/entity/Q12418> .
```

Výpis 2.2: Ukážka syntaxe N-Triples, ktorá popisuje príklad 2.1 [21]

Turtle

Turtle je rozšírením formátu N-Triples. Rozšírenie spočíva v zavedení množstva syntaktických značiek, ako napríklad podpora pre predpony menného priestoru, zoznamy a skratky pre literály s dátovým typom [7]. Syntax Turtle je kompromisom medzi jednoduchosťou písania, čitateľnosťou a jednoduchou analýzou. Táto syntax je zobrazená na ukážke 2.3.

Identifikátory zdrojov sa môžu v Turtle zapisovať ako relatívne či absolútne IRI a tiež i pomocou prefixov. Relatívne a absolútne IRI sú tak, ako pri N-Triples uzavreté v hranatých zátvorkách. Relatívne hodnoty, viď riadok 8, sa viažu na aktuálnu základnú IRI. Tá sa definuje pomocou direktívy *BASE*, viď riadok 1. Prefixy zjednodušujú a sprehľadňujú zápis IRI hodnôt, keď miesto celej hodnoty sa používa iba prefix. Prefix sa definuje direktívou *PREFIX*, prefixovým menom a prefixovou hodnotou oddelenou od mena pomocou znaku dvojbodky, viď riadky 2 až 6. Následné využitie predikátov môžeme vidieť napríklad na riadku 10.

Často nastáva situácia, že jeden subjekt odkazuje na viacero predikátov. Ako je možné vidieť na riadkoch 8 až 12, Turtle poskytuje pre takéto prípady možnosť skrátenia zá-

pisu. Predikát spolu s objektom tvorí pár, ktorý sa od ostatných párov patriacich k ďalším RDF trojiciam oddeľujú pomocou znaku bodkočiarky.

Znak *a* v predikátovej pozícií, viď riadok 9, je špeciálny typ syntaktickej značky a predstavuje identifikátor *http://www.w3.org/1999/02/22-rdf-syntax-ns#type*

```
1  BASE <http://example.org/>
2  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
4  PREFIX schema: <http://schema.org/>
5  PREFIX dcterms: <http://purl.org/dc/terms/>
6  PREFIX wd: <http://www.wikidata.org/entity/>
7
8  <bob#me>
9      a foaf:Person ;
10     foaf:knows <alice#me> ;
11     schema:birthDate "1990-07-04"^^xsd:date ;
12     foaf:topic_interest wd:Q12418 .
13
14 wd:Q12418
15     dcterms:title "Mona Lisa" ;
16     dcterms:creator <http://dbpedia.org/resource/Leonardo_da_Vinci> .
17
18 <http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619>
19     dcterms:subject wd:Q12418 .
```

Výpis 2.3: Ukážka syntaxe Turtle, ktorá popisuje príklad 2.1 [21]

RDF/XML

Formát RDF/XML poskytuje XML syntax pre dátový model RDF [20]. Už z názvu vyplýva, že tento formát je odvodený od jazyka XML. K zapísaniu RDF tvrdenia sa používajú elementy, ich obsah a atribúty s hodnotami. V začiatkoch RDF bola XML syntax pre RDF jediná syntax a aj preto sa definuje ako základná syntax pre RDF. Je to najviac využívaná syntax. Príklad zápisu výrokov v tomto formáte je zobrazený v ukážke 2.4.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF xmlns:dcterms="http://purl.org/dc/terms/"
3           xmlns:foaf="http://xmlns.com/foaf/0.1/"
4           xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5           xmlns:schema="http://schema.org/">
6      <rdf:Description rdf:about="http://example.org/bob#me">
7          <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
8          <schema:birthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
               1990-07-04</schema:birthDate>
9          <foaf:knows rdf:resource="http://example.org/alice#me"/>
10         <foaf:topic_interest rdf:resource="http://www.wikidata.org/entity/Q12418"/>
11     </rdf:Description>
12     <rdf:Description rdf:about="http://www.wikidata.org/entity/Q12418">
13         <dcterms:title>Mona Lisa</dcterms:title>
14         <dcterms:creator rdf:resource="http://dbpedia.org/resource/Leonardo_da_Vinci"/>
15     </rdf:Description>
16     <rdf:Description rdf:about="http://data.europeana.eu/item/04802/243
           FA8618938F4117025F17A8B813C5F9AA4D619">
17         <dcterms:subject rdf:resource="http://www.wikidata.org/entity/Q12418"/>
18     </rdf:Description>
19 </rdf:RDF>
```

Výpis 2.4: Ukážka syntaxe RDF/XML, ktorá popisuje príklad 2.1 [21]

RDF trojice sa v RDF/XML syntaxi špecifikujú v rámci tagu *rdf:RDF*. Atribúty tohto počiatočného tagu poskytujú priestor k definovaniu prefixov pre hodnoty zdrojov, viď riadky 2 až 5. XML element *rdf:Description* slúži k vyjadreniu množiny RDF trojíc s rovnakým subjektom. Subjekt sa definuje ako atribút tohto XML elementu. Všetky podelementy daného prvku sú jeho vlastnosti, čo znamená, že každý podelement predstavuje jednu RDF trojicu. Takýto element so štyrmi vlastnosťami je možné vidieť napríklad na riadkoch 6 až 11. Názvy podelementov určujú predikát trojice. V prípade že objektom je hodnota typu IRI, táto hodnota sa uvádza ako atribút s tagom *rdf:resource*, viď riadok 7. Ak je objektom trojice literál, hodnota sa uvádza ako obsah elementu, viď riadok 8. Na rovnakom riadku je tiež vidieť, že dátové typy literálov sa zapisujú ako atribúty elementu s tagom *rdf:datatype*.

2.3 Ontológia

Vízia sémantického webu od začiatku upriamuje pozornosť na ontológie. Berners-Lee s tímom ich označili za chrbtovú kosť sémantického webu [3].

Ontológia je pôvodne filozofický pojem, ktorý sa prevzal aj do sveta informatiky. Definície tohto pojmu sa však v každom odvetví odlišujú. Vo filozofickom vnímaní ontológia znamená náuku o bytí prípadne tiež univerzálnu sústavu znalostí, ktoré popisujú objekty, javy a zákonitosti sveta [23]. V informatike existuje veľké množstvo definícií pre pojem ontológia. Jednu z najzákladnejších formulácií uviedol v roku 1993 T. Gruber a hovorí [11]: „*Ontologia je explicitná špecifikácia konceptualizácie.*“ V roku 1997 túto základnú definíciu trochu pozmenil W. Borst [5]: „*Ontológia je formálna, špecifikácia zdieľanej konceptualizácie.*“ Na definície Grubera a Borsta nadviazal v roku 1998 R. Studer, ktorý ich zjednotil do jednej formulácie [22]: „*Ontológia je formálna, explicitná špecifikácia zdieľanej konceptualizácie.*“

Gruberova definícia nám vyjadruje, že konceptualizácia, ktorá je chápaná ako systém pojmov modelujúcich časť sveta, je špecifikovaná explicitne. Znamená to, že konceptualizácia nie je len skrytá v predstave, ale je jednoznačne definovaná. Borst vo svojej definícii ontológie vynecháva explicitnú špecifikáciu ale rozširuje ju o požiadavky na formálnosť a zdieľanie. Pod formálnosťou je myslené reprezentovanie ontológie jazykom, ktorý musí mať presne definovanú syntax. Zdieľanie znamená, že ontógia by nemala byť individuálna záležitosť, ale výsledkom dohody určitej cieľovej skupiny ľudí. [23]

Vo všeobecnosti je možné povedať, že ontológia je formálny popis určitej domény vedomostí [11]. Tieto vedomosti zahŕňajú základné pojmy modelovaného sveta a ich vlastnosti. Zmyslom takéhoto formálneho popisu je zdieľanie významov pojmov, ktoré následne zabezbečuje jednotné chápanie takýchto pojmov. Z praktického pohľadu sa dá teda o ontológií hovoriť ako o nástroji pre zdieľanie významov pojmov z určitej oblasti.

Ontológie môžu mať rôzne zamerania. Z pohľadu odborov môžu byť ontológie terminologické (zoznam termínov danej oblasti), informačné (rozvinuté databázové konceptuálne schémy) či znalostné (logické teórie, aplikácia hlavne v umelej inteligencií). Iný pohľad môže byť zameraný na predmet formalizácie a v tomto prípade sú ontológie generické (obecné zákonitosti a vzťahy), doménové (konkrétne oblasti) a aplikačné (prispôsobené konkrétnej aplikácií). [23]

2.3.1 Účel ontológií

Výsledný účel ontologií sa rozdeľuje do troch základných kategórií využitia [25]:

- **Komunikácia** – podpora dorozumievania sa medzi ľuďmi
- **Interoperabilita** – podpora dorozumievania sa medzi počítačovými systémami
- **Systémové inžinierstvo** – zjednodušenie návrhu aplikácií orientovaných na znalosti (napr. pojmové vyhľadávanie, inteligentné výukové systémy, spracovanie prirodzeného jazyka)

2.3.2 Štruktúra ontológie

Ontológia je typicky reprezentovaná konečným zoznamom pojmov a vzťahmi medzi týmito pojmami. Napriek existencií viacerých typov ontológie, ich základná štruktúra je prakticky vo všetkých projektoch, jazykoch a nástrojoch podobná. Základná štruktúra pozostáva z nasledujúcich šiestich prvkov [23]:

- **Triedy** (známe tiež ako koncepty či kategórie) sú abstraktné množiny, ktoré označujú konkrétne objekty. Je to základný prvok znalostnej domény. Predstavujú skupinu, ktorých členovia zdieľajú spoločné vlastnosti. Množina tried zachováva hierarchické usporiadanie, čím vytvára určitú hierarchiu. Ako príklad triedy sa môže uviesť trieda *škola*. Táto trieda môže byť tvorená podtriedami ako *univerzita* či *gymnázium*.
- **Individuá** (známe tiež ako objekty či inštancie) sú komponenty ontológie, ktoré predstavujú konkrétne objekty reálneho sveta. Zvyčajne je to prvok z konceptu alebo triedy a vtedy sa o ňom dá hovoriť ako o inštancií. Nie všetky jazyky podporujú vloženie individua bez naviazania na konkrétnu triedu. Príkladom inštancie môže byť napríklad *VUT*, ktoré je inštanciou triedy *škola* a podtriedy *univerzita*.
- **Vlastnosti** (relácie, sloty, atribúty) sa používajú na vyjadrenie vzťahov medzi dvoma konceptami v danej doméne. V tradičných jazykoch môžu byť vlastnosti špecifikované pomocou ľubovoľných logických podmienok. V iných jazykoch im je možné priradiť iba preddefinované obmedzenia. špeciálny typ vlastnosti, ktorá je viazaná iba medzi dvoma objektami (nie triedami) sa nazýva slot. Napríklad *študuje* môže byť uvedené ako vlastnosť medzi triedou *osoba* a triedou *univerzita*.
- **Meta-sloty** je označenie pre vlastnosti, ktoré sú priradené slotu. Hierarchický vzťah podradeného a nadradeného slotu je najčastejším prípadom meta-slotu. Inými vlastnosťami slotu sú napríklad definičný obor a obor hodnôt, ktoré sa vymedzujú konkrétnymi triedami. Tieto vlastnosti sa vzťahujú ku slotu bez ohladu na jeho použitie, a preto sa označujú ako globálne obmedzenia. O lokálnych obmedzeniach sa hovorí v prípade, keď je potrebné obmedziť hodnoty slotu, ktorý je aplikovaný na konkrétnu triedu. Jedná sa hlavne o obmedzeniach kardinality a oboru hodnôt slotu.
- **Primitívne dátové typy** môžu byť použité na definovanie oboru hodnôt dátotypového slotu. Takýto slot je špeciálny typ slotu, v ktorom je jeho argument reprezentovaný primitívnou hodnotou a nie objektom.
- **Axiómy** (pravidlá) sú logické výrokové formule. Okrem iných výrazov, ktoré explicitne určujú príslušnosť k triedam a reláciam, môžu byť i tieto formule obsahom ontológie. Axiómy môžu vyjadrovať napríklad ekvivalenciu tried či relácií, disjunkciu tried, rozklad tried na podtriedy a iné. Obvykle bývajú súčasťou definície tried a relácií.

Kapitola 3

Jazyky ontológie

Aby boli ontológie zrozumiteľné aj pre počítače, musia byť zapísané v strojovo čitateľnom jazyku. Všeobecne existuje množstvo formálnych jazykov ontológií. Táto práca sa však zameriava len na jazyky, ktoré sú úzko späté s webovými technológiami. V sémantickom webe sa na reprezentovanie ontológie v súčastnosti používa niekoľko jazykov. Najdôležitejšie a najznámejšie z týchto jazykov sú RDF Schéma a OWL. V časti 3.1 tejto kapitoly sa upriamuje pozornosť na základné rysy jazyka RDF Schéma. Táto časť je spracovaná z informácií získaných z oficiálnej špecifikácie tohto jazyka [6]. Následná sekcia 3.2 predstavuje jazyk OWL a jej zdrojom sú časti oficiálnej špecifikácie [28], [12] a [15].

3.1 RDF Schéma

RDF Schéma, skrátene RDFS, je sémantické rozšírenie formátu RDF. Jeho prvotná verzia bola vytvorená konzorciom W3C a bola zverejnená v roku 1998. V tej dobe to bol vôbec prvý jazyk orientovaný na RDF. V roku 2014 bola pubikovaná doposiaľ posledná verzia 1.1.

RDFS poskytuje novú slovnú zásobu, ktorá je využívaná na modelovanie dátového modelu RDF. Táto slovná zásoba rozširuje štruktúru RDF o základné ontologické prvky, ako napríklad triedy a binárne relácie s možnosťou určiť definičný obor a obor hodnôt. Prvky prinášajú mechanizmus pre špecifikovanie skupín súvisiacich zdrojov, vzťahov vyskytujúcich sa medzi zdrojmi či hierarchického usporiadania týchto skupín a vzťahov. RDFS poskytuje veľmi jednoduchý spôsob definovania štruktúr, ktorý ale nie je dokonalý. V porovnaní s inými ontologickými jazykmi, RDFS napríklad neponúka možnosť bližšieho špecifikovania príslušnosti ku triedam (lokálne obmedzenia).

Základná slovná zásoba jazyka RDFS je definovaná v mennom priestore, ktorý sa neformálne nazýva ako *rdfs*. Tento menný priestor je možné identifikovať pomocou IRI *http://www.w3.org/2000/01/rdf-schema#* a bežne sa využíva prefix *rdfs:*.

V nasledujúcich častiach tejto podkapitoly sú postupne predstavené jednotlivé prvky z jazyka RDFS.

3.1.1 Triedy

Triedami v jazyku RDFS sú skupiny, do ktorých sa môžu rozdeľovať jednotlivé zdroje. Členovia tried sú inštanciami danej triedy. K identifikácií tried často slúži IRI a k ich popisu sa používajú vlastnosti RDF. Na priradenie inštancie k určitej triede sa používa vlastnosť *rdf:type*.

Ku každej triede existuje množina jej inštancií, ktorá sa nazýva rozšírenie triedy. Dve triedy môžu mať rovnakú množinu inštancí, ale pri tom to stále môžu byť rozdielne triedy. Takéto dve množiny inštancií sa v takom prípade popisujú pomocou rozličných vlastností. Trieda môže byť tiež súčasťou svojho vlastného rozšírenia triedy, čo znamená, že môže byť inštanciou samej seba. Používané normatívne triedy sú:

- **rdfs:Class** – je to trieda všetkých zdrojov, ktoré sú RDF triedami. Táto trieda je inštanciou sama na seba.
- **rdfs:Resource** – inštanciami tejto triedy sú všetky objekty, ktoré sú popisované pomocou RDF. Všeobecne ju môžeme nazvať ako trieda všetkého, takže všetky ostatné triedy sú jej podtriedami.
- **rdfs:Literal** – trieda literálových hodnôt, napríklad typu integer a string. Táto trieda je inštanciou *rdfs:Class* a podtriedou *rdfs:Resource*.
- **rdfs:Datatype** – je triedou zameranou na dátové typy. Táto trieda je inštanciou a rovnako aj podtriedou *rdfs:Class*. Všetky jej inštancie sú podtriedou *rdfs:Literal* a zodpovedajú modelu dátového typu v RDF.
- **rdf:Property** – určuje triedu RDF vlastností, ktorá je inštanciou *rdfs:Class*.
- **rdf:langString** – je to trieda jazykom označených reťazcových hodnôt. Je inštanciou *rdfs:Datatype* a podtriedou *rdfs:Literal*.

3.1.2 Vlastnosti

Ako už bolo definované v časti 2.2, vlastnosť v RDF je špecifikovaná ako vzťah medzi subjektovým zdrojom a objektovým zdrojom. Táto definícia platí aj v rámci RDFS. Všetky vlastnosti sú popisované pomocou triedy *rdf:Property* a sú tiež inštanciou tejto triedy. Používané vlastnosti definované v RDFS sú:

- **rdf:type** – je to základná vlastnosť pre definície vzťahov. Používa sa pre vyjadrenie, že zdroj je inštanciou nejakej triedy. Táto vlastnosť sa aplikuje vždy pri definovaní novej vlastnosti, a to spôsobom priradenia zdroja k triede *rdf:Property*.
- **rdfs:subClassOf** – je vlastnosť slúžiaca k definícií vzťahov. Táto vlastnosť umožňuje definovanie hierarchií v systéme tried. To znamená umožnenie toho, že jedna trieda je podtriedou inej triedy (supertriedy). Všetky inštancie triedy sú taktiež inštanciami jej supertriedy.
- **rdfs:subPropertyOf** – vyjadruje ďalšiu z RDFS vlastností určených k definícií vzťahov. Pomocou nej sa dá definovať hierarchia vlastností, teda spája vlastnosti so supervlastnosťami. Vyjadruje, že všetky zdroje súvisiace s jednou vlastnosťou súvisia aj s jej nadvlastnosťou.
- **rdfs:range** – je veľmi používanou RDFS vlastnosťou pre obmedzovanie hodnôt vlastností. Jej úlohou je špecifikovanie rozsahu danej vlastnosti a vyjadrenie, že hodnoty danej vlastnosti (objekty) sú inštanciami jednej alebo viacerých tried.
- **rdfs:domain** – je ďalšou veľmi používanou RDFS vlastnosťou obmedzenia hodnôt. Jej vyjadrenie špecifikuje definičný obor vlastnosti. Táto RDFS vlastnosť uvádza, že každý zdroj, ktorý má danú vlastnosť, je inštanciou jednej alebo viacerých tried.

- **rdfs:label** – je užitočnou RDFS vlastnosťou. Poskytuje schopnosť popísania zdroja pomocou ľudsky čitateľnej verzie jeho mena. Tento popis sa následne využíva napríklad aj ako názov uzla v grafickom znázornení dátového modelu.
- **rdfs:comment** – je vlastnosť slúžiaca na ľudsky čitateľný popis zdroja, pomocou textu s dlhším rozsahom. Takýto textový komentár pomáha k pochopeniu významu tried a vlastností.

3.2 OWL

Jazyk RDFS poskytol základ pre vznik ďalšieho jazyka. Pracovná skupina v rámci konzorcia W3C publikovala v roku 2004 nový ontologický jazyk OWL (Web Ontology Language). Vývoj tohto jazyka pokračoval aj v nasledujúcich rokoch. Aktuálna verzia OWL s názvom OWL 2 bola publikovaná v roku 2009. Táto verzia je revíziou a rozšírením pôvodnej verzie. Po svojom vzniku jazyk OWL postupne nahradil jazyk RDFS, ktorý bol v určitých smeroch obmedzený.

OWL je základným jazykom sémantického webu, ktorý slúži na vyjadrenie ontológií. Je navrhnutý tak, aby predstavoval bohaté a komplexné znalosti o objektoch, skupinách objektov a vzťahov medzi objektami. Jeho návrh sa tiež snaží o to, aby uľahčil vývoj a zdieľanie ontológie cez web. Konečným cieľom je sprístupniť webový obsah pôvodne prezentovaný len ľuďom aj na spracovanie počítačom.

3.2.1 Štruktúra jazyka

Obrázok 3.1 zobrazuje diagram základnej štruktúry jazyka OWL 2. Štruktúra pozostáva z niekoľkých stavebných blokov a vzťahov definujúcich súvislosti medzi blokmi. V strede diagramu sa nachádza elipsa, ktorá predstavuje abstraktný pojem ontológie. Tento pojem si možno predstaviť ako abstraktnú štruktúru alebo RDF graf.

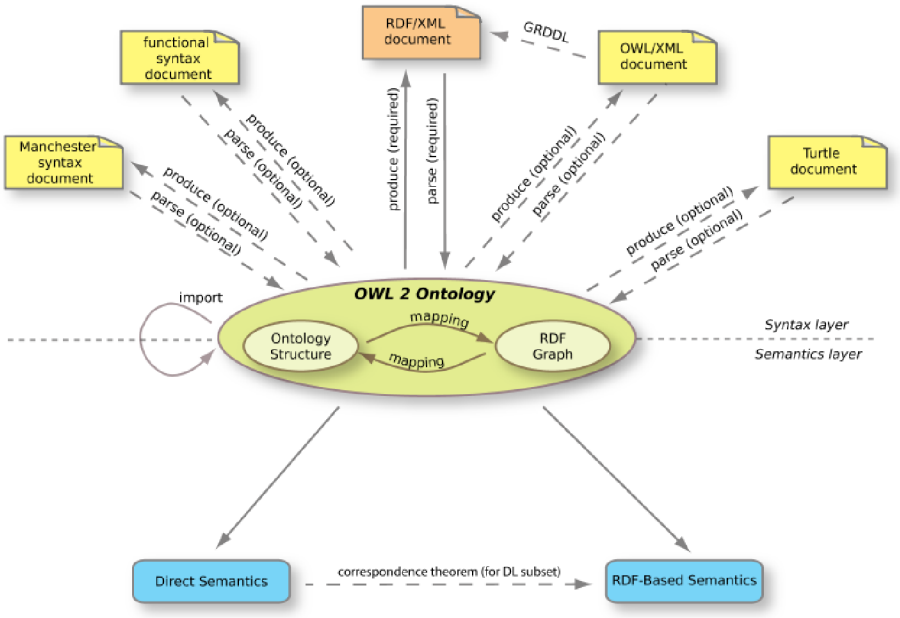
V hornej časti sa nachádzajú rôzne syntaxe. Konkrétna syntax je potrebná k ukladaniu ontológií OWL 2 a k zdieľaniu ontológií medzi nástrojmi a aplikáciami. OWL 2 nedefinuje presnú syntax a preto jeho ontológia môže byť reprezentovaná rôznymi syntaxami. Bežne používanými sú napríklad RDF serializácie RDF/XML a Turtle, predstavené už v kapitole 2.2. Ďalšími sú napríklad syntaxe OWL/XML a Manchester Syntax, ktoré sú špeciálne vytvorené pre jazyk OWL alebo funkcionálna syntax, ktorá sa však hlavne využíva k špecifikácií štruktúry jazyka. Ako primárna syntax pre výmenu ontológií je konzorciom W3C odporúčaná syntax RDF/XML. Táto syntax je teda jedinou, ktorú musia OWL 2 nástroje povinne podporovať.

Na spodnej časti je možné vidieť dve sémantické špecifikácie, *Direct Semantics* a *RDF-Based Semantics*. Sémantiky sa využívajú napríklad na zodpovedanie otázok konzistencie tried, subsumpcie a vyhľadávanie inštancií. Dve sémantiky využívané pri jazyku OWL 2 poskytujú dva alternatívne spôsoby definovania významu ontológií.

Direct Semantics priraďuje význam priamo štruktúram ontológie. Výsledkom je sémantika kompatibilná s modelovo teoretickou sémantikou popisnej logiky. Výhodou takéhoto úzkeho prepojenia je, že literatúra popisnej logiky a implementačné skúsenosti možno priamo využiť nástrojmi OWL 2. Na ontologické štruktúry sú ale pri tomto spôsobe kladené určité podmienky. Jednou z týchto podmienok je, že prechodné vlastnosti sa nemôžu použiť v obmedzeniach počtu. Ontológie spĺňajúce takéto syntaktické podmienky sa označujú OWL 2 DL.

RDF-Based Semantics priraďuje význam priamo RDF grafom a tým nepriamo aj ontologickým štruktúram, prostredníctvom mapovania do RDF grafov. Táto sémantika je plne kompatibilná s RDF sémantikou a rozširuje sémantické podmienky určené pre RDF. Keďže je možné na RDF namapovať akúkoľvek ontológiu, tak *RDF-Based Semantics* je možné použiť bez obmedzení na ktorúkoľvek ontológiu OWL 2. Pre pomenovanie RDF grafov patriacich do OWL 2 ontológie a interpretovaných pomocou tejto sémantiky sa používa označenie OWL 2 Full.

Presné a úzke prepojenie medzi týmito dvoma sémantikami poskytuje korešpondenčný teorém. Zvyčajne sa pri tvorbe v OWL 2 používa iba jeden typ syntaxe a jeden typ sémantiky. V takom prípade je diagram štruktúry výrazne jednoduchší.



Obr. 3.1: Diagram štruktúry jazyka OWL 2 [28]

3.2.2 Profily

Vo všeobecnosti je jazyk OWL veľmi expresívny jazyk. Niekedy môže byť skutočne náročné ho implementovať a pracovať s ním. Práve preto boli v rámci jazyka OWL navrhnuté profily.

Profily OWL sú určitou skrátenou verziou jazyka OWL, zvyknú sa nazývať aj podjazykmi či fragmentami. Každý profil je definovaný ako syntaktické obmedzenie štrukturálnej špecifikácie jazyka OWL. Rôzne aspekty vyjadrovacej sily OWL sa v jednotlivých profiloch vymieňajú za výpočtové alebo implementačné výhody. Takéto výhody sú dôležité pri uplatnení v konkrétnych aplikačných scenároch.

V OWL 2 existuje mnoho podtried, ktoré majú dobré výpočtové vlastnosti. Už v prvej verzií jazyka OWL boli definované tri podjazyky: OWL Lite, OWL DL a OWL FULL. V rámci tejto sekcie sú predstavené tri vybrané profily z OWL 2, ktoré boli identifikované ako najvýznamnejšie. Týmito vybranými profilmi sú OWL 2 EL, OWL 2 QL a OWL 2 RL. Počet existujúcich profilov nie je nemenný. Špecifikácia OWL 2 poskytuje jasnú šablónu na definovanie ďalších profilov.

OWL 2 EL

OWL 2 EL je profil užitočný hlavne pri aplikáciach využívajúcich ontológie s veľkým množstvom vlastností a tried. Štandardné problémy uvažovania môžu byť v týchto ontológiách vykonané za polynomický čas, ktorý je závislý na veľkosti ontológie. Tento profil vymieňa expresívnu silu za záruky výkonu. Medzi nepodporované vyjadrovacie konštrukcie patria napríklad obmedzenie kardinality, negácie tried, disjunkcia, univerzálna kvantifikácia vlastností, definovanie symetrických a asymetrických vlastností.

Získať OWL 2 EL je z mnohých veľkých ontológií orientovaných na výrazy tried pomerne jednoduché. Stačí k tomu vykonať malé zjednodušenie, ktoré však zachováva väčšinu významu pôvodnej ontológie. Použitie OWL 2 EL nie je obmedzené na nejakú užívateľskú doménu. Medzi hlavné oblasti použitia patria systémové konfigurácie, inventáre produktov a mnohé vedecké oblasti.

OWL 2 QL

OWL 2 QL je podtriedou OWL 2 zameranou obzvlášť na aplikácie využívajúce relatívne jednoduché ontológie, ktoré obsahujú veľmi veľké objemy inštancií. K údajom reprezentovaným v týchto inštanciách je vhodné alebo potrebné pristupovať priamo prostredníctvom dotazovania.

Tento profil môže byť pomerne jednoducho realizovaný pomocou štandardnej technológie relačných databáz (napr. SQL). Z toho vyplýva, že je možná úzka integrácia s RDBMS a využívanie výhod ich robustných implementácií a funkcií pre viacerých používateľov. Tak ako aj pri OWL 2 EL, riešenie problémov uvažovania trvá polynomiálny čas a výrazová sila profilu je nevyhnutne dosť obmedzená. Tento profil zakazuje napríklad obmedzenie kardinality, tranzitivné vlastnosti, univerzálne kvantifikácie vyjadrenia triedy a existenciálnu kvantifikáciu rolí k vyjadreniu triedy.

OWL 2 RL

Profil OWL 2 RL sa špecializuje na aplikácie vyžadujúce škálovateľné uvažovanie bez prílišnej straty výrazovej sily. Bol navrhnutý, tak aby prinášal výhodu aplikáciam OWL 2, ktoré sú schopné vymeniť expresívnosť jazyka za efektívnosť, a aplikáciám RDF(S), ktoré potrebujú od OWL 2 určité obohatenie vyjadrovacích schopností. Definované ontológie sú relatívne jednoduché a slúžia k organizácií veľkého počtu jednotlivcov. Tento profil v porovnaní s OWL 2 QL pracuje lepšie, ak sú už dáta vo forme RDF a manipuluje sa s nimi ako s RDF.

3.2.3 Základné modelovacie konštrukcie

Po všeobecných informáciach predstavených v predošlých častiach, sa táto sekcia presunie k implementačným detailom jazyka OWL 2. Výsledkom implementácie je OWL dokument, ktorý sa nazýva aj OWL ontológia. OWL ontológie sú vlastne súbory axiómov, ktoré poskytujú logické tvrdenia o troch typických prvkoch - triedach, jednotlivcoch a vlastnostiach. V nasledujúcej časti si teda predstavíme štandardné modelovacie konštrukcie, ktoré sa často vyskytujú v dokumente.

Hlavička

OWL dokument je vlastne RDF dokument, pretože koreňovým elementom je *rdf:RDF* element. Súčasťou tohto elementu je špecifikovanie použitých menných priestorov.

Ontológia OWL následne zvykne obsahovať jednu hlavičku, ktorá je definovaná elementom *owl:Ontology*. Hlavička zoskupuje niektoré dôležité informácie o ontológií. Jej súčasťou sú anotačné vlastnosti, ktoré neprispievajú k logickým znalostiam špecifikovaným v ontológii. Obsahuje napríklad komentáre, informácie o verziách či vnorených ontológiách. Definíciu hlavičky je možné vidieť na ukážke 3.1.

```
<owl:Ontology rdf:about="http://example.com/family2.owl#Family2">
    <owl:priorVersion>
        <owl:Ontology rdf:about="http://example.com/family.owl#Family"/>
    </owl:priorVersion>
    <owl:imports rdf:resource="http://example.com/world.owl#World"/>
    <rdfs:comment>This ontology represent family.</rdfs:comment>
</owl:Ontology>
```

Výpis 3.1: Ukážka definície hlavičky v OWL dokumente.

Triedy

Trieda je základným hierarchickým prvkom ontológií a reprezentuje usporiadanú množinu prvkov s určitou súvislosťou. Triedy sú v jazyku OWL 2 popisované použitím elementu *owl:Class*. Príklad 3.2 ukazuje definíciu novej triedy „Woman“, bez vlastností.

```
<owl:Class rdf:ID="Woman"/>
```

Výpis 3.2: Príklad definície jednoduchej triedy v jazyku OWL 2

Keďže je trieda hierarchický prvok, jazyk má schopnosť vytvárania hierarchie tried. Hierarchia sa vytvára spôsobom prevzatým z jazyka RDFS, a to pomocou elementu *rdfs:subClassOf*. Priložený príklad 3.3 vyjadruje triedu „Woman“, ktorá je podtriedou triedy „Person“.

```
<owl:Class rdf:ID="Woman">
    <rdfs:subClassOf rdf:resource="#Person"/>
</owl:Class>
```

Výpis 3.3: Príklad definície hierarchie tried v jazyku OWL 2

Triedy môžu účinne odkazovať aj na rovnaké množiny. OWL pre takýto prípad poskytuje mechanizmus, pomocou ktorého sa triedy považujú za sémanticky ekvivalentné. Tento mechanizmus sa definuje použitím elementu *owl:equivalentClass*. Príklad 3.4 uvádza, že trieda „Person“ je ekvivalentná triede „Human“.

```
<owl:Class rdf:ID="Person">
    <owl:equivalentClass rdf:resource="#Human"/>
</owl:Class>
```

Výpis 3.4: Príklad definície ekvivalencie tried v jazyku OWL 2

Členstvo v jednej triede v niektorých prípadoch vylučuje členstvo v druhej triede. Tento vzťah sa označuje ako nesúrodosť. Ako príklad 3.5 slúžia triedy „Men“ a „Woman“. Medzi týmito triedami platí, že žiadny prvok sa nemôže nachádzať v oboch triedach.

```
<owl:AllDisjointClasses>
    <owl:members rdf:parseType="Collection">
        <owl:Class rdf:about="#Woman"/>
        <owl:Class rdf:about="#Man"/>
    </owl:members>
</owl:AllDisjointClasses>
```

Výpis 3.5: Príklad definície nesúrodosti tried v jazyku OWL 2

Triedy môžu byť definované pomocou rôznorodých výrazových prostriedkov. Vyššie predstavené definície s podtriedou, ekvivalenciou a disjunkciou tried sú základnými vyjadreniami definície. OWL ale poskytuje výrazy aj pre definovanie zložitejších tried. Podporované sú napríklad výrazy pre množinové operácie, ktoré sa vyjadrujú nasledujúcimi vlastnosťami:

- **owl:intersectionOf** – Jedná sa o vlastnosť, pomocou ktorej sa definuje operácia prieniku. Trieda s takouto vlastnosťou obsahuje prvky, ktoré ležia v oboch špecifikovaných triedach .
- **owl:unionOf** – Touto vlastnosťou sa vyjadruje zjednotenie tried. Výsledná trieda obsahuje všetky prvky ležiace aspoň v jednej zo špecifikovaných tried.
- **owl:complementOf** – Výraz definuje množinovú operáciu doplnok. Takto definovaná trieda obsahuje všetky prvky, ktoré nie sú uvedené v špecifikovanej triede. Výsledkom môže byť trieda obsahujúca veľké množstvo členov.

Ako príklad tried definovaných pomocou množinových operácií je na ukážke 3.6 zobrazená definícia triedy „Mother“. Táto trieda je určená vlastnosťou prieniku z dvoch tried „Woman“ a „Parent“.

```
<owl:Class rdf:ID="Mother">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Woman"/>
        <owl:Class rdf:about="#Parent"/>
    </owl:intersectionOf>
</owl:Class>
```

Výpis 3.6: Príklad definície triedy pomocou prieniku v jazyku OWL 2

Ďalším typom konštruktorov, pre definíciu nových tried, sú obmedzenia vlastností. Konštruktor tried v týchto prípadoch zahrňuje obmedzenie určitej vlastnosti na istej aplikovanej triede. Množina novej triedy obsahuje všetky prvky z pôvodnej triedy, ktoré spĺňajú dané obmedzenie. Obmedzovať sa dá napríklad hodnota vlastností. Vlastnosti, ktoré obmedzujú hodnotu sú nasledujúce:

- **owl:allValuesFrom** – Vlastnosť popisujúca triedu všetkých jedincov, ktorých určená vlastnosť dosahuje iba hodnôt definovaných obmedzením.
- **owl:someValuesFrom** – Definuje triedu s inštanciami, pre ktoré platí, že určená vlastnosť musí nadobúdať nejakú hodnotu určenú obmedzením. Takýto prípad je zobrazený na príklade 3.7.
- **owl:hasValue** – Trieda definovaná obmedzením s touto vlastnosťou obsahuje všetky jedince, ktorých daná vlastnosť nadobúda hodnotu definovanú v obmedzení.

```
<owl:Class rdf:ID="Parent">
   <owl:Restriction>
      <owl:onProperty rdf:resource="#hasChild"/>
      <owl:someValuesFrom rdf:resource="#Person"/>
   </owl:Restriction>
</owl:Class>
```

Výpis 3.7: Príklad zobrazujúci definovanie triedy pomocou obmedzenia v jazyku OWL 2

Druhou možnosťou obmedzovania vlastností, je obmedzenie kardinality. Využíva sa v prípadoch, keď je potrebné špecifikovať počet jednotlivcov, ktorých sa vlastnosť týka. Pri obmedzovaní kardinality je možné využiť tieto špecifikácie:

- **owl:maxCardinality** – Pomocou tejto vlastnosti sa dá definovať trieda všetkých jedincov, pre ktorých daná vlastnosť nadobúda maximálne počtu hodnôt definovaných v obmedzení.
- **owl:minCardinality** – Popisuje triedu jedincov, ktorých vlastnosť nadobúda kardinalitu aspoň hodnoty definovanej v obmedzení.
- **owl:cardinality** – Vlastnosť popisujúca triedu s jedincami, ktorých daná vlastnosť nadobúda presného počtu hodnôt, ktorý je definovaný obmedzením. Táto možnosť je zobrazená na ukážke 3.8

```
<owl:Class rdf:ID="SingleChildParent">
   <owl:Restriction>
      <owl:onProperty rdf:resource="#hasChild"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
         1
      </owl:maxCardinality>
   </owl:Restriction>
</owl:Class>
```

Výpis 3.8: Príklad definovania triedy pomocou obmedzenia kardinality v jazyku OWL 2

OWL poskytuje možnosť aj priamej definície množiny triedy. Pomocou možnosti enumerácie je umožnené určiť prípustných členov triedy. Takýto prípad zobrazuje priložený príklad 3.9.

```
<owl:Class>
   <owl:oneOf rdf:parseType="Collection">
      <rdf:Description rdf:about="#James"/>
      <rdf:Description rdf:about="#John"/>
      <rdf:Description rdf:about="#Jim"/>
   </owl:oneOf>
</owl:Class>
```

Výpis 3.9: Príklad definície enumerácie triedy v jazyku OWL 2

Inštancie

Konkrétny prvok triedy nazývame inštancia. Jej definícia v jazyku OWL 2 je zobrazená na príklade 3.10. Tento príklad vytvára objekt „Mary“, ktorý je inštanciou triedy „Person“.

```
<Person rdf:ID="Mary"/>
```

Výpis 3.10: Príklad definície inštancie v jazyku OWL 2

Pri jednotlivých jedincoch OWL 2 disponuje tiež definíciou ich rovnosti či nerovnosti k inému jedincovi. K vyjadreniu rovnosti slúži element *owl:sameAs* a k nerovnosti je to element *owl:differentFrom*. Prvá časť príkladu 3.11 poukazuje, že jedinec „Bill“ nie je totožný s jedincom „John“. Druhá časť toho príkladu poukazuje na zhodu jedinca „Jim“ s jedincom „James“.

```
<rdf:Description rdf:ID="John">
   <owl:differentFrom rdf:resource="#Bill"/>
</rdf:Description>

<rdf:Description rdf:ID="James">
   <owl:sameAs rdf:resource="#Jim"/>
</rdf:Description>
```

Výpis 3.11: Príklad definície rovnosti a nerovnosti jedincov v jazyku OWL 2

Vlastnosti

Vďaka vlastnostiam sa triedam a ich členom dajú špecifikovať tvrdenia a spojenia medzi jednotlivými objektami a dátovými typmi. V rámci jazyka OWL 2 sa dá hovoriť o dvoch typoch vlastností: objektové a datotypové.

Vlastnosti, ktoré vytvárajú spojenie medzi dvoma objektami sú objektové vlastnosti. Takáto vlastnosť sa definuje pomocou elementu *owl:ObjectProperty*. Datotypová vlastnosť vyjadruje spojenie medzi určitým objektom a dátovým typom. K jej definícií existuje element *owl:DatatypeProperty*. Vo vnútri týchto oboch elementov sa môžu nachádzať špeciálne konštruktory. Tieto konštruktory popisujú isté podmienky alebo definície druhu vlastností. Jazyk OWL 2 podporuje aj niekotoré podmienky z jazyka RDFS. Takými sú *rdfs:subPropertyOf*, *rdfs:domain* a *rdfs:range*. Všetky sú podrobnejšie popísané v časti 3.1.2. Na príklade definície jednej objektovej a jednej datotypovej vlastnosti je ukázané ich vyjadrenie v jazyku OWL 2. Objektovou vlastnosťou je „hasDog“, ktorá spája triedy „Person“ a „Dog“. Datotypovou vlastnosťou je „hasAge“, ktorá spája triedu „Person“ s dátovým typom „nonNegativeInteger“.

```
<owl:ObjectProperty rdf:ID="hasDog">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Dog"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="hasAge">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#
        nonNegativeInteger"/>
</owl:DatatypeProperty>
```

Výpis 3.12: Príklad definície vlastností v jazyku OWL 2

Ďalším konštruktorom uvedeným v jazyku OWL 2 je element s označením *owl:equivalentProperty*. Pomocou neho sa vyjadruje ekvivalencia dvoch vlastností. Ekvivalentné vlastnosti majú rov-

naké prvky, ale vlastnosti môžu mať rôzny význam. Príkladom v ukážke 3.13 sú vlastnosti „livePlace“ a „liveLocation“.

```
<owl:ObjectProperty rdf:ID="livePlace">
    <owl:equivalentProperty rdf:resource="#liveLocation"/>
</owl:ObjectProperty>
```

Výpis 3.13: Príklad definície ekvivalentných vlastností v jazyku OWL 2

OWL 2 umožňuje tiež definíciu opačnej (inverznej) vlastnosti k istej vlastnosti. Takáto definícia sa vyznačuje elementom *owl:inverseOf*. Jej použitím popisujeme vzťah v oboch smeroch. Nasledujúci príklad 3.14 definuje vlastnosť „hasOwner“ ako vlastnosť inverznú k triede „hasDog“.

```
<owl:ObjectProperty rdf:ID="hasOwner">
    <owl:inverseOf rdf:resource="#hasDog"/>
</owl:ObjectProperty>
```

Výpis 3.14: Príklad definície inverznej vlastnosti v jazyku OWL 2

Obmedzenie kardinality sa vyjadruje pomocou tzv. funkcionálnej vlastnosti. Jej označením je *owl:FunctionalProperty*. Aplikáciou funkcionálnej vlastnosti je zabezpečené, že hodnotou objektu danej vlastnosti bude môcť byť iba jeden prvok. Druhou existujúcou funkcionálnou vlastnosťou je *owl:InverseFunctionalProperty*. Jej použitie obmedzuje vlastnosť v opačnom smere. Vymedzuje teda hodnotu subjektu. Uvedený príklad 3.15 nadväzuje na predchádzajúcu definíciu vlastnosti „hasDog“. Obmedzením je spôsobené, že hodnotou vlastnosti môže byť iba jeden člen z triedy „Dog“.

```
<owl:FunctionalProperty rdf:about="#hasDog"/>
```

Výpis 3.15: Príklad definície funkcionálnej vlastnosti v jazyku OWL 2

Využitie matematických vlastností relácií na množine môže byť ďalším spôsobom definície nejakej vlastnosti. Konštruktory takýchto definícií sú nasledovné:

- **owl:ReflexiveProperty** – Špecifikuje reflexívnu vlastnosť. Takáto vlastnosť vyjadruje, že v nej existuje spojenie všetkých prvkov samích na seba. Pre každý prvok spojený reflexívnou vlastnosťou, existuje inštancia tejto vlastnosti, ktorá spája prvok sám so sebou.
- **owl:IrreflexiveProperty** – Špecifikuje opak reflexívnej vlastnosti. Nereflexívna vlastnosť nemôže obsahovať ani jednu inštanciu, ktorá by spájala rovnaké prvky.
- **owl:SymmetricProperty** – Špecifikuje symetrickú vlastnosť. Vlastnosť je symetrická práve vtedy, keď platí, že pokiaľ vlastnosť spája prvok A s prvkom B, tak táto vlastnosť musí spájať aj prvok B s prvkom A.
- **owl:AsymmetricProperty** – Vlastnosť môže byť tiež asymetrická. To znamená, že ak vlastnosť spája prvok A s prvkom B, potom už nemôže spájať prvok B s prvkom A.
- **owl:TransitiveProperty** – Špecifikuje tranzitívnu vlastnosť. Vlastnosť je tranzitívna práve vtedy, keď platí, že pokiaľ vlastnosť spája prvok A s prvkom B a spája aj prvok B s prvkom C, tak táto vlastnosť musí spájať aj prvok A s prvkom C.

Definíciu symetrickej a asymetrickej vlastnosti je možné vidieť na príklade 3.16.

```
<owl:SymmetricProperty rdf:about="#hasFriend"/>

<owl:AsymmetricProperty rdf:about="#hasDog"/>
```

Výpis 3.16: Príklad definície symetrickej a asymetrickej vlastnosti v jazyku OWL 2

Priradenie akejkoľvek vlastnosti k určitej inštancii vyjadrujeme v jazyku OWL 2 rovnako ako v štruktúre RDF. Na ukážke 3.17 je možné vidieť vytvorenie vlastnosti „hasWife“ v inštancií „John“.

```
<rdf:Description rdf:about="#John">
    <hasWife rdf:resource="#Mary"/>
</rdf:Description>
```

Výpis 3.17: Príklad vytvorenia vlastnosti inštancie v jazyku OWL 2

Špeciálne v jazyku OWL 2 existuje možnosť vyjadriť situáciu, keď dva jedince nie sú spojené vlastnosťou. Tento mechanizmus, ktorý je zobrazený aj na ukážke 3.18, poskytuje možnosť vyjadrenia, že niečo nie je pravda. Jeho definovanie sa vyjadruje pomocou označenia *owl:NegativePropertyAssertion*. Ukážka špecifikuje, že medzi inštanciami „Bill“ a „Mary“ neexistuje spojenie vlastnosťou „hasWife“.

```
<owl:NegativePropertyAssertion>
    <owl:sourceIndividual rdf:resource="Bill"/>
    <owl:assertionProperty rdf:resource="hasWife"/>
    <owl:targetIndividual rdf:resource="Mary"/>
</owl:NegativePropertyAssertion>
```

Výpis 3.18: Príklad definície

Kapitola 4

Existujúce softvérové nástroje a knižnice

Táto kapitola predstavuje dve skupiny existujúcich softvérových nástrojov, ktoré sú podstatné pre ďalšiu činnosť na tejto práci. Prvou skupinou sú nástroje a knižnice pre spracovanie RDF a OWL definícií, ktorým sa venuje časť 4.1. V sekcií 4.2 sú predstavené nástroje a knižnice pre generovanie programového kódu pomocou šablón.

4.1 Spracovanie RDF a OWL definícií

Pre prácu s RDF a OWL definíciami existuje v dnešnej dobe už veľké množstvo nástrojov. Tieto nástroje uľahčujú rôzne aspekty práce s definíciami ontológií a jej prvkov ako napríklad ich vytváranie, načítanie, ukladanie či dotazovanie. V tejto časti je vytvorený prehľad niektorých existujúcich nástrojov a knižníc.

4.1.1 Protégé

Protege je bezplatná, open-source platforma, poskytujúca súbor nástrojov na vytváranie doménových modelov a znalostných aplikácií s ontológiou. Platforma je vyvíjaná inštitúciou *Stanford Center for Biomedical Informations Research*, ktorá patrí pod *Stanford University School of Medicine*. Platforma si vďaka svojim nástrojom našla silnú a neustále sa rozrastajúcu komunitu podporovateľov. Tí ju používajú pre budovanie znalostne založených riešení v rôznorodých oblastiach. Informácií o tomto nástroji boli získané z [16].

Niektoré z nástrojov tejto platformi sú:

- **Protégé Desktop** – je funkčne bohatý editor ontológií s plnou podporou jazyka OWL 2 a priamym pripojením pamäte na riadiace moduly deskriptívnej logiky, ktorými sú napríklad *HermiT* a *Pellet*. Používateľské rozhranie poskytuje pracovný priestor pre vytváranie a úpravu jednej alebo viacerých ontológií. Dostupná je interaktívna navigácia ontologických vzťahov pomocou vizualizačných nástrojov. Pri hľadaní nezrovnalostí pomáha pokročilá podpora vysvetľovania. K dispozícií sú refaktorovacie operácie zahrňujúce spájanie ontológií, presúvanie axióm medzi ontológiami, premenovania entít a ďalšie.

- **WebProtégé**[1] – je webové prostredie pre vývoj ontológií. Slúži k zjednodušeniu vytvárania, načítania, modifikácie a zdieľania ontológií. Zameraný je hlavne na skupinovú spoluprácu, ktorej poskytuje okrem skupinového prezerania a modifikácie aj funkcionalitu ako povolenia, vláknové poznámky a diskusie či emailové notifikácie. Nástroj plne podporuje jazyk OWL 2. Formáty RDF/XML, Turtle, OWL/XML, OBO a ďalšie sú dostupné na nahrávanie a sťahovanie ontológie. Táto webová verzia sa stala extrémne populárnou a prekonala desktopovú verziu vo svojej úrovni využitia.

- **Protege-OWL API** – súčasťou produktov Protégé je tiež aplikačné programové rozhranie Protege-OWL. Je to knižnica v jazyku Java určená pre prácu s OWL a RDF(S). Jej dokumentácia [26] hovorí, že Protege-OWL API poskytuje triedy a metódy pre načítanie a ukladanie OWL dokumentov, pre dotazovanie a manipulovanie s OWL dátovými modelmi a pre vykonávanie uvažovania založeného na riadiacich moduloch deskriptívnej logiky. Toto API je naviac optimalizované pre implementáciu grafických užívateľských rozhraní. Okrem vývoja samotných aplikácií založených na ontológiách, je možné využitie aj k tvorbe komponentov, ktoré sa vykonávajú v užívateľskom rozhraní Protégé-OWL editorov.

4.1.2 Eclipse RDF4J

RDF4J je modulárny framework jazyka Java. Slúžiaci k práci s RDF dátami. Distribuje sa s voľne dostupným zdrojovým kódom. Patrí pod jeden z projektov *Eclipse Foundation*, ktorá sa stará o jeho vývoj. Tento framework poskytuje funkcionalitu pre analyzovanie, ukladanie, dotazovanie a uvažovanie s RDF údajmi a jednoduché aplikačné programové rozhranie, schopné pripojenia k všetkým hlavným RDF databázam. Niektoré z hlavných funkcí RDF4J sú [9]:

- plne podporuje dopytovací a aktualizačný jazyk SPARQL 1.1[2], ktorý poskytuje expresné dopytovanie a transparentný prístup k vzdialeným RDF úložiskám,
- rýchle a efektívne parsovanie všetkých bežne dostupných formátov pomocou nástroja *Rio*,
- poskytuje ľahko použiteľné a moderné API pre jazyk Java, ktoré sa využíva na prácu s RDF v programovom kóde,
- poskytuje širokú škálu nástrojov na využitie sily RDF a súvisiacich štandardov,
- dobre podporuje uvažovanie jazykom RDFS a validáciu pomocou jazyka SHACL[3],
- umožňuje pripojenie ku koncovým bodom SPARQL,
- umožňuje tvorbu aplikácií, ktoré využívajú silu prepojených dát a sémantického webu,
- poskytuje dve rýchle RDF úložiská (jedno pamäťové a druhé natívne) a okrem toho aj pohodlný prístup k množstvom úložných riešení tretích strán.

[1] https://webprotege.stanford.edu/
[2] https://www.w3.org/TR/sparql11-query/
[3] https://www.w3.org/TR/shacl/

4.1.3 Apache Jena

Ďalším frameworkom, slúžiacim k tvorbe aplikácií sémantického webu a prepojených dát, je Apache Jena (skr. Jena). Je to bezplatný nástroj s voľne dostupným zdrojovým kódom.

Jena je podobná frameworku RDF4J. Poskytuje funkcionalitu ako syntaktický analyzátor (pre formáty RDF/XML, Turtle a N-triples), kompletnú podporu dotazovacieho jazyka SPARQL, odvodzovací nástroj pre ontológie, programovacie rozhranie Java API, server RDF využívajúci webové protokoly či dva typy úložísk. Rozdielom medzi RDF4J a Jena je to, že Jena poskytuje podporu pre jazyk OWL. Táto podpora je však len pre verziu OWL 1 a nie pre novšiu verziu. [24]

4.1.4 OWL API

OWL API je vysokoúrovňové aplikačné programové rozhranie pre prácu s OWL ontológiami. Uverejnené bolo v roku 2003 a odvtedy si prešlo množstvom dizajnových úprav, pričom tie sledovali najmä vývoj samotného jazyka OWL. OWL API je implementovaná v jazyku Java dostupná s otvoreným zdrojovým kódom. Od svojho vzniku sa jeho použitie rozšírilo do rôznych nástrojov a aplikácií.

OWL API slúži pre vytváranie, parsovanie, manipuláciu a serializáciu OWL ontológií definovaných v rôznych syntaxoch (Functional Syntax, RDF/XML, OWL/XML a Manchester OWL Syntax). Je úzko zosúladené so štrukturálnou špecifikáciou OWL 2. S ontológiami sa v nej pracuje vo forme axióm a nie priamo vo forme jednotlivých prvkov RDF dátového modelu. Jej kľúčové funkcionality zahrňajú aj axiómovo-centrickú abstrakciu, podporu zmien, univerzálne uvažovacie rozhranie, či validátory pre rôzne profily (OWL 2 QL, OWL 2 EL a OWL 2 RL). Flexibilný dizajn umožňuje použiť alternatívne implementácie hlavných komponentov od tretích strán. Celá časť o OWL API vychádza zo zdroja [13].

4.2 Generovanie programového kódu pomocou šablón

Druhou skupinou predstavovaných nástrojov, sú existujúce nástroje pre generovanie programového kódu pomocou šablón. Šablóny do značnej miery zjednodušujú proces generovania kódu. Tento spôsob generovania bude využitý pri praktickej časti tejto práce. Popísané sú dva šablónové nástroje na generovanie. Jeden z nich je v neskoršej fáze vybraný a použitý pre implementáciu výsledku práce.

4.2.1 String Template

String Template je šablónový nástroj jazyka Java, ktorý poskytuje mechanizmus pre generovanie textu z dátových štruktúr. Vyvíjal sa mnohoročným úsilím a používaním na stránke *jGuru.com* či v generátore jazykových nástrojov *ANTLR v3.* Je distribuovaný ako malá knižnica s jedinými externými závislostiami na ANTLR (používa sa na analýzu jazyka šablón) a štandardných knižniciach. Je určený pre generovanie zdrojových kódov, webových stránok, e-mailov alebo akéhokoľvek iného štruktúrovaného textového výstupu. Je obzvlášť dobrý pri generátoroch viac zacielených kódov, vzhľadoch viacerých stránok a internacionalizácii/lokalizácii. [18]

Charakteristickým znakom StringTemplate je, že na rozdiel od iných porovnateľných šablón striktne presadzuje oddelenie Model-View. Vyznačuje sa aj jednoduchým používaním. Tento nástroj neprotrebuje žiadny špeciálny vzťah s webovým serverom a nepredpokladá ani vedomosti o štruktúre textu v šablóne. [17]

Na ukážke 4.1 sa na jednoduchom príklade zobrazuje princíp generovania pomocou nástroja StringTemplate.

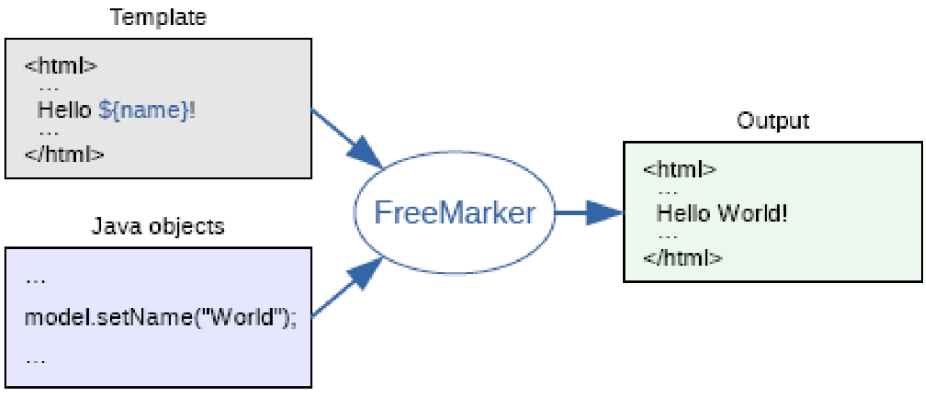
```
1
2  import org.stringtemplate.v4.*;
3  class Simple {
4      public static void main(String[] args) {
5          ST hello = new ST("Hello, <name>");
6          hello.add("name", "World");
7          System.out.println(hello.render());
8      }
9  }
```

Výpis 4.1: Príklad generovanie súboru s textom „Hello, World“ nástrojom StringTemplate

4.2.2 Apache FreeMarker

Druhým popisovaným šablónovým nástrojom je Apache Freemarker. Je to knižnica jazyka Java na generovanie textových výstupov ako HTML webové stránky, e-maily, konfiguračné súbory, zdrojové kódy a podobne. Základ nástroja, zobrazený na obrázku 4.1, je tvorený šablónami a špecifikovanými dátami. Na prípravu dát, ktoré sa majú prezentovať, sa používa všeobecný programovací jazyk, v tomto prípade Java. Pripravené dáta potom Apache FreeMarker zobrazí pomocou šablón. Šablóny určujú to, ako sa dáta zobrazujú. Tento prístup sa často označuje ako vzor MVC (Model View Controller) a je hlavne populárny pre generovanie dynamických webových stránok. Jazykom šablón je jednoduchý špecializovaný jazyk *FreeMarker Template Language*, skrátene FTL. Tento jazyk nie je plnohodnotným programovacím jazykom. [2]

FreeMarker je mocným šablónovým nástrojom, ktorý poskytuje podmienené bloky, iterácie, priradenia, reťazcové a aritmetické operácie, makrá, funkcie a mnoho ďalšieho. Podporuje tiež mnoho konfiguračných možností, je bez závislostí a je schopný generovať akýkoľvek výstupný formát. Ďalšími vlastnosťami sú tiež internacionalizácia/lokalizácia a možnosti spracovania XML. [2]



Obr. 4.1: Zobrazenie základného systému generovania s nástrojom FreeMarker

Kapitola 5

Návrh riešenia

Táto kapitola je vstupnou bránou do praktickej časti tejto práce. Predchádzajúce kapitoly boli venované predstaveniu základných teoretických znalostí z technológií sémantického webu. Tieto znalosti a technológie sú ďalej využité pri vývoji nového nástroja z danej oblasti. Nasledujúca kapitola popisuje prvú fázu vývoja, ktorou je návrh.

Navrhovaný nástroj, nazývaný *OntoCodeMaker*, je určený ku vygenerovanie zdrojového kódu z definície ontológií. Návrh systému a analýza požiadaviek na systém sú dôležitou fázou vývoja od ktorých je priamo závislá kvalita výsledku.

Podkapitola 5.1 sa venuje špecifikovaniu požiadaviek, ktoré by mal výsledný nástroj spĺňať. Ujasňuje sa v nej, k čomu by mal nástroj slúžiť a čo od neho očakáva užívateľ. V nasledujúcej časti 5.2 sú zanalyzované niektoré z už existujúcich generátorov, ktoré poslúžia ako inšpirácia pri návrhu. Ďalej v 5.3 je popísaný návrh generátora z pohľadu celkovej štruktúry. Cieľom posledných dvoch sekcií tejto kapitoly je priblížiť návrh mapovania ontológie do zdrojového kódu v 5.4 a zobraziť navrhovanú celkovú štruktúru vygenerovaného zdrojového kódu v 5.5.

5.1 Analýza požiadaviek

Požiadavky na hlavnú funkcionalitu vytváraného nástroja vyplývajú priamo zo zadania práce. Počas konzultácie s vedúcim boli tieto požiadavky upresnené a získané boli ešte aj ďalšie požiadavky.

Požadovaným výsledkom tejto práce je systém, ktorý umožňuje generovať zdrojový kód z definície ontológie. Vstupom pre nástroj by mal byť súbor obsahujúci ontológiu. Keďže ontológia môže byť definovaná v rôznom syntaxe, dôležitou požiadavkou je podpora najpoužívanejších syntaxí. Užitočnou možnosť by bola aj podpora prijatia viac než len jednej ontológie na vstupe, keďže sú ontológie často na seba závislé.

Požiadavkou priamo zo zadania je, že výstup musí byť generovaný zo šablón. Vygenerovaným výstupom nástroja by mala byť kolekcia súborov určitého programovacieho jazyka. Táto kolekcia súborov musí logicky reprezentovať definíciu ontológie zo vstupného dokumentu. Hlavným uvažovaným programovacím jazykom výstupného kódu je jazyk Java. Nástroj by však malo byť možné pomerne jednoducho rozšíriť o generovanie výstupu v inom objektovo orientovanom programovacom jazyku. Výsledný kód by mal spĺňať všetky štandardy daného programovacieho jazyka. Ďalšou požiadavkou na vygenerovaný kód je zahrnutie serializačnej funkcionality, ktorá bude slúžiť k čítaniu a zapisovaniu inštancií tried do RDF modela ontológie.

Nástroj bude primárne využívaný ľuďmi pohybujúcimi sa v oblasti informatiky, konkrétne hlavne programátormi. Používateľom tento nástroj uľahčí vývoj aplikácií založených na ontológiách, ušetrí čas a ochráni pred chybami. Používanie nástroja by malo byť čo najjednoduchšie a nemalo by byť potrebné si inštalovať žiadne závislosti. Vzhľadom na to, že cieľovým užívateľom je programátor a dôraz sa kladie na funkčnosť a použiteľnosť, aplikácia bude implementovaná ako konzolová aplikácia. Užívateľ teda bude schopný aplikáciu používať priamo cez prostredie príkazového riadku. Možným rozšírením do budúcna, by pre nástroj mohlo byť grafické užívateľské rozhranie. Výsledný generátor je uvažovaný ako lokálny nástroj ku ktorému súčastne pristupuje iba jeden užívateľ. Je určený k jednorázovému vykonávaniu úloh, bez akéhokoľvek ukladania stavu.

5.2 Analýza existujúcich riešení

V súčastnosti je môžné nájisť niekoľko nástrojov, ktoré riešia generovanie zdrojového kódu z definície ontológie. Tieto nástroje disponujú určitými rozlišnosťami. Ich výsledkom sú rozdielne štruktúry generovaného zdrojového kódu, ponúkajú rozdielne mapovanie ontologických prvkov do programovacích jazykov či využívajú rozdielne technológie pre serializáciu. Zvyčajne sa tieto nástroje venujú generovaniu len do jedného jazyka. V nasledujúcej časti sú predstavené dva nástroje pre generovanie výstupného zdrojového kódu v jazyku Java.

5.2.1 RDF4J Class Builder

RDF4J Class Builder[1] je konzolový nástroj pre generovanie zdrojového kódu v jazyku Java z definícií OWL ontológie. Nástroj funguje tak, že na vstup je mu vložený súbor s definíciou ontológie. Podporovaná je definícia v rôznych formátoch (napr. RDF/XML, Turtle a iné). Výstupom sú dva separátne komponenty. Jedným je trieda, ktorá definuje URI konštanty pre všetky objekty a predikáty definované v ontológií. Druhým vygenerovaným komponentom sú jednotlivé triedy jazyka Java reprezentujúce ontológiu. Výsledné balíčky a umiestnenie zložky sa špecifikuje pomocou rôznych nepovinných argumentov na vstupe.

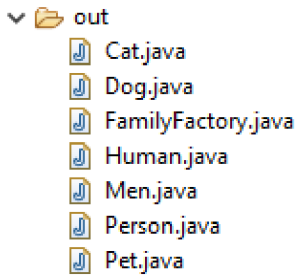
Jednotlivé definície prvkov OWL ontológií sú mapované do zdrojové kódu v jazyku Java nasledovným spôsobom. Každá ontologická trieda reprezentuje triedu jazyka Java. Atribútmi tejto triedy sú ontologické vlastnosti, ktoré sú spojené s touto triedou. Kardinalita vlastností sa udáva použitím definícií *owl:Property*, *owl:InverseFunctionalProperty* a *owl:FunctionalProperty*. Pre každý atribút sú v rámci triedy vygenerované metódy pre získavanie a priraďovanie hodnoty (metódy *get* a *set*). Vygenerovaná trieda v jazyku Java obsahuje tiež metódy *addToModel()* and *loadFromModel()*. Tieto metódy umožňujú načítanie a ukladanie tried z a do modelu RDF. Model RDF je reprezentovaný pomocou frameworku RDF4J. Pre celú ontológiu sa vygeneruje aj továrenské rozhranie, vďaka ktorému sa dajú implementovať vlastné továrne na vytváranie inštancií objektov.

Na obrázku 5.1 je možné vidieť vygenerovanú zložku nástroja RDF4J Class Builder, ktorá vznikla pri analyzovaní tohto nástroja. Výstup reprezentuje ontológiu *Simple Family*, ktorá je dostupná v prílohe C.

5.2.2 Protege-OWL Code Generator

Desktopová verzia editora Protégé, ktorý je predstavený v kapitole 4.1.1 v sebe ponúka aj nástroj na vygenerovanie zdrojového kódu z ontológie. Tak ako aj predošlý nástroj aj

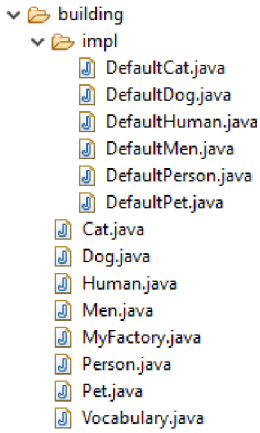
[1] https://github.com/radkovo/rdf4j-class-builder



Obr. 5.1: Zobrazenie výsledku generovania nástroja RDF4J Class Builder

tento generuje výsledný zdrojový kód len v jazyku Java. Nástroj je možné nájsť v aplikácií Protégé po rozkliknutí nástrojov v hornej navigačnej lište, kde je možnosť „Generate Java code from active ontology“. Generovať je možné len aktuálne otvorenú ontológiu.

V porovnaní s RDF4J Class Builder sú pri vygenerovanom kóde podstatné rozdiely. Triedy generované v nástroji RDF4J Class Builder obsahujú metódy pre prácu s modelom, ale nie je povinné pridávať jej inštancie do modelu. Z toho vyplýva, že inštancia triedy môže existovať aj bez toho, aby bola v modeli. Toto pri triedach vytváraných generátorom Protege-OWL možné nie je. Kód vygenerovaný týmto nástrojom poskytuje prístup k už existujúcim inštanciám v dátovom modeli ontológie a možnosť v ňom vytvárať nové inštancie.



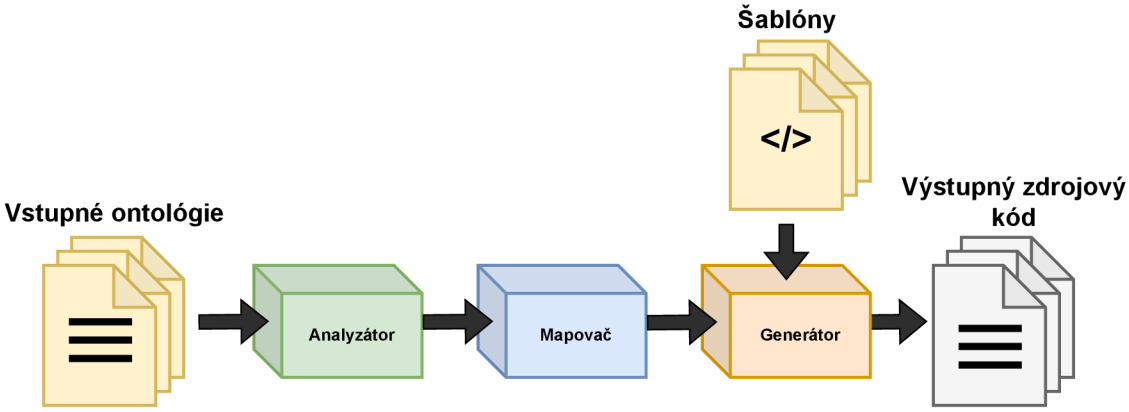
Obr. 5.2: Zobrazenie výsledku generovania nástroja Protege

Štruktúra vygenerovaných tried pomocou tohto nástroja je zobrazená na obrázku 5.2. Balíček s touto štruktúrou je vygenerovaný z ontológie *Simple Family*. Hlavným vygenerovaným súborom je továrenská trieda, ktorá implementuje rozhranie *CodeGenerationFactory*. Táto trieda slúži ako vstupný bod k vygenerovanému kódu a uchováva model ontológie. Model ontológie je v tomto prípade reprezentovaný pomocou OWL API. Ďalšími vygenerovanými súbormi sú zdrojové kódy, ktoré reprezentujú ontológiu. Ontologická trieda je týmto nástrojom mapovaná do kódu v jazyku Java ako rozhranie a trieda, ktorá ho implementuje. Táto reprezentácia ontologickej triedy ale neobsahuje atribúty pre vlastnosti. Pre vlastnosť sa vygeneruje len atribút typu boolean, ktorý hovorí o existencií hodnoty vlastnosti a metódy na priradenie, získanie a vymazanie tejto hodnoty. Výstupom generovanie je aj trieda

slovnej zásoby. Táto trieda špecifikuje konštanty, ktoré poskytujú prístup k reprezentácii jednotlivých tried v Manchester OWL API.

5.3 Architektúra nástroja

Podstatou vyvíjaného nástroja je proces generovania zdrojového kódu z definície ontológie. Tento proces sa dá rozdeliť na tri fázy. Navrhovaný nástroj sa rozdeľuje na tri základné časti, ktorými sú analyzátor (parser), mapovač a generátor. Postupnosť jednotlivých častí je možné vidieť na obrázku 5.3. Každá z týchto častí má na starosti určitý procesný krok, čiže jednu fázu generovania. Jednotlivé fázy spočívajú z vykonávania niekoľkých čiastkových úloh.



Obr. 5.3: Navrhovaná architektúra nástroja OntoCodeMaker

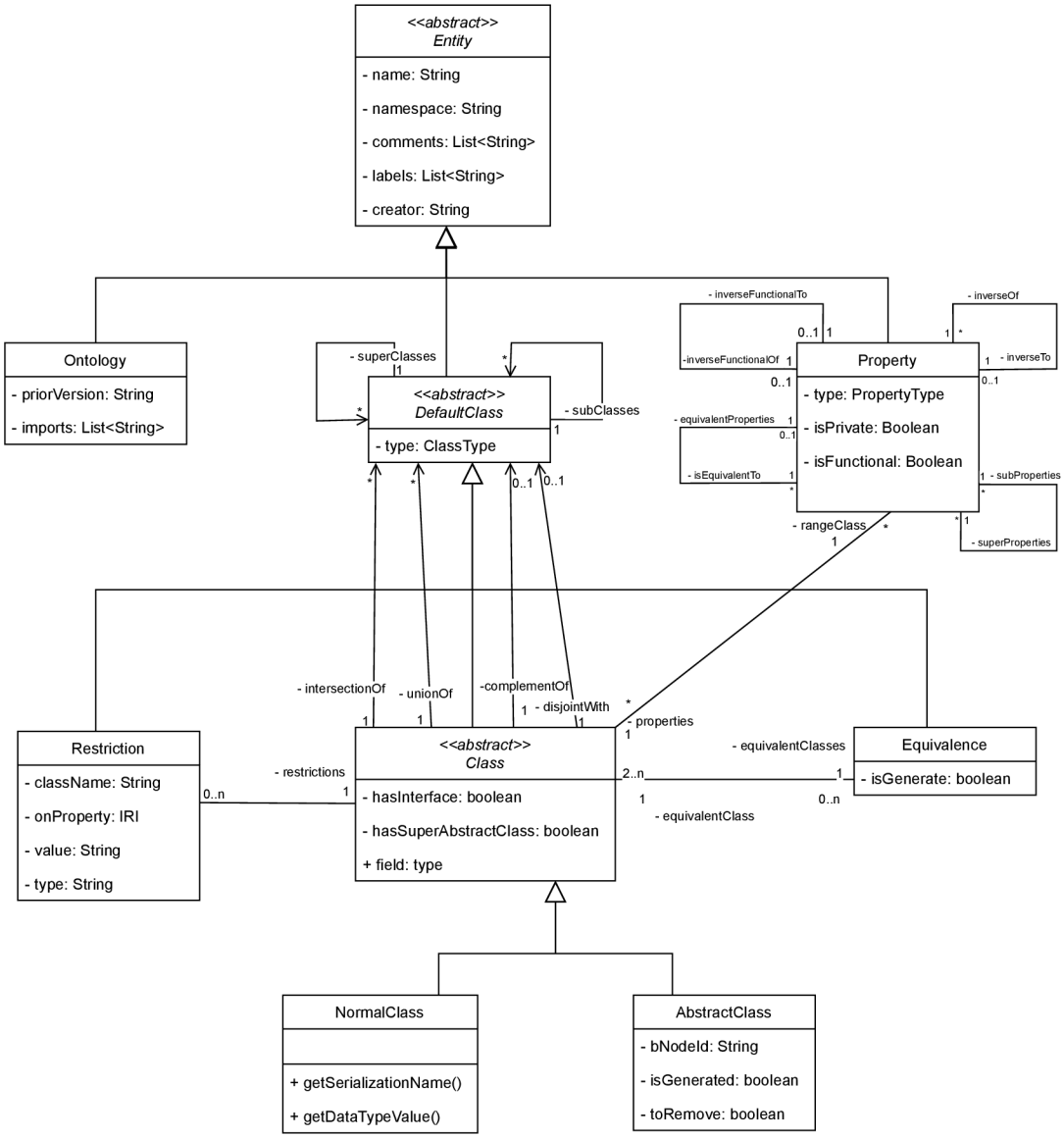
Analyzátor

Hlavnými zmyslami analyzátora je načítanie a overenie správnosti syntaxe vstupných ontológií. Vstupom analyzátora je dokument alebo kolekcia dokumentov, ktoré obsahujú ontológiu. Analyzátor spracováva tieto dokumenty a parsuje v nich zapísanú ontológiu. V prípade úspešného parsovania sa následne vytvára výstup analyzátora. Výstupom je dátový model RDF. Tento model bol predstavený už v kapitole 2.2.1.

Mapovač

Mapovač je ústrednou časťou navrhovaného nástroja a jeho návrh je dôležitý pre celý generátor. Zabezpečuje hlavnú časť premeny ontologickej štruktúry na štruktúru objektovo orientovaného jazyka. Mapovač je miestom, kde sa rozhoduje, ktoré entity a vlastnosti z ontológie sa môžu premietnuť v zdrojovom kóde. Jeho vstupom je dátový model RDF. Mapovač zo vstupného modelu vytvorí reprezentáciu ontologických prvkov. Táto reprezentácia je definovaná pomocou vnútorného modelu ontologických prvkov. Výhodou vnútorného modelu je, že umožňuje zaznamenávať ontologické informácie nezávisle od cieľového programovacieho jazyka. To prináša flexibilitu v podpore cieľových jazykov. Vnútorný model je možné vidieť na obrázku 5.4.

Všetky elementy vnútorného modelu vychádzajú z abstraktnej triedy *Entity*, ktorá sa identifikuje atribútmi mena a menného priestoru. Element *Ontology* uchováva všetky infor-



Obr. 5.4: Návrh vnútorného modelu ontologických prvkov

mácie o ontológii ako celku. Ďalším elementom modelu je abstraktná trieda *DefaultClass*. Z tejto triedy vychádzajú reprezentácie tried a špecifické stavy tried, ktorými sú ekvivalencia a obmedzenie. Prvky tejto triedy môžu medzi sebou definovať hierarchie pomocou supertried a podtried.

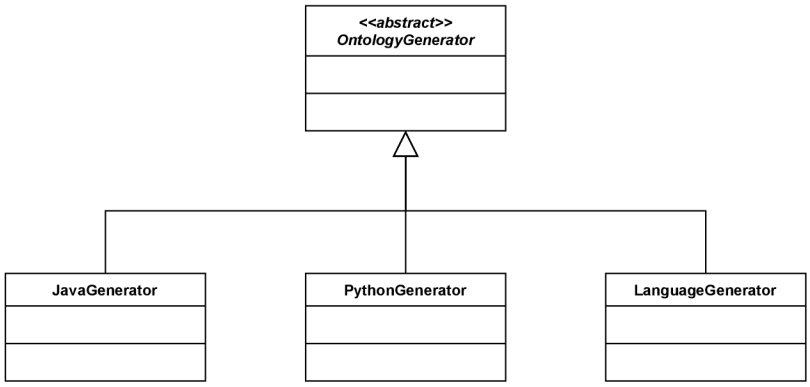
Hlavným elementom modelu je abstraktná trieda *Class*, ktorá vyjadruje ontologické triedy dvoch typov. Prvým typom je trieda *NormalClass*, ktorá reprezentuje klasickú triedu definovanú v ontológií. Trieda *AbstractClass* je druhým typom a reprezentuje prázdny uzol z ontológie. Každý element *Class* môže mať komplement a disjunkciu tvorenú inštanciou *DefaultClass*. Tento element môže byť tiež tvorený zjednotením alebo prienikom viacerých inštancií z elementu *DefaultClass*. Každý element *Class* môže mať tiež niekoľko vlastností, ktoré sú reprezentované elementom *Property*. Vlastnosti môžu medzi sebou vytvárať viaceré spojenia ekvivalencie, hierarchické spojenia a inverzné spojenia. *Equivalence* je konkrétny

typ elementu *DefaultClass* symbolizujúci ekvivalenciu tried. Uchováva v sebe inštancie všetkých tried, ktoré sú navzájom ekvivalentné. Posledným nepredstaveným elementom vnútorného modelu je *Restriction*, ktorý vychádza z *DefaultClass* a vyjadruje určitú reštrikciu na triede.

Generátor

Generovanie zdrojového kódu je záverečná fáza o ktorú sa stará generátor. Požiadavkou priamo zo zadania bolo, aby proces generovania prebiehal pomocou šablón. Úlohou generátora je pripraviť dáta pre šablóny a vyplniť šablóny dátami, čo vygeneruje zdrojový kód. Na začiatku generátor dostane reprezentáciu ontologických tried s vlastnosťami vo forme vnútorného modelu, ktorý bol vytvorený mapovačom. Výstupom generátora je vlastne už výsledný produkt celého nástroja, a to kolekcia súborov so zdrojovým kódom.

Jednou z požiadaviek na výsledný nástroj je nástroj navrhnúť tak, aby bolo potenciálne možné nástroj rozšíriť o generovanie do ďalších programovacích jazykov, okrem primárne zvoleného jazyka Java. K splneniu tejto požiadavky je generátor navrhnutý tak, že sa skladá z generátorov špecializovaných pre konkrétny jazyk. Na obrázku 5.5 je možné vidieť návrh statickej štruktúry tried v generátore. Každý špecifický generátor vychádza zo všeobecného generátora tvoreného pomocou abstraktnej triedy. Pre pridanie nového jazyka bude potrebné len vytvorenie špecifického generátora s naimplementovanými abstraktnými triedami. Pre demonštráciu pridávania ďalších jazykov je okrem primárne žiadaného jazyka Java zamýšlané aj vytvorenie generátora pre jazyk Python.



Obr. 5.5: Diagram tried zobrazujúci statickú štruktúru rozloženia tried v generátore.

5.4 Návrh mapovania ontologických entít do jazyka Java

Ontológie a objektovo orientované dátové modely sú si vo svojom charaktere významovosti veľmi podobné. Mnoho modelovacích čŕt jazyka OWL existuje aj v objektovo orientovaných jazykoch napr. hierarchie. Konkrétne programovacie jazyky ale majú dizajnové špecifiká, kvôli ktorým nie je možné niektoré konštruktory z jazyka OWL priamo namapovať [10]. Iné ontologické prvky sú komplikovanejšie na premapovanie a ich obsiahnutie by neprinášalo veľký úžitok. Z týchto dôvodov nie sú do objektovo orientovaného jazyka premapované všetky konštrukcie ontológií z jazyka OWL. Na základe toho vyplýva, že výsledný navrhovaný nástroj by mal byť schopný generovať zdrojový kód len z určitých prvkov jazyka OWL.

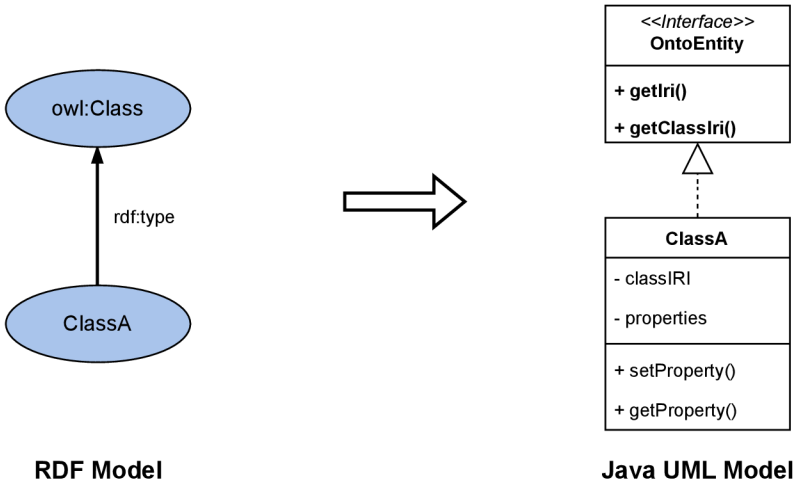
Je teda schopný premapovávať len určitý profil jazyk OWL a nie celú množinu. Princip profilov je bližšie vysvetlený v 3.2.2.

Riešenie pre mapovanie prvkov ontológie do programovacieho jazyka tvorí kľúčový návrh pre zostrojenie nástroja. Návrh zobrazuje akým spôsobom sa jednotlivé prvky RDF sveta, ktorý je orientovaný na trojice, zobrazujú do objektovo orientovaného sveta jazyka Java.

Táto časť postupne predstavuje spôsoby mapovania základných podporovaných prvkov jazyka OWL a ich rôznych definícií. Tieto prvky a definície boli už bližšie predstavené v časti 3.2.3. Zjednodušený popis mapovania je dostupný vo forme tabuľky v prílohe B tejto práce. Príloha tiež obsahuje návrh mapovania do jazyku Python.

Klasické triedy

Klasická OWL trieda sa v Jave zobrazuje ako trieda z jazyka Java, ktorá obsahuje premennú s IRI hodnotou danej OWL triedy a konštruktorom pre túto triedu. Keďže triedy jazyka Java sú schopné len jednoduchého dedenia, pri viacnásobnom dedení je OWL trieda mapovaná do rozhrania jazyka Java a triedy implementujúcej dané rozhranie. Rozhrania v Jave majú schopnosť dedenia viacerých rozhraní, ktorá je v takom prípade využívaná.



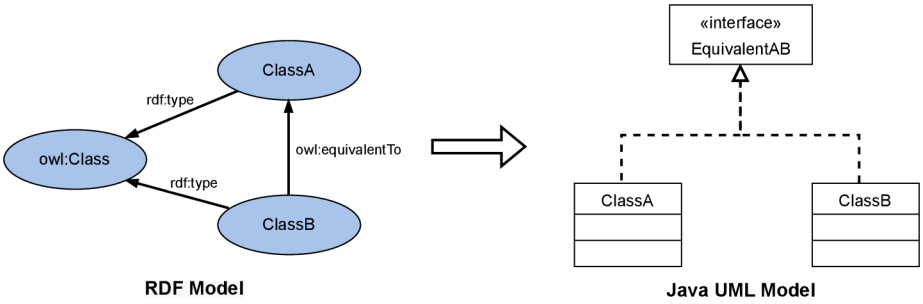
Obr. 5.6: Ilustrácia mapovania jednoduchej triedy.

Ekvivalentné triedy

Ekvivalentné triedy sa do Javy premietajú pomocou rozhrania. Na obrázku 5.7 je možné vidieť modeli reprezentujúce jednoduchý príklad, keď trieda B je ekvivalentná k triede A. V ľavej časti obrázka je zobrazený model RDF takéhoto vzťahu a v pravej časti zodpovedajúci UML diagram tried. V Jave sa zobrazený príkladový vzťah reprezentuje rozhraním EquivalentAB a triedami A a B, ktoré implementujú dané rozhranie.

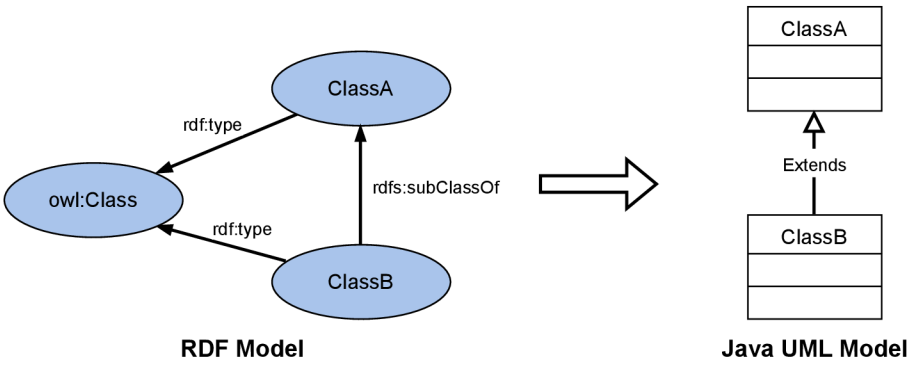
Podtriedy triedy

Navrhované mapovanie podtried určitej definovanej triedy je založené na dedičnosti. Ak máme v ontológií definovanú triedu B, ktorá je podtriedou triedy A, v Jave to bude zobrazené tak, že trieda B rozširuje (extends) triedu A. Tento prípad je zobrazený na modeloch obrázka 5.8. Vyššie v tejto časti už bolo spomínané, že v prípade viacnásobnej dedičnosti sa



Obr. 5.7: Ilustrácia mapovania ekvivalentnej triedy.

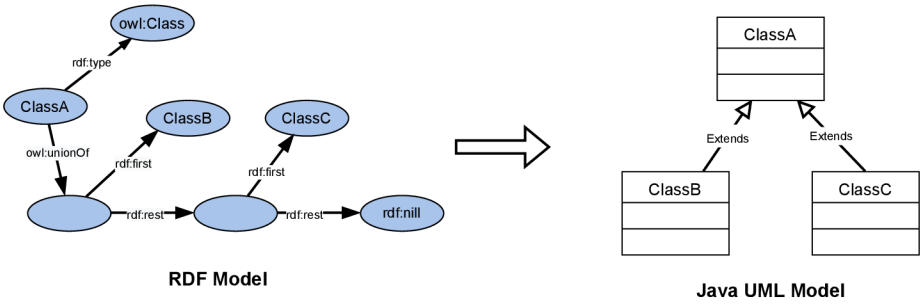
ontologická trieda mapuje ako rozhranie s triedou, ktorá ho implementuje. V takom prípade podtrieda implementuje dané rozhranie nadradenej triedy.



Obr. 5.8: Ilustrácia mapovania podtriedy.

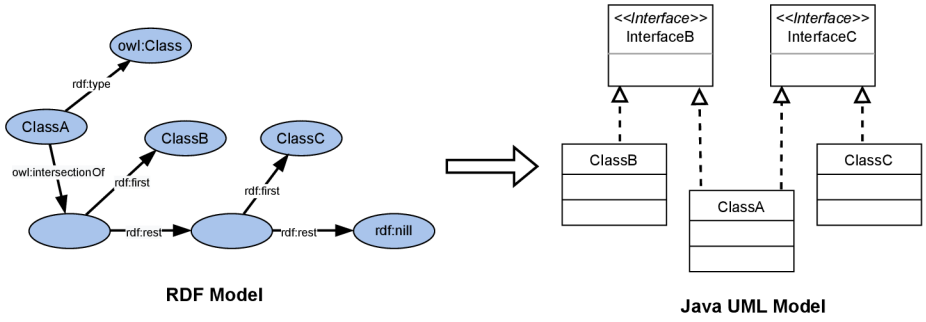
Triedy definované množinovými operáciami

Ďalšími prvkami ontológií, ktoré sú podporované v návrhu výsledného nástroja sú množinové operácie. Predpokladom je ontológia špecifikujúca triedu A tvorenú ako zjednotenie tried B a C. V Jave bude tento predpoklad zobrazený ako triedy A, B a C, pričom platí, že triedy B a C rozširujú triedu A. Mapovanie prípadu zjednotenia je zobrazené na obrázku 5.9.



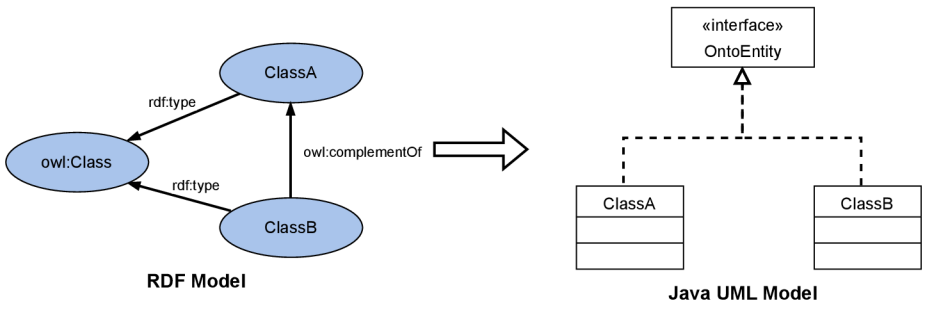
Obr. 5.9: Ilustrácia mapovania triedy definovanej operáciou zjednotenia.

Nasledujúci obrázok 5.10 zobrazuje spôsob definovania množinovej operácie prienik do jazyka Java. Máme dané triedy A a B. Prienik týchto tried bude v Jave tvorený ako trieda, ktorá dedí z tried A a B, teda implementuje ich rozhrania.



Obr. 5.10: Ilustrácia mapovania triedy definovanej operáciou prienik.

Mapovanie komplementu nebolo také jednoznačné ako zvyšné množinové operácie. Java neposkytuje možnosť deklarovania objektu ako inštancie nesúrodých tried. Pri komplemente sa teda využíva skutočnosti, že každá trieda vychádza minimálne z rozhrania OntoEntity alebo z iných tried a rozhraní. Komplement sa teda mapuje ako ďalší potomok nadtriedy od triedy, z ktorej je komplement vytváraný. Príklad 5.11 zobrazuje triedu, ktorá v Jave vychádza zo štandardného rozhrania OntoEntity. Jej komplement je tvorený triedou, ktorá rovnako vychádza z nadradenej triedy, čiže zo štandardného rozhrania.



Obr. 5.11: Ilustrácia mapovania triedy definovanej komplementom.

Vlastnosti

Vlastnosti sú v Jave tvorené pomocou atribútov tried. Definície vlastností určujú všetky špecifiká atribútov, ako napríklad cieľová trieda, dátový typ a kardinalita. Mapovania jednotlivých špecifikácií sú nasledovné:

- **rdfs:domain** – Hodnota tejto špecifikácie vyjadruje triedu, v ktorej je atribút umiestnený. Všetky vlastnosti bez hodnoty domain sa do zdrojového kódu negenerujú. Vlastnosť sa nemusí nachádzať len v definíciach atribútov, ktoré sú ekvivalenciou, inverziou alebo podatribútom iného atribútu.
- **rdfs:range** – Range určuje typ nadobúdaných hodnôt daného atribútu. Jej hodnotou môže byť dátový typ alebo odkaz na určitý objekt. V prípade chýbajúcej definície tejto vlastnosti v špecifikácií atribútu sa tento atribút do zdrojového kódu nepremietne.

Táto vlastnosť nie je potrebná v definícií atribútu len vtedy, keď atribút vychádza z iného atribútu.

- **owl:equivalentProperty** – Táto špecifikácia určuje, že vlastnosť je ekvivalentná s inou vlastnosťou. V Jave táto špecifikácia spôsobuje, že atribút má totožné umiestnenie, dátový typ a kardinalitu ako ekvivalentný atribút.
- **rdfs:subPropertyOf** – Vzťah podvlastnosti je v Jave definovaný ako atribút s identickou špecifikáciou akú má nadradená vlastnosť. Po priradení hodnoty podradenej vlastnosti túto hodnotu automaticky získava aj nadradený atribút.
- **owl:inverseOf** – Túto špecifikáciu mapujeme do Javy, takým spôsobom, že vytvoríme atribút inverzný k atribútu definovanému v tejto špecifikácií.
- **owl:functional** – Pomocou tejto špecifikácie sa obmedzuje kardinalita atribútu na jedna. To znamená, že atribút bude môcť nadobúdať len jednej hodnoty. Východzia kardinalita atribútu nie je obmedzená ale prezentuje sa ako pole o n prvkoch.
- **owl:inverseFunctional** – V Jave je owl:inverseFunctional namapovaný tak, že sa vytvorí inverzný atribút, podobne ako pri inverseOf, a obmedzí sa jeho kardinalita, podobne ako pri functional.

Anotácie

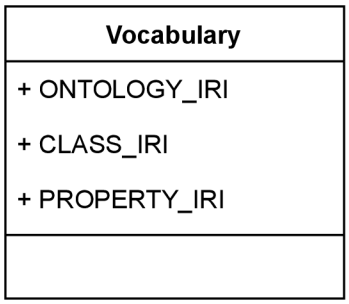
Do vygenerovaného zdrojového kódu by sa mali tiež prenášať všetky základné anotácie ako napríklad label, komentár, importy a iné. Všetky sa budú v kóde jazyka Java vyskytovať vo forme komentárov. Anotácia vlastnosti bude vyjadrená ako komentár pri deklarovaní atribútu predstavujúceho danú vlastnosť. Anotácie triedy budú tvoriť komentár pri deklarovaní triedy.

5.5 Návrh štruktúry generovaného kódu

Súčasťou návrhu je aj popis, ako bude vyzerať štruktúra vygenerovaného kódu a čo bude obsahom jednotlivých tried. Hlavným dielom vygenerovaných súborov sú triedy reprezentujúce samotnú ontológiu. Pre jazyk Java bola táto reprezentácia predstavená v predchádzajúcej sekcií. Všetky triedy tejto reprezentácie sú v celkovej štruktúre uložené v priečinku *entities/*. Okrem týchto tried sú ale súčasťou návrhu generovaného kódu aj ďalšie triedy.

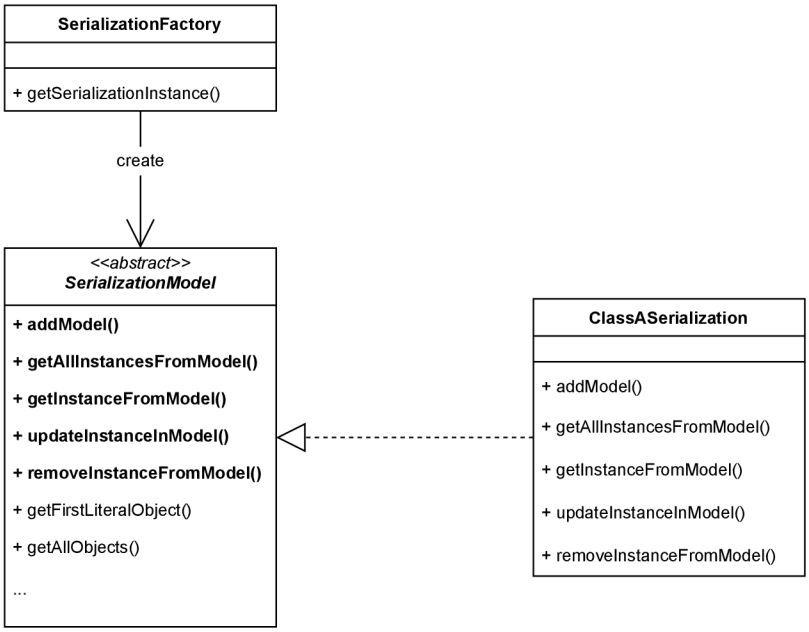
Trieda *Vocabulary*, ktorej diagram je zobrazený na obrázku 5.12, symbolizuje akýsi slovník. Úlohou tohto slovníka je uchovávať identifikátory všetkých tried a vlastností zo vstupných ontológií a aj samotných ontológií. Identifikátory sa v triede deklarujú ako konštanty a následne sa využívajú vo zvyšných triedach vygenerovaného zdrojového kódu.

Jedna z požiadaviek na vygenerovaný zdrojový kód je zahrnutie serializátorov. V návrhu sú serializátory definované ako samostatné serializačné triedy. Pre každú triedu zo vstupných ontológií sa okrem základnej reprezentácie generuje aj serializačná trieda. Trieda bude slúžiť k správe inštancií entity v RDF dátovom modele. V zdrojovom kóde generovanom v jazyku Java je serializácia postavená na RDF dátovom modeli z RDF4J. Jednotlivé serializačné triedy vychádzajú z abstraktnej triedy *SerializationModel*, ktorá deklaruje metódy pre základné operácie s inštanciami v modeli. Do abstraktnej triedy sú tiež vygenerované implementácie metód potrebných pre prácu s modelom, ako napríklad *getFirstLiteralObject()*



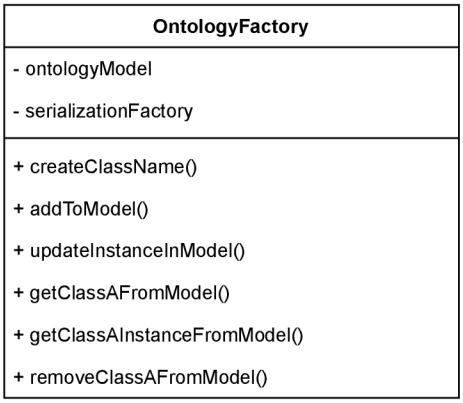
Obr. 5.12: Návrh triedy Vocabulary.

a *getAllObjects()*. Poslednou súčasťou serializačnej časti je továrenská trieda pre jednoduché vytváranie inštancií konkrétneho serializátora. Všetky serializačné komponenty budú generované do priečinku *serialization/*. Na obrázku 5.13 je možné vidieť diagram tried serializačnej časti navrhovanej štruktúry zdrojového kódu.



Obr. 5.13: Návrh serializačných tried.

Posledným popisovaným súborom generovanej štruktúry je trieda *OntologyFactory*, zobrazená na diagrame 5.14. Inšpirácia pre generovanie takejto triedy je získaná z analýzy nástroja Protege-OWL Code Generator, ktorého použitím sa generuje podobná trieda, viď kapitola 5.2.2. Cieľom tejto metódy je zoskupiť používanie všetkých vygenerovaných tried a zjednodušiť prácu s ich objektami. Pre užívateľov je táto trieda vstupným bodom do vygenerovaného zdrojového kódu, ktorý poskytuje schopnosť spravovať nové a už existujúce inštancie v dátovom modeli RDF.



Obr. 5.14: Návrh triedy OntologyFactory.

Kapitola 6

Implementácia

Aktuálna kapitola sa podrobnejšie venuje popisu implementácie. Na základe navrhnutých výstupov z predchádzajúcej kapitoly bola vytvorená implementácia aplikácie s názvom *OntoCodeMaker*.

Úvodná sekcia 6.1 popisuje technológie, ktoré boli použité pri vývoji nástroja. Štruktúra implementovaného zdrojového kódu je predstavená v časti 6.2. V ďalšej podkapitole 6.3 sú popísané vstupné argumenty nástroja. Implementáciu jednotlivých kľúčových činností nástroja postupne predstavujú sekcie 6.4, 6.5 a 6.6

6.1 Použité technológie

Výber vhodných technológií je dôležitou činnosťou, ktorá predchádza samotnej implementácií. Je ovplyvnená niekoľkými faktormi, vychádzajúcimi z požiadaviek, návrhu a využitia výsledného nástroja. Implementačným jazykom výsledného nástroja je objektovo orientovaný programovací jazyk Java. Tento jazyk bol vybraný najmä pre to, že je v ňom dostupný framework RDF4J, ktorý je využitý na parsovanie a vytváranie dátového modelu RDF. Framework RDF4J bol bližšie predstavený v sekcií 4.1.2. RDF4J bol vybraný kvôli tomu, že na rozdiel od iných frameworkov pracuje s ontológiou vo forme dátového modelu RDF a na svojich stránkach poskytuje veľmi dôkladnú dokumentáciu. Použitím jazyka Java je zabezpečená aj jednoduchá prenositeľnosť nástroja medzi rôznymi platformami, čo je jednou z požiadaviek nástroja.

Ďalším potrebným rozšírením jazyka Java bola knižnica pre generovanie za pomoci šablón. Z rôznych možností, diskutovaných v kapitole 4, bola zvolená knižnica FreeMarker. Knižnica poskytuje výkonný a flexibilný mechanizmus pre vkladanie dát z modelov do šablón a pre efektívne generovanie súborov so zdrojovými kódmi. Jej výhodou je podrobná dokumentácia, ktorá uľahčila vývoj.

Pre implementáciu testov bol využitý framework JUnit verzie 5. Je to pravdepodobne najpoužívanejší framework pre písanie jednotkových testoch v jazyku Java. O správu všetkých potrebných závislostí a riadenie buildovacieho procesu aplikácie sa staral nástroj *Apache Maven*[1].

[1] https://maven.apache.org/

6.2 Štruktúra implementovaného zdrojového kódu

Implementácia projektu je členená do dvoch modulov, a to *onto-code-maker-cli* a *onto-code-maker-core*. Dôvodom pre vytvorenie týchto dvoch modulov bol hlavne úmysel mať oddelené jadro aplikácie od rozhrania pre príkazovú riadku. Výhodou je, že v prípadnom rozšírení vieme nahradiť rozhranie príkazovej riadky za grafické rozhranie. Modul jadra sa dá tiež zakomponovávať do iných aplikácií, čím do nich získame schopnosť generovania zdrojového kódu.

Modul *onto-code-maker-cli* obsahuje rozhranie pre príkazovú riadku. Tento modul obsahuje dve triedy *CLIOptionsParser* a *OntoCodeMakerCLI*. Jeho úloha zahŕňa spracovávanie parametrov a volanie funkcionality z jadra aplikácie s príslušnými parametrami. Keďže sa volá funkcionalita z druhého modelu, tak tento modul je na ňom závislý.

Onto-code-maker-core je modul, ktorý tvorí jadro aplikácie. Koncentruje sa v ňom hlavná funkcionalita nástroja. Umiestnené sú v ňom tri balíky, ktoré implementujú jednotlivé časti definované v návrhu. Zmienené balíky sú nasledovné:

- **ontology.tool.parser** – Balík, v ktorom sa vyskytuje implementácia načítania ontológií zo vstupných súborov a ich parsovanie. Je tvorený triedou *OntologyParser*.
- **ontology.tool.mapper** – V tomto balíku sa nachádzajú triedy *OntologyMapper* a *ModelManager*. Tieto triedy slúžia k mapovaniu ontológií do vnútorného modelu nástroja predstaveného v návrhu, viď 5.4. Súčasťou balíka sú aj triedy reprezentujúce vnútorný model.
- **ontology.tool.generator** – Balík obsahujúci triedy pre generovanie výsledných zdrojových kódov v cieľových programovacích jazykoch.

Okrem predstavených balíkov, modul jadra obsahuje aj triedu *OntoCodeMaker*, ktorá prepája jednotlivé časti aplikácie. Súčasťou zdrojových priečinkov modulu sú šablóny, ktoré slúžia ku generovaniu zdrojových kódov v cieľových programovacích jazykoch.

6.3 Spracovanie vstupných argumentov

Nástroj bol implementovaný ako konzolová aplikácia. Pomocou príkazového riadku sa mu pri jeho spustení predávajú vstupné argumenty. Systém poskytuje možnosť definovania nasledujúcich vstupných argumentov:

```
[<input-file> ...]                  Vstupné súbory obsahujúce ontológie
-d, --destination <destination>     Cieľová zložka generovania
-f, --format <format>               Formát vstupných ontológií
-l, --language <language>           Cieľový jazyk generovania
-p, --package <package>             Balík generovaného zdrojového kód
-h, --help                          Vypísanie pomoci
```

Jediným povinným argumentom aplikácie je definovanie minimálne jedného vstupného súboru. Vstupný súbor by mal obsahovať definíciu ontológie, z ktorej je žiaduce vygenerovanie zdrojového kódu. Ostatné argumenty sú voliteľné. Každý z nich má určitú predvolenú hodnotu, ktorá sa využíva pri jeho absencií. Pomocou argumentu *destination* je umožnené

upresniť cieľovú zložku pre vygenerovaný balík zdrojových kódov. V prípade nešpecifikovaniu tohto atribútu sa balík vygeneruje do aktuálnej zložky. Argument *format* definuje syntax, v ktorej je vstupná ontológia napísaná. Predvolenou syntaxou je RDF/XML syntax. Hodnota argumentu *language* určuje programovací jazyk generovaného zdrojového kódu. Pri neuvedení argumentu nástroj automaticky generuje kód v jazyku Java. Dostupným jazykom je tiež Python. Posledným argumentom *package* sa špecifikuje balík, ktorý bude uvedený v zdrojovom kóde. Jej predvolenou hodnotou nie je žiadny balík, čiže v kóde bude vygenerovaný prázdny textový reťazec. Na ukážke 6.1 je možné vidieť použitie týchto argumentov pri spúšťaní nástroja.

```
$ java -jar OntoCodeMaker.jar ont.owl -d /des -f Turtle -l java -p org.pack
```

Výpis 6.1: Ukážka príkladu spustenia nástroja.

6.4 Analyzátor

Funkcionalita analyzátora popísaná v návrhu, je implementovaná v triede *OntologyParser*. Spustená je zavolaním metódy *parseOntology*, ktorej parametrami sú názvy vstupných súborov a zadaný formát ontológie. Metóda postupne parsuje všetky ontológie a vytvára z nich dátový model RDF. Parsovanie aj prevod do modelu RDF je založený na použití frameworku RDF4J, ktorý implementáciu týchto činností veľmi zjednodušil.

V ukážke kódu 6.2 je možné vidieť implementáciu parsovania ontológie z jedného súboru. V prvom kroku parsovania sa získava RDF formát vstupnej ontológie, viď riadky 3 až 11. Využívajú sa k tomu dve metódy. Pri zadanom vstupnom názve formátu sa využíva metóda *getRDFFormat*, ktorá na základe zadaného názvu vráti formát RDF. Ak názov formátu nebol uvedený, pristupuje sa k metóde *getParserFormatForFileName* z triedy *Rio*, ktorá je súčasťou RDF4J. Táto metóda sa formát pokúsi získať z prípony názvu. Ak by sa zadaný názov formátu alebo prípona súboru nezhodovala so žiadnym RDF formátom, v takom prípade sa ako formát nastaví RDF/XML. Následne sa prechádza priamo k parsovaniu, riadok 16. Parsovanie je spustené metódou *parse*. Táto metóda je tiež súčasťou triedy *Rio*. Výsledkom metódy je už priamo RDF model ontológie zo súbora. V prípade viacerých súborov sa ich modeli spájajú do jedného modelu.

```
1  logger.debug("Parsing file: " + filename);
2
3  RDFFormat format;
4  if (formatName.isEmpty()) {
5      format = Rio.getParserFormatForFileName(file.getFileName().toString())
           .orElse(RDFFormat.RDFXML);
6  } else {
7      format = getRDFFormat(formatName);
8      if(format == null){
9          throw new Exception("Not supported format name. Check help see
               supported names.");
10     }
11 }
12
13 try {
14     InputStream inputStream = Files.newInputStream(file);
```

```
15
16      Model newModel = Rio.parse(inputStream, "", format);
17      if (model == null){
18          model= newModel;
19      }else{
20          model.addAll(newModel);
21      }
22      logger.debug("File " + filename + " is loaded in the model.");
23  } catch (IOException e) {
24      e.printStackTrace();
25  }
```

Výpis 6.2: Ukážka parsovania za pomoci RDF4J.

6.5 Mapovanie

Pre mapovanie ontológie z modelu RDF do vnútorného modelu slúži trieda *OntologyMapper*. Okrem tejto triedy je súčasťou implementácie aj trieda *ModelManager*. Jej obsahom sú metódy, ktoré slúžia k získavaniu informácií z dátového modelu RDF. Trieda *OntologyMapper* tieto metódy využíva pri mapovaní konkrétnych entít ontológie.

Mapovanie sa štartuje zavolaním metódy *mapping*. V prvom kroku je potrebné odhaliť triedy ontológie. Túto úlohu sprostredkúva metóda *mapClasses()*, ktorá je zobrazená na ukážke 6.3. Metóda najskôr získa zdrojové hodnoty tried z modelu RDF pomocou metódy *getClasses()*. Nástroj dokáže spracovať triedy definované pomocou owl:class a rdfs:class. Následne sa pre každú triedu vytvára objekt triedy z vnútornej reprezentácie. Špecifickým prípadom triedy sú prázdné uzly, ktoré sú vedené ako objekty triedy *AbstractClassRepresentation*. Pre zvyšné triedy sa vytvárajú objekty z *NormalClassRepresentation*.

```
1   public void mapClasses(){
2       ontologyClasses = new ArrayList<>();
3       Set<Resource> classes = getClasses();
4       for(Resource classResource:classes){
5           if(classResource.isIRI()){
6               NormalClassRepresentation classRep = new
                    NormalClassRepresentation(
                    ((IRI)classResource).getNamespace(),
                    ((IRI)classResource).getLocalName());
7               checkAndChangeDuplicateName(ontologyClasses,classRep);
8               ontologyClasses.add(classRep);
9           }else if(classResource.isBNode()){
10              AbstractClassRepresentation abstractClassRep = new
                    AbstractClassRepresentation(
                    ((BNode)classResource).getID());
11              ontologyClasses.add(abstractClassRep);
12          }
13      }
14  }
```

Výpis 6.3: Ukážka parsovania za pomoci RDF4J.

Druhý krok mapovania sa venuje hierarchií tried. Implementovaný je súborom metód, ktoré pre danú triedu vyhľadajú v modeli RDF jej vzťahy k iným triedam. Ak trieda má nejaké vzťahy, tie sú poznačené vo vnútornej štruktúre. Tieto metódy sa vykonávajú pre každú triedu samostatne. Do vnútorného modelu evidujeme vzťahy podtried, nadtried, ekvivalenčných tried, zjednotení, prienikov a komplementov.

Posledným krokom je mapovanie vlastností. Toto mapovanie funguje na podobnom princípe ako pri triedach. Z modelu RDF sa najskôr vytiahnu identifikátory datotypových vlastností. Pre takúto vlastnosť sa vytvorí inštancia triedy *PropertyRepresentation*, ktorá reprezentuje vlastnosť vo vnútornom modeli. Následne sa postupne mapujú jej vlastnosti. Rovnaký postup sa opakuje pri vlastnostiach objektového typu s tým rozdielom, že sa ešte vytvára vzťah k triede ku ktorej vlastnosť patrí.

6.6 Generovanie za pomoci šablón

Po vytvorení vnútorného modelu ontológie sa nástroj dostáva k záverečnej fáze, ktorou je generovanie výstupnej štruktúry zdrojového kódu. Výstupná štruktúra procesu generovania korešponduje zo štruktúrou predstavenou v návrhu z kapitoly 5.5. Na základe zvoleného jazyka výstupu je vybraná trieda, ktorá generovanie vykonáva. Keďže nástroj momentálne podporuje generovanie do jazykov Java a Python implementácia obsahuje triedy *JavaGenerator* a *PythonGenerator*. Obe triedy sú špecifikáciami abstraktnej triedy *OntologyGenerator*.

Generovanie jednotlivých častí výslednej štruktúry je založené na použití šablón pomocou nástroja FreeMarker. Hlavnými komponentami takéhoto prístupu sú dátové modely a šablóny. Obsah dátových modelov je v implementácií závislý od cieľového jazyka, a preto sa tieto modely vytvárajú pomocou metód v špecifikáciach triedy generátora.

Šablóny sú napísané v jazyku FreeMarker Template. Pre každý podporovaný cieľový jazyk sa vytvárajú vlastné šablóny. Každá časť výslednej štruktúry sa generuje pomocou samostatnej šablóny. Pre generovanie do jedného programovacieho jazyka sú teda definované štyri šablóny. Jednou z definovaných šablón je šablóna pre generovanie triedy Vocabulary, ktorej obsah je možné vidieť na ukážke 6.4.

```
1  package ${package};
2
3  import org.eclipse.rdf4j.model.IRI;
4  import org.eclipse.rdf4j.model.util.Values;
5  <#list imports as item>
6      import ${item};
7  </#list>
8
9  public class ${className} {
10 <#list properties as property>
11     /**
12     * A constant representing the ${property.constantOf}
           ${property.objectName}
13     **/
14     <#if property.isPrivate ==true>
15         private
16     <#else>
```

```
17          public
18      </#if> static IRI ${property.name}
19      <#if property.isValue() == true>
20          = Values.iri("${property.getValue()}")
21      </#if>;
22
23  </#list>
24  }
```

Výpis 6.4: Ukážka šablóny pre generovanie triedy Vocabulary .

Ku generovaniu jednotlivých častí výstupnej štruktúry slúžia metódy *generateClasses*, *generateVocabulary*, *generateSerializationClasses* a *generateFactories*. Úlohou týchto metód je definovať, ktorá šablóna sa naplní akými dátami. Všetky vymenované metódy sú zaobalené do jednej metódy *generateOntology*. Zavolaním tejto metódy sa spúšťa proces generovania.

Na ukážke 6.5 je zobrazené generovanie súboru z metódy *generateVocabulary*. Základný princíp generovania zobrazený na ukážke je totožný vo všetkých ďalších triedach generovania. Pre získanie šablóny sa použí metóda *getTemplate* s parametrom typu šablóny, viď riadok 3. Na riadku 11 je vidieť volanie metódy pre vytvorenie a získanie dátového modelu. Záverečným krokom je naplnenie šablóny dátami a vygenerovanie výsledného dokumentu na riadku 13.

```
1  Writer fileWriter = new FileWriter(new File(this.outputDir +
       VOCABULARY_FILE_NAME + FILE_EXTENSION));
2
3  Template templateFile = getTemplate(TEMPLATE_TYPE.VOCABULARY);
4
5  if(templateFile == null){
6      return;
7  }
8
9  List<VocabularyConstant> properties = createVocabularyConstants();
10
11 Map<String, Object> data = getVocabularyData(properties);
12
13 templateFile.process(data, fileWriter);
```

Výpis 6.5: Ukážka časti kódu, ktorý generuje súbor Vocabulary

Kapitola 7

Testovanie a vyhodnotenie

V tejto kapitole sú predstavené metódy testovania implementovaného nástroja. Cieľom testovania bolo overenie spoľahlivosti a správnej očakávanej funkčnosti v rôznych fázach vývoja.

V úvodnej časti 7.1 sú popísané testovacie dáta. Následne v časti 7.2 sa predstavujú jednotkové testy vytvorené pre jednotlivé časti nástroja. Testovacie procesy výsledného nástroja a výstupu generovaného pomocou výsledného nástroja sú popísané v častiach 7.3 a 7.4. Záverečná časť tejto kapitoly sa zameriava na vyhodnotenie vytvoreného nástroja.

7.1 Dátová sada pre testy

Pre testovanie bola zozbieraná a vytvorená dátová sada rôznorodých ontológií. Táto dátová sada je súčasťou odovzdaného archívu. Nachádza sa v nej približne 20 reálne využívaných ontológií. Ontológie boli získané primárne z dvoch zdrojov. Prvým zdrojom bol vedúci práce, ktorý mi poskytol niekoľko ontológií z vlastných projektov. Druhým zdrojom boli voľne dostupné sady ontológií z dátovej knižnice *Climate Data Library*[1] a z archívu DBpedia[2]. Ontológie z tejto sady boli použité primárne v záverečnej fáze testovania nástroja. Tieto ontológie slúžia ako vstup pre spúšťanie finálneho nástroja a kontroluje sa správnosť vygenerovaného zdrojového kódu.

Niekoľko ontológií je vytvorených špeciálne pre testovanie vytvoreného nástroja. Jednou z takýchto vzniknutých ontológií je napríklad *Family*, ktorá bola inšpirovaná jednotlivými konštrukciami z dokumentácie jazyka OWL [12]. Jej zjednodušená verzia pod názvom *Simple Family* je dostupná v prílohe C. Je to ontológia, ktorá obsahuje všetky základné definovacie konštrukcie konštrukcie. Táto ontológia a aj jej časti sa využívajú v rôznych fázach testovania. Ontológia *Family* bola tiež preformátovaná do iných syntaxí. Ďalšími vytvorenými ontológiami je trojica, ktorá obsahuje rôzne možnosti definovania dátových a objektových vlastností. Vytvorená bola tiež sada 17 jednoduchých ontológií, ktoré sa zameriavali na rôzne definície ontologických tried a ich rôzne hierarchie. Táto sada bola využitá hlavne ako vstupy pre jednotkové testy na zabezpečenie overenia, či sa vygenerovali všetky očakávané triedy. Dominantným syntaxom vytváraných ontológií pre testovanie bol formát RDF/XML. Vlastné ontológie boli vytvárané prevažne s pomocou editora Protege, popísaného v kapitole 4.1.1. Tento editor mi poslúžil aj ku generovaniu ontológií v rozdielnych formátoch.

[1] http://iridl.ldeo.columbia.edu/ontologies/

[2] https://databus.dbpedia.org/ontologies/purl.obolibrary.org/

7.2 Jednotkové testy

Na najnižšej úrovni bolo testovanie realizované jednotkovými testami. Testy boli zamerané na jednotlivé časti nástroja. Testovacie sady pokrývali všetky kľúčové triedy a metódy. Cieľom jednotkových testov bolo zachytenie potenciálnych chýb, v čo najskoršej fáze vývoja.

Jednotlivé triedy implementujúce jednotkové testy a informácie o tom, ktoré triedy z implementácie pokrývajú a hodnoty pokrytia sú uvedené v tabuľke 7.1:

Jednotkové testy				
Názov testovacej triedy	Testovaná trieda	Počet testov	Pokrytie metód	Pokrytie riadkov
CLIOptionsParserTest	CLIOptionsParser	12	93%	89%
ParserTests	OntologyParser	13	100%	84%
ModelManagerTests	ModelManager	30	100%	96%
MapperTests	OntologyMapper	37	100%	88%

Tabuľka 7.1: Tabuľka zobrazuje pokrytia jednotlivých jednotkových testov

7.2.1 Spracovanie argumentov

Keďže súčasťou implementovaného nástroja je modul konzolovej aplikácie, ktorý cez požadované argumenty ovláda funkcionalitu, bolo potrebné overiť správne chovanie parsovania vstupných argumentov. Testovanie spracovania argumentov sa zameriava na triedu *CLIOptionsParser*, ktorá implementuje celý proces analýzy argumentov. Testy sú implementované v triede *CLIOptionsParserTest*. Požiadavkou na testy bolo overenie správnych a aj nesprávnych vstupných argumentov. Testovacia sada pozostáva z 12 testovacích prípadov, ktoré zahrňujú rôzne kombinácie vstupných argumentov.

7.2.2 Analyzátor

Jednotkové testy analyzátora spočívali v overení schopnosti vytvoriť dátový model RDF a spracovávať rôzne formáty vstupnej ontológie. Testy pracujú primárne s ontológiou *Family*, ktorá je vyjadrená v rôznych syntaxoch. Pri testovaní sa verifikuje správnosť použitia frameworku RDF4J k syntaktickej analýze a k vytvorení modelu RDF. Samotné parsovanie rôznych vstupov nie je testované, keďže to zabezpečuje RDF4J. Je vychádzané z toho, že tento framework je výrazne populárny a tým sa počíta aj s jeho vysokou spoľahlivosťou.

7.2.3 Mapovač

Testovanie mapovacej časti sa zameriava na to, či mapovač vytvorí správny vnútorný model ontologických entít z modelu RDF. Testy sú zoskupené v dvoch triedach *ModelManagerTests* a *MapperTests*. Prvá trieda pokrýva metódy k získavaniu dát z modelu RDF implementované v triede *ModelManager*. Testy v *MapperTests* overujú správnosť chovania mapovača ako celku. Potrebné bolo otestovať všetky prvky ontológie a čo najväčší počet rôzne vyjadrených definícií z ontológie.

Toto testovanie odhalilo niekoľko chýb, ako napríklad problémy pri zostavovaní hierarchickej štruktúry tried či hierarchie vlastností.

Na ukážke 7.1 je možné vidieť jeden z jednotkových testov pre mapovač. Tento test testuje mapovanie datotypovej vlastnosti do vnútornej reprezentácie. Zobrazenému testu predchádza fáza nastavenia vstupného modelu RDF. Prvé dva riadky testu na ukážke slúžia ako príprava pre vykonanie hlavnej testovanej metódy *mapDataTypeProperties*. Po príprave nasleduje vykonanie samotnej testovanej metódy. Po ukončení činnosti mapovania vlastnosti sa vo verifikačnej časti testu overuje, či bol dosiahnutý očakávaný výsledok. V tomto prípade je očakávaným výsledkom to, že trieda má naviazanú správnu inštanciu datotypovej vlastnosti.

```
1  @Test
2  @Order(30)
3  @DisplayName("30. Simple test map Datatype Property")
4  void testMapDataTypeProperty(){
5      OntologyMapper mapper = new OntologyMapper(model);
6      mapper.mapClasses();
7      mapper.mapDataTypeProperties();
8
9      int mappedPropSize = mapper.getCollectionOfMappedProperties().size();
10     assertEquals(3, mappedPropSize, "Datatype property was not mapped.");
11
12     PropertyRepresentation property =
           mapper.getMappedProperties().get(hasLuckyNumbers);
13     assertEquals(property.getType(), PROPERTY_TYPE.DATATYPE,"Wrong type.");
14     assertEquals(property.getClassName(),"Human","Wrong class name.");
15     assertEquals(property.getRangeResource(),hasAgeDatatype,"Wrong range
           value.");
16     assertFalse(property.isFunctional(),"Property is functional.");
17
18     ClassRepresentation tc = mapper.getMappedClasses().get(classHuman);
19     PropertyRepresentation result = findProperty(tc.getProperties(),
           hasLuckyNumbers);
20     assertNotNull(result,"hasLuckyNumbers property not found.");
21 }
```

Výpis 7.1: Ukážka jednotkového testu mapovača

7.3 Testy výsledného nástroja na generovanie

Ďalšou časťou testovania je skupina rôznych testov, ktoré preverujú celý implementovaný nástroj. Niektoré z týchto testov sa nachádzajú v triede *FileGeneratorTests*. Testovacie prípady v nej spúšťajú jadro nástroja s rôznymi vstupnými ontológiami. Po spusteniach sa overuje, či sa nástroj skončil s očakávaným výsledkom. Ak nástroj vygeneroval zdrojový kód, overuje sa tiež či vygeneroval všetky očakávané triedy a metódy.

Zvyšné testy v tejto fáze už boli vykonávané manuálne. Sada testov prebiehala tak, že sa ručne spúšťala aplikácia pomocou príkazovej riadky a následne sa kontroloval vygenerovaný zdrojový kód.

Ďalším parametrom testovania je výkonnosť nástroja pri ontológiách s veľkým obsahom entít. Pri tomto testovaní sa zistilo, že nástroj zvláda generovať rozsiahlejšie ontológie za pomerne rozumný čas.

Výsledné testovanie prebiehalo aj overovaním funkčnosti nástroja na rôznych operačných systémoch. Konkrétne bol nástroj testovaný na operačných systémoch Linux (Ubuntu 20.04) a Windows (Windows 10). Spúšťanie prebiehalo úspešne na oboch platformách. Z výsledku tohto testovania môžeme usúdiť, že nástroj je použiteľný na všetkých testovaných operačných systémoch.

7.4 Testy vygenerovaného zdrojového kódu

Dôležitou fázou testovania bolo overiť, či nástrojom vygenerovaný zdrojový kód je syntakticky validný pre daný jazyk a či vygenerované metódy fungujú očakávaným spôsobom. Tieto testy pomohli odhaliť mnohé problémy s vygenerovanými implementáciami serializačných metód, problémy pri závislostiach a chyby v syntaxi. Testovanie vygenerovaného zdrojového kódu v jazyku Java prebiehalo v samostatnom testovacom module s názvom *onto-code-maker-result-tester*, ktorý je súčasťou archívu nástroja. Pre samotný nástroj tento modul nie je potrebný. Tento modul obsahuje testovacie sady a zbierku vygenerovaných zdrojových kódov, ktoré sa testujú.

Testovanie zahŕňalo dva spôsoby. Ako prvé bola overená syntaktická stránka vygenerovaného kódu. K tomuto procesu dopomohlo vývojové prostredie *IntelliJ IDEA*, ktoré po otvorení súboru s kódom verifikuje jeho syntaktickú správnosť a prípadne označí chyby. Keď bol generovaný zdrojový kód syntakticky správny, prešlo sa k overeniu funkčnosti vygenerovaných tried a metód. Pre tento účel bolo vytvorených 5 testovacích sád. Tieto sady overujú funkčnosť 4 balíkov zdrojových kódov v jazyku Java a 1 balík v programovacom jazyku Python. Tieto balíky boli vygenerované s pomocou implementovaného nástroja. Napríklad, jednu sadu v jazyku Java a tiež v jazyku Python tvoria testy pre triedy vygenerované z ontológie *Family*. Všetky sady je možné vidieť v tabuľke 7.2.

Testovacie sady			
Testovacia trieda	Vstupná ontológia	Počet testov	Poznámka
AllRangeTypesTests	allRangeTypes.owl	36	Testuje všetky podporované dátové typy vlastností.
AllDatatypePropTests	allDatatypeProp.owl	16	Testy na rôzne definície datotypovej vlastnosti.
AllObjectPropTests	allObjectProp.owl	19	Testy na rôzne definície objektovej vlastnosti.
FamilyTests	family.owl	31	Testy základnej funkcionality zdrojového kódu v jazyku Java.
FamilyTestsPython	family.owl	26	Testy základnej funkcionality zdrojového kódu v jazyku Python.

Tabuľka 7.2: Tabuľka zobrazuje testovacie sady na testovanie vygenerovaného kódu

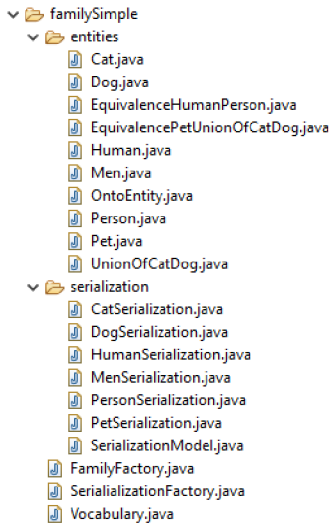
Jednotlivé testovacie prípady z testovacích sád overujú očakávanú činnosť rôznych vygenerovaných metód. Test z ukážky 7.2 verifikuje serializačnú metódu pridávania inštancie triedy *Human* do dátového modelu RDF.

```
1  @Test
2  @Order(6)
3  @DisplayName("6.Simple test to add class instance to model")
4  void testAddToModel() {
5      Human h = factory.createHuman(MartinHuman);
6      int startSize = model.size();
7      factory.addToModel(h);
8
9      assertEquals(startSize+1, model.size(),"Issue add instance to model.");
10
11     Model resultModel = model.filter(Values.iri(MartinHuman), RDF.TYPE,
           Vocabulary.HUMAN_CLASS_IRI);
12     assertEquals(1, resultModel.size(), "Issue add instance to model 2.");
13 }
```

Výpis 7.2: Ukážka testu vygenerovaných triedy zobrazujúca pridávanie do modelu

7.5 Ukážka vygenerovaného zdrojového kódu

Výsledný nástroj generuje zdrojový kód, ktorý je zoskupený do štruktúry popísanej v návrhu 5.5. Vygenerovanú štruktúru ontológie *Simple Family* je možné vidieť na obrázku 7.1.



Obr. 7.1: Zobrazenie výslednej štruktúry generovania nástroja OntoCodeMaker

Hlavnými časťami generovaného zdrojového kódu sú reprezentácie ontologických tried a k nim priľahlé serializačné triedy. Na ukážke 7.3 je možné vidieť vygenerovanú triedu, ktorá reprezentuje ontologickú triedu *Men* z ontológie *Simple Family*. Táto trieda je podtriedou triedy *Human*. Na riadkoch 8 až 15 je možné vidieť komentár s popisom triedy. V tomto komentári sa vyskytujú všetky anotácie, ktoré popisovali odpovedajúcu triedu z ontológie.

Súčasťou komentára sú aj vyskytujúce sa vlastnosti ako *disjointWith* a *restriction*, ktoré sa pre ich kompikovanosť do zdrojového kódu nemapujú. Na ukážke je možné vidieť aj vygenerovanie objektovej vlastnosti *hasCat*, ktorá vyjadruje inštanciu triedy *Cat*. Keďže sa nejedná o funkcionálnu vlastnosť tak jej kardinalita nie je obmedzená. Ku vygenerovanej vlastnosti patria tiež metódy *addHasCat* a *getHasCat*.

```java
package ontology.generator.classes.examples.familySimple.entities;

import org.eclipse.rdf4j.model.*;
import java.util.List;
import java.util.ArrayList;
import ontology.generator.classes.examples.familySimple.Vocabulary;

/**
* This is the class representing the
   Men(http://www.ontocodemaker.org/Family#Men) class from ontology
* This class is subclass of Human
*
* Men
* This is men
* Generated by OntoCodeMaker
**/
public class Men extends Human {

    // IRI Constant of Class
    public static IRI CLASS_IRI = Vocabulary.MEN_CLASS_IRI;

    /**
    * Property http://www.ontocodemaker.org/Family#hasCat
    **/
    private List<Cat> hasCat = new ArrayList<>();

    public Men(IRI iri){
            super(iri);
    }

    public IRI getClassIRI() {
        return CLASS_IRI;
    }

    public void addHasCat(Cat hasCat){
        this.hasCat.add(hasCat);
    }
    public List<Cat> getHasCat(){
        return hasCat;
    }
}
```

Výpis 7.3: Ukážka vygenerovanej triedy *Men* z ontológie *Simple Family*

Na ďalšej ukážke vygenerovaného kódu 7.4 sa zobrazujú niektoré metódy zo serializačnej triedy, ktorá patrí k triede *Men*. Pre zachovanie prehľadnosti sú vybrané tri serializačné metódy. Celú triedu je možné vidieť v prílohe D. Metóda *addToModel* pridáva inštanciu triedy *Men* do modelu ontológie. Metóda *getInstanceFromModel* získava inštanciu definovanú zadaným identifikátorom. Poslednou zverejnenou metódu je *removeInstanceFromModel*, ktorá odstraňuje inštanciu zadaného identifikátora z modelu ontológie.

```
1     @Override
2     public void addToModel(Model model, Men men) {
3         model.add(men.getIri(),RDF.TYPE, men.getClassIRI());
4         addPropertiesToModel(model,men);
5
6     }
7
8     @Override
9     public Men getInstanceFromModel(Model model,IRI instanceIri,int
          nestingLevel) throws Exception{
10        Model statements = model.filter(instanceIri,RDF.TYPE,Men.CLASS_IRI);
11        if(statements.size() != 0){
12            Men men = new Men(instanceIri);
13            if(nestingLevel > 0){
14                nestingLevel--;
15                setProperties(model, men,nestingLevel);
16            }
17            return men;
18        }
19        return null;
20    }
21
22    @Override
23    public void removeInstanceFromModel(Model model,IRI instanceIri) {
24        Set<Value> rdfCollections =
              model.filter(instanceIri,null,null).objects();
25        List<Value> listOfnodes = rdfCollections.stream().filter(o ->
              o.isBNode()).collect(Collectors.toList());
26        for(Value node:listOfnodes){
27            model.removeAll(getModelRDFCollection(model,(BNode)node));
28            model.remove(instanceIri,null,(BNode)node);
29        }
30        model.remove(instanceIri,RDF.TYPE,Men.CLASS_IRI);
31        model.remove(instanceIri,null,null);
32    }
```

Výpis 7.4: Ukážka metód z vygenerovanej serializačnej triedy k triede *Men*

7.6 Vyhodnotenie nástroja

Implementovaný generátor je určite funkčný a spoľahlivý nástroj pre tvorbu korektného programového kódu. O tomto úsudku svedčia vykonané testy generátora ako aj testy vygenerovaného zdrojového kódu. Schopnosťou nástroja je generovať funkčný zdrojový kód v jazykoch Java a Python. Nástroj dokáže prijať vstupné ontológie definované vo formátoch RDF/XML, Turtle, N-Quads, JSON-LD, N-Triples, Trix, Trig a RDF/JSON. Využiteľnosť nástroja je očakávaná hlavne u vývojárov, pri tvorbe aplikácií pracujúcich s dátami definovanými v ontológiách.

Finálny nástroj vychádza z návrhu predstaveného v kapitole 5 a spĺňa všetky definované požiadavky. Obmedzením riešenia je schopnosť generovať kód len z určitej podmnožiny všetkých možných definícií ontologických prvkov, keďže niektoré prvky je príliš náročné premietnuť do objektovo orientovaného jazyka. Nástroj je navrhnutý tak, že jeho jadro s hlavnou funkcionalitou generovania sa dá bez problémov použiť v iných projektoch. Jeho verzia vo forme konzolovej aplikácie teda nemusí byť konečná.

Nástroj umožňuje jednoduché rozšírenie o shopnosť generovať kód v ďalších objektovo orientovaných programovacích jazykoch. Potenciálnym rozšírením nástroja je aj grafické rozhranie a sprístupnenie používania nástroja prostredníctvom internetu. Nástroj by sa mohol v ďalších fázach vylepšovať aj o prípadné požiadavky užívateľov.

Kapitola 8

Záver

Táto práca je zameraná na oblasť sémantického webu. Jej hlavným cieľom bolo vytvorenie nástroja, ktorí umožňuje generovanie programového kódu z definície ontológie. Tento nástroj sa podarilo úspešne navrhnúť, implementovať a otestovať. Je dostupný na priloženom pamäťovom úložisku alebo v repozitáry na adrese https://github.com/tomsvet/onto-code-maker.

Na začiatku tejto práce bolo potrebné sa oboznámiť so sémantickým webom. Oblasť sémantického webu je ale rozsiahla. Práca sa práve preto zameriava iba na vysvetlenie základných blokov a technológií, ktoré boli priamo využité v tejto práci. Týmito technológiami boli RDF, ontológia a ontologické jazyky. Ďalším krokom práce bolo naštudovanie si existujúcich nástrojov pre spracovanie RDF a OWL definícií a pre generovanie programového kódu. Po získaní základných teoretických poznatkov boli analyzované požiadavky na výsledný nástroj a vytvorený jeho návrh. Kľúčovou časťou celého návrhu je spôsob, akým sa budú jednotlivé entity ontológie mapovať do zdrojového kódu. Následne na základe návrhu prebiehala implementácia. Nástroj bol naimplementovaný v jazyku Java s pomocou frameworku na spracovanie ontológií s názvom RDF4J a knižnice Freemarker, ktorá slúži ku generovaniu pomocou šablón. Počas implementácie a aj po nej bolo realizovaných mnoho testov, aby sa overila funkčnosť a použiteľnosť nástroja i generovaného zdrojového kódu.

Výsledkom práce je generátor implementovaný ako konzolová aplikácia. Implementovaný nástroj spĺňa všetky definované požiadavky a je plne funkčný. Momentálne nástroj podporuje generovanie programovacieho kódu v jazykoch Java a Python. Jednou z limitácií nástroja je, že je schopný generovať len z určitej podmnožiny jazyka OWL. OWL je totižto expresívny jazyk a bolo by náročné a v niektorých prípadoch aj zbytočné premapovávať všetky jeho konštrukcie do programového kódu.

Možným rozšírením nástroja môže byť pridanie schopnosti generovania aj do ďalších programovacích jazykov. Nástroj bol navrhnutý tak, aby bolo toto rozširovanie čo najjednoduchšie. Stačí k tomu pridanie šablón a implementovanie jednej triedy generátora, vychádzajúcej z hlavného generátora. Ďalšími možnými rozšíreniami sú vytvorenie grafického užívateľského rozhrania alebo rozšírenie podmnožiny jazyka OWL, ktorú je nástroj schopný premapovať do zdrojového kódu.

Literatúra

[1] Antoniou, G. a Harmelen, F. van. *A Semantic Web Primer*. 2. vyd. MIT Press, 2012. ISBN 9780262018289.

[2] Apache Software Foundation. *What is Apache FreeMarker™?* [online]. 2021 [cit. 28.12.2021]. Dostupné z: https://freemarker.apache.org/.

[3] Berners Lee, T. a Fischetti, M. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. 1. vyd. Harper Business, 2000. ISBN 978-0062515872.

[4] Berners Lee, T., Hendler, J. a Lassila, O. The Semantic Web: A New Form of Web Content That is Meaningful to Computers Will Unleash a Revolution of New Possibilities. *ScientificAmerican.com*. 1. vyd. Máj 2001, zv. 284, č. 5, s. 34–43.

[5] Borst, W. N. *Construction of Engineering Ontologies for Knowledge Sharing and Reuse*. Netherlands, 1997. Dizertačná práca. University of Twente. ISBN 90-365-0988-2.

[6] Brickley, D. a Guha, R. *RDF Schema 1.1* [online]. 2014 [cit. 23.11.2021]. Dostupné z: https://www.w3.org/TR/rdf-schema/.

[7] Carothers, G. a Prud'hommeaux, E. *RDF 1.1 Turtle* [online]. W3C Recommendation. W3C, február 2014 [cit. 26.11.2021]. Dostupné z: https://www.w3.org/TR/2014/REC-turtle-20140225/.

[8] Carothers, G. a Seaborne, A. *RDF 1.1 N-Triples* [online]. W3C Recommendation. W3C, február 2014 [cit. 26.11.2021]. Dostupné z: https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[9] Eclipse Foundation. *The Eclipse RDF4J Framework* [online]. 2021 [cit. 28.12.2021]. Dostupné z: https://rdf4j.org/about/.

[10] Goldman, N. M. Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer. In: Fensel, D., Sycara, K. a Mylopoulos, J., ed. *The Semantic Web - ISWC 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, s. 850–865. ISBN 978-3-540-39718-2.

[11] Gruber, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition*. 1. vyd. 1993, zv. 5, č. 2, s. 199–220. DOI: 10.1006/knac.1993.1008.

[12] Hitzler, P., Krötzsch, M., Parsia, B., Patel Schneider, P. F. a Rudolph, S. *OWL 2 Web Ontology Language, Document Overview* [online]. 2. vyd. 2012 [cit. 23.11.2021]. Dostupné z: https://www.w3.org/TR/owl2-primer/.

[13] Horridge, M. a Bechhofer, S. The OWL API: A Java API for OWL Ontologies. *Semant. Web.* IOS Press. jan 2011, zv. 2, č. 1, s. 11–21. ISSN 1570-0844.

[14] Koivunen, M.-R. a Miller, E. *W3C Semantic Web Activity* [online]. 2001 [cit. 25.11.2021]. Dostupné z: https://www.w3.org/2001/12/semweb-fin/w3csw.

[15] Motik, B., Grau, B. C., Horrocks, I., Wu, Z., Fokoue, A. et al. *OWL 2 Web Ontology Language Profiles* [online]. 2. vyd. 2012 [cit. 23.11.2021]. Dostupné z: https://www.w3.org/TR/owl2-profiles/.

[16] Musen, M. A. a Protégé Team the. The Protégé project: A look back and a look forward. *AI Matters.* 1. vyd. Jún 2015, zv. 1, č. 4, s. 4–12. DOI: 10.1145/2557001.25757003.

[17] Parr, T. J. Enforcing strict model-view separation in template engines. *WWW '04.* Máj 2004, s. 224–233. DOI: 10.1145/988672.988703.

[18] Parr, T. J. *StringTemplate 4 Documentation* [online]. 2018 [cit. 28.12.2021]. Dostupné z: https://github.com/antlr/stringtemplate4/blob/master/doc/index.md.

[19] Quin, L. *Extensible Markup Language (XML)* [online]. 2022 [cit. 26.11.2022]. Dostupné z: https://www.w3.org/XML.

[20] Schreiber, G. a Gandon, F. *RDF 1.1 XML Syntax* [online]. W3C Recommendation. W3C, február 2014 [cit. 26.11.2021]. Dostupné z: https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

[21] Schreiber, G. a Raimond, Y. *RDF 1.1 Primer* [online]. 2014 [cit. 29.11.2021]. Dostupné z: https://www.w3.org/TR/rdf11-primer/.

[22] Studer, R., Benjamins, V. a Fensel, D. Knowledge engineering: Principles and methods. *Data Knowledge Engineering.* 1998, zv. 25, č. 1, s. 161–197. DOI: 10.1016/S0169-023X(97)00056-6.

[23] Svátek, V. *Ontologie a WWW.* DATAKON 2002: sborník databázové konference: Brno. Česká republika, 2002. ISBN 80-210-2958-7.

[24] The Apache Software Foundation. *Jena tutorials* [online]. 2021 [cit. 28.12.2021]. Dostupné z: https://jena.apache.org/tutorials/index.html.

[25] Uschold, M. a Grüninger, M. Ontologies: Principles, methods and applications. *The Knowledge Engineering Review.* Január 1996, zv. 11, s. 93–136. DOI: 10.1017/S0269888900007797.

[26] Vendetti, J. *Protege-OWL API Programmer's Guide* [online]. 2012 [cit. 28.12.2021]. Dostupné z: https://protegewiki.stanford.edu/wiki/ProtegeOWL_API_Programmers_Guide.

[27] W3C. *W3C Semantic Web Activity* [online]. 2001 [cit. 25.11.2021]. Dostupné z: https://www.w3.org/2001/sw/.

[28] W3C OWL Working Group. *OWL 2 Web Ontology Language, Document Overview* [online]. 2. vyd. 2012 [cit. 23.11.2021]. Dostupné z: https://www.w3.org/TR/owl-overview/.

Príloha A

Obsah priloženého pamäťového média

- **onto-code-maker/** – Adresár so zdrojovými kódmi nástroja
- **srcTZ/** – Adresár so zdrojovými kódmi technickej správy
- **xsvetl05_tz.pdf** – Technická správa v PDF
- **README.txt** – Popis k nástroju
- **ontologies/** – Adresár so všetkými ontológiami využívanými v procese testovania
- **outputs/** – Adresár s vygenerovanými zdrojovými kódmi z ontológií

Príloha B

Mapovanie

Mapovanie ontologických elementov do jazyka Java		
OWL entita	Vyjadrenie v OWL	Java entita
Class	Class A	class A
	Class A (s podtriedou vychádzajúcou z viac než jednej triedy)	interface A; class A implements interface A
	A subClassOf B	class A; class B extends A
	A subClassOf B; C subClassOf B	interface A; interface B; class B extends interface A, interface B
	A equivalentClass B	interface AB; A implements interface AB; B implements interface AB
	A intersectionOf B and C	class A implements B,C
	A unionOf B and C	interface/class B and C extends/implements interface A
	A complementOf B	B extends C; A extends C
Property A	domain B	propertyA (v triede B)
	range B	List<B> propertyA
	functional A	B propertyA
	inverse functional A	A propertyX (definované v triede z hodnoty range)
	equivalentOf B	propertyA (je vlastnosť s rovnakou domain a range hodnotou ako v B)
	subPropertyOf B	propertyA (je podvlastnosť s rovnakou domain a range hodnotou ako v B, všetky hodnoty z propertyB sú aj v propertyA)
	inverseOf B	List<A> propertyX (definované v triede z hodnoty range)

Tabuľka B.1: Tabuľka zobrazuje mapovanie ontologických elementov do jazyka Java

Mapovanie ontologických elementov do jazyka Python		
OWL entita	Vyjadrenie v OWL	Python entita
Class	Class A	class A
	A subClassOf B	class A, class B(A)
	A subClassOf B; C subClassOf B	class B(A,B)
	A equivalentClass B	class AB, class A(AB), class B(AB)
	A intersectionOf B and C	class A(B,C)
	A unionOf B and C	class A, class B(A), class C(A)
	A complementOf B	class B(C), class A(C)
Property A	domain B	propertyA (v triede B)
	range B	propertyA = set()
	functional A	propertyA = None
	inverse functional A	propertyX (definované v triede z hodnoty range)
	equivalentOf B	propertyA (je vlastnosť s rovnakou domain a range hodnotou ako v B)
	subPropertyOf B	propertyA (je podvlastnosť s rovnakou domain a range hodnotou ako v B, všetky hodnoty z propertyB sú aj v propertyA)
	inverseOf B	propertyX = set() (definované v triede z hodnoty range)

Tabuľka B.2: Tabuľka zobrazuje mapovanie ontologických elementov do jazyka Python

Príloha C

Ontológia Simple Family

V syntaxe RDF/XML definovaná vytvorená ontológia Simple Family.

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.ontocodemaker.org/familySimple.owl#Family#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://www.ontocodemaker.org/familySimple.owl#Family">
        <rdfs:comment>This ontology was created for testing.</rdfs:comment>
        <rdfs:label>Family</rdfs:label>
    </owl:Ontology>

    <!--////////////////////// Object properties ///////////////////////////////-->

    <owl:ObjectProperty rdf:about="http://www.ontocodemaker.org/Family#hasCat">
        <rdfs:domain rdf:resource="http://www.ontocodemaker.org/Family#Men"/>
        <rdfs:range rdf:resource="http://www.ontocodemaker.org/Family#Cat"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://www.ontocodemaker.org/Family#hasDog">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
        <rdfs:domain rdf:resource="http://www.ontocodemaker.org/Family#Human"/>
        <rdfs:range rdf:resource="http://www.ontocodemaker.org/Family#Dog"/>
        <rdfs:comment>This is hasDog</rdfs:comment>
        <rdfs:label>hasDog</rdfs:label>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://www.ontocodemaker.org/Family#hasDogEq">
        <owl:equivalentProperty rdf:resource="http://www.ontocodemaker.org/Family#
            hasDog"/>
    </owl:ObjectProperty>

    <!--////////////////////// Data properties ///////////////////////////////-->

    <owl:DatatypeProperty rdf:about="http://www.ontocodemaker.org/Family#hasAge">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
        <rdfs:domain rdf:resource="http://www.ontocodemaker.org/Family#Human"/>
        <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#nonNegativeInteger"
            />
        <rdfs:comment>This is hasAge</rdfs:comment>
        <rdfs:label>hasAge</rdfs:label>
    </owl:DatatypeProperty>
```

```
    <!--////////////////////// Classes ///////////////////////////////-->

    <owl:Class rdf:about="http://www.ontocodemaker.org/Family#Cat"/>

    <owl:Class rdf:about="http://www.ontocodemaker.org/Family#Dog"/>

    <owl:Class rdf:about="http://www.ontocodemaker.org/Family#Human">
        <owl:equivalentClass rdf:resource="http://www.ontocodemaker.org/Family#Person"
            />
        <rdfs:comment>Main Human class</rdfs:comment>
        <rdfs:label>Human</rdfs:label>
    </owl:Class>

    <owl:Class rdf:about="http://www.ontocodemaker.org/Family#Men">
        <rdfs:subClassOf rdf:resource="http://www.ontocodemaker.org/Family#Human"/>
        <rdfs:comment>This is men</rdfs:comment>
        <rdfs:label>Men</rdfs:label>
    </owl:Class>

    <owl:Class rdf:about="http://www.ontocodemaker.org/Family#Person"/>

    <owl:Class rdf:about="http://www.ontocodemaker.org/Family#Pet">
        <owl:equivalentClass>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="http://www.ontocodemaker.org/Family#Cat"
                        />
                    <rdf:Description rdf:about="http://www.ontocodemaker.org/Family#Dog"
                        />
                </owl:unionOf>
            </owl:Class>
        </owl:equivalentClass>
    </owl:Class>

</rdf:RDF>
```

Príloha D

Vygenerovaná serializačná trieda k triede Men zo Simple Family

```java
package ontology.generator.classes.examples.familySimple.serialization;

import org.eclipse.rdf4j.model.*;
import java.util.Collection;
import java.util.Set;
import java.util.HashSet;
import java.util.*;
import java.util.stream.Collectors;
import ontology.generator.classes.examples.familySimple.entities.*;
import org.eclipse.rdf4j.model.util.Values;
import org.eclipse.rdf4j.model.vocabulary.RDF;
import java.time.LocalTime;
import ontology.generator.classes.examples.familySimple.Vocabulary;

public class MenSerialization extends SerializationModel<Men>{

    @Override
    public void addToModel(Model model, Men men) {
        model.add(men.getIri(),RDF.TYPE, men.getClassIRI());
        addPropertiesToModel(model,men);
    }

    protected void addPropertiesToModel(Model model, Men men) {
        List<OntoEntity> hasCatPom = new ArrayList<>();
        hasCatPom.addAll(men.getHasCat());
        setRDFCollection(model,men.getIri(),Vocabulary.HASCAT_PROPERTY_IRI,hasCatPom);

         new HumanSerialization().addPropertiesToModel(model, men);
    }

    protected void setProperties(Model model,Men men,int nestingLevel) throws Exception{
        Set<Resource> hasCat =
             super.getAllResourceObjects(model,Vocabulary.HASCAT_PROPERTY_IRI,men.getIri());
        for(Resource propValue:hasCat){
            if(propValue.isIRI()) {
                Cat hasCatInstance = new CatSerialization().getInstanceFromModel(model,
                    (IRI) propValue,nestingLevel);
                if(hasCatInstance == null) throw new Exception("Instance of " +
                    propValue.stringValue() + " is not in model.");
                men.addHasCat(hasCatInstance);
```

```
38            }else if(propValue.isBNode()){
39                List<Value> listOfValues = super.getRDFCollection(model,(BNode)propValue);
40                for(Value value:listOfValues){
41                    if(value.isIRI()){
42                        Cat hasCatInstance = new
                              CatSerialization().getInstanceFromModel(model,
                              (IRI)value,nestingLevel);
43                        if(hasCatInstance == null) throw new Exception("Instance of " +
                              propValue.stringValue() + " is not in model.");
44                        men.addHasCat(hasCatInstance);
45                    }
46                 }
47            }
48        }
49        new HumanSerialization().setProperties(model, men,nestingLevel);
50    }
51
52    @Override
53    public Men getInstanceFromModel(Model model,IRI instanceIri,int nestingLevel) throws
          Exception{
54        Model statements = model.filter(instanceIri,RDF.TYPE,Men.CLASS_IRI);
55        if(statements.size() != 0){
56            Men men = new Men(instanceIri);
57            if(nestingLevel > 0){
58                nestingLevel--;
59                setProperties(model, men,nestingLevel);
60            }
61            return men;
62        }
63        return null;
64    }
65
66    @Override
67    public Collection<Men> getAllInstancesFromModel(Model model,int nestingLevel)throws
          Exception{
68        Model statements = model.filter(null,RDF.TYPE,Men.CLASS_IRI);
69        Collection<Men> allInstances = new ArrayList<>();
70        for(Statement statement:statements){
71            Resource subject = statement.getSubject();
72            if(subject.isIRI()){
73                IRI iri = (IRI) subject;
74                Men men = getInstanceFromModel(model,iri,nestingLevel);
75                allInstances.add(men);
76            }
77        }
78        return allInstances;
79    }
80
81    @Override
82    public void removeInstanceFromModel(Model model,IRI instanceIri) {
83        Set<Value> rdfCollections = model.filter(instanceIri,null,null).objects();
84        List<Value> listOfnodes = rdfCollections.stream().filter(o ->
              o.isBNode()).collect(Collectors.toList());
85        for(Value node:listOfnodes){
86            model.removeAll(getModelRDFCollection(model,(BNode)node));
87            model.remove(instanceIri,null,(BNode)node);
88        }
89        model.remove(instanceIri,RDF.TYPE,Men.CLASS_IRI);
90        model.remove(instanceIri,null,null);
```

```
91      }
92
93      @Override
94      public void updateInstanceInModel(Model model,Men men){
95          Set<Value> rdfCollections = model.filter(men.getIri(),null,null).objects();
96          List<Value> listOfnodes =
                rdfCollections.stream().filter(Value::isBNode).collect(Collectors.toList());
97          for(Value node:listOfnodes){
98              model.removeAll(getModelRDFCollection(model,(BNode)node));
99              model.remove(men.getIri(),null,node);
100         }
101         Model statements = model.filter(men.getIri(),null,null);
102         statements.removeIf(x -> !x.getPredicate().equals(RDF.TYPE));
103
104         addPropertiesToModel(model,men);
105     }
106
107 }
```