Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Bakalářská práce

Nástroj pro poloautomatickou analýzu projektu Java aplikace a vytvoření jeho objektové reprezentace

Plzeň, 2019

Lukáš Pavlík

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 25.června 2019

Lukáš Pavlík

Abstract

This bachelor thesis' main goal is to create a tool for semi-automatic analysis of Java application project and creation of its object representation. Structures of Java projects in three different IDE's and one tool for Java project management, syntactic analysis, data structures used by syntactic analysers and some methods used for syntactic analysis were described in the first part of this thesis. Second part of this thesis contains proposition and description of the implementation of the created tool. Created tool is able to load Java projects, create their object representation, store the representation in database and export the stored projects from database. The created tool is able to extract individual methods and classes from the source code files contained in the project and recreate the source files from the extracted classes and methods.

Abstrakt

Tato práce se zabývá vytvořením nástroje pro poloautomatickou analýzu projektu Java aplikace a vytvoření jeho objektové reprezentace. V první části byly popsány struktury Java projektů ve třech různých IDE a jednom nástroji pro správu Java projektů, syntaktická analýza a datové struktury, které používají syntaktické analyzátory a některé metody používané při syntaktické analýze. Druhá část této práce je věnována návrhu a popisu implementace vytvořeného nástroje. Vytvořený nástroj je schopen načítat Java projekty, vytvářet jejich objektovou reprezentaci, uložit tuto reprezentaci do databáze a exportovat uložené projekty z databáze. Vytvořený nástroj je schopen získat ze souborů se zdrojovým kódem jednotlivé třídy a metody a znovu tyto soubory vytvořit ze získaných tříd a metod.

Obsah

1 Úvod....4
2 Struktura Java projektu v různých IDE....5
2.1 IDE projekt v jazyce Java....5
2.2 Konvence pro pojmenování v jazyce Java....5
2.3 Struktura Java projektu v IDE Eclipse....6
2.4 Struktura Java projektu v IDE NetBeans....7
2.5 Struktura Java projektu v IDE Intellij IDEA....9
2.6 Struktura Java projektu v Apache Maven....10
3 Syntaktická analýza....13
3.1 Regulární výrazy....14
3.2 Formální gramatika....16
4.3 Abstraktní syntaktický strom a derivační strom....18
4.3.1 Abstraktní syntaktický strom....19
4.3.2 Derivační strom....20
4.4 Metody syntaktické analýzy....20
4.4.1 Rekurzivní sestup (shora dolů)....21
4.4.2 Backtracking (shora dolů)....21
4.4.3 Prediktivní syntaktická analýza (shora dolů)....22
4.4.4 Metoda přesun-redukce (zdola nahoru)....23
5 Analýza problému....24
5.1 Specifikace požadavků....24
5.1.1 Vstupní data....24
5.1.2 Zpracování dat....24
5.1.3 Výstup aplikace....24
5.2 Případy užití aplikace....25
5.2.1 Přidání projektu....25
5.2.2 Smazání projektu....25
5.2.3 Prohlížení projektu a jeho částí....26

5.2.4 Export projektu..26
5.3 Načítání souborů...26
5.3.1 DFS...26
5.3.2 BFS...27
5.4 Analýza zdrojového kódu...28
5.5 Existující nástroje pro analýzu zdrojového kódu.......................................28
5.5.1 ANTLR..28
5.5.2 JavaParser...28
5.6 Objektová reprezentace dat...29
5.7 Uložení dat do databáze..30
5.8 Existující nástroje pro práci s databází..31
5.8.1 JDBC...31
5.8.2 Hibernate...31
5.9 Návrh databáze...32
5.10 Export projektu...33
5.11 Grafické uživatelské rozhraní..33
5.12 Návrh grafického uživatelského rozhraní..34
6 Popis implementace...35
6.1 Použité nástroje a knihovny..35
6.2 Popis tříd aplikace...36
6.2.1 Třída App, FileIO, DatabaseConnection a Parser...................................36
6.2.2 Třída AppWindow...36
6.2.3 Třídy ClassRep, MethodRep, FileSource, FileOther, Directory a TreeElement...37
6.2.4 Třídy FileType, MethodType a ClassType..38
6.2.5 Třída Strings..39
6.3 Načítání vstupních dat...39
6.3.1 Načítání souborů se zdrojovým kódem..40
6.3.2 Načítání souborů bez zdrojového kódu..40
6.4 Získání částí zdrojového kódu tříd...42

6.4.1 Zpracování tříd, rozhraní a výčtových typů....................................43
6.4.2 Zpracování metod a konstruktorů..44
6.5 Ukládání a získávání dat z databáze...45
6.5.1 Ukládání dat do databáze..45
6.5.2 Získávání dat z databáze..46
6.6 Export projektu...46
7 Testování...48
7.1 Testování jednotkovými testy...48
7.2 Funkcionální testování...50
7.3 Testování grafického uživatelského rozhraní...50
7.4 Testování na reálných datech..51
8 Omezení a možnosti rozšíření aplikace..52
9 Závěr..53
Zdroje..54
A Uživatelská příručka...57
A.1 Překlad a spuštění aplikace...57
A.2 Hlavní okno aplikace...57
B UML diagram tříd...60

1 Úvod

Cílem této práce je vytvořit v jazyce Java nástroj, který umožní analýzu projektu Java aplikace a vytvoří jeho objektovou reprezentaci až na úroveň tříd a metod. Výslednou objektovou reprezentaci nástroj uloží do databáze a bude schopný ji z databáze vyexportovat. Projekty mohou obsahovat úmyslně zanesené chyby. Tyto exportované projekty budou sloužit pro testování efektivity testovacích metod a umožní porovnávání, které z testovacích metod odhalí více chyb. Úkolem této práce ale není zanášení chyb do projektů, jejichž objektová reprezentace bude vytvářena.

Tato práce popisuje běžně používané struktury projektů v různých IDE nástrojích a několik metod pro analýzu zdrojového kódu jazyka Java. Struktury projektů jsou popsány pro nejčastěji používané IDE nástroje a jeden z nejoblíbenějších nástrojů pro organizaci projektů. Dále tato práce popisuje syntaktickou analýzu, kterou je možné využít pro získání jednotlivých tříd a metod ze zdrojového kódu.

Ve druhé části této práce je pak popsán návrh a samotná implementace vytvořeného nástroje.

2 Struktura Java projektu v různých IDE

2.1 IDE projekt v jazyce Java

Základní konstrukční jednotkou v jazyce Java je třída [1]. Třídy obsahují zdrojové kódy a slouží především jako šablony, které se používají k vytváření instancí objektů a definování typů a metod datových objektů [2]. Každá třída má svůj vlastní soubor, jehož název je totožný s názvem třídy s příponou *.java*. Třídy jsou dále organizovány v balících, které jsou viditelné ve zdrojovém kódu. Balík je jmenný prostor, který organizuje soubor souvisejících tříd a rozhraní. Protože software napsaný v programovacím jazyce Java může být složen ze stovek nebo tisíců jednotlivých tříd, má smysl udržovat věci organizované umístěním souvisejících tříd a rozhraní do balíků. Balíky přímo odpovídají adresářové struktuře projektu v jazyce Java, název balíku se stává součástí názvu třídy [3]. V IDE se typicky pracuje v rámci projektu. Projekt tedy obsahuje zdrojové kódy a všechny související soubory potřebné pro sestavení programu. Vedle zdrojových souborů obsahuje projekt metadata o tom, co patří ke třídě, jak vytvořit a spustit projekt a další [4].

Většina projektů Java dnes rozděluje zdrojové a binární soubory do samostatných adresářů, typicky `src` a `bin`. Dále od sebe odděluje třídy a testovací třídy. Typicky je pro testovací třídy vytvořen adresář `test`, který může být umístěn v adresáři projektu společně s adresáři `src` a `bin` nebo je umístěn v adresáři `src` a třídy programu jsou pak umístěny v adresáři, který se typicky nazývá `main`.

2.2 Konvence pro pojmenování v jazyce Java

Konvence se dodržují při vývoji softwaru v jazyce Java z důvodu údržby a dobré čitelnosti kódu. Pro pojmenování prvků se většinou používá *CamelCase*.

CamelCase označuje způsob psaní víceslovných názvů tak, že jednotlivá slova od sebe nejsou oddělena mezerami, ale každé ze slov začíná velkým písmenem. CamelCase lze dále rozdělit na *Upper CamelCase* a *Lower CamelCase*. Lower CamelCase znamená, že první písmeno prvního slova názvu je malé. U Upper CamelCase je první písmeno prvního slova velké [5].

Pro pojmenování projektů neexistuje žádná konvence, ovšem běžně se používá Upper CamelCase nebo název psaný pouze malými písmeny se slovy oddělenými pomlčkou. Konvence pro pojmenování balíků je, že názvy balíků odpovídají názvu domény organizace, která balík poskytuje a jsou psané pouze malými písmeny. Příkladem názvu balíku je `cz.zcu.název_balíku`. Z toho názvu balíku vyplývá, že byl vytvořen např. v rámci nějakého projektu na ZČU. Pro pojmenování tříd se používá Upper CamelCase. Pro metody, atributy a proměnné se používá Lower CamelCase a pro konstanty se používají názvy psané pouze velkými písmeny s slovy oddělenými podtržítkem. [5].

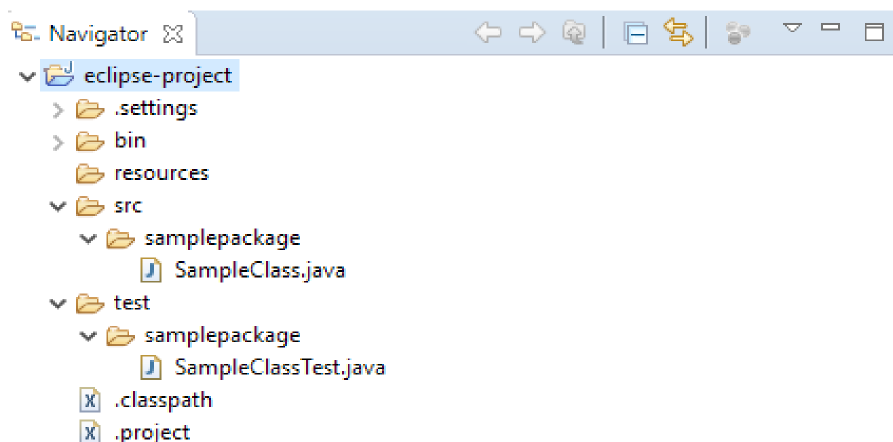
2.3 Struktura Java projektu v IDE Eclipse

Eclipse je integrované vývojové prostředí (IDE) používané v programování. Flexibilní návrh této platformy dovoluje rozšířit seznam podporovaných programovacích jazyků za pomoci pluginů, například o C++ nebo PHP. Standartní struktura projektu v IDE Eclipse je zobrazena na obrázku 1.

Standardní soubory a podadresáře, které se nacházejí v kořenovém adresáři Eclipse IDE projektu jsou:

- `eclipse-project/.classpath` – určuje umístění tříd a balíků definovaných uživatelem pro kompilaci projektu
- `eclipse-project/.project` – soubor, který popisuje projekt. Obsahuje informace jako název projektu a způsob sestavení projektu.

- `eclipse-project/.settings/` – složka `.settings` obsahuje soubory použité k nastavení trvalých vlastností na rozdíl od předvolby v IDE pro zadání nastavení specifických pro konkrétní projekt
- `eclipse-project/bin/` – obvykle obsahuje zkompilované soubory Například se v této složce nachází soubory `.class`, `.jar`, atd., které byly zkompilovány.
- `eclipse-project/src/` – `src` je typický název totoho adresáře ovšem může mít i jiný název, specifikovaný uživatelem, například `source`. Obsahuje balíky a třídy se zdrojovými kódy.
- `eclipse-project/test/` – obsahuje balíky se zdrojovými kódy testovacích tříd
- `eclipse-project/resources/` – typicky se používá název `resources`, ovšem název si může uživatel specifikovat sám. Obsahuje konfigurační soubory, soubory `.xml`, ikony pro grafické rozhraní atd.



Obrázek 1: Ukázka projektu v *Eclipse IDE*.

2.4 Struktura Java projektu v IDE NetBeans

NetBeans IDE je svobodné, zdarma distribuované integrované vývojové prostředí, které vlastní firma Oracle Corporation. Struktura projektu Java v IDE NetBeans je zobrazena na obrázku 2.

Obrázek 2: Ukázka projektu v NetBeans IDE.

Standardní soubory a podadresáře, které se nacházejí v kořenovém adresáři projektu Netbeans IDE jsou:

- `netbeans-project/build.xml` – obsahuje pokyny pro sestavení aplikace
- `netbeans-project/manifest.mf` – obsahuje informace, které budou uloženy do `.jar` souboru, který vznikne sestavením projektu
- `netbeans-project/build/` – obvykle obsahuje zkompilované soubory. Například se v této složce nachází soubory `.class` a `.jar`
- `netbeans-project/nbproject/` – do této složky IDE ukládá informace o projektu, složka obsahuje skript pro vytváření Ant a soubor vlastností, který řídí nastavení sestavení a spuštění a soubor `project.xml`, který mapuje Ant cíle na příkazy IDE
- `netbeans-project/src/` – `src` je typický název totoho adresáře ovšem může mít i jiný název, specifikovaný uživatelem, například `source`. Obsahuje jednotlivé balíky, které obsahují třídy se zdrojovými kódy

• `netbeans-project/test/` – obsahuje všechny testovací třídy

• `netbeans-project/resources/` – typicky se používá název `resources`, ovšem název si může uživatel specifikovat sám. Obsahuje konfigurační soubory a zdroje, které program potřebuje, například konfigurační soubory, *json* soubory, obrázky, atd.

2.5 Struktura Java projektu v IDE Intellij IDEA

IntelliJ IDEA je integrované vývojové prostředí Java (IDE) pro vývoj počítačového softwaru, vyvíjené společností *JetBrains*. Struktura projektu Java v IDE Intellij IDEA je zobrazena na obrázku 3.

Standardní soubory a podadresáře, které se nacházejí v kořenovém adresáři projektu v Intellij IDEA IDE jsou:

• `idea-project/.idea/` – obsahuje sadu souborů `.xml`, které definují všechna specifická nastavení pro daný projekt

•`idea-project/out/` – obsahuje všechny zkompilované zdrojové kódy

• `idea-project/idea-project.iml` – obsahuje informace o vývojovém modulu, popisuje závislosti a další nastavení

• `idea-project/src/main/` – obsahuje všechny soubory se zdrojovými kódy programu a zdroje

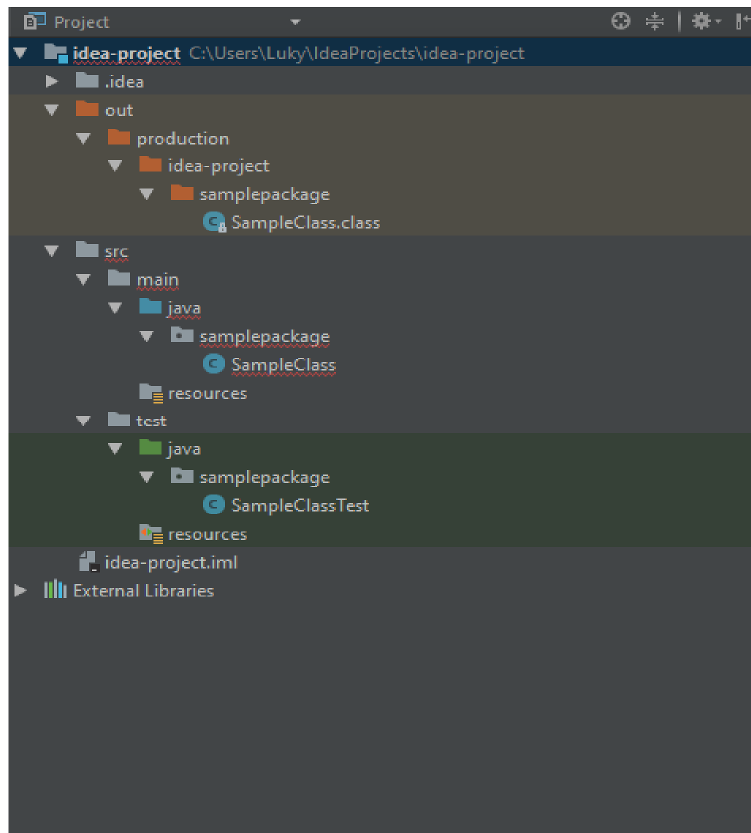
• `idea-project/src/test/` – obsahuje všechny soubory se zdrojovými kódy testů a zdroje

• `idea-project/src/main/java/` – obsahuje balíky a třídy se zdrojovými kódy programu

• `idea-project/src/main/resources/` – obsahuje konfigurační soubory, například soubory `.xml`, ikony pro grafické rozhraní atd..

• `idea-project/src/test/java/` – obsahuje zdrojové kódy testovacích tříd

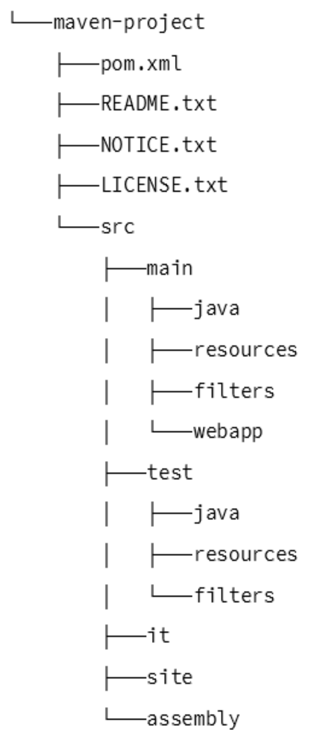
• `idea-project/src/test/resources/` – konfigurační soubory a další soubory používané testovacími třídami



Obrázek 3: Ukázka projektu v Intellij IDEA IDE.

2.6 Struktura Java projektu v Apache Maven

Apache Maven sice není IDE, ale je jedním z nejoblíbenějších nástrojů pro vytváření projektů Java. Kromě decentralizace závislostí a úložišť také prosazuje jednotné struktury adresářů projektů. Typický projekt Maven obsahuje soubor `pom.xml` a má adresářovou strukturu založenou na definovaných konvencích [6]. Struktura projektu Java v nástroji Maven je zobrazena na obrázku 4.



Obrázek 4: Ukázka struktury Java projektu v Apache Maven [6].

Standardní soubory a podadresáře, které se nacházejí v kořenovém adresáři projektu Apache Maven jsou:

- `maven-project/pom.xml` – definuje závislosti a moduly potřebné během životního cyklu sestavení projektu
- `maven-project/LICENSE.txt` – licenční informace o projektu
- `maven-project/README.txt` – shrnutí projektu
- `maven-project/NOTICE.txt` – informace o knihovnách třetích stran, které jsou v projektu použity.
- `maven-project/src/main/` – obsahuje zdrojový kód a zdroje, které se stanou součástí artefaktu

• `maven-project/src/test/` – obsahuje všechny soubory se zdrojovými kódy programu a zdroje

• `maven-project/src/it/` – vyhrazené pro integrační testy používané pluginem *Maven Failsafe*

• `maven-project/src/site/` – obsahuje dokumentaci *site* vytvořenou pomocí pluginu *Maven Site*

• `maven-project/src/assembly/` – konfigurace sestavení pro binární soubory

• `maven-project/src/main/java/` – obsahuje balíky a třídy se zdrojovými kódy programu

• `maven-project/src/main/resources/` – například konfigurační soubory pro prostředí a konfigurace `.xml`, ikony pro grafické rozhraní atd.

• `maven-project/src/main/webapp/` – používá se pro webové aplikace, obsahuje zdroje jako `.CSS`, `.HTML` soubory, šablony zobrazení a obrázky

• `maven-project/src/main/filters/` – obsahuje soubory, které vkládají hodnoty do konfiguračních vlastností ve složce `resources` během fáze sestavení.

• `maven-project/src/test/java/` – obsahuje všechny soubory se zdrojovými kódy testovacích tříd a zdroje

• `maven-project/src/test/resources/` – konfigurační soubory a další soubory používané testy

• `maven-project/src/test/filters/` – obsahuje soubory, které v průběhu testovací fáze vkládají hodnoty do konfiguračních vlastností ve složce `resources`

3 Syntaktická analýza

Překladače a kompilátory používají k analýze zdrojového kódu metody syntaktické analýzy. Tyto metody je tedy možné využít pro analýzu zdrojového kódu tříd jazyka Java a získání částí jejich zdrojového kódu, například atributů a metod.

Syntaktická analýza je proces analýzy posloupnosti formálních prvků. Používá se v informatice a lingvistice. Cílem toho procestu je analýza posloupnosti formálních prvků za účelem určení jejich gramatické struktury vůči předem dané formální gramatice. Během procesu syntaktické analýzy je vstupní text transformován na datovou strukturu. Touto strukturou je většinou syntaktický strom nebo derivační strom. Tyto struktury uchovávají uspořádání vstupních symbolů a jsou vhodné pro další zpracování [7, 8].

Vstupem do syntaktického analyzátoru je většinou text v nějakém programovacím jazyce, ale může to také být text v přirozeném jazyce nebo jiná strukturovaná textová data. Syntaktické analyzátory lze využít pro jednoduché funkce jako je například funkce scanf jazyka C, ale také pro kompilaci zdrojového kódu nějakého programovacího jazyka nebo pro analýzu HTML kódu. Důležitá část jednoduché syntaktické analýzy se provádí pomocí regulárních výrazů. Při syntaktické analýze pomocí regulárních výrazů je pomocí těchto výrazů definován regulární jazyk. Pro tento jazyk pak lze vygenerovat analyzátor, který umožňuje porovnávání řetězců se vzorem a extrakci dat. Regulární výrazy se také mohou použít pro lexikální analýzu. Výstup lexikální analýzy je pak použit syntaktickým analyzátorem [7, 8].

Použití syntaktických analyzátorů se liší podle vstupu. Analyzátor může být využit jako zařízení pro čtení souborů, které program využívá. Příkladem těchto souborů jsou soubory, které obsahují text ve formátu HTML nebo XML. Syntaktická analýza také představuje důležitou část zpracování zdrojového kódu programovacího jazyka. Využívá se při překladu a interpretaci. S datovými strukturami, které se získají pomocí syntaktické analýzy, pak

pracuje interpret nebo kompilátor. Kompilátor nebo interpret analyzuje zdrojový kód programovacího jazyka a vytvoří nějakou formu interní reprezentace [7, 8].

Syntaxe programovacích jazyků je obvykle specifikována pomocí bezkontextových gramatik. Pro bezkontextové gramatiky lze vytvořit rychlý a efektivní syntaktický analyzátor. Syntaktické analyzátory jsou obvykle vygenerované pomocí programu, který se nazývá parser generator, na základě gramatiky, která programovací jazyk specifikuje [7, 8].

3.1 Regulární výrazy

Regulární výraz je posloupnost znaků, které definují vzor. Tato posloupnost se skládá z literálů a speciálních znaků. Literály přesně definují znaky, které se v řetězci na dané pozici mohou vyskytovat. Speciální znaky slouží pro definici množin, počtu výskytů, atd. Tyto vzory obvykle využívají např. algoritmy pro vyhledávání nebo náhradu řetězců, validaci vstupů nebo je lze také využít pro lexikální analýzu. Regulární výrazy se dají použít pro analýzu některých jednodušších jazyků, to ovšem většinu programovacích jazyků vylučuje. Pokud gramatika jazyka obsahuje rekurzivní nebo vnořené elementy, nejedná se o regulární jazyk a nelze jej tedy analyzovat pomocí regulárních výrazů. Například HTML kód může obsahovat libovolný počet značek uprostřed jakékoliv značky a není tedy regulárním jazykem, tedy nelze jej analyzovat pouze za použití regulárních výrazů [9].

Regulární výrazy se také často používají pro definování gramatiky jazyka. Přesněji jsou využity pro definici pravidel lexeru nebo syntaktického analyzátoru. Například operátor „*“ udává, že určitý prvek se může vyskytovat neomezeně nebo také vůbec. Obvykle jsou regulární výrazy v gramatikách převedeny na konečné automaty, kvůli dosažení většího výkonu [9].

Pro syntaktickou analýzu programových jazyků jako je Java nelze využít jen regulární výrazy. Proto v této práci nebudou regulární výrazy detailně popsány.

Příklad regulárního výrazu je zobrazen na obrázku 5. Některé ze speciálních symbolů, používaných v regulárních výrazech jsou popsány v tabulce 1.

```
^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|
25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|
2[0-4]\d|25[0-5])$
```

Obrázek 5: Regulární výraz, který akceptuje pouze platnou IP adresu.

<table>
<tr><th>Symbol</th><th>Popis</th></tr>
<tr><td>\w</td><td>Akceptuje alfanumerické znaky a podtržítka</td></tr>
<tr><td>\W</td><td>Akceptuje vše kromě alfanumerických znaků a podtržítek</td></tr>
<tr><td>\d</td><td>Akceptuje pouze čísla</td></tr>
<tr><td>\D</td><td>Akceptuje vše kromě čísel</td></tr>
<tr><td>\s</td><td>Akceptuje bílé znaky</td></tr>
<tr><td>\S</td><td>Akceptuje vše kromě bílých znaků</td></tr>
<tr><td>[]</td><td>Akceptuje všechny znaky uvedené z závorkách</td></tr>
<tr><td>[^]</td><td>Akceptuje všechny znaky kromě znaků uvedených v závorkách</td></tr>
<tr><td>*</td><td>Akceptuje 0 nebo více výskytů předcházejícího znaku</td></tr>
<tr><td>+</td><td>Akceptuje 1 nebo více výskytů předcházejícího znaku</td></tr>
<tr><td>?</td><td>Akceptuje 0 nebo 1 výskyt předcházejícího znaku</td></tr>
<tr><td>{n, m}</td><td>Přesně definuje počet možných výskytů znaku. Počet výskytů znaku musí být alespoň n ale nesmí být více než m.</td></tr>
<tr><td>(abc)</td><td>Akceptuje pouze znaky uvedené v závorkách v daném pořadí</td></tr>
<tr><td>|</td><td>Akceptuje znak uvedený před tímto symbolem nebo znak uvedený za ním.</td></tr>
<tr><td>\</td><td>Escapování symbolů.</td></tr>
<tr><td>^</td><td>Značí začátek vstupu.</td></tr>
<tr><td>$</td><td>Značí konec vstupu.</td></tr>
</table>

Tabulka 1: Příklady některých specialních znaků používaných v regulárních výrazech.

3.2 Formální gramatika

Formální gramatika je soubor pravidel formování pro řetězce ve formálním jazyce. Pravidla popisují, jak vytvořit řetězce z abecedy jazyka, které jsou platné podle syntaxe jazyka. Gramatika nepopisuje význam řetězců. Formální gramatika je sada pravidel pro přepisování řetězců, spolu s počátečním symbolem, ze kterého musí začít přepisování. Proto je gramatika obvykle považována za generátor jazyka. Nicméně, to může také někdy být používáno jako východisko pro rozpoznávání. Při rozpoznávání je určeno, zda daný řetězec patří do jazyka nebo je gramaticky nesprávný a do jazyka nepatří [7, 8, 9].

Pro každou gramatiku obecně existuje nekonečný počet řetězců, které lze pomocí této gramatiky vytvořit. To znamená, že gramatika s konečnou velikostí může generovat nekonečný počet řetězců [7, 8, 9].

Jak již bylo zmíněno, programovací jazyky jsou obvykle specifikovány pomocí bezkontextových gramatik. Bezkontextové gramatiky ovšem nemusí být schopny vyjádřit vše, co jazyk vyžaduje. Při použití bezkontextové gramatiky není způsob, jak si zapamatovat předchozí výskyt prvku. Nelze tedy říci, jestli například proměnná již byla deklarována. Sofistikovanější gramatiky, které toto dokáží, zase nemohou být efektivně syntakticky analyzovány. Obvykle se tedy používá bezkontextový syntaktický analyzátor v kombinaci s dalšími nástroji, obvykle *tabulkou symbolů* [7, 8, 9].

Tabulka symbolů je datová struktura, která se používá pro uložení všech identifikátorů nalezených ve zdrojovém kódu programu [10].

Příklad gramatiky je zobrazen na obrázku 6.

Formální gramatika je definována jako uspořádaná čtveřice G=<N,T,S,P>, kde

- N je konečná neprázdná množina neterminálních symbolů,

- T je konečná neprázdná množina terminálních symbolů,
- S je počáteční symbol,
- P je konečná neprázdnámnožina přepisovacích pravidel ve tvaru $\alpha \rightarrow \beta$. Na levé straně přepisovacího pravidla musí být alespoň jeden neterminální symbol. Na druhé straně přepisovacího pravidla může být cokoliv, i prázdný řetězec [11, 12].

Konvence značení ve formálních gramatikách je následující [12] .

- Neterminální symboly značíme velkými písmeny např. X, Y
- Terminální symboly značíme malými písmeny např. a,b
- Řetězce značíme řeckými písmeny např. α,β,γ

Gramatiky se dále klasifikují na čtyři typy podle tvaru přepisovacích pravidel. Tato klasifikace se nazývá *Chomského hierarchie* [11, 12].

Typ 0 - Všechny formální gramatiky. Nejobecnější gramatiky, které v sobě zahrnují všechny formální gramatiky. Gramatiky typu 0 generují jazyky, které jsou rozpoznatelné turingovým strojem. Tvar pravidel gramatiky typu 0 je $\alpha \rightarrow \beta$.

Typ 1 - Kontextové gramatiky. Přepisovací pravidla kontextových gramatik mají tvar $\alpha X\beta \rightarrow \alpha\gamma\beta$. To znamená, že levé i pravé strany pravidel mohou být obklopeny řetězci, které se skládají z terminálních i neterminálních symbolů

Typ 2 - Bezkontextové gramatiky. Přepisovací pravidla bezkontextových gramatik mají tvar $X \rightarrow \gamma$. To znamená, že levé strany pravidel se skládají pouze z jediného neterminálního symbolu a mohou být přepsána na řetězec, který se skládá z terminálních i neterminálních symbolů.

Typ 3 - Regulární gramatiky. Přepisovací pravidla regulárních gramatik mají tvar $X \rightarrow Y\alpha$, $X \rightarrow \alpha Y$ *nebo* $X \rightarrow \alpha$. Gramatika typu 3 se podle přepisovacích pravidel dále dělí na levou a pravou. Levá gramatika smí obsahovat pouze pravidla $X \rightarrow Y\alpha$ a $X \rightarrow \alpha$. Pravá gramatika smí obsahovat pouze pravidla $X \rightarrow \alpha Y$ a $X \rightarrow \alpha$. Pokud gramatika obsahuje pravidla obou gramatik stále se jedná o gramatiku typu 3, ale už se neurčuje, jestli je pravá nebo levá.

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{UT} \\
\mathrm{U} &\rightarrow \mathrm{aa} \mid \mathrm{ab} \mid \mathrm{ba} \mid \mathrm{bb} \\
\mathrm{T} &\rightarrow \mathrm{aTa} \mid \mathrm{bTb} \mid \mathrm{V} \\
\mathrm{V} &\rightarrow \mathrm{aa} \mid \mathrm{ab} \mid \mathrm{ba} \mid \mathrm{bb}
\end{aligned}
$$

Obrázek 6: Ukázka jednoduché formalní gramatiky [13].

4.3 Abstraktní syntaktický strom a derivační strom

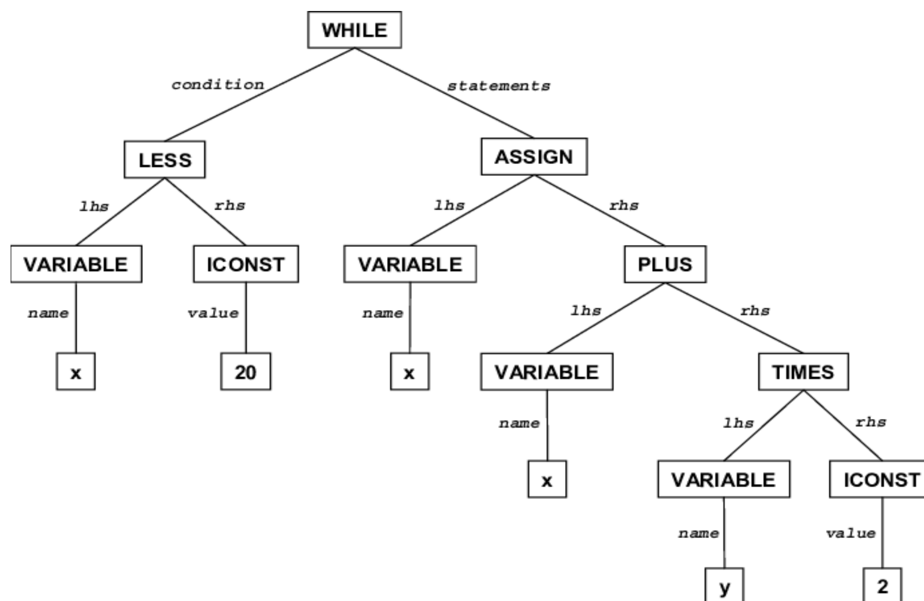
Abstraktní syntaktický strom i derivační strom reprezentují celý zdrojový kód. Všechny vnitřní uzly těchto stromů tvoří další podstromy, tyto podstromy představují menší a menší části kódu, dokud se ve stromě neobjeví jednotlivé terminální symboly jako listy stromu. Rozdíl mezi těmito stromy spočívá v úrovni abstrakce. Derivační strom může obsahovat všechny symboly, které se objevily v programu a případně sadu odvozovacích pravidel. Abstraktní syntaktický místo toho je obecnější verze stromu analýzy, ve kterém jsou zachovány pouze informace relevantní pro pochopení kódu. Některé informace mohou chybět jak v abstraktním syntaktickém stromu, tak v derivačním stromu. Například komentáře a symboly seskupení (závorky) nejsou obvykle ve stromu reprezentovány. Komentáře jsou pro program zbytečné a symboly seskupení jsou implicitně definovány strukturou stromu [9].

4.3.1 Abstraktní syntaktický strom

Abstraktní syntaktický strom je stromová reprezentace abstraktní syntaktické struktury zdrojového kódu. Vnitřní uzly tohoto stromu představují operátory a listy představují jednotlivé operandy. Tento strom je označován jako abstraktní, protože nepředstavuje konkrétní implementaci, ale spíše představuje strukturální a obsahové detaily.

Abstraktní syntaktické stromy se využívají hlavně pro překlad a optimalizace. Jako příklad optimalizace si lze představit strom, který reprezentuje logický součet. Pokud je jedna z větví tohoto stromu vždy pravdivá, není třeba aby překladač vyhodnocoval i druhou větev. Do abstraktního syntaktického stromu můžou být přidávány další informace i po jeho vytvoření pomocí následného zpracování, např. kontextové analýzy. Abstraktní syntaktické stromy se také používájí v programové analýze a programových transformačních systémech [9].

Na obrázku 7 je zobrazen abstrakní syntaktický strom pro cyklus while v programovacím jazyce *Modelica*.

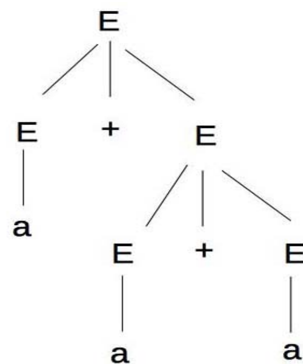


Obrázek 7: Příklad abstraktního syntaktického strom pro cyklus while [14].

4.3.2 Derivační strom

Derivační strom je stromová reprezentace syntaktické struktury řetězce, který je definovám podle formální gramatiky. V derivačním stromě jsou vnitřní uzly označeny jako neterminální symboly gramatiky, zatímco listy stromu jsou označeny jako terminální symboly. Tyto stromy jsou běžně vytvářeny během překladu a kompilace zdrojového kódu. Derivační stromy mohou také být využity pro generování nebo analýzu vět v přirozeném jazyce. Pokud je formální gramatika víceznačná, může existovat více derivačních stromů pro daný řetězec. Jednou z dalších vlastností derivačního stromu je, že procházením stromu do šířky lze zpět získat vstupní řetězec [9].

Ukázka jednoduchého derivačního stromu je zobrazena na obrázku 8.



Obrázek 8: Ukázka derivačního stromu. Uzly stromu představují neterminální symboly. Listy reprezentují terminální symboly [15].

4.4 Metody syntaktické analýzy

Existují dvě základní metody syntaktické analýzy. Syntaktická analýza shora dolů a syntaktická analýza zdola nahoru [16].

Metoda syntaktické analýzy shora dolů pracuje tak, že analyzuje vstup a začne vytvářet syntaktický strom z kořenového uzlu. Pomocí přepisovacích

pravidel gramatiky pak postupuje směrem dolů k listům stromu. Jednou z metod syntaktické analýzy shora dolů je rekurzivní sestup [17].
Syntaktická analýza zdola nahoru funguje tak, že začne od řetězce a používá odvozovací pravidla gramatiky obráceně tak, aby bylo dosaženo počátečního symbolu. Zpracování zdola nahoru tedy začíná od listovů stromu a pracuje směrem nahoru, dokud nedosáhne kořenového uzlu [18].

Překladače programovacích jazyků založené na syntaktické analýze zdola nahoru používají metodu, která identifikuje nejdříve terminální symboly. Poté je kombinuje postupně tak, že produkuje neterminální symboly. Výsledek zpracování může být použit k vytvoření derivačního stromu ze zdrojového kódu programu tak, že může být zkompilován, aby vznikl program, který může provádět procesor nebo který lze interpretovat virtuálním procesorem [18].

4.4.1 Rekurzivní sestup (shora dolů)

Rekuzivní sestup je metoda syntaktické analýzy shora dolů, která vytváří derivační strom od kořene stromu a vstupní řetězec čte zleva doprava. Tato metoda využívá sadu vzájemně rekurzivních procedur, kde každá z těchto procedur zpravidla implementuje jedno z odvozovacích pravidel gramatiky, která generuje vstupní řetězec. Výsledná struktura programu zrcadlí strukturu gramatiky, kterou má rozpoznávat. Metoda rekurzivně analyzuje vstupní řetězec za účelem vytvoření derivačního stromu. Rekurzivní sestup využívá k analýze vstupního řetězce backtracking. Existuje i metoda na základě rekurzivního sestupu, která backtracking nevyužívá. Tato metoda se nazývá prediktivní syntaktická analýza [17].

4.4.2 Backtracking (shora dolů)

Metoda rekurzivního sestupu využívající backtracking funguje tak, že začíná v kořenovém uzlu stromu, tedy počátečním symbolu. Tento symbol pak

porovnává s odvozovacími pravidly. Metoda vybere první pravidlo a počáteční symbol podle tohoto pravidla přepíše. Pokud se začátek výsledného řetězce rovná vstupnímu řetězci, ale výsledný řetězec není složený pouze z terminálů, posune se ve stromu dál a použije další pravidlo. Pokud už je výsledný řetězec složen pouze z terminálních symbolů a řetězce se rovnají metoda končí. Pokud se však výsledný řetězec po odvození nerovná vstupnímu řetězci, metoda se vrátí zpět a zkouší další pravidla, dokud nenalezne pravidlo jehož výsledkem je řetězec, který se rovná vstupnímu řetězci [17].

Pokud tato metoda pracuje s bezkontextovou gramatikou, není zaručené, že někdy skončí. Ovšem i v případě, že tato metoda úspěšně dokončí analýzu řetězce, může pro zpracování výsledku vyžadovat exponenciální čas [17].

4.4.3 Prediktivní syntaktická analýza (shora dolů)

Prediktivní syntaktická analýza je rekurzivní metoda na základě rekurzivního sestupu. Tato metoda dokáže predikovat, které odvozovací pravidlo bude použito k nahrazení řetězce. Výhodou této metody je, že nevyužívá backtracking. Ke zpracování řetězce bez potřeby využít backtracking metoda využívá ukazatel na následující znak v řetězci. Prediktivní syntaktická analýza je možná pouze pro gramatiky typu LL(k). Gtamatiky typu LL(k) jsou gramatiky, pro které existuje nějaké kladné celé číslo k, které umožňuje, aby analyzátor s rekurzivním sestupem rozhodoval, jaké odvozovací pravidlo se má použít, na základě prohlédnutí dalších k symbolů na vstupu. Pro analýzu vstupního řetězce a vygenerování derivačního stromu tato metoda používá zásobník a odvozovací tabulku. Zásobník i vstupní řetězec obsahují ukončovací symbol, který indikuje, že zásobník je prázdný a celý vstupní řetězec byl zpracován. Prediktivní syntaktický analyzátor analyzuje vstupní řetězec v lineárním čase [17].

4.4.4 Metoda přesun-redukce (zdola nahoru)

Metoda přesun-redukce používá dva unikátní kroky pro syntaktickou analýzu shora dolů. Tyto kroky se nazývají přesun a redukce. Analyzátor, který k analýze vstupního řetězce používá metodu přesun-redukce používá zásobník, do kterého ukládá symboly, na které později aplikuje redukci. Symboly ze vstupu jsou postupně přesouvány do zásobníku, to je nazýváno přesun. Pokud se prefix symbolů v zásobníku rovná právě straně některého z odvozovacích pravidlech, analyzátor je nahradí levou stranou tohoto pravidla. Tento krok se nazávý redukce. Tyto kroky se provádí tak dlouho, dokud analýza neskončí přijetím vstupu nebo selháním [18].

5 Analýza problému

Cílem této práce je vytvořit aplikaci, která vytvoří objektovou reprezentaci uživatelem daného Java projektu až do úrovně jednotlivých metod a výsledek uloží do databáze.

5.1 Specifikace požadavků

5.1.1 Vstupní data

Uživatelem vybraný adresář projektu bude nutné prohledat a získat z něj všechny složky a soubory. Po získání všech souborů a složek v adresáři bude nutné vybrat soubory, které obsahují zdrojový kód a tento kód zpracovat pomocí některé z metod syntaktické analýzy a získat jednotlivé třídy a jejich metody včetně komentářů. Zdrojové kódy tříd v daném projektu mohou obsahovat chyby nemusí být přeložitelné.

5.1.2 Zpracování dat

Pro celý projekt bude třeba vytvořit jeho objektovou reprezentaci. Aplikace nebude ukádat zdrojový kód celé třídy, ale rozdělí ho na deklarace balíku, importy, hlavičky tříd, atributy tříd, hlavičky metod a jednotlivá těla metod. Soubory, které neobsahují zdrojový kód bude nutno také uložit, tyto soubory se budou ukládat jako binární data.

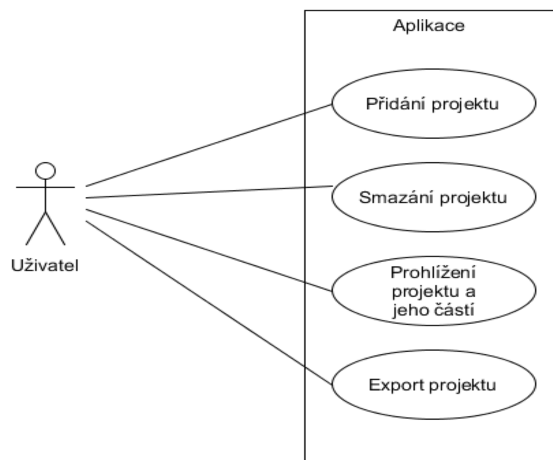
5.1.3 Výstup aplikace

Aplikace bude poskytovat uživatelské rozhraní, ve kterém zobrazí uložený projekt. Bude zobrazovat i jednotlivé části uloženého projektu, například třídy, metody tříd, atd. a jejich zdrojový kód nebo v případě souborů s textovým

obsahem zobrazí tento text. Aplikace také umožní exportovat projekt z databáze. Vyexportované soubory se zdrojovým kódem se znovu složí z uložených dat. Zdrojový kód tříd tedy bude zpět složen z uložených atributů, metod a vnitřních tříd.

5.2 Případy užití aplikace

Diagram případů užití aplikace je zobrazen na obrázku 9.



Obrázek 9. Diagram případů užití aplikace.

5.2.1 Přidání projektu

Uživatel bude moci prostředníctvím grafického uživatelského rozhraní přidat adresář projektu ze souborového sytému do databáze, kterou bude aplikace používat.

5.2.2 Smazání projektu

Uživatel bude moci prostřednictvím grafického uživatelského rozhraní smazat vybraný projekt z databáze projektů. Aplikace povolí pouze odstranění celého projektu, jednotlivé části projektu nebude možné odstranit.

5.2.3 Prohlížení projektu a jeho částí

Po přidání projektů si uživatel bude moci prostřednictvím grafického uživatelského rozhraní zobrazit vybraný projekt a jeho jednotlivé části. Uživateli se také zobrazí další informace o vybraném porjektu nebo některé z jeho částí, například zdrojový kód třídy, zdrojový kód metody, atd.

5.2.4 Export projektu

Uživatel bude moci vybraný projekt exportovat z databáze do zvoleného adresáře v souborovém systému.

5.3 Načítání souborů

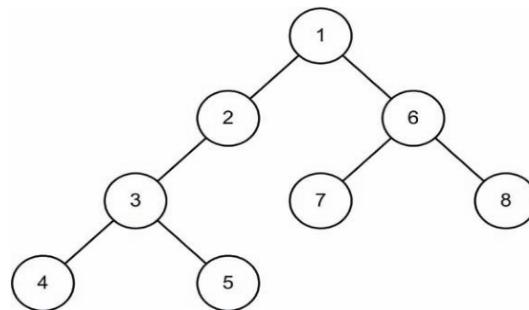
Uživatel předá aplikaci cestu k adresáři Java projektu. Cestu bude uživatlel vybírat prostřednictvím dialogového okna. Adresář je třeba prohledat a nalézt všechny soubory, složky a soubory obsahující zdrojový kód. Struktura adresářů v souborovém systému je stromová, pro prohledání adresáře tedy lze použít například algoritmus *DFS* nebo *BFS*.

5.3.1 DFS

DFS (deep first search) je rekurzivní algoritmus. Algoritmus označí počáteční vrchol jako navštívený a přesune se do jednoho z nenavštívených sousedních vrcholů, který opět označí jako navštívený a přesune se do jeho sousedního vrcholu. Takto algoritmus pokračuje, dokud nenarazí na vrchol, ze kterého již nevede žádná hrana, nebo všechny jeho sousední vrcholy jsou již označené jako navštívené, tento vrchol označí jako dokončený. Algoritmus se vrátí do předchozího vrcholu a přesune se dodalšího souseda, pokud nějaký je. Algoritmus končí, když se vrátí do počátečního vrcholu a již nejsou žádné

nedokončené sousední vrcholy nebo dokud nejsou prozkoumány všechny dosažitelné vrcholy [19].

Pořadí zprácování uzlů algoritmem DFS je zobrazeno na obrázku 10 .

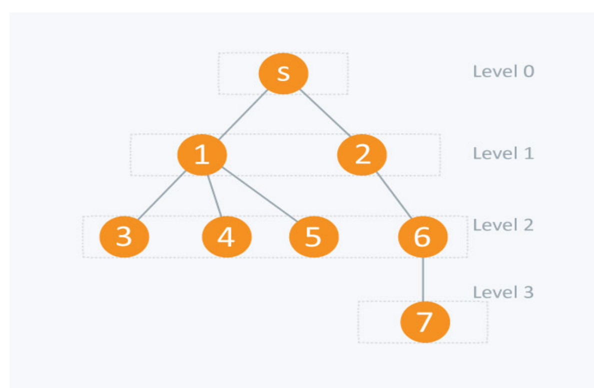


Obrázek 10: Pořadí, ve kterém algoritmus DFS zpracuje uzly grafu [19].

5.3.2 BFS

BFS (breadth first search) je algoritmus využívající frontu. Algoritmus vloží do fronty všechny sousedy počátečního vrcholu, označí je jako navštívené a počáteční vrchol označí jako dokončený. Poté z fronty vybere první vrchol, označí ho jako dokončený a opět do fronty vloží všechny jeho sousedy, které ještě nebyly navštívené. Takto algoritmus pokračuje dokud nejsou všechny vrcholy označeny jako dokončené, nebo dokud nedosáhne koncového vrcholu [20].

Pořadí zprácování uzlů algoritmem BFS je zobrazeno na obrázku 11.



Obrázek 11: Pořadí, ve kterém algoritmus BFS zpracuje uzly grafu [20].

5.4 Analýza zdrojového kódu

Aplikace musí zpracovat všechny soubory se zdrojovým kódem z uživatelem zadaného adresáře až na úroveň jednotlivých metod. Pro každý soubor se zdrojovým kódem je nutné získat deklaraci balíku, importy, všechny třídy, které jsou v tomto souboru deklarované a jejich metody a vnitřní třídy.
Pro analýzu zdrojového kódu je možné využít metody, které jsou popsány v kapitole 4.4.

5.5 Existující nástroje pro analýzu zdrojového kódu

Existuje množství již vytvořených nástrojů a knihoven, které provádí syntaktickou analýzu zdrojového kódu. Mezi tyto nástroje patří například *ANLTR a JavaParser.*

5.5.1 ANTLR

ANTLR (ANother Tool for Language Recognition) je tzv. parser generator, který v roce 1992 vytvořil Terence Parr. Tento nástroj je schopný z jemu předané formální gramatiky jazyka nebo strukturovaného textu vygenerovat syntaktický analyzátor, který je schopný generovat a procházet derivační stromy. ANTLR je Vytvořený syntaktický analyzátor se následně dá využít například pro čtení, zpracování nebo i transformaci zdrojového kódu napsaného v jednom jazyce do zdrojového kódu jiného jazyka [21].

5.5.2 JavaParser

JavaParser je knihovna vytvořená pro programovací jazyk Java. Tato knihovna umožňuje vytvořit objektovou reprezentaci zdrojového kódu, kterou autoři označují jako abstraktní syntaktický strom. Tento strom je pak možno procházet a identifikovat úseky zdrojového kódu, například třídy a metody, které

programátora zajímají. Knihovna také umožňuje další manipulaci se zdrojovým kódem. Tento kód lze pak znovu zapsat do souboru, knihovna tedy umožňuje i generování kódu.

JavaParser také provádí pouze syntaktickou analýzu zdrojového kódu. Tedy není schopný nalézt sémantické chyby, kdy je například proměnná použita v kódu aniž by byla předtím deklarována [22] .

5.6 Objektová reprezentace dat

Pro získaná data bude nutné vytvořit objektovou reprezentaci. Pro složky a soubory, které neobsahují zdrojový kód, bude stačit vytvořit třídu, která uloží název a jejich obsah. Pro složky bude vhodné vytvořit tři seznamy, jeden seznam bude obsahovat všechny složky, které se v této složce nachází. Další seznam bude obsahovat všechny soubory, které neobsahují zdrojový kód a poslední seznam pro soubory obsahující zdrojový kód. Pro soubory bez zdrojového kódu se uloží pouze jejich obsah v binární podobě.

Pro třídy by byla vhodná taková reprezentace, která uloží komentář třídy, anotace, hlavičku třídy, seznam konstruktorů, seznam metod, které třída obsahuje a seznam tříd, které třída obsahuje. Objekt pro reprezentaci metod bude obsahovat komentář, anotace, hlavičku metody a tělo metody. Pro uložení konstruktorů je možné vytvořit samostatnou třídu nebo lze využít podobnosti konstruktoru a metod a uložit konstruktor jako metodu. Pro jejich následné rozlišení by mohl být využit výčtový typ.

Pro reprezentaci třídy, rozhraní a výčtového typu je také možné vytvořit samostatné třídy nebo použít jeden objekt, do kterého bude možné uložit všechny a jednotlivé typy od sebe odlišit výčtovým typem. Výsledná reprezentace dat by měla tvořit stromovou strukturu, kde kořenem stromu je adresář projektu a listy stromu budou tvořit soubory, které neobsahují zdrojový kód, metody, popř. třídy, pokud nebudou obsahovat žádné metody.

5.7 Uložení dat do databáze

Získanou objektovou reprezentaci dat bude zapotřebí uložit do databáze. Pro uložení dat bude důležité vhodně navrhnout databázi. Složky, soubory, třídy a metody budou reprezentované samostatnými entitami, tedy pro všechny bude vytvořena tabulka. Jednotlivé sloupce tabulky budou obsahovat údaje, které umožní projekt prohlížet ve stromové struktuře a zároveň později z databáze exportovat.

Problémem může být stromová struktura dat. Například adresář může obsahovat libovolný počet podadresářů, které také mohou obsahovat další podadresáře. V případě tříd exisuje stejný problém, protože třídy mohou obsahovat vnořené třídy. Soubory se zdrojovými kódy, ostatní soubory a metody nemohou obsahovat další potomky, vždy musí být umístěny v nějakém adresáři nebo třídě. Tento předpoklad vychází ze vstupních dat, čímž je adresář projektu. V tabulkách tedy budou mít sloupec, ve kterém bude uloženo *id* prvku, který je jim nadřazený. Mezi těmito prvky a jim nadřazenými prvky se tedy bude jednat o vazby *1* ku *n*. V případě adresářů a tříd by data mohla být uložena v tabulkách tak, že v tabulkách bude sloupec, který bude obsahovat *id* předka. Tabulka tedy bude mít vazbu *1* ku *n* sama se sebou. Při potřebě získat celý projekt z databáze tedy bude nutné nejdříve získat kořenový adresář projektu. Kořenový adresář projektu lze od ostatních odlišit tak, že hodnota ve sloupci, který obsahuje id předka, bude nastavena na null.

Pro metody a třídy není v Javě nastaveno žádné omezení počtu znaků. Jejich zdrojový kód je možné uložit jako datový typ *varchar* nebo datový typ *text* nebo *clob*. U datového typu varchar je zapotřebí uvést maximální velikost ukládaného řetězce. V databázích *Oracle* je datový typ varchar omezen maximální velikostí *4000* bajtů a databázích *Mysql* je tato velikost omezena maximálně na *65535* bajtů. Ovšem v Mysql platí i omezení na maximální velikost řádku tabulky a to *65535* bajtů. Datový typ varchat ukládá řetězec

přímo do tabulky, tudíž nelze do jednoho řádku tabulky uložit dva řetězce s velikostí *65535* bajtů jako datový typ varchar [23, 24].
Druhou možností pro uložení těchto dat je v Mysql datový typ text nebo v databázích Oracle datový typ clob. U datového těchto datových typů není třeba specifikovat maximální velikost ukládaného řetězce a jejich maximální velikost je *4GB* [23, 24].

5.8 Existující nástroje pro práci s databází

Pro programovací jazyk Java existuje několik knihoven pro práci s databázemi. Mezi tyto knihovny patří například *JDBC* nebo *Hibernate*.

5.8.1 JDBC

JDBC (Java Database Connectivity) je standardní jednotné rozhraní pro programovací jazyk Java, umožňující připojení aplikace k relačním databázím. Knihovna JDBC obsahuje rozhraní pro každý z níže uvedených úkolů, které jsou běžně spojeny s použitím databáze [25].

- Připojení k databázi.
- Vytváření příkazů SQL nebo MySQL.
- Spuštění dotazů SQL nebo MySQL v databázi.
- Zobrazení a úprava výsledných záznamů.

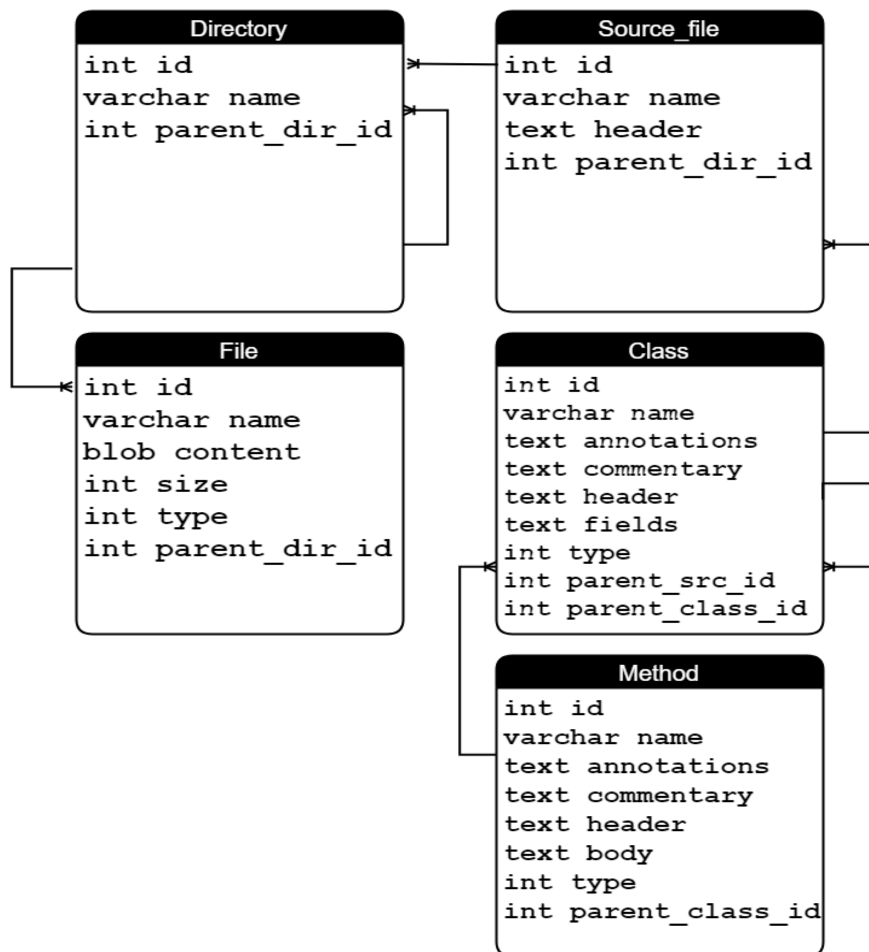
5.8.2 Hibernate

Hibernate je framework pro programovací jazyk Java, který umožňujeobjektově relační mapování objektů. Hibernate mapuje objekty na tabulky databáze a jednotlivé atributy těchto objektů mapuje na sloupce těchto tabulek. K tomu

používá mapovací soubory nebo anotace, které jsou umístěné přímo ve zdrojovém kódu. Kromě mapování objektů Hibernate také umožňuje vytvářet dotazy na databázi pomocí *HQL* (Hibernate query language), který je odvozen přímo z SQL [26].

5.9 Návrh databáze

Návrh relačního modelu databáze je zobrazen na obrázku 12.



Obrázek 12: Relační model navržené databáze.

5.10 Export projektu

Aplikace musí být schopna exportovat projekt uložený v databázi do uživatelem zvoleného adresáře na disku. To lze provést například tak, že bude z databáse z tabulky pro adresář vybrán uživatelem zvolený adresář a získáno jeho id. Pomocí tohoto id pak z tabulek pro adresáře, soubory se zdrojovým kódem a ostatní soubory získáme jednotlivé potomky tohoto adresáře. Pro potomky, které jsou adresáře lze tento postup opakovat. Potomci, které představují soubory bez zdrojových kódu, již žádné potomky nemají. Pro potomky představující třídy bude použit podobný postup, při kterém se získají všechny třídy, které mohou být v souboru deklarovány a v případě tříd mohou být takto získány jednotlivé metody. Metody také žádně další potomky nemají. Celý tento postup se bude opakovat tak dlouho, dokud nebudou získány všechny části projektu, to znamená, že už nebude možné přidat žádné další potomky.

Z takto získané objektové reprezentace projektu pak bude vytvořen projekt v uživatelem zvoleném adresáři. Objektová reprezentace projektu bude mít stromovou struktur, projekt tedy lze vytvořit tak, že pomocí DFS nebo BFS algoritmu se bude strom procházet a pro každý prvek bude vytvořen odpovídající soubor nebo adresář. Do jednotlivých souborů pak bude stačit zapsat data, která budou uložena v jednotlivých objektových reprezentací daných souborů.

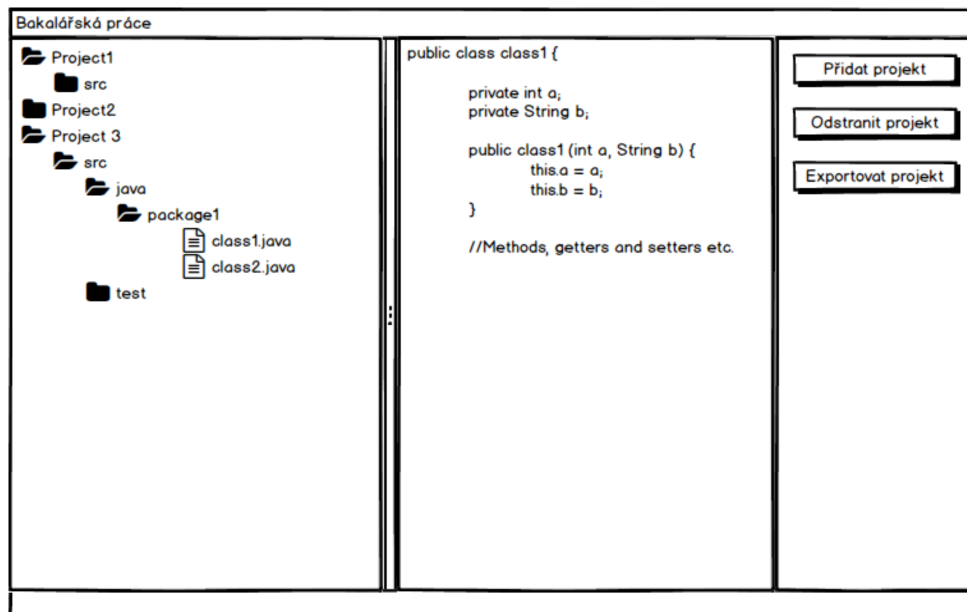
5.11 Grafické uživatelské rozhraní

Zpracované projekty je nutné nějakým vhodným způsobem zobrazit v grafickém uživatelském prostředí. Vzhledem ke stromové struktuře projektu se jako nejvhodnější způsob zobrazení nabízí zobrazení ve stromu. Kořenem stromu bude databáze, ve které jsou data uloženy. Uzly, tvořící potomky kořene, budou představovat jednotlivé projekty. Další uzly budou reprezentovat podadresáře v jednotlivých projektech. Listy stromu budou soubory, které

neobsahují zdrojový kód, prázdné adresáře, metody nebo třídy, které neobsahují žádné metody. Dále by bylo vhodné zobrazit kód souborů se zdrojovým kódem, tříd a metod, pokud je uživatel ve stromu vybere. Hlavní částí okna aplikace by tedy mělo být zobrazení stromu a textová oblast pro zobrazení obsahu vybraného prvku stromu. Pro vybrané metody se zobrazí hlavička a tělo metody včetně všech komentářů a anotací, které k metodě patří. Pro třídy se zobrazí hlavička třídy, tělo třídy včetně všech metod, komentářů a anotací, které ke třídě patří. Při zobrazení souboru se zdrojovým kódem se zobrazí deklarace balíku, importy a všechny třídy, které jsou v souboru deklarovány. Pro ovládání aplikace bude v gui vytvořeno několik tlačítek. V době návrhu to zatím jsou tlačítka pro přidání projektu, odebrání projektu a export projektu.

5.12 Návrh grafického uživatelského rozhraní

Grafické rozhraní aplikace se bude skládat pouze z jednoho hlavního okna. Aplikace bude dále zobrazovat různá vyskakovací okna jako reakce na akce uživatele. Návrh vzhledu hlavního okna aplikace je zobrazen na obrázku 13.



Obrázek 13: Návrh hlavního okna aplikce

6 Popis implementace

Aplikace byla vytvořena v programovacím jazyce Java ve verzi *1.8*. Třídy aplikace jsou rozděleny do čtyř balíků. Jsou to balíky `app`, `gui`, `data` a `utils`. V balíku `app` jsou umístěny třídy, které slouží pro spuštění programu, načítání dat ze souborů a zápisování dat do souborů, získávání jednotlivých částí zdrojového kódu ze zpracovávaných tříd a připojení k databázi a práci s databází. V balíku `gui` je umístěna třída, která reprezentuje hlavní okno aplikace. V balíku `data` se nacházejí třídy, které slouží pro reprezentaci dat získaných ze zdrojových kódů tříd. V balíku `utils` je pouza třída, která obsahuje konstanty.

6.1 Použité nástroje a knihovny

Projekt byl vytvořen v IDE Eclipse s použitím nástroje pro správu projektů Apache Maven. Grafické uživatelské rozhraní aplikace bylo vytvořeno pomocí frameworku *JavaFX*. Důvodem pro výběr frameworku *JavaFX* je, že *JavaFX* je dnes používaným standardem pro tvorbu uživatelských rozhraní v jazyce Java.

Pro analýzu zdrojového kódu byla použita knihovna *JavaParser*. Důvodem pro použití této bylo, že na rozdíl od nástroje *ANTLR* lze knihovnu použít přímo v programu bez nutnosti generování tříd syntaktického analyzátoru na základě gramatiky jako je tomu u nástoje *ANTLR*.

Pro spojení s databází a operace s databází byl vybrán framework *Hibernate*. Důvodem pro výběr tohoto frameworku bylo to, že framework velice usnadňuje ukládání dat do databáze a získávání dat z databáze díky svému objektovému mapování. Toto je výhodou zejména v případě, kdy data mají stromovou strukturu. Díky *Hibernate* je možné pouze předat objekt, který reprezentuje kořenový adresář projektu a *Hibernate* už všechny další podadresáře a soubory vloží do databáze sám. Není tedy třeba stromovou strukturu projektu procházet a každý soubor nebo adresář vkládat do databáze. Mezi další výhody tohoto

framworku patří například to, že Hibernate dokáže sám vytvářet tabulky databáze na základě vytvořeného mapovacího souboru nebo anotací umístěných ve zdrojovém kódu. Není tedy třeba psát skript pro vytvoření příslušných tabulek v databázi, který by v případě změn bylo nutné upravovat.

Pro uložení dat byla zvolena databáze *Mysql*. Důvodem pro její použití je snadná instalace a jednoduché používání prostřednictvím programu *XAMPP*.

6.2 Popis tříd aplikace

6.2.1 Třída `App`, `FileIO`, `DatabaseConnection` a `Parser`

Tyto třídy jsou umístěny v balíku `app` a tvoří aplikační část programu. Třída `App` je hlavní třídou programu a její činnost spočívá pouze ve spuštění gui aplikace.

Třída `FileIO` slouží pro načítání dat ze souborů, vytváření souborů a adresářů a zápisu dat do souborů. Tato třída načte uživatelem vybraný adresář projektu, prochází všechny soubory a podadresáře, a s pomocí třídy `Parser` jsou již během načítání souborů se zdrojovým kódem získávány jednotlivé části zdrojového kódu. Poslední činnost třídy je, že vytvoří a vrátí objektovou reprezentaci adresáře projektu.

Třída `DatabaseConnection` zodpovídá za připojení k databázi a operace s databází. Obsahuje metody, které navazují spojení s databází, ukládají projekt do databáze, odstraňují data projektů z databáze, získají daný projekt z databáze nebo získají všechny projekty z databáze.

6.2.2 Třída `AppWindow`

Tato třída je umístěna v balíku `gui`. Třída reprezentuje hlavní okno aplikace. Sllouží pro zobrazování projektů ve stromu, zobrazování obsahu souborů nebo

informací o souborech v textové oblasti a umožňuje uživateli ovládat aplikaci prostřednictvím tlačítek. Obsahuje metody, které vytváří jednotlivé části hlavního okna aplikace, metody, které slouží jako obsluha tlačítek v okně. V poslední řadě obsahuje podpůrné metody pro řazení projektů ve stromu a získání popisu momentálně vybraného prvku, který se zobrazí v textové oblasti.

6.2.3 Třídy `ClassRep`, `MethodRep`, `FileSource`, `FileOther`, `Directory` a `TreeElement`

Tyto třídy jsou umístěny v balíku `data` a slouží jako objektová reprezentace projektu.

Třída `TreeElement` je abstraktní třída, od které všechny tyto třídy dědí. Účelem této třídy je umožnit zobrazení projektu ve stromu. Obsahuje pouze několik abstraktních metod, které potomci této třídy musí implementovat, aby bylo možné je správně zobrazit ve stromu. Tyto metody slouží pro získání obsahu souboru nebo popisu souboru, získání potomků tříd a získání ikon pro zobrazení ve stromu.

Třída `ClassRep` slouží pro repezentaci tříd, rozhraní a výčtových typů. Ty od sebe odlišuje pomocí výčtového typu `ClassType`. Tato třída ukládá informace o zpracovaných třídách. Pokud je třída vnitřní třídou jiné, pak ukládá referenci na tuto nadřazenou třídu. Dále ukládá komentář třídy, anotace třídy, hlavičku třídy, `List` všech metod, které tato třída obsahuje a `List` všech tříd, které tato třída obsahuje. Jediné metody, které tato třída používá kromě svých konstruktorů, getrů, setrů a metod zděděných ze třídy `TreeElement`, jsou metody pro přidání dalších prvků do listů metod a tříd. Třída neukládá zdrojový kód třídy, ale pouze atributy a metody. Výsledný zdrojový kód třídy při exportu je z těchto částí znovu složen.

Třída `MethodRep` slouží pro repezentaci metod a konstruktorů, které od sebe také rozlišuje pomocí výčtového typu `MethodType`. Tato třída ukládá

informace o metodách, to jsou komentáře, které k metodám patří, anotace metod, hlavičky metod a těla metod. Tato třída nemá kromě konstruktorů, getrů, setrů a zděděných metod žádné vlastní metody.

Třída `Directory` reprezentuje adresáře. Tato třída obsahuje pouze `Listy` s adresáři, soubory se zdrojovými kódy a ostatními soubory, které daný adresář obsahuje. Kromě konstruktorů, getrů, setrů a zděděných metod obsahuje metody pro přidání soubory se zdrojovými kódy, ostatními soubory a adresáři do listů.

Třída `FileSource` repezentuje soubor se zdrojovým kódem. Protože třída může obsahovat více tříd, které nejsou do sebe vnořené, obsahuje deklarace balíku a importy. Kromě konstruktorů, getrů, setrů a zděděných metod obsahuje pouze metodu pro přídání třídy do `Listu` tříd.

Třída `FileOther` reprezentuje soubory, které neobsahují zdrojový kód. Obsahuje obsah těchto souborů a jejich velikost. Pomocí výčtového typu `FileType` od sebe odlišuje několik různých souborů. Odlišovány jsou soubory `.class`, soubory s textovým obsahem, soubory, které nemají textový obsah a soubory se zdrojovými kódy, které obsahují syntaktické chyby, které znemožňují syntaktickou analýzu. Jelikož rozlišování souborů podle toho, jestli je jejich obsah textový nebo ne, není cílem této práce, probíhá rozlišování pouze na základě přípon souborů. Obsah souborů, které jsou akceptovány jako soubory s textový obsahem je zobrazen v textové oblasti, pokud uživatel daný prvek ve stromu vybere. Tato třída kromě konstruktorů, getrů, setrů a zdeděných metod neobsahuje žádné další metody.

6.2.4 Třídy `FileType`, `MethodType` a `ClassType`

Tyto třídy také patří do balíku data. Třída `ClassType` je použita pro odlišení tříd, rozhraní a výčtových typů, protože pro jejich reprezentaci je použita pouze jedna třída. Třída také načítá a obsahuje ikony, které reprezentují třídy, rozhraní a výčtové typy v zobrazení projektů ve stromu.

Třída `MethodType` od sebe odlišuje metody a konstruktory, důvodem je také to, že pro reprezentaci metod a konstruktorů je použita jedna třída. Také načítá a ukládá ikony, které jsou použity v zobrazení projektů ve stromu.
Třída `FileType` od sebe odlišuje soubory s textovým obsahem, soubory, které obsahují například obrázky nebo audio, soubory `.class`, které představují zkompilovaný zdrojový kódy tříd a soubory se zdrojovým kódem, které obsahují syntaktické chyby a nemohly z tohoto důvodu být analyzovány. Také načítá a ukládá ikony, které jsou použity v zobrazení projektů ve stromu.

6.2.5 Třída `Strings`

Třída `Strings` je umístěna v balíku `utils` a obsahuje definované konstantní řetězce, které jsou použity především v hlavním okně aplikace a v dialogových oknech.

6.3 Načítání vstupních dat

Potom, co uživatel prostřednictvím grafického uživatelského prostředí vybere adresář projektu, je načten celý obsah adresáře a vytvořena jeho objektová reprezentace. Pro prozkoumání celého adresáře byl použit algoritmus *DFS*. Jako první se daný adresář prohledá a zkontroluje se, jestli adresář obsahuje nějaké soubory `.java`. Pokud adresář žádné `.java` soubory neobsahuje, je na tuto skutečnost uživatel upozorněn dialogovým oknem, které vyžaduje potvrzení, jestli chce uživatel opravdu vybraný adresář a jeho obsah nahrát. Po potvrzení nebo po zadání adresáře, který `.java` soubory obsahuje, program začne procházet daný adresář a postupně vytvářet jeho objektovou reprezentaci. To probíhá následujícím způsobem. Postupně jsou procházeny soubory, které adresář obsahuje. Pokud je nalezený soubor adresářem, metoda se rekurzivně zavolá s tímto adresářem jako parametrem a začne zpracovávat soubory, které se nachází v tomto adresáři. Postup je také popsán pseudokódem v algoritmu 1.

Pro jednoduchost byly vynechány kontroly návratových hodnot a namísto určování typu souborů jsou podle typu pojmenované proměnné.

6.3.1 Načítání souborů se zdrojovým kódem

Pokud nalezený soubor není adresář a končí příponou `.java`, tak je následovně zkontrolován, jestli je možné analyzovat jeho zdrojový kód, a pomocí knihovny získat objektovou reprezentaci zdrojového kódu, který soubor obsahuje. Zdrojový kód v souboru není možné analyzovat pouze v případě, že obsahuje syntaktické chyby. Pokud je možné zdrojový kód v souboru analyzovat, tak jsou následně z objektové reprezentace zdrojového kódu získané pomocí metod knihovny JavaParser pomocí třídy `Parser` získány jednotlivé části zdrojového kódu a vytvořena objektová reprezentace tohoto souboru, která je pak přidána do `Listu` souborů se zdrojovým kódem patřícího adresáři, který je zrovna zpracováván. Pokud zdrojový kód v souboru není možné analyzovat, pak je tento soubor zpracován stejně jako soubory, které neobsahují zdrojový kód, s tím rozdílem, že jeho typ je nastaven na typ *`UNPARSEABLE`*, který označuje soubor se zdrojovým kódem, který nebylo možné analyzovat. Tento soubor je pak přidán do `Listu` souborů bez zdrojového kódu, který patří zpracovávanému adresáři.

6.3.2 Načítání souborů bez zdrojového kódu

Soubory, které neobsahují zdrojové kódy, jsou zpracovány následovně. Pokud soubor končí příponou `.class` je mu přidělen typ *`CLASSTYPE`*, který jej označuje jako soubor, který obsahuje zkompilovaný zdrojový kód a program se nebude snažit zobrazit jeho obsah v textové oblasti pro zobrazení informací o prvku. Pokud soubor nekončí příponou .class, tak se zkontroluje, jestli název souboru nekončí některou z přípon, které jsou definovány v souboru `text_extensions.txt`. Tento soubor obsahuje přípony souborů, jejichž

obsah lze číst jako text. Jsou mezi nimi například přípony `.xml`, `.txt`, `.ini`, atd. soubor je umístěn v `.jar` souboru aplikace. Pokud se tedy přípona kontrolovaného souboru shoduje s některou z přípon uvedených v souboru s příponami, je tento soubor označen typem *`TEXT`*, který určuje, že obsah souboru je možné zobrazit v textové oblasti, pokud uživatel daný soubor vybere. Soubory, jejichž přípona se neshoduje se žádnou z přípon v souboru se známými příponami textových souborů je označen typem *`BINARY`*, podle kterého program pozná, že jeho obsah není textový a nebude se pokoušet tento obsah zobrazit. Potom se načte soubor obsahu a určí jeho velikosti. Na základě těchto dat je pak vytvořena objektová reprezentace souboru, která je přidána do `Listu` souborů bez zdrojového kódu v právě zpracovávaném adresáři.

Algoritmus 1 Algoritmus pro vytvoření objektové reprezentace projektu

```
    Directory → načtený adresář
    DirectoryObject → objektová reprezentace adresáře
    procedure getDirectoryContents(Directory, DirectoryObject)
        for each file in directory do
            if file is Directory then
                newDir = create new DirectoryObject
                getDirectoryContents(file, newDir)
            else if file.endsWith(".java") then
                if file.isParseable() then
                    srcFile = parseFile(file)
                    DirectoryObject.srcFiles.add(srcFile)
                else
                    unparseableFile = readFile(file)
                    DirectoryObject.files.add(unparseableFile)
                end if
            else
                if isTextFile(file) then
                    textFile = readFile(file)
                    DirectoryObject.files.add(textFile)
                else if file.endsWith(".class") then
                    classFile = readFile(file)
                    DirectoryObject.files.add(classFile)
                else
                    binaryFile = readFile(file)
                    DirectoryObject.files.add(binaryFile)
                end if
            end if
        end for
    end procedure
```

6.4 Získání částí zdrojového kódu tříd

Při nalezení souboru se zdrojovým kódem, který je možné analyzovat se získá objektová reprezentace zdrojového kódu ze souboru. Získání objektové reprezentace kódu zajišťuje knihovna JavaParser. Tato objektová reprezentace v podobě stromu je pak dále zpracovávána a to následujícím způsobem. Potomky kořene tohoto stromu tvoří uzly reprezentující deklarace balíku, importy a třídy. `List` těchto potomků je procházen jako pole a pomocí operátoru `instanceof` je zjištěno, kterou část zdrojového kodu daný uzel reprezentuje. Nejprve je z toho balíku získána deklarace balíku a importy. Ty tvoří hlavičku objektové reprezentace souboru se zdrojovým kódem. Poté jsou zpracovávané všechny třídy, které jsou ve zdrojovém kódu deklarovány a následně jejich metody. Po zpracování všech tříd a metod se vytvoří objektová reprezentace tohoto souboru a je přidána do `Listu` souborů se zdrojovým kódem v přišlusném adresáři. Postup při zpracování zdrojového kódu v souboru je popsán v algoritmu 2.

Algoritmus 2 Algoritmus pro zpracování zdrojového kódu v souboru

```
    codeObject → stromová reprezentace zdrojového kódu
    header → řetězec obsahující hlavičku
    srcFileName → název souboru
    procedure parseSourceFile(codeObject, srcFileName)
        header
        for each node in codeObject do
            if node instanceof package OR node instance of import then
                header.append(node)
        end for
        srcFile = create new SourceFile(srcFileName, header)
        for each node in codeObject do
            if node instanceof ClassOrInterface then
                class = parseClass(node)
                srcFile.addClass(class)
            else if node instanceof Enum then
                enum = parseEnum(node)
                srcFile.addClass(enum)
        end for
    return srcFile
    end procedure
```

6.4.1 Zpracování tříd, rozhraní a výčtových typů

Při zpracovávání tříd je opět pomocí operátoru `instanceof` zjištěno, jestli se jedná o uzel reprezentující třídu nebo výčtový typ. V případě, že se jedná o třídu, je pak ještě zjištěno, jestli se nejedná o rozhraní. Knihovna *JavaParser* sjednocuje třídy a rozhraní jako jeden typ a pro jejich odlišení existuje metoda `isInterface()`, která vrací `true`, pokud se jedná o rozhraní.

Třídy a rozhraní jsou dále zpracovávány tak, že se získají hlavičky, atributy, anotace a komentáře. Hlavičky obsahují modifikátory, mezi ně patří modifikátory přístupu, modifikátor `static` a modifikátor `final`. Dále hlavička obsahuje název třídy nebo rozhraní, datový typ, pokud je deklarován, dále rozhraní, která jsou implementována a třídy, od kterých dědí. Na základě těchto dat je vytvořena instance reprezentující tuto třídu. Při zpracování výčtového typu je použit podobný postup s tím rozdílem, že výčtový typ nemůže být typovaný a nemůže dědit od jiných tříd. Po vytvoření objektové reprezentace zpracovávané třídy, rozhraní nebo výčtového typu se dále procházejí potomci uzlu, který je reprezentuje ve stromu zdrojového kódu a opět pomocí operátoru `instanceof` jsou vybrány a postupně zpracovány všechny metody a konstruktory, které daná třída, rozhraní nebo výčtový typ obsahuje.

Potom, co jsou všechny metody a konstruktory třídy zpracovány, metoda, která zpracovává třídu, rozhraní nebo výčtový typ, vrátí jejich objektovou reprezentaci. Tato objektová reprezentace je pak přidána do `Listu` tříd příslušného souboru se zdrojovým kódem nebo, pokud se jedná o vnořenou třídu, rozhraní nebo výčtový typ, do `Listu` tříd příslušného objektu, ve kterém jsou definovány. Postup při zpracování zdrojového kódu tříd je popsán algoritmem 3. Pro zjednodušení je uveden pouze postup pro třídy, pro rozhraní i výčtový typ je postup odbodný.

6.4.2 Zpracování metod a konstruktorů

V případě, že nalezený uzel reprezentuje vnitřní třídu, rozhraní nebo výčtový typ je tato metoda rekurzivně spuštěna s tímto nalezeným uzlem jako parametrem. Metody a konstruktory se zpracovávají stejně a to tak, že je pouze získán komentář, anotace hlavička a tělo metody. Uzly metody a konstruktorů sice mají ve stromu další potomky, ovšem tito potomci už jsou z hlediska této práce nezajímavé. Po vytvoření objektové reprezentace metody nebo konstruktoru je tato reprezentace metodou vrácena a přidána do listu metod příslušné třídy. Postup je také popsán algoritmem 4. Pro zjednodušení je uveden pouze algoritmus pro metody, pro konstruktory je použit stejný postup.

Algoritmus 3 Algoritmus pro zpracování zdrojového kódu tříd

```
    class→uzel ve stromu zdrojového kódu
    parentClass →nadřazená třída
    procedure parseClass(class, parentClass)
        header →hlavička třídy
        fields →atributy
        comment → komentář třídy
        annotations → anotace třídy
        for each modifier in class.getModifiers() do
            header.append(modifier) // stejný postup pro annotations
        end for
        header.append(className)
        header.append(classType)
        comment = class.comment
        classObject = create new ClassObject(header, fields, comment,
                                  annotations, parent)
        for each node in class.nodes do
            if node instanceof Constructor then
                constructor = parseMethod(node)
                classObject.addMethod(contructor)
            else if node instanceof Method then
                method = parseMethod(node)
                classObject.addMethod(method)
            else if node instanceof ClassOrInterface then
                class = parseClass(node)
                classObject.addClass(class)
            else if node instanceof Enum then
                enum = parseEnum(node)
                classObject.addClass(enum)
        end for
    return classObject
    end procedure
```

Algoritmus 4 Algoritmus pro zpracování zdrojového kódu metod a konstruktorů

```
method → uzel ve stromu zdrojového kódu
procedure parseMetod(method)
    modifiers → řetězec pro modifikátory
    annotations → řetězec pro anotace
    parameters → řetězec pro parametry metody
    comment → komentář metody
    returnType → návratový typ
    body → tělo metody
    for each modifier in method.getModifiers() do
        modifiers.append(modifier)
    end for
    // stejný postup pro annotations a parameters
    returnType = method.getType()
    if method.comment.isPresent() then
        comment = method.comment
    end if
    if method.body is present() then
        body = method.body
    methodObject = create new MethodObject(modifiers, returnType,
                                annotations,parameters, comment, body)
return methodObject
end procedure
```

6.5 Ukládání a získávání dat z databáze

6.5.1 Ukládání dat do databáze

Pro ukládání a získávání dat z databáze byl použit framework Hibernate. Po načtení a zpracování uživatelem zadaného adresáře projektu je výsledná objektová reprezentace projektu nahrána do databáze. Nahrání projektu do databáze probíhá tedy prostřednictvím Hibernate. Ve zdrojovém kódu tříd pro reprezentaci dat bylo pomocí anotací definováno, do které tabulky v databázi se budou data týkající se této třídy ukládat. Dále tyto anotace definují datové typy, které budou použity pro uložení atributů tříd a definují vazby mezi tabulkami v databázi. V programu tedy stačí předat objekt, který reprezentuje kořenový adresář projektu jedné z metod frameworku a ten už zařídí, aby byly do databáze uloženy i všechny další objekty, které objekt adresáře obsahuje

ve svých `Listech`. Pro všechny objekty z těchto listů je proces opakován. Hibernate tedy vloží celý projekt do databáze bez nutnosti použití psaní dotazů.

6.5.2 Získávání dat z databáze

Po spuštění programu jsou z databáze vybrány všechny projekty, aby mohly být zobrazeny ve stromu v grafickém uživatelském rozhraní. Při získání všech projektů se opět o získání celé struktury projektu stará Hibernate. Avšak už je nutné použití dotazu, aby byly získány pouze kořenové adresáře projektů, protože v tabulce, ve které jsou uloženy kořenové adresáře projektů jsou uloženy všechny adresáře, které se v projektu nacházejí. Použitý dotaz tedy vybírá pouze adresáře, jejichž sloupec, který obsahuje *id* nadřazeného adresáře obsahuje hodnotu *null*. *Hibernate* pak vrátí *`List`* obsahující všechny uložené projekty.

Pro získání pouze jednoho projektu je použit stejný postup s tím rozdílem, že namísto dotazu, který vybírá pouze adresáře s hodnotou *null* ve sloupci s *id* nadřazeného adresáře, vybírá pouze adresář podle konkrétního *id*.

6.6 Export projektu

Potom, co uživatel prostřednictvím grafického uživatelského rozhraní zvolí proekt k exportu, je tento projekt na základě *id* kořenového adresáře projektu získán z databáze projektů. Po získání projektu se jako první vytvoří kořenový adresář projektu v uživatelem zvoleném adresáři na disku. Program poté prochází potomky adresáře projektu a postupně je vytváří. Při vytváření podadresářů se začne rekurzivně zpracovávat daný podadresář a začnou se vytvářet soubory a podadresáře, které obsahuje.

Při zapisování souborů se zdrojovým kódem je využita metoda, kterou dědí od třídy TreeElement, která vrací obsah souboru jako řetězec. Obsah souboru se složí z hlavičky, která je uložena v jeho objektové reprezentaci a jednotlivých

tříd. Třídy se také složí z hlaviček, atributů a jednotlivých metod. Původní uspořádání kódu tedy nemusí být zachované, nicméně výsledná funkčnost kódu musí být stejná jako předtím, než byl projekt, který obsahuje tento soubor se zdrojovým kódem, nahrán do databáze. Při vytváření souborů bez zdrojového kódu se pouze zapíše uložený obsah těchto souborů.

7 Testování

Třídy a metody aplikace byly otestovány jednotkovými testy s pomocí frameworku JUnit verze *4.1.2*. Vzhledem ke struktuře aplikace bylo dosaženo pokrytí kódu *70%*. Největší část aplikace tvoří metody pro reprezentaci dat a výčtové typy určené k odlišení například tříd, rozhraní, a výčtových typů. Metody těchto tříd obsahují především getry a setry, které nebyly testovány, jelikož byly automaticky vygenerovány pomocí IDE Eclipse.

Třídy pro grafické uživatelské rozhraní, připojení k databázi, metody pro vytváření a zápis souborů, třída s konstantami a hlavní třída aplikace nebyly testovány pomocí jednotkových testů. Tyto třídy a metody byly testovány pomocí funkcionálního testování. To bylo provedeno pomocí množství vytvořených tříd. Grafické uživatelské rozhraní bylo testováno pomocí scénářů, ve kterých uživatel postupuje podle předepsaného návodu. Celá aplikace byla nakonec otestována na projektech, které byly vytvořeny jako semestrální práce v rámci studia.

7.1 Testování jednotkovými testy

Při testování pomocí jednotkových testů byla vytvořena jednoduchá třída. Tato třída nemá žádnou skutečnou funkčnost a nebylo by možné ji přeložit, avšak neobsahuje žádné syntaktické chyby. Pro testování zdrojového kódu se syntaktickou chybou byla vytvořena jedna vedlejší třída, jejíž obsah je minimální. Třída je načtena a zpracována pomocí třídy pro načítání souborů a zdrojový kód třídy je analyzován syntaktickým analyzátorem. Jednotkové testy se pak provádí nad výslednou vytvořenou objektovou reprezentací této třídy.

Testovány byly jednotlivé části tříd a metod, jako jsou hlavičky, atributy, těla atd. Následně byly testovány i celé zdrojové kódy tříd, které se neukládají, ale vždy se zpět složí z uložených atributů a zdrojových kódů metod, které jsou ve třídě deklarovány. Soubor se zdrojvý kódem, obsahuje třídy, rozhraní,

výčtové typy, abstraktní třídu, vnitřní třídy atd. Třída `Strings`, která obsahuje pouze zadefinované konstany řetězců, které aplikaci používá v grafickém uživatelském rozhraní, testována nebyla.

Pro otestování objektové reprezentace souborů bez zdrojového kódu a adresářů byl vytvořen jednoduchý adresář, který obsahuje pouze několik souborů a jeden podadresář. Tento adresář je načten a následně jsou testovány typy a obsah souborů, které obsahuje. Zároveň byly otestovány i metody, které prohledávají adresář a zjišťují, jestli obsahuje zdrojové kódy a porovnávají přípony souboru a podle nich určí zda se jedná o soubor s textovým obsahem.

Nejvíce chyb v aplikaci bylo odhaleno při testování s jednotkovými testy. Zároveň byly nalezeny úseky kódu, které nebyly nikdy vykonávány a došlo tak k mírnému zkrácení a zpřehlednění kódu. Mezi tyto chyby patřilo například špatné zpracování hlavičky metody v případě, kdy byl použit generický návratový typ.

Pokrytí kódu je zobrazeno na obrázku 14.

AppTest (1) (27.6.2019 0:11:34)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
ProjectManager	70,5 %	2 981	1 250	4 231
src/main/java	64,8 %	2 283	1 240	3 523
cz.pavlikl.bp.gui	0,0 %	0	718	718
AppWindow.java	0,0 %	0	718	718
cz.pavlikl.bp.app	79,3 %	1 502	392	1 894
DatabaseConnection.java	0,0 %	0	224	224
FileIO.java	62,4 %	256	154	410
App.java	0,0 %	0	11	11
Parser.java	99,8 %	1 246	3	1 249
Parser	99,8 %	1 246	3	1 249
cz.pavlikl.bp.data	86,0 %	781	127	908
ClassRep.java	87,9 %	196	27	223
Directory.java	80,9 %	106	25	131
MethodRep.java	78,1 %	82	23	105
FileSource.java	83,3 %	95	19	114
TreeElement.java	40,9 %	9	13	22
FileOther.java	88,7 %	86	11	97
ClassType.java	96,1 %	73	3	76
FileType.java	96,1 %	73	3	76
MethodType.java	95,3 %	61	3	64
cz.pavlikl.bp.utils	0,0 %	0	3	3
Strings.java	0,0 %	0	3	3
src/test/java	98,6 %	698	10	708

Obrázek 14: Pokrytí kódu jednotkovými testy.

7.2 Funkcionální testování

Při funkconálním testování byla aplikace testována pomocí množství vytvořených tříd. Některé z těchto tříd fungovaly samostaně, jiné byly seskupeny do menších projektů. Zároveň byly v některých z těchto tříd úmyslně zaneseny syntaktické i sémantické chyby. Třídy, které byly přeložitelné prováděly pouze jednoduché výpočty, na kterých bylo možné ověřit, že zdrojový kód i po zpracování aplikací stále správně provádí svou činnost.

Třídy byly načteny a zpracovány programem a po uložení do databáze byly zobrazeny v grafickém uživatelském rozhraní aplikace. Zde bylo zkontrolovány, že zdrojový kód souborů byl správně rozdělen na jednotlivé metody a třídy. Po jejich exportu z databáze na disk byly spuštěny a porovnány výsledky jejich činnosti s výsledky jejich původních verzí.

Tuto kontrolu by bylo možné provádět automaticky, bohužel, zdrojový kód v souborech nemusí být po zpracování a uložení do databáze uspořádaný stejně jako před zpracováním zdrojového kódu. V případě, kdy jsou třídy přeložitelné a mají jednoduchý výstup by tento fakt automatické kontrole nebránil. Ovšem jedním z požadavků je, že do programu mohou být ukládány i soubory se zdrojovými kódy, které nelze přeložit z důvodu úmyslně zanesených chyb. V tomto případě by automatické porovnávání neposkytovalo správné výsledky. Chyby, které by přeložené třídy vypsaly by měly být stejné, ale ve výpisu může být rozdíl v řádce kódu, na které se chyba vyskytla.

7.3 Testování grafického uživatelského rozhraní

Tato aplikace má pouze jednoduché grafické uživatelské rozhraní, které tvojí jedno hlavní okno aplikace. Aplikace dále používá dialogová okna k upozornění uživatele na např. neexistující adresář nebo pokus o smazání adresáře, když není žádný vybrán.

Grafické uživatelské rozhraní tedy bylo testováno pomocí scénářů, ve kterých uživatel provádí akce podle předepsaného návodu. Při provádění akcí uživatel kontroluje, jestli aplikace na dané události reaguje správně. Uživatel také během provádění scénářů kontroluje, jestli byly vložené projekty zpracovány správně.

7.4 Testování na reálných datech

Nakonec byla aplikace testována na reálných datech. Tyto data tvořily zejména projekty, které byly vytvořeny jako semestrální práce v rámce studia na vysoké škole. Tyto projekty obsahují složitější konstrukce, než vytvořené testovací třídy a tak představují vhodná testovací data. Úplně posledním testem aplikace bylo, že aplikace zpracuje, uloží a exportuje své vlastní zdrojové kódy. Další důležitá vlastnost těchto projektl je, že byly vytvořeny v různých nástrojích IDE a bylo tak otestováno i to, že program správně pracuje s různými strukturami projektů.

8 Omezení a možnosti rozšíření aplikace

Největším omezením vytvořené aplikace je, že nedokáže zpracovávat zdrojové kódy, které obsahují syntaktické chyby. Tento problém je v programu částečně vyřešen tím, že jsou zdrojové kódy v souborech kontrolovány, jestli je lze analyzovat. Pokud není možné zdrojový kód analyzovat, aplikace načte soubor stejně jako soubory, které neobsahují zdrojové kódy.

Databáze používá pro ukládání souborů, které neobsahují zdrojové kódy, datový typ, který může pojmout téměř *4GB* dat. Ovšem pro nahrávání větších souborů je nutné v databázi upravit hodnotu proměnné `max_packet_size`, jinak se větší soubory do databáze neuloží. Aplikace na tuto skutečnost upozorní, pokud nastane. Hodnotu této proměnné nelze nastavit z aplikace. Nicméně aplikace je zamýšlena pro zpracovávání souborů se zdrojovým kódem a u těchto souborů tato situace zpravidla nenastane.

Jednou možností budoucího rozšíření by bylo vytvoření metod, které umožní například přeskočit, a bez zpracování uložit, tu část zdrojového kódu, která obsahuje syntaktickou chybu. Část zdrojového kódu obsahující chybu by pak mohla být v textové oblasti, kde se zobrazuje obsah souborů, zvýrazněna popř. by mohla být zvýrazněna přímo pozice, kde program nalezl chybu.

Dalším možným rozšířením je umožnění editace zdrojového kódu, který je uložen v databázi nebo vytvoření metod, které možní porovnat soubory, třídy a metody z nahrávaného projektu s existujícími prvky v databázi a při nalezení velmi podobných souborů uživatele upozornit, popř. zobrazit rozdíly mezi prvky v grafickém uživatelském rozhraní.

9 Závěr

Tato práce obsahuje popis několika v praxi používaných struktur projektů Java. Dále je v této práci popsána syntaktická analýza, datové struktury, které syntaktická analýza využívá a některé metody používané syntaktickými analyzátory pro analýzu zdrojového kódu a v kapitole 5.5 jsou zmíněny existující nástroje pro analýzu zdrojového kódu.

Vytvořený nástroj byl otestován a na základě výsledků testů je možné usoudit, že nástroj pracuje správně. Vytvořený nástroj umožňuje načtení Java projektů, vytváření jejich objektové reprezentace, uložení této reprezentace do databáze a export uložených projektů z databáze. Nástroj tedy splňuje požadavky zadání.

Zdroje

[1] Philippe Faes, *How to organize source code? Learn from Java.*
URL: https://insights.sigasi.com/tech/how-organize-source-code-learn-java.html (cit. 5. 5. 2019)

[2] *Class*
URL: www.techopedia.com/definition/3214/class-java
(cit. 5. 5. 2019)

[3] *Java – Packages*
URL: www.tutorialspoint.com/java/java_packages.htm
(cit. 5. 5. 2019)

[4] *Creating, Importing, and Configuring Java Projects*
URL: www.netbeans.org/kb/74/java/project-setup.html#ide-concepts
(cit. 6. 5. 2019)

[5] Gaurav Miglani, *Java naming conventions*
URL: www.geeksforgeeks.org/java-naming-conventions/
(cit. 7. 5. 2019)

[6] Apache Maven Standard Directory Layout
URL: https://www.baeldung.com/maven-directory-structure
(cit. 8. 5. 2019)

[7] Dick Grune, Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide*
In: Ellis Horwood, Chichester, England (1990)

[8] Dick Grune, Ceriel J.H. Jacobs. *Parsing Techniques - A Practical Guide: 2nd edition.* In: www.springer.com (2010)

[9] Frederico Tomassetti, *A Guide to Parsing Algorithms and Terminology*
URL: https://tomassetti.me/guide-parsing-algorithms-terminology/
(cit. 10. 05. 2019)

[10] *Compiler Design - Symbol Table*
URL:https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm (cit. 10. 05. 2019)

[11] *Noam Chomsky, THREE MODELS FOR THE DESCRIPTION OF LANGUAGE*
URL:https://chomsky.info/wp-content/uploads/195609-.pdf
(cit. 11. 5. 2019)

[12] Václav Vais, *Teoretická informatika 1. část, Konečné automaty a regulární jazyky*
URL: http://home.zcu.cz/~vais/Vais%20-%20KA%20a%20RG.pdf
(cit. 14. 5. 2019)

[13] *Introduction to Finite Automata*
URL: https://elearningatria.files.wordpress.com/2013/10/cse-v-formal-languages-and-automata-theory-10cs56-solution.pdf
(cit. 18. 5. 2019)

[14] Peter Fritzson, Pavol Privitzer, Martin Sjölund, Adrian Pop, *Towards a Text Generation Template Language for Modelica*
URL:*https://www.researchgate.net/figure/Abstract-syntax-tree-of-the-while-loop_fig1_228792639*
(cit. 25. 5. 2019)

[15] *Parse Trees*
URL:*https://www.cs.wcupa.edu/rkline/fcs/parse-trees.html*
(cit. 30. 5. 2019)

[16] *Compiler Design - Types of Parsing*
URL:https://www.tutorialspoint.com/compiler_design/compiler_design_types_of_parsing.htm (cit. 4. 6. 2019)

[17] *Compiler Design - Top-Down Parser*
URL:https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm (cit. 10. 6. 2019)

[18] *Compiler Design - Bottom-Up Parser*
URL:https://www.tutorialspoint.com/compiler_design/compiler_design_bottom_up_parser.htm (cit. 11. 6. 2019)

[19] *Depth-First Search (DFS)*
URL:https://github.com/ramlaxman/CL-II/blob/master/4.%20DFS-and-its-variations.md (cit. 12. 6. 2019)

[20] Prateek Garg, *Breadth First Search*
URL:https://www.hackerearth.com/ru/practice/algorithms/graphs/breadth-first-search/tutorial/ (cit. 12. 6. 2019)

[21] Terence Parr, *The Definitive ANTLR 4 Reference.* In: Parr Pragmatic Bookshelf 2013

[22] Nicholas Smith,Danny van Bruggen, and Federico Tomassetti, *JavaParser: Visited*
URL: https://leanpub.com/javaparservisited (cit. 14. 6. 2019)

[23] *Datatype Limits*
URL:https://docs.oracle.com/cd/B28359_01/server.111/b28320/limits001.htm#i287903 (cit. 15. 6. 2019)

[24] *Data Type Storage Requirements*
URL:https://dev.mysql.com/doc/refman/8.0/en/storage-requirements.html (cit. 16. 6. 2019)

[25] *JDBC - Introduction*
URL: https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm (cit. 17.6. 2019)

[26] *Hibernate Tutorial*
URL: https://www.tutorialspoint.com/hibernate/index.htm (cit. 21.6. 2019)

A Uživatelská příručka

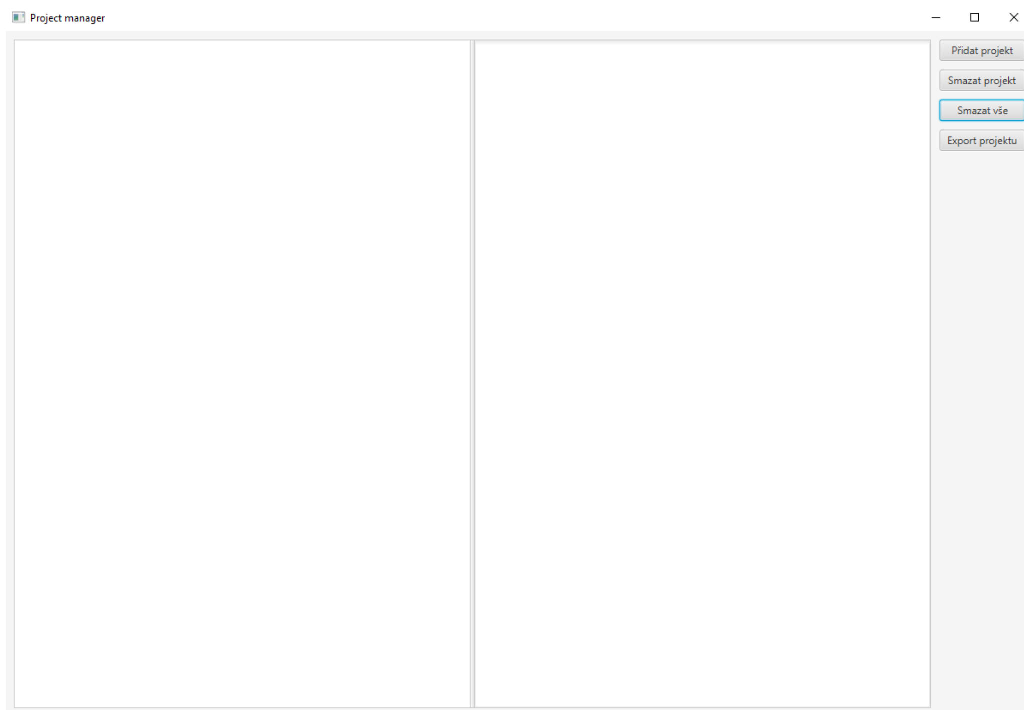
A.1 Překlad a spuštění aplikace

Pro předklad a spuštění aplikace je zapotřebí mít v počítači nainstalovanou Javu alespoň ve verzi *1.8*. Dále je pro překlad potřeba nástroj pro správu projektů Apache Maven. Na přiloženém CD je adresář s timto nástrojem připraven a není jej třeba instalovat. Pro spuštění překladu je dále přiložen soubor build.cmd. Po jeho spuštění se provede překlad aplikace a výsledný `.jar` soubor bude umístěn v adresáři projektu v podadresáři target. Vytvořený `.jar` soubor obsahuje všechny knihovny potřebné pro běh aplikace a lze tedy aplikaci rovnou spustit. Aplikace se spouští bez parametrů a žádné použití parametrů neumožňuje.

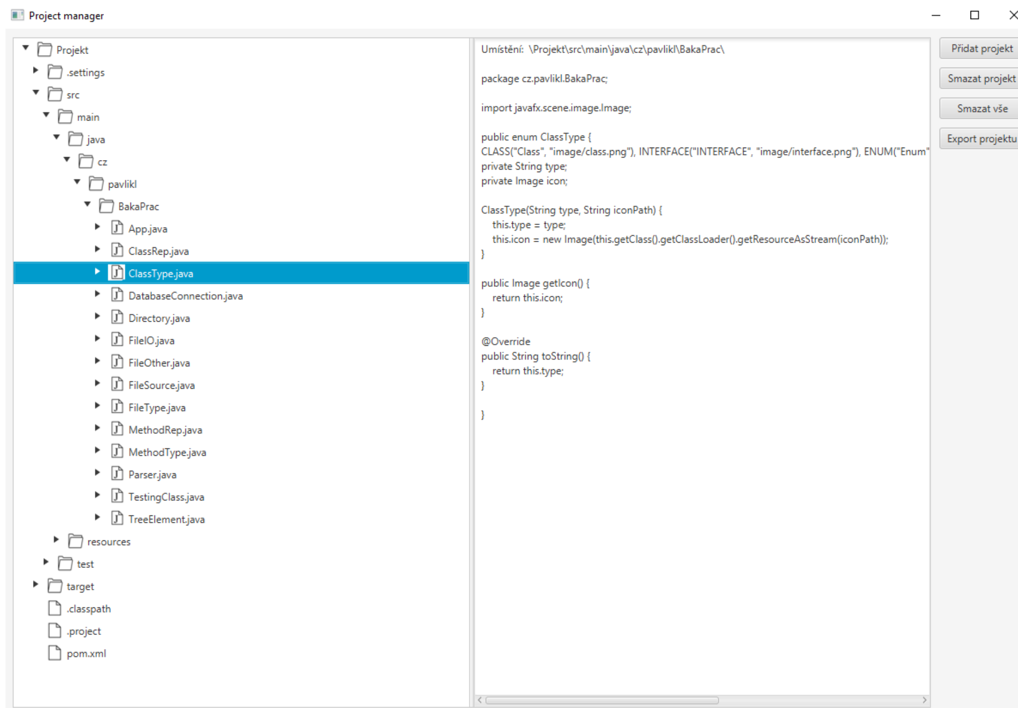
A.2 Hlavní okno aplikace

Po spuštění aplikace se zobrazí hlavní okno aplikace. Aplikace kromě dialogových okno, které slouží pro zobrazení chyb, varování nebo specifikování cesty nemá žádné další okna. Hlavní okno aplikace je rozděleno na 3 části. V první části, která je pravděpodobně při prvním spuštění programu prázdná, se zobrazují projekty, které jsou uloženy v databázi. V další části okna se zobrazují informace nebo obsah vybraného prvku ve stromu projektu. Poslední částí okna jsou tlačítka, která slouží pro ovládání aplikace. Tlačítko „Přidat projekt“ otevře dialogové okno, ve kterém je uživatel vyzván, aby zvolil adresář projektu. Po zvolení projektu se projekt nahraje do databáze a zobrazí se ve stromu projektů. Dalším tlačítkem je tlačítko „Smazat projekt“. Před kliknutí na toto tlačítko je třeba zvolit kořenový adresář projektu, který má být smazán. Pokud je vybrán jiný prvek stromu, zobrazí se upozornění v podobě dialogového okna, že musí být vybrán kořenový adresář projektu.

Po vybrání kořenového adresáře projektu a kliknutí na tlačítko smazat se objeví dialogového okno žádající potvrzení akce. Akce smazání projektu je nevratná. Dalším tlačítkem je tlačítko „Smazat vše“. Po kliknutí na toto tlačítko budou vymazány všechny projekty v databázi. Před provedením akce se zobrazí dialogové okno žádající o potvrzení akce. Tato akce je nevratná. Posledním tlačítkem je tlačítko „Export projektu“. Před kliknutím na toto tlačítko musí být vybrán kořenový adresář projektu. Pokud je vybrán jiný prvek stromu, zobrazí se dialogové okno upozorňující na to, že musí být vybrán kořenový adresář. Po kliknutí na toto tlačítko se zobrazí dialogové okno, ve kterém je uživatel vyzván, aby určil do kterého adresáře na disku se projekt exportuje. Po potvrzení bude do daného adresáře projekt exportován. Hlavní okno aplikace je zobrazeno na obrázku A.1. Hlavní okno aplikace s přidaným projektem je zobrazeno na obrázku A.2.

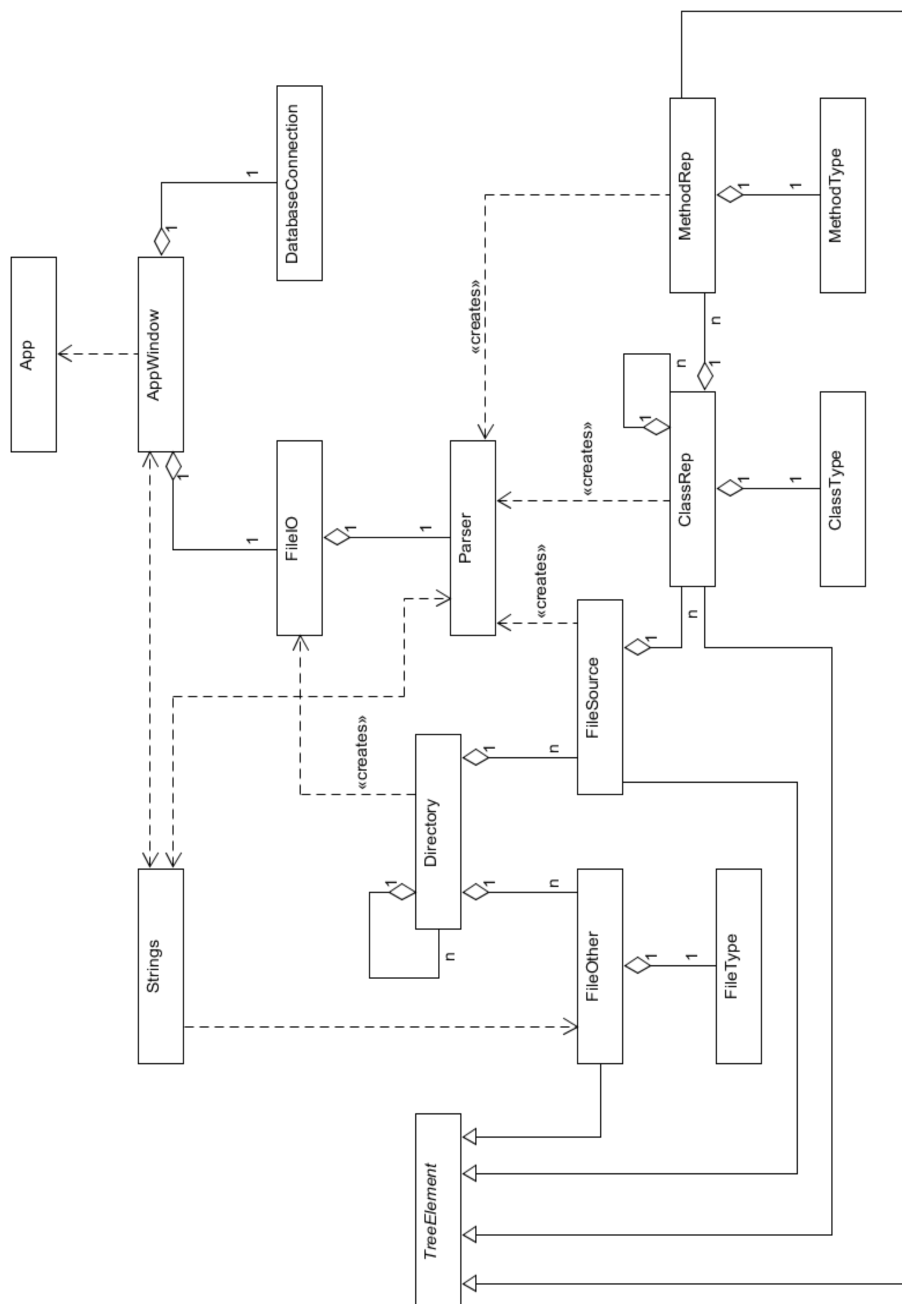


Obrázek A.1: Hlavní okno aplikace.



Obrázek A.2: Ukázka vloženého projektu a zobrazení obsahu vybraného prvku.

B UML diagram tříd



Obrázek B.1: Uml diagram tříd.