UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Analýza zdrojového kódu aplikace v jazyku Java

Petr Mojžíš

Bakalářská práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Petr Mojžíš**
Osobní číslo:	**I09199**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Analýza zdrojového kódu aplikace v jazyku Java**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Cílem práce je lexikální a syntaktická analýza jazyka Java. Data budou využity pro sestavení UML modelu analyzovaného projektu a uloženy do XML-souboru.
- Stručný popis a zápis gramatiky jazyku Java.
- Výběr hostujícího jazyka analyzátoru.
- Sestavení scannera (lexikálního analyzátoru).
- Sestavení parsera (syntaktického analyzátoru).
- Uložení získaných informací ve tvaru XML.
- Grafické zobrazení UML analyzovaného projektu.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

Herout, P.: Učebnice jazyka Java, Koop, České Budějovice, 2001
MOLNÁR, Ľ. - ČEŠKA, M. - MELICHAR, B. Gramatiky a jazyky. Bratislava: ALFA 1987.
Hopcroft J. E. - Ullman J. D.: Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 2000

Vedoucí bakalářské práce: **RNDr. Miroslav Benedikovič**
Katedra softwarových technologií

Datum zadání bakalářské práce: **21. prosince 2012**
Termín odevzdání bakalářské práce: **10. května 2013**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 29. března 2013

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

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

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 28. 03. 2013 Petr Mojžíš

Poděkování

Rád bych zde poděkoval vedoucímu bakalářské práce panu RNDr. Miroslavu Benedikoviči za poskytnutí odborných rad, věcné připomínky, ochotu a vstřícný přístup během zpracování této práce. Dále patří velké poděkování mé rodině a přítelkyni Báře za to, že mi byli oporou a umožnili mi tak tuto práci dokončit.

Anotace

Cílem této bakalářské práce je podrobit libovolný projekt napsaný v programovacím jazyce Java lexikální a syntaktické analýze. Výsledky analýz jsou následně využity k vytvoření UML diagramu tříd, který představuje grafické znázornění souborů projektu, jejich atributů, operací a vazeb mezi nimi. Aplikace rovněž obsahuje možnost uložení analyzovaného projektu do souboru XML, který umožňuje lepší správu obsahu projektu než samotný projekt v programovacím jazyce Java.

Klíčová slova

lexikální analýza, syntaktická analýza, Java, UML, XML, analyzátor

Title

Analysis of the source code in Java applications

Annotation

The aim of this bachelor thesis is to subject any project written in Java programming language lexical and syntactic analysis. The results of analyzes are then used to create a UML class diagram which is a graphical representation of project files, their attributes, operations and relationships between them. The application also includes the ability to save analyzed project into an XML file, which allows better content management project than the project itself in Java programming language.

Keywords

lexical analysis, syntactic analysis, Java, UML, XML, analyzer

Obsah

Seznam zkratek

ASCII	American Standard Code for Information Interchange
CWM	Common Warehouse Model
IDE	Integrated Development Environment
J2EE	Java 2 Enterprise Edition
JNI	Java Native Interface
JVM	Java Virtual Machine
LINQ	Language Integrated Query
MOF	Meta Object Facility
OCL	Object Constraint Language
OOP	Object-Oriented Programming
SQL	Structured Query Language
UML	Unified Modeling Language
UTF-16	UCS Transformation Format-16
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	eXtensible Markup Language

Seznam obrázků

1 Úvod

Tato bakalářská práce popisuje specifické vlastnosti jazyka Java a její obecný gramatický zápis. Dále hlouběji vysvětluje význam jednotlivých slov definujících jazyk Java. V rámci lexikální struktury jsou popsány pojmy, které specifikují syntakticky správný zápis programového kódu. Jsou zde přiblíženy komentáře, klíčová slova, operátory a další elementy popisující jazyk Java.

V další části práce je popsán hostující jazyk, ve kterém byla bakalářská práce napsána. Jsou zde uvedeny důvody volby vybraného hostujícího jazyka. Zejména je popsán Windows Presentation Foundation, značkovací jazyk XAML a dotazovací jazyk LINQ.

Následující kapitola vysvětluje funkci překladače a jeho obecný návrh. Je zde uveden průběh, jak je programový kód postupně zpracováván překladačem, až po jeho konečný výpis.

Praktická část bakalářské práce se věnuje problematice lexikální a syntaktické analýze projektu napsaného v programovacím jazyce Java, kde je načtený projekt lexikálně i syntakticky analyzován a poté graficky znázorněn pomocí UML diagramu tříd. Aplikace rovněž umožňuje uložit výsledný projekt do XML souboru.

Jedna z posledních kapitol je věnována samotné implementaci aplikace, kde je představena a vysvětlena její funkčnost, vzhled a obsluha.

Závěrem této kvalifikační práce bude proveden popis konečného stavu, funkčnosti aplikace a zhodnoceny výsledky práce.

2 Stručný popis a zápis gramatiky jazyka Java

2.1 Úvodem do programovacího jazyka Java

Programovací jazyk Java se řadí mezi objektově orientované jazyky. Základním prvkem aplikace je objekt, což je soubor atributů a metod pro práci s nimi. Objektem může být skutečný objekt reálného světa, ale není to nutnou podmínkou. Objektově orientovaný jazyk, jakým je Java, musí splňovat tři základní vlastnosti. Datovou abstrakci, tvorbu vlastních datových typů a tvorbu jejich hierarchie. Datovou abstrakcí je myšleno skrytí vnitřního stavu objektu. Koncový uživatel aplikace by neměl vědět, jak funguje vnitřní struktura objektu či třídy, ale pouze jak jej má použít. Další klíčovou vlastností návrhu OOP je možnost vytvoření vlastního datového typu, jelikož není uživatel nijak omezen předdefinovanými datovými typy daného programovacího jazyka. S možností vytvářet hierarchickou strukturu objektů můžou objekty své vlastnosti dědit od nadtypu (rodiče) a rozšiřovat jejich funkčnost o další atributy či metody.

2.2 Základní vlastnosti programovacího jazyka Java

Je mnoho názorů, co lze považovat za klíčové vlastnosti programovacího jazyka Java, a tak popíšu vlastnosti z mého hlediska důležité pro tento jazyk. Zásadní vlastností je velmi jednoduchá syntaxe, která je převzata z jazyka C++, neobsahuje však ukazatele, které jsou pro začínající programátory těžkopádné a mohou odradit od používání takového programovacího jazyka. Neméně velkou výhodou je zavedený Garbage collector, který automaticky vyhledává nepoužívané části paměti a uvolňuje je pro další využití. Zaručuje tím nižší náročnost na operační paměť a programátorovi odpadá nutnost kontroly dealokace paměti. Programovací jazyk Java je celkově velmi robustní. Používá tzv. silnou typovou kontrolu, která zaručuje správnost použitého primitivního typu a nenastávají tak situace, kdy dostaneme různorodé výsledky příkazů. Java je jazykem distribuovaným, a proto široce podporuje různé úrovně síťového spojení, práci se vzdálenými soubory a umožňuje vytvářet distribuované klientské aplikace a servery. Další zásadní vlastností je multiplatformní přístup. Javu lze použít jak v platebních kartách, mobilních zařízeních, stolních počítačích, tak i v rozsáhlých informačních systémech. Do jisté míry tato vlastnost souvisí s tím, že se jedná o interpretovaný jazyk, který místo strojového kódu vytváří přenositelný kód, který je nezávislý na architektuře cílového počítače nebo zařízení. Podmínkou pro spuštění aplikace je pouze interpret Javy, Java Virtual Machine (JVM). Ten přeloží mezikód na kód cílového zařízení a umožní spuštění aplikace.

2.3 Lexikální struktura programovacího jazyka Java

Každá aplikace musí splňovat předem stanovená pravidla jazyka a řídit se jimi. Platí to tak i u Javy. Jednotlivá slova napsaná programátorem mohou představovat pro cílový jazyk identifikátory, klíčová slova, literály a mnoho dalšího. V následujících podkapitolách je popsán význam slov, které jsou pro jazyk Java velmi důležitá a tvoří se z nich programový kód.

Aplikace v programovacím jazyce Java je napsaná ve znakové sadě Unicode. Výhodou této znakové sady je tabulka, která obsahuje všechny existující abecedy. Nenastávají poté problémy s abecedami jazyků zemí, které obsahují i několik tisíc abecedních znaků.

2.3.1 Unicode escape sekvence

Základním prvkem zdrojového kódu je Unicode escape sekvence. Skládá se ze znaků *'\uXXXX'*, kde každé písmeno *X* označuje hexadecimální číslici (ASCII znak) v kódování UTF-16. Unicode escape sekvence tedy nepředstavuje nic jiného než jeden znak abecedy. Například *'\u0041'* je znak velkého písmene *'A'*.

Postup, jak překladač zpracovává vstupní znaky je následující. Překladač Javy první rozpozná Unicode escape sekvenci na vstupu, přeloží její čtyři ASCII znaky, které následují za znaky *'\u'* a zjistí, jaký znak představuje v kódování Unicode. Tento postup opakuje, dokud nepřečte všechny znaky na vstupu.

2.3.2 Escape sekvence

V Javě existují Escape sekvence, které začínají znakem zpětné lomítko '\'. Ve výpise do konzole nejsou standardně vidět a pro překladač mají speciální význam. Jejich účelem je formátování textu, zalomování textu, horizontální tabulátory, neviditelné znaky pro odřádkování kódu a mnoho dalšího. Příkladem může být Escape sekvence *'\n'*, která odřádkuje text v používaném editoru zdrojového kódu. Escape sekvence je také Unicode escape sekvencí, která je popsána výše. Znak *'\n'* jak jsem uvedl, představuje Unicode escape sekvenci *'\u000a'*. Splňuje tedy formu zápisu v kódování Unicode, jakou Java používá.

2.3.3 Komentáře

Důležitou součástí programování je komentování složitých nebo nepřehledných částí aplikace. Java nabízí dva typy komentářů. Prvním je tradiční komentář. Uvozuje se znaky *'/*'* a ukončuje se znaky *'*/'*. Veškerý text mezi těmito znaky je při překladu vynechán a nemá žádný vliv na běh programu. Druhou možností tvorby komentářů je uvodit text znaky *'//'*. To má za následek vynechání veškerého textu při kompilaci až po konec řádku, kde byla lomítka napsána.

```
// Toto je jednořádkový komentář, který vždy končí na konci řádku.

/*
 * Toto je víceřádkový komentář (tradiční komentář), kterým je možné
 * popsat problematickou část kódu na více řádcích.
 */
```

Obrázek 1 - Možnosti komentářů
Zdroj: vlastní

Komentáře se v Javě nedají nijak vnořovat. Text by se stal nepřehledným a vedlo by to ke komplikacím, kde komentář začíná a kde končí.

2.3.4 Identifikátory

Identifikátory jsou názvy atributů, metod, tříd a dalších objektů. Používá se jich pro smysluplný název, případně popis daného objektu. V Javě se používají ke kontrole správného zápisu identifikátoru dvě metody:

- *isJavaIdentifierStart(int codePoint)* – tato metoda kontroluje, zda je prvním znakem identifikátoru velké písmeno *'A'* až velké písmeno *'Z'*, malé písmeno *'a'* až malé písmeno *'z'*, znak podtržítko *'_'* nebo znak dolaru *'$'*;

- *isJavaIdentifierPart(int codePoint)* - pokud identifikátor splňuje podmínku první metody, dojde obdobně k vyhodnocení ostatních znaků identifikátoru touto metodou. Ta je naprosto shodná s první, s tím rozdílem, že může identifikátor navíc obsahovat znaky číslic *'0'* až *'9'*.

Identifikátorem nesmí být klíčové slovo, boolean literály nebo null literál, protože již mají svůj logický význam. Identifikátor by neměl být příliš dlouhý, protože se jinak stěží čte a při používání podobných dlouhých identifikátorů může dojít snadno k záměně. To následně vede k chybě v programu, která musí být zbytečně hledána a odstraňována.

2.3.5 Klíčová slova

Klíčová slova programovacího jazyka Java jsou slova rezervovaná a nelze je použít jako identifikátory. Vedlo by to k chybnému zápisu kódu a následně by se aplikace při překladu zastavila právě v místě, kde byl zapsán nesprávný identifikátor jako jedno z klíčových slov. Zde je uveden seznam klíčových slov Javy:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Obrázek 2 - Klíčová slova programovacího jazyka Java
Zdroj: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

Klíčová slova *const* a *goto* jsou rezervovaná, avšak aktuálně nepoužívaná.

2.3.6 Literály

Literály reprezentují hodnoty primitivních datových typů, datového typu String, boolean literálů nebo null literálu.

U primitivních datových typů je literálem například hodnota 458 pro datový typ *int*, 0.4 pro *double* a podobně pro ostatní datové typy. Boolean literál může nabývat pouze dvou hodnot a to *true* nebo *false*. Literálem primitivního datového typu *char* může být například *'a'*, *'\n'*, *'\u0123'*, *'$'* apod. String literál se skládá z nula nebo více znaků uzavřených v dvojitých uvozovkách. Například jím je *"ahoj"*, *"567"* nebo i *""* – prázdný řetězec. Posledním chybějícím literálem je null literál. Ten představuje prázdnou hodnotu. Používá se především při referencích, kdy je potřeba nastavit objekt na hodnotu, která se neodkazuje nikam a jeho hodnotou je „nic“.

2.3.7 Oddělovače

Programovací jazyk Java obsahuje i tzv. oddělovače. Jsou jimi kulaté závorky *'('* a *')'*, složené závorky *'{'* a *'}'*, hranaté závorky *'['* a *']'*, znak středník *';'*, znak čárka *','* a znak tečka *'.'*. Každý z oddělovačů má svůj určitý význam. Kulaté závorky se používají pro seznam parametrů metody, kde se jednotlivé parametry oddělují znakem čárka *','*. Složené závorky vymezují rozsah platnosti kódu, ohraničují těla tříd, metod apod. Můžeme jimi také zastínit proměnné. Hranaté závorky se využívají pro vytvoření pole, například *int [ ] nazevPole*. Znakem středníku vždy ukončujeme příkaz. Pro správné zkompilování programu nesmí nikde na konci výrazu chybět. Znakem tečky se přistupuje k členským metodám daného objektu, používá se pro oddělení desetinné části čísla apod.

2.3.8 Operátory

Posledním prvkem z chybějících lexikálních struktur jsou operátory. Používají se pro matematické či aritmetické operace, bitový posun, negaci, porovnání typů, logické a bitové operace, porovnávání výrazů a další. Na obrázku je seznam všech operátorů používaných v programovacím jazyce Java.

```
=    >    <    !    ~    ?    :
==   <=   >=   !=   &&   ||   ++   --
+    -    *    /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=  >>>=
```

Obrázek 3 - Operátory programovacího jazyka Java
Zdroj: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

3 Výběr hostujícího jazyka pro sestavení analyzátorů

Rozhodnout se, v jakém programovacím jazyce naprogramovat aplikaci bakalářské práce, nebylo lehké. Po celou dobu studia byla většina prací požadována právě v Javě a tak připadala v úvahu na prvním místě. Vedle toho jsem byl seznámen s programovacím jazykem C#, který mi byl velmi sympatický a podobný tomu, co jsem znal z Javy. Splňoval mnou požadované vlastnosti a dokonce mnohem více. Zvolil jsem tedy programovací jazyk C#.

Důvodem tohoto kroku bylo několik zásadních vlastností jazyka C#:

- je jednoduchý a přehledný,
- objektově orientovaný,
- má v sobě implementovaný Garbage collector pro správu paměti,
- umožňuje jednoduchý návrh grafického rozhraní použitím WPF,
- umožňuje použití tzv. Vlastností (Properties).

Neméně důležitým faktorem bylo vývojové prostředí (IDE) Microsoft Visual Studio verze 2010, později 2012. Oproti Netbeans, které jsem používal na programování v Javě, bylo daleko rychlejší a především ladění programu bylo pro mě snazší a tím i efektivnější. Celkově se mi zdá rozvržení Microsoft Visual Studia lepší a jednodušší. Microsoft Visual Studio zajišťuje velmi dobrý základ pro tvorbu kvalitních aplikací, ať už malých či velkých rozměrů.

3.1 Využití Windows Presentation Foundation

Technologie Windows Presentation Foundation (WPF) umožňuje vytvářet grafické uživatelské rozhraní pomocí rozsáhlých knihoven, které obsahují velké množství řešení. WPF umí pracovat s vektorovou grafikou, různými animacemi, práci s multimédii či interaktivními 3D aplikacemi. Velkou výhodou je používání Direct3D knihoven pro vykreslování grafických částí aplikací. Tímto způsobem není zatěžován procesor, ale grafická karta, která bývá jinak nepříliš využita. Aplikace je celkově rychlejší a méně náročná na cílový počítač, kde bude spuštěna. V dnešní době jsou sice ve většině případů počítače dostatečně výkonné, ale pro tzv. chytré telefony s operačními systémy nebo internetové aplikace je stále důležité vytvářet aplikace umožňující efektivní správu operační paměti a optimální programový kód.

3.2 Využití značkovacího jazyka XAML

Součástí WPF je značkovací jazyk XAML, který je založený na XML. XAML je jazyk využívaný k popisu grafického rozhraní aplikací. Syntakticky je velmi jednoduchým jazykem a jeho použití je snadné, srovnatelně s XML. Důvodem použití jazyka XAML je dokonalejší oddělení funkčnosti a vzhledu aplikace. Nenastává tedy problém, kdy nejsme schopni tyto dvě odlišné části oddělit na samostatné bloky. Výhodou XAML je, že návrh, který v něm provedeme, můžeme napsat stejně tak pomocí .NET jazyků jako je C#, VB.NET a další.

```
<Window x:Class="Mojzis.GUI.UmlWin"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Analýza zdrojového kódu - UML"
        Height="600" Width="900"
        WindowStartupLocation="CenterScreen">
    <Canvas Name=" canvas" Tag=" canvasMove">
        <Canvas.Background>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="LightBlue" Offset="0" />
                <GradientStop Color="AliceBlue" Offset="1" />
            </LinearGradientBrush>
        </Canvas.Background>
    </Canvas>
</Window>
```

Obrázek 4 - Použití jazyka XAML
Zdroj: vlastní

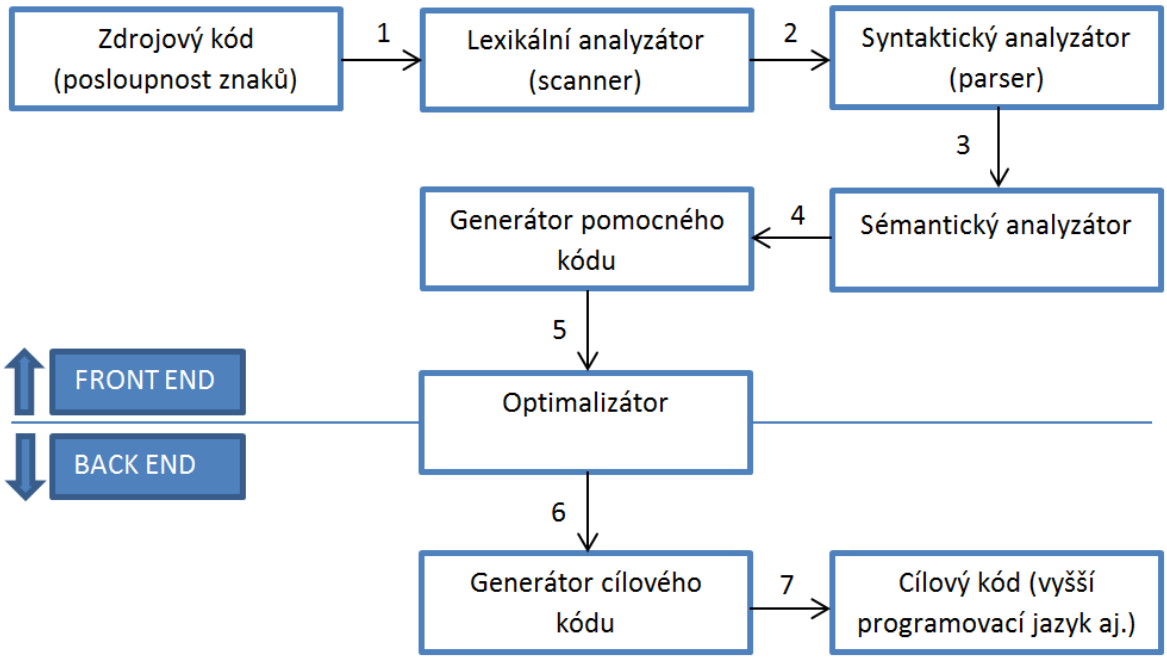
Jak je vidět z implementace jazyka XAML v aplikaci bakalářské práce, nastavení jednotlivých vlastností je opravdu jednoduché a maximálně podobné XML. Pomocí atributů je pak nastavena počáteční výška a šířka, lokace na zobrazovacím zařízení, kde se má aplikace vykreslit a celkové vykreslení grafického návrhu s nastavením barev pozadí.

3.3 Využití dotazovacího jazyka LINQ

Dotazovací jazyk LINQ, který byl integrován do .NET verze 3.5, umožňuje vykonávat dotazy nad daty. LINQ je přímo integrovaný do programovacích jazyků C# a Visual Basic .NET. Hlavním důvodem vytvoření tohoto jazyka bylo ulehčení práce programátorům při procházení kolekcí a prozkoumávání datových složek každého objektu. K procházení dat dochází velice často a je vždy potřeba psát velmi podobný kód znovu a znovu. Další nevýhodou vlastní implementace pro procházení kolekcí je většinou těsná vazba na dané objekty. Jakmile dojde ke změně struktury, kterou chceme procházet, dostáváme se do problému, kdy je nutné přepisovat již jednou napsaný návrh tak, aby umožňoval projití kolekce i po nově vytvořených změnách. LINQ je velice podobný sémantikou i syntaxí jazyku SQL, ale nabízí o mnoho širší nabídku logických datových struktur. Použitím jazyka LINQ odpadá nutnost zbytečně vytvářet vlastní návrhy, které nejsou nikdy dostatečně abstraktní, optimalizované a bezproblémové. Výhodou je pak mnohonásobné použití na různých typech objektů či struktur.

4 Obecný návrh překladače

Překladač je základním vybavením každého počítače. S jeho pomocí je počítač schopný přeložit zdrojový kód programovacího jazyka nejčastěji do strojového kódu. Bez překladače by bylo nutné, aby programátoři vytvářeli aplikace ve strojovém kódu, protože by nebyla možná přenositelnost kódu. Strojový kód se člověku těžko pamatuje a je velmi nečitelný, nehledě na to, že ladění je velice časově i logicky náročné. Z tohoto důvodu vznikly programovací jazyky, které značně usnadňují tvorbu aplikací. K překladu vyššího programovacího jazyka do strojového kódu slouží právě překladač. Jeho běžná organizace vypadá následovně:



Obrázek 5 - Obecná organizace překladače
Zdroj: vlastní

Průběh kompilace zdrojového kódu se provádí v několika na sebe navazujících krocích. Jak jednotlivé elementy překladače pracují, popíše následující postup, který odpovídá obrázku výše:

1. Vstupem celého překladače je zdrojový kód, ve kterém je aplikace napsána. Pro překladač to není nic jiného, než čistě proud za sebou jdoucích znaků, které mohou mít určitý význam. Tento proud znaků je čtený zleva doprava lexikálním analyzátorem, který vyhledává významové posloupnosti znaků tvořící jedno nebo více znakový symbol – lexém. Každému lexému je následně přiřazen tzv. token, což je entita jazyka s definovaným významem (může jím být literál, proměnná, klíčové slovo, symbol apod.). Lexikální analyzátor dále vynechává zbytečné znaky z hlediska programu, jako jsou mezery, komentáře apod.

2. Tokeny jsou lexikálním analyzátorem odeslány do syntaktického analyzátoru, kde jsou seskupovány do gramatických vět – frází. Syntaktický analyzátor poté posuzuje předložený zdrojový kód, zda je syntakticky správný. Kontroluje tak předem daná pravidla vybraného jazyka, pro které je překladač sestaven.

Takto opakovaně volá lexikální analyzátor, který mu vrací token po tokenu, dokud nevyhledá syntakticky špatný zápis. Pokud není nalezena syntaktická chyba, jako je například neukončení bloku metody, chybný zápis výrazu bez středníku konci řádku apod., je dodaný kód vyhodnocen jako syntakticky správný. Následuje vygenerování vnitřní reprezentace programu ve tvaru abstraktního syntaktického stromu a vybudování programu ve vnitřním jazyce.

3. Syntaktický analyzátor odešle do sémantického analyzátoru gramatické věty (konstrukce symbolů) ve formě abstraktního stromu. Sémantický analyzátor poté kontroluje typovou kontrolu. V případě, že najde element, který je nutný typově konvertovat, přidá do uzlu stromu konverzní funkci, která zajistí správný výsledný typ. Další funkcí, kterou provádí sémantický analyzátor, je odhalování nedeklarovaných proměnných v kódu. Abstraktní syntaktický strom se tak v sémantickém analyzátoru rozšíří o převodní funkce.

4. Generátor vnitřního (pomocného) kódu přijme rozšířený abstraktní syntaktický strom, který je následně přeložen do vnitřního kódu (tzv. intermediální kód). Intermediální kód je již nezávislý na cílovém jazyce. Je-li určený pro kompilační překladač, musí být snadno optimalizovatelný a snadno přepsatelný do strojového kódu nebo Assembleru. V případě intermediálního kódu určeného pro interpretační překladač, musí být snadno interpretovatelný, pokud možno bez úprav. Výstupem tohoto generátoru je tzv. vnitřní tvar programu – intermediální reprezentace, která je připravená pro zpracování v závěrečné syntetizační části překladače.

5. Další fází je optimalizace kódu, kterou provádí tzv. optimalizátor. Jeho hlavními úkoly je vylepšení paměťové náročnosti, nahrazení pomalých instrukcí rychlejšími a případně vhodný kompromis předcházejícího. Optimalizátor odstraňuje nadbytečné nebo nepotřebné příkazy. Na úrovni optimalizátoru dělíme dvě etapy optimalizace – tzv. Front End a Back End. Front end, který je první částí optimalizace, je zatím nezávislý na výstupním jazyce. Back end je již podřízen cílovému prostředí, čímž je například strojový kód procesoru, Assembler apod.

6. Po optimalizaci je nutné vytvořit z vnitřního kódu kód cílový, který již bude finální a naprosto závislý na vybrané architektuře. To má za úkol generátor cílového kódu. Důležitý je správný výběr cílového jazyka podle účelu použití:

    I. Jazyk symbolických adres (Assembler) se používá v případě existence kvalitního překladače Assembleru.

    II. Čistý strojový kód je vhodný ke generování kódu bez předpokladu existence operačního systému a knihoven. Tento strojový kód funguje nezávisle.

    III. Rozšířený strojový kód se používá pro konkrétní operační systém. Podmínkou pro správnou funkci je přístupnost k prostředkům operačního systému, jako je alokace, vstupně výstupní operace apod.

IV. Virtuální strojový kód se skládá z tzv. virtuálních instrukcí a vytváří tak přenositelný kód. Na počítači, kde bude překladač spuštěn, je nutné mít nainstalovaný virtuální stroj pro cílový jazyk.

7. Nakonec je vybraný cílový jazyk předán jako kompletní bez jakýchkoliv chyb a je ho možné zpracovat procesorem.

5 Grafické zobrazení UML

5.1 Definice modelovacího jazyka UML

Modelovací jazyk UML byl vytvořen za účelem vizualizace, specifikace, navrhování a tvorby dokumentace programových systémů. Důvodem vývoje UML bylo sjednotit různé metody a syntaxe návrhových modelů, které byly do té doby značně rozdílné. Stejná formální syntaxe umožňuje sdílení prací s ostatními návrháři a neomezuje tak rozsah použití tohoto modelovacího jazyka. UML usnadňuje návrh a vizualizaci různých typů aplikací, ať už se jedná o byznys procesy, konkrétní prvky jako jsou příkazy programovacího jazyka, databázová schémata či znovupoužitelné programové komponenty. Specifikace standardu UML 2.0 je rozdělena do čtyř základních částí:

- definice infrastruktury – definuje základní elementy, základní architekturu, jádro UML společné pro UML a související či podobné standardy,
- definice superstruktury - popis prvků metamodelu, definuje konstrukty používané uživateli UML – elementy diagramů a diagramy,
- definice výměnné struktury – pro export a import datagramů, formát pro výměnu dokumentů (diagramů) mezi různými nástroji,
- Object Constraint Language (OCL) verze 2.0 – jazyk pro formálně přesný popis různých omezení platných v modelu.

Metamodel pro UML 2.0 byl navržen tak, aby splňoval následující principy:

- modularitu – tento princip požaduje velkou soudržnost a malou souvztažnost při organizaci konstruktů do balíků a komponent metamodelu,
- vrstvení – UML je založen na čtyřvrstvé architektuře, kde nižší vrstvy využívají konstrukty z vyšších vrstev,
- dělení – uvnitř každé vrstvy jsou odděleny samostatné koncepty tak, že se současné prvky odlišují od budoucích prvků standardu,
- rozšiřitelnost – rozšíření stávajícího UML je umožněno dvěma způsoby:

1. definování pomocí profilů tak, aby se obecné definice přizpůsobily konkrétní platformě (např. J2EE, .NET/COM+ apod.) a konkrétní doméně aplikace (např. telekomunikace, letectví apod.)
2. nově vytvořený jazyk může být budován s využitím částí infrastruktury UML, které se doplní o potřebné metatřídy a metavztahy.

- znovupoužití – UML je navržen tak, aby byl znovupoužitelný i pro podobné metamodely (MOF, CWM apod.).

UML tedy umožňuje vytvořit kompletní přípravu zamýšleného projektu. Na začátku vývoje projektu stojí důsledná analýza, která je následně vyhodnocena a teprve pokud je budoucí majitel projektu spokojen s výsledky testování, přenese se fáze do konkrétního implementačního návrhu. Toto je obvyklý postup při tvorbě jakéhokoliv UML diagramu. Předchází se tak skrytým problémům, které by mohly později nastat při nedostatečném zkoumání.

5.2 Diagram tříd

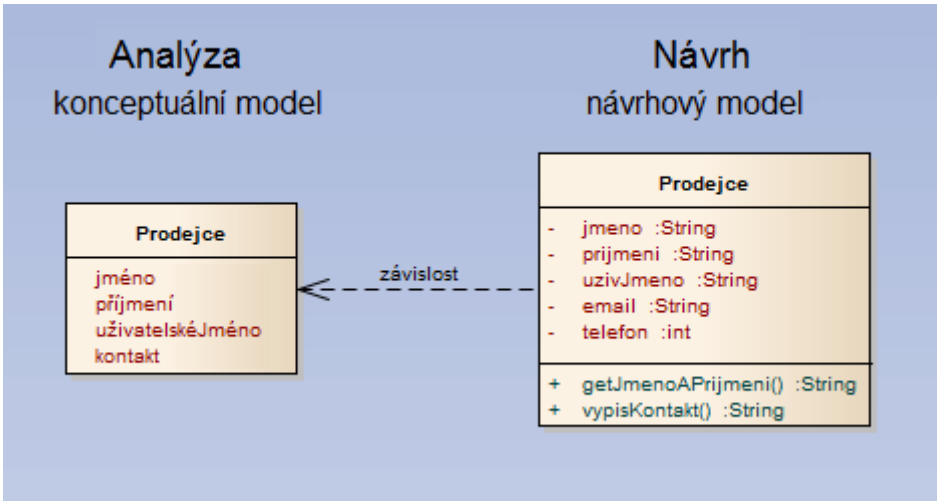
Jednou z možností grafického zobrazení modelovaného systému pomocí UML je diagram tříd (Class diagram), který představuje statický pohled na tento systém. Zobrazuje strukturu objektových tříd, jejich vzájemné vztahy a omezení. Návrh tříd, jejich odpovědností a následné vytvoření tohoto diagramu je základním krokem analýzy navrhovaného programového systému. Při tvorbě diagramu tříd je nutné rozlišit, zda potřebujeme vyjádřit požadavky na modelovaný software nebo podrobně popsat návrh daného řešení. Z toho důvodu se rozdělují tři úrovně modelu tříd – konceptuální, návrhová (designová) a implementační.

5.2.1 Konceptuální model

Konceptuální (doménový, analytický) model tříd je vytvářen za účelem analýzy požadavků na software bez implementačních detailů, a proto je takový model implementačně nezávislý. Uvedeny jsou obvykle pouze názvy klíčových atributů a některé klíčové metody. Je-li diagram tříd vytvářen pouze za účelem znázornění vztahů mezi třídami, neuvádí se ani atributy a metody.

5.2.2 Návrhový model

Návrhový model rozšiřuje konceptuální model o datové typy, přístupnost atributů, metod apod. Do modelu jsou také přidány třídy uživatelského rozhraní a třídy obsluhující systémové události. Ze závislosti mezi konceptuálním a návrhovým modelem plyne, že z jedné třídy konceptuálního modelu můžeme vytvořit několik tříd v návrhovém modelu. Mezi třídami těchto dvou modelů tedy existuje vztah typu závislost a tak lze považovat návrhový model za implementačně závislý, jelikož obsahuje některé implementační charakteristiky.



Obrázek 6 - Porovnání třídy konceptuálního a návrhového modelu tříd
Zdroj: vlastní

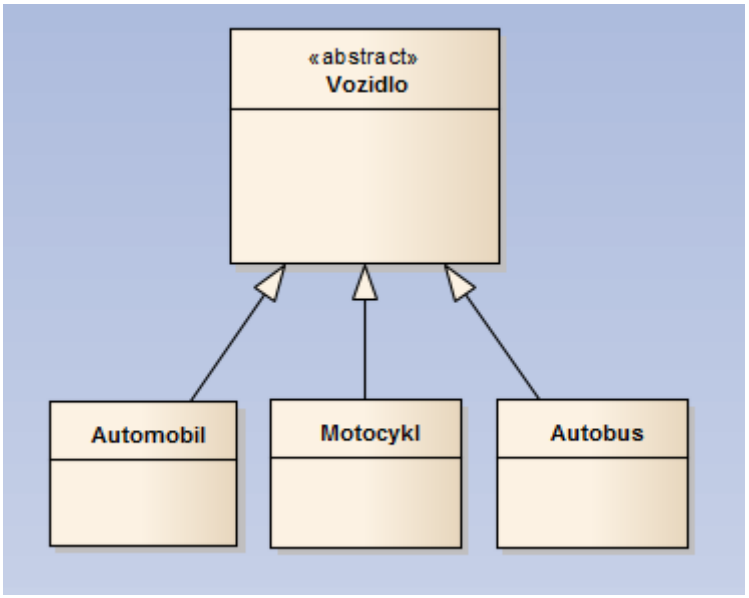
5.2.3 Implementační model

Tento model již obsahuje veškeré implementační charakteristiky (identifikátory, metody, vztahy mezi třídami apod.). Zde je již nutné pojmenovat identifikátory bez diakritiky, používat datové typy pro vybraný programovací jazyk apod. Z implementačního modelu lze takto vygenerovat platný zdrojový kód pro programovací jazyk.

Každý diagram tříd má své elementy, které tvoří jeden celek. Mezi tyto elementy patří třídy, vztahy mezi nimi, rozhraní, výčtové typy a balíčky. Jednotlivé elementy mají specifické grafické znázornění.

5.3 Abstraktní třída

Někdy se můžeme setkat se situací, kdy je výhodné vytvořit jedinou bázovou třídu pro více tříd odpovídajících konkrétním objektům, i když tato samotná bázová třída žádnému konkrétnímu objektu neodpovídá. Může však obsahovat některá data a poskytovat metody, které jsou odvozeným třídám společné. Takovou třídu nazýváme abstraktní a tvoříme ji pomocí klíčového slova *abstract*. Nad abstraktní třídou nelze vytvářet instance, mohou se vytvářet pouze instance konkrétních tříd, pro které je abstraktní třída vytvořena. Příklad abstraktní třídy je následující:



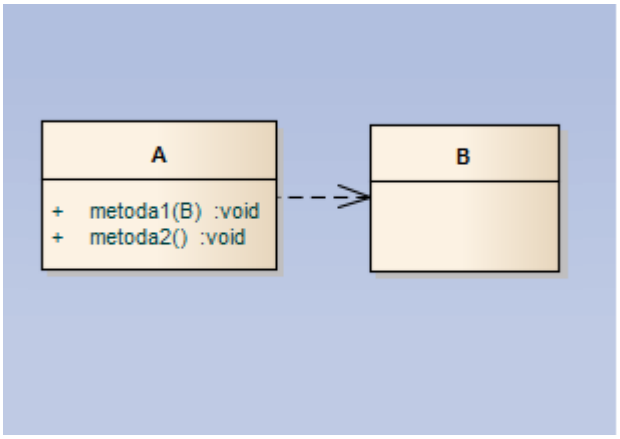
Obrázek 7 - Příklad abstraktní třídy
Zdroj: vlastní

Abstraktní třída *Vozidlo* představuje obecný dopravní prostředek, kterým může být *Automobil*, *Motocykl*, *Autobus* a mnoho dalších. Každý z těchto dopravních prostředků má některé vlastnosti nebo operace společné a jiné odlišné. Společné prvky jsou navrženy v abstraktní třídě a obsahují je všechny dopravní prostředky. Specifické vlastnosti či operace pak definuje konkrétní třída.

5.4 Vztah závislost

Obecným vztahem mezi třídami je vztah závislost (Dependency), který je nejslabším vztahem indikující vazbu jedné třídy na jiné. Znamená to, že třída, ze které šipka v UML

diagramu tříd vychází, má nějaký druh závislosti na třídě, kam šipka ukazuje. Na obrázku níže tedy třída *A* závisí na třídě *B*.




```
public class A {

    public void metoda1(B b) {
        // ...
    }

    public void metoda2() {
        B promennaB = new B();
        // ...
    }
}
```

Obrázek 8 - Vztah závislost
Zdroj: vlastní

Při tomto vztahu sice třída *A* používá třídu *B*, ale sama si nevytváří instanci třídy *B*. Závislostí je například použití třídy *B* jako parametru nebo lokální proměnné v některé z metod třídy *A*.

5.5 Vztahy mezi datovými typy

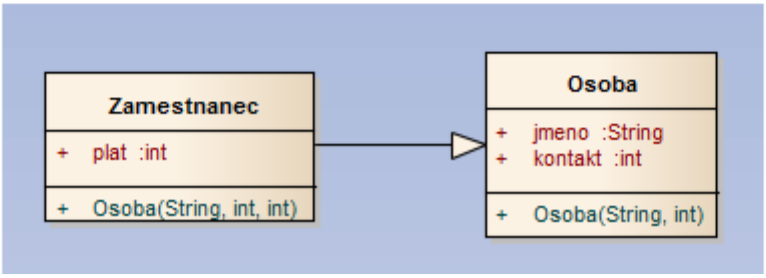
V objektově orientovaném programování se používá datových typů navrhnutých uživatelem – referenčních datových typů. Tyto datové typy, které většinou představují objekty reálného světa, si mezi sebou potřebují sdělovat informace a operace nad nimi. Aby to bylo možné, vytváří mezi sebou různé typy vztahů, které jim komunikaci umožní.

5.5.1 Vztah generalizace (specializace)

Vztah generalizace-specializace je vztah mezi dvěma objekty, kde první objekt je obecnějším případem (generalizací) druhého objektu. Druhý objekt je pak zvláštním případem (specializací) prvního objektu. Tento druh vztahu zavádí do objektově orientovaného návrhu tzv. dědičnost. Myšlenkou dědičnosti je fakt, že každá třída má svého rodiče, po kterém dědí veškeré atributy a metody. Dědičnost se v Javě deklaruje pomocí klíčového slova *extends*. Jazyk Java neumožňuje vícenásobnou dědičnost z důvodu přehlednosti programů, a tak lze vždy dědit pouze od jednoho rodiče.

Příkladem dědičnosti je následující návrh:

```
public class Osoba {

    public String jmeno;
    public int kontakt;

    public Osoba(String jmeno, int kontakt) {
        this.jmeno = jmeno;
        this.kontakt = kontakt;
    }
}

public class Zamestnanec extends Osoba {

    public int plat;

    public Zamestnanec(String jmeno, int kontakt, int plat) {
        super(jmeno, kontakt);
        this.plat = plat;
    }
}
```



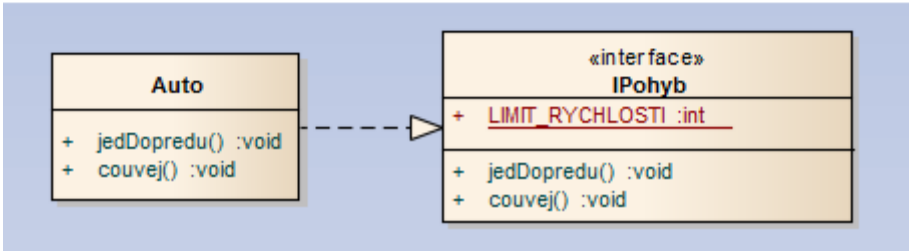
Obrázek 9 - Příklad dědičnosti
Zdroj: vlastní

Máme dvě třídy, kde třída *Zamestnanec* dědí od třídy *Osoba* veškeré vlastnosti. Třída *Zamestnanec* navíc obsahuje atribut plat. Vidíme, že třída *Zamestnanec* využívá konstruktoru svého předka pro nastavení atributů pomocí klíčového slova *super*, které tuto funkci umožňuje. Poté se dodatečně nastaví atribut *plat*. Výhodou dědění je použití již existující třídy, kterou rozšíříme podle potřeby o další atributy či operace. Toto rozšíření upřesňuje definici třídy potomka. U generalizace se používá pomocného slova „je“, kde v tomto případě *Zamestnanec* je *Osobou*.

5.5.2 Vztah realizace

Vztah mezi třídou a rozhraním se nazývá realizace. Používá se v momentě, kdy si nejsou některé třídy příbuzné k určitému rodiči, ale i tak mají chování operací stejné. Rozhraní neboli *interface* se nejběžněji vytváří jako skupina konstant a souvisejících metod bez těla předepisující třídě operace, které musí implementovat. Pro implementující třídu ale není důležité, jakým způsobem budou tyto operace navrženy. Pro vztah realizace se používá klíčového slova *implements*. Příkladem rozhraní je následující implementace:

```
public class Auto implements IPohyb {

    @Override
    public void jedDopredu() {
        System.out.println("Jedu rychlostí "
                + (LIMIT_RYCHLOSTI - 10));
    }

    @Override
    public void couvej() {
        System.out.println("Couvám rychlostí "
                + (LIMIT_RYCHLOSTI / 10));
    }
}
public interface IPohyb {

    public static final int LIMIT_RYCHLOSTI = 50;

    public void jedDopredu();

    public void couvej();
}
```



Obrázek 10 – Vztah realizace
Zdroj: vlastní

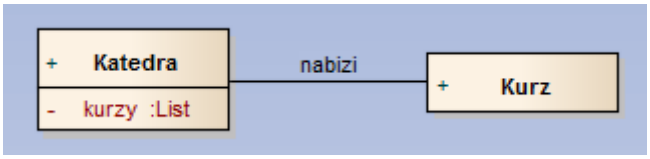
Na obrázku máme třídu *Auto*, která implementuje rozhraní *IPohyb*. Toto rozhraní disponuje jedním atributem a dvěma metodami. Třída *Auto* tedy musí obsahovat metody *jedDopredu( ), couvej( )* a atribut *plat*, což splňuje a vytváří tak realizaci mezi třídou *Auto* a rozhraním *IPohyb*.

5.6 Vztahy mezi instancemi tříd

Instance třídy není nic jiného, než objekt vytvořený na základě určitého předpisu – třídy. Aby mezi sebou mohli objekty komunikovat, musí být spolu svázány pomocí spojení. Existuje několik druhů spojení neboli vztahů, které se liší sílou závislosti objektů, která mezi nimi vzniká.

5.6.1 Vztah asociace

Asociace (Association) představuje vztah mezi instancemi dvou a více tříd. Typickým příkladem je možnost jedné instance poslat zprávu druhé instanci. Děje se tak obvykle pomocí referenčních proměnných.

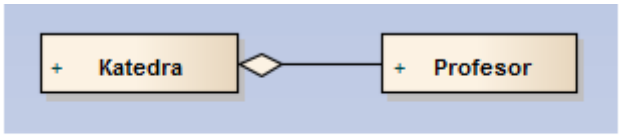


Obrázek 11 - Vztah asociace
Zdroj: vlastní

Asociace se znázorňuje plnou čarou a v případě, že se jedná o jednosměrnou asociaci, kdy o vztahu ví pouze jedna třída, používá se jednoduchá šipka.

5.6.2 Vztah agregace

Agregace (Aggregation) je silnějším vztahem než asociace. Majitel, který se nachází na vrcholu, představuje celek, který má pod sebou komponenty jeho třídy. Zánik majitele zde nemusí nutně zapříčinit zánik odkazovaného objektu.



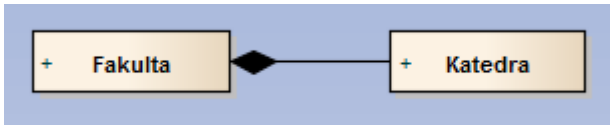

```
public class Profesor {
    // ...
}

public class Katedra {
    private List<Profesor> seznamProfesoru;
    // ...
}
```

Obrázek 12 - Vztah agregace
Zdroj: vlastní

U agregace se používá pomůcky v podobě slova „má“. V uvedeném případě výše platí, že *Katedra* má *Profesory*.

5.6.3 Vztah kompozice

Kompozice (Composition) je vztahem, který má nejsilnější vazbu mezi instancemi tříd. U agregace nebylo nutnou podmínkou, aby zánikem majitele zanikl i odkazovaný objekt. Zde je to přesně naopak. Odkazovaný objekt je nedělitelnou součástí majitele a existence tohoto objektu bez majitele nemá smysl. *Katedra* bez *Fakulty* fungovat nemůže.



Obrázek 13 - Vztah kompozice
Zdroj: vlastní

5.7 Modifikátory přístupu

Každé proměnné, metodě či třídě lze nastavit viditelnost vůči ostatním objektům. Slouží k tomu tzv. modifikátory, kterými je možné omezit přístup ostatních objektů. Pokud proměnnou nebo metodu nastavíme jako skrytou, není ji možné číst, zapisovat a volat. Seznam modifikátorů, které Java nabízí je následující:

Název modifikátoru	Třída	Balíček	Potomek	Okolní svět
public	ano	ano	ano	ano
protected	ano	ano	ano	ne
bez modifikátoru	ano	ano	ne	ne
private	ano	ne	ne	ne

Obrázek 14 - Modifikátory přístupu
Zdroj: vlastní

5.8 Kvalifikátory

Kvalifikátory upravují vlastnosti vytvořeného objektu. Umožňují tak ovlivnit chování objektů z několika hledisek.

5.8.1 Kvantifikátor final

Prvním z popisovaných kvalifikátorů je *final*. Má různé významy pro jednotlivé objekty:

- pro třídu – znemožnění třídě mít potomky,
- pro metodu – metodu nelze dále upravovat v podtřídách,
- pro atribut – atributu lze přiřadit hodnotu pouze jednou,
- pro proměnnou – proměnné lze přiřadit hodnotu pouze jednou,
- pro parametr metody – parametr nemůže být uvnitř metody změněn.

5.8.2 Kvalifikátor volatile

Atribut s tímto kvalifikátorem umožňuje modifikaci více vlákny současně a tak nesmí být uchováván ve vyrovnávací paměti (cache).

5.8.3 Kvalifikátor transient

Kvalifikátor *transient* se používá u atributu a zajišťuje, že takový atribut nebude uložen při serializaci. Serializací je myšleno ukládání objektů do formátu, který může být uložen nebo poslán po síti.

5.8.4 Kvalifikátor synchronized

Pomocí kvalifikátoru *synchronized* je možné ovlivnit přistupování vláken ke kritické sekci. V bloku kódu zajistí *synchronized* přístupnost pouze jednoho vlákna v jednom momentě (mutex). U metody je použití téměř shodné, jelikož do metody může najednou vstoupit pouze jedno vlákno. Tímto způsobem je možné využít více vláknového návrhu aplikace,

kdy je nutné ošetřit použití sdíleného prostředku tak, aby nedocházelo k vícenásobnému přístupu v jednom momentě.

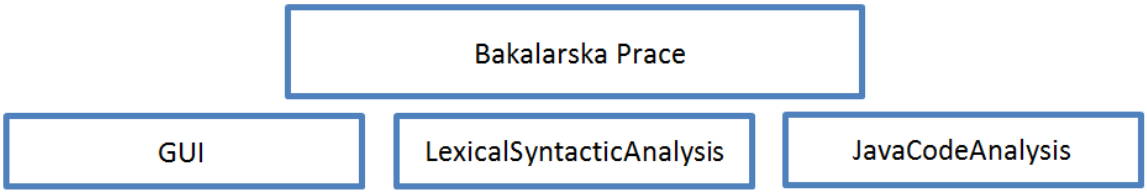
5.8.5 Kvalifikátor native

Kvalifikátorem *native* je možné označit metody, které mohou být implementovány v jiném jazyce než Java. Podmínkou použití je implementování rozhraní Java Native Interface (JNI). Toto rozhraní umožňuje propojit kód běžící na virtuálním stroji Javy s nativními programy a knihovnami v jiných jazycích jako je například C, C++ apod.

6 Implementace aplikace

6.1 Rozvržení aplikace

Celý projekt je logicky rozdělen do tří částí, které zpracovávají jim přidělenou úlohu. Část, která se nazývá *GUI*, má na starost zobrazení grafického prostředí pro komunikaci s uživatelem. *LexicalSyntacticAnalysis* zpracovává načtený projekt, zkoumá ho lexikální a poté syntaktickou analýzou na základě *JavaCodeAnalysis*, kde jsou předloženy pravidla programovacího jazyka Java.



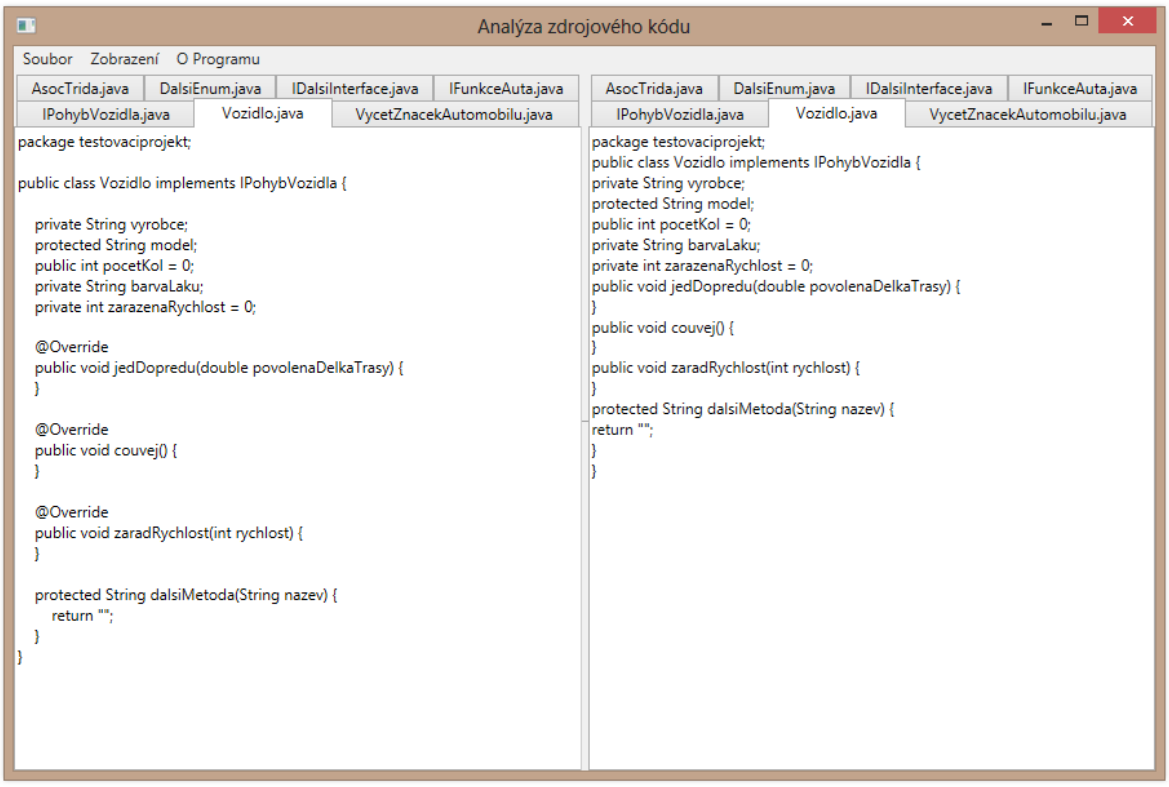
Obrázek 15 - Rozvržení aplikace
Zdroj: vlastní

Bakalarska Prace je ve vývojovém prostředí Microsoft Visual Studia jako tzv. řešení, které obsahuje tři projekty, jimiž jsou právě *GUI*, *LexicalSyntacticAnalysis* a *JavaCodeAnalysis*. Jednotlivé projekty budou podrobněji popsány v následujících podkapitolách.

6.2 Projekt GUI

Tento projekt se skládá celkem ze čtyř hlavních složek – *MainWindow*, *UmlWindow*, *XmlWindow* a *AboutWindow*. Každá z nich představuje jiné grafické okno.

MainWindow zobrazuje hlavní okno, ve kterém má uživatel přístupné veškeré možnosti aplikace, jako je načtení Java souborů, vytvoření nového projektu, zobrazení UML diagramu tříd či výpis projektu ve formátu XML. *MainWindow* je základní konstrukcí celé aplikace. Toto hlavní okno je vertikálně rozděleno na dvě části, kde levá část zobrazuje v záložkách všechny aktuálně načtené soubory tak, jak jsou načteny programu a pravá část vypisuje již jednotlivé lexémy seskupené do gramatických vět, které vznikly po analýzách projektu.



Obrázek 16 - Hlavní okno aplikace
Zdroj: vlastní

Úkolem *UmlWindow* je zobrazit UML diagram tříd. Na základě elementů (proměnná, třída, metoda apod.), které jsou v třídě *UmlWin* ve složce *UmlWindow* zpracovány, dojde k vykreslení UML diagramu tříd. Jednotlivé elementy jsou vykresleny spolu se vztahy mezi nimi. *UmlWindow* dále umožňuje změnu velikosti zobrazených elementů tak, aby si mohl uživatel diagram tříd uspořádat podle své volby. S jednotlivými grafickými elementy lze pohybovat v rámci nadřazeného grafického elementu. Při pohybu nadřazeného elementu se zároveň pohybují i jeho všechny podřízené grafické elementy (například vnitřní třída). Vztahy, které existují mezi třídami, se vždy vykreslí mezi nejbližšími hranami obou tříd, kde vztah vzniká.

XmlWindow vytvoří nové okno, kde zobrazí načtený soubor ve tvaru značkovacího jazyka XML. Vytvořenou stromovou strukturu lze libovolně rozbalovat a sbalovat do jednotlivých elementů, jako je balíček, třída, metoda, proměnná apod. Tvar XML je možné uložit do textového souboru pro pozdější použití.

Složka *AboutWindow* zobrazuje dialogového okno obsahující základní údaje o bakalářské práci.

6.3 Projekt LexicalSyntacticAnalysis

Projekt, který se zabývá lexikální a syntaktickou analýzou, je nazván *LexicalSyntacticAnalysis*. Je rozdělen do tří složek – *LexicalAnalysis*, *SyntacticAnalysis* a *ObjectsInterfaces*.

LexicalAnalysis provádí lexikální analýzu na základě načteného souboru. Na každý lexém je volána metoda, která aplikuje pravidla vztahující se na daný element. Pozná tedy,

zda je lexémem entita programovacího jazyka, pro který je analyzátor vytvořen. Lexikální analýza dále vyhledává závislosti mezi třídami a objekty, které jsou následně využity pro zobrazení diagramu tříd, a odstraňuje komentáře, které nejsou z hlediska analýzy potřebné.

SyntacticAnalysis aplikuje na programový kód pravidla, která jsou vytvořena v *JavaCodeAnalysis*. Analyzátor tyto pravidla používá abstraktně a tím není implementačně závislý pouze na programovacím jazyce Java a po dodání pravidel je možné tento analyzátor aplikovat i na jiný programovací jazyk.

Složka *ObjectsInterfaces* pouze definuje rozhraní, které musí implementovat jednotlivé elementy zpracovávané v *JavaCodeAnalysis*.

6.4 Projekt JavaCodeAnalysis

Poslední částí řešení bakalářské práce je projekt *JavaCodeAnalysis*. Tento projekt obsahuje implementaci pravidel na cílový hostující jazyk, který bude analyzován, tedy Javu. *JavaCodeAnalysis* obsahuje výčet všech klíčových slov jazyka Java. Na základě těchto klíčových slov provádí analýzu kódu a elementy, které projdou lexikální analýzou, jsou uloženy do kolekce, která je následně využita k dalšímu zpracování.

7 Závěr

Cílem bakalářské práce byl návrh aplikace, který umožní lexikální a syntaktickou analýzu programového kódu jazyka Java. Nedílnou součástí bylo využití analyzovaných dat k zobrazení UML diagramu tříd a také uložení stromové struktury kódu programu do tvaru XML, který umožňuje uložení do textového souboru.

Aplikace byla vytvořena v programovacím jazyce C# ve vývojovém prostředí Microsoft Visual Studio 2012 za použití technologií .NET Frameworku 4.5.

Při vývoji programu jsem vycházel ze znalostí získaných během studia a to především z předmětů Základy programování, Počítačová grafika, Datové struktury, Programovací techniky v jazyce Java a Objektově orientované programování. Díky znalostem z těchto předmětů jsem byl schopen vytvořit kvalitní návrh aplikace jako celku. Aplikace splňuje veškeré požadavky, které byly vedoucím práce zadány.

Mezi možná rozšíření, která by mohla být v budoucnosti do této aplikace zakomponována, bych navrhl implementaci pravidel pro další programovací jazyky, jelikož je návrh aplikace postaven tak, že stačí pro nový jazyk pouze doplnit syntaktická a lexikální pravidla v projektu *CodeAnalysis*.

Vypracování této kvalifikační práce mi bylo velkým přínosem. Nejen z hlediska získání nových zkušeností v oblasti programování, ale také prohloubení dosavadních znalostí převážně z objektově orientovaného programování.

Literatura

HEROUT, Pavel. 2001. *Učebnice jazyka Java.* České Budějovice : Kopp nakladatelství, 2001, ISBN 978-80-7232-398-2.

MOLNÁR, Ľudovít, ČEŠKA, Milan, MELICHAR, Bořivoj. 1987. *Gramatiky a jazyky.* Bratislava : ALFA, 1987.

HOPCROFT, John E., MOTWANI, Rajeev, ULLMAN, Jeffrey D. 2006. *Introduction to Automata Theory, Langugages, and Computation (3rd Edition).* New Jersey : Prentice Hall Press, 2006. ISBN 978-0321455369.

SHARP, John. 2012. *Microsoft Visual C# 2010 Krok za krokem.* Brno : Computer Press, 2012. ISBN 978-80-251-3147-3.

PETZOLD, Charles. 2008. *Mistrovství ve Windows Presentation Foundation.* Brno : Computer Press, 2008. ISBN 978-80-251-2141-2.

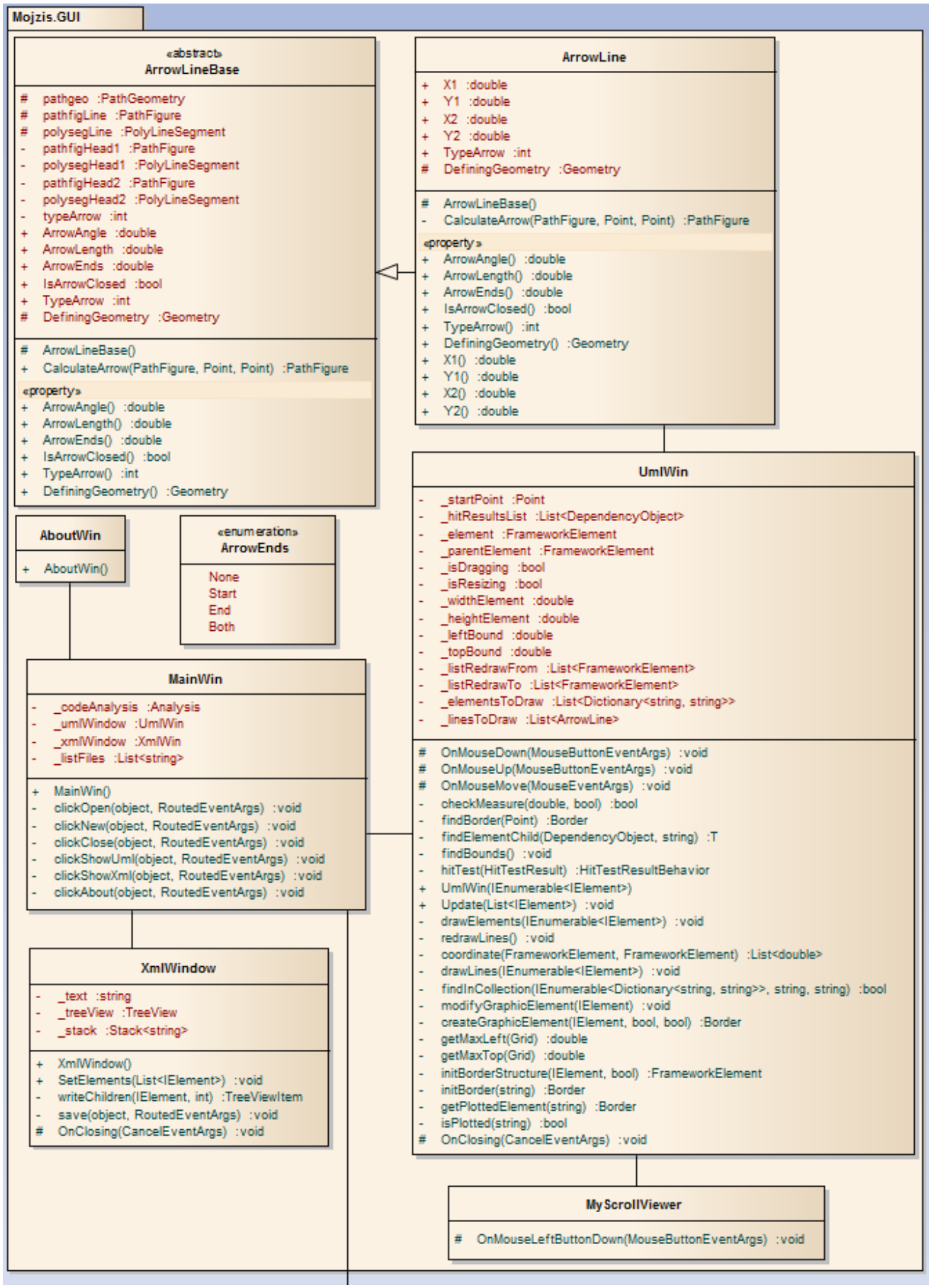
KANISOVÁ, Hana, MÜLLER Miroslav. 2004. *UML srozumitelně.* Brno : Computer Press, 2004. ISBN 80-251-0231-9.

PECINOVSKÝ, Rudolf. 2008. *Myslíme objektově v jazyku Java.* Praha : Grada Publishing, 2008. ISBN 978-80-247-2653-3.

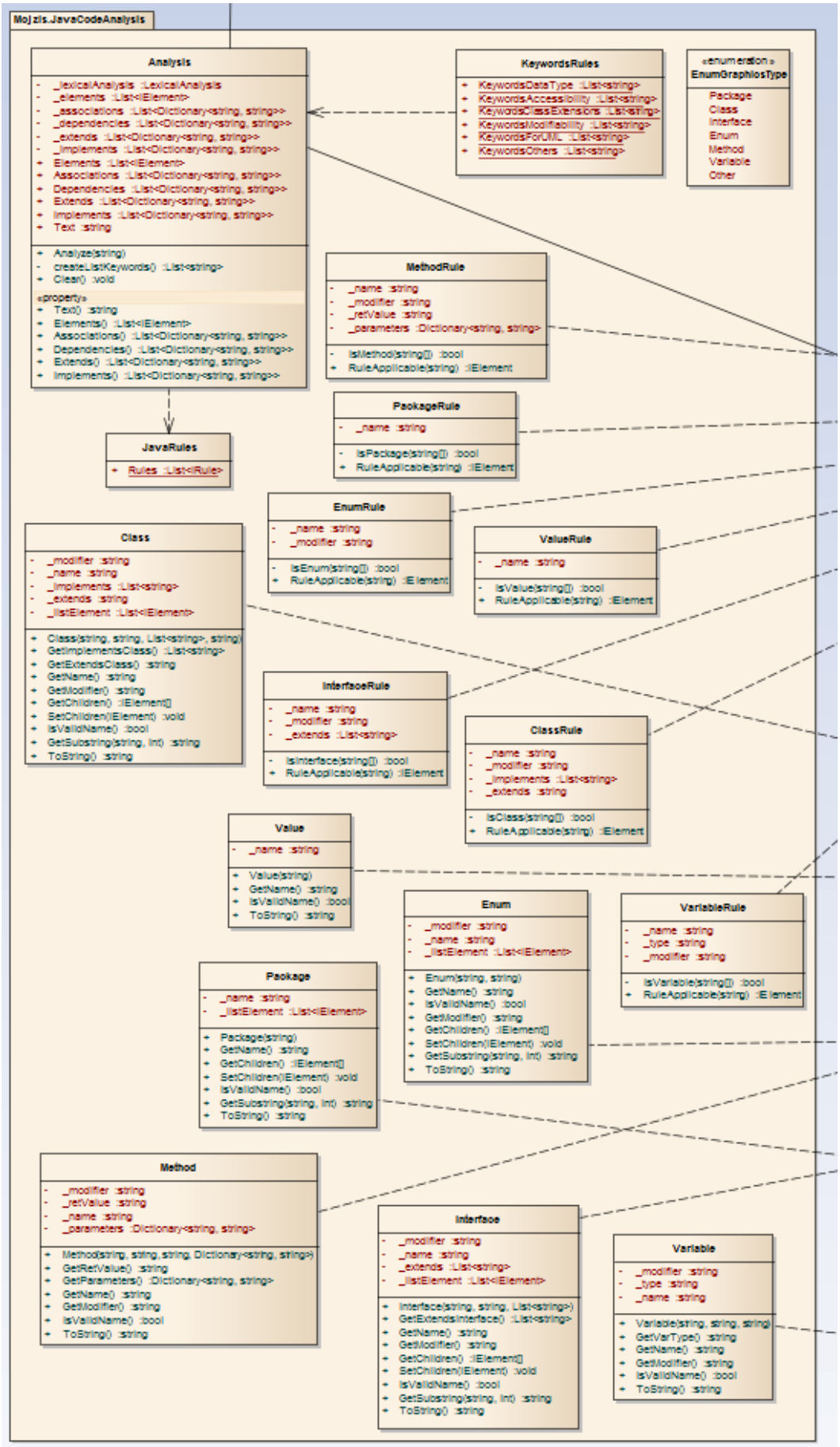
ZAKHOUR, Sharon, HOMMEL, Scott, ROYAL, Jacob, RABINOVITCH, Isaac, RISSER, Tom, HOEBER, Mark. 2007. *Java 6 Výukový kurz.* Brno : Computer Press, 2007. ISBN 978-80-251-1575-6.

GOSLING, James, JOY, Bill, STEELE, Guy, BRACHA, Gilad, BUCKLEY, Alex. 2013. *The Java® Language Specification Java SE 7 Editon.* Redwood City : Oracle America, Inc., 2013 [cit. 11. 4. 2013]. Dostupné na: <http://docs.oracle.com/javase/specs/jls/se7/html/index.html>.

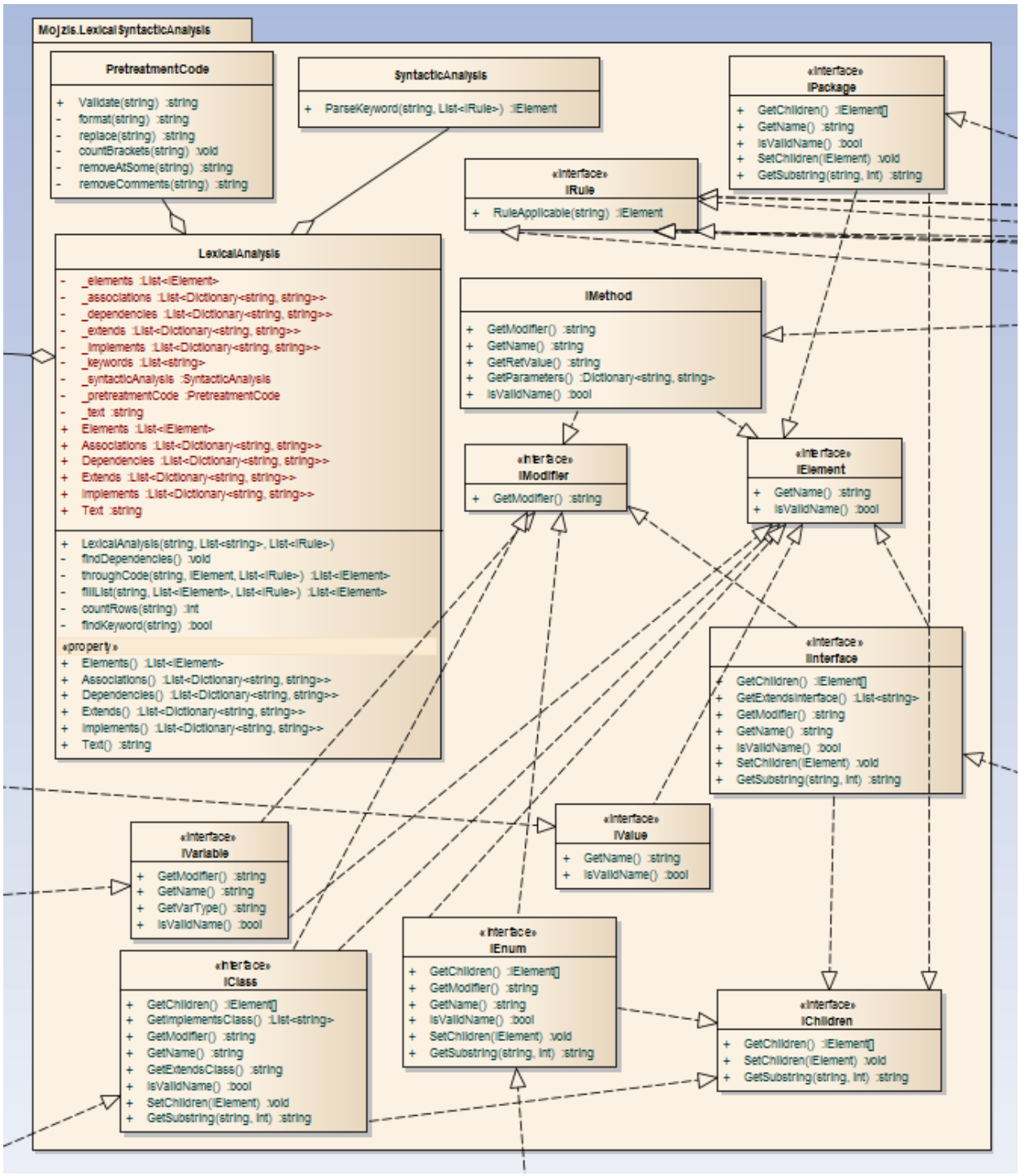
Příloha A – Diagram tříd Mojzis.GUI



Příloha A – Diagram tříd Mojzis.JavaCodeAnalysis



Příloha A – Diagram tříd Mojzis.LexicalSyntacticAnalysis



Příloha B – SyntacticAnalysis – syntaktická analýza

```
using System.Collections.Generic;
using System.Linq;

namespace Mojzis.CoreAnalysis
{
    class SyntacticAnalysis
    {
        /// <summary>
        /// Pouzije pravidla a vrati vytvoreny element
        /// </summary>
        /// <param name="command">Text pro parsovani</param>
        /// <param name="rules">Pravidla parsovani</param>
        /// <returns></returns>
        public IElement Parse(string command, List<IRule> rules)
        {
            for (int i = 0; i < rules.Count(); i++)
            {
                var element = rules[i].RuleApplicable(command);
                if (element != null)
                    return element;
            }
            return null;
        }
    }
}
```

Příloha C – ClassRule – pravidla pro třídy

```
/// <summary>
/// Pravidla pro tridy
/// </summary>
public class ClassRule : IRule
{
    private string _name;
    private string _modifier;
    private List<string> _implements = new List<string>();
    private string _extends;

    private bool isClass(string[] texts)
    {
        try
        {
            int index = 0;

            if (KeywordsRules.KeywordsAccessibility.Contains(texts[index]))
            {
                _modifier = texts[index];
                index++;
            }

            if (KeywordsRules.KeywordsModifiability.Contains(texts[index]))
                index++;

            if (KeywordsRules.KeywordsForUML[3] != texts[index])
                return false;
            index++;

            _name = texts[index];
            index++;

            if (KeywordsRules.KeywordsClassExtensions[0] == texts[index] &&
                KeywordsRules.KeywordsClassExtensions[1] == texts[index + 2])
            {
                index++;
                _extends = texts[index];

                index += 2;
                while (true)
                {
                    string implements = texts[index].Replace(",", "");
                    _implements.Add(implements);
                    index++;
                    if (texts[index] == "{")
                        return true;
                }
            }

            if (KeywordsRules.KeywordsClassExtensions[0] == texts[index])
            {
                index++;
                _extends = texts[index];
            }

            else if (KeywordsRules.KeywordsClassExtensions[1] == texts[index])
            {
                index++;
                while (true)
                {
                    string implements = texts[index].Replace(",", "");
```

```
                    _implements.Add(implements);
                    index++;
                    if (texts[index] == "{")
                        return true;
                }
            }
            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }

    public IElement RuleApplicable(string text)
    {
        _name = "";
        _modifier = "";
        _extends = "";
        _implements = new List<string>();

        var words = new List<string>(text.Split(' '));
        var temp = new List<string>(words);
        foreach (var word in temp)
            if (word == "")
                words.Remove(word);

        if (isClass(words.ToArray()))
        {
            IElement classElement = new Class(_modifier, _name, _implements,
_extends);
            return classElement;
        }
        return null;
    }
}
```

Příloha D – Metoda na skládání gramatických vět

```
/// <summary>
/// Rozdeli kod po slovech a vytvari gramaticke vety
/// </summary>
/// <param name="text">Kod</param>
/// <returns>Upraveny kod bez mezer</returns>
private string format(string text)
{
    var arrayTexts = text.Split(' ');
    var listTexts = new List<string>();
    foreach (var arrayText in arrayTexts)
        listTexts.AddRange(arrayText.Split('\n'));

    string output = "";
    string row = "";
    foreach (var arrayText in listTexts)
    {
        if (arrayText == "")
            continue;
        row += arrayText + " ";
        if (arrayText[arrayText.Count() - 1] == ';' ||
            arrayText[arrayText.Count() - 1] == '{')
        {
            if (row.Count() > 3)
            {
                if (row[row.Count() - 3] == ' ' &&
                    arrayText[arrayText.Count() - 1] == ';')
                    row = row.Remove(row.Count() - 3, 1);
            }
            output += row + "\n";
            row = "";
        }
        if (arrayText[arrayText.Count() - 1] == '}')
        {
            if (row.Count() > 2)
            {
                row = row.Insert(row.Count() - 2, "\n");
            }
            output += row + "\n";
            row = "";
        }
    }
    return output;
}
```

Příloha E – Metoda na odstranění komentářů

```
/// <summary>
/// Odstrani komentare
/// </summary>
/// <param name="text">Kod</param>
/// <returns>Upraveny kod</returns>
private string removeComments(string text)
{
    bool isCorrect = false;
    while (!isCorrect)
    {
        int startIndex = -1;
        int endIndex = -1;
        for (int i = 0; i < text.Count() - 1; i++)
        {
            if (text[i] == '/' && text[i + 1] == '*')
                startIndex = i;
            if (text[i] == '*' && text[i + 1] == '/')
            {
                if (startIndex == -1)
                    throw new ValidateFailed();
                endIndex = i;
                text = text.Remove(startIndex, i - startIndex + 2);
                break;
            }
        }
        if (startIndex == -1)
            isCorrect = true;
        if(startIndex != -1 && endIndex == -1)
            throw new ValidateFailed();
    }

    isCorrect = false;
    while (!isCorrect)
    {
        int startIndex = -1;
        int endIndex = -1;

        for (int i = 0; i < text.Count() - 1; i++)
        {
            if (text[i] == '/' && text[i + 1] == '/')
                startIndex = i;
            if (text[i] == '\n')
            {
                if (startIndex != -1)
                {
                    endIndex = i;
                    text = text.Remove(startIndex, i - startIndex);
                    break;
                }
            }
        }
        if (startIndex == -1)
            isCorrect = true;
        if (startIndex != -1 && endIndex == -1)
            throw new ValidateFailed();
    }
    return text;
}
```