Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Analýza kódu s využitím .NET Compiler Platform

Bc. Filip Růžička

© 2018 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Filip Růžička

Informatika

Název práce

Analýza kódu s využitím .NET Compiler Platform

Název anglicky

Code analysis using .NET Compiler Platform

Cíle práce

Tato diplomová práce se zaměřuje na formální popis a následné praktické využití sady open source překladačů a aplikačního programového rozhraní pro analýzu kódu určeného pro jazyky C# a Visual Basic firmy Microsoft, které je soustředěno a implementováno v knihovnách .NET Compiler Platform.

Hlavním cílem práce je vytvoření aplikace využitelné pro podporu výuky objektově orientovaného programování v jazyce C#. V rámci implementace aplikace se budou dále demonstrovat vybrané metody lexikální, syntaktické a sémantické analýzy, refaktoringu a emitování dynamických sestavení souvisejících s .NET Compiler Platform.

Metodika

V teoretické části práce jsou popsána teoretická východiska z oblasti softwarového inženýrství s primárním zaměřením na oblast objektového návrhu software.

Na základě těchto východisek je v praktické části práce implementována jednoduchá aplikace pro grafický návrh modelu tříd, jeho převod do zdrojového kódu jazyka C# a následnou analýzu toho kódu s využitím knihoven .NET Compiler Platform. V závěru bude zhodnocen přínos open source prostředků pro analýzu kódu a případné využití demonstrační aplikace jako didaktické pomůcky pro výuku objektově-orientovaných modelovacích technik.

Doporučený rozsah práce

60-80 stran

Klíčová slova

.NET, C#, compiler, Roslyn, UML, open source, forward engineering

---

Doporučené zdroje informací

Compilers: Principles, Techniques and Tools – Alfred V. Aho, Ravi Sethi, Jeffrey Ullman, Monica S. Lam
Graph Drawing – Patrick Healy, Nikola S. Nikolov. 2005 University of Limerick Ireland. ISBN-10: 3-540-31425-3
.NET Windows Form in a Nutshell – Ian Griffiths, Matthew Adams. 2003 O'Reilly & Associates, Inc. ISBN: 0-596-00338-2-
Objects First in Java: Practical Introduction Using BlueJ – David J. Barnes. 2016 Pearson Education Limited. ISBN: 1292159081, 9781292159089
Open source software ve veřejné správě a veřejném sektoru – Bohumír Štědroň. 2009 Grada Publishing Praha. ISBN: 978-80-247-3047-9
Roslyn Succinctly – Alessandro Del Sole. 2016 Syncfusion Inc. Morrisville, NC 27560. ISBN: 9781542827102
Tvorba open source software – Karl Fogel. 2005 CZ.NIC Praha. ISBN: 978-80-904248-5-2
UML 2 a unifikovaný proces vývoje aplikací – Jim Arlow, Ila Neustadt. 2007 Computer Press a.s. Brno. ISBN: 978-80-251-1503-9

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 23. 2. 2018

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 23. 2. 2018

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 26. 02. 2018

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Analýza kódu s využitím .NET Compiler Platform" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 29.3.2018 ______________________________

Poděkování

Rád bych touto cestou poděkoval vedoucímu diplomové práce Ing. Jiřímu Brožkovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování této práce.

Analýza kódu s využitím .NET Compiler Platform

Abstrakt

S příchodem nové verze integrovaného vývojového prostředí Microsoft Visual Studio 2010 se začala psát nová kapitola pohledu na svět otevřeného softwaru očima technologického giganta a lídra v oblasti vývoje programového vybavení nejen osobních počítačů - společnosti Microsoft. V říjnu 2011 se ve formě rozšiřujícího balíčku objevuje první verze dlouhodobě připravovaného projektu .NET Compiler Platform s kódovým označením Roslyn. Tento projekt mění dosavadní vnímání překladače jazyků C# a Visual Basic .NET jako černé skříňky a poskytuje sadu otevřených modulů, které prostřednictvím API nabízejí lexikální a sémantickou analýzu kódu, refaktoring a emitování IL. Tato diplomová práce se zabývá využitím funkcí takového aplikačního programového rozhraní, a to v podobě jednoduchého CASE nástroje, který umožňuje vytvářet nejen grafický objektový návrh dle konvencí modelovacího jazyka UML, ale především disponuje technikami forward engineeringu pro generování zdrojového kódu v jazyce C#, jeho analýze, překladu a testování. Tuto aplikaci je možno využít jak pro rychlé prototypování a testování skeletu různých programových modulů, tak pro didaktiku v oblasti objektově modelovacích technik s přesahem do programování v jazyce C#.

Klíčová slova: .NET, C#, compiler, Roslyn, UML, open source, forward engineering

Code analysis using .NET Compiler Platform

Abstract

With advent of new version of the Microsoft Visual Studio integrated development environment, a new chapter began to be written looking at the world of open software with the eyes of the technology giant and leader in domain of development software equipment not just for personal computers – Microsoft company. In October 2011, the first version of the long-awaited project .NET Compiler Platform, codenamed Roslyn, appears in the form of an expansion pack. This project changes the existing perception of the C# and Visual Basic .NET compiler as black boxes and provides a set of open modules, that allows a lexical and semantic code analysis, refactoring, and IL emission through the API. This master thesis deals with the use of functions of this application software interface, in the form of a simple CASE tool that allows not just graphic object design according to UML conventions, but it also has forward engineering techniques for generating of source code in language C#, its analysis, compilation and testing. This application can be used for both quick prototyping and testing of various software modules skeleton, as well as for didactics in area of object-modeling techniques with overlapping in software development in C# language.

Keywords: .NET, C#, compiler, Roslyn, UML, open source, forward engineering

Obsah

1 Úvod **11**

2 Cíl práce a metodika **13**

2.1 Cíl práce 13

2.2 Metodika 13

3 Teoretická východiska **14**

3.1 Principy a techniky kompilátorů 14

3.1.1 Definice překladače 14

3.1.2 Klasifikace překladačů 15

3.2 Struktura překladače 21

3.2.1 Lexikální analyzátor 22

3.2.2 Syntaktický analyzátor 23

3.2.3 Sémantická analýza 24

3.2.4 Překlad a optimalizace kódu 24

3.3 .NET Compiler Platform (Roslyn) 26

3.3.1 Architektura .NET Compiler Platform 26

3.3.2 Compiler API 27

3.3.3 Workspaces API 30

3.4 Diagram tříd v jazyce UML 32

3.4.1 Elementy v diagramu tříd 32

3.4.2 Vztahy v diagramu tříd 35

4 Vlastní práce **38**

4.1 Uživatelské rozhraní 38

4.1.1 Základní ovládací prvky 42

4.2 Diagram tříd 48

4.2.1 Elementy Třída, Rozhraní a Výčet 48

4.2.2 Relace v diagramu tříd 52

4.2.3 Generování zdrojového kódu 53

4.3 Analýza kódu 59

4.3.1 SyntaxTree API 60

4.3.2 Workspaces API 74

4.3.3 Emit API 81

4.3.4 Scripting API 86

5 Výsledky a diskuse **91**

5.1 Realizace diagramu tříd 91

5.1.1 Katalog změn v komponentě NClass 93

5.1.2 Vytváření instancí z diagramu tříd 93

5.1.3 Generování zdrojového kódu 94
5.2 Použití komponenty TreeViewAdv 95
5.3 Testování aplikace 96

6 Závěr 97

7 Seznam použitých zdrojů 99

8 Přílohy 100

Seznam obrázků

Obrázek 1-Zobrazení z množiny zdrojových jazyků na množinu cílových jazyků 15
Obrázek 2- Překladač jako "černá skříňka" 16
Obrázek 3- Interpretované zpracování zdrojového kódu a vstupních dat 16
Obrázek 4- Hybridní kompilátor..17
Obrázek 5- Základní části překladače - Front-end a Back-end 21
Obrázek 6- Dekompozice překladače na jednotlivé fáze 22
Obrázek 7- Syntaktický strom příkazu if z Př.1 24
Obrázek 8- Architektura .NET Compiler Platform..27
Obrázek 9- Tradiční kompilační pipeline a API překladače Roslyn.....................................28
Obrázek 10- Hierarchické uspořádání řešení, projektů a dokumentů 31
Obrázek 11- Třída s UML notací (vlevo) a třída z aplikace dotCORN (vpravo) 34
Obrázek 12- Role a multiplicita na vazbě typu kompozice..36
Obrázek 13- Ovládací prvky na záložce "Diagram tříd" 42
Obrázek 14- Ovládací prvky na záložce "Analýza kódu" 45
Obrázek 15- Editace záhlaví třídy..48
Obrázek 16- Editace atributů a operací třídy 50
Obrázek 17- Editace atributů výčtu..51
Obrázek 18- Nastavení parametrů vazby typu Agregace 53
Obrázek 19- Dialog pro generování kódu z diagramu tříd 54
Obrázek 20- Kontextové menu na elementu třída...57
Obrázek 21- Syntaktický strom pro Document.cs 63
Obrázek 22- Zástupci dynamicky získaných typů a instancí tříd z diagramu UML................69
Obrázek 23- Inspektor instance třídy Osoba...70
Obrázek 24- Výsledek volání metody VratVek ve Virtuální konzoli 73
Obrázek 25- Struktura řešení s užitím třídy MSBuildWorkspace 77
Obrázek 26- Výběr konstruktoru a vyplnění jeho parametrů 78
Obrázek 27- Emitované sestavení spuštěné jako konzolová aplikace 85
Obrázek 28- Scripting API – kolekce...87
Obrázek 29- Globální nastavení parametrů dotCORN..89
Obrázek 30- Čtení ze souboru pomocí Scripting API 89

Seznam tabulek

Tabulka 1- Tabulka lexémů po zpracování zdrojového kódu lexikálním analyzátorem 23
Tabulka 2- Základní elementy v diagramu tříd implementovaných v nástroji dotCORN 33
Tabulka 3- Obecné vztahy v UML..36
Tabulka 4- Jmenné prostory použité v praktické části práce..60
Tabulka 5- Tabulka parametrů metody InvokeMember...72
Tabulka 6- Parametry metody Create třídy CSharpCompilation..83
Tabulka 7- Parametry metody Emit třídy CSharpCompilation 84
Tabulka 8- Změny provedené v komponentě NClass 93

1 Úvod

Zhruba od konce 90. let minulého století se může široká laická, ale především odborná veřejnost setkávat se zajímavým trendem ve sféře tvorby a poskytování aplikací, nástrojů a služeb pro různé operační systémy. Přelomovým milníkem v této oblasti je nesporně vstup společnosti Microsoft do světa otevřeného software s technickou dostupností kódu a licencí. Na open source však není vždy bezpodmínečně nutné nahlížet jako na programy s otevřeným zdrojovým kódem, ale spíše je chápán jako software s bezplatnou dostupností, podporou zajišťovanou částečně nebo zcela vývojářskou komunitou, nekomerčností atd. Microsoft, který je dlouhodobě vnímán jako globální technologický lídr snad ve všech směrech programového vybavení osobních počítačů, serverů, notebooků, herních konzolí, mobilních telefonů či vestavěných jednoúčelových systémů zcela zásadním způsobem mění svou historickou filozofii. Jeho přijetí smyslu a principů open source software se projevuje ve dvou základních důležitých aspektech. Tím prvním je fakt, že Microsoft nyní vytváří kromě proprietárních programových balíků také služby a aplikace pro otevřené systémy. Druhá vývojová větev se ubírá směrem uvolňování řady aplikací, nástrojů a služeb jako otevřený software, což je vnímáno u tohoto softwarového giganta jako naprosto revoluční přístup. Pro příklad není třeba chodit daleko. Celá řada stavebních bloků v ekosystému .NET Framework má nyní charakter open source řešení. V kontextu změn v dalším vývoji některých produktů došlo např. ke kompletnímu přepsání překladačů Visual Basic .NET a C# do řízeného kódu a jejich následnému vydání se statusem open source projektu nazvanému .NET Compiler Platform, který je však spíše znám pod kódovým označením Roslyn.

Tradičně jsou překladače vnímány jako černé skříňky, do nichž na jedné straně vstupuje zdrojový kód, uvnitř se odehraje cosi magického a na druhém konci pak vystupují objektové soubory a programové jednotky k nasazení a opakovanému použití, řízení verzí a bezpečnosti, tzv. assembly. V průběhu své činnosti budují překladače poměrně hlubokou znalostní bázi zpracovávaného kódu, nicméně tyto nashromážděné poznatky nejsou v danou chvíli dostupné nikomu jinému než specialistům, kteří daný překladač implementovali. Navíc jsou takové informace ireverzibilně zrušeny ihned poté, co překladač dokončí svou činnost a vytvoří výstupní soubory.

Po několik dekád vývojářům takový, poněkud mlhavý vhled, do světa překladu zdrojových kódů vcelku vyhovoval, avšak v současné době již není tento stav nadále udržitelný. Stále častěji se totiž obrací na některé užitečné vlastnosti integrovaných vývojových prostředí (IDE),

jakými jsou např. IntelliSense, refaktoring, inteligentní přejmenování proměnných a konstant, vyhledání referencí, rychlý přechod k definici metody atd., které významně zvyšují produktivitu práce. Programátoři se opírají jednak o nástroje pro analýzu kódu, aby zvýšili jeho kvalitu a využívají generátory kódu pro usnadnění prototypování a konstrukci aplikací. Jelikož se tyto nástroje stále zdokonalují, vyžadují přístup ke stále většímu a většímu objemu detailních znalostí získaných z kódu, které doposud vytvářely a ovládaly pouze kompilátory. Klíčovým posláním projektu .NET Compiler Platform (Roslyn) je otevření výše zmiňované příslovečné černé skříňky monolitických překladačů a zpřístupnění jejich bohatých, v kontextu jejich činnosti však temporárních, znalostí analytickým nástrojům a koncovým uživatelům. Namísto klasických netransparentních překladačů s velmi jednoduchou filozofií: zdrojový kód na vstupu, objektový kód na výstupu přichází Roslyn. Kompilátory se stávají platformou, tj. sadou aplikačních programových rozhraní, které je možné využít v nástrojích a aplikacích pro řešení úloh spojených s kódem.

Posun k překladačům jako platformě (Compiler as a Platform) dramaticky snížil bariéry pro vstup do oblasti tvorby nástrojů a aplikací zaměřených na kód. Objevuje se mnoho nových příležitostí pro inovace v takových oblastech jako je meta-programování, generování a transformace kódu, interaktivní používání jazyků C# a Visual Basic, vkládání C# a VB do doménově specifických jazyků atd.

2 Cíl práce a metodika

2.1 Cíl práce

Tato diplomová práce se zaměřuje na formální popis a následné praktické využití sady open source překladačů a aplikačního programového rozhraní pro analýzu kódu určeného pro jazyky C# a Visual Basic .NET firmy Microsoft, které je soustředěno a implementováno v knihovnách .NET Compiler Platform.

Hlavním cílem práce je vytvoření aplikace využitelné pro podporu výuky objektově orientovaného programování v jazyce C#. V rámci implementace aplikace se budou dále demonstrovat vybrané metody lexikální, syntaktické a sémantické analýzy, včetně emitování dynamických sestavení souvisejících s .NET Compiler Platform.

2.2 Metodika

V teoretické části práce jsou popsána teoretická východiska z oblasti softwarového inženýrství s primárním zaměřením na oblast objektového návrhu software. Na základě těchto východisek je v praktické části práce implementována jednoduchá aplikace pro grafický návrh modelu tříd, jeho převod do zdrojového kódu jazyka C# a následnou analýzu toho kódu s využitím knihoven .NET Compiler Platform. V závěru bude zhodnocen přínos open source prostředků pro analýzu kódu a případné využití demonstrační aplikace jako didaktické pomůcky pro výuku objektově-orientovaných modelovacích technik.

3 Teoretická východiska

3.1 Principy a techniky kompilátorů

Ačkoli jsou principy a techniky překladačů intenzivně vyvíjeny a studovány počítačovými experty, matematiky a inženýry již od 60. let minulého století, jsou tato témata stále aktuální a prostupují svět moderní počítačové vědy. Jejich interdisciplinární charakter zasahuje do oblasti vyšších programovacích jazyků, strojových architektur, teorie konečných automatů, formálních jazyků a gramatik, algoritmů a softwarového inženýrství. Od doby, kdy Grace Hopperová[1] vytvořila v roce 1952 první překladač A-0 pro systém UNIVAC však došlo na tomto poli k podstatnému posunu, který byl indukován zejména vývojem stále pokročilejšího hardware a současně s tím i s rostoucí potřebou rychlého, efektivního a bezchybného zpracování dat. Překladače odstartovaly novou epochu vývoje nejen systémového, ale především uživatelského aplikačního software, čímž de facto redefinovaly stávající výpočetní model. Přes poměrně velkou škálu překladačů různé konstrukce, vnitřního uspořádání, složitosti a určení[2] jsou dnes dostupné a dobře zdokumentované i nástroje, s jejichž pomocí lze vytvořit i překladač pro vlastní jazyk s vlastní formální gramatikou a specifickým zaměřením[3]. (Aho, et al., 1986)

3.1.1 Definice překladače

Zjednodušeně řečeno, kompilátor je program, který dokáže číst vstupní kód v jednom jazyce (zdrojovém jazyce) a přeložit ho na ekvivalentní kód v jiném jazyce (cílovém jazyce). Důležitou vlastností kompilátoru je schopnost reportovat jakékoli chyby ve zdrojovém kódu, které detekuje v průběhu překladu. Pokud je cílový program ve formě spustitelného strojového kódu, může být uživatelem vyvolán, obvykle za účelem zpracování vstupních dat a vytvoření požadovaného výstupu. (Harrison, 2017)

Z matematického hlediska je překlad surjektivním zobrazením, tj. zobrazením **na** množinu. Je to takové zobrazení množiny X (v tomto případě množiny zdrojových jazyků) **na** množinu Y (množinu cílových jazyků, resp. cílových architektur), jestliže se na každý prvek z Y zobrazí alespoň jeden prvek množiny X. Formálně zapsáno:

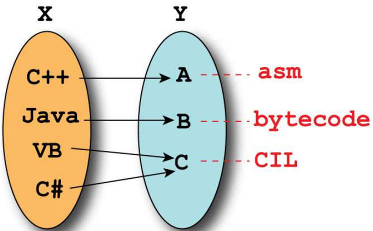
---

[1] Grace Murray Hopperová (1906 - 1992) byla americká matematička, počítačová vědkyně a důstojnice námořnictva Spojených států.

[2] Např. pro nejrozšířenější jazyky jako C/C++, Pascal, C#, Java atd.

[3] YACC, GNU bison pro generování syntaktického analyzátoru, Flex pro generování lexikálního analyzátoru, ANTLR – generátor překladače pro jazyky Java a C# využívající bezkontextovou gramatiku LL(k)

$\forall y \in Y\ \exists x \in X; y = f(x)$



Obrázek 1-Zobrazení z množiny zdrojových jazyků na množinu cílových jazyků
Zdroj: vlastní tvorba

Je to funkce, která mapuje jeden nebo více zdrojových kódů podle překladových parametrů na kód v cílovém jazyce. (Aho, et al., 1986)

3.1.2 Klasifikace překladačů

Přestože by bylo možno nalézt v literatuře celou řadu ucelených a podrobných klasifikačních struktur a taxonomií překladačů, budou zde vymezeny jen základní skupiny, s jejichž zástupci přijde do kontaktu nejen profesionální vývojář, ale i poučený uživatel. Příkladem běžného užití jazyků a jazykových procesorů může být skriptování v kancelářských aplikacích Microsoft Office, používání značkovacích jazyků (HTML, XML, CSS) v internetových prohlížečích, skript v nástrojích pro tvorbu vektorové grafiky (Adobe Illustrator®, AutoCAD®), SQL dotazy nad relační databází či práce v shellu operačního systému. Na tomto místě však není smyslem poskytnout vyčerpávající přehled všech typologií překladačů od doby jejich vzniku a praktického využívání, ale pouze představit základní kategorie současných kompilátorů včetně jejich nejběžnějších představitelů.
Kategorizovat lze dle mnoha aspektů, avšak primárně se kompilátory rozdělují podle způsobu zpracování zdrojového programu a schopnosti vytvářet cílový program na

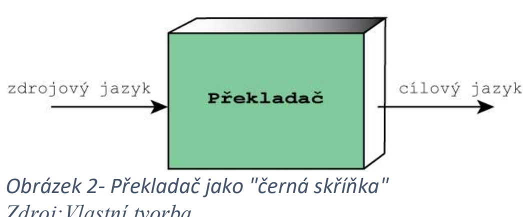
- generační překladače
- interprety
- hybridní překladače

Klasické překladače (generační)

Jsou jedním z nejrozšířenějších typů překladačů. Generační kompilátor překládá vstupní algoritmus zapsaný v nějakém, nikoli nutně vyšším, programovacím jazyce do cílového jazyka, čímž bývá obvykle spustitelný tvar, který může být reprezentován např. nativním EXE souborem, .NET assembly, přemístitelným objektovým souborem atd. To však není vždy podmínkou. Existují i tzv. konverzní překladače, které převádějí zdrojový jazyk do cílového

jazyka, který není strojovým kódem, tj. není v binární tvaru. Je to např. převod objektově orientovaného jazyka Eiffel na nízkoúrovňový jazyk C.

Samotný překlad funguje v principu jako roura[4]. Je rozdělen do několika na sebe navazujících fází, přičemž výstupy jejich činnosti nejsou většinou v době překladu uživateli dostupné. Nejprve probíhá série analytických procesů nad vstupním jazykem – lexikální, syntaktická a sémantická analýza. Následuje optimalizace a generování cílového programu. Překlad může probíhat v jednom nebo několika průchodech zdrojovým kódem. Podrobněji bude činnost generačního překladače, resp. jeho jednotlivých částí, popsána v kapitole 3.2.

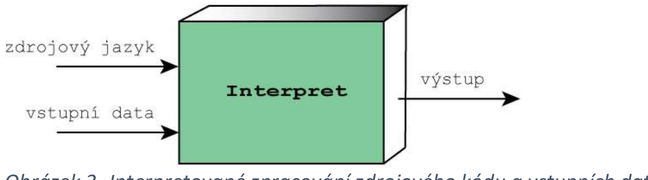


Obrázek 2- Překladač jako "černá skříňka"
Zdroj: Vlastní tvorba

Historicky nejznámějšími zástupci kompilovaných jazyků jsou *Fortran, Algol, C/C++, Pascal*.

Interprety

Ryzí interpret není překladačem v pravém slova smyslu, ale je označován spíše termínem jazykový procesor[5]. Je charakteristický tím, že nevytváří cílový program, ale přímo přistupuje k vykonávání jednotlivých operací specifikovaných ve zdrojovém programu, zpracovává vstupní data a produkuje požadovaný výstup. Ve srovnání s prováděním strojového kódu cílového programu vygenerovaného kompilátorem je činnost interpretu obvykle řádově pomalejší, nicméně interpretované zpracování zdrojového programu dokáže mnohdy poskytnout výrazně lepší diagnostiku chyb něž překladač, neboť vykonává zdrojový program příkaz po příkazu.



Obrázek 3- Interpretované zpracování zdrojového kódu a vstupních dat
Zdroj: Vlastní tvorba

Typickými představiteli interpretů a interpretovaných jazyků jsou *Perl*, *Python*, *Tcl*, *Smalltalk*, *Lisp*.

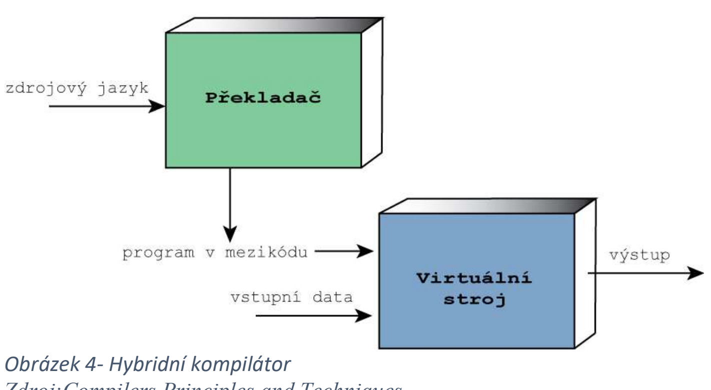
---

[4] V anglické literatuře se způsob zřetězení několika funkčních celků (modulů) v němž výstup jednoho je současně vstupem do dalšího označuje termínem „*Pipeline*“.

[5] angl. *language processor*

Hybridní kompilátory

Některé jazykové procesory, typicky pro jazyk Java nebo pro rodinu jazyků v prostředí .NET Frameworku, kombinují překlad a interpretaci. Zdrojový kód je nejprve přeložen do mezitvaru nazývaného bajtkód[6], resp. CIL[7], který je následně interpretován tzv. virtuálním strojem. Výhodou tohoto uspořádání je fakt, že bajtkód (CIL) vytvořený na jednom počítači může být interpretován na zcela jiném počítači, typicky v síťovém prostředí.



Obrázek 4- Hybridní kompilátor
Zdroj:Compilers Principles and Techniques

Překladače JIT (Just-in-Time)

Aby bylo dosaženo vyššího výkonu virtuálního stroje interpretujícího program v mezikódu, převádějí některé kompilátory za běhu tento mezikód (bajtkód nebo CIL) do nativního strojového kódu cílové platformy. Obvykle se tak děje bezprostředně před tím, než se začne určitá sekvence kódu vykonávat, resp. zpracovávat vstupní data. Takové metody překladu nesou v informatice označení JIT[8] (Just-In-Time, tj. „právě včas"). Nevýhodou tohoto přístupu je poměrně velká časová režie kompilace do strojového kódu, která je částečně řešena identifikací mnohonásobně odkazovaných částí kódu a uložením jejich překladu do cache paměti, odkud mohou být velmi efektivně vyvolány. Výhodou techniky JIT je naopak lepší optimalizace kódu pro konkrétní procesor s danou instrukční sadou. Většina JIT kompilátorů je schopna monitorovat činnost programu v aktuálním běhovém prostředí a shromažďovat

---

[6] angl. *bytecode*
[7] Původně MSIL – Microsoft Intermediate Language. CIL je procesorově a zároveň platformě nezávislý soubor instrukcí, které mohou být realizovány v jakémkoli prostředí podporujícím CLI (Common Language Infrastructure).
[8] JIT je také označení pro jednu z nejznámějších logistických technik vyvinutých v Japonsku firmou Toyota Motors Corp. a od 80. let minulého století masivně využívaných ve výrobě po celém světě. Princip řízení výrobního procesu je zde koncipován tak, že vše je podřízeno aktuálním potřebám (odtud označení „právě včas").

statistické informace, na jejichž základě pak dynamicky volí různé strategie relokace, přeskupení a rekompilace kódu pro optimální výkon.

Další typy překladačů

- Překladače jednoprůchodové, víceprůchodové[9]

  Klasifikace překladačů podle počtu průchodů má svůj původ v omezení hardwarových zdrojů počítačů. Proces překladu zahrnuje vykonání řady operací a původní počítače neměly dostatek paměti na to, aby v ní byl uložen jeden program, který vykonává všechny tyto činnosti. Z toho důvodu byly překladače rozděleny na menší programy, z nichž každý procházel zdrojový kód (nebo nějakou jeho reprezentaci) a prováděl specifické operace z požadovaných analýz a překladů.

  Jednoprůchodové překladače

  Schopnost kompilovat zdrojový kód v jednom průchodu byla klasicky vnímána jako benefit, jelikož zjednodušovala implementaci překladače a na rozdíl od víceprůchodových překladačů poskytovala rychlejší překlad. Částečně díky omezeným prostředkům původních systémů, řada prvotních jazyků byla specificky navržena tak, aby mohly být překládány jednoprůchodovými kompilátory (např. *Pascal*). Nevýhodou jednoprůchodového překladu je skutečnost, že není možné provádět řadu sofistikovaných optimalizací nutných pro vygenerování vysoce kvalitního cílového kódu. (Aho, et al., 1986)

  Víceprůchodové překladače

  V některých případech může design určitých vlastností jazyka vyžadovat provedení více než jednoho průchodu zdrojovým programem. V prvním průchodu se např. shromáždí informace o deklaracích objevujících se až po příkazech, kterých se týkají a teprve v dalším průchodu dojde k vlastnímu překladu. Je velmi obtížné předpovědět, kolik průchodů při optimalizaci překladač provede, než vygeneruje cílový program. V prvním průchodu se obvykle čte zdrojový kód, provádějí se základní úlohy jako je lexikální analýza, konstrukce abstraktního syntaktického stromu, sestavení tabulky symbolů, sémantická analýza atd. Poté se výsledky uloží do souboru nebo paměti a stanou se vstupními daty, které budou čteny při druhém průchodu. Každý následný průchod *N* bude číst výsledky předcházejícího průchodu *N-1* a tím bude měnit reprezentaci programu stále blíže ke strojovému kódu, přičemž výsledky průchodu *N* budou vstupem

[9] V anglické literatuře je jednoprůchodový překladač označován termínem one-pass, single-pass nebo narrow compiler. Víceprůchodový překladač je označován jako multi-pass nebo wide compiler.

pro průchod *N+1*. Tento proces se opakuje tak dlouho, dokud poslední průchod nevygeneruje finální tvar kódu. (Melichar, a další, 1999)

Víceprůchodové překladače jsou konstruovány např. pro jazyk *C/C++, Java* nebo pro jazyky z .NET Frameworku (*C#, Visual Basic*).

- Threaded-code

  Je technika využívaná k implementaci virtuálních interpretačních strojů. Kód, který je generován tímto typem překladačů je téměř výhradně tvořen voláním podprogramů. Threaded-code je typickým představitelem překladače pro jazyky FORTH, většinu implementací BASICu a některé verze COBOLu.

- Stage kompilátory
- Parallelizing compilers

  Představují celou skupinu překladačů (parallelizing/concurrentizing/vectorizing compilers), které převádějí běžný sekvenční program (např. v jazyce Fortran nebo C) do jeho paralelní formy, tj. takové podoby, která mu umožňuje efektivně pracovat v multiprocesorovém prostředí se sdílenou pamětí. Výsledný kód využívá tzv. master/slave paradigma. (Appel, a další, 2004)

- Cross compiler

  Křížový překladač je takový překladač, který je schopný vytvářet spustitelný kód pro jinou platformu, než je ta, na které je spouštěn. Příkladem může být překladač spuštěný na PC s operačním systémem Windows 10 a generující kód pro systém Android. Někdy není cílová platforma ani uzpůsobena tomu, aby na ní byl spouštěn jakýkoli překladač. To je případ mikrokontrolerů a vestavěných (embedded) systémů, která neobsahují žádný operační systém.

- Self-hosting compiler

  Nazývaný také bootstrapping, je implementace překladače v jazyce, který je určen ke kompilaci tímto překladačem. Počáteční verze jádra překladače je vytvořena v jiném jazyce (obvykle v assembleru) a každé další verze překladače jsou spuštěny s využitím minimálního jádra tohoto jazyka.

  BASIC, Algol, Haskell, Rust, Python, Scala, Eiffel

- Retargetable compiler

  Tento typ překladačů byl navržen tak, aby bylo možné ho velmi snadno modifikovat pro generování kódu pro různé cílové platformy a procesory různých instrukčních sad.

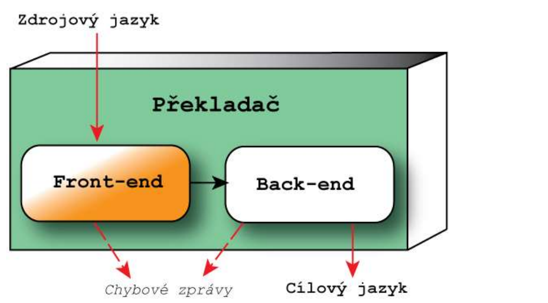
Někteří zástupci tohoto typu překladačů, např. GCC[10], se staly tak široce portovanými a vyvíjenými, že nyní zahrnují podporu pro mnoho optimalizačních a strojově specifických detailů a kvalita kódu, který generují často předčí běžné kompilátory. Hlavními představiteli jsou kromě již zmiňovaného *GCC* také ACK[11], *lcc*[12], *VBCC*[13], *Portable C Compiler, Small-C* (Appel, a další, 2004)

[10] Angl. GNU Compiler Collection
[11] Amsterdam Compiler Kit
[12] Local C Compiler nebo též Little C Compiler
[13] Volker Barthelmann C Compiler – ISO/ANSI C překladač

3.2 Struktura překladače

Na nejvyšší úrovni abstrakce lze na kompilátor nahlížet jako na jednoduchou černou skříňku, která mapuje zdrojový program na sémanticky ekvivalentní cílový program. Pokud se tato skříňka pootevře, je zřejmé, že je pro toto mapování vybavena dvěma částmi, a to částí analytickou a syntetickou.

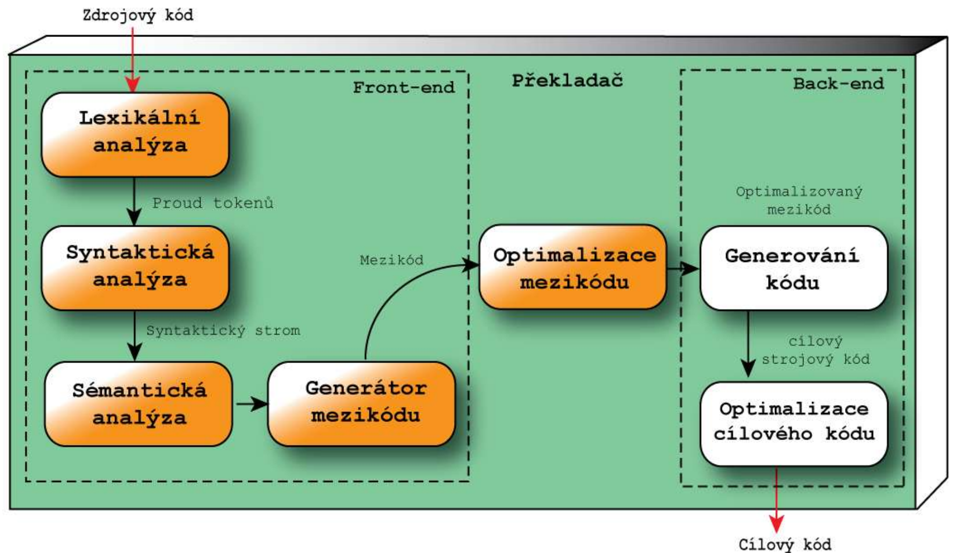
- **Analytická část** rozděluje zdrojový kód na základní složky a kontroluje, zdali vyhovují předepsané gramatické struktuře. Takto strukturovaný kód je pak využit pro vytvoření přechodové reprezentace zdrojového programu. Jestliže analytická část detekuje ve zdrojovém programu syntaktickou nebo sémantickou chybu, potom o této skutečnosti informuje uživatele, který provede korekce na příslušných místech zdrojového kódu. Analytická část také shromažďuje informace o zdrojovém programu a ukládá je do datové struktury nazývané tabulka symbolů, která je následně předána, společně s přechodovou reprezentací, do syntetické části kompilátoru.
- **Syntetická část** již konstruuje požadovaný cílový program z přechodové reprezentace a informací v tabulce symbolů. Analytická část je často označována jako front-end kompilátoru a syntetická část jako back-end. (Aho, et al., 1986)



Obrázek 5- Základní části překladače - Front-end a Back-end
Zdroj: (Hummel, a další, 2014) - upraveno autorem

Proces kompilace probíhá jako řetězec fází, z nichž každá transformuje jednu reprezentaci zdrojového programu na jinou. Některé fáze mohou být seskupeny a mezikód mezi seskupenými fázemi nemusí být konstruován explicitně. Tabulka symbolů, která uchovává informace o celém zdrojovém programu je využívána napříč všemi fázemi překladu.

Některé překladače mají mezi front-endem a back-endem strojově nezávislou optimalizační fázi. Ta slouží k provedení transformací na mezikódu tak, aby back-end mohl generovat kvalitnější cílový program, než by jinak generoval z neoptimalizované mezikódové reprezentace. (Melichar, a další, 1999)



Obrázek 6- Dekompozice překladače na jednotlivé fáze
Zdroj: (Hummel, a další, 2014) - *upraveno autorem*

3.2.1 Lexikální analyzátor

První fáze kompilace se označuje jako lexikální analýza, resp. skenování. Lexikální analyzátor čte znakový proud ze zdrojového programu a seskupuje je do smysluplných sekvencí nazývaných lexémy. Pro každý lexém je vytvářen tzv. token[14], obvykle uspořádaná dvojice ve tvaru <*název-tokenu, hodnota-atributu*>, který pak vstupuje do další fáze překladu, kterou je syntaktická analýza. První složka tokenu, tedy *název-tokenu*, je abstraktní symbol, který je využíván během syntaktické analýzy. Druhá složka, *hodnota-atributu*, odkazuje na záznamu v tabulce symbolů, jenž je jednoznačně určený touto hodnotou pro daný token. Informace z tabulky symbolů jsou důležité pro sémantickou analýzu a generování kódu. Lexémy jsou popsány pomocí regulárních výrazů a pro jejich rozpoznávání se používá konečný automat. (Hummel, a další, 2014)

Jako příklad může posloužit triviální příkaz pro větvení programu **if**.

Př.1 Zpracování příkazu **if** lexikálním analyzátorem.

```
// Comment
if (score > 100)
  grade = "A++";
```

Výstup lexikálního analyzátoru pak bude vypadat následujícím způsobem:

[14] V angl. doslova známka, žeton

Název tokenu	Span	Hodnota atributu
IfKeyword	@ Span=[12..14)	
OpenParenToken	@ Span=[15..16)	
IdentifierToken	@ Span=[16..21)	score
GreaterThanToken	@ Span=[21..22)	
NumericLiteralToken	@ Span=[22..25)	100
CloseParenToken	@ Span=[25..26)	
IdentifierToken	@ Span=[30..35)	grade
EqualsToken	@ Span=[36..37)	
StringLiteralToken	@ Span=[38..43)	A++
SemicolonToken	@ Span=[43..44)	

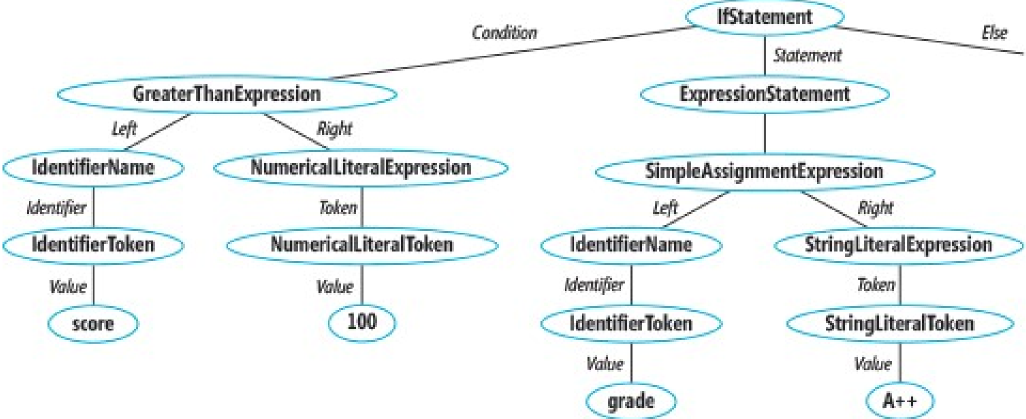
Tabulka 1- Tabulka lexémů po zpracování zdrojového kódu lexikálním analyzátorem
Zdroj: (Hummel, a další, 2014) – upraveno autorem

Prostřední sloupec ve výše uvedené tabulce obsahuje informaci označenou jako **Span**, což je absolutní počáteční a koncová pozice tokenu ve zdrojovém textu počítaná ve znacích od prvního znaku v řetězci (souboru či obecně libovolného streamu). Je zřejmé, že rozdíl těchto dvou hodnot pak bude představovat délku tokenu ve znacích. Při stanovení pozice se zohledňují i tzv. *trivia*, tj. znaky jako mezera, tabulátor, konec řádku a pochopitelně celá řada znaků reprezentovaných např. escape sekvencemi atd. **Span** a **FullSpan** jsou jedny z četné sady informací dostupných v SyntaxTree API platformy Roslyn při zkonstruování syntaktického stromu a jejich získání a využití je předvedeno na praktickém příkladu v kapitole 4.3.1 praktické části této práce. (Vasani, 2017)

3.2.2 Syntaktický analyzátor

Druhou fází překladu je syntaktická analýza[15]. Syntaktický analyzátor využívá tokeny, které byly vytvořeny lexikálním analyzátorem, aby sestavil stromovou reprezentaci popisující jejich gramatickou strukturu. Typickou reprezentací je syntaktický strom, ve kterém každý vnitřní uzel představuje určitou operaci a potomci uzlu představují argumenty této operace.

[15] Hovorově označovaná jako parsování či parsing.



Obrázek 7- Syntaktický strom příkazu if z Př.1
Zdroj: (Hummel, a další, 2014) – *upraveno autorem*

3.2.3 Sémantická analýza

Sémantický analyzátor využívá syntaktického stromu a informací z tabulky symbolů, aby ověřil, zdali je zdrojový program po sémantické stránce konzistentní s definovaným formálním jazykem. Shromažďuje také informace o typech a ukládá je buď do syntaktického stromu nebo do tabulky symbolů pro následné použití během generování mezikódu.

Důležitou částí sémantické analýzy je typová kontrola, při které překladač kontroluje, má-li každý operátor odpovídající typ operandů. Specifikace jazyka však mnohdy připouští určité typové konverze. Příkladem budiž použití binárního operátoru sčítání. Ten může být použit jak pro součet celočíselných operandů, tak pro součet operandů v oboru reálných čísel (čísel v plovoucí řádové čárce). Je-li však operátor + použit pro součet celého a reálného čísla, může překladač vynutit konverzi celého čísla na číslo reálné.[16] (Hummel, a další, 2014)

3.2.4 Překlad a optimalizace kódu

V procesu překladu zdrojového programu na cílový kód může kompilátor vytvořit jednu nebo více přechodových reprezentací, které mohou mít mnoho různých podob. Syntaktické stromy jsou jistou formou této přechodové reprezentace. Obvykle jsou využívány během syntaktické a sémantické analýzy.

Po průchodu těmito dvěma fázemi generuje řada překladačů nízkoúrovňový strojový mezikód, o kterém lze hovořit jako o programu pro abstraktní stroj. Tento mezikód by měl mít dvě důležité vlastnosti – měl by jít snadno vygenerovat, a především snadno přeložit do cílového tvaru. Obvykle je uvažována forma tzv. **tříadresného kódu**. Tento kód je složen ze sekvence

[16] Tento typ konverze je v zahraniční literatuře označován jako *coercions* – donucení, vynucení.

instrukcí podobných asembleru, obvykle se dvěma nebo třemi operandy pro každou instrukci. Každý operand může vystupovat jako registr. (Aho, et al., 1986)

- **Optimalizace kódu**

Fáze strojově nezávislé optimalizace kódu se pokouší zlepšit mezikód tak, aby bylo dosaženo vyšší kvality cílového kódu, což obvykle znamená jeho zkrácení, zvýšení rychlosti a efektivity, snížení časových a prostorových nároků atd. Techniky optimalizace jsou obecně netriviální a některé překladače věnují této fázi významnou část svého kompilačního času. Existují však optimalizační algoritmy, které dokáží podstatně zrychlit běh cílového programu, aniž by při tom zpomalovaly samotnou kompilaci. (Aho, et al., 1986)

- **Generování kódu**

Generátory kódu zpracovávají jako svůj vstup mezikód vytvořený ze zdrojového programu a převádějí ho na cílový jazyk. Pokud je cílový jazyk strojovým kódem, pak jsou pro každou proměnnou využívanou programem vybrány registry nebo segmenty paměti. Následně je mezikód přeložen na ekvivalentní sekvenci strojových instrukcí vykonávajících tutéž úlohu. Zcela zásadním aspektem při generování kódu je správné přidělení registrů pro uchování proměnných. (Aho, et al., 1986)

- **Správa tabulky symbolů**

Podstatnou funkcí kompilátoru je zaznamenávání jmen proměnných použitých ve zdrojovém programu a shromažďování informací o různých atributech každé proměnné. Tyto atributy mohou představovat velikost alokované paměti, typ proměnné, rozsah proměnné (tj. na kterých místech programu je její hodnota použita). V případě jmen procedur nebo funkcí jsou to informace o počtu a typu parametrů, způsobu jejich předávání (hodnotou či odkazem), typu návratové hodnoty apod.

K tomuto účelu slouží datová struktura označovaná jako tabulka symbolů[17]. Je navržená tak, aby umožňovala kompilátoru velmi rychle vyhledat požadovaný záznam a číst, případně modifikovat data v něm uložená. (Aho, et al., 1986)

[17] Angl. Symbol-Table

3.3 .NET Compiler Platform (Roslyn)

Ze zkušeností získaných přípravou a realizací rešeršní (a samozřejmě i praktické) části této práce snad nelze tuto kapitolu uvést jinak než prostým konstatováním, že odborných publikací o Roslynu příliš mnoho není, o to intenzivněji se však s jeho využitím programuje. Prostřednictvím celé řady aplikačních rozhraní a v nich soustředěných funkcí, datových struktur a implementací objektových modelů lze vyvíjet sofistikované nástroje či rozšíření vývojového prostředí Microsoft Visual Studio, o kterých dříve nebylo možno z pohledu aplikačního programátora ani uvažovat. Otevření zdrojových kódů a uvolnění licence nebyl ze strany společnosti Microsoft počin náhodný ani neměl altruistický podtext. Jejich dosavadní překladače, byť vysoce ceněné pro svou výkonnost, efektivitu, rychlost a kvalitu generovaného kódu, napsané v drtivé většině případů v jazyce C, trpěly značnou erozí kódu a implementace nových, moderních prvků jazyka se stávala čím dál tím více obtížnější.
Roslyn je tedy otevřenou platformou (zdrojové kódy jsou veřejně dostupné na GitHubu[18]) určenou především pro analýzu kódu (lexikální, syntaktickou a sémantickou) a jeho kompilaci do sestavení (assembly) odpovídající formátu PE souboru ve Frameworku .NET, přičemž kompilátory pro konkrétní jazyk (v tomto případě pro jazyky C# a Visual Basic .NET) jsou napsány v daném jazyce (viz. kapitola 3.1.2, odstavec self-hosting compiler). V neposlední řadě je třeba zmínit zjevný fakt, že Roslyn je platformou značně rozsáhlou. I rámcový přehled se stručným popisem funkcí by vydal na samostatnou publikaci, což není smyslem této práce. V dalším textu bude tedy popis omezen pouze na oblasti, které jsou relevantní pro praktickou část. (Sole, 2016)

3.3.1 Architektura .NET Compiler Platform

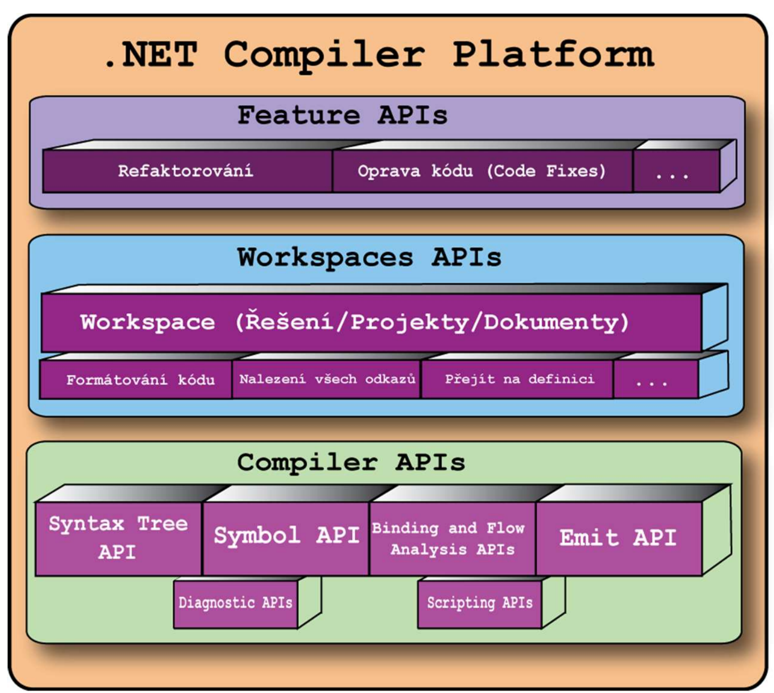
Jak je patrné z následujícího obrázku, Roslyn je vybudován ze tří základních vrstev, které představují komplexní množiny API vztahující se k dané problémové doméně. Jsou to následující skupiny API:

- Compiler API
- Workspaces API
- Features API

S odkazem na stanovené cíle a z nich vyplývající praktickou část této práce je nutné podotknout, že prvé dvě uvedené množiny API lze s úspěchem využívat mimo vývojové prostředí Microsoft Visual Studio, neboť jsou na jeho komponentách zcela nezávislé. Naproti

[18] GitHub je webová služba podporující vývoj software za pomoci verzovacího nástroje Git.

tomu Features API je s Visual Studiem velmi úzce svázáno a až na velmi ojedinělé případy je jeho použití v samostatně stojících aplikacích zcela nemožné nebo značně omezené a obtížně implementovatelné. Features API totiž disponuje paletou funkcí pro refaktorování, styl kódu, opravy kódu apod. Typickým příkladem může být grafické označení syntakticky nesprávného či neúplného kódu v editoru Visual Studio.[19]



Obrázek 8- Architektura .NET Compiler Platform
Zdroj: (Sole, 2016) – *upraveno autorem*

3.3.2 Compiler API

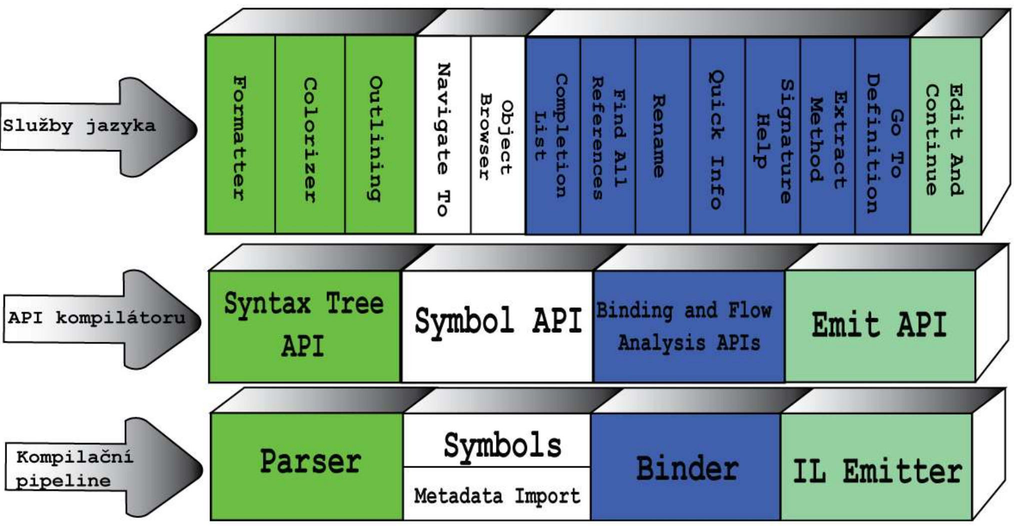
Compiler API lze definovat jako soubor aplikačních rozhraní implementujících objektový model, který koresponduje s informacemi vznikajícími v různých fázích procesu překladu, a to jak syntaktickými, tak sémantickými. Tyto informace jsou shromažďovány a sdíleny prostřednictvím vhodných .NET objektů. Compiler API také zahrnuje jakýsi statický snímek volání kompilátoru, který je složen ze souborů zdrojového kódu, nastavení předvoleb kompilace a odkazů na jiná sestavení. Compiler API je vrstva nezávislá na komponentách Visual Studia. (Uhlenhuth, 2016)

[19] Červená, popř. modrá vlnovka pod problematickou částí kódu. V angličtině se označuje termínem „squiggle“, doslova klikyhák.

Funkční oblasti překladače

Každá fáze kompilačního řetěze je, na rozdíl od klasického překladače, nyní realizována samostatnou komponentou. V první fázi dochází k syntaktickému rozboru. Zdrojový kód je rozložen na tokeny (viz.3.2.1) a následně na syntaktické prvky dle gramatiky daného jazyka (C# nebo Visual Basic .NET). Druhá fáze je deklarační. V ní jsou deklarace ze zdrojových a importovaných metadat rozloženy do tvaru pojmenovaných symbolů a následně jsou identifikátory v kódu navázány na odpovídající symboly. Ve finální fázi jsou veškeré informace sestavené kompilátorem emitované ve formě assembly.

Jednotlivé etapy překladu využívají odpovídající množinu API s objektovým modelem, který umožňuje přístup k informacím, které v tomto stádiu překladu vznikly. Fáze parsování je zpřístupněna jako syntaktický strom, fáze deklarace jako tabulka hierarchických symbolů, vazebná fáze jako model, který vystavuje výsledky sémantické analýzy překladače a fázi emitování jako API, které generuje IL bajtkód. (Uhlenhuth, 2016)



Obrázek 9- Tradiční kompilační pipeline a API překladače Roslyn
Zdroj: (Uhlenhuth, 2016) – upraveno autorem

SyntaxTree API

Ačkoli Roslyn obsahuje stovky metod a prostředků, jichž lze použít na tisíce různých způsobů, existuje struktura, která zaujímá zcela výjimečné postavení a tou je syntaktický strom. Ten lze definovat jako stromovou datovou strukturu, jejíž neterminální strukturální elementy jsou současně rodiči dalších strukturálních elementů. Každý syntaktický strom je tvořen z uzlů, tokenů a dalších doplňujících jazykových elementů označovaných jako *trivia*. Syntaktické stromy slouží ke dvěma základním účelům: (Harrison, 2017)

1. Poskytují nástroje pro zpracování syntaktické struktury zdrojového kódu
2. Editaci zdrojového kódu bez použití přímých textových úprav

Syntaktické stromy mají tři klíčové vlastnosti:

1. Plně zachovávají veškeré zdrojové informace (každý literál uvedený ve zdrojovém kódu je reprezentován přesně tak, jak byl nadefinován). Chybějící token v syntaktickém stromu pak indikuje syntaktickou chybu ve zdrojovém textu.
2. Syntaktický strom sestavený parserem je zcela zpětně převeditelný do původního textu, ze kterého byl získán. (pozn. autora: zcela snadno – voláním metody *ToFullString* na kořenovém uzlu syntaktického stromu)
3. Syntaktický strom po svém vytvoření zachycuje aktuální stav zdrojového kódu a je neměnný. Je také threadově zabezpečený, což umožňuje konkurenční přístup uživatelů z různých vláken, aniž by došlo k jeho uzamčení či vzniku duplicit. Syntaktický strom není možné přímo modifikovat. Pro tyto účely však existují speciální metody (tzv. factory methods). (Vasani, 2017)

Jakýsi most mezi SyntaxTree API a Symbol API představuje sémantický model, který je dostupný přes kešovaný objekt *CSharpCompilation* nebo prostřednictvím tzv. *AdhocWorkspace*. Význam a použití obou přístupů bude demonstrováno v praktické části v kapitolách 4.3.3, resp. 4.3.2.

Emit API

Na první pohled stojí Emit API na konci řetězce operací vedoucích od zdrojového kódu ke spustitelnému sestavení (či dynamicky linkované knihovně), nicméně nemusí tomu tak být vždy. Nejfrekventovanější jsou zřejmě třídy *CSharpCompilation* ze jmenného prostoru `Microsoft.CodeAnalysis` a *Compilation* z `Microsoft.CodeAnalysis.Emit`. Obě totiž mimo jiné implementují metody, které vrací referenci na sémantický model (*GetSemanticModel* resp. *GetSemanticModelAsync*). To může být signifikantní v různých fázích vývoje, kdy není přímo vyžadována fyzická reprezentace sestavení v podobě souboru, ale pouze existence objektu kompilace jako poskytovatele prostředků pro realizaci jiné očekávané funkcionality. Instance obou uvedených tříd vyjadřují neměnnou reprezentaci jednoduchého volání kompilátoru. Je zde opět zachován princip neměnnosti, podobně jako je tomu u syntaktického stromu, řešení, projektů a mnoha dalších stavebních kamenů Roslynu. Jakmile je objekt kompilace vytvořen, nelze jej změnit. Je však možné vytvářet nové kompilace z kompilací již existujících, což může

být v jistých případech velmi efektivní s přihlédnutím na paměťovou náročnost procesu překladu. (Varty, 2014)

Scripting API

Scripting API je sada prostředků, které poskytují tytéž funkce, jaké jsou implementované v interaktivním okně Visual Studia 2015 (a vyšší), popř. z příkazové řádky systému (Developer Command Prompt VS2015). Microsoft označuje tento nástroj termínem C# REPL[20] a obdobné řešení je též implementováno v praktické části a popsáno v 4.3.4. Scripting API bylo součástí Roslynu již v rané verzi 1.0. Po řadě funkčních zdokonalení, zejména v oblasti asynchronního vyhodnocování výrazů (*EvaluateAsync*) a asynchronního spouštění skriptů (*RunAsync*) bylo API přesunuto pod jmenné prostory `Microsoft.CodeAnalysis.Scripting` (obecné nastavení atributů *ScriptOptions* pro tzv. skriptovací továrny) a `Microsoft.CodeAnalysis.CSharp.Scripting`. V říjnu 2015 bylo API publikováno v pre-release formátu. Je závislé na instalování .NET Frameworku verze 4.6. (Varty, 2014)

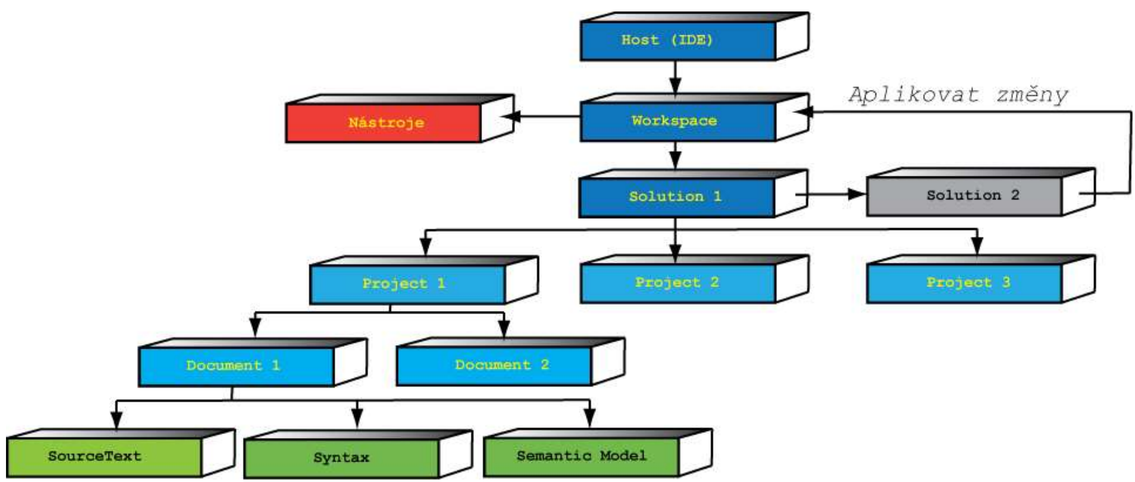
3.3.3 Workspaces API

Tato vrstva představuje bránu k analýze kódu napříč všemi řešeními, projekty a dokumenty. Prostřednictvím této kolekce API jsou veškeré relevantní informace uspořádány do jednotného objektového modelu, který k nim poskytuje přímý přístup. Ve verzi .NET Compiler Platform 2.6.0 je však použití celé řady metod z této vrstvy silně vázáno na přítomnost sestavení z rodiny MSBuild (`Microsoft.Build`, `Microsoft.Build.Engine` a `Microsoft.Build.Framwork`). Fundamentální třídou, která pak umožní vstup do světa řešení (*Solution*), projektů (*Project*) a dokumentů (*Document*) je *MSBuildWorkspace*. Ukázka využití tohoto API je pak součástí praktické části této práce.

Význam pracovních prostor (Workspaces) a uspořádání řešení, projektů a dokumentů v C# je zachycen na následujícím obrázku. Kořenovým uzlem označeným jako „Host“ je obvykle integrované vývojové prostředí Visual Studio, ale není to nutnou podmínkou. Může se jednat o libovolnou aplikaci, která je schopná otevřít Solution a jistým, byť omezeným způsobem, s ním manipulovat. Host na základě interakcí s uživatelem, jiným systémem či aplikací předává informace do vrstvy pracovního prostředí, která dle povahy události, kterou vyvolal Host

[20] REPL – Read-Eval-Print-Loop označován jako „language shell“ je forma interaktivního programování, kdy interpret čte (Read) z konzole výrazy či jiné programové bloky. Ty následně vyhodnocuje (Eval) a výsledky posílá na standardní výstup (Print). Tento proces se může libovolně dlouho opakovat (Loop).

zachází s řešeními, projekty, dokumenty i zdrojovými texty, kterými je dokument tvořen. Workspaces API zveřejňuje objektový model, který reprezentuje řešení jako statický snímek, tj. neměnnou strukturu. Přes objektové modely projektů lze přistupovat k vlastnostem projektu, jeho referencím apod. Objektové modely dokumentů pak poskytují přístup ke zdrojovému kódu. (Bock, 2016)



Obrázek 10- Hierarchické uspořádání řešení, projektů a dokumentů
Zdroj: Vlastní tvorba

3.4 Diagram tříd v jazyce UML

S ohledem na skutečnost, že jedním z pilířů praktické části této diplomové práce je jednoduchý nástroj pro grafický[21] návrh diagramu tříd v jazyce UML[22], poskytne tato kapitola stručný, nicméně nezbytný, úvod do této problematiky, a to především s těsnou vazbou na elementy implementované právě v praktické části (viz 4.2.1).

Specifikace UML obsahuje 4 základní části, které popisují:

- infrastrukturu (základní elementy a vztahy)
- superstrukturu (diagramy a jejich význam)
- výměnný formát (jak zápis v UML exportovat a importovat)
- definici jazyka OCL[23] (jazyk pocházející z metodiky Syntropy, ve kterém lze formálně přesně zapisovat integritní omezení modelů)

Jazyk UML je definován pomocí modelu v UML – metamodel UML je zapsán pomocí diagramů tříd UML opatřených popisem sémantiky a doplněn formálním vyjádřením sémantiky v OCL. (Benešovský, a další, 2002)

Diagram tříd v UML je jedním ze šesti základních diagramů popisujících strukturu systému. Poskytuje statický pohled na popisovaný systém a je vynikajícím nástrojem pro pochopení modelované reality prostřednictvím vizualizace entit a vztahů mezi nimi. V současné době je prakticky nepostradatelný při návrhu a dokumentaci systémů, neboť poskytuje zásadní vhled do jeho struktury a mnohdy lze na jeho základě vyvozovat klíčové předpoklady o složitosti implementace a s ní související efektivitu programátorské práce (a z ní plynoucí nákladovou složku vývoje).

3.4.1 Elementy v diagramu tříd

Každý diagram je sestaven z jistých elementů a vztahů mezi nimi. Obecně se v UML připouští, aby v každém modelu byl použít libovolný element. Avšak ne všechny kombinace jsou smysluplné, vždy určitá kombinace elementů a vztahů představuje superstrukturu diagramu jistého typu. (Arlow, a další, 2007)

V následující tabulce jsou uvedeny čtyři základní prvky, které jsou dostupné v designeru diagramu *dotCORN*. V této fázi dokumentace bude spíše akcentována úroveň využívání UML jako vyjadřovacího prostředku-programovacího jazyka, ze kterého se následně generuje kód

[21] Existuje možnost vytváření diagramu tříd i jinými způsoby, např. z příkazové řádky v nástroji PlantUML.

[22] Unified Modeling Language - „Standard konsorcia OMG (Object Management Group) pro záznam, vizualizaci a dokumentaci artefaktů systému s převážně softwarovou charakteristikou“.

[23] Object Constraint Language

do cílového programovacího jazyka. S přihlédnutím k jednomu z faktorů motivace vzniku této práce, který je uveden i jako jeden ze stanovených cílů, tj. vytvoření aplikace využitelné také jako didaktické pomůcky pro výuku objektově-modelovacích technik, bylo vyhodnoceno použití následujících elementů (ale též i vazeb mezi nimi) jako plně dostačujícího pro modelování běžných situací, které mohou být předmětem semestrálních či jiných projektů.

Prvek	UML Notace	Popis
Třída (Class)	Class1 Public Class	Třída objektů, které nesou kvalitativně stejné vlastnosti.
Rozhraní (Interface)	Interface1 Public Interface	Je speciálním stereotypem[24] třídy, který představuje kolekci atributů a signatur operací, které definují množinu chování směrem vně systému.
Výčet (Enumeration)	Enum1 Public Enum	Prvek umožňující vymezit škálu dostupných hodnot pro určitý atribut (resp. příznak - Tag)
Poznámka (Note)	Dvojklikem upravíte komentář	Obecně použitelný prvek podávající doplňující informace k libovolnému elementu v diagramu.

Tabulka 2- Základní elementy v diagramu tříd implementovaných v nástroji dotCORN
Zdroj: vlastní tvorba

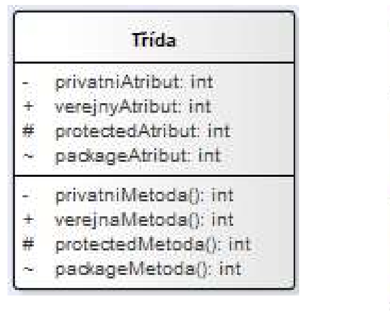
Diagram tříd, který bude v následujících kapitolách zmiňován není reprezentací doménového modelu, ale je skutečným diagramem implementace, tj. je platformě závislým (jazyk C# v prostředí .NET) a atributy stejně jako parametry a návratové hodnoty operací budou typovány v souladu s datovými typy definovanými v cílové platformě.

- **Element Třída (Class)**

Definic pojmu třída je přesně tolik, kolik je autorů publikací o objektově orientovaném přístupu při návrhu systémů. V širším slova smyslu ji lze chápat jako šablonu (předpis) pro vytváření objektů[25] se stejnou datovou strukturou a stejným chováním (instance třídy). Objekty stejné třídy mají stejně definované atributy a metody. (Kanisová, a další, 2007)

[24] Stereotyp v UML zastupuje určitou variantu v daném modelu existujícího prvku, který má sice stejnou podobu (atributy a relace), ale používá se s jiným záměrem.
[25] Objekt je pojem, abstrakce nebo věc s dobře definovanými hranicemi a významem.

Notace tříd – viditelnost symbolem a ikonou



Obrázek 11- Třída s UML notací (vlevo) a třída z aplikace dotCORN (vpravo)
Zdroj: vlastní tvorba

Atributy tříd

Atributy definují statickou strukturu objektové třídy. Atribut je nositelem informací o objektu a je definován svým jménem, formátem (typem) a viditelností. Název atributu jednoznačně pojmenovává danou vlastnost objektu.

Notace pro definici je atributů:

viditelnost jméno [násobnost] : typ = počáteční hodnota { omezení, vlastnost }

Operace tříd

Nedílnou součástí struktury objektu je jeho chování, které je definováno operacemi. Ty jsou určeny svým názvem, seznamem parametrů a návratovou hodnotou. Této charakteristice operace se říká **signatura**. Signatura operace v rámci objektové třídy musí být jednoznačná a unikátní. (Kanisová, a další, 2007)

Notace pro definici metod:

viditelnost jméno(argument1: typ, argument2: typ…): návratový typ { omezení, vlastnost }

Viditelnost je poměrně významnou charakteristikou atributu nebo operace v objektové třídě. UML rozlišuje několik základních typů viditelnosti:

Public – veřejný přístup. Kterýkoli element systému může k těmto atributům (operacím) přistupovat

Private – soukromý přístup. K atributům mají přístup pouze operace implementované v dané třídě

Protected – chráněný přístup. K atributům mají přístup pouze operace implementované v dané třídě a v jejích potomcích.

- **Element Rozhraní (Interface)**

Interface je třídou se stereotypem <<*interface*>>. S jistou nadsázkou lze interface vnímat jako „smlouvu", jejíž partner, tj. implementující třída se zavazuje této smlouvě dostát, a to tak, že bude implementovat všechny metody v rozhraní. Rozhraní specifikuje chování objektu. Z této skutečnosti plynou některé podstatné vlastnosti interface. Každý člen rozhraní (atribut či operace) má vždy přístupový modifikátor typu ***Public***. Metody v rozhraní nemají implementaci. Ta je ponechána na realizující třídě. (Benešovský, a další, 2002)

- **Element Výčet (Enumeration)**

Tento element se používá v případě, že v aplikační doméně existují atributy s neměnnou sadou možných hodnot. Tyto atributy se nazývají atributy výčtu a množiny definující jejich hodnoty jsou typem tohoto výčtu. Opět s odkazem na ukázkový diagram z praktické části (Příloha A) je pomocí výčtového typu např. možné specifikovat směr otáčení listů v knize (množina *Listovani* a její prvky *Vpřed, Vzad*). Argument *směr* v metodě *OtocStranku* třídy *Kniha* je pak typu *Listovani* a nabývá pouze hodnot *Vpřed* či *Vzad*.

- **Element Komentář (Note)**

Poznámka může být umístěna v diagramu buď zcela libovolně nebo může být připojena k elementu pomocí vazby typu souvislost (comment link). V druhém případě může být i poznámka doplněna o stereotyp (např. <<*constraint*>> by znamenalo specifikaci omezení). Při generování zdrojového kódu se však přítomnost poznámky v diagramu nijak neuplatňuje.

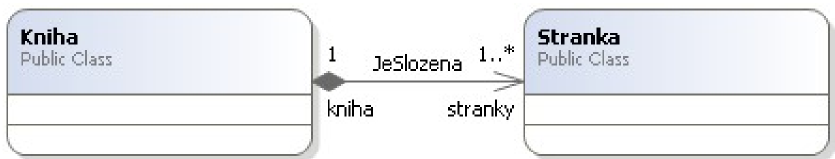
3.4.2 Vztahy v diagramu tříd

Relace umožňuje ukázat na modelu, jaký je vztah mezi dvěma entitami. Explicitní analogií role, kterou relace hrají v modelech UML, je např. rodina a vztahy mezi jejími jednotlivými členy. Relace umožňují zachytit významový (sémantický) vztah mezi elementy (Arlow, a další, 2007). Vztahují se na strukturní abstrakce a seskupování a jsou znázorněny v níže uvedené tabulce. Nesmírně důležitou součástí modelování v jazyku UML je porozumění přesné sémantice různých typů relací. Její stručný výklad bude předmětem následujícího textu.

Vztah (relace)	UML notace	Popis
Asociace (Association)		Symetrický vztah mezi elementy
Souvislost (Comment Link)		Související elementy (nebo připojení poznámky)
Závislost (Dependency)		Jeden element závislí na jiném elementu
Agregace (Aggregation)		Element obsahuje jiný element (vazba část-celek)
Kompozice (Composition)		Silnější typ agregace. Podřízený objekt nemůže samostatně existovat bez nadřízeného
Generalizace (Generalization)		Zdrojový element je specializací cílového
Realizace (Realization)		Zdrojový element je realizací cílového (např. třída-rozhraní)

Tabulka 3- Obecné vztahy v UML
Zdroj: https://en.wikipedia.org/wiki/Class_diagram

- **Asociace** – znázorňuje vztahy mezi jednou či více třídami, které jsou abstrakcí množiny spojení mezi instancemi (objekty) těchto tříd. Asociační vazba a vazby z ní odvozené, tedy agregace a kompozice, bývá často doplněna o informace, které mohou mít při generování zdrojového kódu cílové platformy zcela zásadní význam. Jedná se především o tzv. **multiplicitu** vazby, která určuje, kolik výskytů jedné třídy může mít vztah k jednomu výskytu přiřazené třídy (Vrana, 2008). U agregačních či kompozičních vazeb s multiplicitou 1:N (resp. 1..* nebo 0..*) se v nadřazené třídě generuje deklarace kolekce instancí podřízené třídy (u modelového případu, který je uveden v praktické části a jehož kompletní diagram je v dostupný v příloze je to vztah Kniha – Stranka, kde *Kniha* obsahuje proměnnou *Stranky* typu *List<Stranka>*)



Obrázek 12- Role a multiplicita na vazbě typu kompozice
Zdroj: vlastní tvorba (aplikace dotCORN)

- **Generalizace** – (dědění) je vztah mezi obecnou objektovou třídou a jejími konkrétními potomky. Podřízené objektové třídy (subclass, child) dědí ze svého předka (superclass) všechny vlastnosti (atributy), operace a omezení. Rozšiřují svou nadřízenou objektovou třídu o nové vlastnosti a operace. V této souvislosti je na místě zmínit pojem **abstraktní**

třída. V objektovém modelování je to značně využívaný speciální typ třídy, kterému nebude ve vývojovém prostředí cílové platformy nikdy vytvořena její konkrétní instance. Její účel tkví ve zobecnění některých společných vlastností a operací modelovaných věcí (v UML je předmět modelování označován jako věc-thing) a jejich následné odvození do specializovaných tříd. (Kanisová, a další, 2007)

- **Agregace** – je jednou z nejčastěji se vyskytujících vazeb v modelování objektových tříd. Je to tranzitivní a antisymetrická relace (vlastnosti přecházejí z celku na součást, ne opačně). Vazba typu agregace říká, že jedna třída je součástí druhé, např. stránka v knize či položka na faktuře. (Vrana, 2008)
- **Kompozice** – je speciálním případem agregace, pro kterou platí dvě základní omezení. Součást může patřit pouze jednomu celku. Zánikem celku zaniká i součást. (Vrana, 2008)
- **Realizace** – vztah realizace je analogií vztahu generalizace s tím rozdílem, že vztah realizace se uplatňuje mezi rozhraním a třídou, která ho implementuje (realizuje). Jak bylo popsáno výše, rozhraní (interface) je třídou se speciálním stereotypem, nicméně vztah generalizace se zde neuplatňuje. Realizace je tedy vztah mezi dvěma elementy, ve kterých jeden realizuje (implementuje nebo provádí) chování, které určuje jiný modelový element.
- **Závislost -** je slabší forma vazby, která označuje, že jedna třída závisí na jiné, jelikož ji používá v určitém okamžiku. Jedna třída závisí na jiné, pokud je nezávislá třída parametrická proměnná nebo lokální proměnná metody závislé třídy.

4 Vlastní práce

Praktickou část této práce představuje aplikace naprogramovaná v jazyce C# v prostředí .NET pro operační systém Microsoft Windows, jež nese název *dotCORN*[26]. Díky svému zaměření by se dal tento software stručně charakterizovat jako velmi jednoduchý CASE, tj. nástroj, který v sobě spojuje nejen možnost objektově orientovaného návrhu aplikací s využitím unifikovaného grafického modelovacího jazyka UML, ale přidává také podporu pro generování zdrojového kódu pro jazyk C#. Potud se od obvyklých CASE nástrojů koncepčně příliš neliší. Co jej však činí jiným je integrace prostředků pro analýzu vygenerovaného kódu s využitím otevřené platformy .NET Compiler Platform (Roslyn). Na tomto místě lze již hovořit o zárodku edukativní pomůcky, která umožňuje v rámci jediného prostředí navrhovat, analyzovat, kompilovat a spouštět jednoduché aplikace, aniž by bylo nutné sáhnout po standardním vývojovém prostředí pro jazyky z rodiny .NET, které však v danou chvíli nemusí být vůbec k dispozici.

4.1 Uživatelské rozhraní

Aplikace *dotCORN* je kromě běžných vizuálních komponent dostupných ve Frameworku .NET 4.6 (tlačítka, nástrojové lišty, roletové a kontextové nabídky, editační prvky atd.) vystavěna na čtyřech stěžejních ovládacích prvcích uživatelského rozhraní[27], které definují nejen vzhled, ale především způsob interakce uživatele se systémem. Ačkoli byl u třech z nich vývoj a podpora ukončena, jedná se o relativně populární, komunitou vývojářů prověřené a technologicky pokročilé open-source komponenty (Fogel, 2005), jejichž zdrojové kódy jsou dostupné prostřednictvím webové služby GitHub. Jedná se o následující ovládací prvky:

- **WeifenLuo DockPanel Suite**

Knihovna WeifenLuo umožňuje poměrně snadným způsobem vytvářet flexibilní dokovací aplikace se stejně komplexním rozhraním, jaké má například IDE MS Visual Studio, Enterprise Architect, AutoCAD LT a řada dalších současných vývojových prostředí. Vývoj knihovny probíhal v letech 2006 – 2017, nicméně pro svou oblibu, stabilitu, jednoduchost použití a příležitostnou podporu ze strany DSO[28] je knihovna stále stahována a integrována do softwarových řešení amatérských i profesionálních programátorů.

[26] Prefix *dot* odkazuje na použití v prostředí .NET. *CORN* je akronym složený ze počátečních písmen slov Classes, Objects, Relations, Notations.
[27] V terminologii .NET se používá anglického označení UI Controls.
[28] DSO – The DockPanel Suite Organization

Myšlenka zónového uspořádání panelů, resp. „plovoucích" oken v prostředí .NET WinForms (Griffiths, et al., 2003) je založena na použití tzv. dokovacích kontajnerů třídy *DockPanel* a oken třídy *DockContent*. Dokovací kontejnery pak mohou vnořená okna formovat do různých uspořádání jako např. záložky podobné komponentě *TabControl.* Kontejnery typu *DockPane* uspořádají podřízená okna ve vodorovném či svislém směru apod. Okna také disponují funkcí automatického skrytí (ikona špendlíku v pravém horním rohu záhlaví okna), resp. automatického zviditelnění. Výsledná podoba pracovní plochy pak do značné míry závisí na potřebách či kreativitě uživatele.
(Zdroj: http://dockpanelsuite.com/)

Př.2 Vytvoření a zadokování okna s UML diagramem z hlavního formuláře aplikace.

```
/*DiagramUmlClassDiagram.cs */

public partial class DiagramUmlClassDiagram : DockContent

/* MainForm.cs */

DiagramUmlClassDiagram umlClassDiag = new DiagramUmlClassDiagram();
umlClassDiag.AllowEndUserDocking = false;
umlClassDiag.CloseButtonVisible = false;
umlClassDiag.Show(dpSite, DockState.Document);
```

Knihovna **WeifenLuo** ve verzi 2.4.0, která je součástí praktické části této práce nabízí ve jmenném prostoru `WeifenLuo.WinFormsUI.Docking` pouze jednu vizuální komponentu, kterou je *DockPanel*. Pro dosažení požadovaného efektu dokovacích a plovoucích oken se postupuje v několika krocích. Nejprve se přidá do projektu standardní formulář (Project -> Add Windows Form…). Následně se ve zdrojovém kódu tohoto formuláře přejmenuje nadřazená třída *Form* na *DockContent*, čímž se vzhled i vlastnosti formuláře za běhu aplikace podstatným způsobem změní. Implicitní pozice a zobrazení dokovacího okna se určí volbou jednoho ze šesti přetížení metody *Show* třídy *DockContent*.

- **NClass – Free UML Class Designer**

NClass je bezplatný open-source nástroj, jehož hlavním cílem je poskytnout jednoduchý, ale výkonný návrhář diagramu tříd dle standardu UML, který má velmi intuitivní ovládání a obsahuje všechny elementy a relace, které jsou pro modelování pomocí diagramu tříd podstatné. Předdefinované grafické styly umožňují tvorbu profesionálně vypadajících diagramů, stejně jako je tomu např. ve Visual Studiu nebo jiných komerčních produktech.

NClass je kompletně napsaný čistě v jazyce C#, tj. bez volání služeb operačního systému či použití nezabezpečeného kódu. Knihovna je také dostupná i pro platformu Mono v operačním systému Linux.
Základní charakteristiky komponenty NClass lze shrnout do několika následujících bodů:

- Plná podpora jazyků C# a Java při generování kódu z diagramu
- Snadné pochopení a intuitivní ovládání
- Konfigurovatelné styly diagramu
- Ukládání a načítání diagramu v otevřeném formátu XML
- Tisk, náhled před tiskem a konfigurace tiskárny
- Export diagramu do PDF
- Uložení diagramu do některého z běžných grafických formátů (PNG, JPEG, BMP, WMF…)

NClass je koncipována jako prostředí pro multiprojektové a multidiagramové modelování. Pro jednoduchost se však tato práce omezuje pouze na situaci, kdy existuje jeden projekt a v rámci něj pouze jeden diagram.
(Zdroj: http://nclass.sourceforge.net/)

- **TreeViewAdv for .NET**

Tato komponenta podstatným způsobem rozšiřuje funkcionalitu běžného *TreeView*, který je s úspěchem v této práci použit např. pro zobrazení stromové struktury projektu (*DiagramProjectBrowser.cs*) nebo v okně nástrojů (prvků a vazeb) pro návrh diagramu tříd (*DiagramToolbox.cs*). *TreeViewAdv* je založen na návrhovém vzoru Model-View-Controller a je silně ovlivněný podobnou komponentou z knihovny uživatelských prvků Swing platformy Java.
Mezi významné vlastnosti *TreeViewAdv* patří především:

- Podpora sloupců (kombinace stromu a tabulky)
- Neomezený počet editačních prvků v uzlu stromu
- Oddělení dat a zobrazovací logiky (MVC)
- Čistý C# kód bez volání Windows API

(Zdroj: https://sourceforge.net/projects/treeviewadv/)

- **ScintillaNET**

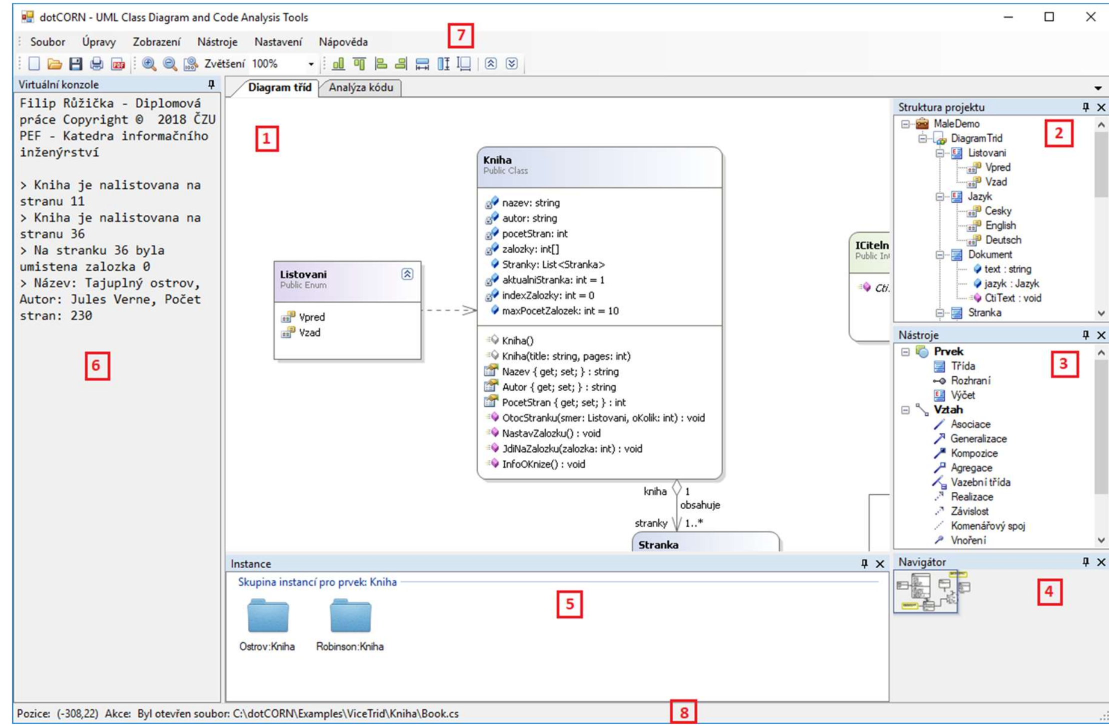
Textový editor Scintilla je jedinou ze všech čtyř zmiňovaných fundamentálních komponent, která je trvale intenzivně vyvíjena. Tento editor samozřejmě obsahuje všechny standardní funkce, které jsou běžné v komponentách pro úpravy textu, nicméně je vybaven i poměrně širokým portfoliem funkcí užitečných zejména při editaci a ladění zdrojového kódu. Mezi tyto funkce patří podpora různých stylů syntaxe, indikátory chyb, dokončení kódu a tipy pro volání. Okraj editoru může obsahovat různé značky užívané v nástrojích pro ladění, např. breakpointy, záložky, označení aktuálního řádku, čísla řádků apod. Editor také dokáže rozpoznat logické bloky kódu, popř. explicitně vyznačený region a nabízí funkci code-foldingu[29] v podobě expandovatelného uzlu. Samozřejmostí je podpora mnoha fontů, současného použití více barev pozadí i písma apod.

ScintillaNET nabízí především možnost otevření a editace více dokumentů. Této vlastnosti využívají např. editory Notepad++, SciTE, Code::Blocks, Anjuta aj., které jsou na editoru Scintilla postavené. Podobně jako v případě komponenty NClass je v této práci požit editor Scintilla bez multidokumentové podpory.

(Zdroj: https://github.com/jacobslusser/ScintillaNET)

[29] Díky code-foldingové funkci je možné logické bloky zdrojového kódu "skrýt" pro větší přehlednost v průběhu editace zdrojového kódu.

4.1.1 **Základní ovládací prvky**



Obrázek 13- Ovládací prvky na záložce "Diagram tříd"
Zdroj: vlastní tvorba

1. Diagram tříd

   Diagram tříd představuje plochu (canvas) komponenty NClass, která je umístěna v dokovacím okně *DiagramUmlClassDiagram.cs*. Aby bylo možné na plochu umísťovat prvky z okna „Nástroje“, musí být založen nový projekt a nový diagram, popř. musí být projekt s diagramem otevřen z již existujícího souboru. Okno diagramu tříd nelze zavřít, skrýt, přemístit ani z něj udělat okno plovoucí. Společně s oknem pro analýzu kódu je součástí tzv. *TabContaineru*.

2. Struktura projektu

   Strukturou projektu se zde rozumí seznam prvků v projektu a jejich atributů a operací. Struktura nevyjadřuje závislosti či vazby mezi jednotlivými prvky. Je to pouze jakási osnova projektu, která usnadňuje navigaci v diagramu, neboť kliknutím myší na uzel s názvem elementu dojde v diagramu k jeho zvýraznění.

3. Nástroje – prvky a vazby v diagramu

   Jak již bylo zmíněno v bodě 1, prvek či vazba lze umístit pouze do existujícího aktivního diagramu v rámci existujícího projektu. Umístění se provádí přetažením vybraného prvku na plochu diagramu (drag-and-drop, tedy táhni a pusť). Vazba mezi dvěma prvky

se tímto způsobem nevytváří. V tomto případě stačí kliknout myší na typ požadované vazby (ve stavovém řádku se o tom objeví zpráva) a pak najet kurzorem myši na zdrojový prvek a kliknout, poté přejet nad cílový prvek a opět kliknout. Pokud má vazba mezi těmito dvěma prvky smysl (ne vždy tomu tak musí být), pak se vytvoří i její grafická podoba.

4. Navigace v rozsáhlém diagramu

   Navigátor je nástroj přímo obsažený v knihovně NClass, jehož základem je diagram zmenšený zhruba na 10% své původní velikosti. Navigaci lze pak přirovnat k prohledávání mapy pomocí lupy. Každý region či zóna zaměřená v okně Navigátoru pomyslnou „lupou“ je pak v diagramu zobrazena ve velikosti určené hodnotou parametru Zoom.

5. Kontajner instancí

   Je dokovací okno *DiagramInstanceContainer.cs* s komponentou *ListView*, ve které jsou zobrazeny a do skupin uspořádány položky, které představují dynamicky získané typy a instance tříd vytvořené z elementů typu třída v UML diagramu. Každá položka (*ListViewItem*) v kontajneru nese v property **Tag** informaci o typu dynamicky získaného z assembly kompilací zdrojového kódu elementu, instanci třídy a seznam konstruktorů (pokud existuje). V procesu získávání těchto informací se uplatňuje řada technik a mechanismů z .NET Compiler Platform a reflexe.

6. Virtuální konzole

   Virtuální konzolí je v této práci nazýváno okno *OutputWindow.cs*, které obsahuje komponentu *TextBox* ze jmenného prostoru `System.Windows.Forms`. Do tohoto okna, resp. jeho textové komponenty jsou přesměrovány veškeré výstupy volání metod *Console.Write* a *Console.WriteLine*. Týká se to nejen výstupů, které jsou součástí aplikace jako takové, ale také skriptů jazyka C#, které jsou vykonávané prostřednictvím Scripting API Roslynu. Dále do tohoto okna směřuje většina výjimek zachycených v *try-catch* blocích, výsledky diagnostiky kompilace a emitování assembly a v neposlední řadě také výstupy volání metod v inspektoru instancí. Všechny ostatní informační, varovné, chybové nebo dotazovací dialogy se zobrazí standardní formou *MessageBox*u. Stavové informace jsou přes property *UserActionStatus* v *MainForm.cs* nasměrovány do stavového řádku hlavního okna (viz. bod 8).

Př.3 Vlastní přesměrování aplikační konzole se zajistím celkem snadno změnou parametru *TextWrite* metody *SetOut* třídy *System.Console*.

```
/* Program.cs */

static void Main()
{
 using (var consoleWriter = new ConsoleWriter())
 {
  consoleWriter.WriteEvent += EditorHelper.consoleWriter_WriteEvent;
  consoleWriter.WriteLineEvent += EditorHelper.consoleWriter_WriteLineEvent;
  Console.SetOut(consoleWriter);
  Application.EnableVisualStyles();
  Application.SetCompatibleTextRenderingDefault(false);
  Application.Run(new MainForm());
  }
}

/* EditorHelper.cs */

public class ConsoleWriter : TextWriter
{
  public override Encoding
  {
     get { return Encoding.UTF8;  }
   }
  public override void Write(string value)
  {
    WriteEvent?.Invoke(this, new ConsoleWriterEventArgs(value));
    base.Write(value);
   }
   public override void WriteLine(string value)
   {
     WriteLineEvent?.Invoke(this, new ConsoleWriterEventArgs(value));
     base.WriteLine(value);
    }
 public event EventHandler<ConsoleWriterEventArgs> WriteEvent;
 public event EventHandler<ConsoleWriterEventArgs> WriteLineEvent;
}
```

Klíčem k řešení je vytvoření potomka *ConsoleWriter* třídy *TextWriter* a přepsání jeho virtuálních metod *Write* a *WriteLine* a následná implementace vlastních *handlerů*, které budou zpracovávat události *WriteEvent* a *WriteLineEvent* v instanci třídy *ConsoleWriter*. Instance consoleWriter musí být přiřazena do property *Console.Out* metodou *Console.SetOut* před spuštěním aplikace, tedy před voláním metody *Application.Run*.

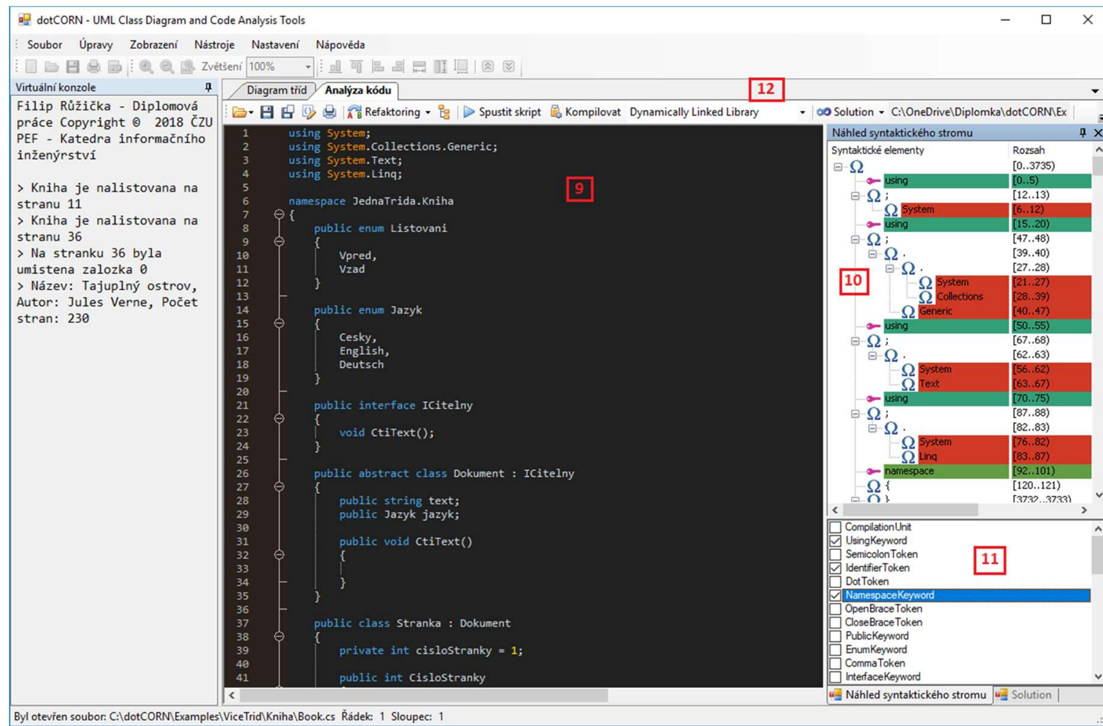
7. Hlavní menu a nástrojová lišta diagramu

   Hlavní roletová nabídka a nástrojová lišta obsahují, až na výjimky, funkce spojené s ovládáním diagramu tříd. Jsou v ní soustředěny nástroje pro načítání, ukládání, tisk a export projektu a diagramu. Dále jsou to funkce pro manipulaci s elementy a vazbami v diagramu jako je kopírování, mazání, vyjmutí, vložení, změna měřítka (Zoom),

zarovnávání apod. V položce nástroje je volba pro alternativní způsob vytváření prvků diagramu, a především operace generování zdrojového kódu z modelu tříd. Aplikace je také připravena na lokalizaci do angličtiny a změna jazyka je možná volbou Nastavení -> Jazyk -> Angličtina. Lokalizace je realizována pomocí nástroje **ResX Manager** integrovaného do vývojového prostředí MS Visual Studio. (https://marketplace.visualstudio.com/items?itemName=TomEnglert.ResXManager)

8. Stavová lišta

   Stavová lišta slouží především ke zobrazování informací, které mají povahu uživatelských akcí nad diagramem nebo v editoru zdrojového textu na záložce „Analýza kódu“. Texty zpráv jsou uloženy v řetězcových konstantách v dokumentu *GlobalSettings.cs* a dle svého určení mají prefix `DiagramAction_` resp. `EditorAction_`. Ve stavové liště je zobrazena poslední operace, kterou uživatel provedl. Pokud by byly tyto zprávy ukládány do souboru, vznikl by jednoduchý log, tedy seznam činností, kterými se uživatel dostal do aktuálního stavu. Zprávy by bylo také možno směrovat do Virtuální konzole (viz. bod 6)



Obrázek 14- Ovládací prvky na záložce "Analýza kódu"
Zdroj: vlastní tvorba

9. Editor zdrojového kódu

   Editor kódu je založen na nastylované komponentě ScintillaNET. Do něj je směrován výstup z generátoru kódu nebo v něm uživatel píše skript v jazyce C# (viz. Scripting API). Editor podporuje veškeré běžné funkce, které editory textu mají. Navíc však poskytuje barevné zvýraznění syntaxe a automatické doplnění kódu příkazů jazyka C#.

10. Náhled syntaktického stromu

    Syntaktický strom lze volitelně zobrazit kliknutím na tlačítko se symbolem stromové struktury v nástrojové liště. Pokud je v editoru smysluplný kód, dojde k sestavení syntaktického stromu a jeho zobrazení prostřednictvím komponenty *TreeViewAdv*. Jednotlivé uzly jsou označeny ikonou červeného klíče, pokud jde o klíčové slovo, popř. modrým řeckým písmenem omega, pokud jde o token. Existují však i další symbolická označení uzlů (viz. 4.3.1). Strom se chová interaktivně, tj. pokud uživatel kliká na uzly stromu, příslušné klíčové slovo či token se zvýrazní v editoru. Strom je zobrazen v infixové notaci.

Př.4 Využití informace **Span** ze syntaktického stomu.

```
private void twSyntaxTree_MouseClick(object sender, MouseEventArgs e)
{
  if (e.Button == MouseButtons.Left)
  {
   TreeNodeAdv node = twSyntaxTree.GetNodeAt(e.Location);
   if (node != null)
   {
   Point spanSelection = GetElementSpan(((TreeViewRowNode)node.Tag).Span);
 AnalyzeCodeAnalysis.Instance.ScintillaEditor.SelectionStart = spanSelection.X;
   AnalyzeCodeAnalysis.Instance.ScintillaEditor.SelectionEnd = spanSelection.Y;
   AnalyzeCodeAnalysis.Instance.ScintillaEditor.Select();
   }
  }
}
```

Span je řetězec ve tvaru *[start..end)*, ve kterém je *start* počáteční a *end* koncovou pozicí syntaktického prvku v rámci zdrojového textu. Funkce *GetElementSpan* tyto hodnoty extrahuje z řetězce a transformuje je do proměnné typu *Point*. Následně se tyto hodnoty použijí pro naplnění vlastností editoru **SelectionStart** a **SelectionEnd**. Metoda *Select* pak provede barevné zvýraznění takto pozičně vymezeného textu, kterému odpovídá prvek v syntaktickém stromu.

11. Lexémy a tokeny

    Při konstrukci syntaktického stromu se vytváří seznam typů elementárních syntaktických prvků – lexémů. Každý uzel syntaktického stromu nese o tomto typu informaci v property **Kind**, která je výčtového typu *SyntaxKind* ze jmenného prostoru `Microsoft.CodeAnalysis.CSharp`. Tento seznam je pak deduplikován a zobrazen formou seznamu označovacích polí - komponentou *CheckedListBox*, která je běžně dostupná ve Frameworku .NET. Uživatel může označit „zaškrtnutím“ libovolné množství položek a toto označení barevně zvýrazní veškeré výskyty zvoleného prvku ve struktuře syntaktického stromu. Barvy pro odlišení jsou generovány náhodně, a tudíž jsou při každém spuštění aplikace jiné.

12. Nástrojová lišta pro operace s kódem

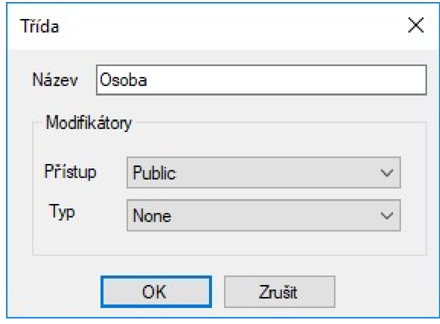
    Na této liště se nacházejí ovládací prvky (tlačítka, textové pole, pole pro výběr prvku), které se vztahují ke zdrojovému textu v editoru. Jsou to zejména funkce pro načtení či uložení textu ze souboru, uložení kódu do elementu před vytvořením jeho instance, načtení, zobrazení či kompilace struktury Solution, kompilace zdrojového kódu, spuštění skriptu, vyhledávání řetězce znaků v textu atd.

4.2 Diagram tříd

V předkládané verzi aplikace poskytuje diagram tříd pro tvorbu modelu tři základní UML elementy (třídu, rozhraní a výčet) a sedm typů vazeb (asociace, generalizace, agregace, kompozice, realizace, závislost a komentářový spoj). Doplňujícím prvkem je pak komentář (poznámka). Komponenta NClass sice nabízí navíc elementy stereotypu struktura a delegát, které však nejsou v aplikaci *dotCORN* využity, stejně jako vazba typu vnoření (Nesting).
Prvky diagramu lze vytvořit dvěma různými způsoby. Oba byly v předchozích kapitolách již zmíněny. Umístění prvku na plochu diagramu lze provést jeho přetažením myší z panelu „Nástroje" nebo volbou nabídky Nástroje -> Nový z hlavního roletového menu aplikace. Celou akci lze urychlit použitím odpovídajících klávesových zkratek, které jsou uvedeny u jednotlivých položek v submenu.

4.2.1 Elementy Třída, Rozhraní a Výčet

Ať již uživatel vytváří třídu, rozhraní či výčet operací „táhni a pusť", volbou z menu nebo kombinací kláves, vždy je vytvoření elementu rozděleno do dvou kroků, které předcházejí úplnému vykreslení prvku na plátno diagramu. V prvním kroku uživatel specifikuje především název prvku a přístupový modifikátor (Access Modifier), tj. *Default, Public, Protected Internal, Internal, Protected* nebo *Private*. Pokud je vytvářeným prvkem třída, pak lze určit její typ (Class Modifier), kterým je jeden z následujících: *None, Abstract, Sealed* a *Static*. Na následujícím obrázku je formulář pro editaci záhlaví všech tří implementovaných prvků, tj. třídy, rozhraní a výčtu. Nastavení typu je v případě rozhraní a výčtového typu uživateli zamezeno, neboť dle specifikace UML nedává smysl. Všechny tři prvky mají implicitně nastavený přístupový modifikátor na hodnotu *Public*.



Obrázek 15- Editace záhlaví třídy
Zdroj: Vlastní tvorba

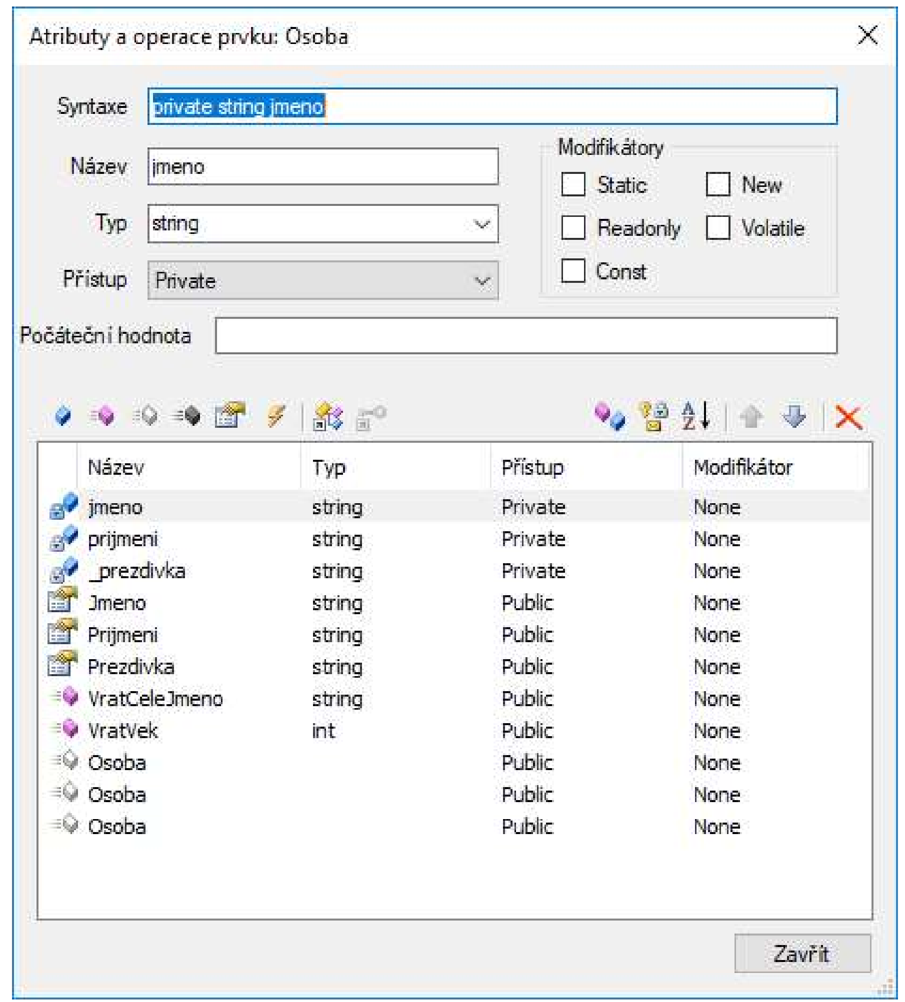
Pokud dojde k potvrzení údajů ve formuláři stiskem tlačítka OK, pak se v závislosti na typu vytvářeného prvku otevře formulář pro definici atributů a operací v případě třídy a rozhraní,

resp. pro zadání prvků výčtu u výčtového typu. Ačkoli je editor atributů a operací pro třídu i interface zdánlivě totožný, i zde opět existují pro rozhraní principiální omezení. To vyplývá především z podstaty rozhraní jakožto prvku, který předepisuje určité chování a třída, která interface realizuje se zavazuje toto chování implementovat. Chování objektů je určeno jejich metodami, resp. událostmi, na které reagují při výměně zpráv. Proto je výběr možných členů rozhraní omezen na metody, vlastnosti a události. Taktéž je odepřena možnost měnit přístupový modifikátor členů rozhraní, neboť se u nich automaticky předpokládá, že jsou vždy veřejná, tedy *Public*.

U tříd je samozřejmě dostupná úplná nabídka členů, tj. polí, metod, vlastností, událostí, konstruktorů a destruktorů. Jak v případě třídy, tak i rozhraní je možné specifikovat modifikátor členu, případně kombinace modifikátorů, které jsou syntakticky validní. Kontrola syntaktické i sémantické správnosti definice členů třídy či rozhraní je striktně vynucována a garantuje tím mimo jiné formální korektnost, která má klíčový význam při generování zdrojového kódu z diagramu.

Za povšimnutí též stojí definice typu atributu, resp. návratové hodnoty operace. Na následujícím obrázku je toto pole označeno jako „Typ“ a realizováno komponentou pro výběr prvku ze seznamu (*ComboBox*). Seznam však není omezen na základní datové typy jazyka C#. Pole je editovatelné a seznam dostupných typů se rozšiřuje s každým uživatelským zadáním nového typu, ať už se jedná o typ hodnotový, referenční, výčtový apod.

V editačním poli „Syntaxe“ vzniká náhled konečné podoby definice atributu či operace. Chce-li uživatel definovat parametrický tvar metody či konstruktoru, je nutné toto rozšíření provést právě v tomto poli, přičemž syntaktická kontrola provedené modifikace zůstává opět zachována.



Obrázek 16- Editace atributů a operací třídy
Zdroj: Komponenta NClass – MembersDialog.cs (upraveno autorem)

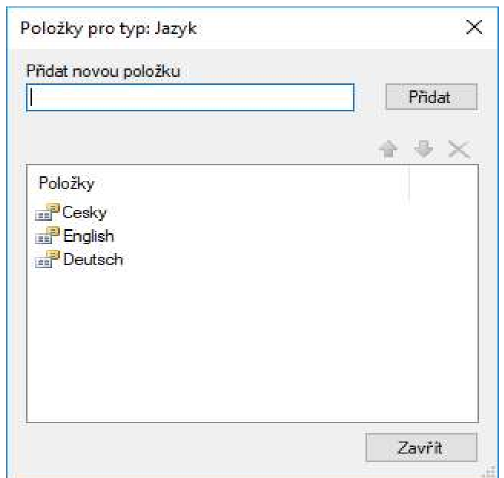
Př.5 Vytvoření elementu třída operací „táhni a pusť“ z okna „Nástroje“

```
private void canvas_DragDrop(object sender, DragEventArgs e)
{
  string actionResult = string.Empty;
  var draggedItem = e.Data.GetData(typeof(TreeNode));
  if (draggedItem is TreeNode) {
  switch (Convert.ToInt16((draggedItem as TreeNode).Tag.ToString()))
  {
// Trida
    case ToolBoxItem.ToolBoxClass:
      if (MainForm.Instance.ActiveDiagram != null)
      {
       ClassType classType = MainForm.Instance.ActiveDiagram.AddClass();
       classType.Name = $"Class{GetEntityCount(EntityType.Class)}";
       foreach (Shape shape in MainForm.Instance.ActiveDiagram.Shapes)
       {
     if (shape.Entity.Name.Equals(classType.Name))  {
        shape.Location = Canvas.PointToClient(new Point(e.X, e.Y));    }
    }
    MainForm.Instance.ActiveDiagram.InsertClass(classType);
    if (ShowClassHeaderEditor(classType)) {
```

```
            DiagramProjectBrowser.Instance.UpdateProjectTree(classType, onlyLabel:
          true, newElement: true);
           actionResult = classType.Name;
           MembersDialog membersDialog = new MembersDialog();
           membersDialog.ShowDialog(classType);
           MainForm.Instance.IsCodeGenerated = false;
           DiagramProjectBrowser.Instance.UpdateProjectTree(classType);
      }
      else { return; }
     }
     break;
    }
```

Každý uzel stromu v okně „Nástroje" obsahuje v property **Tag** konstantu, která jej v rámci hierarchické struktury kategorizuje a dle její hodnoty se rozhoduje, jak bude s uzlem manipulováno. Např. uzly s konstantou *ToolBoxClass*, *ToolBoxInterface* a *ToolBoxEnum* půjdou přetáhnout myší na plochu diagramu, jiné mění způsob chování diagramu, některé představují dosud neimplementovanou funkcionalitu. Voláním funkce *GetEntityCount* se zjistí nejvyšší dosud nepoužitý index daného typu entity, v tomto případě třídy a použije se pro sestavení unikátního jména třídy. Metoda *InsertClass* vytvoří třídu v repozitáři entit diagramu a vykreslí ji na plochu. Podstatné je nastavení property **IsCodeGenerated** na logickou hodnotu *false*, což bude rozvedeno dále v kapitole 4.2.3

V případě definice atributů výčtového typu je situace o poznání jednodušší. U nich se nedefinuje datový typ ani přístupový modifikátor. Atributy se pak zadávají pouze názvem, jak je patrné z následujícího obrázku.



Obrázek 17- Editace atributů výčtu
Zdroj: Komponenta NClass – EnumDialog.cs (upraveno autorem)

Na tomto místě je však třeba s obecnou platností připomenout, že značně rizikovým faktorem, který se uplatňuje při pojmenování libovolného elementu diagramu či jeho členů, a to

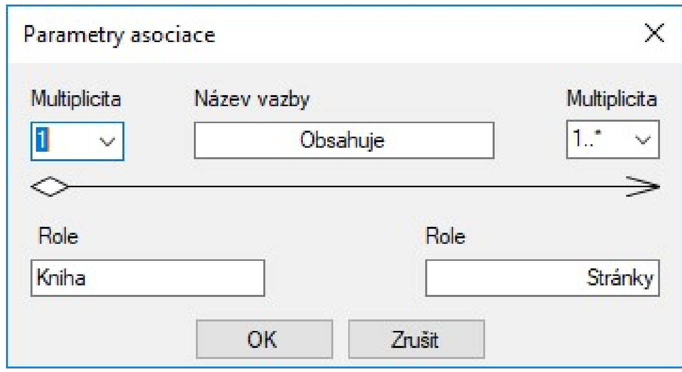
především s ohledem na budoucí generování zdrojového kódu, je používání diakritických znamének v názvech. V případě, že uživatel vytváří pouze doménový model, aniž by aspiroval na následnou analýzu kódu pořízeného z diagramu není užití národních abeced na závadu. V opačném případě dochází k problémům při kompilaci kódu, neboť překladač vyhodnotí znaky opatřené diakritikou jako nedovolené a překlad je ukončen s chybou.

4.2.2 **Relace v diagramu tříd**

Definice a význam vztahů mezi prvky UML diagramu tříd byly objasněny v teoretickém partu této práce v kapitole 3.4.2. Jak již bylo uvedeno, v aplikaci *dotCORN* je prostřednictvím knihovny NClass k dispozici celkem sedm typů vazeb. Cílem tohoto odstavce tedy bude stručně představit postup vytvoření vazby mezi prvky na ploše diagramu a v případě některého z asociačních typů vazby (asociace, agregace, kompozice) ukázat, jak jej opatřit parametry, tj. názvem, pojmenováním rolí, definováním multiplicity pro každou stranu relace atd.

Propojení dvou elementů v diagramu se poněkud odlišuje od způsobů, které jsou uživatelům známé např. z profesionálních modelovacích nástrojů Enterprise Architect či Sybase PowerDesigner. Propojení se zde neprovádí operací „drag-and-drop“, ale označením požadovaného typu spojení ve stromu nástrojů diagramu v sekci „Vztah“. Aplikace ve stavovém řádku oznámí, že je připravena akci realizovat a změní chování plochy diagramu při pohybu kurzoru myši nad prvky, které nemusí být nutně předem označeny pro propojení. Výběr uzlů se pak provede velmi intuitivně - kliknutím na počáteční uzel spojení, který se zvýrazní modrým čárkovaným orámováním a následně volbou koncového uzlu, který bude před finálním konfirmačním kliknutím orámován červenou přerušovanou čarou a ze zdrojového uzlu na něj bude mířit šipka určující směr spojení.

U relací, které nejsou typu generalizace, realizace či komentářový spoj je na místě uvažovat o parametrech vazby, což je zřejmé z následujícího obrázku. V případě knihovny NClass se však bohužel jedná pouze o doplňující informace bez korelace s generovaným kódem. Naopak zmiňované vazby typu generalizace a realizace jsou v kódu naprosto korektně vytvářeny jako dědění. Navíc je během vytváření vazeb ošetřeno, aby úmyslně či nedopatřením nedocházelo k realizaci relací vedoucí k vícenásobné dědičnosti apod.



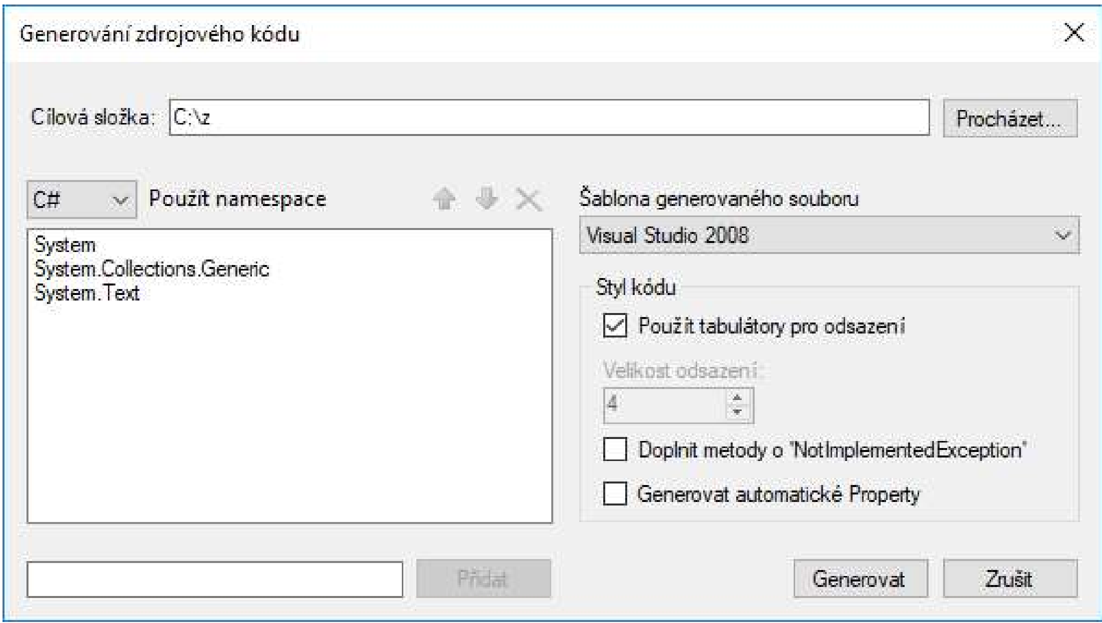
Obrázek 18- Nastavení parametrů vazby typu Agregace
Zdroj: Komponenta NClass – AssociationDialog.cs (upraveno autorem)

4.2.3 Generování zdrojového kódu

Pokud je reálná situace abstrahována a zachycena v modelu tříd, jehož elementy obsahují atributy a metody včetně datových typů vyhovujících syntaxi jazyka C#, otevírá se v aplikaci *dotCORN* řada možností, jak vytvořený model dále zhodnotit. Schopností odvodit, byť s jistými nedostatky, z diagramu tříd relevantní, syntakticky a sémanticky správný zdrojový kód, se vytváří průsečík mezi dvěma, zatím zdánlivě oddělenými světy – světem grafické reprezentace modelu a světem manipulace se zdrojovým kódem.

V této práci jsou k dispozici dva přístupy, jak zdrojový text z diagramu produkovat, avšak oba využívají stejný algoritmus implementovaný v knihovně NClass a částečně upravený autorem této práce. Pro správnou funkci obou dále uvedených přístupů platí nutná podmínka přítomnosti souborů *csproj.template* a *sln.template* v adresáři Templates, který musí být umístěn ve stejné složce jako spustitelný soubor aplikace dotCORN.exe. Generátor kódu nevytváří pouze samostatné soubory s kódem, ale dle zvolené šablony sestaví i projektový soubor s referencemi na dokumenty (soubory s extenzí .cs) a doplněný implicitními předvolbami pro kompilaci. Analogickým způsobem vytvoří také soubor pro řešení ve formátu odpovídajícímu struktuře uspořádání projektů v sadě MS Visual Studio (soubor .sln).

Prvním přístupem pro získání kompaktního řešení (*Solution*) z diagramu tříd je dialogy řízený proces spustitelný volbou „Nástroje -> Generovat zdrojový kód“ z hlavní nabídky programu. V úvodním dialogu jsou vyplněny základní parametry pro umístění vytvořených souborů na disku, seznamu použitých jmenných prostor, šablony souboru řešení atd.



Obrázek 19- Dialog pro generování kódu z diagramu tříd
Zdroj: Komponenta NClass – Dialog.cs (upraveno autorem)

Množství parametrů modifikujících způsob tvorby výsledného kódu by mohl být samozřejmě mnohem širší. Jako příklad lze uvést upravenou metodu, která převede **property** z elementu třída do zdrojového kódu.

Př.6 Rozšířený kód pro přepis **Property** třídy z diagramu do souboru zdrojového textu.

```
private void WriteProperty(Property property)
{
string pField = FieldsCollection.Where(x => (x.Access == AccessModifier.Private) &
& ((x.Name.ToLower().Equals(property.Name.ToLower())) ||
(x.Name.StartsWith("_") && (x.Name.TrimStart('_').ToLower().Equals(property.Name.T
oLower())))).FirstOrDefault().Name;

WriteLine("{");
IndentLevel++;
if (!property.IsWriteonly) {
  if (property.HasImplementation && !Settings.Default.GenerateAutoProperties) {
                         {
    WriteLine("get");
    WriteLine("{");
    if (!string.IsNullOrEmpty(pField)) {
         IndentLevel++;
         WriteLine($"return {pField};");
         IndentLevel--;
    }
    IndentLevel++;
    WriteNotImplementedString();
    IndentLevel--;
    WriteLine("}");
  }
  else  {
   WriteLine("get;");
```

```
  }
}
if (!property.IsReadonly)
{
  if (property.HasImplementation && !Settings.Default.GenerateAutoProperties) {
                                 {
    WriteLine("set");
    WriteLine("{");
    if (!string.IsNullOrEmpty(pField))
    {
      IndentLevel++;
      WriteLine($"{pField} = value;");
      IndentLevel--;
     }
     IndentLevel++;
     WriteNotImplementedString();
     IndentLevel--;
     WriteLine("}");
  }
  else {
    WriteLine("set;");
  }
 }
 IndentLevel--;
 WriteLine("}");
}
```

Lambda výraz, kterým metoda *WriteProperty* začíná, vyhodnocuje prvky v kolekci datových polí deklarovaných v rámci třídy a z nich vybírá takové, které mají privátní přístupový modifikátor a jejichž jméno je bez rozlišení malých a velkých písmen shodné se jménem property. Akceptované je i takové jméno pole, jež vyhovuje předchozí podmínce, a navíc je uvozeno znakem „podtržítko“. Pokud je takové jméno pole nalezeno (v případě, že by podmínce vyhovovalo více prvků kolekce, akceptuje se první nalezený) je použito pro *getter* a *setter* property, a to za předpokladu, že nebyla zvolena možnost „Generovat automatické Property“. Tyto modifikace, kterých je v kódu samozřejmě podstatně více jsou vedeny snahou minimalizovat množství kódu, které je nutné do vytvořeného skeletu doplnit, aby byl soubor kompilovatelný.

Pokud generátor vytvoří korektní kód, tedy včetně rozšíření cílové složky o podadresáře představující název projektu (zde je lokalizován soubor řešení .sln) a název diagramu (zde jsou umístěny dokumenty .cs a projektový soubor .csproj), je možné ihned přejít k editaci či analýze vzniklého kódu. Ten je vytvořen sloučením obsahu všech dokumentů v cílové složce, jejichž jména bez přípony odpovídají jménům entit v modelu tříd. V rámci vytvořeného jmenného prostoru (dle konvence *nazev_projektu.nazev_diagramu*) jsou pak všechny třídy, rozhraní, výčtové typy a další fragmenty kódu „uzavřeny“ do třídy **static class Program**. Klávesovou

zkratkou Ctrl+Shift+M je pak možné do této třídy doplnit vzor metody **public static void Main()**, čímž lze velmi expresně a efektivně prototypovat např. konzolovou aplikaci.

Př.7 Generování zdrojového kódu z hlavní nabídky aplikace.

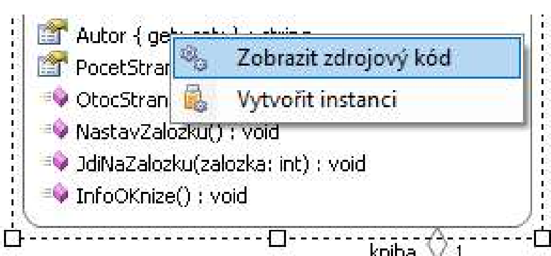
```
private void GenerateSourceCode()
{
  if ((ActiveDiagram != null) && (ActiveDiagram.Project != null))
  {
   using (Dialog codegenDialog = new Dialog())
   {
    try
    {
      codegenDialog.ShowDialog(ActiveDiagram.Project);
      if (!string.IsNullOrEmpty(codegenDialog.Path)) {
       if (!ElementImplementationExists())  {
         AssignImplementationToElements(codegenDialog.Path);
       }

      DialogResult dialog = MessageBox.Show("Přejete si vytvořený kód zobrazit v e
      ditoru?", "Otázka", MessageBoxButtons.YesNo, MessageBoxIcon.Question, Messag
      eBoxDefaultButton.Button1);

      if (dialog == DialogResult.Yes)     {
      List<string> selShapes = new List<string>();
      selShapes = activeDiagram.Shapes.Select(x => x.Entity.Name).ToList();
      string oneFile = FileHelper.MergeFiles(codegenDialog.Path, selShapes, ".cs",
       System.IO.SearchOption.AllDirectories);

      AnalyzeCodeAnalysis.Instance.ScintillaEditor.Text = oneFile;
      AnalyzeCodeAnalysis.Instance.Activate();
    }
   }
  }
  catch (Exception ex) {
        MessageBox.Show(ex.Message, "Chyba", MessageBoxButtons.OK, MessageBoxIcon.
        Error); }
    }
  }
}
```

Propojení grafického znázornění entity diagramu a zdrojového kódu (obecně libovolného textu), který je s ní souvztažný je zajištěno pomocí vlastností **Tag** a **FilePath**, o které byla entita rozšířena (abstraktní třída *DiagramElement* a tím pádem z ní všechny odvozené). Při generování kódu ve výše uvedeném příkladu se funkcí *ElementImplementationExists* testuje, zdali nějaká entita – prvek diagramu má v property **Tag** text již připojen. Pokud tomu tak není, pak se metodou *AssignImplementationToElements* vloží každému prvku diagramu do property **Tag** obsah vytvořeného souboru, který je ve svém názvu (bez cesty a přípony) shodný s názvem entity. Název souboru včetně své úplné cesty je pak uložen v property **FilePath**. Fakt, že grafický element diagramu nese textovou informaci o způsobu implementace v cílové

platformě má klíčový význam při analýze kódu, přesněji řečeno při vytváření instancí tříd, kterou nejsou abstraktní či statické.

Metoda *MergeFiles* ze statické třídy *FileHelper* pak výše popsaným způsobem spojí obsah všech vyprodukovaných dokumentů do zformátovaného řetězce, který může být dle volby uživatele zobrazen v editoru na záložce „Analýza kódu“.

Druhý způsob, jak získat zdrojový text k entitám bez nutnosti spouštět proces generování s dialogem předvoleb je možný přes kontextové menu, které se zobrazí kliknutím pravého tlačítka myši na označeném prvku diagramu. Pokud je tímto prvkem třída, která není specifikována jako abstraktní nebo statická, pak je situace stejná jako na následujícím obrázku.



Obrázek 20- Kontextové menu na elementu třída.
Zdroj: Vlastní tvorba (aplikace dotCORN)

Zobrazení zdrojového kódu se v tomto případě týká pouze vybrané entity. Nedochází při něm k žádnému spojování ani úpravám vygenerovaných textů. Uživatel je rovněž „odstíněn“ od procesů generování, které mohou probíhat na pozadí. Zde je rozhodující hodnota stavové proměnné **IsCodeGenerated**, která byla zmíněna v souvislosti s příkladem Př.5 kapitoly 4.2.1. Zatímco v prvém uvedeném případě dochází ke generování kódu vždy, zde pouze tehdy, pokud má entita property **Tag** rovnu hodnotě *null* nebo je proměnná **IsCodeGenerated** nastavena na hodnotu *false*. Toto přiřazení logické nuly nastane vždy, když je element diagramu vytvořen nebo je jakýmkoli způsobem modifikován (přidání, odebrání, změna atributu, metody atd.)

Ať už je nutné před zobrazením v editoru kód pro vybranou entitu generovat či nikoli, dojde automaticky ke změně okna z diagramu do úprav textu. Upravený kód entity, tj. obvykle doplněná implementace metod, vlastností a dalších funkčních bloků se do entity přenáší automaticky při změně okna „Analýza kódu“ na okno „Diagram tříd“. Připojení upraveného a doplněného kódu k entitě lze provést explicitně, kliknutím na tlačítko s ikonou složených závorek a pera. To se nachází v nástrojové liště editoru (viz. kapitola 4.1.1 ovládací prvek č.12). Při zaměření tohoto tlačítka kurzorem myši se objeví kontextová nápověda „Uložit implementaci“.

Kód entity, který se zobrazí v editoru po volbě „Zobrazit zdrojový kód“ je možné uložit do souboru nebo naopak ze souboru načíst. To má zcela kardinální význam, neboť rozsáhlé uživatelské úpravy vygenerovaného kódu mohou být ztraceny v okamžiku, kdy dojde ke změně v libovolném prvku diagramu. Aplikace nedokáže selektivně rozlišit, jak zásadní vliv změna přináší pro strukturu diagramu, a tudíž i pro relevanci kódu, proto vygeneruje nový kód pro všechny prvky diagramu.

Díky absenci dialogového okna s volbou cílové složky je drobnou odlišností od prvního způsobu vytváření kódu implicitní nastavení úložiště do dočasného adresáře, což je obvykle cesta „%AppData%\Local\Temp\“. V případě nouze je možné v těchto místech vyhledat vygenerované soubory, projekt i řešení ve struktuře, kterou poskytuje první varianta generování kódu.

Pokud byla pro vytvořený model tříd alespoň jednou volána operace „Nástroje -> Generovat zdrojový kód“, pak bude property **FilePath** každého elementu diagramu (nikoli vazby) obsahovat název souboru se zdrojovým kódem pro daný element. Obsah této property se serializuje do souboru, který popisuje strukturu diagramu pomocí značkovacího jazyka XML. Při deserializaci dochází k testování, zdali je uveden název souboru uvnitř párové značky <FilePath>. Pakliže takový soubor existuje, je otevřen a jeho obsah je automaticky přenesen do property **Tag**.

Př.8 Ukázka fragmentu XML entity třída se zvýrazněnou párovou značkou <FilePath>

```
<Entity type="Class">
    <Name>Kniha</Name>
    <Access>Public</Access>
    <Location left="277" top="53" />
    <Size width="270" height="366" />

     <FilePath>C:\dotCORN\Examples\ViceTrid\Kniha\MaleDemo\DiagramTrid\Kni
     ha.cs</FilePath>
     <Collapsed>False</Collapsed>
     <Member type="Field">private string nazev</Member>
     <Member type="Field">private string autor</Member>
</Entity>
```

Př.9 Deserializace entity a přenos obsahu souboru do property **Tag**.

```
protected virtual void OnDeserializing(SerializeEventArgs e)
{
    XmlElement pathNode = e.Node["FilePath"];
    if (pathNode != null)  {
      FilePath = pathNode.InnerText;
      if (File.Exists(FilePath)) {
        Tag = File.ReadAllText(FilePath);
      }
}
```

4.3 Analýza kódu

Analýzou kódu se bude v této a následujících kapitolách rozumět praktická aplikace některých prostředků z vybraných API platformy Roslyn. Již od prvních úvah o celkové koncepci této práce bylo stanoveno, že primárně bude kód určený k analýze vznikat v procesu forward engineeringu z modelu tříd, a to s plným vědomím existence problémů, které při generování kódu mohou nastat. Hlavním úskalím je totiž fakt, že modely UML obsahují informace, které nemohou být vyjádřeny v objektově orientovaných jazycích, zatímco objektově orientované jazyky vyjadřují implementační charakteristiky, které nemají protipól v modelech UML. Některé z těchto nedostatků byly zmiňovány v předchozích kapitolách, zejména v souvislosti s asociačními, agregačními či kompozičními vazbami mezi elementy diagramu tříd. V pokročilých CASE nástrojích mohou být sice simulovány ukazateli a odkazy, ale pak struktura systému není zřejmá, což často vede v průběhu dopředného inženýrství k rozporům mezi specifikací a kódem. Navíc algoritmus forward engineeringu implementovaný v knihovně NClass zcela opomíjí i tyto techniky mapování vazeb do konstrukcí objektově orientovaného jazyka C#. Zdrojový kód pro analýzu však v žádném případě nemusí být výsledkem výše uvedeného procesu. Do editoru lze manuálně či ze souboru vložit zcela libovolný kód v jazyce C#, kterým může být nejen program či třída vyhovující syntaktickým pravidlům jazyka, ale také skript či jen matematický výraz.

Ačkoli .NET Compiler Platform nabízí také analýzu kódu pro jazyk Visual Basic .NET, v této práci je uvažován pouze jazyk C# a analytické nástroje s ním spojené. Pro demonstraci praktického využití těchto nástrojů je v případě Roslyn verze 2.6.0 nutné registrovat a importovat následující jmenné prostory.

Namespace	Popis
`Microsoft.CodeAnalysis`	Základní namespace označovaný v Roslynu jako „All-In-One“
`Microsoft.CodeAnalysis.CSharp`	Překladač pro jazyk C#, tj. tvorba kompilačního objektu, syntaktického stromu, metod traverzování syntaktickým stromem a přepis jeho uzlů a mnoho dalších funkcí.
`Microsoft.CodeAnalysis.CSharp.Scripting`	Obsahuje metody pro asynchronní zpracování skriptů, tj. spouštění skriptů (*RunScriptAsync*), vyhodnocení výrazů (*EvaluateAsync*) apod.
`Microsoft.CodeAnalysis.CSharp.Syntax`	Speciální metody pro modifikaci syntakti ckého stromu (factory methods) zmíněné

	v kapitole 3.3.2 (např. *SyntaxFactory.ParseExpression*)
`Microsoft.CodeAnalysis.Emit`	Nastavení předvoleb emitování, shromažďování debuggovacích informací pro emitované sestavení, podpora techniky „Edit and Continue“
`Microsoft.CodeAnalysis.MSBuild`	Nástroje pro manipulaci s dokumenty, projekty a řešeními v rámci *MSBuildWorkspaces*.
`Microsoft.CodeAnalysis.Scripting`	Předvolby pro vytváření a spouštění skriptů, zpracování výsledků spuštěných skriptů, sledování hodnot proměnných ve skripu atd.

Tabulka 4- Jmenné prostory použité v praktické části práce
Zdroj: Vlastní tvorba

4.3.1 **SyntaxTree API**

Rozhraní SyntaxTree API ve vrstvě Compiler API představuje monumentální sadu tříd, rozhraní, enumerátorů, konstant, struktur a metod, které nabízejí široké možnosti pro práci se syntaktickým stromem, který reprezentuje esenciální element a vstupní bod pro analýzu kódu. Syntaktický strom je sestaven ze zdrojového kódu jazyka a každý jeho uzel definuje určitou konstrukci tohoto jazyka, ať už se jedná o klíčové slovo, deklaraci metody, proměnnou, datový typ, řetězec, závorky, mezery atd. Pro každý takto definovaný typ uzlu existuje metoda, která jej dokáže vyhledat, resp. zpracovat. Syntaktickým stromem se nejen traverzuje, ale jeho obsah i struktura lze modifikovat. Je třeba mít však stále na paměti, že syntaktický strom je neměnná struktura a její úpravou nedochází k modifikaci obsahu či struktury původního syntaktického stromu, ale vzniká strom nový.[30] Některé další charakteristiky syntaktického stromu byly popsány v teoretické části práce v kapitole 3.3.2.

Syntaktický strom je z obecného pohledu tvořen třemi typy prvků – syntaktické uzly, syntaktické tokeny a trivia. (Uhlenhuth, 2016)

- **Syntaktické uzly** (Syntax Nodes) jsou jedním z primárních prvků syntaktického stromu. Tyto uzly představují syntaktické konstrukce, jako jsou deklarace, příkazy, klauzule a výrazy. Každá kategorie syntaktických uzlů je reprezentována samostatnou třídou odvozenou od třídy *SyntaxNode*.

[30] Princip neměnnosti některých objektů (angl. immutability) je jedním ze základních principů v .NET Compiler Platform. Z témat uvedených v této práci se týká např. sémantického modelu, pracovních prostor (workspaces), kompilačních objektů atd.

- **Syntaktické tokeny** (Syntax Tokens) jsou terminální symboly jazykové gramatiky, představující nejmenší syntaktické fragmenty kódu. Nikdy nejsou rodiči jiných uzlů nebo tokenů. Syntaktické tokeny tvoří klíčová slova, identifikátory, literály a interpunkce. Jsou reprezentovány strukturou *SyntaxToken* implementující rozhraní *IEquatable<SyntaxToken>*.

- **Syntaktická Trivia** (Syntax Trivia) je struktura *SyntaxTrivia*, která představuje syntakticky nevýznamné informace, jakými jsou mezery mezi tokeny, direktivy pro kompilátor, komentáře apod. (Varty, 2014)

V této práci se aplikace nástrojů ze SyntaxTree API uplatňuje prakticky ve všech ukázkových příkladech (výjimkou je pouze demo využívající prostředků Scripting API). Není však možné teoretické znalosti o SyntaxTree API zhodnotit lépe než implementací jednoduchého syntaktického stromu, který bude konstruován na základě zadaného zdrojového kódu. Tento kód může být samozřejmě výstupem procesu generování kódu z diagramu tříd, čímž je opět podpořena správnost předpokladu o smysluplné symbióze modelovacího a analytického nástroje v jedné aplikaci.

Syntaktický strom

Syntaktický strom je přístupný klikem na tlačítko v nástrojové liště, které zobrazí po najetí kurzoru myši kontextovou nápovědu „Syntaktický strom“. Pokud ještě není otevřeno okno „Náhled syntaktického stromu“, pak se nejprve toto okno otevře a strom se ihned vygeneruje. Pokud již okno otevřené je, dojde pouze k sestavení stromu na základě zdrojového kódu v editoru.

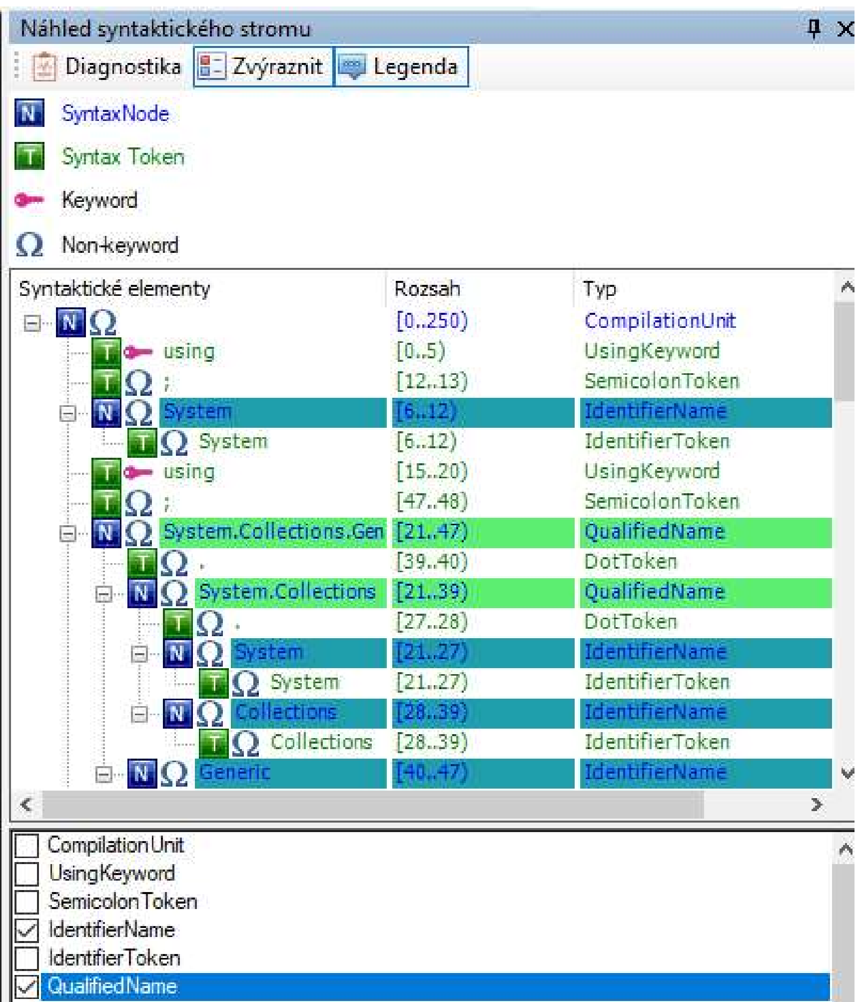
Nechť je dán následující zdrojový kód, který je vygenerován z modelu tříd demonstračního diagramu „MaleDemo“, který je uveden v příloze (Příloha A) této práce.

Př.10 Zdrojový kód abstraktní třídy *Dokument*.

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

namespace MaleDemo.DiagramTrid
{
        public abstract class Dokument : ICitelny
        {
                public string text;
                public Jazyk jazyk;

                public void CtiText()
                {

                }
        }
}
```

V příkladu je jmenný prostor `MaleDemo.DiagramTrid`, v němž je deklarována veřejná abstraktní třída *Dokument* implementující rozhraní *ICitelny*. V této třídě jsou dvě veřejná pole (jedno typu **string** a druhé výčtového typu *Jazyk*) a implementace metody *CtiText* předepsané v rozhraní *ICitelny*. Zdrojový kód v souboru *Document.cs* má 18 řádků.

Na následujícím obrázku je zřejmá struktura stromu. Každý uzel je označen dvěma symboly. Je-li prvním symbolem ikona bílého písmene „N“ v modrém poli, pak představuje syntaktický uzel (Syntax Node). Pokud je na první místě bílé písmeno „T“ v zeleném poli, je tímto uzlem stromu syntaktický token (Syntax Token). Ve struktuře je dobře patrné, že podřízené uzly mohou mít výhradně syntaktické nody (modré N), naopak syntaktické tokeny (zelené T) jsou listy stromu. Druhý grafický symbol informuje o tom, zdali je syntaktický prvek klíčovým slovem (symbol červeného klíče) či nikoli (symbol modrého řeckého písmene omega).



Obrázek 21- Syntaktický strom pro Document.cs
Zdroj: Vlastní tvorba (aplikace dotCORN)

Pro zjednodušení implementace i výslednou přehlednost nebyla do zobrazení zahrnuta *Trivia*. Součástí dokovacího okna se stromem je nástrojová lišta se třemi tlačítky. První tlačítko s titulkem „Diagnostika“ spouští jednoduchou diagnostiku syntaktického stromu. Ta spočívá ve volání metody *GetDiagnostics* na kořenovém uzlu stromu (je označován jako CompilationUnit) a zpracování jejích výsledků. Tlačítko „Zvýraznit“ má povahu stavového indikátoru. Pokud je aktivní (modrý rámeček tlačítka), pak mají nody syntaktického stromu kromě příslušných grafických symbolů identifikujících syntaktický uzel nebo syntaktický token ještě odpovídající barvou zvýrazněn text (modrá pro třídu *SyntaxNode* a zelená pro strukturu *SyntaxToken*). Tlačítko „Legenda“ má funkci přepínače a buď zobrazí nebo skryje panel s velice jednoduchou formou vysvětlivek ke způsobu označení uzlů stromu.

Př.11 Diagnostika syntaktického stromu

```
private void tsbDiagnostic_Click(object sender, EventArgs e)
{
  if (root != null) {
     bool err = false;
     foreach (Diagnostic diag in root.GetDiagnostics())
     {
        err = true;
        OutputWindow.Instance.VirtualConsole.AppendLine(diag.ToString());
        }
        if (!err) {
       OutputWindow.Instance.VirtualConsole.AppendLine("Tento syntaktický strom je
       v pořádku.");
       }
   }
}
```

Uživatel má také možnost volitelně barevně zvýrazňovat různé typy uzlů. Tato funkcionalita již byla poměrně podrobně popsána v kapitole 4.1.1 bod č.11. Její implementace zde nebude komentována, neboť přímo nesouvisí s analýzou kódu, ale pouze vizuálně usnadňuje orientaci ve stromové struktuře a nabízí rychlý přehled o zastoupení zvolených typů syntaktických elementů v kódu. Tato funkcionalita, stejně jako ostatní metody vztažené k sestavení a zobrazení syntaktického stromu jsou k nahlédnutí v souboru *AnalyzeSyntaxTreePreview.cs*. Vlastní implementace sestavení syntaktického stromu bez zohlednění trivií je pak poměrně snadná a je realizována metodou *ShowSyntaxTree*.

Př.12 Implementace syntaktického stromu pomocí rekurze – 1. část

```
public bool ShowSyntaxTree()
{
if (!string.IsNullOrEmpty(AnalyzeCodeAnalysis.Instance.ScintillaEditor.Text))
{
      SyntaxTree tree = CSharpSyntaxTree.ParseText(AnalyzeCodeAnalysis.Instance.Sc
      intillaEditor.Text);
  root = tree.GetRoot();
  SyntaxKind syntaxKind = root.Kind();
  availableKinds = new List<string>();
  availableKinds.Add(syntaxKind.ToString());

  foreach (SyntaxNode child in root.ChildNodes())
  {
    PrintSyntaxTree(child, rootNode);
  }
  return true;
}
return false;
}
```

Kód této metody je podstatně rozsáhlejší, neboť řeší vytváření uzlů v grafické komponentě *TreeViewAdv* a plní seznam typů uzlů *List<string> availableKinds*, který pak bude zobrazen

komponentou *CheckListBox* pro potřeby již zmiňovaného barevného odlišení syntaktických nodů a syntaktických tokenů. Podstatné na tomto fragmentu kódu je sestavení stromu voláním metody *ParseText* třídy *CSharpSyntaxTree*. Kořen takto sestaveného stromu je vždy třídy *SyntaxNode* (má vždy nějaké potomky) a je dostupný voláním metody *GetRoot*, jež má i svou asynchronní modifikaci *GetRootAsync*. V cyklu **foreach**, který prochází přes všechny potomky třídy *SyntaxNode* kořenového uzlu (root) je volána metoda *PrintSyntaxTree*.

Př.13 Implementace syntaktického stromu pomocí rekurze – 2. část

```
private void PrintSyntaxTree(SyntaxNode syntaxNode, TreeViewRowNode treeNode)
{
  if (syntaxNode != null) {
    foreach (SyntaxToken token in syntaxNode.ChildTokens())
    {
     InsertSyntaxTreeNode(SyntaxTreeNodeType.SyntaxTreeToken, token, treeNode);
    }
    foreach (SyntaxNode child in syntaxNode.ChildNodes())
    {
      TreeViewRowNode rNode = InsertSyntaxTreeNode(SyntaxTreeNodeType.SyntaxTreeNo
      de, child, treeNode);
       PrintSyntaxTree(child, rNode);
     }
    }
  }
```

V této metodě se procházejí kolekce tokenů (*ChildTokens*) i kolekce uzlů (*ChildNodes*) nodu, který byl předán v parametru *syntaxNode*. Tokeny, které nemohou mít žádné potomky se k vytvářenému uzlu *syntaxNode* pouze připojí jako uzly podřízené a konečné (listy). Pokud uzel *syntaxNode* obsahuje kromě tokenů i další syntaktické nody, rekurzivně se zavolá metoda *PrintSyntaxTree* a algoritmus se opakuje s novým uzlem třídy *SyntaxNode*. Metoda *InsertSyntaxTreeNode* zajišťuje správné umístění uzlu ve stromu a opatří ho odpovídajícím popisem a grafickými symboly.

V úvodu této kapitoly bylo s jistou hyperbolou řečeno, že možnosti Roslynu jsou značné a v případě SyntaxTree API tomu tak skutečně je. Možností, jak sestavit a vizuálně prezentovat syntaktický strom je celá řada. Velmi vhodným nástrojem pro tento typ úlohy je použití třídy *CSharpSyntaxWalker*, jejíž aplikace bude uvedena v následující kapitole, i když k poněkud jinému účelu.

Vytváření instancí z diagramu tříd

Tento part praktické části práce představuje určitý eklekticismus v rámci prostředků .NET Compiler Platform a jeho zařazení plně koresponduje s uvažovaným cílem práce, tj. s jejím nasazením jako didaktické pomůcky. Z uživatelského hlediska se jedná o interaktivní

validaci zpracované libovolné úlohy namodelované v diagramu tříd UML a doplněné o implementaci metod. Je zde agregována a uplatněna většina teoretických znalostí z dosud prostudovaných oblastí vrstvy Compiler API. Některé techniky použité při řešení této problematiky budou podrobněji popsány v následujících kapitolách. Tato subkapitola je však jakýmsi logickým pokračování a rozvinutím tématu generování zdrojového kódu z diagramu tříd v kapitole 4.2.3. Zde je na Obrázek 20 zobrazené kontextové menu vyvolané pravým tlačítkem myši nad zvolenou třídou diagramu. Pokud není třída specifikovaná jako abstraktní nebo statická, je součástí kontextové nabídky také položka „Vytvořit instanci“. Pojmenování položky je mírně zavádějící, neboť operace, která je touto akcí spuštěna neprodukuje přímo instanci třídy, ale v prvé fázi vrací typ objektu. Dynamické získávání typů objektů a vytváření instancí se zde nerealizuje pomocí nástrojů modelu CodeDOM[31], ale s využitím reflexe a prostředků platformy Roslyn.

Volba zmiňované položky „Vytvořit instanci“ předpokládá, že grafická reprezentace elementu třída v modelu tříd je také vybavena úplnou a bezchybnou implementací metod, vlastností a událostí, jejichž šablony byly vytvořeny v procesu generování kódu. Vzhledem k tomu, že je kód převeden do formy syntaktického stromu, parsován a kompilován, překladač by chyby detekoval a informoval o nich ve „Virtuální konzoli“.

Celá problémová doména získávání typů objektů a instancí je soustředěna do třídy/formuláře *DiagramInstanceContainer.cs*. Tento formulář (dokovací okno) obsahuje ovládací prvek *ListView*, který byl již zmíněn v kapitole 4.1.1 odstavec 5. Každá položka tohoto kontajneru obsahuje v property **Tag** instanci třídy *InstantiatedElement*, která má následující deklaraci.

Př.14 Deklarace třídy *InstantiatedElement*, jejíž instance je hodnotou property **Tag** položky *ListViewItem*.

```
public class InstantiatedElement
{
  public Type ElementType { get; set; }
  public object InstanceOfType { get; set; }
  public List<KeyValuePair<string, object>> ConstructorsList;
  public InstantiatedElement()
  {
    ConstructorsList = new List<KeyValuePair<string, object>>();
  }
 }
```

[31] CodeDOM – Code Document Object Model. Je mechanismus poskytovaný platformou .NET Framework pro generování a kompilaci kódu v různých jazycích (C#, JScript, Visual Basic .NET) s použitím jediného modelu. Mnohdy bývá označován jako prekurzor platformy Roslyn.

Třída obsahuje property **ElementType** typu *Type*. Tato property je za předpokladu konjunktivního splnění čtyř následujících podmínek naplněna hodnotou dynamicky získaného typu objektu. Tyto podmínky jsou:

- vytvoření instance třídy *CSharpCompilation* s parametrem syntaktického stromu zkonstruovaného ze zdrojového kódu entity diagramu
- emitování sestavení do paměti (*EmitResult* musí nabývat hodnoty *Success*)
- načtení paměťového proudu metodu *Load* třídy *Assemby (*reflexe*)*
- obdržení typu z instance třídy *Assembly* voláním metody *GetType* (reflexe)

Splněním podmínek se zde rozumí, že nebude vyvolána žádná z výjimek *IOException*, *ArgumentException, ArgumentNullException*, *BadImageFormatException* a volání metody *Emit* vrátí hodnotu *Success*.

Property **InstanceOfType** typu *object* je při vytvoření položky v kontajneru instancí (tedy položky *ListViewItem* komponenty *ListView*) naplněna hodnotou *null*, neboť instance bude vytvořena až voláním jednoho z dostupných konstruktorů.

Pole typu *List<KeyValuePair<string, object>>* je seznam všech konstruktorů (pokud existují). Způsob, jakým je tento seznam naplněn je podrobně popsán v následující kapitole. V dále uvedeném příkladu bude představen segment kódu, který provede operace vymezené uvedenými podmínkami a nastaví hodnotu property **ElementType** na typ objektu.

Př.15 Sled operací: kompilace syntaktického stromu, emitování IL do paměti, načtení IL do instance třídy *Assembly* a získání typu objektu metodou *GetType*.

```
SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(sourceCode);
string nameSpace = syntaxTree.GetRoot().DescendantNodes().OfType<NamespaceDeclarat
ionSyntax>().FirstOrDefault().Name.ToString();

CSharpCompilation compilation = CSharpCompilation.Create(
   "dotCORNCompilation",
   new[] { syntaxTree },
   new[]
   {
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
      MetadataReference.CreateFromFile(typeof(System.Linq.Enumerable).Assembly.Loc
      ation
   },
      new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary, checkOverf
      low: true));
 using (var dllStream = new MemoryStream())
 {
  EmitResult emitResult = compilation.Emit(dllStream);
  if (emitResult.Success)  {
```

```
Assembly assembly = Assembly.Load(dllStream.ToArray());
string elementName = (element as Shape).Entity.Name;
Type type = assembly.GetType($"{nameSpace}.{elementName}");
ListViewGroup lvGroup = GetGroupForInstance(elementName);
ListViewItem lvItem = new ListViewItem()
{
 Name = (element as Shape).Entity.EntityType.ToString(),
 Group = lvGroup,
 Text = $"instance{GetInstancesCount(elementName)}:{elementName}",
 ImageIndex = 1,

   Tag = new InstantiatedElement { ElementType = type, InstanceOfType = nul, Co
   nstructorsList = walker.ConstructorsList }
 };
lvInstances.Items.Add(lvItem);
}
```

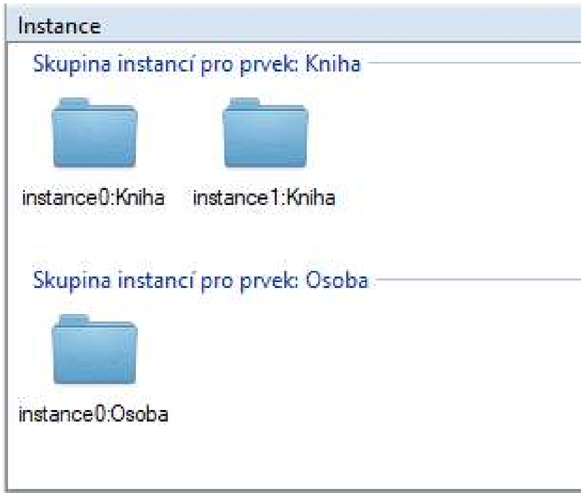
První řádek kódu je inherentní charakteristikou většiny případů využití platformy Roslyn a v různých obměnách se bude vyskytovat i v dalších příkladech uvedených v této práci. Voláním metody *ParseText* třídy *CSharpSyntaxTree* je sestaven syntaktický strom – základní jednotka pro analýzu kódu.

Aby bylo možno později vytvořit tzv. plně kvalifikované jméno objektu, jehož typ bude požadován, je potřeba zjistit název jmenného prostoru, ve kterém je třída deklarována. Toho lze velmi snadno dosáhnout zúžením kolekce potomků kořenového uzlu syntaktického stromu a to tak, že požadovaný typ uzlu bude specifikován jako *NamespaceDeclarationSyntax*. V případě, že podmínce bude vyhovovat více uzlů, přijme se první z nich. Zde se vychází z předpokladu, že ve zdrojovém kódu byla třída obsažena pouze v jediném jmenném prostoru.

Vytvoření kompilačního objektu včetně parametrů metody *Create* a modifikací kompilace prostřednictvím parametrů objektu *CSharpCompilationOptions* je vysvětleno v kapitole 4.3.3 týkající se Emit API.

V bloku vymezeném klíčovým slovem **using** pak dochází k požadovaným operacím. Kompilační objekt *compilation* je emitován do paměťového proudu *dllStream* třídy *MemoryStream* a odtud je následně načten metodou *Load* (resp. jedním z jejich sedmi přetížení *Load(byte[] rawAssembly)*), která vrací objekt třídy *Assembly*. V posledním kroku se voláním metody *assembly.GetType* s parametrem plně kvalifikovaného jména třídy získá typ objektu.

Následný kód již jen zajistí vytvoření položky *ListViewItem* a případně ji zařadí do odpovídající skupiny. Pokud skupina pro danou kategorii typů (instancí) ještě neexistuje, pak je vytvořena. Podstatná je inicializace property **Tag** po vytvoření instance voláním konstruktoru **new** *ListViewItem()*. Graficky se pak dynamicky vytvořený typ zobrazí způsobem uvedeným na následujícím obrázku.



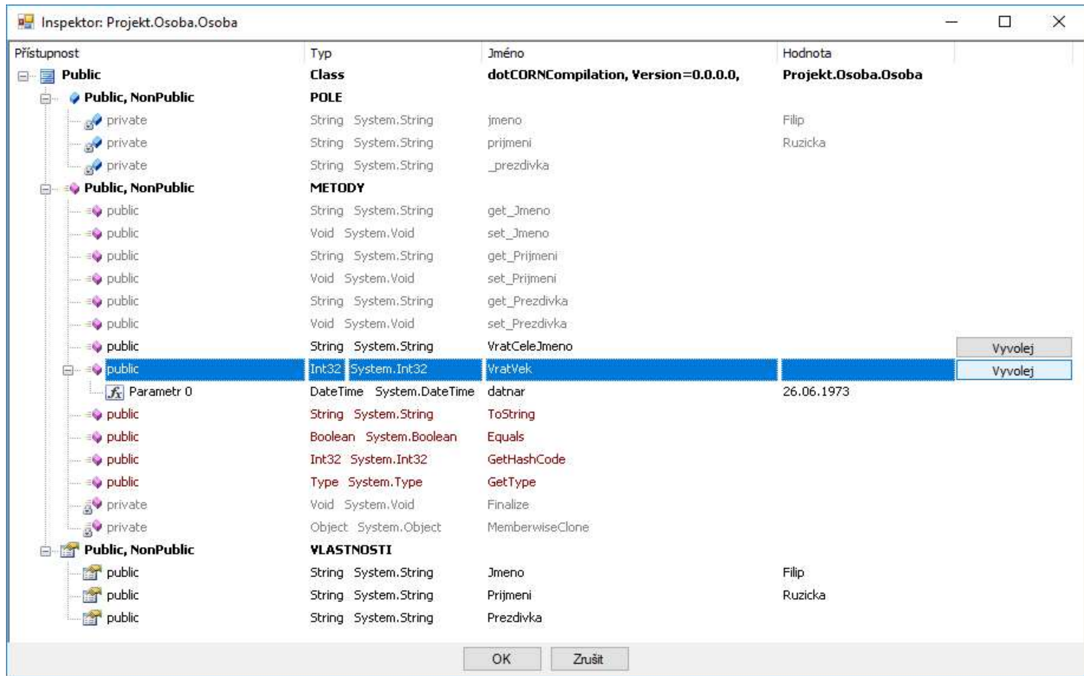
Obrázek 22- Zástupci dynamicky získaných typů a instancí tříd z diagramu UML
Zdroj: Vlastní tvorba (aplikace dotCORN)

Po dvojkliku myší na ikonu zastupující typ (instanci) se hodnoty atributů třídy *InstantiatedElement* předají do eponymních atributů (dvě property a pole stejného jména i datového typu) třídy *InstanceMemberEditor*. Vzhledem k tomu, že tato třída je formulářem, tj. je odvozená ze třídy *Form*, zobrazí se voláním metody *ShowDialog*. Následující vývoj závisí na hodnotě property **InstanceOfType**, tedy na skutečnosti, zdali se formulář pro daný typ (z uživatelského pohledu lze vyjádřit spíše jako pro danou složku v kontajneru instancí) otevírá poprvé či už byl otevřen dříve. Jde-li o první pokus o otevření formuláře, provede se test, jestli je kolekce *ConstructorsList* prázdná či nikoli. Pokud není prázdná, znamená to, že třída má ve svém těle alespoň jednu deklaraci parametrického konstruktoru, kterou je potřeba při vytváření instance zohlednit.

Př.16 Ověření, zda kolekce *ConstructorsList* obsahuje alespoň jeden parametrický konstruktor.

```
bool isParametric = ConstructorsList.Where(x => (x.Value as List<KeyValuePair<stri
ng, Type>>).Count() != 0).Count() != 0;
```

Za předpokladu, že bude mít proměnná **isParametric** hodnotu *true*, otevře se formulář uvedený na Obrázek 26. Poté, co uživatel označením pole před názvem vybere právě jeden konstruktor pro vytvoření instance třídy, a je-li to potřeba tak i korektně vyplní všechny parametry konstruktoru, kterým předchází grafický symbol „*fx*“, pak se otevře formulář, jež vytvoří instanci (přesněji řečeno pokusí se o vytvoření instance, neboť tato operace nemusí vždy skončit úspěšně). Současně se zobrazí výběr relevantních polí, vlastností a metod.



Obrázek 23- Inspektor instance třídy Osoba
Zdroj: Vlastní tvorba (aplikace dotCORN)

V otevřeném formuláři označeném jako „Inspektor“ je v záhlaví uvedeno plně kvalifikované jméno třídy (to je složeno z názvu projektu, názvu diagramu a názvu třídy, které jsou odděleny tečkou). Stromová struktura pak poskytuje náhled na vnitřní uspořádání dynamicky získaného typu. Větve stromu první úrovně představují kategorie, v následující úrovni pak kolekce privátních i veřejných polí, metod, vlastností, popř. událostí. Text je ve stromové struktuře barevně rozlišen a má následující význam. Uzly stromu, jejichž text má šedou barvu jsou pro editaci nedostupné, a to i za předpokladu, že jsou specifikovány jako veřejné (public). Nedostupná metoda nemá na řádku tlačítko s popiskem „Vyvolej“. V případě, že má metoda signaturu s parametry, není ani možné tento seznam parametrů zobrazit (uzel nemá grafický symbol rodičovského uzlu pro expandování či sbalení seznamu podřízených potomků). Je-li nedostupné pole či vlastnost, nelze do nich přiřazovat hodnoty ve sloupci „Hodnota“.

Nedostupná veřejná metoda může mít dva druhy barevného rozlišení

- šedou barvou jsou obvykle zvýrazněny metody, které systém automaticky generuje pro *gettery* a *settery* vlastností
- červeně jsou označeny nedostupné veřejné metody zděděné ze třídy *Object*

Jsou-li nedostupná pole svázána s property, jejíž hodnotu lze měnit, pak je možné sledovat i změnu hodnoty u příslušného pole. Změna hodnoty je však interní, nikoli uživatelsky dostupná. Při zadávání hodnot polí, vlastností či parametrů metod ve sloupci „Hodnota“ by měl být respektován datový typ zadávané hodnoty, který je svým plně kvalifikovaným jménem uveden

ve sloupci „Typ“. Při zadávání hodnoty nejsou editorem kladeny na vstupní řetězec znaků žádné restrikce. Vstup korektní hodnoty není vynucen maskou, regulárním výrazem či jiným prostředkem. Ověření konzistence zadané hodnoty s požadovaným typem je provedeno pouze v okamžiku, kdy je ukončena editace veřejného pole nebo vlastnosti. Konverze zadané hodnoty na daný typ probíhá také v případě volání parametrické metody kliknutím na tlačítko „Vyvolej“.

Př.17 Obsloužení události *OnLoad* při otevření formuláře „Inspektor“ třídy *InstanceMembersEditor*

```
private void EntityMembersEditor_Load(object sender, EventArgs e)
{
 try
 {
   Type type = (elementType as Type);
   this.Text = $"Inspektor: {type.FullName}";
   object[] constructorArgs = null;
   if (InstanceOfType == null)   {
      if (ConstructorsList.Count != 0) {
        bool isParametric = ConstructorsList
        .Where(x => (x.Value as List<KeyValuePair<string, Type>>)
        .Count() != 0).Count() != 0;
        if (isParametric)  {
          ConstructorArgsEditor argsEditor = new ConstructorArgsEditor();
          DialogResult argsDialog = argsEditor.ShowDialog();

        if (argsDialog == DialogResult.OK)  {
            constructorArgs = new object[argsEditor.ConstructorArgs.Count()];
            constructorArgs = argsEditor.ConstructorArgs;
         }
         else  {
            return;
          }
        }
      }
      InstanceOfType = type.InvokeMember(null, BindingFlags.Public | BindingFlags.
      NonPublic | BindingFlags.Instance | BindingFlags.CreateInstance, null, null,
       constructorArgs)
    }
    isLoading = true;
    InspectMembers(type);
    isLoading = false;
  }
  catch (Exception ex)   {
    OutputWindow.Instance.VirtualConsole.AppendText(ex.Message);
 }
}
```

Některé konstrukce z výše uvedeného zdrojového kódu již byly diskutovány v předcházejícím odstavci. V každém případě je zde podstatné větvení programu na základě hodnoty vlastnosti **InstanceOfType**. Pokud je rovna konstantě *null*, řízení programu přejde na scénář testování přítomnosti parametrických konstruktorů, vyvolá dialogové okno *argsEditor* pro uživatelské vyplnění hodnot parametrů a těmito hodnotami naplní pole *constructorArgs* s prvky typu *object*.

Instance se pak z dynamicky získaného typu **type** obdrží voláním metody *InvokeMember*, která je v kontextu tohoto formuláře poměrně frekventovaná.

Jestliže však property **InstanceOfType** obsahuje jinou hodnotu než *null*, přejde se k výpisu vybraných členů typu (polí, vlastností, metod a událostí) a má-li to smysl, pak i jejich hodnot, tj. volá se metoda *InspectMembers* s parametrem **type**.

Parametry předávané metodě *InvokeMember* jsou uvedeny v následující tabulce.

Parametr	Popis
`string name`	Řetězec obsahující jméno konstruktoru, metody, vlastnosti nebo pole, které má být předmětem volání.
`BindingFlags invokeAttr`	Bitová maska složená z jednoho nebo více atributů výčtového typu BindingFlags (složená operátorem logického součtu), která specifikuje, jak se provádí vyhledávání členů typu. Pokud je parametr vynechán, je implicitně nastaven na hodnotu BindingFlags.Public | BindingFlags.Instance | BindingFlags.Static
`Binder binder`	Objekt, který definuje sadu vlastností a umožňuje vazbu, což může zahrnovat výběr přetížené metody, vynucení typové konverze argumentů nebo vyvolání člena typu prostřednictvím reflexe.
`object target`	Objekt, ze kterého bude vyvolán specifikovaný člen.
`object[] args`	Pole obsahující argumenty pro předání konstruktoru, metodě, poli či vlastnosti, která má být vyvolána.

Tabulka 5- Tabulka parametrů metody InvokeMember
Zdroj: Vlastní tvorba

Pole, vlastnosti i metody jsou včetně svých metadat získány prostřednictvím reflexe[32] – voláním metod *GetFields*, *GetMethods* resp. *GetProperties* inspektovaného typu. Jejich zobrazení je založeno na procházení polí, které tyto metody vracejí a současném začleňování vytěžených informací do datových struktur vizualizační komponenty *TreeViewAdv*.

Realizací nosné myšlenky tohoto modulu praktické části práce je však testování metod, jejichž implementace je součástí elementu typu třída v navrženém modelu tříd. Zde je možné ověřit,

[32] Reflexí je označováno prozkoumávání existujících typů prostřednictvím metadat. Uskutečňuje se prostřednictvím sady typů ve jmenném prostoru `System.Reflection`. Reflexe představuje procházení a manipulování s objektovým modelem, který představuje nějakou aplikaci včetně všech jejích elementů kompilace a běhu.

jak metoda pracuje, aniž by bylo třeba její kód přenášet do jiných vývojových prostředí, kompilovat ho a spouštět v rámci konzolové či jiné aplikace.

Pokud metoda vrací jiný typ než **void** nebo jsou v jejím kódu použity metody pro standardní výstup, pak je možné činnost metody zachytit v okně „Virtuální konzole". Metoda se provede po stisknutí tlačítka „Vyvolej".

Na Obrázek 23 je např. modře podbarvena metoda *VratVek* (vrať věk) s parametrem datnar (datum narození) ze třídy *Osoba*. Zápis této metody v jazyce C# má následující tvar:

Př.18 Implementace metody *VratVek* třídy *Osoba*.

```
public int VratVek(DateTime datnar)
{
  var today = DateTime.Today;
  var a = (today.Year * 100 + today.Month) * 100 + today.Day;
  var b = (datnar.Year * 100 + datnar.Month) * 100 + datnar.Day;

  return (a - b) / 10000;
 }
```

Virtuální konzole

Filip Růžička - Diplomová práce Copyright © 2018 ČZU PEF - Katedra informačního inženýrství

Volání metody VratVek: 44

Obrázek 24- Výsledek volání metody VratVek ve Virtuální konzoli
Zdroj: Vlastní tvorba (aplikace dotCORN)

V následujícím příkladu bude předveden fragment kódu, který se provede po kliknutí na tlačítko „Vyvolej" u veřejné metody v inspektoru instancí.

Př.19 *Handler* tlačítka „Vyvolej" pro vykonání kódu metody v inspektoru instancí

```
private void methodInvoke_ButtonMouseDownHandler(TreeNodeAdvMouseEventArgs args)
{
  object[] obj = new object[args.Node.Children.Count()];
  int errCount = 0;
  foreach (TreeNodeAdv paramNode in args.Node.Children)
  {
    TreeViewRowNode rowParamNode = (TreeViewRowNode)(paramNode.Tag);
    if (!string.IsNullOrEmpty(rowParamNode.MemberValue))
    {
      try
      {
        if (!rowParamNode.MemberType.IsEnum)
       {
       obj[paramNode.Index] = Convert.ChangeType(rowParamNode.MemberValue, rowParam
       Node.MemberType);
    }
    else {
```

```
        obj[paramNode.Index] = Enum.Parse(rowParamNode.MemberType, rowParamNode.Memb
        erValue);
      }
    }
    catch (Exception ex)
    {
      OutputWindow.Instance.VirtualConsole.AppendLine(ex.Message);
      return;
    }
  }
  else {
   errCount++;
  }
}
if (errCount == 0)
{
        result = ElementType.InvokeMember(rowNode.MemberName, BindingFlags.InvokeMet
        hod | BindingFlags.Public | BindingFlags.Instance, null, InstanceOfType, obj
        );
       RefreshPrivateFields();
}
```

Uvedený kód nejprve prochází polem vstupních parametrů metody a konvertuje jejich typy. Implicitním typem je **string**, tj. řetězec znaků zadaný uživatelem. Vlastnost **MemberType** ještě algoritmus rozděluje na případy, kdy se jedná o datový typ výčet a typy ostatní. Výsledkem volání konverzních metod *Convert.ChangeType* resp. *Enum.Parse* je pole prvků typu *object*, které však již mají datový typ odpovídající typu očekávanému, tedy takovému, jež je uveden ve sloupci „Typ". Pokud konverze parametrů proběhne bez vyvolání výjimky a současně je proměnná **errcount** rovna nule, provede požadovaná metoda volání funkce *InvokeMember* a následně se aktualizují hodnoty privátních, tj. pro editaci nepřístupných polí.

4.3.2 Workspaces API

Na první pohled by se mohlo zdát, že API určené pro manipulaci s řešeními, projekty a dokumenty není reálné ani příliš smysluplné využívat mimo vývojové prostředí MS Visual Studio. Opak je však pravdou, neboť Workspaces API nemá vazbu na žádnou z jeho komponent, tudíž funkci hostitele (IDE) registrujícího události z okolí a zasílajícího zprávy do pracovního prostoru může plnit i jiná aplikace s načteným řešením. Pracovní prostory byly stručně charakterizovány jako kořenový uzel hierarchie C#, která se skládá z řešení (*Solutions*), podřízených projektů (*Projects*) a v nich obsažených dokumentů (*Documents*) (Varty, 2014). Podstatnou vlastností pracovních prostor, stejně jako u mnoha jiných objektů v Roslynu, je jejich neměnnost, což bylo již zmíněno v teoretické části práce v kapitole 3.3.2.

V praktické části práce se aplikace obrací na služby Workspaces API ve dvou modelových případech. V prvém případě se jedná o triviální načtení Solution (.sln) a zobrazení jeho

hierarchické struktury prostřednictvím komponenty *TreeViewAdv*, jež je uvedená v teoretické části práce jako jeden z klíčových vizualizačních a ovládacích prvků integrovaných v demonstrační aplikaci *dotCORN*. Pro zdůraznění těsnosti vazby mezi grafickým návrhem modelu tříd a aplikací analytických nástrojů z API Roslynu je zpracování Solution, které vzniklo v procesu generování zdrojového kódu z UML diagramu. Lze samozřejmě analyzovat libovolné řešení, které vytváří MS Visual Studio, a to včetně jeho poslední uvolněné verze (v době vzniku této práce VS2017). V druhém případě jde o vytvoření tzv. *AdhocWorkspace* pro alternativní přístup k sémantickému modelu, jehož nástroji se na formuláři *DiagramInstanceContainer.cs* získávají datové typy argumentů přetížených parametrických konstruktorů deklarovaných v objektové třídě.

Struktura řešení (Solution)

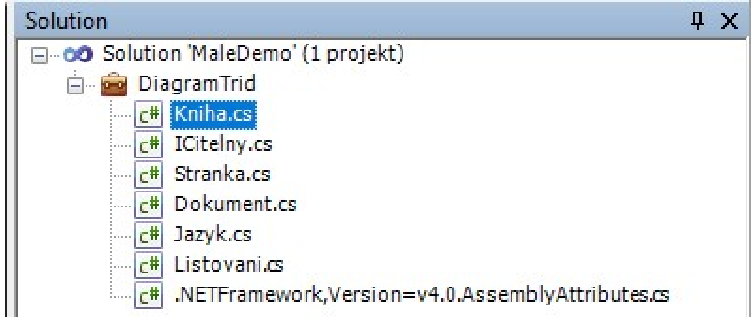
Soubor řešení (.sln) se načte do aplikace ze submenu dostupného klinutím na tlačítko opatřené ikonou žluté rozevřené složky a doplněné kontextovou nápovědou „Otevřít“ na nástrojové liště editoru. Zde je možno volit mezi otevřením dokumentu (souboru s příponou .cs) „Zdrojový kód jazyka C#“ a Solution „Visual Studio Solution“. Po otevření souboru řešení se jeho název, včetně cesty k místu uložení, zapíše do textového pole, kterému předchází tlačítko s popiskem „Solution“. Po načtení se zpřístupní nabídky „Odstranit“ a „Podrobnosti“. Volbou položky „Podrobnosti“ se zavolá metoda *ShowSolutionTree*, která je implementovaná ve třídě *AnalyzeSolutionStructurePreview*.

Př.20 Zobrazení struktury Solution s využitím *MSBuildWorkspace*

```
public void ShowSolutionTree(string solutionFile)
{
 MSBuildWorkspace workspace = null;
 Solution solution = null;
 Image nodeImage = null;
 try
 {
  workspace = MSBuildWorkspace.Create();
  solution = workspace.OpenSolutionAsync(solutionFile).Result;
  tvSolutionTree.BeginUpdate();
  treeModel.Nodes.Clear();
       nodeImage = new Bitmap(this.ilSolutionTree.Images[TreeNodeImageIndex.Solutio
       nTree_Solution]);
       TreeViewRowNode rootNode = new TreeViewRowNode
       {
         Icon = nodeImage,
       Name = $"Solution '{FileHelper.GetPrettyFileName(solutionFile)}' ({solution.
       Projects.Count()} projekt)",
       FullName = solutionFile
   };
   rootNode.Tag = rootNode;
```

```
    treeModel.Root.Nodes.Add(rootNode);
    foreach (Project project in solution.Projects)
    {
        nodeImage = new Bitmap(this.ilSolutionTree.Images[TreeNodeImageIndex.Solutio
        nTree_Project]);
          TreeViewRowNode projNode = new TreeViewRowNode
          { Icon = nodeImage,
            Name   = project.Name,
            FullName = project.FilePath
           };
          projNode.Tag = projNode;
          rootNode.Nodes.Add(projNode);
          foreach (Document document in project.Documents)
          {
          nodeImage = new Bitmap(this.ilSolutionTree.Images[TreeNodeImageIndex.Solut
          ionTree_Document]);
          TreeViewRowNode documentNode = new TreeViewRowNode
          { Icon = nodeImage,
            Name = document.Name,
            FullName = document.FilePath
          };
          documentNode.Tag = documentNode;
          projNode.Nodes.Add(documentNode);
      }
    }
    tvSolutionTree.EndUpdate();
    tvSolutionTree.ExpandAll();
  }
  catch (ReflectionTypeLoadException ex)   {
        OutputWindow.Instance.VirtualConsole.AppendLine("Nastala výjimka při zpracov
        ání solution: " + ex.Message);
         return;
  }
}
```

Ve výše uvedeném příkladu je použit pracovní prostor třídy *MSBuildWorkspace* ze jmenného prostoru `Microsoft.CodeAnalysis.MSBuild`. Instance této třídy pak poskytuje metodu pro asynchronní otevření řešení typu MSBuild, jejíž návratová hodnota je typu *Solution*. Proměnná *solution* typu *Solution* reprezentuje sadu projektů obsažených v řešení a k projektům jejich podřízené dokumenty se zdrojovým kódem. Nelze přejít bez povšimnutí fakt, že volání metody *OpenSolutionAsync* vykazuje zřetelnou časovou prodlevu (řádově sekundy). Pomocí dvou cyklů **foreach** lze pak celkem snadno zkonstruovat stromovou strukturu řešení, přesněji řečeno obsahu pracovního prostoru. Prvek (proměnná) *document* typu *Document* v kolekci *project.Documents* reprezentuje nejen zdrojový kód dokumentu, který je součástí projektu, ale také zdrojový text, parsovaný syntaktický strom a odpovídající sémantický model.



Obrázek 25- Struktura řešení s užitím třídy MSBuildWorkspace
Zdroj: Vlastní tvorba (aplikace dotCORN)

Uzly třetí úrovně stromového uspořádání *TreeViewAdv* (v dokovacím okně „Solution“) představují dokumenty projektu a mají v property **FullName** uloženu cestu k odpovídajícímu souboru zdrojového kódu. Dvojklikem myši na uzel dojde k otevření tohoto souboru a přenesení jeho obsahu do editoru. To umožňuje upravovat kód dokumentů např. před kompilací řešení.

Př.21 Obsluha události dvojkliku na uzlu stromu *TreeViewAdv*

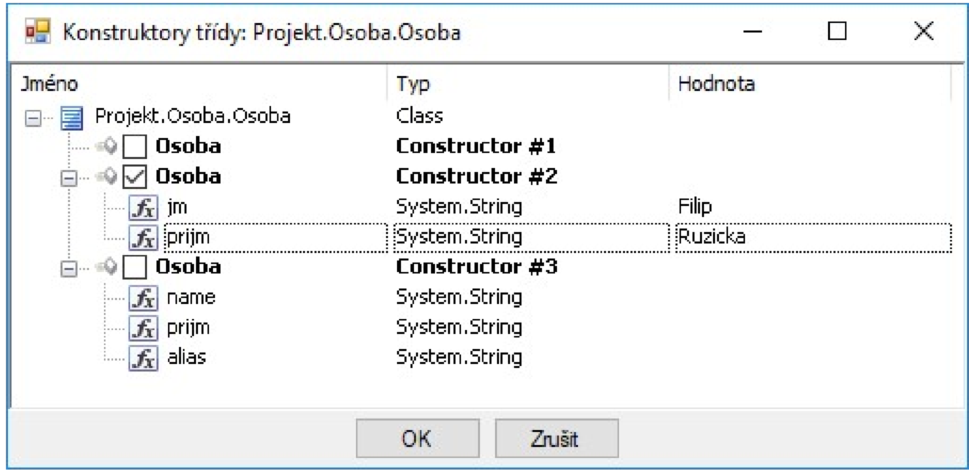
```
private void tvSolutionTree_NodeMouseDoubleClick(object sender, TreeNodeAdvMouseEv
entArgs e)
{
  if (e.Button == MouseButtons.Left && e.Node.Level == 3)  {
      string fakeFileName = string.Empty;
      AnalyzeCodeAnalysis.Instance.ScintillaEditor.Text = FileHelper.GetFileConten
      t(((TreeViewRowNode)(e.Node.Tag)).FullName, out fakeFileName);
   }
 }
```

AdhocWorkspace

Tento pracovní prostor se v porovnání s ostatními prostory ve Workspaces API značně liší. Je určen k rychlému vytvoření a následně ke snadnému manuálnímu přidávání řešení, projektů a dokumentů. Motivací pro jeho použití bylo otevřít přístup k sémantickému modelu, z něhož by bylo možné extrahovat datové typy parametrů v konstruktorech třídy (viz. kapitola 4.3.1) Pro tento případ existuje několik možných způsobů řešení, ze kterých se jako nejspolehlivější a nejrychlejší ukázalo získání sémantického modelu z kompilačního objektu, tj. z instance třídy *CSharpCompilation*. Navzdory tomu zde byla pro názornost ponechána i varianta s vytvořením *AdhocWorkspace*, a to především z důvodů naprosté výjimečnosti, kdy není nutným předpokladem pro analýzu v Roslynu konstrukce syntaktického stromu.

Před vytvořením instance elementu třída v diagramu tříd je nutné nejprve zjistit, zda je ve zdrojovém kódu deklarován konstruktor či nikoli. V případě, že třída neobsahuje žádný nebo neparametrický konstruktor, není třeba takovou situaci ošetřit. Avšak v situaci, že je v rámci

třídy deklarován parametrický konstruktor či více jeho přetížení, je nezbytné, aby uživatel jeden z dostupných konstruktorů zvolil a jím pak bude instanciována příslušná třída. Jde tedy nejen o nalezení deklarací konstruktorů a jejich parametrů (v gesci SyntaxTree API), ale také datových typů těchto parametrů. Popsanou situaci ilustruje následující obrázek.



Obrázek 26- Výběr konstruktoru a vyplnění jeho parametrů
Zdroj: Vlastní tvorba (aplikace dotCORN)

Formulace problému tedy zní: vyhledat v parsovaném syntaktickém stromu všechny deklarace konstruktorů a vytvořit objekt, který bude obsahovat seznam uspořádaných dvojic <jméno konstruktoru, parametry konstruktoru>. Tento objekt pak předat dále ke zpracování, tedy jeho vizualizaci ve formě stromu s uživatelsky editovatelnými parametry zvoleného konstruktoru. Takto definovaný problém je z hlediska použití prostředků z API .NET Compiler Platform velmi komplexní, avšak řešení je poměrně snadné a elegantní.

Př.22 Vytvoření seznamu konstruktorů pomocí *CSharpSyntaxWalker*

```
private class ConstructorWalker : CSharpSyntaxWalker
{
      public List<KeyValuePair<string, object>> ConstructorsList = new List<KeyVal
      ueP air<string, object>>();
    private readonly SemanticModel semanticModel;
    public ConstructorWalker(SemanticModel model)
    {
       semanticModel = model;
     }
      public override void VisitConstructorDeclaration(ConstructorDeclarationSynta
      x node)
      {
        ParameterListSyntax parameters = node.ParameterList;
        List<KeyValuePair<string, Type>> constructorArgs = new List<KeyValuePair<s
        tring, Type>>();
        foreach (ParameterSyntax parameter in node.ParameterList.Parameters)
        {
            ITypeSymbol typeParamSymbol = semanticModel.GetDeclaredSymbol(paramet
            er).Type as INamedTypeSymbol;
```

```
                Type type = Type.GetType($"{typeParamSymbol.ContainingNamespace.Name}
                .{typeParamSymbol.Name}");
                constructorArgs.Add(new KeyValuePair<string, Type>(parameter.Identifi
                er.Text, type));
            }
           ConstructorsList.Add(new KeyValuePair<string, object>(node.Identifier.Text
           , constructorArgs));
           base.VisitConstructorDeclaration(node);
          }
  }
```

CSharpSyntaxWalker je abstraktní třída zděděná ze třídy *CSharpSyntaxVisitor<TResult>* a určená pro situace, kdy je z nějakého důvodu požadován průchod všemi uzly syntaktického stromu. Obvykle to bývá za účelem sestavení množiny uzlů nějaké specifické vlastnosti. Nadřazená třída *CSharpSyntaxVisitor*[33] obsahuje veřejnou virtuální metodu *VisitConstructorDeclaration*, která je přepsána pro potřeby požadovaného řešení.

Z příkladu Př.22 je také zjevné, že tvar požadovaného objektu se seznamem konstruktorů a jejich parametrů je *List<KeyValuePair<string, object>>*, přičemž **object** ve struktuře *KeyValuePair* je kolekcí uspořádaných dvojic jména a typu parametru, který je analogicky implementován způsobem *List<KeyValuePair<string, Type>>*. Vlastní implementace původní virtuální metody *VisitConstructorDeclaration* provádí rekurzivní průchod syntaktickým stromem a hledá v něm deklarace konstruktorů. Pokud deklaraci najde, prochází seznamem parametrů konstruktoru (*node.ParameterList.Parameters*) a voláním metody *GetDeclaredSymbol* z instance sémantického modelu zjistí symbolický pojmenovaný typ parametru, tj. není to prozatím datový typ, který je specializací třídy *Type*. Třída *System.Type* představuje abstrakci deklarace daného typu a je vstupní branou k jeho metadatům v mechanismu reflexe. V tento okamžik je vhodné připomenout, že na úrovni sémantického modelu se stále pracuje s uzly syntaktického stromu, tudíž zde nelze použít operátor **is** ani metodu *GetType*. Pokud se však toto symbolické jméno typu doplní na jeho plně kvalifikovanou formu (tj. název jmenného prostoru.název symbolického typu), pak jej lze transformovat na typ metodou *GetType* třídy *Type*. Takto vytvořený pár <název parametru, typ parametru> se přidá do seznamu *constructorArgs*. Sémantický model nabízí v tomto ohledu ještě jiné způsoby, jak s využitím rozhraní *ITypeSymbol* zjistit typ analyzovaného objektu. Alternativní cestou je např. volání metody *GetTypeInfo*, jež vrací strukturu *TypeInfo* se dvěma podstatnými vlastnosti – **ConvertedType** a **Type**, nicméně uvedený způsob implementován nebyl. V okamžiku, kdy je seznam parametrů kompletně sestaven, je přiřazen do struktury *KeyValuePair* na pozici

[33] CSharpSyntaxVisitor obsahuje 204 veřejných virtuálních metod určených k procházení uzlů požadovaného typu, např. VisitClassDeclaration, VisitInvocationExpression, VisitForEachStatement atd.

hodnoty. Klíčem je v tomto případě název konstruktoru. Jelikož se jedná o konstruktory téže třídy, které se liší pouze počtem parametrů, je klíčová složka struktury vždy stejná.

Konstruktor odvozené třídy *ConstructorWalker* je parametrický a jeho argumentem je sémantický model. Ten je před voláním konstruktoru získán jedním ze dvou způsobů, a to na základě hodnoty proměnné *AccessSemanticModel* ze statické třídy *GlobalSettings*. Tato proměnná je výčtového typu *SemanticModelSource* a nabývá hodnoty CSharpCompilation nebo AdhocDocument. Implicitně je hodnota této proměnné nastavena na CSharpCompilation, avšak za běhu programu ji lze měnit prostřednictvím dialogu „Nastavení" podobně, jako je tomu u přidávání jmenných prostor v případě skriptování (viz kapitola 4.3.4)

Př.23 Použití zděděné třídy *ConstructorWalker* pro získání sezamu konstruktorů a jejich parametrů.

```
SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(sourceCode);
if (GlobalSettings.AccessSemanticModel == GlobalSettings.SemanticModelSource.CShar
pCompilation) {
  semanticModel = compilation.GetSemanticModel(syntaxTree);
}
else {
  Document doc = GetAdHocDocument(sourceCode);
  semanticModel = doc.GetSemanticModelAsync().Result;
}
ConstructorWalker walker = new ConstructorWalker(semanticModel);
walker.Visit(syntaxTree.GetRoot());
```

Sémantický model je dostupný v kompilačním objektu voláním metody *GetSemanticModel* s parametrem syntaxTree. Pokud však nastane situace, že je nutné se k sémantickému modelu dostat jinou cestou, lze použít již zmiňovaný *AdhocWorkspace*.

Př.24 Použití *AdhocWorkspace* pro přístup k sémantickému modelu.

```
private Document GetAdHocDocument(string text)
{
 try
 {
   AdhocWorkspace adhocWorskpace = new AdhocWorkspace();
       SolutionInfo solutionInfo = SolutionInfo.Create(SolutionId.CreateNewId(),Ver
       sionStamp.Default);
       adhocWorskpace.AddSolution(solutionInfo);
       ProjectInfo projectInfo = ProjectInfo.Create(ProjectId.CreateNewId(), Versio
       nStamp.Default, "AdHocProject", "AdHocProject", "C#");
    adhocWorskpace.AddProject(projectInfo);
    SourceText sourceText = SourceText.From(text);
    adhocWorskpace.AddDocument(projectInfo.Id, "AdHocDocument.cs", sourceText);
return adhocWorskpace.CurrentSolution.Projects.SingleOrDefault().Documents.FirstOr
Default();
}
catch (Exception ex) {
OutputWindow.Instance.VirtualConsole.AppendLine("Nastala výjimka :" + ex.Messag
e);
  return null; } }
```

Z uvedeného příkladu jsou zřejmé kroky, kterými se lze postupně dopracovat až k instanci třídy *Document*, ze která je možné volat asynchronní metodu *GetSemanticModelAsync* pro přístup k sémantickému modelu.

Nejprve se tedy vytvoří instance třídy *AdhocWorkspace*. Následně se vytvoří řešení, které je k instanci pracovního prostoru připojeno. K řešení se dále připojí projekt a k projektu dokument. Názvy „AdHocProject“ pro projekt a „AdHocDocument.cs“ pro dokument jsou pouze fiktivní a ve skutečnosti mohou být zcela libovolné, neboť takové řešení s projektem a dokumentem bude zkonstruováno pouze dočasně v paměti počítače.

Na tomto místě je vhodné uvést rozdíl mezi zdrojovým textem a zdrojovým kódem, což jsou v kontextu Roslynu dvě odlišné věci. Zdrojový text je abstraktní třída *SourceText* deklarovaná ve jmenném prostoru `Microsoft.CodeAnalysis.Text`. Tato třída obsahuje řadu vlastností (např. kódování textu, délku, kolekci řádků, algoritmus kontrolního součtu atd.) a metod (kopírování, nahrazování, porovnávání, načtení textu apod.) Důležitou metodou je přetížená metoda *From*, která umožňuje načíst zdrojový kód (z řetězce, souboru, streamu, pole) do struktur zdrojového textu. Zdrojovým kódem je pak obecně řetězec nebo proud znaků, které vyjadřují konstrukce nějakého programovacího jazyka. V literatuře jsou pojmy zdrojový text a zdrojový kód většinou považovány za synonyma. Také v této práci je s nimi, mimo kapitoly o analýze kódu, nakládáno poměrně volně a zpravidla je tím myšleno totéž.

Závěrem této kapitoly je třeba konstatovat, že ačkoli měla představit jen některé vybrané prostředky z Workspaces API, její obsah byl podstatně komplexnější. Je to důkazem skutečnosti, která se na první pohled může zdát banální, nicméně má poměrně obecný rozměr. Jednotlivé vrstvy Roslynu a jejich API nefungují odděleně, ale jejich použití se může prolínat i na několika málo řádcích zdrojového kódu. Vždy pochopitelně záleží na povaze řešeného problému a na cestě, kterou vývojář pro svůj postup zvolí.

4.3.3 **Emit API**

Po teoretické stránce bylo již Emit API stručně představeno v kapitole 3.3.2. Prostředky tohoto rozhraní umožňují finalizovat proces překladu emitováním kompilačního objektu, kterým se rozumí instance třídy *CSharpCompilation* do souboru na disku nebo do paměti počítače. Po ukončení této operace je možné testovat výsledný stav a v případě neúspěchu detailně diagnostikovat jeho příčiny.

V aplikaci *dotCORN* jsou funkce pro generování sestavení z větší části soustředěny pod událost kliknutí na tlačítku „Kompilovat“ umístěném na nástrojové liště nad editorem kódu. V závislosti na stavu aplikace se vykoná jedna z následujících akcí:

- Pokud v editoru není žádný kód a ani v textovém poli označeném jako „Solution“ v nástrojové liště není uvedena cesta k souboru se strukturou projektů a dokumentů (.sln), pak se neprovede žádná akce.
- Jestliže v editoru není žádný kód, ale v poli „Solution“ je uvedena cesta k souboru (.sln), pak aplikace uživateli oznámí, že je aktivní Solution a dotáže se, zdali má pokračovat v jeho kompilaci. Pokud uživatel odpoví kladně, dojde k překladu a emitování způsobem uvedeným dále. V případě negativní odpovědi na dotaz kompilace Solution se proces ukončí.
- V případě, že je v editoru zdrojový kód a současně je aktivní Solution, pak se postupuje dle stejného scénáře jako v předešlém bodě jen s tím rozdílem, že pokud uživatel zamítne zpracování Solution, pak proces nekončí, ale přejde se ke kompilaci zdrojového textu, který je v editoru.

Chce-li se uživatel soustředit pouze na kompilování kódu v editoru, není příliš vhodné, aby měl aktivní také Solution. To lze z analýzy odstranit volbou „Solution -> Odstranit“ v nástrojové liště, nikoli však vymazáním cesty v textovém poli (to má vlastnost *ReadOnly*, tudíž jeho obsah nelze přímo uživatelsky modifikovat).

V případě, že je zdrojem kódu pro kompilaci text v editoru, je možné před spuštěním překladu nastavit typ výsledného sestavení, kterých je v enumerátoru *OutputKind* definováno celkem šest:

ConsoleApplication
DynamicallyLinkedApplication
WindowsApplication
NetModule
WindowsRuntimeApplication
WindowsRuntimeMetadata

V aplikaci *dotCORN* lze však zvolit pouze z prvních tří, a to formou výběru ze seznamu vpravo od tlačítka „Kompilovat“. Parametr *OuputKind* je jedním z 25 parametrů trojnásobně přetíženého konstruktoru třídy *CSharpCompilationOptions*, jejíž instance je parametrem metody *Create* třídy *CSharpCompilation*. Nastavení atributů objektu *CSharpCompilationOptions* zásadním způsobem ovlivňuje výsledné sestavení a lze v něm zohlednit veškeré požadavky na kompilaci stejně jako je tomu v IDE Visual Studio.

Třída *CSharpCompilation* má ve světě Emit API podobně výsadní postavení jako syntaktický strom v SyntaxTree API, i když jeho význam je zcela stěžejní napříč všemi API Roslynu. Vždy prvním krokem k vytvoření již mnohokráte zmiňovaného kompilačního objektu je volání konstruktoru *Create*, který má následující parametry:

Parametr	Popis
`string assemblyName`	Název sestavení
`[IEnumerable<SyntaxTrees> syntaxTrees = null]`	Kolekce instancí třídy SyntaxTree. (nepovinný parametr)
`[IEnumerable<MetadataReference> references = null]`	Kolekce referencí na sestavení potřebná ke kompilaci. (nepovinný parametr)
`[CSharpCompilationOptions options = null]`	Nastavení předvoleb kompilace. (nepovinný parametr)

Tabulka 6- Parametry metody Create třídy CSharpCompilation
Zdroj: Vlastní tvorba

Tímto voláním se vytvoří kompilační objekt tzv. „od píky“[34], tj. alokuje se nový paměťový prostor, nastaví se proměnné dle vstupních parametrů, resp. jejich implicitních hodnot atd. Voláním metod na již vytvořené instanci této třídy (např. *AddSyntaxTrees*, *AddReferences*) bude kompilace probíhat inkrementálně (pouze promítání změn do již existujícího stavu).

Př.25 Překlad zdrojového kódu v editoru aplikace *dotCORN*

```
if (!string.IsNullOrEmpty(sciEditor.Text))
{
  string outFile = "output";
  SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(sciEditor.Text);
      bool containsMain = syntaxTree.GetRoot().DescendantNodes().OfType<MethodDecl
      arationSyntax>().Where(x => x.Identifier.Text.Equals("Main")).Count() != 0;
  tscbOutputKind.SelectedIndex = containsMain ? 0 : 2;
  OutputKind kind = (OutputKind)tscbOutputKind.SelectedIndex;
  CSharpCompilation compilation = CSharpCompilation.Create(
"dotCORNCompilation",
new[] { syntaxTree },
new[] { MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
        MetadataReference.CreateFromFile(typeof(System.Linq.Enumerable).Assembly.L
        ocation) },
new CSharpCompilationOptions(kind, checkOverflow: true));
if (MainForm.Instance.ActiveDiagram != null) {
  outFile = MainForm.Instance.ActiveDiagram.Project.Name.ToLower();
}
string extension = string.Empty;
switch (kind)
{
 case OutputKind.ConsoleApplication:
 case OutputKind.WindowsApplication:
   extension = ".exe";
   break;

 case OutputKind.DynamicallyLinkedLibrary:
   extension = ".dll";
   break;
}
string peFile = outFile + extension;
string pdbFile = outFile + ".pdb";

var emitResult = compilation.Emit(peFile, pdbFile);
```

[34] V terminologii Roslynu je to tzv. „from scratch“

Prvním, téměř vždy nezbytným, krokem je sestavení syntaktického stromu voláním metody *ParseText* třídy *CSharpSyntaxTree*. Dále pak aplikace testuje, zdali zdrojový kód neobsahuje metodu **Main** (booleovská proměnná *containsMain*). Vyhledávání této metody (ale i jakékoli jiné) v syntaktickém stromu spočívá v prohledávání všech podřízených uzlů uzlu kořenovému, která jsou typu *MethodDeclarationSyntax* a pokud text tohoto uzlu odpovídá požadovanému názvu metody (v tomto případě **Main**). Pokud tato metoda v kódu existuje, aplikace předpokládá, že uživatel bude chtít takový kód kompilovat do spustitelné podoby a automaticky zvolí typ emitovaného sestavení jako *ConsoleApplication* (.exe soubor)

Emit je rozšiřující metoda kompilačního objektu, která odešle bajtkód do specifikovaného proudu (souboru, paměťového streamu, řetězcového streamu apod.). Má následující parametry:

Parametr	Popis
`string outputPath`	Výstupní soubor
`[string pdbPath = null]`	Výstupní soubor PDB (Program DataBase)
`[string xmlDocumentationPath = null]`	Soubor pro generování XML dokumentace z komentářů v kódu.
`[string win32ResourcesPath = null]`	Specifikuje cestu k souboru, ze kterého budou čteny Win32 zdroje (.RES formát)
`[IEnumerable<ResourceDescription> manifestResources = null]`	Umožňuje připojení .resx souborů, např. řetězců, ikon atd.
`[CancellationToken = default]`	

Tabulka 7- Parametry metody Emit třídy CSharpCompilation
Zdroj: Vlastní tvorba

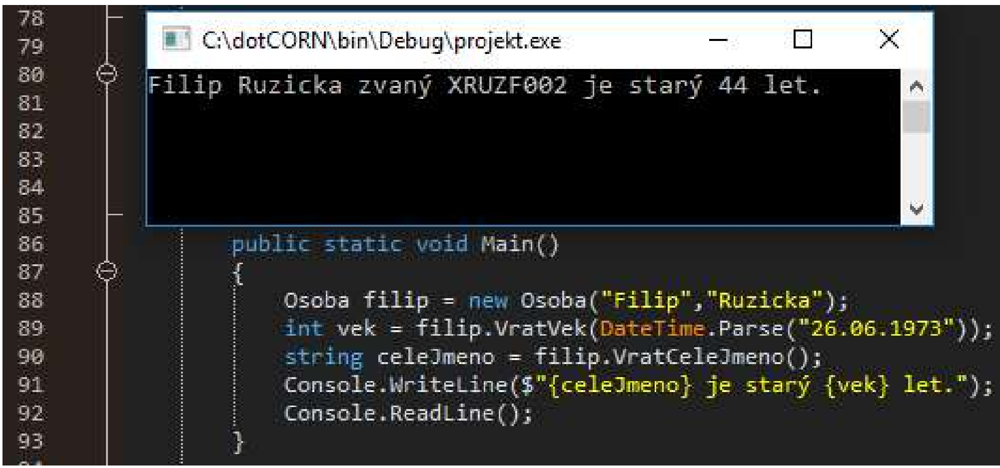
Parametr pdbPath je volitelný a uživatel by měl vždy rozvážit, zdali .pdb soubor skutečně potřebuje, neboť jeho zápis na disk může často znamenat poměrně značnou časovou a paměťovou náročnost.

Metoda *Emit* vrací v závislosti na výsledku své činnosti jednu ze dvou hodnot *Success* v případě úspěšného emitování IL do stremu nebo *Diagnostics* při neúspěchu. *Diagnostics* je pole typu *ImmutableArray<Diagnostic>*, ve kterém jsou zaznamenány veškeré chyby spojené nejen s emitováním sestavení, ale i chyby parsování, deklarací, kompilování atd.

Aplikace výsledek operace *Emit* ověřuje a pokud je pozitivní (*emitResult* je *Success*), pak je možné vytvořené sestavení přímo spustit. V opačném případě je do okna „Virtuální konzole“ vypsán obsah pole *Diagnostics*.

Př.26 Zpracování výsledku metody *Emit*.

```
if (emitResult.Success)
{
        DialogResult emitDialog = MessageBox.Show($"Byl vytvořen soubor {peFile}\n
        Přejete si ho spustit?", "Otázka", MessageBoxButtons.YesNo, MessageBoxIcon
        .Question, MessageBoxDefaultButton.Button2);
  if (emitDialog == DialogResult.Yes)
  {
    ProcessStartInfo startInfo = new ProcessStartInfo();
    startInfo.UseShellExecute = false;
    startInfo.FileName = peFile;
    startInfo.WindowStyle = ProcessWindowStyle.Normal;
    try
    {
      using (Process exeProcess = Process.Start(startInfo))
      {
        exeProcess.WaitForExit();
       }
     }
      catch (Exception ex)  {
        OutputWindow.Instance.VirtualConsole.AppendLine(ex.Message);
       }
    }
   }
   else {
     foreach (Diagnostic diagnostic in emitResult.Diagnostics)
     {
       OutputWindow.Instance.VirtualConsole.AppendLine(diagnostic.ToString());
      }
    }
}
```



Obrázek 27- Emitované sestavení spuštěné jako konzolová aplikace
Zdroj: Vlastní tvorba (aplikace dotCORN)

Pro vytvoření nového procesu a připojení programu, kterým bude proces řízen je použita třída *ProcessStartInfo* ze jmenného prostoru `System.Diagnostics`.

Závěrem této kapitoly, a s ohledem na kapitolu následující, je třeba konstatovat, že použití prostředků Emit API je opravdu přínosné v případech, kdy uživatel požaduje sestavení ve formě

diskového souboru. Při emitování IL do paměti a jeho následném spouštění je na zvážení, zdali není pro tento účel výhodnější užít prostředky Scripting API.

4.3.4 **Scripting API**

Toto rozhraní umožňuje aplikacím běžícím v prostředí .NET Frameworku vytvářet instance skriptovacího „stroje" a spouštět fragmenty kódu s využitím hostování objektů tohoto API. Prvotní realizací myšlenky skriptování v rámci Visual Studia a následně prostřednictvím kompilátoru jako služby byl v roce 2008 představen Andersem Hejlsbergem nástroj REPL (viz. kapitola 3.3.2). Od té doby projekt postupovat tak, aby zahrnoval také výkonnou analýzu kódu a API kompilátoru. Scripting API se však stalo plnohodnotnou součástí Roslynu až v roce 2011. V této práci je použito skriptování ve verzi 2.6.0, které vyžaduje .NET Framework 4.6 a vyšší. (Uhlenhuth, 2016)

Co se týče vlastního použití, to je v aplikaci *dotCORN* z uživatelského hlediska poměrně snadné. Skript se zapíše do editoru nebo se načte ze souboru a spustí se kliknutím na tlačítko „Spustit skript" v nástrojové liště. Pokud skript poskytuje výstup ve formě metod *Console.Write*, resp. *Console.WriteLine*, pak je tento výstup zobrazen v okně „Virtuální konzole". Na následujícím obrázku je ukázka skriptu, který vytvoří krátkou posloupnost přirozených čísel a z této řady pak využitím klauzule **Where** s připojenou podmínkou vytvoří kolekci pouze sudých prvků posloupnosti. Tato nově vytvořená posloupnost je pak procházena v cyklu příkazem **foreach** a každý prvek této posloupnosti je vypsán na standardní výstup.

Jak je na první pohled patrné, zdrojový kód není součástí žádné metody, nepředchází mu import žádných jmenných prostor, není použita statická metoda **void Main()** ani žádné jiné jazykové formy, kterými by bylo nutné za normálních okolností kód opatřit, aby byl spustitelný.

Pokud by se ve skriptu vyskytla syntaktická chyba, kompilátor (v tomto případě je přesnější spíše použít termín „scripting engine") ji zachytí a standardním způsobem zpracuje, tj. vyvolá výjimku třídy *CompilationErrorException*. Většina výjimek je však v této aplikaci po zachycení směrována do okna „Virtuální konzole". Pro ilustraci, pokud by např. na řádku č.4 chyběl za koncovou složenou závorkou středník, překladač by při pokusu o spuštění takového skriptu ohlásil chybu: „Chyba kompilace: (4,49): error CS1002: ; expected"

Diagram tříd | Analýza kódu

Refaktoring · Spustit skript · Kompil

```
1   // Roslyn Scripting API - Příklad 1 - kolekce
2
3   Console.Write("Vybere jen sude prvky");
4   List<int> seznam = new List<int> { 1,2,3,4,5,6 };
5   List<int> sude = new List<int>();
6   sude = seznam.Where(x => (x % 2) == 0).ToList();
7   foreach (int cislo in sude)
8   {
9       Console.Write(cislo.ToString());
10  }
11
```

Virtuální konzole

```
Filip Růžička - Diplomová práce Copyright © 2018 ČZU PEF
- Katedra informačního inženýrství

> Vybere jen sude prvky
> 2
> 4
> 6
```

Obrázek 28- Scripting API – kolekce
Zdroj: Vlastní tvorba (aplikace dotCORN)

Př.27 Spuštění skriptu asynchronní metodou *RunAsync*

```
private async void ExecuteScriptAsync(string script)
{
  ScriptOptions scriptOptions = ScriptOptions.Default;
  var mscorlib = typeof(System.Object).Assembly;
  var systemCore = typeof(System.Linq.Enumerable).Assembly;
  scriptOptions = scriptOptions.AddReferences(mscorlib, systemCore);
      scriptOptions = scriptOptions.AddImports(GlobalSettings.ScriptOptionsImportL
      ist.ToArray());
  try
  {
    var state = await CSharpScript.RunAsync(script, scriptOptions);
   }
   catch (CompilationErrorException ex)  {
      OutputWindow.Instance.VirtualConsole.AppendLine("Chyba kompilace: " + ex.Mes
      sage);
    }
   }
```

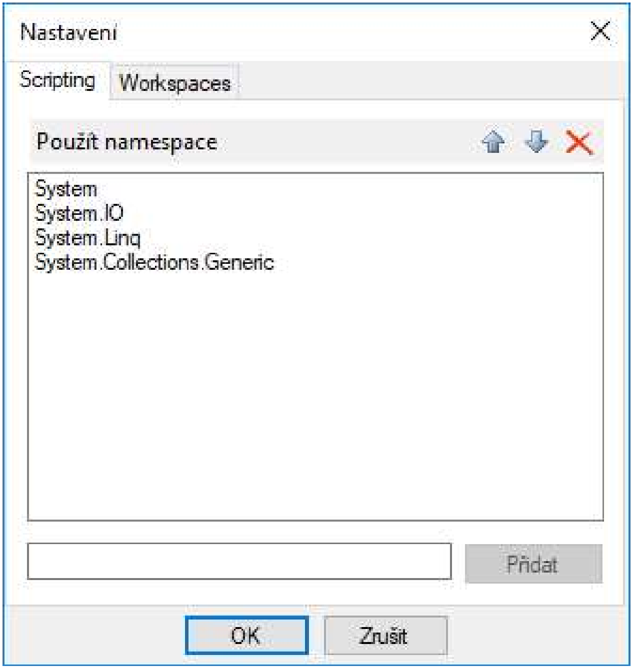
Na tomto místě by bylo vhodné udělat vsuvku a ve stručnosti objasnit princip asynchronního vykonávání metod, neboť je to technika, která bez nadsázky tvoří páteř platformy Roslyn a do jazyka C# byla začleněna od verze 5.0. Prostředky pro práci s vlákny jsou soustředěny do jmenného prostoru `System.Threading.Tasks`. Obecně řečeno, asynchronní techniky jsou přínosné v místech aplikace, které jsou potenciálně kritické z hlediska výpočetní složitosti a tím pádem mohou v průběhu svého vykonávání blokovat provádění jiných akcí, které uživatel nebo

jiný modul systému požaduje. Typickými příklady může být např. složitý matematický výpočet (renderování scény) či čekání na odezvu webového serveru. Asynchronie je obzvláště vhodná pro aplikace, které přistupují k vláknu UI, protože všechny aktivity související s uživatelským rozhraním obvykle sdílí jediné vlákno. Pokud je jakýkoli proces blokován v synchronní aplikaci, jsou blokovány všechny ostatní. Aplikace přestane reagovat a uživatel často předjímá, že selhala, i když tomu tak ve skutečnosti není.

Základem asynchronního programování jsou klíčová slova **async** a **await**. Klíčovým slovem **async** je deklarována metoda, v jejímž těle se bude alespoň jednou vyskytovat klíčové slovo **await**. Tato metoda pak bude implementována jako orchestrace běhu asynchronního workflow sestaveného z volání jiných existujících asynchronních funkcí**.** Tím se změní způsob jejího překladu do stavového automatu, který bude zajišťovat spouštění částí kódu za a mezi jednotlivými **await** jako tzv. callback jednotlivých asynchronních volání**.** Metoda označená klíčovým slovem **async** musí vracet typ *Task*, *Task<T>* nebo případně *void*. V metodě označené klíčovým slovem **async** příkaz **return** určuje okamžik dokončení asynchronní úlohy a v případě návratového typu *Task<T>* se za něj uvádí pouze výraz typu *T* – výstup z asynchronní metody.

Klíčové slovo **await** uvedené před asynchronním voláním zajistí, aby se další běh kódu umístěného za tímto operátorem naplánoval a spustil až po dokončení volané operace. V důsledku toho jsou v kódu za **await** již k dispozici výsledky prováděného asynchronního volání (tj. v případě metody vracející *Task<T>* je k dispozici hodnota typu *T*). Klíčové slovo **await** lze uvést pouze před výraz typu *Task* nebo *Task<T>*. (MSDN, 2013)

Před spuštěním skriptu je třeba si uvědomit, jaké problematiky se skript dotýká (kolekce, manipulaci se soubory, LINQ, XML, práce s textem apod.) a dle toho nastavit potřebné reference a jmenné prostory. Implicitně jsou v proměnné *ScriptOptionsImportList*, která je umístěná ve statické třídě *GlobalSettings* nastaveny prostory `System`, `System.IO`, `System.Linq` a `System.Collections.Generic`. V případě, že je nutné přidat další *namespace*, pak lze tuto operaci provést otevřením formuláře z hlavní nabídky aplikace, a to výběrem „Nastavení -> Volby…“ na záložce „Scripting“



Obrázek 29- Globální nastavení parametrů dotCORN
Zdroj: Vlastní tvorba (aplikace dotCORN)

Diagram tříd | Analýza kódu

Refaktoring | Spustit skript | Kompilovat

```
1   // Příklad 2 - čtení souboru po řádcích
2
3   int counter = 0;
4   string line;
5
6   StreamReader file = new StreamReader(@"d:\demo.txt");
7   while((line = file.ReadLine()) != null)
8   {
9       Console.WriteLine(line);
10      counter++;
11  }
12  file.Close();
13  Console.WriteLine("Soubor má {0} řádek.", counter);
```

Virtuální konzole

```
Filip Růžička - Diplomová práce Copyright ©  2018 ČZU PEF
- Katedra informačního inženýrství

> Demonstrační textový soubor pro Scripting API
> ----------------------------------------------
> 1. Dobrý den
> 2. Hello
> 3. Guten Tag
> 4. Buenos Días
> 5. Bonjour
> Soubor má 7 řádek.
```

Obrázek 30- Čtení ze souboru pomocí Scripting API
Zdroj: Vlastní tvorba (aplikace dotCORN)

Scripting API není pochopitelně jen o jedné či dvou metodách, které zde byly zmíněny. Tento jmenný prostor je plný veřejných tříd, výčtů a rozhraní, které je možné využít celou řadou různých způsobů. Na výše uvedeném obrázku je např. zobrazen velmi jednoduchý příklad, jak pomocí skriptu otevřít textový soubor a vypsat jeho obsah po řádcích na konzoli.

Scripting API stojí svým zaměřením poněkud stranou vytyčené linie analýzy kódu z diagramů tříd, nicméně je též součástí .NET Compiler Platform a jeho použití je velmi flexibilní, efektní a velice účinně posouvá aplikaci ke stanovenému cíli, kterým je jednoduchá didaktická pomůcka, neboť je to právě jednoduchost použití skriptů, jež vybízí ke hrám a experimentům, které mohou v budoucnu přerůst v seriózní vývoj software.

5 Výsledky a diskuse

Pro posouzení, do jaké míry je praktická část této práce přínosná, a především svou úrovní způsobilá být kvalifikační prací určenou k obhajobě lze zřejmě aplikovat dva obvyklé typy metrik – měkké (subjektivní) a tvrdé (objektivní). Je nutné předeslat, že tyto míry jakosti jsou stanoveny zcela individuálně a nemají nic společného s exaktním hodnocením kvality softwarových produktů. Nezohledňují se charakteristiky aplikace ani sběr a způsob vyhodnocení měřitelných veličin předkládaného software, který vznikl v rámci praktické části této práce. Také není přihlíženo k žádným normám (ISO/IEC, SQuaRE), které jsou v oblasti hodnocení jakosti zavedené. Přesto mohou být měkké metriky v některých oblastech hodnocení praktických i teoretických výsledků tvůrčí činnosti zcela obvyklé a bez újmy na kvalitě posudku je k nim přihlíženo jako k doplňkovým hodnotícím kritériím. Lze se s nimi běžně setkat v segmentu projektového řízení, vícekriteriální analýzy variant, analýzy časových řad apod. Měkkou metrikou by mohlo být u této práce např. stanovení, jak aplikace působí z hlediska UX/UI, zda byla volba komponent pro grafické uživatelské rozhraní učiněna racionálně a vedena aktuálními trendy reflektujícími moderní aplikační design, tj. použití dokovacích oken, nástrojových lišt, kontextových nabídek, využití intuitivních ikon jako náhrada popisků tlačítek atd. Za objektivní (tvrdou) metriku může být v tomto případě považováno splnění či nesplnění všech stanovených cílů a dodržení metodických pokynů, navzdory skutečnostem, že cesta k finálnímu stavu zjevně nebyla vždy snadná a přímočará. Některá „velká“ témata nejsou v rešeršní ani praktické části této práce vůbec naznačena, přestože byla s vedoucím práce Ing. Jiřím Brožkem, Ph.D. diskutována. Jedním z takových témat je např. realizace jednoduchého debuggeru s plným využitím funkcí platformy Roslyn a editoru ScintillaNET. V každém případě však nebylo smyslem práce okopírovat integrované vývojové prostředí MS Visual Studio, ale především odkrýt možnosti, které přináší .NET Compiler Platform.

5.1 Realizace diagramu tříd

První fáze řešení praktické části diplomové práce započala v létě 2017 hledáním vhodné komponenty pro oblast grafického návrhu diagramu tříd. V té době ještě nebyly jasně vymezeny podmínky, zdali bude práce realizována s podporou technologie WPF[35] či jen s využitím WinForms ve Fameworku .NET. Úvahy o vývoji vlastní komponenty pro návrh diagramu byly

[35] WPF – Windows Presentation Foundation

z důvodů značné časové a implementační náročnosti zamítnuty již v samém počátku při formulování cílů a rozsahu práce.

Prvním přiblížením k dané problematice byla volba pokročilé open-source multiplatformní vizualizační knihovny **GraphX for .NET**. Popis knihovny na webových stránkách *CodePlex Archive* sliboval podporu řady grafových a dislokačních algoritmů, ale především implementaci pro jazyky C# a Visual Basic .NET s plnou podporou technologií WPF, WinForms, METRO a UWP[36]. Velmi záhy se však ukázalo, že celý projekt je postaven výhradně na technologii WPF, tudíž přizpůsobování zdrojových kódů tak, aby funkcionalita vyhovovala potřebám diplomové práce by bylo extrémně obtížné. Poslední stabilní verze knihovny v2.3.6 byla vydána v září 2016 pod hlavičkou ruské softwarové společnosti Panthernet. Byl proveden pokus o začlenění některých ovládacích prvků z knihovny GraphX do prostředí WinForms pomocí techniky hostování a s užitím komponenty *ElementHost* z palety nástrojů *WPF Interoperability* Visual Studia. Tento experiment se sice zdařil, nicméně perspektiva využití této knihovny byla z výše uvedených důvodů velmi mlhavá a v konečném rozhodování byla zamítnuta.

Další testovanou otevřenou sadou prostředků pro realizaci diagramů v prostředí WinForms Frameworku .NET byla knihovna **Crainiate Open Diagram**, taktéž dostupná na webu *CodePlex Archive*. Poslední stabilní verzí je verze v4.1 z června 2010. Zde, s aplikací komponenty *Diagram* založené na návrhovém vzoru Model-View-Controller a s využitím dalších tříd ve jmenných prostorech `Crainiate.Diagramming` a `Crainiate.Diagramming.Forms` bylo celkem smysluplné začít budovat grafický návrhář, tj. uživatelsky přívětivé vytváření tříd s atributy a operacemi, rozhraní, výčtových typů a především jejich spojení vazbami typů asociace, generalizace atd. Ukládání a načítání elementů diagramu bylo řešeno serializací a deserializací do/ze souboru ve formátu XML. Metody pro serializaci a deserializaci však ignorovaly konektory (propojení prvků v dokumentu diagramu) a tzv. markery (značky na konektorech – např. prázdný diamant pro agregaci, trojúhelník pro generalizaci apod.). Tato chyba nebyla odstraněna ani ve verzi v5.01beta a autor komponenty 3.2.2012 oznámil ukončení práce na stabilizaci programu i dalšího rozvoje celého projektu. Práce na implementaci vlastního serializeru konektorů a markerů se ukázaly jako příliš náročné a v konečném důsledku značně neefektivní. Po tomto nezdaru byla i tato vývojová větev prohlášena za slepou.

Třetím testovaným balíkem open-source řešení pro návrh diagramu tříd pro WinForms byla knihovna **NClass**. Ačkoli autor ukončil vývoj i podporu tohoto nástroje v červnu 2011 ve verzi v2.04 a poslední příspěvek v diskuzním fóru je datován k únoru 2017, jedná se velmi komplexní

[36] UWP – Universal Windows Platform

soubor prostředků pro uživatelsky nenáročné a intuitivní vytváření UML diagramů tříd s plnou podporou jazyků C# a Java. NClass umožňuje vytvářet třídy, rozhraní, delegáty, výčty, struktury a komentáře, přičemž definice atributů (polí, vlastností) a operací (metod, konstruktorů, destruktorů, event) je podepřena striktní syntaktickou a sémantickou kontrolou. Komponenta implementuje engineering jak dopředný (generování zdrojového kódu z diagramu) tak i zpětný (import specifických elementů jazyka z existujících sestavení a jejich převod do grafických prvků a vazeb v diagramu tříd UML). Také s přihlédnutím k precizní logické struktuře projektu, transparentnosti a vynikající čitelnosti zdrojového kódu byl tento superset komponent, tříd a grafických rozhraní zvolen jako základní stavební kámen pro diagram tříd v této diplomové práci.

5.1.1 Katalog změn v komponentě NClass

Ačkoli se komponenta NClass jevila jako téměř ideální pro okamžité nasazení, tj. bez nutnosti modifikace jejího zdrojového kódu, některé úpravy bylo třeba přeci jen udělat, a to především s ohledem na vlastní představu o jejím chování a vzhledu v rámci demonstrační aplikace *dotCORN*. Úprav bylo celkem 15, nicméně jen některé z nich mají zcela zásadní význam. Katalog změn uvádí následující tabulka.

Namespace	Dokument	Metoda/Třída
CodeGenerator	SolutionGenerator	CheckDestination
CodeGenerator	Dialog	btnGenerate_Click
ClassDiagram	Diagram	GetContextMenu
ClassDiagram	Diagram	AddClass
ClassDiagram	Diagram	AddStructure
ClassDiagram	Diagram	AddInterface
ClassDiagram	Diagram	AddEnum
ClassDiagram	Diagram	AddDelegate
ClassDiagram	Diagram	AddComment
ClassDiagram	DiagramElement	DiagramElement
ClassDiagram\Dialogs	ListDialog	ItemSelected
ClassDiagram\Dialogs	ListDialog	ClearInput
ClassDiagram\Shapes	CompositeTypeShape	ActiveMember
ClassDiagram\Shapes	Shape	GetContextMenuItems
ClassDiagram\Shapes	TypeShape	CaptionRegion

Tabulka 8- Změny provedené v komponentě NClass
Zdroj: vlastní tvorba

5.1.2 Vytváření instancí z diagramu tříd

Jedním z mála nedořešených problémů, který podstatným způsobem limituje aplikaci v jejím potenciálním rutinním a efektivním provozu je vytváření instance třídy, která má v diagramu vazby na jiné elementy, popř. obsahuje atributy (nebo parametry metod a jejich

návratové hodnoty) výčtových typů. Soubor se zdrojovým kódem, který vzniká v první fázi individuálního instanciování třídy z atributů a operací elementu třída v diagramu musí pro korektní vytvoření instance obsahovat deklarace všech relevantních prvků k dané třídě (tříd, rozhraní, enumeračních typů atd.). Pokud by tomu tak nebylo, nedošlo by k vytvoření kompilačního objektu ze třídy *CSharpCompilation* a z něj následně v paměti uložené assembly voláním metody *Emit.*

Jinak řečeno, dosud není implementován algoritmus, který by buď zajistil sloučení zdrojových textů všech elementů, které jsou v relaci s instanciovanou třídou nebo na pozadí vygeneroval syntaktické stromy těchto elementů a jimi doplnil proces kompilace (voláním metody *AddSyntaxTrees* v instanci třídy *CSharpCompilation*).

5.1.3 Generování zdrojového kódu

V kapitole 4.2.3 je popsána jedna z důležitých vlastností aplikace *dotCORN*, kterou je generování zdrojového kódu z modelu tříd. Tato vlastnost je charakteristická pro CASE nástroje, které ji realizují implementací algoritmu forward engineeringu. Transformace elementů diagramu na odpovídající zdrojový kód má však jistá úskalí a každý nástroj řeší tuto problematiku individuálně, ačkoli zápis v UML danou vlastnost či vazbu jasně specifikuje. Pro srovnání kvality kódu generovaného aplikací dotCORN, resp. komponentou NClass, která tvoří jádro designeru modelu tříd byl proveden test, jak tentýž jednoduchý modelový případ zpracují aplikace Enterprise Architect verze 12.0 a Sybase PowerDesigner verze 11.1.

Nechť příklad modeluje třídu Osoba s atributy jméno, příjmení, přezdívka a třídu Auto s atributy značka, barva, SPZ. Vztah mezi těmito dvěma třídami je modelován jako agregace – jedna osoba může vlastnit žádné nebo více aut. Existence osoby a automobilu jsou na sobě nezávislé. Auto není neoddělitelnou součástí celku Osoba a s jejím zánikem také nezaniká. Proto je vazba modelována jako agregace a nikoli kompozice (viz. kapitola 3.4.2)

Při převodu do zdrojového kódu libovolného objektově orientovaného jazyka se očekává, že uvedená vazba bude realizována jako kolekce prvků třídy Auto ve třídě Osoba.

Komponenta NClass nemapuje asociační vazby typu agregace či kompozice na odpovídající zdrojový kód. Pojmenování vazby, rolí a definování multiplicit je čistě formální bez vlivu na výsledný tvar kódu.

Nástroje PowerDesigner a Enterprise Architect se pokoušejí vazbu konvertovat na kód v cílovém jazyce pouze tehdy, je-li správně definována multiplicita rolí. PowerDesigner příliš mnoho prostoru pro uživatelské nastavení způsobu generování kódu neposkytuje. Vazbu převede na typ pole, jehož velikost lze modifikovat v detailu vazby v položce „Array size“.

Výsledný kód pro modelový příklad je pak **public Auto[] Association1**, kde Association1 je název vazby.

Enterprise Architect (dále jen EA) umožňuje volbou „Collection Classes" na záložce „Detail" formuláře vlastností třídy Auto specifikovat, jakým způsobem bude vazba v daném jazyce implementována. Textovou položku označenou titulkem „Default Collection Classes" lze pro jazyk C# vyplnit např. takto: List<#TYPE#>. Při generování kódu volbou „Package -> Code Engineering -> Generate Source Code for Package" pak EA substituuje formální parametr #TYPE# názvem třídy a výsledný kód bude mít následující tvar: **public List<Auto> auto**; Kolekce nemusí být vždy nutně typu *List*, ale mohou se uplatnit i kolekce S*tack, Queue, CArray, CMap* apod.

Závěrem této krátké kapitoly je konstatování, že ze tří hodnocených nástrojů CASE poskytuje Enterprise Architect poměrně racionální a univerzální řešení, jak zdrojovým kódem vyjádřit graficky modelovanou vazbu mezi entitami v UML diagramu tříd. Modifikace komponenty NClass způsobem, který by při generování kódu reflektoval multiplicitu rolí asociačních vazeb by výrazně zvýšila její užitnou hodnotu a posunula by také aplikaci *dotCORN* do kategorie vyspělejších řešení z oblasti počítačové podpory návrhu software.

5.2 Použití komponenty TreeViewAdv

V případě syntaktického stromu jsou důvody pro použití *TreeViewAdv* zcela opodstatněné. Strom, který umožňuje rozdělit data v uzlech do sloupců, a tím de facto simuluje funkci gridu je pro potřeby zobrazení kombinujícího hierarchické uspořádání dat s řadou doplňujících informací zcela ideální. *TreeViewAdv* je však použit i pro inspektor instancí, kde situace již tak jednoznačná není. Informace vytěžené z instance třídy, tj. seznamy polí, vlastností, metod a událostí mají na první pohled charakter dat vhodných pro uspořádání v tabulce. Komponenta *DataGridView*, která je běžně dostupná v .NET Frameworku je na rozdíl od jiných (*TreeView*) poměrně propracovaná a umožňuje celkem elegantně vkládat do sloupců tlačítka, ikony a jiné editační prvky, což může být v případě *TreeViewAdv* problematické. Značným handicapem je zde i ukončená podpora ze strany autora Andreje Gliznetsova a velmi řídké zdroje informací o výsledcích řešení stejného nebo podobného problému od jiných programátorů. Zcela selhávají osvědčené weby jako např. StackOverflow.com, SourceForge.net, CodeProject.com apod.

Inspektor instancí, stejně jako editor přetížených konstruktorů třídy, je nakonec přeci jen postaven na komponentě *TreeViewAdv*. Důvodem byla úvaha, že v případě metod (konstruktorů) s parametry bude lepší vytvořit odpovídající počet listů uzlu, které budou

obsahovat editační prvek pro zadání hodnot parametrů. Stejná situace by se v případě tabulkového zobrazení musela řešit nejprve vyvoláním dialogu s příslušným počtem editačních prvků a teprve pak by došlo k přenosu hodnot do volání metody.

5.3 Testování aplikace

Přestože by testování mělo být nedílnou součástí každého vývojového cyklu nového softwarového produktu a některé techniky jsou přímo založené na existenci ověřovacích procedur předcházející vlastnímu naprogramování funkčních jednotek (přístup programování řízeného testy[37]) nebyla aplikace v tomto směru nijak testována. Nebyly napsány žádné jednotkové automatické testy pro ověření správnosti na úrovni tříd, resp. volání třídních metod, událostí, konstruktorů či destruktorů. S ohledem na příspěvky některých uživatelů v diskuzních fórech, kteří upozorňovali např. na problémy s uvolňováním neúmyslně alokované paměti při spouštění událostí v komponentě *TreeViewAdv*, byly alespoň některé kritické části aplikace (sestavení syntaktického stromu a jeho zobrazení, inspektor instancí tříd atd.) prověřeny nástrojem .NET Memory Profiler[38].

Na rozdíl od tohoto, spíše vývojářského, pohledu na problematiku testování aplikace bylo provedeno poměrně velké množství uživatelských testů – mnohonásobně opakovaná tvorba diagramu tříd, editace elementů diagramu, generování zdrojového kódu z diagramu, přizpůsobení a změna rozložení dokovacích oken aplikačního uživatelského rozhraní, serializace a deserializace diagramu, ukládání a načítání modifikovaného vygenerovaného zdrojového text a mnoho dalšího.

---

[37] TDD – Test-Driven Development.

[38] Diagnostický nástroj švédské společnosti SciTech Software AB

6 Závěr

Doba a technologické změny v oblasti IT se svými evolučními a involučními amplitudami mnohdy přináší nejedno překvapení. Tak by se dal vnímat i krok společnosti Microsoft, která licenčně i fyzicky zpřístupnila kompletní platformu pro analýzu kódu a kompilaci v ekosystému .NET. Široké vývojářské obci se tím dostává do rukou sada velmi mocných nástrojů pro tvorbu uživatelských aplikací ze zcela nové oblasti či programování rozšiřujících modulů pro vývojové prostředí Visual Studio za účelem zvýšení kvality a přehlednosti zdrojového kódu, dodržování definovaných firemních politik, ověřování validity generovaných sestavení a mnoho dalšího. Myšlenka dynamického programování a manipulace s objekty za běhu programu v prostředí .NET však není nová. Microsoft zde navazuje a podstatně rozšiřuje oblasti reflexe, generování dynamických metod, sestavení a modelu CodeDOM.

Předkládaná diplomová práce ve své praktické části demonstruje formou řady spíše kratších funkčních bloků použití základních prostředků .NET Compiler Platform (Roslyn) definovaných v množinách aplikačních rozhraních nezávislých na komponentách Microsoft Visual Studia. Tyto funkční bloky nestojí osamoceně, ale jsou součástí vrstvy ležící pod grafickým uživatelským rozhraním aplikace a harmonicky rezonují s konceptem celé práce, která sice poněkud simplifikovaně, ale v celku přesně ukazuje cestu vývoje software od grafického návrhu ve standardu jazyka UML, resp. jeho diagramu tříd, přes generování aplikačního skeletonu ve zdrojovém kódu jazyka C#, až po jeho finální kompilaci do podoby spustitelné konzolové aplikace či dynamicky linkované knihovny. Z analýzy kódu jsou akcentovány partie týkající se sestavení, prohledávání a přepisování syntaktického stromu a dále pak oblast tvorby a použití kompilačních objektů.

Z grafických reprezentací tříd v diagramu lze za stanovených podmínek a při dodržení definované posloupnosti operací určit jejich typ v prostředí .NET Frameworku za běhu programu. S využitím mechanismů reflexe a dostupných metadat se již dají vytvářet instance takto získaného typu. Tyto objekty je pak možné velmi detailně nejen zobrazit, ale i interaktivně prověřit po stránce naplnění datových polí, vlastností a volání veřejných metod. Tento přístup, který se částečně inspiroval existující aplikací ve světě programovacího jazyka Java (Barnes, a další, 2016) se tak jeví jako vhodný pro rychlé prototypování a zejména validaci programových jednotek bez nutnosti spouštět Visual Studio, popř. zasazovat kód do kontextu složitějšího celku.

Aplikace, která vznikla v rámci této práce zřejmě ještě nemůže aspirovat na to, aby se stala didaktickou pomůckou, a to navzdory poměrně ambiciózně stanoveným cílům a již

zpracovanému obsahu. V každém případě je v ní podchycena mohutnost prostředků kompilační platformy Roslyn, jejichž využití je nepochybně možné dále rozšiřovat. Práce je záměrně koncipována jako polytématická. Nesnaží se jen striktně naplnit cíl, který je determinován jejím názvem, ale celou problematiku uchopuje v širších souvislostech, což jí dává zajímavou přidanou hodnotu. Čtenář tak v úvodních partiích nedostává informace pouze o Roslynu, ale seznamuje se s taxonomií překladačů a jejich charakteristickými vlastnosti, fázích klasického kompilačního řetězce, ale také o struktuře a elementech diagramu tříd v jazyce UML. Některých met však bohužel nebylo dosaženo, nicméně cesta je vyznačená a nic nebrání možnosti jít dál. První krok byl již učiněn.

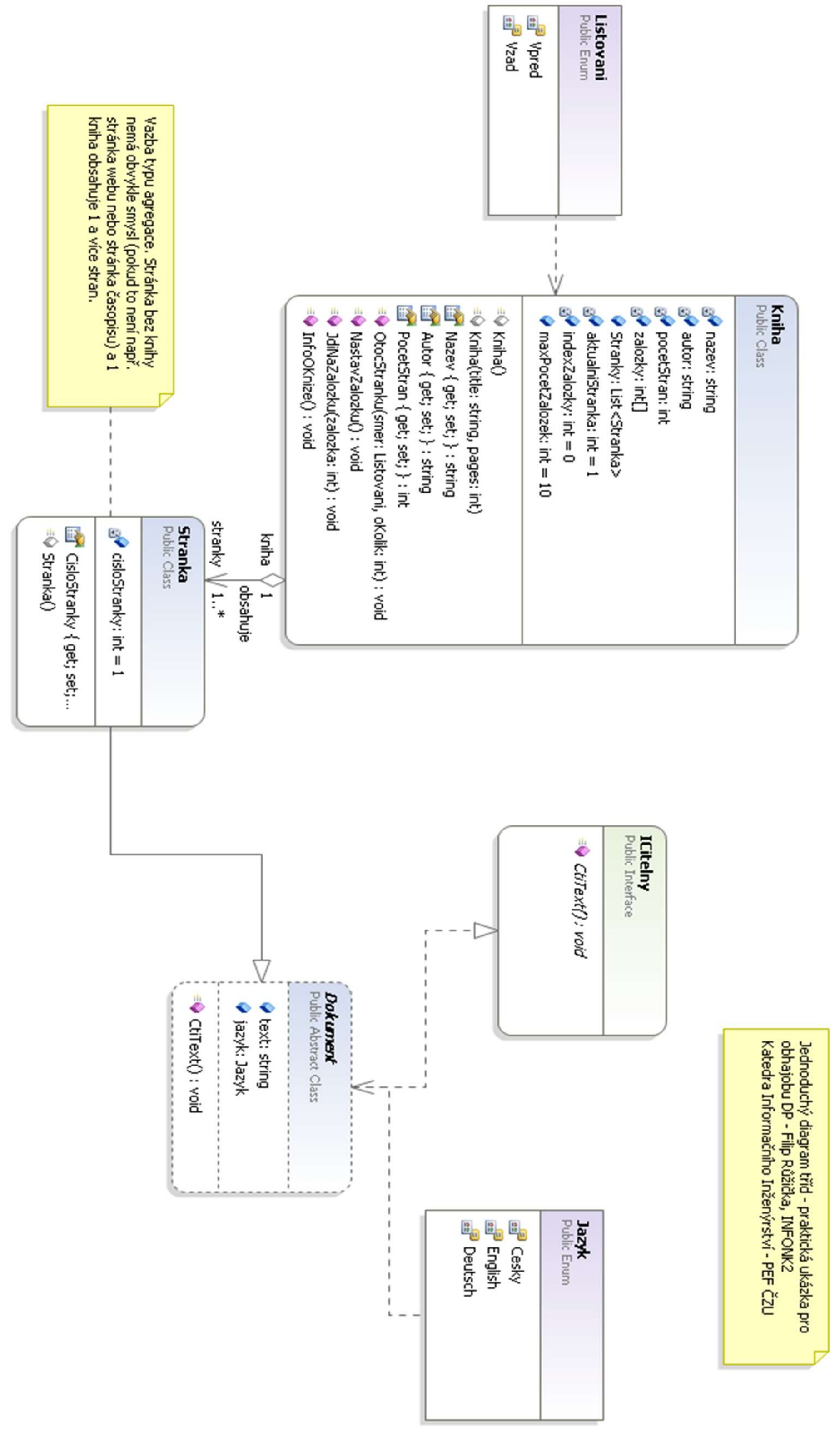
7 Seznam použitých zdrojů

Aho, V. Alfred, et al. 1986. *Compilers: Principles, Techniques and Tools.* s.l. : Pearson Education, 1986. ISBN: 0-201-10088-6.
Appel, Andrew W. a Ginsburg, Maia. 2004. *Modern Compiler Implementation in C.* Cambridge : Cambridge University Press, 2004. ISBN: 9780521607650.
Arlow, Jim a Neustadt, Ila. 2007. *UML 2 a unifikovaný proces vývoje aplikací.* Brno : Computer Press a.s., 2007. ISBN: 978-80-251-1503-3.
Barnes, David J. a Kolling, Michael. 2016. *Object First in Java: Practical Introduction Using BlueJ.* 6. místo neznámé : Pearson Education Limited, 2016. ISBN: 978-1-292-15904-1.
Benešovský, Miroslav a Richta, Karel. 2002. *UML, alea iacta est!* Brno : autor neznámý, 2002. Tutoriál UML. Konference DATAKON - Hotel Saton.
Bock, Jason. 2016. *.NET Development Using the Compiler API.* New York City : Apress, 2016. ISBN: 978-1-4842-2111-2.
Fogel, Karl. 2005. *Tvorba open source software.* Praha : CZ.NIC, 2005. ISBN: 978-80-904248-5-2.
Griffiths, Ian and Adams, Matthew. 2003. *.NET Windows Forms in a Nutshell.* s.l. : O'Reilly & Associates, Inc., 2003. ISBN: 0-596-00338-2.
Harrison, Nick. 2017. *Code Generation with Roslyn.* Lexington : Apress, 2017. ISBN: 978-1-4842-2211-9.
Healy, Patrick and Nikolov, Nikola S. 2006. *Graph Drawing.* Limerick : Springer Science & Business Media, 2006. ISBN: 978-3-540-31667-1.
Hummel, Joe a Neward, Ted. 2014. The Working Programmer : Rise of Roslyn. *Microsoft Developer Network Magazine Blog.* [Online] 1. 11 2014. https://msdn.microsoft.com/en-us/magazine/dn818501.aspx.
Kanisová, Hana a Muller, Miroslav. 2007. *UML srozumitelně.* 2. Brno : Computer Press a.s., 2007. ISBN: 80-251-1083-4.
Melichar, Bořivoj, a další. 1999. *Konstrukce překladačů I,II.* Praha : ČVUT, 1999. ISBN: 80-01-02028-2.
MSDN, Dokumentace. 2013. Asynchronní programování pomocí modifikátoru Async a operátoru Await. *Microsoft Developer Network.* [Online] Microsoft, Říjen 2013. https://msdn.microsoft.com/cs-cz/library/hh191443(v=vs.120).aspx.
Sharp, John. 2010. *Visual C# 2010 krok za krokem.* [překl.] Lukáš Krejčí. Brno : Computer Press a.s., 2010. ISBN: 978-80-251-3147-3.
Sole, Alessandro Del. 2016. *Roslyn Succinctly.* Morrisville : Syncfusion Inc., 2016. ISBN: 9781542827102.
Uhlenhuth, Kasey. 2016. .NET Compiler Platform ("Roslyn") Overview. *Roslyn Overview.* [Online] 6. Červen 2016. https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview.
Varty, Josh. 2014. Learn Roslyn Now. *Shotgun Debugging.* [Online] 6. Červenec 2014. https://joshvarty.com/learn-roslyn-now/.
Vasani, Manish. 2017. *Roslyn Cookbook.* Birmingham : Packt Publishing Ltd., 2017. ISBN: 978-1-78728-683-2.
Vrana, Ivan. 2008. *Projektování informačních systémů s UML.* Praha : PEF ČZU, 2008. ISBN: 978-80-213-1817-5.

8 Přílohy

Příloha A - Jednoduchý diagram tříd vytvořený v demonstrační aplikaci *dotCORN*
Příloha B - Struktura dynamicky získaného typu třídy Kniha

Příloha A



Příloha B