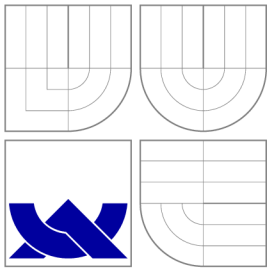
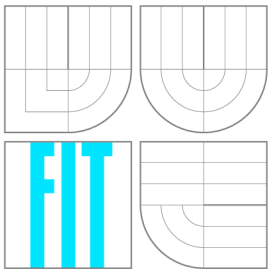


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATICKÉ GENEROVÁNÍ UML DIAGRAMU TŘÍD

AUTOMATIC GENERATION OF UML CLASS DIAGRAM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN BRÁZDIL

VEDOUCÍ PRÁCE
SUPERVISOR

doc. RNDr. JITKA KRESLÍKOVÁ, CSc.

BRNO 2015

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem a implementací aplikace pro automatické generování UML diagramu tříd. Aplikace je koncipována jako webová služba, což umožňuje vzdálený přístup, ale především neustálou aktuálnost vygenerovaného diagramu tříd. Vstupem služby je již přeložená libovolná aplikace psaná pro platformu C# .NET nebo Java. V práci je čtenář obeznámen se základy reverzního inženýrství pro zmíněné platformy a strukturou UML diagramu tříd. Následně jsou tyto znalosti aplikovány v návrhu a implementaci. Hlavním cílem práce je usnadnění a urychlení činnosti členů softwarových vývojových týmů.

Abstract

This master's thesis describes the analysis, design and implementation of an application for automatic generation of UML class diagram. Application is designed as a web service, which provides remote access, especially permanent actuality of generated class diagram. Input of the service is a compiled application written for C# .NET or Java platform. The reader is acquainted with basics of reverse engineering of mentioned platforms and with structure of UML class diagram. Then are these knowledge applied in design and implementation of the service. The main goal is to facilitate and accelerate the activities of software development team members.

Klíčová slova

reverzní inženýrství, generování, UML, diagram tříd, webová služba, Java, C#, .NET, reflexe, JVM, CLR, IL Assembler

Keywords

reverse engineering, generate, UML, class diagram, web service, Java, C#, .NET, reflection, JVM, CLR, IL Assembler

Citace

Martin Brázdil: Automatické generování UML diagramu tříd, diplomová práce, Brno, FIT VUT v Brně, 2015

Automatické generování UML diagramu tříd

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval zcela samostatně pod vedením paní doc. RNDr. Jitky Kreslíkové, CSc. a pana Ing. Jana Vernera ze společnosti Siemens, s. r. o., oddělení Corporate Technology Brno. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Brázdil
27. května 2015

Poděkování

Tímto bych chtěl poděkovat své vedoucí diplomové práce, doc. RNDr. Jitce Kreslíkové, CSc., a externím konzultantům ze společnosti Siemens, s. r. o., oddělení Corporate Technology Brno, Ing. Janu Vernerovi a Ing. Zdeňku Jurkovi, za trpělivost a systematické vedení při vývoji aplikace a tvorbě technické zprávy. Také bych rád poděkoval své rodině a přátelům za projevenou podporu a trpělivost v celém průběhu práce.

Obsah

Kapitola 1

Úvod

V dnešní době agilních přístupů k vývoji softwarových produktů, se může stát, že tým, jakožto celek, ztratí přehled nad aktuálním postupem vývoje z pohledu architektury aplikace. A vzhledem k jednomu z bodů agilního manifestu – „Fungující software před vyčerpávající dokumentací“ [1] – není na dokumentaci produktu ve fázi vývoje kladen přílišný důraz, protože by mohla zpomalovat vývoj samotný. Projektový vedoucí však musí mít přehled a prezentovat dosažené výsledky svému zadavateli. Z výše uvedeného plyne motivace této práce, a sice vytvořit nástroj, který dopomůže k optimalizaci procesu udržení konzistentních informací o aktuální architektuře.

Situace, kdy softwarový inženýr má k dispozici pouze přeložený program a potřebuje znát jeho strukturu, je poměrně obtížně řešitelná. Navíc pokud je potřeba mít strukturu vyvíjené aplikace neustále aktuální, musí jej inženýr manuálně obnovovat. Řešení uvedených problému poskytuje tato aplikace.

Struktura programu je poměrně široký a obecný pojem, a proto jej má práce zužuje pouze na diagram tříd v objektově orientovaném návrhu aplikací, který se řadí mezi diagramy rodiny unifikovaného modelovacího jazyku. Aplikace podporuje dvě nejpoužívanější platformy C# .NET a Java. Mé řešení automatické obnovy diagramu představuje přístup k aplikaci skrze internetové prostředí za použití webové služby. Pak lze jednoduchým odkazem (dotazem) na službu získat obrázek s požadovaným diagram a společně s pravidelně automaticky spouštěnou analýzou se bude udržovat i jeho stálá aktuálnost.

Tuto technickou zprávu tvoří čtyři stěžejní části. Nejprve je nutné si analyzovat a ujasnit požadavky ze strany zákazníka, o čemž pojednává kapitola 2. Jsou zde představeny potřebné technologie pro korektní návrh výsledné aplikace včetně analýzy již dostupných řešení na trhu. Poté následuje kapitola 3 specifikující návrh aplikace, respektive služby, pro automatické generování UML diagramu tříd. Mezi poslední kapitoly se řadí část o implementaci 4 a testování 5 jak jednotlivých knihoven, tak i výsledné aplikace.

Dále popisovaná diplomová práce navazuje na semestrální projekt vypracovaný v zimním semestru 2014/2015 v plném rozsahu kapitola věnovaná analýze požadavků 2. Z uvedeného projektu jsou použity i závěry konceptuálního návrhu architektury aplikace. V celém průběhu tvorby obou prací se využívalo poskytnutých konzultací u externího zadavatele, společnosti Siemens, s. r. o., oddělení Corporate Technology Brno, která z větší části vytvářela seznam požadavků na výslednou podobu aplikace.

Kapitola 2

Analýza požadavků

V druhé kapitole jsou popsány výchozí znalosti potřebné ke korektnímu návrhu a implementaci požadované funkcionality. Nejprve je vhodné uvést co a k čemu vlastně je reverzní inženýrství z pohledu softwarového vývoje (podkapitola 2.1). Další podkapitola 2.2 se již dostává k jádru problému, a sice jazyku UML, respektive k diagramu tříd, který bude výstupem mé aplikace.

Podkapitoly 2.3 a 2.4 uvádějí informace o programovacích jazycích, nebo spíše platformách, v nichž napsané programy jsou vstupem mé aplikace. Jedná se o jazyky Java a C# s použitým rozhraním .NET Framework.

Poslední podkapitola 2.5 analyzuje a porovnává dostupná řešení na trhu, tedy programy, které disponují funkcionalitou pro softwarové reverzní inženýrství (v kontextu mé práce s možností generovat diagram tříd v jazyku UML).

2.1 Reverzní inženýrství

V situaci, kdy analytik má k dispozici pouze výsledný program a je pověřen úkolem získání jeho „know-how“ (myšleno strukturu či zdrojové kódy), je jeho pozice obtížná. Avšak existují metodiky, které může využít. Tyto metodiky a nástroje jsou obecně nazývány *reverzní inženýrství*. Výborně jej definoval pan Elliot Chikovsky: Reverzní inženýrství je proces analýzy nějakého systému, jehož cílem je odhalení komponent, provázanosti a popsání daných skutečností odlišnou formou. [2] Jednoduše řečeno jde o proces jdoucí směrem z nízko úrovňové reprezentace k abstraktnějšímu popisu.

Tento proces se obecně skládá z různých postupů, které je možno kombinovat:

- studování dostupné dokumentace a případných zdrojových kódů,
- analýza běžícího systému,
- interakce s autory,
- využívání různých analyzátorů nad statickými daty systému (přeložené soubory, databáze, ...),
- a jiné.

Opak reverzního inženýrství popisuje *dopředné inženýrství*, které je obecně známým termínem pro postupné konkretizování abstraktního systému. [3]

2.2 UML

UML, neboli *Unifikovaný modelovací jazyk* (angl. „Unified Modeling Language“), se pokládá za jeden z nejpoužívanějších a nejperspektivnějších nástrojů pro modelování systémů. Dokonce lze tvrdit, že se jedná o nejužitečnější nástroj na poli softwarového inženýrství. Nejedná se však tak úplně o nástroj jako spíše o grafický jazyk použitelný v několika stupních vývoje s různými úrovněmi detailů popisu. Obecně lze použití UML rozdělit do třech kategorií [4]:

a) *náčrt* (angl. „sketch“) – Abstraktní použití principů UML při dennodenní komunikaci členů vývojových týmů. UML zde slouží jako nástroj pro rychlý transfer jednoduchých myšlenek a proto není nutné striktně dodržovat úplnost či validitu modelů.

b) *modrotisk* (angl. „blueprint“) – Modely při daném použití se naopak vyznačují jistou mírou úplnosti a správnosti popisu navrhovaného systému, přičemž se však ustupuje z vysoké detailnosti a je ponechána určitá volnost při následné implementaci. Tento přístup je v současné době, kdy se používají agilní vývojové metodiky a metodiky unifikovaného procesu (z angl. „Unified Process“), nejpoužívanější.

c) *programovací jazyk* – Maximální detailnost, úplnost a validita jsou vlastnosti, kterými oplývá zmíněný přístup. UML se používá pro vygenerování zdrojových kódů v požadovaném programovacím jazyce, který je přeložitelný a spustitelný.

UML vznikalo v průběhu 90. let minulého století za přispění pánů Booche, Rumbaugha a Jacobsona. Následně bylo vytvořeno konsorcium i s významnými softwarovými společnostmi této doby, jako je Mircosoft, Oracle a další. A v roce 1997 spatřilo světlo světa UML v1.0, jež konsorcium poskytlo OMG skupině[1], která jej v nynější době spravuje. Následující výčet mapuje nejvýznamnější změny v jednotlivých verzích UML standardu [4]:

- v1.1 (1997) – podpora implementačních tříd (vztah mezi rozhraním a implementující třídou), rozlišení mezi rolí asociační a rolí spolupráce,
- v1.3 (1999) – nahrazení vyznačení dědičnosti z pouhé závislosti se stereotypem «`extends`» na dnes již běžný vztah generalizace,
- v1.4 (2001) – přidána viditelnost atributu třídy v balíku (angl. „package visibility“) symbolem ∼,
- v2.0 (2005) – nový OCL (angl. „Object Constraint Language“), přidány diagramy komunikace, přehledu interakcí a časování,
- v2.3 (2010) – přidáno klíčové slovo `final`, vyjasnění termínu asociace a asociačních tříd,
- v2.4.1 (2011) – aktuální standard jazyka UML, přidán datový typ `Real` pro reálná čísla.

UML dále nabízí možnost dopředného i zpětného inženýrství. Kdy při potřebě generování zdrojových kódů v požadovaném programovacím jazyce z již vytvořených diagramů se

[1] OMG, neboli angl. „Object Management Group“, je mezinárodní standardizační komise spravující několik standardů z oblasti softwarového průmyslu.

použije inženýrství dopředné. Pokud však je nutné získat diagramy z již existujících zdrojových, ba dokonce z přeložených, kódů, lze použít metodu zpětného inženýrství. Nástroje podporující oba přístupy se obecně nazývají „round-trip“[2].

Nástroje podporující modelování za použití UML s možností využití pokročilých funkcí, ať už se jedná o integraci dokumentace, round-trip inženýrství apod., existuje na trhu mnoho a obecně se označují jako *CASE nástroje* (angl. „Computer Aided Software Engineering“), neboli vývoj software s využitím počítačové podpory. Tyto nástroje analyzuji v podkapitole 2.5.

2.2.1 Základní komponenty

Jazyk UML se sestává z pouhých tří základních stavebních bloků [5]:

1. *Předměty* – představují konkrétní elementy modelů (např. třídy, rozhraní, komponenta, případ užití . . .),
2. *vztahy* – jsou vazby mezi předměty (blíže popsány v podkapitole 2.2.2),
3. a *diagramy* – seskupují modely a vztahy mezi nimi, čímž tvoří ucelené pohledy na UML modely.

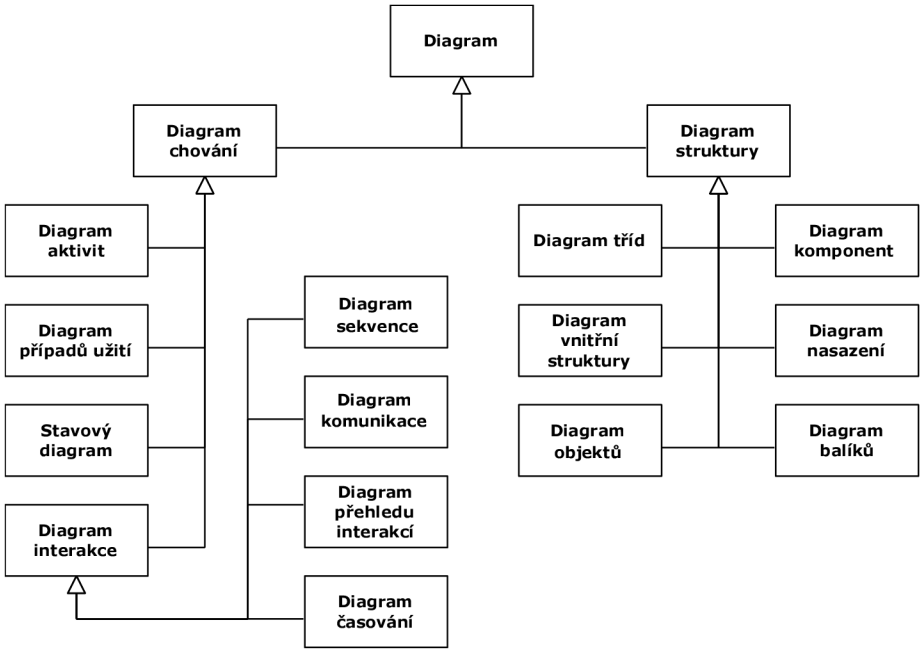
Celkový systém modelovaný pomocí UML se může skládat z vícero typů diagramů, neboli z vícero úhlů pohledů na náš systém. UML od verze 2.0 zavedlo klasifikaci diagramů do dvou skupin (graficky zpracováno v obrázku 2.1):

1. *Diagramy chování a interakce* – popisují, jak mezi sebou jednotlivé prvky vyvíjeného sytému komunikují a interagují.
2. *Strukturální diagramy* – vyjadřují fyzické uspořádání a závislosti prvků systému.

Diagramy chování a interakce lze dále dělit na:

- *Stavový diagram* (angl. „State machine diagram“) – popisuje vnitřní stavy elementu a přechody mezi nimi.
- *Diagram aktivit* (angl. „Activity diagram“) – rozšiřuje stavový diagram tím, že modeluje toky informací mezi elementy v systému.
- *Diagram případů užití* (angl. „Use case diagram“) – umožňuje zachytit funkční požadavky na systém.
- *Diagram sekvence* (angl. „Sequence diagram“) – znázorňuje interakci mezi jednotlivými elementy v čase tím, že mezi nimi lze modelovat posloupnost zasílání zpráv nebo znázornit časově omezenou existenci elementů.
- *Diagram komunikace* (angl. „Communication diagram“) – je podobný diagramu sekvence s tím rozdílem, že zaznamenává statickou strukturu (propojení) elementů a komunikaci mezi nimi.
- *Diagram přehledu interakcí* (angl. „Interaction overview diagram“) – jedná se o kombinaci diagramů sekvence a aktivit.

[2] Pro anglický výraz „round-trip“ neexistuje ustálený český překlad, proto jej zde uvádím v originální podobě.



Obrázek 2.1: Hierarchická klasifikace typů UML diagramů [4]

- *Diagram časování* (angl. „Timing diagram“) – slouží k modelování systémů, které běží v reálném čase.

Strukturální diagramy se dále člení na:

- *Diagram tříd* (angl. „Class diagram“) – popisuje statické vlastnosti tříd a rozhraní a vztahy mezi nimi. Blíže se tomuto druhu diagramu věnuje podkapitola 2.2.2.
- *Diagram komponent* (angl. „Component diagram“) – umožňuje logicky seskupit elementy systému do větších celků a ty mezi sebou propojit skrze vstupní a výstupní místa zvané porty.
- *Diagram vnitřní struktury* (angl. „Composite structure diagram“) – zachycuje spojení mezi diagramy tříd a komponent (náhled na celkovou dekompozici systému).
- *Diagram nasazení* (angl. „Deployment diagram“) – slouží k popisu rozmístění a navázání elementů systému na konkrétní hardware.
- *Diagram objektů* (angl. „Object diagram“) – je podobný diagramu tříd s tím rozdílem, že ukazuje objekty (instance tříd) a jejich propojení v určitém časovém okamžiku.
- *Diagram balíků* (angl. „Package diagram“) – znázorňuje logické seskupení jednotlivých tříd, rozhraní i dalších balíčků či jmenných prostorů.

2.2.2 Diagram tříd

Nejběžnějším typem UML diagramu je strukturální diagram zvaný *diagram tříd* (angl. „Class diagram“), který poskytuje návrhářům a programátorům vhled do vnitřního uspořádání systému. Ve smyslu použití diagramu tříd při návrhu mluvíme spíše o *návrhových třídách*, které lze popsat následovně: „Návrhové třídy jsou třídy, jejichž specifikace je na takovém stupni, že je lze implementovat“. [5] Jeho strukturu popisují následující podkapitoly.

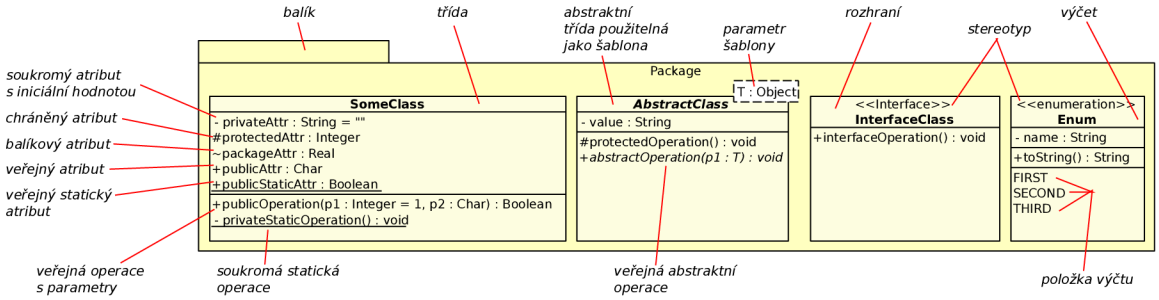
Třídy

Třídy jsou jedním ze základních předmětů UML diagramu tříd. Jejich rozdělení, respektive použití, odpovídá požadavkům objektově orientovaných programovacích jazyků. Dělí se tedy na (viz obrázek 2.2) :

- *třída* – klasická instanciovatelná třída (v obrázku zvaná `SomeClass`),
- *abstraktní třída* – abstraktní třída, jakožto i metoda, se vždy zapisuje kurzívou (v obrázku zvaná `AbstractClass`),
- *rozhraní* – třída se stereotypem «`interface`» nad názvem (v obr. `InterfaceClass`),
- *výčet* (neboli *enumerace*) – označuje se stereotypem «`enumeration`» (v obr. `Enum`),
- *asociační třída* – speciální druh instanciovatelné třídy, kterému se věnuje podkapitola 2.2.2.

Pojmenování všech druhů tříd se řídí specifikací implementačního jazyka, tedy velké počáteční písmeno následováno dalšími znaky (bez bílých znaků).

Třídy (viz obrázek 2.2) jsou složeny ze skupin atributů a operací, přičemž UML standard zavádí pojem *vlastnosti* (angl. „features“), kterým se věnuje podkapitola 2.2.2. [7] Je vhodné zmínit, že rozhraní by ze své podstaty nemělo obsahovat atributy. Toto známé pravidlo ovšem porušuje nová verze prog. jazyka Java ve verzi 8.



Obrázek 2.2: Hierarchická klasifikace typů UML diagramů [4]

Balíky

Balíky, neboli *jmenné prostory*, slouží k logickému členění tříd a dalších balíků do skupin. Jedná se tedy o určité hierarchické dělení, které podporuje jeden z konceptů objektově orientovaného návrhu zvaného *zapouzdření*.

Ukázka grafického znázornění daného předmětu je ukázáno na obrázku 2.2, kde balík nese název `package.name`. Je tedy zřejmé, že pojmenování se sestává z malých písmen a je odděleno tečkou (pro jazyk C# i Java). Ve zkratce tedy odpovídá notaci výsledného programovacího jazyka.

K propojení balíků s dalšími balíky či třídami, lze využívat vztahů *závislosti*, případně i *dědičnosti*.

Šablony (Generické třídy)

Namísto určení konkrétního datového typu použitého uvnitř dané třídy, je návrhářovi umožněno definovat tento typ více abstraktně a tak rozšířit použitelnost dané třídy. V takovém případě mluvíme o tvorbě *šablon* (šablonování podporuje pouze prog. jazyk C++), respektive *generických* (*parametrizovatelných*) *třídách*. Rozdíl mezi generickou a parametrizovanou třídou je následující – parametrizovatelná třída je potomkem generické třídy s přiřazenou hodnotou datového typu. [5]

Na obrázku 2.2 se jedná o generickou třídu (navíc abstraktní) s názvem `AbstractClass`, kde `T : Object` v pravém horním rohu značí *generický parametr* s názvem `T` a s datovým typem `Object`.

Například v programovacím jazyce Java by takto navržená generická třída měla deklaraci zapsanou následovně:

```
public abstract class AbstractClass<T extend Object>
```

a případná deklarace parametrizované třídy by mohla vypadat takto:

```
public class ParamClass extends AbstractClass<String>
```

Vlastnosti

Vlastnosti, neboli atributy a operace, třídy jsou seskupovány a odděleny vodorovnou čárou, přičemž atributy se ve třídě uvádějí jako první.

Grafická notace pro *atribut třídy* má následující tvar [7]:
`viditelnost název : typ <[násobnost]> <{omezení}> <= počáteční_hodnota>`
kde elementy uvozené do ostrých závorek jsou volitelné a popis jednotlivých prvků je následující:

- `Viditelnost` atributu lze zaznačit znakem `+` pro veřejnou, `#` chráněnou, ∼ balíkovou a `-` soukromou viditelnost.
- `Název` je složen ze znaků (kromě bílých) odpovídající notaci programovacích jazyků.
- `Typ` je datový typ atributu, jehož název se také řídí možnostmi zvoleného prog. jazyka.
- `Násobnost` udává počet instancí daného typu v atributu. Může obsahovat stejné násobnosti jako u asociace (podkapitola 2.2.2).
- `Omezení` atributu je text obsahující buď abstraktní popis podmínky, nebo podmínka ve tvaru OCL (podkapitola 2.2.2).
- `Počáteční_hodnota` je hodnota odpovídající použitému datovému typu atributu.

Grafická notace pro *operaci třídy* se skládá z [7]:
`viditelnost název(<seznam_parametrů>) : návratový_typ <{omezení}>`
kde elementy uvozené do ostrých závorek jsou volitelné a popis jednotlivých prvků je následující:

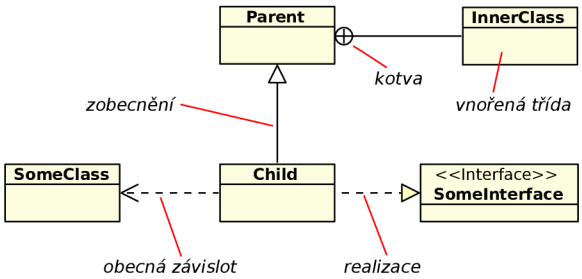
- `Seznam_parametrů` je seznam obsahující čárkou oddělené parametry operace ve stejném tvaru jako atribut třídy jen bez uvedené viditelnosti.
- `Návratový_typ` je datový typ výsledku operace, jehož název se také řídí možnostmi zvoleného prog. jazyka.

UML notace dále poskytuje možnost znázornit další typy modifikátorů jako je *statický člen* pomocí podtržení celé notace vlastnosti, nebo *abstrakci* (pouze u operací) pomocí notace psané v kurzívě.

Vztahy

UML diagram tříd definuje několik typů vztahů, neboli relací mezi předměty, (příklad jejich použití lze vidět na obrázcích 2.3 a 2.4), které popisuje následující výčet [4][5]:

- *Závislost* – nejobecnější vztah, který diagram tříd podporuje, vyjadřující, že změna v jednom předmětu ovlivňuje význam závislého předmětu. Jedná se o přerušovanou orientovanou čáru s šipkou na konci (u závislého elementu) – – ⩾. Z tohoto typu vychází všechny ostatní vztahy, které lze odlišit buď grafickým znázorněním, čemuž odpovídají další zde uvedené vztahy, nebo přidáním stereotypu (viz podkapitola 2.2.2).
- *Zobecnění* – jedná se o způsob, jak znázornit dědičnost mezi dvěma předměty, tedy relaci mezi obecnějším a přesněji definovaným předmětem. Vztah zobecnění se znázorňuje plnou čarou s trojúhelníkovou šipkou na konci (u obecnějšího elementu) —▷.
- *Realizace* – vztah udávající, že závislý element realizuje (implementuje) předepsané chování jiného elementu. Grafická notace je obdobná jako u vztahu zobecnění s tím rozdílem, že čára je přerušovaná – –▷. V kontextu diagramu tříd se používá mezi rozhraním a implementační třídou.
- *Kotva* – obecný symbol ve tvaru znaménka plus vnořeného do kružnice spojeného plnou čarou ⊕— s obsaženým elementem uvnitř zdrojového elementu. Jednoduše řečeno tento vztah slouží pro znázornění vnořování. U popisovaného diagramu tříd lze tímto způsobem znázornit vnořené třídy.
- *Asociace* – popisuje množinu spojení mezi elementy. Jedná se nejkomplexnější vztah, který lze v UML diagramu tříd použít a proto je jeho popis vyčleněn do samostatné podkapitoly 2.2.2.

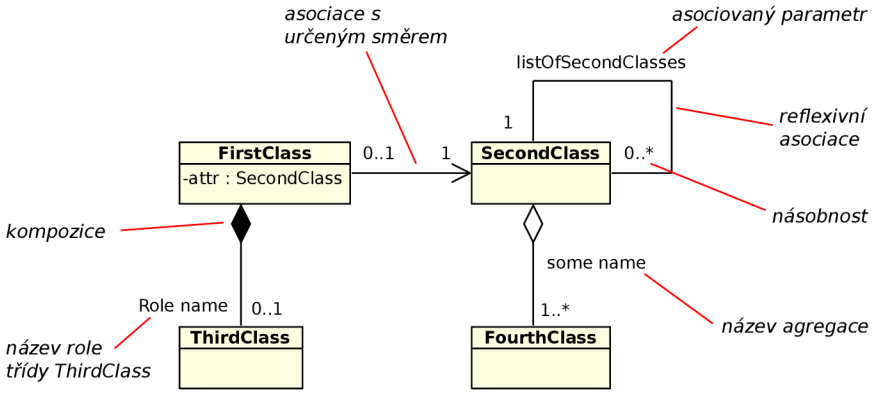


Obrázek 2.3: Základní vztahy mezi předměty (kromě asociace)

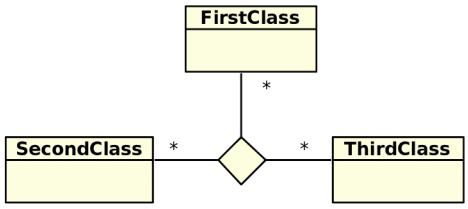
Vztah asociace a její druhy

Pokud existují dvě koncepčně spojené třídy, pak říkáme, že dané spojení se nazývá *asociace*. [6] V UML diagramu tříd se graficky označují plnou čárou (případně jednostranně orientovanou šipkou u asociovaného elementu) a syntaxe asociace může obsahovat následující prvky:

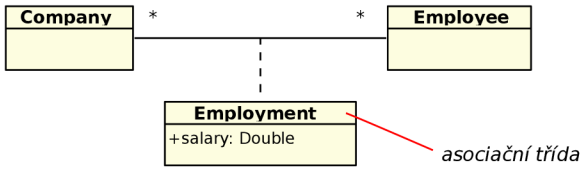
- *Název asociace* – fráze uváděná uprostřed vztahu. Jedná se o nepovinný údaj, jeho uvedení však zvyšuje čitelnost diagramu.
- *Název role* (asociovaný parametr) – uvádí se název proměnné na konci asociace (v případě obousměrné na obou koncích). Případně lze vynechat a uvést v seznamu atributů dané třídy.
- *Násobnost* – je jednoduchým typem omezení udávající počet instancí, které lze mít asociovány. Uvádí se vždy na obou koncích asociace ve tvaru 0..1 (nula nebo jedna), 1 (přesně jedna), 0..* (příp. *, značí nula a více), 1..* (jedna nebo více), $n..m$ (kde $n, m \in (\mathbb{N}_0 \cup \{^*\}), n \leq m$ v obdobných významech).
- *Orientovanost* – směr asociace vyznačený šipkou. V případě obousměrné asociace není třeba šipky uvádět.



(a) Základní a reflexní asociační vazba, agregace a kompozice



(b) Vícenásobná (n-ární) asociační vazba



(c) Asociační třída (řešení problému `M:N`) [5]

Obrázek 2.4: Různé způsoby vyznačení asociační závislosti

Často se stává, že v objektově orientovaném návrhu vznikají tzv. „M ku N“ ($M : N$) asociace mezi dvěma třídami, jak je vidět na příkladu 2.4c – na obou koncích asociace je násobnost „. . . a více“. V případě, pokud bychom chtěli přidat atribut, který lze zadat do obou tříd, tak musíme využít *asociační třídy*. Asociační třída se graficky znázorňuje stejně jako normální třída s tím rozdílem, že je propojena s asociací čárkovanou čarou.

Asociace však mohou být komplexnější, lze mít asociováno více elementů než jen 2, pak se tyto asociace označují za *vícenásobné*, neboli *n-ární*. Graficky jsou znázorněny prázdným kosočtvercem, jak je tomu na obrázku 2.4b pro ternární asociaci. Dále je možné mít i *reflexní asociaci*, tedy případ kdy element asociuje sám sebe (případ třídy `SecondClass` na obrázku 2.4a).

UML standard dále rozšiřuje vztah asociace na:

- *Agregace*[3] – je volná, příp. částečná, asociace mezi předměty znázorňující, že agregující předmět se skládá z agregovaných. Agregované předměty mohou existovat i samostatně, pokud jsou však vytvořeny agregujícím elementem, pak zanikají současně při zániku vlastníka. V UML se jedná o plnou čáru s prázdným kosočtvercem na konci (u agregujícího předmětu) —◇. Pro lepší vizualizaci vazby si lze představit agregaci mezi osobním počítačem (agregující předmět) a periferiemi (agregované předměty).

- *Kompozice* – je těsná, příp. pevná, asociace mezi předměty, které udává, že komponující element je složen z komponovaných s tím, že komponované elementy nemohou existovat samostatně a při zániku komponujícího předmětu zanikají též. Kompozice je tedy rozšířením agregace. V UML ji lze zakreslit plnou čarou s vyplněným kosočtvercem na konci (u komponujícího elementu) —◆, kde se násobnost nemusí uvádět (vždy 1). V reálném prostředí si lze představit vztah mezi stromem a jeho listy.

Mechanismy rozšiřitelnosti

Pro dodržení určité míry univerzálnosti při modelování pomocí UML slouží mechanismy rozšiřitelnosti. UML definuje tři typy [5][7]:

1. *Omezení* (angl. „Constraints“) jsou podmínky ve tvaru textového řetězce uzavřené ve složených závorkách {}. Podmínka se vztahuje k danému elementu a lze ji uvézt buď abstraktním popisem, nebo využít jazyka OCL[4].

2. *Stereotypy* (angl. „Stereotypes“) rozšiřují počet elementů (předmětů a vztahů) v metamodelu UML tím, že rozšiřují, respektive blíže specifikují, použití již existujícího elementu. Stereotypy se uvozují do dvojitých ostrých závorek «». Existuje jich mnoho, například [4]:

   - «`interface`» rozšiřující obyčejnou třídu na rozhraní,
   - «`use`» obyčejný vztah závislosti se specifickým významem značící, že jedna třída používá (ve smyslu použití vlastností) třídu závislou,
   - «`create`» vztah závislosti definující, že jedna třída vytváří instanci závislé třídy.

3. *Označené hodnoty* (angl. „Tagged values“) také rozšiřují použitelnost již existujících elementů. Tento typ byl však od UML verze 2.0 odstraněn a nahrazen výše uvedenými typy, tedy omezeními a stereotypy.

2.3 Java

Java není pouhý imperativní objektově orientovaný programovací jazyk, ale také celá platforma vznikající od počátku 90. let minulého století pod hlavičkou společnosti Sun Microsystem (nyní Oracle). Jakožto platforma disponuje sadou nástrojů pro vývoj, běh i podporu softwaru založeného na programovacím jazyku Java, respektive na Java *bytecode*. Bytecode lze popsat jako *mezikód* získaný přeložením zdrojových kódů psaných v prog. jazyce Java, dokonce existují i překladače z jiných jazyků (např. Python, Ruby, Scala a další). Mezikód je

[3] Pro zajímavost M. Fowler zastává názor, že agregace je nadbytečný vztah matoucí většinu uživatelů UML a doporučuje jej nepoužívat. A to především z důvodu, že jej lze bez problému nahradit asociací. [4]

[4] Object Constraint Language (OCL) je deklarativní funkcionální specifikační jazyk sloužící pro vyjádření invariantů, neboli podmínek, v UML diagramech. Jeho specifikaci lze nalézt na internetových stránkách OMG skupiny http://www.omg.org/spec/OCL/.

spustitelný na virtuálním stroji zvaném *Java Virtual Machine* (dále pouze JVM), kterému se blíže věnuje podkapitola 2.3.1. Tímto je zajištěna určitá nezávislost na platformách, které dokáží aplikace psané pro platformu Java spustit.

Platforma Java se štěpí dle použití na určitých zařízeních na tři hlavní podskupiny [8]:

- *Java SE* – platforma pro aplikace běžně provozované na osobních počítačích,
- *Java ME* – pro mobilní a vestavěné aplikace,
- *Java EE* – pro podnikové a jiné rozsáhlé systémy.

Java se dále dělí do dvou distribucí – pro koncové uživatele a pro vývojáře. Koncový uživatel potřebuje pouze zmíněný JVM, který je implementován v distribuci *Java Runtime Environment* (dále JRE) společně se sadou základních knihoven zvaných *Java API*. Zatímco vývojářům nabízí společnost Oracle JRE společně se sadou pomocných nástrojů (např. překladač zvaný `javac`) v distribuci zvané *Java Development Kit* (JDK).

Poslední obecný odstavec se věnuje nejdůležitějším změnám v posledních čtyřech verzích platformy a programovacího jazyka Java. Začneme od verze 5, kde byly představeny zásadní novinky a sice – podpora generických tříd, anotace, samostatná třída pro výčet a tzv. „for-each“ cykly. Verze 6 nepřinesla žádné zásadní novinky pouze opravovala předchozí verzi. Oproti tomu v roce 2011 v Java 7 byla do platformy přidána podpora pro dynamické jazyky (tedy programovací jazyky bez typové kontroly). Z pohledu prog. jazyka se usnadnila práce s použitím generických tříd a práce se soubory. Aktuální verze Java 8 přinesla podporu pro lambda výrazy a mírně kontroverzní možnost zapsat definici metody již v rozhraní. [8]

2.3.1 Java Virtual Machine

Java Virtual Machine, zkráceně JVM, je abstraktní výpočetní stroj využívající *virtualizaci*[5], který běží nad požadovaným operačním systémem. Tato architektura tak umožnila platformní nezávislost ale i zabezpečení proti hrozbám, které mohou do hostitelského systému proniknout, protože JVM běží nad uzavřeným počtem zdrojů (v tzv. „sandboxu“). Dále právě díky virtualizaci je možné v jazyku Java využít mimo jiné automatickou správu paměti a správu výjimek.

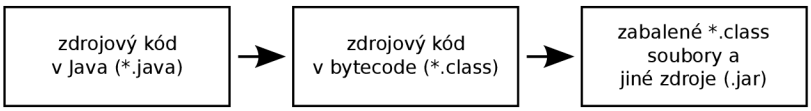
Fundament JVM specifikuje skupina odborníků pod vedením Tima Lindholma v knize The Java® Virtual Machine Specification, ve které se vyskytují tři podstatné vlastnosti:

1. Tak jako ostatní reálné výpočetní stroje má i JVM svoji instrukční sadu zapsanou v mezikódu zvaném *bytecode*. Tyto instrukce jsou specifikovány ve výše zmíněné knize.
2. Binární formát souborů zvaných `class`, který obsahuje mezikód a další informace o třídě v binární formě (dále podkapitola 2.3.2).
3. Algoritmus pro verifikaci zaváděných programů tak, aby nedošlo k ohrožení integrity samotného virtuálního stroje. [9] [10]

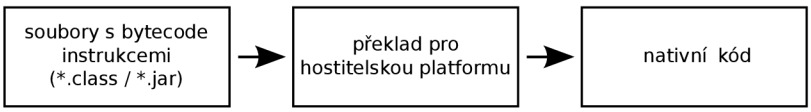
Obrázek 2.5 vizualizuje nejprve průběh tvorby `class` souborů, respektive `jar` souboru, který je pouhým archivem obsahující tyto a případně další pomocné soubory. Překlad nemusí nutně probíhat ze zdrojových kódů v prog. jazyce Java ale, za použití patřičného překladače, i z jiných jazyků (např. Python, Scala a jiné). Při spuštění se používá tzv. „Class loader“,

[5] Virtualizace v podstatě představuje iluzi, v níž nějaký zdroj zmnožíme a každý uživatel dostane jednu nebo více z těchto kopií k dispozici. [11]

který nalezne, zkontroluje a zinicializuje potřebné `class` soubory. Dále se využívá služeb překladače z mezikódu do nativního kódu hostitelského systému, který se nazývá *Just-In-Time* (zkráceně JIT)[6].



(a) Překlad zdrojových kódů do spustitelného formátu v JVM



(b) Spuštění aplikace na JVM

Obrázek 2.5: Tvorba a spuštění aplikace pro JVM

Jazyky založené na mezikódu umožňují převést svůj mezikód zpět do čitelné podoby. Takový postupu se nazývá *dekompilace*. Pro tyto účely lze v jazyku Java využít nástroje `javap`, který je distribuován v rámci JDK. [12] Nebo lze za běhu v Java platformě přistupovat k informacím o objektu, s nimiž pracuje (jako je počet a název atributů třídy, metod atd.) pomocí vlastnosti zvané *reflexe*, které se však u této platformy věnovat nebudu, neboť má práce ji neuvažuje.

2.3.2 Struktura `class` souboru

Nejpodstatnější vlastností platformy Java jsou dozajista soubory s instrukcemi pro JVM, `class` soubory, a proto je třeba si je rozepsat podrobněji.

Obecně šířený fakt, že každý zdrojový soubor `.java` má svůj `.class` soubor, není tak úplně validní a je třeba ho poupravit – každá třída v jazyce Java má svůj `.class` soubor. V jazyce Java je totiž možné zapsat více tříd do jednoho `.java` souboru. Klasická třída, rozhraní nebo výčet nese unikátní název, a proto je tento název použit s již dobře známou příponou `.class`. Mějme tedy definovanou třídu `A`, pak názvy souborů s instrukcemi tvoří předpona `A$` a následuje:

- pro každou *vnitřní*, i *statickou*, třídu název této třídy,
- pro každou *anonymní* třídu generovaný index (kladné celé číslo),
- a pro každou *lokální* třídu generovaný index a název této třídy. [12]

Pokud tedy má každá třída svůj `.class` soubor, tak jej lze udržovat jednoduchý a přehledný. Strukturu samotného souboru s mezikódem popisuje následující datový typ:

```
ClassFile {
   uint32          magic;
   uint16          minor_version;
   uint16          major_version;
   uint16          constant_pool_count;
   cp_info         constant_pool[constant_pool_count-1];
```

[6] Just-In-Time překladač, který se zavolá až v době spuštění požadovaného programu, čímž urychluje běh programů založených na mezikódu.

```
    uint16          access_flags;
    uint16          this_class;
    uint16          super_class;
    uint16          interfaces_count;
    uint16          interfaces[interfaces_count];
    uint16          fields_count;
    field_info      fields[fields_count];
    uint16          methods_count;
    method_info     methods[methods_count];
    uint16          attributes_count;
    attribute_info  attributes[attributes_count];
}
```

Konstantní hodnota `magic` (známá `0xCAFEBABE`) identifikuje, že se jedná o `.class` soubor a hodnoty pro `minor_version` a `major_version` zase určují verzi Java platformy. Každá třída má svůj seznam použitých literálů zvaný *Constant pool* (více podkapitola 2.3.2). `access_flag` je maska identifikující typ (rozhraní, výčet, ...) a modifikátory (veřejné, abstraktní, ...) třídy. Hodnota `this_class`, resp. `super_class`, je odkaz do seznamu Constant pool se záznamem reprezentujícím danou, resp. nadřazenou, třídu. Následuje seznam implementovaných rozhraní `interfaces`. Samotný obsah třídy, tedy atributy a metody, popisují hodnoty `fields` (podkapitola 2.3.2) a `methods` (podkapitola 2.3.2). Poslední hodnota zvaná `attributes` udává dodatečné informace o třídě, například informace o vnořených třídách, generované výjimky apod. [9]

Constant pool

Tabulka, nebo spíše seznam literálů, která je adresovatelná uvnitř `.class` souboru, tak lze popsat *Constat pool*[7]. Jedná se, co do velikosti, o nejrozsáhlejší část. Každý literál je označen indexem, na který se lze v rámci daného mezikódu odkazovat, a značkou („tagem“) specifikujícím o jaký typ se jedná. Jsou zde uloženy jak číselné a řetězcové konstanty, tak i další reference (případně dvojice referencí) dle významu záznamu.[13] Například se zde uvádí typ a název atributu nebo operace třídy, typ a název případného rozhraní, odkaz na název třídy apod. [9]

Datové typy jsou v Constant pool zaznamenány pomocí řetězce. V případě primitivních typů se použije první písmeno zapsané velkým písmem (datový typ `boolean` se zapisuje pomocí znaku `Z`). Pokud se jedná o datový typ určující třídu, pak je ve tvaru `L<absolutní_cesta_třídy>;`. A nakonec pro pole, respektive víceúrovňová pole, se použije před znak/řetězec datového typu pole levá hranatá závorka `[`, respektive více závorek v závislosti na úrovni pole. Pro představu uvádím pár příkladů:

```
short[][]       [[S
com.package.A   L/com/package/A;
List<String>    Ljava/util/List<Ljava/lang/String;>;
```

Obsah Constant pool si lze zobrazit pomocí dekompilátoru `javap` s příznakem `-v`. Je však vhodné poznamenat, že pro co nejvíce zobrazených informací je třeba zdrojové kódy překládat se zapnutými „ladícími (debug) informacemi“.

[7] V češtině neexistuje ustálený přepis pro anglický výraz „Constat pool“, proto jej budu používat v originálním znění.

Deklarované atributy

Deklarované atributy třídy, v JVM specifikaci zvané „fields“, lze popsat datovou strukturou:

```
field_info {
   uint16          access_flags;
   uint16          name_index;
   uint16          descriptor_index;
   uint16          attributes_count;
   attribute_info  attributes[attributes_count];
}
```

Vlastnosti atributu třídy, mezi které se řadí modifikátor viditelnosti, statičnost (`static`), konstantnost (`final`) apod., určuje maska `access_flags`. Hodnota `name_index` je odkaz do Constant pool na název atributu. Následuje hodnota, respektive odkaz na řetězec v Constant pool, popisující typový deskriptor zvaný `descriptor_index`. Poslední část vyplňují dodatečné informace `attributes` o daném atributu třídy, jako jsou anotace, odkaz na jméno zdrojového souboru a další. [9]

Úplný deskriptor atributu třídy je tvořen jmenným deskriptorem, což je absolutní cesta k třídě (v případě deklarovaných atributů pouze názvem třídy) následovaná tečkou a názvem atributu. Pak se zde nachází oddělovač v podobě dvojtečky a nakonec je uveden samotný typový deskriptor atributu. Příklad jednoduchého atributu třídy `public Integer[] array` ve třídě `A` by vypadal takto:
`A.array:[L/java/lang/Integer;`

Deklarované metody

Metody třídy v mezikódu platformy Java zastupuje záznam s datovou strukturou [9]:

```
method_info {
   uint16          access_flags;
   uint16          name_index;
   uint16          descriptor_index;
   uint16          attributes_count;
   attribute_info  attributes[attributes_count];
}
```

Významem jsou jednotlivé hodnoty obdobné popisu v předchozí podkapitole o deklarovaných atributech třídy 2.3.2.

Kompletní deskriptor metody je tvořen jmenným deskriptorem, což je absolutní cesta k vlastnické třídě (v případě deklarovaných metod pouze název třídy) následovaný tečkou a názvem metody. Poté následuje dvojtečka a typový deskriptor, který obsahuje seznam datových typů parametrů metody uvozený do kulatých závorek. Nakonec je uveden dat. typ návratové hodnoty. Příklad jednoduché metody `void test(int i, Object[] array) {}` ve třídě `A` by vypadal takto:
`A.test:(I[/java/lang/Object;)V`

2.4 C# .NET

C# je jednoduše použitelný objektově orientovaný programovací jazyk vyvinutý společností Microsoft, nyní schvalovaný standardizační organizací ECMA [14], vycházející ze dvou pro-

gramovacích jazyků Java a C++. Stejně jako Java disponuje C# automatickou správou paměti a správou výjimek.

.NET Framework, spravovaný též společností Microsoft, lze popsat jako platformu obsahující několik softwarových produktů pro bezproblémový běh aplikací v operačním systému Microsoft Windows. Platformu tvoří především dva produkty:

1. *Framework Class Library* (FCL), neboli rozsáhlý soubor knihoven použitelných při vývoji aplikací napříč různými programovacími jazyky.

2. *Common Language Runtime* (CLR), což je stejně jako JDK abstraktní výpočetní stroj používající virtualizaci. Více se mu věnuje podkapitola 2.4.1.

Pokud tedy aplikujeme .NET Framework při vývoji aplikací (podporované jsou jazyky C#, Visual Basic .NET, Delphi, Managed C++ apod.) lze dosáhnout vyšší bezpečnosti a nezávislosti na daném hardwaru i na verzi operačního systému Windows.[15] Existuje dokonce více distribucí, které tuto nezávislost rozšiřují:

- Microsoft .NET Framework – standardní distribuce pro stolní počítače s OS Windows,
- Microsoft .NET Compact Framework – pro mobilní zařízení,
- Microsoft .NET Micro Framework – pro vestavěné zařízení,
- Mono – nezávislá distribuce pro operační systémy Unix (Linux, Mac OS).

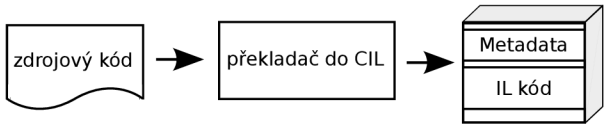
V poslední odstavci představím nejdůležitější změny mezi posledními třemi verzemi jazyka C#, kde u každé verze bude v závorce uvedena společně představená verze .NET Framework. C# 3.0 (.NET 3.5) přidává především podporu pro jednodušší práci s dotazy nad seznamy i databázemi zvanou LINQ a podporu pro lambda výrazy. Další verze C# 4.0 (.NET 4.0) představuje možnost dynamického typování. Poslední verze 5.0 (.NET 4.5, aktualizováno na 4.5.2) dodává možnost používat asynchronní metody a tím i lépe využívat možnosti moderního hardware. [16]

2.4.1 Common Language Runtime

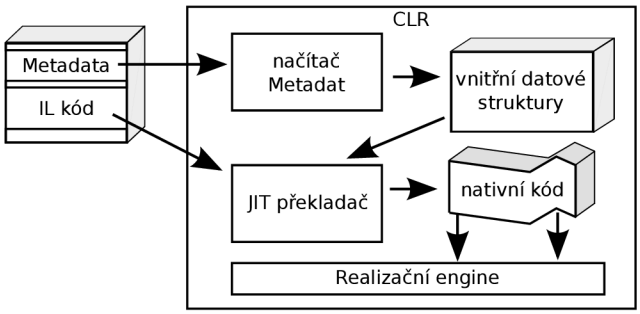
Common Language Runtime (dále jen CLR) si lze, stejně jako JVM, představit jako abstraktní virtuální stroj běžící nad hostitelským operačním systémem. Díky tomu mohou programy psané v jazyku C# mimo jiné využívat automatickou správu paměti a správu výjimek. [16] CLR je implementací standardizované specifikace *Common Language Infrastructure* (dále CLI) popisující vlastnosti proveditelného kódu (*Common Language Specification*) a prostředí pro jeho běh (*Virtual Execution System*). [17]

V CLR se tedy spouští programy psané v tzv. řízeném („managed“)[8] kódu, který lze získat pomocí překladače ze zdrojového kódu podporovaného jazyka do mezijazyka zvaného *Common Intermediate Language* (dále CIL). Například mnou popisovaná platforma .NET používá CIL jazyk zvaný .NET IL Assembly, který je popsán v podkapitole 2.4.2. CIL obalí takto vygenerovaný kód společně s dodatečnými informacemi o vygenerovaném kódu (metadata) do struktury zvané *assembly*. Assembly lze do češtiny přeložit jako sestavení, ovšem výstižnější je používat anglický výraz. Ta je umístěna v souboru s koncovkou `.exe` nebo `.dll`. Při spuštění se načte požadovaná assembly a pomocí *Just-In-Time* překladače se přeloží do nativního kódu hostitelského systému, který lze spustit. [16] Celý postup od zdrojového kódu programu k jeho spuštění popisuje obrázek 2.6.

[8] Řízený („managed“) kód se vyznačuje vlastností, kdy je kód plně pod správou CLR. Neřízený („unmanaged“) kód se převede přímo do strojového kódu a nelze například využívat automatické správy paměti.



(a) Překlad zdrojových kódů do formátu pro CLR



(b) Spuštění aplikace na CLR

Obrázek 2.6: Tvorba a spuštění aplikace pro CLR [18]

2.4.2 .NET IL Assembly

Jazyk *IL Assembly* (dále jen ILAsm), dříve známý jako Microsoft Intermediate Language, je druhem CIL jazyka, tedy nízko úrovňového jazyka speciálně navrženého tak, aby popsal každou funkcionalitu, kterou nabízí CLR. Pro překlad ILAsm do spustitelné formy lze využít překladače `ilasm`.

Jazyk ILAsm představím jen okrajově, neboť pro účely mé aplikace je podstatnější znalost *reflexe* (podkapitola 2.4.3), která vychází přímo z vlastností jazyka C# v kombinaci s .NET platformou, tím je jednodušší na použití. Následující příklad, zda je zadané číslo sudé, nebo liché, je převzat z knihy pana Lidina .NET IL Assembler [18]:

```
//----------- Hlavička programu
.assembly extern mscorlib { auto }
.assembly OddOrEven { }
.module OddOrEven.exe
//----------- Deklarace třídy
.namespace Odd.or {
    .class public auto ansi Even extends[mscorlib]System.Object {
//----------- Deklarace atributů
        .field public static int32val
//----------- Deklarace metod
        .method public static void check() cil managed {
            .entrypoint
            .locals init(int32Retval)
        AskForNumber:
            ldstr"Enter a number"
            call void[mscorlib]System.Console::WriteLine (string)
            ...
            call vararg int32sscanf(string, int8*, ..., int32*)
            ...
        }
    }
}
```

```
//----------- Globální položky
.field public static valuetype CharArray8 Format at FormatData
//----------- Definování konstatních dat
.data FormatData = bytearray(25 64 00 00 00 00 00 00)// % d . . . . . .
//----------- Deklarace typu pro pole znaků
. class public explicit CharArray8 extends[mscorlib]System.ValueType { .size8 }
//----------- Volání neřízeného kódu
.method public static pinvokeimpl("msvcrt.dll" cdecl)
    vararg int32sscanf(string, int8*) cil managed preservesig{ }
```

Všechny *assembly* začínají svojí hlavičkou, kde se uvádí použité knihovny (většina programů pro .NET Framework používá základní knihovnu `Mscorlib.dll`). Dále se uvede název assembly a poté se již deklaruje modul, jeho jmenné prostory a třídy.

V případě použití tříd v jiných assembly je nutné použít notaci ve tvaru: název assembly v hranatých závorkách následovaný cestou (v tečkovém tvaru) k použité třídě. Statické elementy takto popsané třídy jsou poté běžně přístupné při uvedení řetězce `::` jako oddělovače názvu třídy a elementu.

Uvnitř deklarované třídy pak již lze deklarovat další elementy, které CLR v .NET Framefork podporuje. Ty jsou obdobné elementům v jazyku C#[18]:

- *Metoda* – začíná klíčovým slovem `.method` následován různými modifikátory (viditelnost apod.), návratovým typem, názvem, parametry metody uzavřenými v závorkách, implementačními příznaky a nakonec implementací metody uzavřené do složených závorek. Implementace metody je pak tvořena instrukcemi jazyka ILAsm. jejich podrobný výčet uvádí výše zmíněná kniha.
- *Konstruktor* – je speciálním druhem metody, a proto je složený stejně jako obyčejná metoda s tím rozdílem, že namísto názvu metody se uvádí klíčové slovo `.ctor`.
- *Událost* – deklaraci události lze provést začínajícím klíčovým slovem `.event`, názvem a deklarací funkcí událostí.
- *Atribut* – začíná klíčovým slovem `.field` následován různými modifikátory (viditelnost apod.), datovým typem a samotným názvem. Případnou výchozí hodnotu lze zadat za deklaraci atributu přidáním znaku `=` a požadovanou hodnotou.
- *Vlastnost* – vytváří se klíčovým slovem `.property` podobně jako atribut s tím, že za něj lze do složených závorek uvést deklaraci metody pro získání hodnoty (začíná `.get`), nastavení hodnoty (začíná `.set`) a jiné metody `.other` (např. obnovení hodnoty).
- *Konstanta* – začíná klíčovým slovem `.data`, pak je uveden název již deklarované proměnné/atributu, znak `=` a požadovaná hodnota.

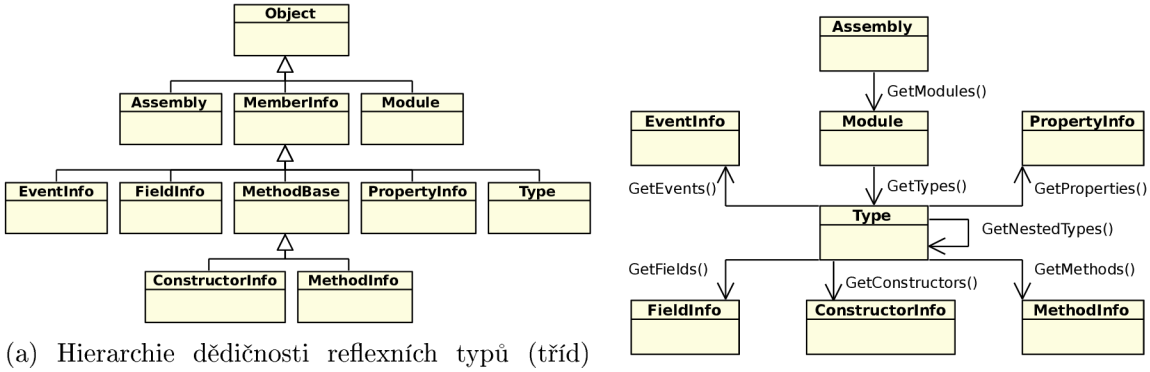
ILAsm umožňuje deklarovat i globální proměnné a funkce, které představuje poslední část ukázkového zdrojového kódu. Detailnější informace o jazyku .NET IL Assembly poskytuje kniha od pana Lidina. [18]

2.4.3 Reflexe

Jak již bylo zmíněno při překladu zdrojových kódů do CIL, dochází také k tvorbě metadat, tedy dodatečných informací pro zavádění tříd při běhu programu. Kód v CIL obsahuje stejné

informace jako původní zdroj. kód s výjimkou názvů lokálních proměnných, komentářů a preprocesorových direktiv. *Reflexí* se pak označuje procházení těchto metadat a přeloženého kódu za běhu programu. [16]

V .NET Framework je aplikační rozhraní reflexe soustřeďováno do jmenného prostoru s názvem `System.Reflection`, jehož hierarchickou strukturu popisuje obrázek 2.7a. Programy psané pro .NET jsou tvořeny základními jednotkami zvanými *typy* (třída `Type`), které obsahují členy a další vnořené typy. Typy jsou zapouzdřeny v *modulech* (třída `Module`) a moduly zase v *assembly* (třída `Assembly`). Instance třídy `Type` reprezentuje metadata pro deklaraci typů (tříd) v aplikaci. Typy pak obsahují další informace, jako jsou konstruktory, atributy třídy (angl. „fields“), vlastnosti (angl. „properties“), metody a události (angl. „events“). Pokud tedy známe hierarchii, tak lze využít patřičných metod pro navigaci skrze ni (viz obrázek 2.7b). Všechny dostupné funkce jsou přehledně vypsány včetně příkladu použití v knize bratrů Albahariových C# 5.0 in a Nutshell. [16]



(a) Hierarchie dědičnosti reflexních typů (tříd) .NET [16]

(b) Princip získávání informací o assembly

Obrázek 2.7: Struktura a pohyb reflexí v .NET[9]

Zbývá poslední důležitá otázka: Jak se lze dostat k metadatům uvnitř programu? Každý program psaný pro .NET má svůj *doménový prostor* zvaný `AppDomain`, který slouží k oddělení spuštěných programů tak, aby do sebe nezasahovaly. Doménový prostor obsahuje jednu a více assembly (např. pokud program využívá jiné knihovny, tak se jejich assembly zavedou do domény programu). Nyní je již jednoduché se k požadovaným metadatům dostat, jak uvádí příklad úseku kódu:

```
foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies()) {
   foreach (Module m in a.GetModules()) {
      ...
   }
}
```

2.5 Dostupná řešení

Dostupných řešení pro tvorbu UML diagramů, konkrétně diagramů tříd, poskytuje trh celou řadu jak placených, tak i zdarma. Takové nástroje se označují jako CASE (angl. „Computer Aided Software Engineering“), neboli vývoj software s využitím počítačové podpory. Mnoho

[9] Detailní struktura reflexních typů (tříd) je přístupná na http://msdn.microsoft.com/en-us/library/system.reflection(v=vs.110).aspx

placených aplikací je multifunkčních mající ovšem poměrně nákladné licence. Pro příklad lze uvést Altova UModel, MagicDraw, Enterprise Architect nebo Visual Paradigm. Poslední dvě zmíněné popíši blíže v následujících podkapitolách.

Aplikace pro modelování UML, které jsou zdarma, popřípadě šířené pod licencí otevřeného kódu (angl. „Open source“), obecně nedisponují takovým množstvím funkcí nebo neustálým vývojem jako konkurence z komerční úrovně. Lze však nalézt i velmi kvalitní bezplatné aplikace a ve spojení s filozofií otevřeného kódu se může jednotlivec, případně celá komunita, podílet na dalším vývoji. Dále se zaměřím pouze na dvě, ArgoUML pro jazyk Java a NClass pro jazyk C#.

Při porovnávání nástrojů jsem se orientoval především na použitelnost z pohledu vývojáře, na možnost použití i v ostatních operačních systémech než je Microsoft Windows a především podporu pro reverzní inženýrství a generování zdrojových kódů z diagramu tříd pro objektově orientované jazyky Java a C#.

Enterprise Architect

Prvním zde prezentovaným nástrojem pro tvorbu UML diagramů je aplikace s názvem *Enterprise Architect* [19] od společnosti Sparx Systems. Jelikož se Enterprise Architect řadí mezi placené aplikace, je tím zaručena i podpora a neustálé aktualizace ze strany tvůrců. Hlavní výhodou je plná podpora pro reverzní inženýrství i generování zdrojového kódu z diagramu tříd obou, pro mne podstatných jazyků, Java a C#.

⊕ Možnost týmové spolupráce.
Neustálé aktualizace.
Pokročilé možnosti editace diagramů a možnost generovat dokumentaci v HTML formátu.

⊖ Není multiplatformní (nutno prvně nainstalovat Wine na OS Linux či Mac OS).
Množstvím funkcí ztrácí přehlednost a jednoduchost ovládání.
Je placená.

Visual Paradigm

Dalším zástupcem modelovacího softwaru s podporou UML je *Visual Paradigm* [20], který také disponuje plnou podporou jazyků Java a C# včetně reverzního generování jak ze zdrojového kódu, tak i z přeložené aplikace či knihoven. Uživatelské rozhraní je subjektivně až nečekaně přívětivé vzhledem k jeho pokročilým funkcí. Tento nástroj používají jedny z nejznámějších společností světa jako je Apple, Intel, Adobe aj., což prezentuje jeho kvalitu.

⊕ Možnost integrace do vývojových prostředí Eclipse, Visual Studio nebo NetBeans.
Možnost týmové spolupráce.
Je plně multiplatformní.
Subjektivně přehlednější než nástroj z předchozí podkapitoly.

⊖ Nemá podporu pro rozšiřující moduly.
Vyšší pořizovací cena než u Enterprise Architect.

ArgoUML

Aplikace *ArgoUML* [21] se prezentuje především otevřeným zdrojovým kódem psaným v jazyce Java, čímž se předurčuje jeho primární funkce, a sice tvorba diagramů tříd v propojení s již zmíněným jazykem, včetně přeložených tříd `*.class`, respektive souboru `*.jar`. Podporu zdrojových kódů dalších objektových programovacích jazyků lze ovšem rozšířit formou zásuvných modulů. Aplikace však umožňuje i tvorbu ostatních diagramů z rodiny UML.

⊕ Je zdarma a nové funkce si lze naprogramovat.
Přidávání zásuvných modulů.
Aplikace je postavená čistě na JVM („Java Virtual Machine“) – multiplatformní.

⊖ Nepodporuje novější verze Java.
Nemožnost analýzy zkompilovaných zdrojových kódů psaných v jazyce v C#.
Poslední aktualizace na verzi 0.34 v roce 2011.

NClass

NClass [22] je určen pouze ke tvorbě UML diagramů tříd, což se odráží v uživatelském rozhraní – velmi přehledné a intuitivní. Jedná se o aplikaci psanou v jazyce C# za použití platformy .NET ve verzi 4.0, proto lze snadno využít vlastnosti reflexe zmíněného jazyka při analýze přeložené aplikace v něm psané. Avšak na vývoji se podílí pouze jeden člověk, takže aplikace má mnoho nedodělků. Nevýhod pro použití je tedy zatím více než výhod.

⊕ Je zdarma a nové funkce si lze naprogramovat.
Aplikace je přehledná.
Poskytuje generování zdrojového kódu v jazyce C# i Java.

⊖ Není zcela vyvinuta (nemá například podporu pro pohyb v historii úprav).
Potenciální problém při použití na OS Linux nebo OS X s projektem Mono, který není zcela totožný s .NET platformou na OS Windows.
Nelze analyzovat zdrojový kód jazyka Java ani C#.

Závěrečné srovnání

Nástroj	Diagram tříd				Zdrojový kód z diagramu tříd		Zdarma
	ze zdrojového kódu		z přeložené aplikace				
	Java	C#	Java	C#	Java	C#	
Enterprise Architect	✓	✓	✓	✓	✓	✓	✗
Visual Paradigm	✓	✓	✓	✓	✓	✓	✗
ArgoUML	✓	✓	✓	✗	✓	✓	✓
NClass	✗	✗	✗	✓	✓	✓	✓

Po otestování i ostatních UML modelujících nástrojů jsem nenalezl žádný, který by podporoval provedení analýzy nad přeloženým kódem jazyků C# .NET i Java skrze webovou službu nebo příkazový řádek. Všechny aplikace bylo třeba nejprve spustit do grafického rozhraní a odtud manuálně zadat požadavek na analýzu.

Na základě uvedených zdůvodnění nelze použít stávající nástroje a tedy požadavek mé diplomové práce na vytvoření takové aplikace, respektive webové služby, je zcela validní.

Kapitola 3

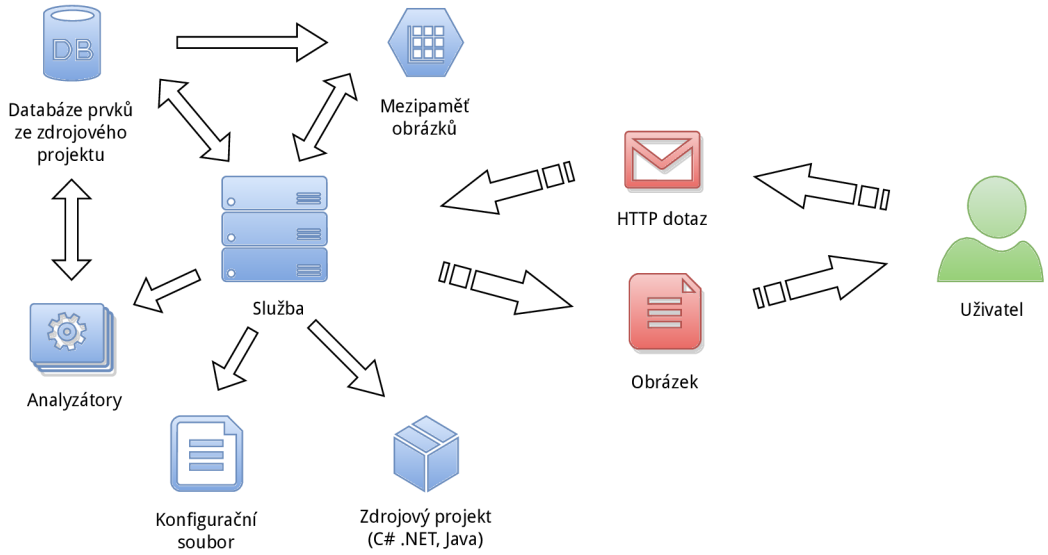
Návrh služby

V této kapitole je představen návrh aplikace, respektive služby, samotné. Jelikož se jedná o webovou službu, je nutné navrhnout, mimo jiné, způsob komunikace s koncovým uživatelem, čemuž se věnuje podkapitola 3.1. Konceptuální návrh architektury služby, popsaný v podkapitole 3.2, a konkrétní její konkrétní návrh z podkapitoly 3.4 obsahuje dvě vrstvy, a to pro analýzu vstupního zdrojového projektu přeloženého z platformy C# .NET nebo Java (podkapitola 3.3) a pro generování diagramu tříd do obrázku na základě dotazu od uživatele (podkapitola 3.5).

3.1 Základní prvky a komunikace

Schéma komunikace se službou znázorněné na obrázku 3.1 obsahuje dva hlavní aktéry – službu samotnou a koncového uživatele.

Uživatelem služby je vzdálený klient, konkrétně pak klientský systém fyzického uživatele, vzdálený server nebo další služba. Uživatel na svůj dotaz, v případě korektního zadání doménového jména (identifikátoru) vedoucího ke spuštěné službě, obdrží obrázek s požadovaným diagramem tříd.



Obrázek 3.1: Návrh komunikace mezi uživatelem a službou

Základní komponenty komunikace mezi uživatelem a službou popisuje následující výčet:

Databáze prvků slouží k uložení informací o analyzovaném projektu tak, aby nebylo nutné pro každý dotaz neustále analyzovat celý zdrojový projekt a použít jen výsledky analýzy pro sestavení požadovaného diagramu tříd.

Analyzátory jsou spustitelné soubory, které provedou analýzu zdrojového projektu na základě jeho platformy (C# .NET nebo Java aktuálně). Analyzátorům se blíže věnuje kapitola 3.2.1.

Mezipaměť obrázků poskytuje posledních n výsledků dotazů nebo výsledky často opakovaných dotazů. Definici hodnoty n nebo nutného počtu opakování obsahuje konfigurační soubor služby.

Konfigurační soubor poskytuje určitou míru přizpůsobitelnosti použití navrhované služby. Jsou zde uloženy informace o umístění zdrojového projektu, umístění nutných nástrojů podpory služby, konfiguraci mezipaměti obrázků apod.

Zdrojový projekt je umístěn, stejně jako konfigurační soubor, na službě přístupném místě, ať už se jedná o lokální nebo vzdálené umístění. Služba podporuje analýzu pro projekty vytvořené v platformách C# .NET a Java.

HTTP dotaz zasílá koncový uživatel, ve kterém specifikuje informace o chtěném diagramu tříd. Má podobu URI s přenosovým protokolem HTTP, tedy GET dotaz. Uživatel tak jednoduchým způsobem vytvoří obyčejný odkaz, který vrací obrázek.

Služba vždy na dotaz vrátí **obrázek** buď sestaveného diagramu tříd, nebo chybový obrázek obsahující chybovou zprávu. Tímto je zajištěna její jednotnost při použití. Obrázek může mít jak rastrovou, tak i vektorovou podobu, záleží na preferencích uživatele uvedených v parametrech dotazu.

3.2 Architektura služby

Aplikace je koncipována jako webová služba (angl. „Web Service“) bez uživatelského rozhraní s dvěma oddělenými vrstvami:

1. pro *analýzu* zdrojového projektu, kterou popisuje podkapitola 3.2.1,
2. pro *tvorbu* diagramů (webová vrstva) v podkapitole 3.2.2.

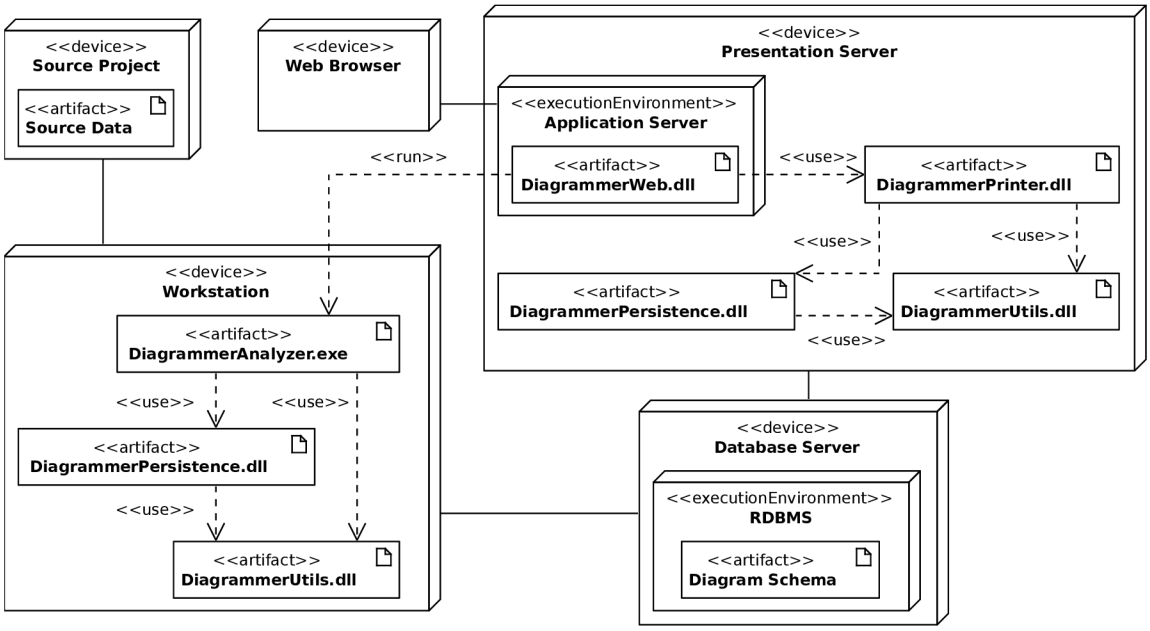
Obě vrstvy však budou spolupracovat s jednou databází a konfiguračním souborem.

Služba je postavena na platformě .NET a psaná v jazyce C#. Tento fakt umožňuje využít základní vlastnosti pro reverzní inženýrství zvolené platformy, a sice *reflexe*, která je přehlednější, rychlejší a také snazší pro realizaci analýzy vstupního přeloženého projektu na stejné platformě, než jakou by poskytovala statická analýza skrze .NET IL Assembler, nebo dokonce analýza zdrojového kódu (podkapitola 2.4.2 a 2.4.3).

Tento koncept ilustruje i diagram nasazení na obrázku 3.2. Každá vrstva obsahuje dvě podpůrné knihovny `DiagrammerPersistence`, pro přítup k databázi, a `DiagrammerUtils`, pro obecné operace (různé konverze, záznam událostí, ...).

Webová vrstva pak na příslušném aplikačním serveru obsahuje nasazené webové rozhraní aplikace `DiagrammerWeb`, které využívá knihovnu pro tvorbu diagramů `DiagrammerPrinter`.

Pro aplikační server je vhodné v prostředí Windows využít „Internet Information Services“, zkráceně IIS. Jelikož webové aplikace psané pro platformu C# .NET, jsou implicitně připravené pro nasazení právě na zmíněném typu softwarového webové serveru.



Obrázek 3.2: Diagram nasazení znázorňující výsledné propojení všech vytvořených modulů (knihoven)

3.2.1 Vrstva analýzy

Jak již bylo zmíněno v nadřazené podkapitole, pro analýzu projektů v C# s platformou .NET lze využít reflexe. Je ovšem nutné přizpůsobit analýzu i pro aplikace platformy Java. Aby se předešlo problémům s kompatibilitou, služba využije překladače z Java mezikódu do .NET IL Assembly zvaného `ikvmc` ze stále aktivního projektu IKVM.NET [23].

K oběma typům vstupních projektů tak lze přistupovat jednotně, avšak vzhledem k případnému rozšíření podpory je vhodné skrýt implementaci za rozhraní, čehož lze nejlépe docílit použitím návrhového vzoru strategie.

Analýza se provede vždy v předem uvedený čas v konfiguračním souboru, typicky v nočních hodinách, kdy je vytížení serveru nejnižší. Po provedení analýzy jsou v databázi prvků zaznamenány všechny informace o zdrojovém projektu, jako jsou názvy obsažených tříd jmenných prostorů/balíků, názvy a typy vlastností, metod apod.

3.2.2 Vrstva tvorby diagramů

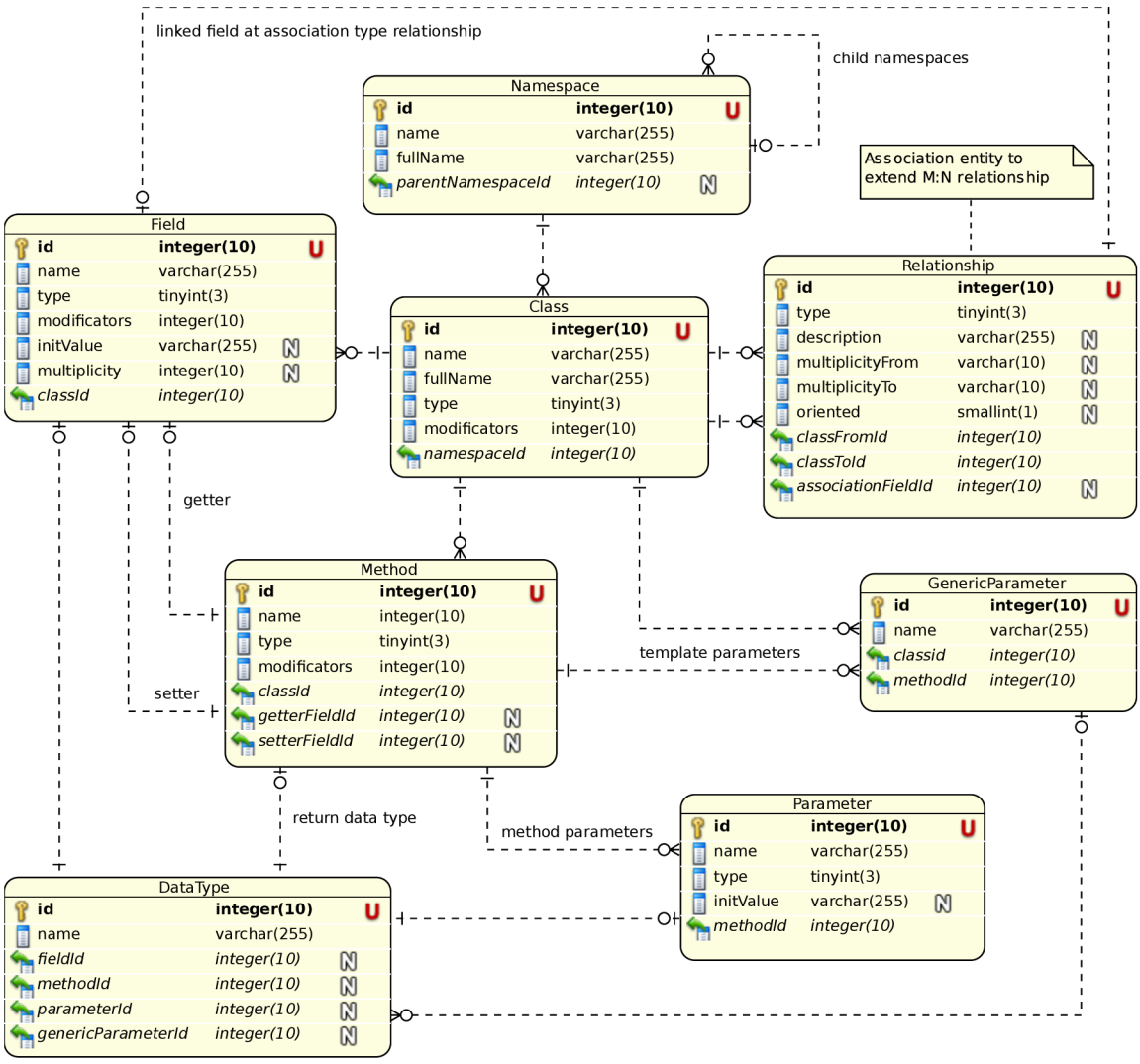
Jakmile je databáze prvků naplněna daty analyzovaného zdrojového projektu, může služba začít poskytovat diagramy tříd na základě obdržených *HTTP dotazů*, které ve svých parametrech specifikují *nastavení* – název hlavní třídy nebo jmenného prostoru/balíku (úplné umístění s tečkovou notací nezávislé na typu projektu), úroveň zanoření v hierarchii závislostí, zobrazení různých druhů informací o třídách (veřejné, chráněné nebo soukromé atributy a metody, pouze názvy metod) a typ obrázkového formátu.

Vizualizace předmětů (elementů) a vztahů v diagramu tříd se řídí klasickou notací definovanou v UML standardu (podkapitola 2.2) používanou i v ostatních CASE nástrojích. Umístění předmětů a vztahů se řeší automaticky pomocí grafických .NET knihoven.

V případě nastalé chyby při generování diagramu tříd se uživateli vrací zpět obrázek s bílým pozadím a textem obsahujícím popis chyby.

3.2.3 Databáze

Pro účely uchování analyzovaných dat je využito *relační databáze*, jehož strukturu prezentuje model na obrázku 3.3. Relační databáze rozšiřuje oblast možného nasazení na různé databázové servery. Oproti NoSQL (například grafová databáze) přístupu vyžaduje méně prostoru a vkládání/odstraňování dat je rychlejší, avšak za cenu pomalejších selekčních dotazů. [24][25] Jako pozitivum pro relační databázi lze také uvést množství různých implementací SŘBD (systém řízení báze dat, angl. zkratka „RDBMS“). Této vlastnosti využívá i knihovna `DiagrammerPersistence`, která podporuje Firebird, různé verze MSSQL, MySQL, Oracle DB, Postgre a SQLite.



Obrázek 3.3: Databázový relační model pro uchování analyzovaných dat

V databázi jsou uchovány informace o jmenných prostorech/balíčcích a jejich elementech (třídy, rozhraní, výčty, struktury, delegáti a anotace) včetně vazeb mezi nimi. Detailní horizontální dělení elementů, atributů, operací (metod), parametrů operací a vztahů mezi elementy umožňuje využití sloupce `type` u zmíněných entit. Všechny tyto možnosti jsou popsány v příloze E. Daná struktura odpovídá potřebám generátoru diagramů (viz kapitola 3.5).

Jako optimalizace pro přístup do databáze jsou dvě hlavní entity, které slouží jako hlavní předměty dotazů, opatřeny plným názvem `fullName` (tečková notace pro členění cesty ve jmenném prostoru a prefix + pro vnořené třídy). Není tedy třeba se při selekčním dotazu, kde se zadává plný název, zanořovat pro jeho doplnění. Konkrétně se jedná o entity `Namespace` a `Class`.

3.3 Návrh analyzátoru

První hlavní částí práce je tedy tvorba *analyzátoru* pro zpracování vstupního projektu. Jedná se o samostatnou spustitelnou aplikaci, která vloží analyzovaná data do databáze pro následné generování diagramů tříd.

Na úvod je třeba uvést *možnosti* zobrazitelné pomocí *UML diagramu tříd* (viz kapitola 2.2.2), které *nelze analyzovat* buď z důvodu sémantické informace, která se ztratí při implementaci, nebo z nedostupnosti informací v přeloženém projektu:

- název agregace a název role,
- asociační třída,
- vícenásobná (n-ární) asociační vazba,
- konkrétní druhy asociací (agregace a kompozice)
- a obecná násobnost $n..m$ pro $n, m \in \mathbb{N}$ a $n \leq m$.

Každá vstupní knihovna či spustitelný soubor (obecně assembly) musí být nejprve *zaveden do aplikace* tak, aby bylo možno využít reflexe (vysvětlené v kapitole 2.4.3) pro jeho analýzu. Danou část zavedení popisuje podkapitola 3.3.1. Po načtení assembly do aplikace se spustí systém jejího zpracování. Začíná se *strukturální* analýzou, která naplní celou databázi kromě vztahů mezi elementy (třídy, rozhraní apod.). Následně se provede analýza *vztahů* mezi elementy (asociace, dědičnost apod.). Obě analýzy jsou blíže vysvětleny v implementační kapitole jim určené 4.2.

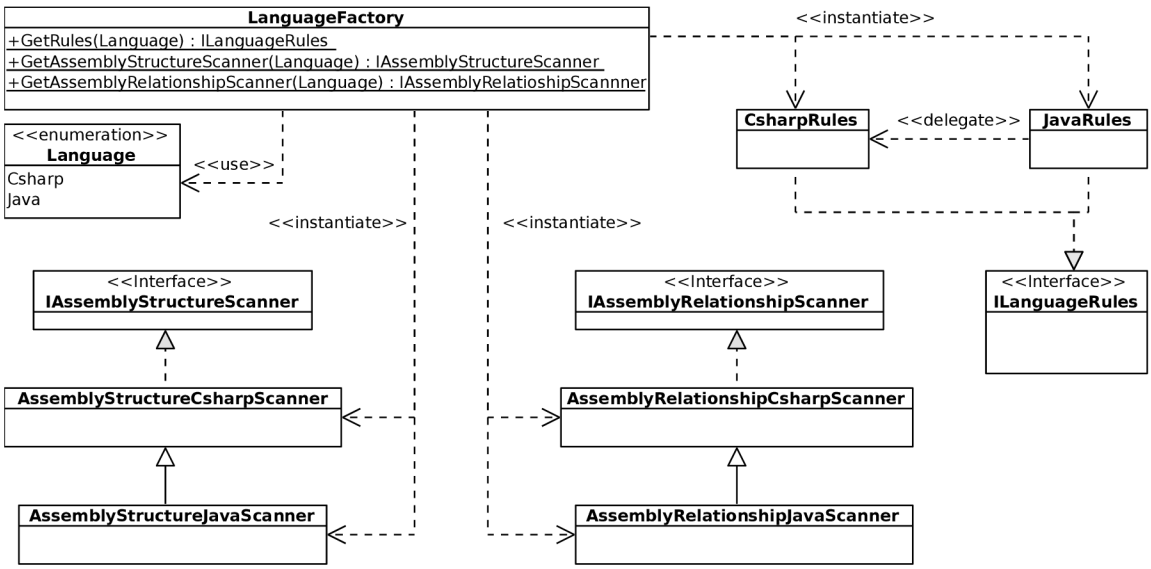
Vstupním požadavkem zadání práce bylo zpracovat jak projekty platformy C# .NET, tak i Java. V případě C# .NET lze využít reflexe, ale platforma Java je odlišná, především co se týče struktury přeloženého kódu. Vytvořit analyzátor přímo v Java určený pro její projekty je možné, jelikož analyzátory jsou odděleny od prezentačního jádra a snadno si lze v souboru s nastavením aplikace zvolit potřebný analyzátor. Časově a rozsahově by se však druhý analyzátor nedal stihnout do produkční podoby, proto je zvolena možnost převést mezikód platformy Java do .NET IL Assembly (CIL pro C# .NET) za použití již zmíněného nástroje IKVM.

Jak u podobných konverzních nástrojů bývá zvykem, neprodukují naprosto ekvivalentní výstup vzhledem k originálu. Nástroj IKVM není výjimkou, což je dáno jednak odlišnými vlastnostmi obou platforem, a jednak implementačním řešením konverze. Mezi největší rozdíly se řadí:

- Fundamentální rozdílnost *konstrukce výčtů*, kde v Java se jedná o subtyp třídy, lze tedy specifikovat výčtové hodnoty (stejně jako u C#) a společně s tím i implementovat různé operace. Proto jsou konvertované výčty zaobaleny do třídy tak, aby byl splněn onen fundament výčtu v Java.
- Java ve verzi 8 umožňuje *implementaci* metod *v* deklaraci *rozhraní*. Takové metody se nazývají výchozí (angl. „default“).
- Ztráta implicitní *informace o parametrizovatelných* třídách a metodách. V Java jsou generické typy při překladu nahrazeny přímo za patřičné typy. Ale její signaturu, vč. signatury metody, lze získat explicitně z anotace. V IKVM se jedná o anotaci typu `Signature`. Signatury v Java jsou popsány v kapitole 2.3.2.
- Rozdílná implementace *Lambda funkcí* v obou platformách. Avšak vzhledem ke konverzi obě implementace využívají totožného prostředku k jejich vnitřní reprezentaci – lokální (anonymní) třídy, i když s odlišnými názvy.
- Platforma Java nedefinuje *různé typy atributů* (prostý atribut, vlastnost, událost apod.) na rozdíl od cílové platformy C#.

Diagram tříd na obrázku 3.4 znázorňuje zohlednění výše uvedených vlastností. V návrhu se využívá návrhového vzoru *Tovární metoda*, která poskytuje řešení, jak diferencovat analýzu konkrétního zdrojového jazyku pomocí reflexe. Popisovaný diagram tříd obsahuje dva koncepty analýzy, přičemž Java analýza vždy zakládá na C# analýze:

1. *Minoritní rozdíly*, jako je rozlišení druhů tříd a vyloučení celých tříd, metod či atributů, lze zachytit v rámci jednotné analýzy pomocí skupiny pravidel v `ILanguageRules`.
2. *Ostatní*, majoritní, *rozdíly*, mezi které patří především problematika explicitního určení generických signatur, obstarávají specifické analyzátory (pro analýzu struktury i vztahů).



Obrázek 3.4: Diagram tříd pro rozlišení typů zdrojových projektů využívající návrhový vzor Tovární metoda

Jelikož se jedná o *konzolovou aplikaci*, tedy bez grafického uživatelského rozhraní, je třeba, stejně jako u ostatních mnou navržených knihoven, dát uživateli informaci o aktuálním průběhu analýzy. Proto se využívá poměrně detailní *zaznamenávání činností* jak do konzole, tak i do souboru. Pokud tedy v průběhu analýzy dojde k jakémukoliv chybovému stavu, je zaznamenán a přeskočen, čili pokračuje se v analýze následujícího prvku.

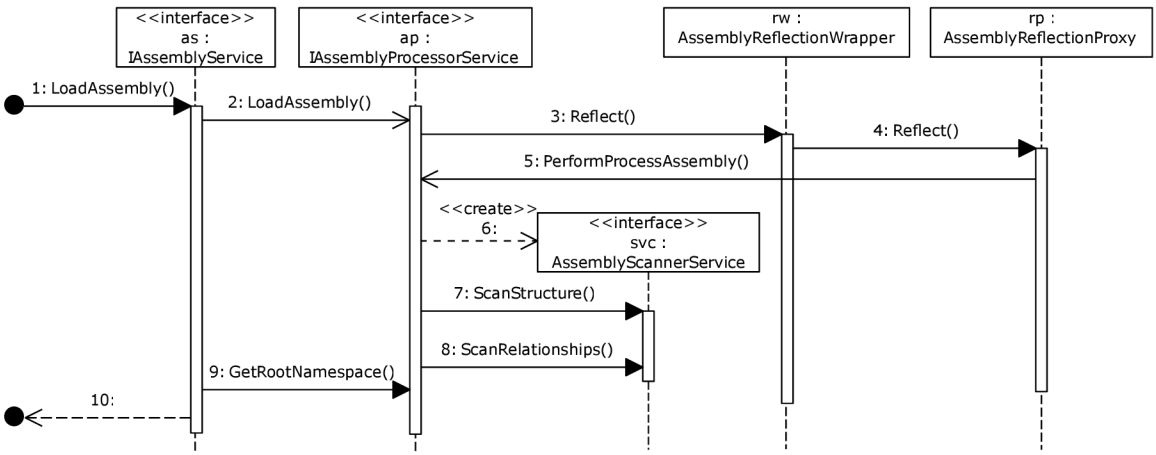
3.3.1 Načtení assembly

Pro provedení analýzy pomocí reflexe musí být vstupní assembly nejprve zavedena do aplikace. Jelikož je pravděpodobné, že vstupní projekt bude složen z více provázaných assembly, je také nutné daný předpoklad zohlednit v návrhu přístupu k jejímu zavedení.

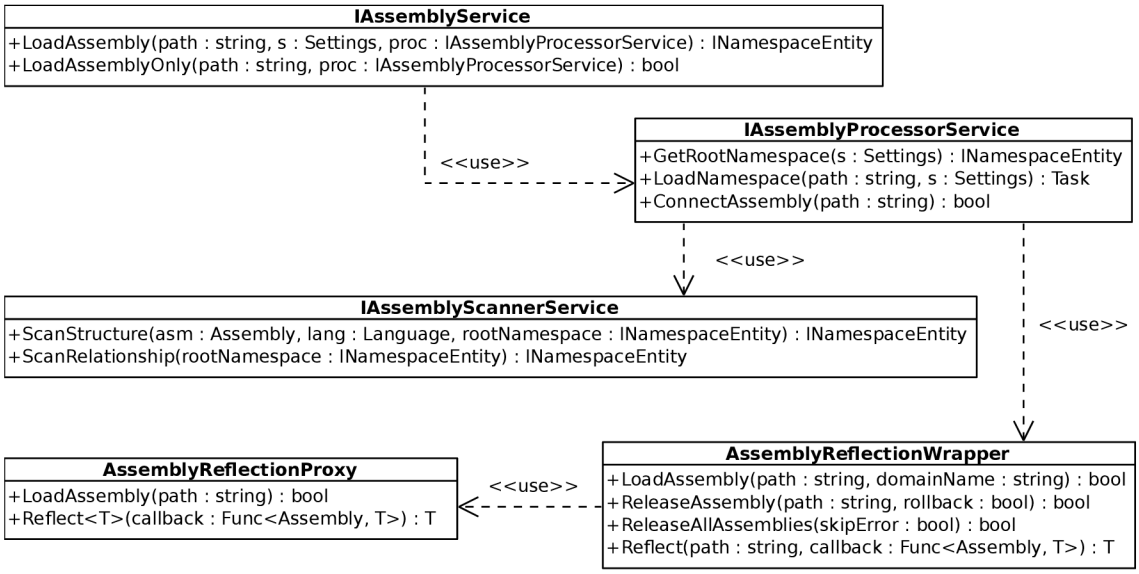
Všechny části vstupního projektu jsou postupně načteny do nového, pouze však *jednoho*, *doménového prostoru* (viz kapitola 2.4.3). Využití pouze jednoho doménového prostoru nabízí hned několik výhod. Prvně je potřeba uvést, že tvorba a udržení kontextu vyžaduje jak časové, tak i prostorové zdroje. Dále pokud by se každá část projektu zavedla do zvláštní aplikační domény, docházelo by ke zbytečné komunikaci mezi nimi z důvodu potenciálních vazeb mezi jednotlivými assembly. A v neposlední řadě postačuje udržovat pouze jedno připojení k databázi.
Lze provést dvojí načtení:

1. *úplné*, tj. včetně analýzy
2. a *neúplné*, tj. bez analýzy, které je navrženo pro zavedení pomocných knihoven nástroje IKVM.

Pro samotné zavedení assembly se využívá principu návrhového vzoru *Proxy* (obrázek 3.6), kdy se postupně proces zavedení deleguje do pomocných tříd. [26] Každá taková třída přidá kontrolu, či zaobalení do *asynchronní úlohy*, což v případě příliš dlouhého načítání ukončí činnost s chybovým hlášením skrze rozhraní `IAssemblyProcessorService`, na které je napojeno i rozhraní pro analýzu načtené assembly `IAssemblyScannerService`. Zavedení provádí dvě „Proxy“ třídy, přičemž `AssemblyReflectionWrapper` udržuje všechny zavedené assembly, pro jejich uvolnění. Po zavedení lze provést zpracování pomocí zpětného volání metody, tzv. „callback“, obstarávající analýzu struktury i vztahů. Celý popsaný proces načtení a analýzy popisuje zjednodušený (metody bez parametrů a návratových hodnot) diagram sekvence na obrázku 3.5.



Obrázek 3.5: Zjednodušený diagram sekvence pro úspěšné načtení a analyzování assembly



Obrázek 3.6: Zjednodušený diagram tříd pro načtení assembly využívající princip návrhového vzoru Proxy

3.4 Návrh webové části

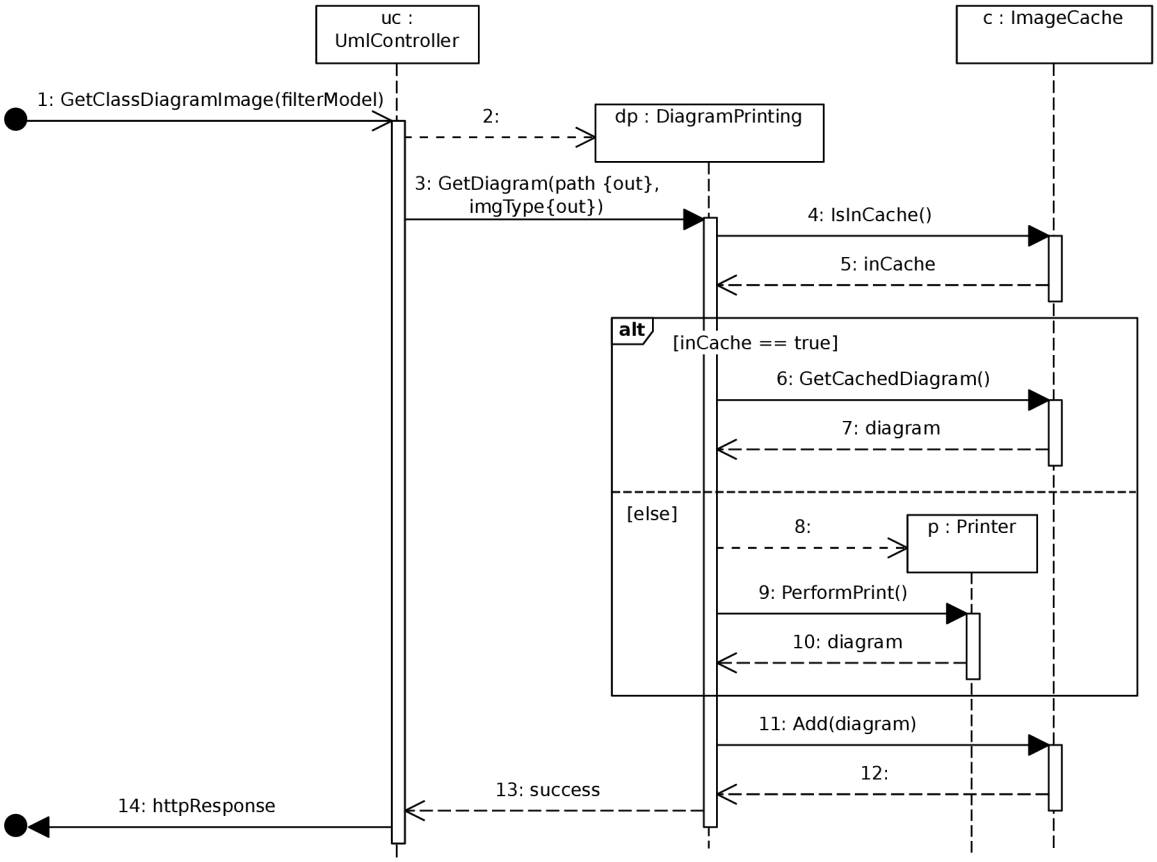
Webový server, neboli přístupový bod pro koncové uživatele, zabezpečuje jak *generování diagramů*, tak i plánování a spouštění *analýzy* zdrojového projektu. Každý takový projekt je spustitelný ve vlastním webovém prostředí. Proto je nutné mít možnost patřičně nastavit analýzu i web samotný. Pro tyto účely má administrátor k dispozici *konfigurační XML soubor*, jehož podrobnou strukturu popisuje příloha F.
Konfigurační soubor je rozdělen na tři části:

1. nastavení *analyzátoru* (popisuje kapitola 3.3),
2. nastavení *mezipaměti* obrázků (angl. „cache“),
3. *globální* nastavení.

Mezipaměti lze nastavit pracovní adresář, maximální počet obsažených souborů (velikost mezipaměti) a čas exspirace záznamu po vložení do mezipaměti. U globálního nastavení se zadává adresář pracovní, nástrojový (adresář s nástroji IKVM a Graphviz) a záznamový. Dále pak se musí nastavit typ zdrojového projektu a databázové připojení (vč. druhu SŘBD).

Celkový návrh chování severu po přijetí HTTP GET dotazu od klienta podkládá sekvenční diagram na obrázku 3.7. První fázi, a sice možnosti parametrizace dotazu a jeho zpracování, přibližuje podkapitola 3.4.1. Jakmile patřičný kontrolér obdrží validní model (zvaný `FilterModel`), vytvoří si pracovní třídu pro generování diagramů `DiagramPrinting`, na kterou deleguje požadavek na diagram. V případě, že požadovaný diagram byl již v minulosti vytvořen a je stále v mezipaměti, je použit tento (včetně aktualizace exspiračního času záznamu). Jinak dochází k předání řízení tvorby diagramu přímo knihovně `DiagrammerPrinter` (detail generování popisuje kapitola 3.5). Použitá knihovní funkce vrací příznak stavu generování z množiny $\{OK, ERROR, FATAL\_ERROR\}$, na který reaguje pracovní třída tak, že v případě OK stavu se vygenerovaný obrázek uloží do mezipaměti, pokud již je vložen, tak se

danou operací pouze aktualizuje jeho časová známka. Pro chybový stav $FATAL\_ERROR$ se vrací uživateli obecný chybový obrázek, jinak vygenerovaný chybový obrázek.



Obrázek 3.7: Zjednodušený diagram sekvence vyjadřující úspěšné zpracování požadavku uživatele na obrázek diagramu

Mezipaměť obrázků pracuje se soubory a unikátním identifikátorem pro kombinaci parametrů požadavku. Vzhledem k obecnému asynchronnímu zpracování požadavků uživatele, je třeba myslet i na *výhradní přístup* k mezipaměti, respektive přidávání a odstraňování záznamů, což implikuje použití synchronizace.

3.4.1 Dotazovací jazyk

Pro rozšíření hlavního případu užití, vygenerování kompletního diagramu tříd, a individualizaci generovaného diagramu slouží jednoduchý *dotazovací jazyk*. Tento jazyk využívá parametrizace u klasického HTTP GET dotazu. Například u následujícího dotazu `http:\\localhost:80?param1=value&param2=value` jsou parametry dva, `param1` a `param2`, s hodnotou řetězce `value`.

Požadovaný případ užití se tedy rozšířil na:

- získání úplného diagramu,
- vygenerování pouze požadovaných tříd
- a získání tříd závislých do specifikované úrovně na zdrojových třídách.

```
a.f.C          → ^a\.f\.C$
a*.f.C*        → ^a[^.]*\.f\.C[^.]*(\.[^.]*\.?)?$
a**.f.C**      → ^a.*\.f\.C[^.]*.*
a**.f.C+       → ^a.*\.f\.C$
a**.f.C+*      → ^a.*\.f\.C(\+[^+]*\+?)?$
a**.f.C+**     → ^a.*\.f\.C\+.*
a**.f.C**+F    → ^a.*\.f\.C.*\+F$
```

Tabulka 3.1: Příklad mapování zástupných znaků do regulárního výrazu pro úplný název `abc.de.f.C+F`

Společně s možností skrývat generované části (UML elementy samotné, části elementů a vztahy), vybrat si celkový styl diagramu (moderní, nebo klasický) nebo skrývat barevné části, je tak uživateli poskytnuta dostatečná variabilita při používání webové aplikace. Všechny přístupné parametry a jejich přípustné hodnoty popisuje příloha D.

Pokud uživatel zadá parametry chybně, je vygenerován chybový obrázek. Pokud se ovšem generování nezdaří, je navrácen obecný chybový obrázek informující o neplatně zadaném vstupu.

Při zadávání plných názvů tříd, tedy včetně jmenného prostoru/balíčku a nadřazených tříd v případě vnořených tříd, lze využít zástupných znaků, konkrétně znak * pro dokončení názvu třídy nebo jmenného prostoru bez obsažených elementů, a dvojice znaků ** pro všechny obsažené elementy, i zanořené. Pro oddělení názvů jmenných prostorů se využívá tečkové notace a pro vnořené třídy prefix znaku +. Příklad použití a vyhledaných elementů popisuje tabulka 3.1.

3.5 Návrh generátoru diagramů

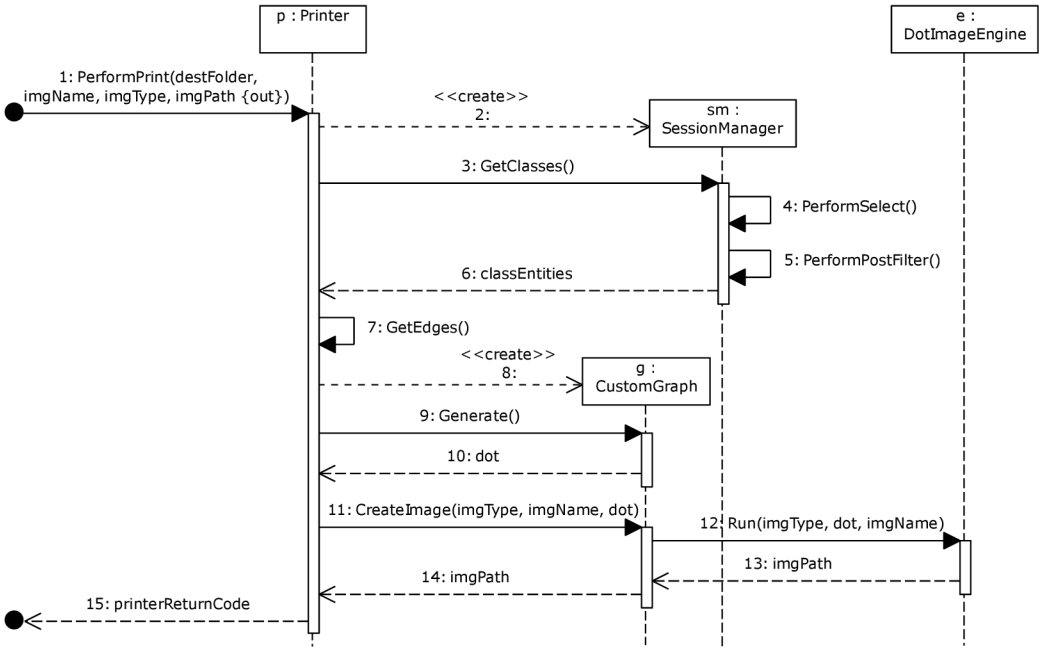
Generátor, respektive vizualizace diagramů tříd ve tvaru obrázku, obstarává vlastní knihovna `DiagrammerPrinter`. Z podstaty diagramu tříd jde o *orientovaný graf*, který tvoří *uzly* (elementy UML) a *hrany* (vztahy mezi elementy). Tato skutečnost usnadňuje návrh generátoru, neboť práce s hierarchickou strukturou, jakou je graf, je přehlednější než se surovými daty získanými z databáze.

Knihovna samotná pracuje ve třech navazujících krocích, které prezentuje i diagram sekvence 3.8:

1. *Tvorba* grafových uzlů a hran z analyzovaných entit uložených v databázi.
2. *Vygenerování* DOT[1] řetězce z grafové struktury.
3. *Vykreslení* DOT do obrázku, k čemuž se využívá nástroj `Graphviz`. Ten podporuje nepřeberné množství formátů jak rastrových, tak i vektorových. [27]

Knihovna se v každém případě snaží poskytnout žadateli obrázek. Pokud tedy dojde k chybě při tvorbě diagramu a současně se nejedná o chybu kritickou (například přímo při převodu DOT řetězce do obrázkového formátu), je daná chyba zapsána do obrázku.

[1] DOT je jednoduchý textový jazyk sloužící pro popis grafových struktur.



Obrázek 3.8: Zjednodušený diagram sekvence pro úspěšné generování obrázků diagramu tříd, který navazuje na diagram 3.7

V předchozím diagramu sekvence se vyskytuje instance třídy `SessionManager` z knihovny `DiagrammerPersistence`, která zprostředkovává přístup k analyzovaným datům, jelikož pro přenesení dotazovacího jazyku (popsaného v kapitole 3.4.1) do selekčního dotazu v databázi se využívá *systému filtrů*, kterému jsou data předána skrze skupinu nastavení (modelů) obdržených z webového kontroléru od uživatele.

Filtry jsou obsaženy ve zmíněné knihovně `DiagrammerPersistence` na základě návrhového vzoru *Stavitel*, což prezentuje diagram tříd na obrázku 3.9. Tyto filtry korelují se čtyřmi hlavními entitami reprezentující třídu, atributy třídy, její metody/operace a vztahy mezi třídami. Nastavují se v nich skryté druhy entity, případně vyjmutí celé entity z výsledku a skrytí modifikátorem viditelnosti.
Na úrovni SQL dotazu však nelze filtrovat vše, a proto se používá dvojí filtrování:

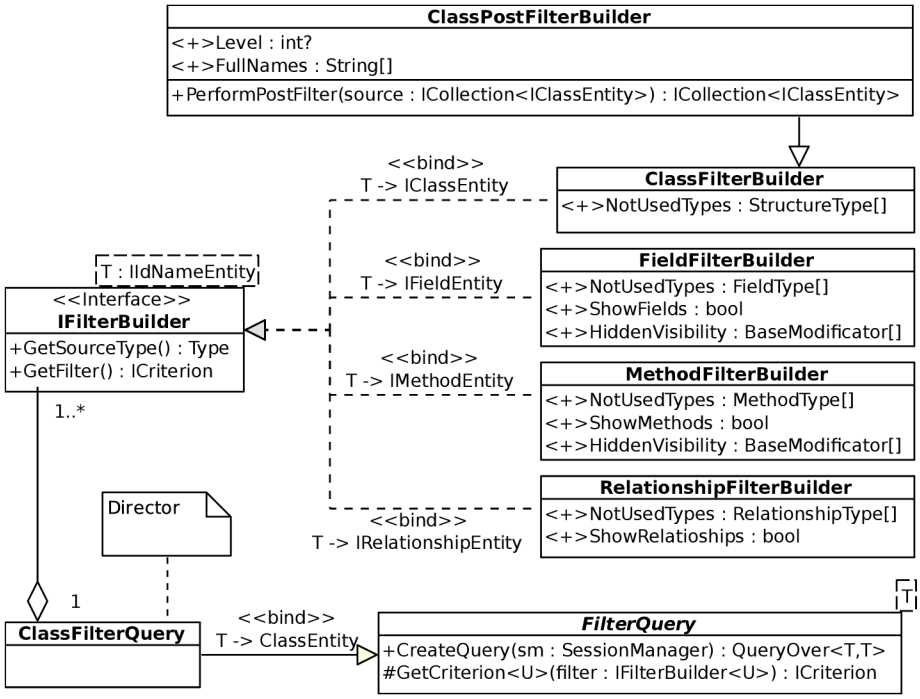
1. *přímo v SQL dotazu* pro omezení výsledku na úrovni požadovaných druhů entit,

2. a až *po provedení dotazu*, kde se filtruje název za pomocí regulárních výrazů, které skýtají více možností než SQL „LIKE“ operátor, a úroveň provázanosti elementů UML.

Hlavním přístupovým bodem pro tvorbu selekčního dotazu je metoda `CreateQuery()` ze třídy `ClassFilterQuery`. Ta vytvoří dotaz, který se předá ke zpracování v databázi.

3.5.1 Grafická podoba diagramu

Grafická reprezentace uzlů a hran, popsaná v kapitole 2.2.2, odpovídá, s mírnými odchylkami, standardu UML 2.4.1 [7] tak, aby byla zajištěna srozumitelnost diagramu. Vhledem k určitým omezením použité technologie pro generování diagramů v podobě obrázků, je nutné UML standard mírně poupravit. Jedná se o tři změny:

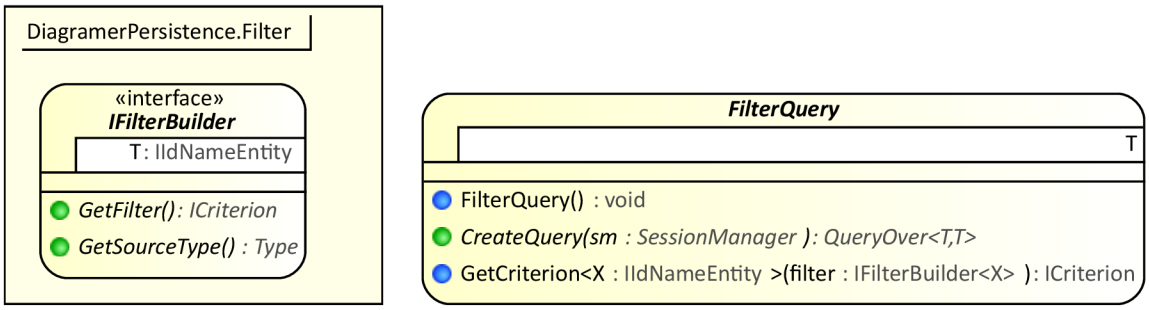


Obrázek 3.9: Zjednodušený diagram tříd pro filtr databázového dotazu využívající upravený návrhový vzor Stavitel

1. Grafická *podoba jmenných prostorů/balíků* je tvaru obdélníku s názvem balíku, respektive úplným názvem, ve vnitřním levém horním rohu. Uvedené řešení je použito v obrázku 3.10a.

2. *Umístění šablony* (generických parametrů) třídy pod název třídy, viz obrázek 3.10.

3. Rozšíření možnosti *rozlišení* specifických typů *vlastností* „property“ platformy C# .NET (atribut, prostá vlastnost, událost a indexer). Navíc pro zřetelnější rozlišení modifikátorů viditelnosti a typů prvků tříd jsem navrhl grafickou notaci, která je viditelná v tabulce 3.2.



(a) Generická třída (vč. balíku) (b) Generická metoda uvnitř generické třídy

Obrázek 3.10: Ukázka návrhu a řešení generických tříd a metod

	základ	atribut	vlastnost	událost	indexer	operace	getter	setter
`public`	+							
`protected`	#							
`internal`	~							
`private`	−							

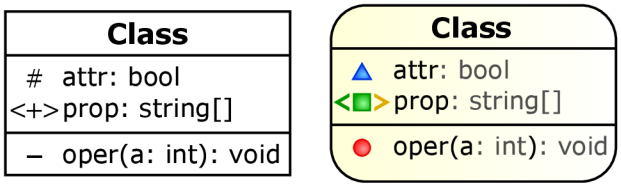
Tabulka 3.2: Vizualizace modifikátorů viditelnosti jednotlivých složek třídy

V jednotlivých třídách se obsažené prvky řadí do dvou skupin (oddělené horizontální čarou):

- *Stavy* v pořadí – prvky výčtu (pokud se jedná o výčet), atributy, vlastnosti, události a indexery[2].
- *Operace* v pořadí – konstruktory, destruktory, operace typu getter (pak jeho případný setter), operace typu setter (dosud nevypsané) a ostatní operace.

Jednotlivé podskupiny jsou seřazeny v abecedním pořadí vzestupně. Lze si vybrat zda zobrazit existenci operací typu getter a setter (přístupové metody) pro prvky typu vlastnost přímo kolem modifikátoru viditelnosti pomocí ostrých závorek (< zleva pro getter a > zprava pro setter), nebo jako prostou metodu v seznamu operací třídy.

Aplikace disponuje dvěma základními typy obecných *grafických stylů* diagramů tříd (viz obrázek 3.11), jedním *klasickým* hranatým černobílým a *moderním* zaobleným s barevným přechodem a rozlišením datových typů a méně podstatných informací od názvů pomocí šedé barvy. Tyto styly lze rozšířit použitím barevných modifikátorů viditelnosti dle uvážení uživatele, jak již bylo prezentováno výše.



(a) Jednoduchý styl (b) Moderní styl

Obrázek 3.11: Náhled základních grafických stylů generovaných diagramů tříd

[2]Pro slovo „indexer“ se mi bohužel nepodařil nalézt ekvivalentní český překlad, proto jej budu dále používat v originálním znění.

Kapitola 4

Implementace

Předposlední část vývoje aplikací tvoří část praktická, respektive implementační. Proto se v této kapitole věnuji samotné realizaci návrhu jak části o analýze zdrojového projektu (podkapitola 4.2), tak i webové, prezentační, služby (podkapitola 4.3). Webová služba ke své činnosti potřebuje generátor diagramu tříd ve tvaru obrázku, kterému se věnuje podkapitola 4.4. Obě části využívají databázový modul, jehož implementace je představena na začátku celé implementační kapitoly v podkapitole 4.1, a pomocné komponenty (např. konfigurační XML souboru a zaznamenávání činnosti aplikace), jejichž realizaci prezentuje poslední podkapitola 4.5.

4.1 Databáze

K uchování a přístupu k analyzovaným prvkům se používá *relační databáze*. Přístup k daným prvkům a databázi řeší knihovna `DiagrammerPersistence.dll`. Obecně nelze specifikovat přesný typ SŘBD, neboť ten si volí uživatel v konfiguračním souboru a vzhledem k implementačnímu prostředí objektově relačnímu mapování knihovny NHibernate 4.0, je podpora SŘBD opravdu široká (viz kapitola 3.2.3). [28] *Přístup k databázi* obstarává třída `SessionManager`, která definuje metody pro vytvoření spojení, uložení změn skrze transakci, odstranění obsahu databáze a také získání vyfiltrovaných data z databáze pomocí sady filtrů, které popisuje kapitola 3.5. Veškeré chybové informace jsou přístupné prostřednictvím vlastnosti `ErrorContent`, jelikož se v průběhu připojení mohou vyskytnout zanořené výjimky, jejichž chybový stav je komplexní, tak se dané řešení projevilo jako velice užitečné. Třídu `SessionManager` lze použít v konstrukci `using()`, což zpřehledňuje řízení připojení k databázi.

Použité *entity* odpovídají svoji strukturou tabulkám z databázového modelu na obrázku 3.3. Každý atribut entity je definován jako vlastnost s veřejnými přístupovými metodami. Všechny entity jsou skryty za rozhraním tak, aby bylo možno případně změnit implementaci entit bez výraznějšího zásahu do zbytku aplikace. Rozlišení poddruhů entit uskutečňují výčty, jejichž hodnoty jsou uvedeny v příloze E. Pro řešení rozlišení přístupových modifikátorů (viditelnost, abstrakce, konstanta apod.) se používá *příznaků* v jedné celočíselné hodnotě. Je nutné uvést, že .NET Framework rozšiřuje základní trojici modifikátorů viditelnosti o další dva: *chráněná nebo balíková* (angl. „protected or internal“) a *chráněná a současně balíková* (angl. „protected and internal“). Všechny příznaky jsou uvedeny ve výčtu `BaseModificators`.

Schéma databáze je definováno v třídách v souboru `Mapping.cs` s využitím knihovny

Fluent NHibernate, která usnadňuje objektově relační mapování entit bez použití XML souborů. [29]
Pro optimalizaci analýzy si každá entita uchovává až dva další dočasné atributy, které se však neukládají do databáze:

1. originální analyzovaný datový typ pod názvem `ReflectedType`
2. a případně i dodatečné specifické informace o tomto typu (pro metody, vlastnosti apod.) pod názvem `ReflectedInfo`.

Proto není nutné neustále dohledávat originální analyzované části v následných krocích analýzy. Celou implementaci analýzy popisuje následující kapitola 4.2.

4.1.1 Filtrování

Implementace filtrování na úrovní selekčního dotazu a i jeho výsledku se řídí návrhem provedeným v kapitole 3.5. Jelikož přístup k databázi je implementován knihovnou NHibernate, tak i selekční dotaz musí být vytvořen pro prostředí této knihovny. Splnění uvedené podmínky zabezpečuje rozhraní NHIbernate zvané `ICriterion`. Výsledný dotaz tedy tvoří skupina SQL operací „JOIN“ s potřebnou podmínkou spojení dle obdržených instancí tříd `FilterBuilder`.

Na úrovni SQL dotazu nelze zařadit/vyřadit všechny výsledky na základě všech podmínek filtrování poskytnuté uživateli, proto se používá tzv. „Post“ filtr. Tento filtr implementuje dva druhy omezení:

1. *Úplný název* (cesta) tříd – obdržené názvy s možností zástupných znaků v podobě * a ** se zkonvertují do regulárního výrazu dle pravidel popsaných kapitolou návrhu dotazovacího jazyku 3.4.1. Následně se lineárním cyklem odstraní nevyhovující třídy z obdrženého výsledku SQL dotazu. Časová složitost uvedeného řešení je lineární $O(n)$.
2. *Úroveň provázanosti* tříd (elementů UML obecně) – upravený obsah výsledného seznamu tříd, tedy po provedení filtrování dle úplného názvu, se tyto třídy použijí jako kořen stromu pro *prohledávání do šířky* s modifikací na hloubku prohledávání. Časová složitost uvedeného řešení je kvadratická $O(n^2)$.

4.2 Analyzátor

Implementace části aplikace pro analyzování, s názvem `DiagrammerAnalyzer.exe`, vstupního projektu se opírá především o návrh z kapitoly 3.3. Aby bylo možno využít metody reverzního inženýrství, zvané *reflexe*, platformy C# .NET musí se použít stejná platforma (aktuálně ve verzi 4.5.2) i pro implementaci analyzátoru.

4.2.1 Implementační detaily načtení assembly

Do aplikace lze za běhu zavést pouze assembly totožné platformy, tedy C# .NET. Tyto assembly se nacházejí v souborech typu `*.exe` a `*.dll`. V případě platformy Java je aktuálně podporován pouze archiv typu `*.jar`, který se za použití funkce `ConvertJars()` řídící třídy analyzování `Analyzer`, převede do zaveditelných assembly pomocí nástroje `ikvmc.exe` spuštěného přes příkazovou řádku. Pro korektní zavedení konvertované assembly je nezbytné do

aplikace předem načíst i knihovny nástroje IKVM ve verzi 8.0.5449.1, což obstarává pomocná třída `AnalyzerHelper`.

Vstupní soubory pro analýzu aplikace obdrží skrze konfigurační XML soubor. Kromě konkrétních souborů lze zadat i zdrojové adresáře, ze kterých se pomocí funkce C# .NET `Directory.EnumerateFiles()` rekurzivně naleznou obsažené soubory. Soubory, které si uživatel nepřeje analyzovat (specifikuje v XML uzlech konfig. souboru `<ExcludedDirectories>` a `<ExcludedFiles>`) se vyloučí. Nutno upozornit, že každý načítaný soubor musí mít dostupné všechny závislé části (knihovny), v opačném případě jej nelze načíst!

Dle návrhu z kapitoly 3.3.1 se assembly zavádějí pouze do *jednoho doménového prostoru*, který se vytvoří pomocí následujícího algoritmu (`parentDomain` je nadřazená aplikační doména):

```
private AppDomain CreateChildDomain(AppDomain parentDomain, string domainName)
{
    Evidence evidence = new Evidence(parentDomain.Evidence);
    AppDomainSetup setup = parentDomain.SetupInformation;
    return AppDomain.CreateDomain(domainName, evidence, setup);
}
```

Nově vytvořená aplikační doména, včetně zavedených assembly, se musí následně i korektně *uvolnit*. K tomuto účelu se udržuje struktura *slovníku* udržovaná v instanci třídy `AssemblyReflectionWrapper`, jehož klíčem je cesta k načtenému souboru a hodnotu zastupuje dvojice – aplikační doména a instance (v dané doméně) přístupové „Proxy“ třídy `AssemblyReflectionProxy`, na kterou se delegují operace pro práci se zavedenou assembly. Pokud se slovník vyprázdní, je doména uvolněna příkazem `AppDomain.Unload(domain)`.

Třída `AssemblyReflectionProxy` vykonává samotné zavedení vstupních assembly (včetně závislých assembly) za použití tzv. „Lazy loading“, tj. zavedení assembly až když je opravdu potřeba. Tato funkcionalita je implementována pomocí zachycení události uvnitř doménové prostoru `ReflectionOnlyAssemblyResolve` pomocí metody `OnResolve()`. Ta je optimalizována kontrolou, zda již požadovaná assembly není zavedena pomocí statické metody `ReflectionOnlyGetAssemblies()` doménového prostoru, jež vrací seznam všech zavedených assembly. Jinak se využije statická metoda `Assembly.ReflectionOnlyLoadFrom()` pro načtení ze souboru a `Assembly.ReflectionOnlyLoad()` pro načtení implicitní assembly platformy C# .NET. [26]

4.2.2 Strukturální analýza

Po zavedení assembly se spustí *strukturální analýza* se vstupním bodem, který představuje metoda `ScanStructureOfAssembly()` třídy `AssemblyStructureCsharpScanner`, která pomocí metody načtené assembly `GetTypes()` získá všechny její elementy, základní třídy `Type`, (třídy, rozhraní, struktury, delegáti a výčty), na které se následně aplikuje metoda `ScanType()`, jež zvolí pro každý specifický element metodu k analýze:

- *Výčet* – jeho rozpoznání je, stejně jako seznam hodnot, platformě specifické. V C# .NET je pro rozpoznání definována vlastnost `IsEnum` v implicitní třídě `Type`. Dále obecná metoda `GetFields()` pro získání seznamu hodnot. Rozpoznání v Java popisuje podkapitola 4.2.4[1]. Všechny platformně závislé rozpoznávání jsou implementovány pomocí rozhraní `ILanguageRules`.

[1] Pro zdroj. projekty platformy Java se výčet analyzuje jako třída.

- *Delegát* – tento element lze rozpoznat pouze pomocí ověření dědičnosti implicitní třídy `Delegate`. Obsahuje pouze jednou metodu s názvem `Invoke`.
- *Anotace* – je platformně specifická. U C# .NET ji lze poznat pomocí dědičnosti z implicitní třídy `Attribute` a v základu jde o element typu třídy. V platformně Java se jedná o rozhraní, které po konverzi dědí IKVM rozhraní `java.lang.annotation.Annotation`.
- *Rozhraní* – je rozpoznatelný pomocí vlastnosti `IsInterface` implicitní třídy `Type`. Provádí analýza pro třídu obecně pomocí metody `ScanSimpleStructure()`, jelikož může obsahovat jak metody, tak i vlastnosti a navíc může být parametrizovatelná.
- *Struktura* – musí splňovat podmínku `Type` vlastností `IsValueType && !IsEnum`. Dále se pak provede analýza jako pro třídu.
- *Třída* – se pozná pomocí vlastnosti `IsClass` a provádí se analýza popsaná níže.

Obsah třídy (vč. rozhraní, struktura, anotace i výčtu) zpracovává zmiňovaná metoda `ScanSimpleStructure()`. Nejprve se analyzují atributy včetně vlastností (jejich přístupových metod), indexerů a událostí. Výchozí hodnoty lze získat použitím metody `GetValue()` z implicitních tříd `FieldInfo`, nebo `PropertyInfo`. Poté se zpracují metody společně s konstruktory. Metody se dále dělí na destruktory, mají název `Finalize`, a operátory. Operátory, respektive jejich názvy, jsou přeloženy v .NET IL Assembly do textové podoby začínající prefixem `op_`, což je z uživatelského hlediska v diagramu nepřehledné. Z tohoto důvodu je vytvořena konverzní třída `Operator`, nebo spíše rozšířený výčet pomocí návrhového vzoru „Type-Safe Enum“ (bohužel neexistuje ekvivalentní český překlad). Tato třída například umožní převést přeložený název `op_Increment` na uživatelsky čitelnější `++`. Dále je také nutné analyzovat i parametry metod/konstruktorů, které lze získat ze zastupující třídy `MethodBase` pomocí metody `GetParameters()`, a generické parametry metody, které se analyzují obdobně jako je tomu u tříd (viz dále).

Poslední část zpracování obsahu třídy představuje *analýza generických parametrů*. Tyto parametry jsou instancemi implicitní třídy `Type`. Každý takový parametr má svůj název a může mít více obsažených datových typů, který lze získat pomocí metody třídy `Type` s názvem `GetGenericParameterConstraints()`.

Analýza každé části je vždy *strukturovaná* do menších metod tak, aby byl zdrojový kód přehledný a pochopitelný, například struktura volání metod pro analýzu atributů vypadá následovně:

```
ScanClassAttributes()
├─ ScanFields()
│  └─ ScanField()
│     └─ ScanFieldOrPropertySimple()
├─ ScanProperties()
│  └─ ScanProperty()
│     ├─ ScanFieldOrPropertySimple()
│     ├─ ScanMethod() // getter
│     └─ ScanMethod() // setter
└─ ScanEvents()
   └─ ScanEvent()
```

Jednotlivé analyzované prvky lze *přeskočit* pomocí metod začínající prefixem `Skip` v rozhraní `ILanguageRules`, pokud jsou pro UML diagram tříd nadbytečné nebo se jedná o spe-

cifické konstrukce vygenerované při překladu do .NET IL Assembly. Například se generují vnořené třídy s názvy obsahující `c__DisplayClass` (pro každý lambda výraz), `>d__0` (pro asynchronní metody) a další. Daného konceptu lze využít i pro konvertovanou platformu Java (viz podkapitola 4.2.4).

4.2.3 Analýza vazeb

Po vytvoření struktury elementů zdrojového projektu dochází k *analýze vztahů* mezi nimi, pomocí metody `ScanRelationships()` třídy `AssemblyRelationshipCsharpScanner`, která pro všechny elementy provede volání metody `ScanRelationshipsOfClass()`. Tato metoda pomocí různých implementací *delegátu* `ScanRelationship` postupně analyzuje vtahy následujících druhů postupně (pokud analýza vztahu selže, jsou další přeskočeny):

- *Kotva* (vnořené elementy) – pro získání všech vnořených, ale pouze do jedné úrovně, elementů slouží metoda `GetNestedTypes()` z implicitní třídy `Type`.
- *Zobecnění* (dědičnost) – v platformně `CsharpNet` lze dědit maximálně jednu třídu/rozhraní, pak je uvedena ve vlastnosti `BaseType` analyzovaného typu `Type`.
- *Realizace* – na rozdíl od výše popsaného vztahu dědičnosti, je možno implementovat více rozhraní. Pro jejich seznam slouží metoda `GetInterfaces()`. Pro analýzu realizací i zobecnění se používá metoda `ScanRelationshipGenericBinding()`, která uloží navázané generické parametry na parametry definované ve zdrojové třídě (v databázi jsou uloženy ve tvaru „název z nadřazené třídy/rozhraní -> název z analyzované třídy/rozhraní“ a odděleny středníkem).
- *Asociace* – pro všechny atributy daného elementu se provádí analýza asociací. Avšak rozpoznává se pouze obecná asociace, nikoliv agregace a kompozice, jelikož tyto specifické druhy souvisí se sémantikou jejich použití ve zdrojovém kódu a nelze je tak rozpoznat. Tato skutečnost by byla řešitelná za použití *explicitního označení* při tvorbě zdrojového projektu, například využít anotací. Pro samotnou analýzu se volá metoda použitá i pro obecné závislosti `ScanTypeDependecies()` s tím rozdílem, že se entita, zastupující vztah, vytváří odlišným způsobem (nastavují se násobnosti, orientovanost a zdrojový atribut).
- *Obecná závislost* – aplikuje se analýza závislostí nad generickými parametry elementu a následně závislosti související s deklarovanými metodami elementu: návratový typ, parametry metody, generické parametry metody i obsah metody samotné, pokud má definované tělo. Pro návratový typ, parametry metody a generické parametry metody i třídy se volá metoda `ScanTypeDependecies()`. Souvislosti s vyhledáváním vztahů uvnitř těla metody přibližuje podkapitola 4.2.3.

Tolik zmiňovaná metoda `ScanTypeDependecies()` pracuje ve *dvou možných režimech*:

1. Pokud se jedná o generický (parametrizovatelný) typ, využije se nepřímé rekurze s metodou `ScanTypeDependeciesOfGenericArgument()`, která získá všechny obsažené datové typy parametrizovatelného typu, jež se zpracují opět pomocí metody `ScanTypeDependecies()`.
2. Analyzovaný datový typ není generický, pak se určí, zda se jedná o již zpracovaný element strukturální analýzou a případně se vytvoří záznam o vztahu do databáze.

Vyhledávání je optimalizováno uložením úplných názvů elementů v databázi, a tak není třeba použít rekurzivní metodu jeho tvorby.

Popsaný přístup vytváří množství *duplicitních vztahů*, proto jsou před uložením do databáze seskupeny podle všech atributů entity pro vztah. Další možností byla kontrola na duplicitu před každým vložením do seznamu vztahů, avšak toto řešení je méně přehledné, od použité implicitní (tedy optimalizované) metody `Distinct()` nad seznamem po dokončení kompletní analýzy vztahů.

Analýza závislostí těla metody

Pro analýzu těla metody je definovaná metoda `ScanMethodBodyDependencies()` třídy strukturálního rozboru `AssemblyRelationshipCsharpScanner`. Tato metoda nejdříve zpracuje tělo metody pomocí třídy `MethodBodyReader` do seznamu instrukcí CIL platformy .NET IL Assembly. [30] Z tohoto seznamu jsou extrahovány *instrukce* názvu `newobj` (tvorba nové instance dané třídy), `newarr` (tvorba jedno dimenzionálního pole daná třídy), `call` (volání metody), `calli` (volání virtuální metody). [31] Nad použitým *operandem* dané instrukce je provedena analýza pomocí metody `ScanTypeDependecies()`.

4.2.4 Java specifické řešení

Již v návrhu analyzátoru (kapitola 3.3) byly představeny největší rozdíly mezi oběma platformami. Jejich řešení popisuje tato kapitola. Na úvod je třeba uvést, že převodem pomocí nástroje IKVM *nevzniká* naprosto *ekvivalentní výstup*, jako by tomu bylo u přepsání zdrojových kódů Java do platformy C# .NET. Převodem tak naopak vznikají nadbytečné konstrukce uvnitř dané třídy tak, aby byl daný kód spustitelný a choval se stejně jako u originálu.

Výčet (angl. „Enum“), po převodu do C# .NET , je prostá třída, která dědí z IKVM třídy `java.lang.Enum` a jeho prvky lze získat z vnořeného klasického C# výčtu nazvaného „`__`Enum“. Bohužel jsou však ztraceny implicitní hodnoty prvků, které jsou uloženy až uvnitř soukromého statického konstruktoru. Výčtům v platformě Java je umožněno překrýt základní metody třídy `Object` (jako je `toString()`). V konvertované třídě se navíc nachází lokální třídy s názvem tvaru čísla, statické atributy zastupující hodnoty výčtu, jejichž prefix začíná řetězcem `__<>`, prázdná metoda `__<clinit>()`, seznam všech hodnot výčtu v atributu `$VALUES` a specifické konstruktory s parametry začínající názvem `$enum$`.

Java platforma jako taková ve svém základu nepodporuje atributy typu vlastnost. Jako rozšíření je implementováno toto rozpoznávání pomocí detekce metod typu getter a setter.

Výchozí metody (v Java rozhraních označeny klíčovým slovem `default`) se po konverzi uloží do rozhraní jednak pomocí vnořené statické třídy s názvem `__DefaultMethods`, ale také metodou rozhraní nesoucí prefix `<default>`. Analyzátor takové metody pokládá za statické, tak aby byly odlišeny od ostatních metod rozhraní.

Nejvýznamnější problém představuje ztráta implicitní informace o parametrizovatelných metodách a třídách. Informace jsou však explicitně dostupné skrze anotaci `Signature` ve tvaru řetězce signatury jednak u deklarace třídy/metody, ale i při použití třídy (např. deklarace atributu). Pro účel zpracování řetězce pomocí statické analýzy (znak po znaku), která používá nabyté znalosti z kapitoly představení platformy Java 2.3.2, je vytvořena třída `JavaSignatureParser` již zaobaluje pro jednodušší použití statická třída s názvem `JavaSignatureReader`. Nalezení konkrétní reprezentace třídy ze získaného řetězce s plným názvem třídy usnadňuje optimalizace databáze, která obsahuje plný název třídy přímo

a netřeba jej složit. Uvedený přístup tak urychlil analýzu vtahů mezi třídami ve třídě `AssemblyRelationshipJavaScanner`. Strukturální analýzu platformy Java provádí třída `AssemblyStructureJavaScanner`.

4.3 Webová služba

Webová služba je postavena na základu ASP.NET MVC Framework verze 4.5.2, což je rozšíření základního ASP.NET v podobě respektující moderní přístup k tvorbě aplikací s grafickým uživatelským rozhraním, jelikož využívá architektury návrhového vzoru *Model-Pohled-Řadič* „Model-View-Controller“. Daný základ je volen s ohledem na potenciální rozšiřitelnost webové služby v oblasti grafického rozhraní.

V současné podobě je uživateli poskytnuto jednoduché rozhraní s dvěma možnými cestami zadanými skrze HTTP GET dotaz:

1. V *kořenu* nebo po zadání chybné cesty, což by vedlo k známé HTTP chybě s kódem 404, se zobrazí *základní nápověda* na použití aplikace. Mapování na kořen cesty obstarává třída `RouteConfig` a přesměrování 404 chyby je definováno v konfiguračním souboru služby `Web.config`.

2. `/DrawClass` je mapován na metodu `GetClassDiagramImage()` z třídy řadiče služby `UmlController`. Uživatel v daném případě musí navíc poskytnout i parametry pro konfiguraci výsledného *diagramu tříd* ve tvaru obrázku (dle návrhu z kapitoly 3.4.1). Tato metoda využívá služeb třídy obstarávající proces generování diagramu s názvem `DiagramPrinting`.

Výše zmíněná třída pro generování diagramu tříd používá jak knihovnu pro generování přímo určenou `DiagrammerPrinter.dll`, tak i mezipaměť výsledných diagramů pro urychlení odpovědi na uživatelův požadavek (viz podkapitola 4.3.3). V totožné třídě se také nalézá i metoda s názvem `GetErrorImage()`, která obdržený text vloží do obrázku.

Postup nasazení webové služby a celé aplikace popisuje příloha C.

4.3.1 Plánování spuštění analýzy

Webová služba, dle návrhu z kapitoly 3.1, používá analyzátor, což musí být spustitelný soubor, předaný skrze XML soubor s nastavením včetně času spuštění analýzy a periody opakování. Při spuštění aplikace se tedy *naplánuje* první spuštění analyzátoru a nastaví se *perioda* opakovaného spuštění, tj. naplánuje se vyvolání události. Pokud čas prvního spuštění již pominul, naplánuje se na následující den v požadovaný čas z konfiguračního souboru. K plánování a spouštění událostí slouží v C# .NET třídy ze jmenného prostoru `System.Threading` použité v plánovací třídě `DiagramAnalyzing`, konkrétněji:

- `Timer` – generátor opakovaných událostí přesně v naplánovanou chvíli, pomocí třídy `TimeSpan`. Ke své činnosti používá následující dvě třídy.
- `TimerCallback` – delegát vyvolaný při vzniklé události. Z implementačního hlediska jde o metodu `TimerNotification()` provádějící spuštění analyzátoru skrze příkazový řádek a inicializaci mezipaměti diagramů.
- `AutoResetEvent` – indikuje, zda je aktuální událost aktivní, či nikoliv. V podstatě se jedná o zámek.

V konečném důsledku použití daného přístupu se analyzování spustí *asynchronně* v separátní vlákně webové služby, čímž nedojde k porušení fundamentu souběžného přístupu ke službě, potažmo serveru. Vzniká zde však problém s přístupem k aktuálním datům v průběhu analýzy, protože se upravuje obsah databáze. Proto doporučuji provádět analýzu v nočních hodinách, jednak z důvodu nízkého vytížení aplikačního serveru a jednak z důvodu nízké počtu dotazů od klientů.

4.3.2 Dotazovací jazyk

Dle návrhu dotazovacího jazyku z kapitoly 3.4.1 se jednotlivé části výsledného filtru zadávají jako *parametry* HTTP GET dotazu. ASP.NET Framework poskytuje možnost navázání jednoduchých vstupních parametrů na vlastní model (v aplikaci se jedná o třídu `FilterModel`), tj. „MVC model binding“. Navržený jazyk však disponuje i komplexními typy, konkrétněji pak výčty ve tvaru řetězce a seznamy takových výčtů oddělené čárkou.

Každý takový specifický typ musí mít svoji *vlastní převodní třídu*, respektive datový typ, s podstatnou anotací `ModelBinder`, jejíž parametrem je obalující třída převodu s názvem `TypeModelBinder`. Vstupem konverze je vždy řetězec, který obdrží daná konverzní třída požadovaného atributu skrze implementovanou metodu `Parse()` z rozhraní `IModelType`. Uvedené rozhraní implementují všechny konverzní třídy. Případné předpokládané výčtové hodnoty ve tvaru řetězce se do dané hodnoty převádějí pomocí C# konstrukce `Enum.TryParse()`.

V přídě jakékoliv chyby při konverzi se přeruší zpracování zbylých parametrů a je vyvolána speciální výjimka buď typu `BindingTypeException` (chybný vstupní formát dat), nebo `MandatoryAttributeException` (nezadán povinný parametr). Vyvolaná výjimka je zachycena na globální úrovni aplikace a uživatel je informován o svém prohřešku skrze vygenerovaný obrázek s chybovou zprávou. Pokud nastane chyba při generování, je vrácen obecný chybový obrázek.

4.3.3 Mezipaměť

Mezipaměť obrázků, neboli známější „cache“ paměť, slouží jako druh kruhového seznamu o předem dané velikosti k uchování úspěšně vygenerovaných obrázků diagramů tříd. Pro mezipaměť na úrovni platformy C# .NET již existuje řešení v podobě struktury udržované v paměti s názvem třídy `MemoryCache`. Pro naplnění záznamu v mezipaměti je třeba znát unikátní klíč, dále potřebnou hodnotu a čas exspirace záznamu v mezipaměti. Klíč představuje „hash kód“ vstupní konfigurace parametrů dotazu, tedy celé nezáporné unikátní číslo pro danou kombinaci parametrů. Uložený záznam představuje třída `ImageCacheRecord` obsahující *cestu* k obrázkovému souboru na disku a formát obrázku (png, jpeg, svg apod.), který slouží k nastavení *formátu dat* v hlavičce HTTP odpovědi.

Třída zastupující mezipaměť v aplikaci nese název `ImageCache`. Její metody umožňují přidat, aktualizovat nebo odebrat potřebné záznamy. V případě, že je mezipaměť naplněna a přišla žádost o přidání záznamu, tak se odstraní právě jeden záznam dle algoritmu *LRU* (angl. „Least recently used“) tj. algoritmu, který vyřadí nejdéle nepoužívaný záznam. [32] Každý odebraný záznam ze struktury odebere i navázaný soubor obrázku na disku.

Úprava nebo získání záznamu z mezipaměti může být asynchronní, jelikož dotazy na server přicházejí také asynchronně. Proto je zapotřebí ošetřit souběžný přístup pomocí synchronizačního primitiva – *zámku*. Kritická sekce se pomocí blokového příkazu jazyku C# `lock` označí daná kritická sekce.

4.4 Generátor obrázků

Hlavním přístupovým bodem pro práci s *generátorem diagramů tříd* ve tvaru obrázku je třída s názvem `Printer`. Ostatní obsah implementační knihovny `DiagrammerPrinter.dll` zůstává skrytý pro použití mimo knihovnu. Jedinou výjimku tvoří skupina tříd pro nastavení výstupu generátoru umístěných ve jmenném prostoru `DiagramerPrinter.Graph.Settings`.

Pro implementaci samotného procesu generování diagramů se využívají dvě externí knihovny zamýšlené pro podporu vizualizačního nástroje Graphviz 2.29 [27]:

1. `QuickGraph` pro zaobalení grafových uzlů a hran z používaných entit a tvorbu DOT řetězce. [33] Tato knihovna se bohužel od roku 2012 nadále neaktualizuje, a proto je upravena pro podporu nových možností nástroje Graphviz, především v oblasti HTML struktury uzlu grafu, viz dále.

2. `GraphVizWrapper` pro zaobalení přímého volání nástroje Graphviz skrze příkazovou řádku ve třídě `DotImageEngine` implementující rozhraní `IDotEngine` knihovny QuickGraph, což propojuje obě dvě zmíněné knihovny. GraphVizWrapper se udržuje aktualizovaný na webové službě GitHub [34], avšak úpravy jsou provedeny i u ní v podobě rozšíření možnosti spuštění nástroje Graphviz ve specifickém adresáři. Všechny změny jsou zaznamenány i s požadavkem autorovi na vložení do hlavní vývojové větve.

4.4.1 Tvorba DOT řetězce

Jak již bylo zmíněno v návrhové kapitole 3.5, diagram tříd je orientovaný graf s uzly jsou dvojího druhu (oba pouze zaobalují ekvivalentní entity): *uzel tříd* zastupuje třída `ClassVertex` (případně struktury, výčtu apod.) a *uzel jmenného prostoru* zase třída `ClassNamespace`. Uzel jmenného prostoru se aktuálně nevyužívá a je zde implementován pro možné rozšíření v podobě generování diagramu balíků. Uzly jsou propojeny hranami, které se liší dle daného typu vztahu (dědičnost, asociace apod.) a zaobaluje je třída `GraphEdge`.

Tvorbu jednotlivých uzlů a hran řídí třída grafu `DiagramGraph`, kde se po zavolání funkce `Generate()` nejprve namapují uzly a hrany na reálné databázové entity a následně se zaregistrují události pro tvorbu grafu, respektive *DOT řetězce*. Poté se aktivuje generování. Události se aktivují jak pro uzly, kdy dochází k vytvoření jeho jmenovky (angl. „label“), tak i pro hrany, kdy se nastaví potřebné atributy (např. jmenovky, typ čáry hrany, typ šipky na koncích hrany apod.).

Uzel

Zaznamenání uzlu do grafu, respektive jeho grafické podoby, se vytváří na základě tabulky z jazyku HTML, kterou nástroj Graphviz podporuje ve zjednodušené podobě. Jedná se prakticky o jediný způsob, jak si vytvořit vlastní grafové struktury v dané technologii. Proces tvorby HTML řetězce obstarává třída `DotVertexBuilder`, kdy po zavolání metody `GetVertexLabel(classEntity)` s jediným atributem představující entitu třídy z databáze se vygeneruje postupným skládáním fragmentů HTML tabulky výsledný řetězec. Výsledný obsah elementu a jeho grafickou podobu lze parametrizovat pomocí skupiny tříd s nastavením zmíněné na začátku kapitoly.

Implementace nástroje Graphviz a jazyku DOT umožňuje seskupovat uzly do skupin s vlastním názvem, čehož se využívá při grafické notaci balíků v UML diagramu tříd. V případě stylu zobrazení balíků vnořeně je navíc hierarchie vnoření odlišena střídáním světlejší a tmavší barvy pozadí skupiny.

Hrana

Pro kvalitnější grafické ztvárnění propojení uzlů (HTML tabulky) s hranami tak, aby konce čar navazovaly na uzly, nástroj Graphviz definuje atribut tabulky s názvem `PORT`. Tím se dostáváme k tvorbě hran mezi uzly, kterou především obstarává třída `DotEdgeBuilder` s veřejně přístupnými metodami pro získání jmenovky (angl. „label“), tvarů hrany vč. symbolů na koncích hrany a jmenovky násobností umístěných u vstupů a výstupů hrany. Z principu se tedy jedná o rozdílný přístup oproti tvorbě uzlů, neboť dle implementace nástroje Graphviz se tato struktura uživatelská struktura uzlu jako jeho jmenovka.

Propojení uzlů hranami generuje knihovna QuickGraph automaticky na základě vytvořené grafové struktury.

Omezení výsledného diagramu

Dále pak z omezení použité technologie a univerzálnosti jazyku UML plynou, mimo konceptuálních třech změn z kapitoly 3.5.1, následující úpravy:

- Vztah „kotva“, který v UML slouží pro zaznamenání vnořování a značí se kružnicí se znaménkem plus $\oplus$, nelze v Graphviz nalézt, proto je nahrazen prázdnou kružnicí $\bigcirc$.
- Pro rozšířený modifikátor viditelnosti chráněná nebo balíková, respektive chráněná a současně balíková, se využívá totožného symbolu jako u viditelnosti balíkové, respektive chráněné. Uvedené zjednodušení je dáno jednak podporou pouze základních viditelností v UML a jednak nejvíce omezujícím přístupem k danému atributu.
- Problém také představuje znázornění variabilního seznamu parametrů metody. Programovací jazyk Java jej znázorňuje trojicí teček za názvem atributu (např. `names...`), kdežto C# definuje klíčové slovo `params`. Z uvedeného plyne, že znázornění variabilního seznamu parametrů je závislé na použitém implementačním jazyku. Proto kompromisní řešení reprezentuje omezení `{params}` uvedené za daným parametrem metody.

4.5 Pomocné komponenty

Každá větší konzolová aplikace potřebuje informovat uživatele a uchovávat informace o své činnosti. Není tomu jinak v případě popisovaných řešení analyzátoru, webové služby a jejich knihoven. Proto se využívá služeb *zaznamenávání činnosti* (angl. „logging“) prostřednictvím třídy `System.Diagnostics.Trace` ze C# .NET Framework, jejíž použití je zaobaleno do uživatelsky použitelnější třídy `Logger` s metodami pro záznam zpráv v různých úrovních důležitosti. Při použití zaznamenávání činnosti se musí respektovat následující kroky:

1. Přesměrování zaznamenávání do souboru (implicitně naslouchá pouze příkazová řádka):

```
Logger.AddLoggerListener(logFilePath, listenerName);
```

2. Vytvoření instance třídy `Logger` pro vkládání záznamů:

```
private static readonly Logger Log = Logger.GetLogger(typeof(sourceClass));
```

3. Vložení záznamu s možností formátování řetězce (metody `Debug`, `Info`, `Warning` a `Error`):

```
Log.Info("Text {0}", "param");
```

Jelikož se u analyzátoru využívá *rozdílná aplikační doména*, které si vytváří svůj vlastní kontext vč. zaznamenávání činnosti, což v důsledku znamená, že zapisované záznamy jsou duplikované, je nutné, zajistit zaznamenávání činnosti skrze více domén. K tomuto popsanému účelu je vytvořena třída `CrossDomainTraceHelper` s veřejnou statickou registrační metodou `StartListening(appDomain)`, které se předá aplikační doména, jejíž záznamová činnost má být delegována do hlavní aplikační domény. Celé funkcionalita je implementována v podpůrné knihovně `DiagrammerUtils.dll`.

Dále pak obě části aplikace (analyzátor i webová služba) potřebují ke své činnosti *konfigurační XML soubor* popsaný v kapitole 3.4 a v příloze F. Knihovna `DiagrammerUtils.dll` implementuje načtení a uchování konfigurace v programu pomocí třídy `Settings` a dvou obsažených tříd nastavení pro analyzování `AnalyzeSettings` a mezipaměť `CacheSettings`. Veškerý obsah XML souboru se pak pomocí konverze „deserializace“ převede do objektové podoby. Pro konverzi se každému atributu musí nastavit patřičná anotace XML elementu ze jmenného prostoru `System.Xml.Serialization` platformy C# .NET.

Kapitola 5

Testování a výkonnost

Fáze testování výsledného produktu probíhala skrze celý vývojový proces analyzátoru, který je popsán v podkapitole . Testování vrcholilo při konečném ladění, kdy se prováděly testy výsledného řešení, jak líčí podkapitola 5.2. Z obou fází testování vzešlo i několik optimalizací.

5.1 Testy analyzátoru

Pro testování a především ladění analyzátoru byl zvolen *poloautomatický postup*, který kombinuje automatické testování s manuálním. Pro tyto účely je připravena jedna sada testů pro každou platformu zdrojových projektů. Pro každou sadu byl předpřipraven obsah databáze v MSSQL. Vývojové studio od společnosti Microsoft určené pro vývoj aplikací na základě platformy C# .NET s názvem Visual Studio, jež mimo jiné dokáže porovnat dvě MSSQL databáze automaticky porovnat na obsah, jen se musí manuálně spustit. Lze také porovnat obsah souborů s protokolem o analýze.

Sada testů pro projekty platformy C# .NET tvoří testovací projekt obsahující analyzované konstrukce tak, aby byly pokryty pokud možno všechny možné scénáře relevantní pro diagram tříd. Kromě známých konstrukcí dané platformy jsou přidány i ty méně známé, mezi které se řadí:

- *Direktiva* `using` pro tvorbu zástupných symbolů, tzv. „alias“. Například symbol `int` zastupuje třídu `System.Int32`.
- *Částečné definice* jedné třídy, tzv. „partial class“, kdy jedná třída je rozdělena do více zdrojových souborů.
- *Modifikátor* `new` použitý při deklaraci metod nebo atributů, který umožní předefinovat daný zděděný prvek bez nutnosti ho označit v rodičovské třídě jako `virtual`, tedy jako přepsatelnou.
- Možnost definovat *více dimenzionální pole* pomocí syntaxe `[,,]`, což značí tří dimenzionální pole `[][][]`.
- Přetížení *operátorů*, například operátor sčítání +, bitový operátor & apod.
- Definice vlastních *anotací*.

Testy pro projekty platformy Java jsou také tvořeny testovacím projektem pokrývající ekvivalentní konstrukce popsané výše. Dané testy jsou rozšířeny o různé verze Java (od

verze 6 a výše) totožného projektu. U Java verze 8 byla přidána nová konstrukce, a sice výchozí metody (angl. „default“), které představují možnost implementovat metodu přímo v rozhraní.

Při optimalizaci struktury zdrojových kódů analyzátoru, tzv. „refaktorování“, nebo při úpravě funkcionality, jsou tyto jednoduché vstupní projekty vhodné pro i regresní testování.

Výsledky popsaného přístupu testování dopomohly k výsledné funkčnosti analyzátoru jak z pohledu *optimalizace rychlosti* analyzování, tak i z pohledu *přidaných rozšíření*, mezi které se řadí především podpora pro zanesení informací o definovaných anotacích. Rychlost analýzy je bohužel z velké části (téměř třetina) ovlivněna zavedením vstupních assembly za běhu a přístupem k databázi, a to i po optimalizaci v podobě využití pouze jedné nové aplikační domény. Dále pak se poměrně vysoké procento zkonzumuje výpisem nastalých událostí do souboru, jedná se v průměru o 4%. Výsledky testů[1] přehledně představuje tabulka 5.1.

Zdroj dat	**Počet souborů**	**Počet řádků kódu**	**Čas před optimalizacemi**	**Čas po optimalizacích**	**Zlepšení**
Tato práce	5	~ 4200	$7,8s$	$6,1s$	28%
C# .NET test	1	~ 90	$4.9s$	$3.9s$	26%
Java test[2]	1	~ 320	$7,9s$	$6,4s$	24%

Tabulka 5.1: Tabulka s výsledky testování analyzátoru pro tři různé projekty před i po provedení optimalizací v rámci zdrojového kódu i konceptu analýzy. Tabulka prezentuje i prokazatelné zrychlení v průměru o 26%

5.2 Testy celku

Ověřování výsledné funkčnosti probíhalo pouze manuálním způsobem, vzhledem k tomu, že výstupem aplikace jsou obrázková data diagramu, jehož elementy mohou měnit svoji pozici při každém novém vygenerování obsahu. Testy výsledné aplikace, společně s kontrolou splnění požadavků, probíhalo souběžně s konzultanty zákazníka, společnosti Siemens, s. r. o., oddělení Corporate Technology Brno.

Testování bylo v této části zaměřeno na korektní provázání všech knihoven, dostupnost služby a konverzi parametrů pro filtrování výsledného diagramu, ale především se věnovalo generování UML diagramu tříd z databáze s analyzovanými informacemi a jeho odeslání ve tvaru obrázku.

Při provádění verifikace a testů výkonnosti se projevily tři problémy (omezení), které dokládá tabulka 5.2, potažmo i 5.3:

1. Složitost vstupního dotazu, tedy počet ovlivněných prvků v databázi, respektive jejich navrácený počet nebo složitost databázového selekčního dotazu samotného, ovlivňuje výslednou rychlost více jak z 50%. S počtem obdržených prvků z databáze pro zpracování koreluje i rychlost generování obrázku pomocí nástroje Graphviz. Řešení na úrovni implementace představuje použití mezipaměti pro již vygenerované obrázky. Optimalizace přímo pro generování nebyla nalezena a měla by být předmětem dal-

[1] Testy a profilování probíhaly na PC s konfigurací CPU Intel i5 4570 s frekvencí 3.20GHz, 2x4GB DDR3 RAM a SSD 480GB v operačním systému Windows 8.1 v programu JetBrains dotTrace.

[2] Test Java platformy je ovlivněn nutností načíst potřebné knihovny nástroje IKVM a potřebo konverze do platformy C# .NET.

šího vývoje aplikace. Ve výsledku je třeba s rychlostí prvního generování počítat při používání nástroje.

2. Rychlost konverze DOT řetězce do obrázku za použití nástroje Graphviz je závislá na jeho formátu i složitosti DOT řetězce. Obecně jsou obrázky rastrového formátu generovány pomaleji než vektorové a s větší složitostí DOT řetězce se tento rozdíl čím dál více zvětšuje.

3. Aktuální hardwarová konfigurace a vytížení aplikačního serveru může i několikanásobně zpomalit rychlost tvorby diagramu.

Konfigurace	**Připojení k DB**	**Filtrování**	**Generování DOT**	**Generování obrázku**	**Čas**
Úplný diagram tříd	$0,5\%$	$44,5\%$	$3,0\%$	$50,3\%$	$19,1s$
Diagram tříd z obr. A.1	$24,7\%$	$58,9\%$	$2,3\%$	$7,1\%$	$0,9s$
Diagram tříd z obr. A.2	$12,7\%$	$50,0\%$	$0,5\%$	$4,8\%$	$1,0s$
Diagram tříd z obr. A.3	$1,0\%$	$80,1\%$	$0,1\%$	$2,9\%$	$14,6s$
Diagram tříd z obr. A.4	$40,0\%$	$24,0\%$	$2,0\%$	$10,0\%$	$0,2s$

Tabulka 5.2: Tabulka s výsledky rozložení časové náročnosti u testování vygenerování diagramu v `*.svg` formátu na lokálním serveru z webového prohlížeče. Zbylé procentní body zastupují ovládací logika. Lokální server představuje lokální testovací stroj (viz tabulka 5.1)

Konfigurace	**Čas na serveru pro `*.svg`**		**Čas na serveru pro `*.png`**	
	lokální	**vzdálený**	**lokální**	**vzdálený**
Úplný diagram tříd	$19,1s$	$31,0s$	$34,1s$	$59,6s$
Diagram tříd z obr. A.1	$0,9s$	$1,3s$	$1,3s$	$1,7s$
Diagram tříd z obr. A.2	$1,0s$	$1,1s$	$1,2s$	$1,3s$
Diagram tříd z obr. A.3	$14,6s$	$22,1s$	$14,8s$	$22,3s$
Diagram tříd z obr. A.4	$0,2s$	$0,3s$	$0,5s$	$0,5s$

Tabulka 5.3: Tabulka s výsledky testování průměrné rychlosti vygenerování diagramu různých formátů na různých serverech z webového prohlížeče. Lokální server představuje lokální testovací stroj (viz tabulka 5.1) a vzdálený server zastupuje aplikace nasazená na http://ac-diagrammer.aspone.cz

U zákazníka se aplikace nasadila na dva rozsáhlé projekty (platforma Java i C# .NET) zaměřené na správu dopravy. Z obou zdrojových projektů se však analyzují jen vybrané moduly jednak z důvodu omezení se jen na potřebné třídy, a jednak pro snížení celkové velikost databáze, což vede k urychlení generování. Získané diagramy jsou následně využity v online dokumentačním nástroji Trac[3].

[3]Trac dokumentační nástroj je dostupný na domovské stránce http://trac.edgewall.org.

Kapitola 6

Závěr

V rámci mé diplomové práce byla představena aplikace pro automatické generování diagramu tříd pomocí unifikovaného modelovacího jazyka. Jedná se o webovou službu přístupnou vzdáleným klientům, kteří si mohou na základě HTTP dotazů získat obrázky s požadovaným diagramem tříd. Jedná se tak o jeden z možných způsobů optimalizace udržení konzistentních informací o odvedené práci v procesu vývoje softwarového produktu, v daném případě aplikací založených na objektově orientovaném paradigmatu.

Analyzován byl unifikovaný modelovací jazyk, zvaný UML, a především pak diagram tříd a jeho části včetně vztahů. Také byly popsány podporované platformy zdrojových projektů C# .NET a Java včetně možností přístupu k již přeloženým souborům za použití metody reverzního inženýrství, zvané reflexe. Nabyté informace jsou využity v návrhu i implementaci výsledného řešení aplikace. Již v konceptuálním návrhu byl kladen důraz na využívání libovolného analyzátoru (spustitelný soubor) zdrojových projektů, který dokáže korektně naplnit databázi analyzovaných informací potřebných pro vygenerování diagramu tříd.

Výsledná aplikace byla představena i v příspěvku ve sborníku Excel@FIT. [35] Současná podoba webové služby je také nasazena v omezeném prezentačním režimu, tj. bez neustálé aktualizace zdrojových informací, dostupné na adrese http://ac-diagrammer.aspone.cz. Řešení úspěšně pracuje ve společnosti Siemens, s. r. o., oddělení Corporate Technology Brno.

6.1 Budoucí možnosti rozšíření

Možnosti rozšíření aktuální podoby aplikace jsou poměrně široké a lze je rozdělit do dvou skupin – menší vylepšení (vč. urychlení) a obsáhlejší rozšíření funkcionality.

Do první skupiny rozšíření se řadí přidání podpory pro *specifické druhy archivů* platformy Java , například `*.war` (soubor webového modulu), `*.ear` (soubor obsahující více modulů), nebo `*.apk` (soubor aplikace Android platformy). Dále pak je možno zařadit i větší optimalizaci rychlosti analýzy i generování diagramů samotných.
Druhá skupina, s rozsáhlejší analýzu a implementaci, obsahuje tři potencionální rozšíření:

1. Generování *diagramu balíků* pomocí unifikovaného modelovacího jazyka.
2. Vyhledávání případně vyznačení *závislostí mezi* zadanými *třídami* (UML elementy).
3. Identifikace a zvýraznění softwarových *návrhových vzorů* v diagramu tříd.

Samozřejmě by bylo možné uvedený seznam rozšířit o další různé analýzy architektury vstupního projektu, dle požadavků zákazníka.

Literatura

[1] BECK, Kent, aj. *Manifest Agilního vývoje software* [online]. ©2001 [cit. 2015-05-15]. Dostupné z: http://agilemanifesto.org/iso/cs

[2] EILAM, Eldad. *Reversing: Secrets of Reverse Engineering*. Indianapolis: Wiley Publishing, 2005. ISBN 07-645-7481-7. Dostupné z: http://www.ece.ualberta.ca/~marcin/aikonsoft/reverse.pdf

[3] LANZA, Michele. *Object-Oriented Reverse Engineering*. Bern, 2013. Doktorská práce. Universität Bern, Philosophisch-naturwissenschaftliche Fakultät, Institut fur Informatik und angewandte Mathematik. Vedoucí práce prof. Dr. S. Ducasse, prof. Dr. O. Nierstrasz. Dostupné z: http://scg.unibe.ch/archive/phd/lanza-phd.pdf.

[4] FOWLER, Martin. *UML distilled: Brief Guide to the Standard Object Modeling Language*. 3. vyd. Boston: Addison-Wesley, 2003. ISBN 978-032-1193-681.

[5] ARLOW, Jim a NEUSTADT, Ila. *UML a unifikovaný proces vývoje aplikací: průvodce analýzou a návrhem objektově orientovaného softwaru*. 1. vyd. Brno: Computer Press, 2003. ISBN 80-722-6947-X.

[6] SCHMULLER, Joseph. *Myslíme v jazyku UML*. 1. vyd. Praha: Grada, 2001. ISBN 80-247-0029-8.

[7] UML 2.4.1. *OMG Unified Modeling LanguageTM (OMG UML): Superstructure*. Needham: Object Management Group, 2011. Dostupné z: http://www.omg.org/spec/UML/2.4.1/Superstructure

[8] ORACLE. *Oracle Java Help Center: Java Documentation* [online]. ©2014 [cit. 2014-12-28]. Dostupné z: http://docs.oracle.com/en/java

[9] LINDHOLM, Tim, YELLIN, Frank, BRACHA, Gilad a BUCKLEY, Alex. *The Java Virtual Machine Specification: Java SE 8 Edition*. Redwood City: Addison-Wesley Professional, 2014. ISBN 978-013-3905-908.

[10] ENGEL, Joshua. *Programming for the Java Virtual Machine*. 2. vyd. Reading: Addison-Wesley Professional, 1999. ISBN 02-013-0972-6.

[11] MATYSKA, Luděk. Virtualizace výpočetního prostředí. In: *Zpravodaj ÚVT MU Bulletin pro zájemce o výpočetní techniku na Masarykově univerzitě*. Brno: Masarykova univerzita v Brně, 2006, s. 9-11. roč. XVII, č. 2. ISSN 1212-0901. Dostupné z: http://ics.muni.cz/bulletin/clanky_tisk/540.pdf

[12] TIŠNOVSKÝ, Pavel. Pohled pod kapotu JVM: 1. část - prohlížení a modifikace bajtkódu. In: *Root.cz* [online]. 2011 [cit. 2014-12-27]. Dostupné z: http://www.root.cz/clanky/pohled-pod-kapotu-jvm-1-cast-prohlizeni-a-modifikace-bajtkodu

[13] TIŠNOVSKÝ, Pavel. Pohled pod kapotu JVM: 2. část - podrobnejší analýza obsahu constant pool. In: *Root.cz* [online]. 2011 [cit. 2014-12-30]. Dostupné z: http://www.root.cz/clanky/pohled-pod-kapotu-jvm-2-castpodrobnejsi-analyza-obsahu-constant-poolu

[14] ECMA-334. *C# Language Specification*. Geneva: Ecma International, 2006. Dostupné z: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf

[15] MICROSOFT. Overview of the .NET Framework. In: *Microsoft Developer Network* [online]. ©2014 [cit. 2014-12-30]. Dostupné z: http://msdn.microsoft.com/en-us/library/zw4w595w.aspx

[16] ALBAHARI, Joseph, ALBAHARI, Ben a DRAYTON, Peter. *C# 5.0 in a Nutshell*. 5. vyd. Sebastopol: O'Reilly Media, 2012. ISBN 978-144-9320-102.

[17] ECMA-335. *Common Language Infrastructure (CLI)*. Geneva: Ecma International, 2012. Dostupné z: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf

[18] LIDIN, Serge. *.NET IL Assembler*. Berkeley: Apress, 2014. ISBN 15-905-9646-3.

[19] SPARX SYSTEMS. *UML tools for software development and modelling: Enterprise Architect UML modeling tool* [online]. ©2000 - 2014, 2014-09-10 [cit. 2014-12-30]. Dostupné z: http://www.sparxsystems.com

[20] VISUAL PARADIGM. *Software Design Tools for Agile Teams, with UML, BPMN and More* [online]. ©1999 - 2015, 2014-12-15 [cit. 2014-12-30]. Dostupné z: http://www.visual-paradigm.com

[21] TIGRIS.ORG. *ArgoUML* [online]. ©2001 - 2009, 2011-12-15 [cit. 2014-12-30]. Dostupné z: http://argouml.tigris.org

[22] TIHANYI, Balazs. SOURCEFORGE. *NClass: Free UML Class Designer* [online]. 2006, 2011-06-06 [cit. 2014-12-30]. Dostupné z: http://nclass.sourceforge.net

[23] FRIJTERS, Jeroen. *IKVM.NET* [online]. ©2002 - 2011, 2014-06-29 [cit. 2015-01-11]. Dostupné z: http://www.ikvm.net

[24] HADJIGEORGIOU, Christoforos. *RDBMS vs NoSQL: Performance and Scaling Comparison*. Edinburgh, UK, 2013. Dostupné z: http://static.ph.ed.ac.uk/dissertations/hpc-msc/2012-2013/RDBMS%20vs%20NoSQL%20-%20Performance%20and%20Scaling%20Comparison.pdf. Disertační práce. The University of Edinburgh.

[25] VICKNAIR, Chad, MACIAS, Michael, ZHAO, Zhendong, NAN, Xiaofei, CHEN, Yixin a WILKINS, Dawn. A comparison of a graph database and a relational database: A Data Provenance Perspective. In: *Proceedings of the 48th Annual*

Southeast Regional Conference on - ACM SE '10. New York, USA: ACM, 2010, 42:1–42:6. ISBN 978-1-4503-0064-3. Dostupné z: http://cs.olemiss.edu/~ychen/publications/conference/vicknair_acmse10.pdf

[26] BANCILA, Marius. *Loading Assemblies from Anywhere into a New AppDomain* [online]. 2012-09-05 [cit. 2015-05-24]. Dostupné z: http://www.codeproject.com/Articles/453778/Loading-Assemblies-from-Anywhere-into-a-New-AppDom

[27] GRAPHVIZ. *Graph Visualization Software* [online]. ©2000 - 2014, 2014-04-13 [cit. 2015-04-28]. Dostupné z: http://www.graphviz.org

[28] NHIBERNATE. *The object-relational mapper for .NET* [online]. ©2015, 2015-05-23 [cit. 2015-05-24]. Dostupné z: http://nhibernate.info

[29] GREGORY, James. *Fluent NHibernate* [online]. ©2008 - 2012, 2015-05-02 [cit. 2015-05-24]. Dostupné z: http://www.fluentnhibernate.org

[30] SERBAN, Sorin. *Parsing the IL of a Method Body* [online]. 2007-06-28 [cit. 2015-05-24]. Dostupné z: http://www.codeproject.com/Articles/14058/Parsing-the-IL-of-a-Method-Body

[31] MICROSOFT. *OpCodes Fields* [online]. ©2015 [cit. 2015-05-24]. Dostupné z: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes_fields

[32] MICROSOFT. *MemoryCache.Trim Method* [online]. ©2015 [cit. 2015-05-24]. Dostupné z: https://msdn.microsoft.com/en-us/library/system.runtime.caching.memorycache.trim

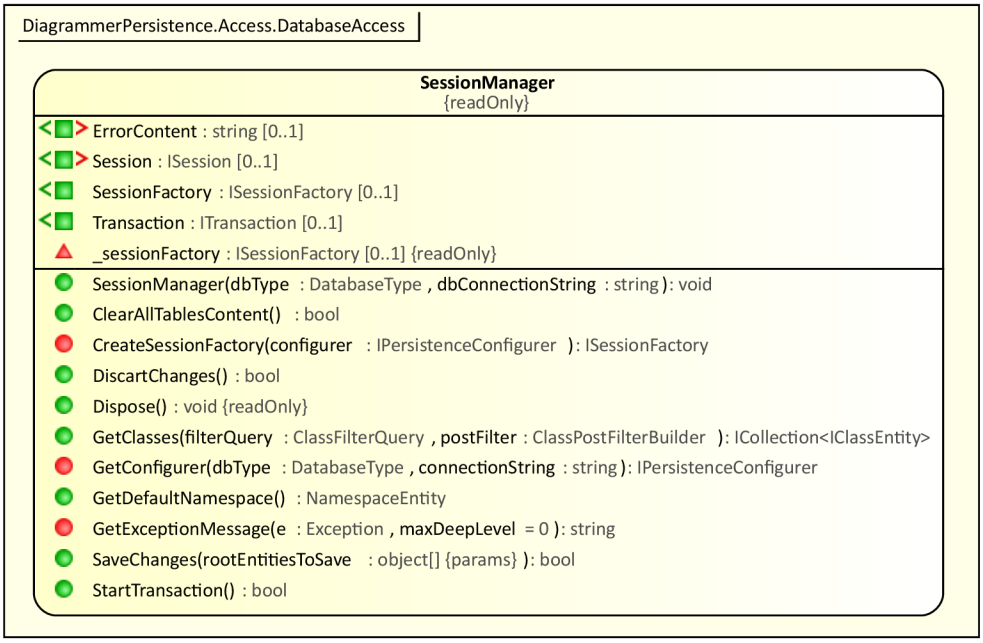
[33] CODEPLEX. *QuickGraph, Graph Data Structures And Algorithms for .NET* [online]. ©2006 - 2015, 2011-11-19 [cit. 2015-05-24]. Dostupné z: http://quickgraph.codeplex.com

[34] DIXON, Jamie. *GraphViz C# Wrapper* [online]. ©2015, 2015-04-05 [cit. 2015-05-24]. Dostupné z: https://github.com/JamieDixon/GraphViz-C-Sharp-Wrapper

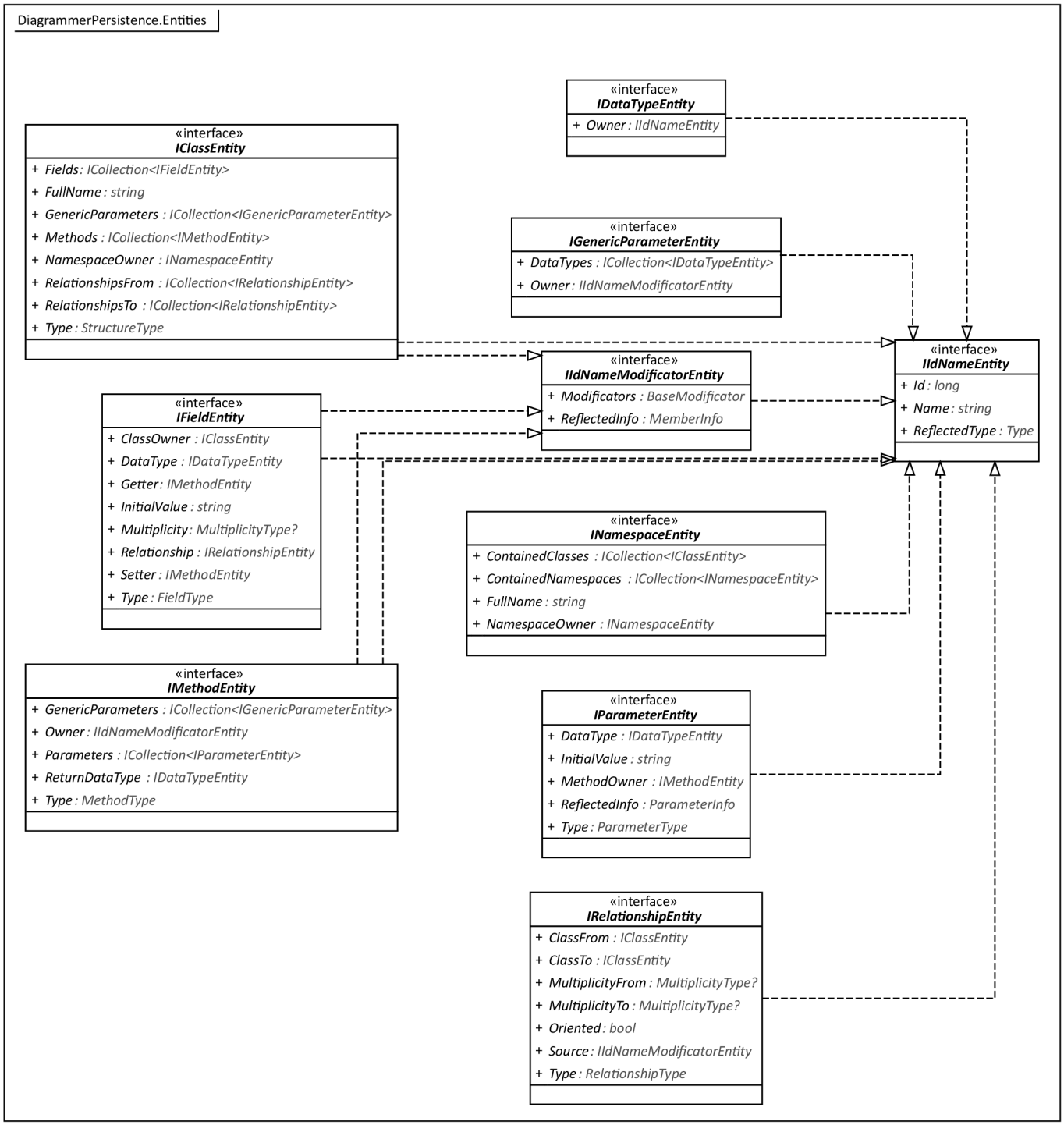
[35] BRÁZDIL, Martin. Automatické generování UML diagramu tříd. In: FIT VUT V BRNĚ. *Excel@FIT 2015* [online]. Brno, 2015, s. 2–7 [cit. 2015-05-24]. Dostupné z: http://excel.fit.vutbr.cz/download/Excel@FIT-2015-sbornik.pdf

Příloha A

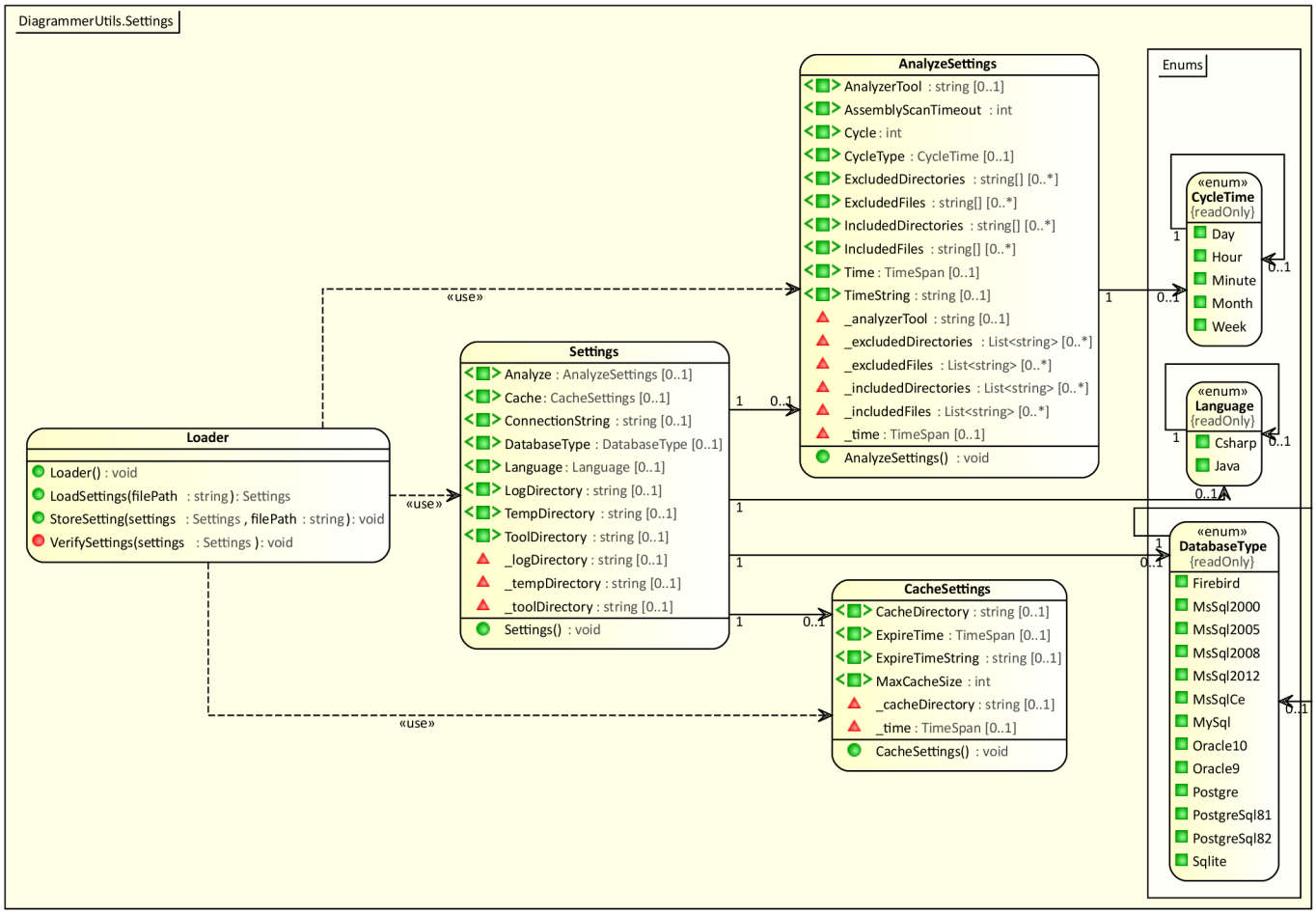
Vygenerované diagramy tříd



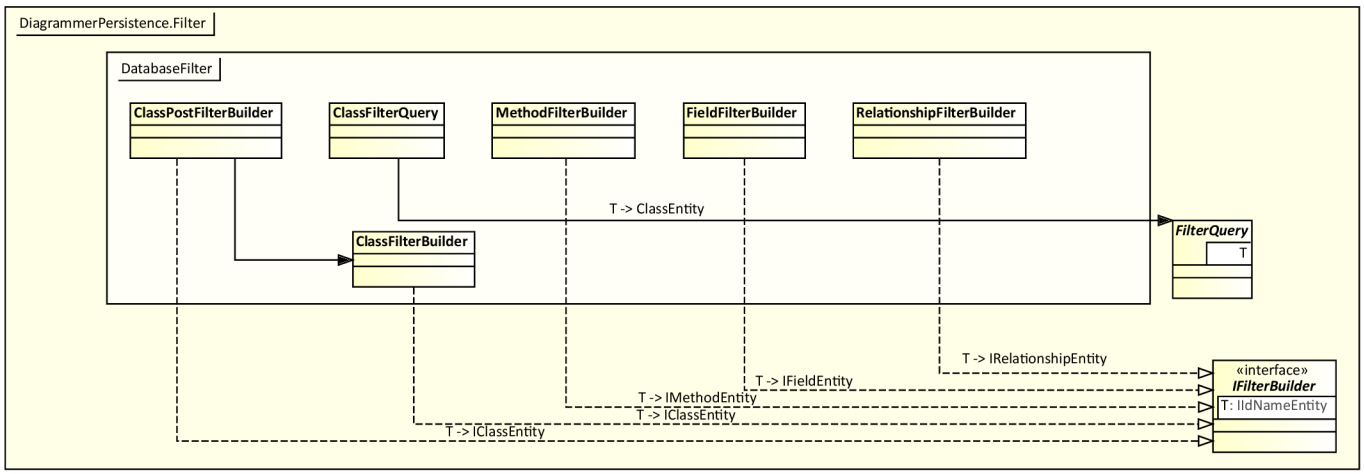
Obrázek A.1: Diagram tříd obsahující jednu třídu `SessionManager` bez žádných vztahů. Lze jej získat použitím parametrů http://ac-diagrammer.aspone.cz/DrawClass?source=DiagrammerPersistence.Access.DatabaseAccess.SessionManager&image=svg&noElements=relationship



Obrázek A.2: Diagram tříd pro rozhraní pro databázové entity v jednoduchém zobrazení. Lze jej získat použitím parametrů dle http://ac-diagrammer.aspone.cz/DrawClass?source=DiagrammerPersistence.Entities.*&image=svg&style=normal&landscape=false&noRelationships=dependency,association&noDetails=accessor,multiplicity&noColors=visibility,background



Obrázek A.3: Diagram tříd pro XML konfigurační soubor a práci s ním. Lze jej získat použitím parametrů dle http://ac-diagrammer.aspone.cz/DrawClass?source=DiagrammerUtils.Settings.**&image=svg&landscape=false



Obrázek A.4: Zjednodušený diagram tříd pro filtrování nad databází. Lze jej získat použitím parametrů dle http://ac-diagrammer.aspone.cz/DrawClass?source=DiagrammerPersistence.Filter.**&image=svg&style=normal&noParts=field,method&noRelationships=dependency

Příloha B

Obsah přiloženého CD

`./deploy/tools/` (adresář) – obsahuje analyzátor, a nástroje IKVM a Graphviz

`./deploy/web/` (adresář) – soubory potřebné k nasazení v IIS (viz příloha C)

`./doc/` (adresář) – elektronická verze této technické zprávy a její zdrojové soubory v LaTeXu

`./src/Diagrammer/` (adresář) – zdrojové soubory aplikace

`./src/ThirdPartyTools/` (adresář) – upravené zdrojové soubory knihoven třetích stran

`./test/data/` (adresář) – testovací projekty

`./test/result/` (adresář) – referenční výstupy a obsahy databází pro každý testovací projekt

`./test/src/` (adresář) – zdrojové soubory testovacích projektů

`./README.txt` (soubor) – manuál k aplikaci

Příloha C

Nasazení aplikace

Aplikace, webová služba, pro svoji činnost musí být nasazena na webovém aplikačním serveru podporující C# .NET Framework verze 4.5. Nejvhodnějším kandidátem, který byl také použit při testování v reálném prostředí, je „Internet Information Services“, zkráceně IIS, implementovaný pro operační systém Windows. Obě části výsledné architektury potřebují ke své korektní funkci konfigurační XML soubor, jehož cesta k souboru se v případě webové služby zadává při nasazení, a analyzátoru se předává skrze argument příkazové řádky (je vhodné jej vsadit mezi uvozovky z důvodu bílých znaků uvnitř cesty k souboru). Kromě konfiguračního souboru musí být dostupný i uživatelem zvolený databázový server[1].

Veškeré soubory potřebné k nasazení jsou umístěny uvnitř přiloženého CD v adresáři `./deploy/`. Pro webovou službu je vytvořen nasaditelný archiv ve formátu ZIP s názvem `diagrammer-web.zip`, což uživateli zjednodušuje celý proces nasazení[2].
Při nasazení dbejte především na:

- Dostupnost všech adresářů a souborů skrze IIS, respektive nastavit patřičná práva přístupu uživatelské skupině `IIS_IUSRS`.
- Možnost přístupu (i s editačními právy) k předem vytvořené databázi.
- Mít nástroj IKVM, respektive Graphviz, umístěn ve správné adresářové struktuře – spustitelné soubory v adresáři `./tools/IKVM/bin/`, respektive `./tools/Graphviz/bin/`.
- Mít povolenou komunikaci na daném portu v nainstalovaném „firewall“.

[1] Pro tvorbu řetězce pro připojení databáze (angl. „Connection string“) existuje internetová stránka http://www.connectionstrings.com.

[2] Postup nasazení do IIS je popsán krok za krokem na internetové stránce http://www.asp.net/web-forms/overview/deployment/web-deployment-in-the-enterprise/manually-installing-web-packages

Příloha D

Dotazovací jazyk

Všechny seznamy řetězců jsou odděleny čárkou a řetězce samotné nejsou zaobaleny v uvozovkách. Názvy parametrů označené hvězdičkou jsou povinné.

název	**typ hodnoty**	**popis**
`source`*	seznam řetězců	Seznam kompletních cest k třídám nebo jmenným prostorům možností využití zástupných znaků (hvězdičky). Cesta jmenných prostorů využívá tečkovou notaci. Pro označení vnořené třídy se používá znak $+$.
`level`	celé nezáporné číslo výchozí je ∞	Hodnota zanoření při hledání závislostí v rámci UML elementů. Centrem hledání jsou třídy ze `source` parametru. V případě skrytí některých typů závislostí v `noRelationships` se nepoužijí při zanořování.
`image`*	řetězec $\{jpeg, png, svg\}$	Typ výstupního obrázku.
`style`	řetězec $\{normal, modern\}$ výchozí je *modern*	Grafický styl generovaného diagramu. Moderní je barevný a zaoblený, oproti klasickému hranatému černobílému stylu.
`landscape`	řetězec $\{true, false\}$ výchozí je *true*	Výchozí umístění elementů UML diagramu.

Tabulka D.1: Základní úroveň dotazu

název	**typ hodnoty**	**popis**
`noElements`	seznam řetězců $\{annotation, enum, struct, class, interface, delegate, relationship\}$	Seznam typů elementů UML skrytých ve výsledném diagramu.

Tabulka D.2: První úroveň dotazu

název	typ hodnoty	popis
`noParts`	seznam řetězců $\{field, method, generic\}$	Seznam hlavních částí elementů UML skrytých ve výsledném diagramu.
`noVisibility`	seznam řetězců $\{public, protected, internal, private\}$	Seznam přístupových modifikátorů hlavních částí elementů UML skrytých ve výsledném diagramu.

Tabulka D.3: Druhá úroveň dotazu

název	typ hodnoty	popis
`noFields`	seznam řetězců $\{attribute, property, indexer, event, enumValue\}$	Seznam typů atributů skrytých ve výsledném diagramu.
`noMethods`	seznam řetězců $\{simple, constructor, destructor, operator, accessor\}$	Seznam typů metod (operací) skrytých ve výsledném diagramu.
`noRelationships`	seznam řetězců $\{dependency, generalization, realiazation, containtment, association, aggregation, composition\}$	Seznam typů vztahů skrytých ve výsledném diagramu.

Tabulka D.4: Třetí úroveň dotazu

název	typ hodnoty	popis
`noDetails`	seznam řetězců $\{accessor, visibility, dataType, initValue, constraint, multiplicity, relationshipLabel\}$	Seznam detailů částí elementů skrytých ve výsledném diagramu.

Tabulka D.5: Čtvrtá úroveň dotazu

název	typ hodnoty	popis
`noColors`	seznam řetězců $\{background, accessor, visibility, info\}$	Seznam neobarvených částí ve výsledném diagramu.
`namespace`	řetězec $\{none, normal, nested\}$	Způsob zobrazení balíků v UML diagramu tříd.

Tabulka D.6: Pátá úroveň dotazu

Příloha E

Specifické výčty použité v databázi

název	hodnota
Class	0
Interface	1
Enumeration	2
Structure	3
Delegate	4
Annotation	5

Tabulka E.1: Typy elementů UML diagramu tříd ve výčtu `StructureType`

název	hodnota
Dependency	0
Generalization	1
Realiazation	2
Containtment	3
Association	4
Aggregation	5
Composition	6

Tabulka E.2: Typy vztahů (angl. „relationships“) ve výčtu `RelationshipType`

název	hodnota
Attribute	0
Property	1
Indexer	2
Event	3
EnumValue	4

Tabulka E.3: Typy atributů (angl. „fields“) ve výčtu `FieldType`

název	hodnota
Simple	0
Constructor	1
Destructor	2
Operator	3
Accessor	4

Tabulka E.4: Typy metod (angl. „methods“ nebo „operations“) ve výčtu `MethodType`

název	hodnota
Normal	0
Ref	1
Out	2
Params	3

Tabulka E.5: Typy parametrů metod ve výčtu `ParameterType`

název	hodnota
Zero	0
One	1
ZeroOrOne	2
ZeroOrMore	3
OneOrMore	4

Tabulka E.6: Typy násobností (angl. „multiplicity“) ve výčtu `MultiplicityType`

název	**hodnota**
NoAccess	0
Public	1
Internal	2
Protected	4
ProtectedOrInternal	8
ProtectedAndInternal	16
Private	32
Static	64
Constant	128
Abstract	256
Final	512
Extern	1024

Tabulka E.7: Typy základních modifikátorů ve výčtu `BaseModificator` ve tvarů příznaků

Příloha F

Schéma XML souboru s nastavením

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Settings">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Analyze">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:time" name="Time" />
              <xs:element type="xs:byte" name="Cycle" />
              <xs:element name="CycleType">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="Minute"/>
                    <xs:enumeration value="Hour"/>
                    <xs:enumeration value="Day"/>
                    <xs:enumeration value="Week"/>
                    <xs:enumeration value="Month"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
              <xs:element type="xs:int" name="AssemblyScanTimeout" />
              <xs:element type="xs:string" name="AnalyzerTool" />
              <xs:element name="IncludedDirectories">
                <xs:complexType>
                  <xs:sequence><xs:element type="xs:string" name="Directory" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="ExcludedDirectories">
                <xs:complexType>
                  <xs:sequence><xs:element type="xs:string" name="Directory" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="IncludedFiles">
                <xs:complexType>
                  <xs:sequence><xs:element type="xs:string" name="File" />
                  </xs:sequence>
                </xs:complexType>
```

```
            </xs:element>
            <xs:element name="ExcludedFiles">
              <xs:complexType>
                <xs:sequence><xs:element type="xs:string" name="File" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Cache">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string" name="Directory" />
            <xs:element type="xs:byte" name="MaxCacheSize" />
            <xs:element type="xs:time" name="ExpireTime" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Language">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Csharp"/>
            <xs:enumeration value="Java"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="DatabaseType">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="Firebird"/>
            <xs:enumeration value="MsSql2000"/>
            <xs:enumeration value="MsSql2005"/>
            <xs:enumeration value="MsSql2008"/>
            <xs:enumeration value="MsSql2012"/>
            <xs:enumeration value="MsSqlCe"/>
            <xs:enumeration value="MySql"/>
            <xs:enumeration value="Oracle9"/>
            <xs:enumeration value="Oracle10"/>
            <xs:enumeration value="Postgre"/>
            <xs:enumeration value="PostgreSql81"/>
            <xs:enumeration value="PostgreSql82"/>
            <xs:enumeration value="Sqlite"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element type="xs:string" name="ConnectionString" />
      <xs:element type="xs:string" name="ToolDirectory" />
      <xs:element type="xs:string" name="TempDirectory" />
      <xs:element type="xs:string" name="LogDirectory" />
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```