

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

NÁVRH A IMPLEMENTACE PROGRAMU PRO PŘEVOD UML STRUKTUR DO PROGRAMOVACÍHO JAZYKA

DESIGN AND IMPLEMENTATION OF A TOOL TO CONVERT UML STRUCTURES INTO PROGRAMMING LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Alice Minářová
AUTHOR

VEDOUCÍ PRÁCE — Ing. Petr Jindra, Ph.D.
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav automatizace a informatiky
Studentka:	**Bc. Alice Minářová**
Studijní program:	Strojní inženýrství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**Ing. Petr Jindra**
Akademický rok:	2016/17

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Návrh a implementace programu pro převod UML struktur do programovacího jazyka

Stručná charakteristika problematiky úkolu:

Při návrhu programů a systémů bývá obvykle jako součást dokumentace veden model součástí programu (například hierarchie tříd, databázový model atd) formou E–R (Entity Relationship) digramu, respektive diagramu tříd (class diagram) popsaném pomocí UML. Během vývoje se většinou tento návrh provádní jako první krok a implementace se provádí až po dokončení návrhu. Převod tohoto diagramu do programovacího jazyka bývá zpravidla časově náročný a náchylný k lidské chybě.

Cíle diplomové práce:

Popište problematiku převodu UML do programových struktur (včetně přehledu programů pro modelování UML struktur).
Proveďte analýzu problematiky.
Na základě zjištěných informací implementujte program s licencí typu Open Source provádějící převod UML do programových struktur (například databázové struktury, kostry tříd atd). Vyžadována je podpora alespoň jednoho vstupního formátu UML a alespoň dvou výstupních formátů (programovací jazyky) s jednoduchou metodou rozšíření o další vstupní a výstupní formáty.
Zhodnoťte možnosti dalšího vývoje programu (integrace podpory dalších vstupních a výstupních formátů, nové funkce atd) a jeho využití v praxi.

Seznam doporučené literatury:

SCHMULLER Joseph. Myslíme v jazyku UML. 1. vyd. Praha: Grada, 2001. Knihovna programátora. ISBN 80-247-0029-8.

HOLUBOVÁ Irena, POKORNÝ Jaroslav. XML technologie: principy a aplikace v praxi. 1. vyd. Praha: Grada, 2008. Průvodce. ISBN 978-80-247-2725-7. Dostupné také z: http://kramerius.mzk.cz/search/handle/uuid:74f376e0-66ea-11e3-8fee-005056827e51.

ARLOW Jim, NEUSTADT Illa. UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky. 2., aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-2-1-1503-9. Dostupné také z: http://kramerius.mzk.cz/search/handle/uuid:1bb62180-abda-11e6-b707-005056827e51.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2016/17

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

ABSTRAKT

Tato práce se zabývá problematikou UML diagramů na kód. Nejprve jsou analyzována existující řešení na poli této problematiky a na základě získaných zjištění je navržen a implementován vlastní nástroj. Tento přijímá UML diagramy tříd a databázové modely z volně dostupného modelovacího prostředí Dia, které každý dokáže převést do tří cílových jazyků. Stěžejní vlastností tohoto nástroje je možnost snadno modulárně přidávat další cílové jazyky.

ABSTRACT

This work deals with the problematics of conversion of UML diagrams to code. Initially, existing solutions in this field are analyzed. Based on the gained findings, a new tool is designed and implemened. It accepts UML class diagrams and database models from the free diagramming environment Dia and for each of these offers three target languages. The key feature of this tool is its modular design allowing new target languages to be added easily.

KLÍČOVÁ SLOVA

Diagramy UML, relační databázový model, převod diagramů na kód

KEYWORDS

UML diagrams, relational database model, diagram to code conversion

BIBLIOGRAFICKÁ CITACE

MINÁŘOVÁ, Alice. *Návrh a implementace programu pro převod UML struktur do programovacího jazyka*, Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky.

PODĚKOVÁNÍ

Chci poděkovat svému manželovi za podporu ve chvílích, kdy se realizace této práce zdála kvůli zdravotním problémům nemožná. Naší dcerce Sofii za něhu a útěchu, kterou plní můj život. Jejím babičkám a dědečkům za to, že mi svou obětavostí při hlídání umožnili vůbec dojít k tomuto momentu v mém studiu. Velký dík patří i vedoucímu práce, Petru Jindrovi za to, že mi tuto práci nabídl a že komunikace s ním byla vždy příjemná a přátelská.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracovala jsem ji samostatně pod vedením Ing. Petra Jindry, Ph.D. a s použitím literatury uvedené v seznamu literatury.

V Brně dne 24. 5. 2017 ……

Alice Minářová

OBSAH

1 ÚVOD 15

2 MOTIVACE 17

3 DIAGRAMY RELEVANTNÍ V PROBLEMATICE PŘEVODU UML DIAGRAMŮ NA KÓD 19

3.1 Jazyk UML 19
3.1.1 Historie 19
3.1.2 Členění UML 20
3.1.3 Prvky podporující rozšiřitelnost 20
3.1.4 Kritika 21
3.2 Diagram tříd 21
3.2.1 Třída 21
3.2.2 Vazby 22
3.3 Relační databázový model 23
3.3.1 Historie 24
3.3.2 Princip 24
3.3.3 Příklad relačního modelu 25
3.3.4 Modelování relačního databázového modelu pomocí UML 26

4 TYPY CÍLOVÝCH PROGRAMOVACÍCH JAZYKŮ 29

4.1 Objektově orientované programování 29
4.1.1 Dědičnost 29
4.1.2 Třídně orientované jazyky 30
4.1.3 Objektově orientované jazyky 31
4.2 Jazyk SQL 31
4.2.1 Příkaz `CREATE TABLE` 31
4.2.2 Databázové systémy užívající SQL 32

5 ANALÝZA EXISTUJÍCÍCH ŘEŠENÍ NA POLI PŘEVODU DIAGRAMŮ UML NA KÓD 33

5.1 Komerční řešení 33
5.1.1 Visual Paradigm 33
5.1.2 UModel 2017 33
5.1.3 Enterprise Architect 33
5.1.4 BOUML 34
5.1.5 Yatta UML Lab 34
5.1.6 StarUML 2 34
5.2 Volně dostupná řešení 34
5.2.1 ArgoUML 34
5.2.2 Moduly pro Eclipse 35
5.3 Shrnutí 35

6 NÁVRH VLASTNÍHO NÁSTROJE PRO GENEROVÁNÍ KÓDU Z UML 37

6.1 Diagramové prostředí Dia 37
6.1.1 Mód UML a jeho použití při modelování diagramu tříd 37
6.1.2 Mód Database 41
6.1.3 XML reprezentace diagramů 42

6.2 Návrh nástroje Dia2Code 44
6.2.1 Volba programovacího jazyka 44
6.2.2 Zpracování vstupních diagramů ve formátu Dia XML 44
6.2.3 Návrh struktur pro uchování získaných dat 45
6.2.4 Návrh modulů pro generování kódu s ohledem na žádanou podporu přidání dalších jazyků 45
6.2.5 Výběr cílových jazyků 46
7 IMPLEMENTCE NAVRŽENÉHO ŘEŠENÍ 47
7.1 Adresářová struktura 47
7.2 Parametry příkazové řádky 47
7.2.1 Poziční parametry 48
7.2.2 Volitelné parametry 48
7.2.3 Kontrola validity zadaných parametrů 49
7.3 Parser 49
7.3.1 Třída `XmlUmlParser` 50
7.3.2 Třída `ClassUmlParser` 50
7.3.3 Třída `DatabaseUmlParser` 52
7.4 Třídy pro uchování dat 53
7.4.1 Třída `Class` 54
7.4.2 Třídy `Atribute`, `Method` a `Parameter` 54
7.4.3 Třída `Association` 55
7.4.4 Třída `Table` 56
7.4.5 Třída `DbAttribute` 57
7.5 Generátor 57
7.5.1 Třída `ClassGenerator` 58
7.5.2 Třída `DatabaseGenerator` 59
7.6 Třídy textbank 60
7.6.1 Charakteristika textbanky 61
7.6.2 Struktura textbanky 61
7.6.3 Princip činnosti textbanky 62
7.6.4 Význačné vlastnosti jednotlivých implementovaných textbank 63
7.7 Modul zpracování chyb 67
8 DEMONSTRACE ČINNOSTI 69
8.1 Mód `class` 69
8.2 Mód database 70
9 ZHODNOCENÍ A NÁVRH ROZŠÍŘENÍ 73
10 ZÁVĚR 75
SEZNAM OBRÁZKŮ A TABULEK 81

1 ÚVOD

Rozvoj a směřování naší civilizace jsou stále více spjaty s informačními technologiemi a možnostmi, které tyto nabízí. Informační technologie prostupují všemi vrstvami vyspělé společnosti, od vesmírných stanic a špičkových výzkumných center, po ty nejběžnější každodenní činnosti.

Česká republika i Evropa se v současnosti potýkají s nedostatkem kvalifikovaných pracovníků na poli IT [1]. V důsledku toho se zvyšují požadavky na efektivitu práce programátora. Předcházení obtížím v průběhu životního cyklu softwaru představuje možnou cestu, jak efektivitu zvýšit.

Jedna z mnoha takových obtíží může nastat ve fázi návrhu softwaru. Vhodně řešený návrh je z hlediska IT firem klíčový, je však v zájmu každého programátora, který se pouští do většího projektu, nezačínat s psaním kódu bez předchozí přípravy. Vypuštění této fáze mívá za následek sníženou systematičnost práce, opomínání součástí a jejich dodatečné přidávání (což má zpravidla negativní vliv na přehlednost kódu) a v neposlední řadě zhoršuje komunikaci v týmu, pracuje-li na projektu více lidí.

Častým nástrojem používaný v této fázi je modelovací jazyk UML (Unified Modelling Language). Diagramy UML umožňují vizualizovat součásti navrhovaného systému a usnadňují softwarovým inženýrům a programátorům vzájemnou komunikaci a orientaci v návrhu. Proces vývoje však dříve nebo později dospěje do stádia, kdy je potřeba struktury zaznamenané v UML diagramech přenést do kódu. Jedná se zpravidla o zdlouhavý proces náchylný k chybám (vynechání položky, překlep, přehlédnutí se o řádek), které se obtížně hledají.

Cílem této práce je navrhnout a vytvořit Open Source program, který na vstupu přijme UML diagram a transformuje struktury v něm zanesené do uživatelem zvoleného programovacího jazyka. Výsledný program bude přijímat diagramy vytvořené ve volně dostupném diagramovém prostředí Dia a to diagramy dvou často používaných typů: relační databázový model a model tříd. Žádoucí vlastností programu bude možnost modulárně přidávat další cílové jazyky, případně typy zdrojových diagramů. a tedy potenciálně neomezená možnost uživatele přizpůsobit výstup programu svým individuálním potřebám. Rovněž bude kladen důraz na ovládání přes příkazovou řádku, aby byl program snadno využitelný ve složitějších skriptech při testování.

2 MOTIVACE

Problematika převodu UML na kód cílového jazyka není v oboru ničím novým. Existují komerční modelovací prostředí, která nabízí širokou škálu cílových jazyků, jejich cena však může být pro mnoho vývojářů nepřijatelná či odrazující. Další volně dostupné generátory bývají omezené na konkrétní jazyky (a konkrétní typ jazyků) a nevyhovují-li tyto uživateli, přizpůsobit výstup svým potřebám buď není možné, nebo stěží realizovatelné v daném časovém rámci. Nutno podotknout, že výše zmíněná řešení jsou grafické povahy, ovládání přes příkazovou řádku uživateli neumožňují.

Ideou této práce je vytvoření jednoduchého nástroje pro převod UML na kód, který v jedné aplikaci spojí pozitiva výše zmíněných existujících řešení a odstraní aspekty negativní. Nástroj bude přijímat diagramy vytvořené v neplaceném diagramovém prostředí. Jeho funkcionalita nebude omezena pouze na jeden konkrétní typ vstupního diagramu a pevně stanovené výstupní jazyky. Bude navržen tak, aby přidání vlastních jazyků, či modifikace výstupu byly zcela v režii jednotlivých uživatelů a tato modifikace nezabrala více než jednotky hodin. Nástroj bude ovládán přes příkazovou řádku, čímž umožní uživateli funkcionalitu nástroje zahrnout do komplexnějších skriptů a usnadnit tak například testování.

3 DIAGRAMY RELEVANTNÍ V PROBLEMATICE PŘEVODU UML DIAGRAMŮ NA KÓD

Abychom se mohli zabývat problematikou generování kódu z namodelovaných diagramů a existujícími řešeními v této doméně, je nutné si nejprve detailněji představit vstupní a výstupní elementy, se kterými tyto generátory pracují. V této kapitole se budeme zabývat elementy vstupními, tedy modely realizovanými v jazyce UML. Těmito jsou především diagram tříd (kapitola 3.2) a relační databázový model (kapitola 3.3).

3.1 Jazyk UML

Jazyk UML (*Unified Modellig Language*) je všestranný modelovací prostředek užívaný v oblasti softwarového inženýrství pro návrh a vizualizaci vyvíjeného systému. Nabízí bohatou grafickou notaci a obsáhlou bázi jednotlivých typů diagramů. V současnosti se jedná o standard při vývoji softwaru.

3.1.1 Historie

Počátkem 90. let došlo k prudkému rozmachu objektově orientovaného programování (OOP), když došlo k rozšíření jazyků podporujících tyto techniky. Spolu se zvětšující se rolí, kterou OOP hrálo na poli vývoje softwaru, vznikl zájem vytvořit formální notaci pro vhodný popis takto řešených systémů.

V letech 1994-1995 svedla společnost *Rational Software* dohromady autory tří objektových přístupů k modelování. Těmito byli Grady Booch, autor tzv. Boochovy metody (modelovací jazyk a metodologie s ideou "více tříd zpřehlední kód"), James Rumbaugh, spoluautor "objektově-modelovací techniky" (*object-modelling technique, OMT*) a Ivar Jacobson, autor objektově orientovaného softwarového inženýrství (*Object-oriented software engeneering, OOSE*). [2]

Booch, Rumbaugh a Jacobson své metodologie zainegrovali do jediného jazyka, který nazvali UML - U*nified Modelling Language*. V roce 1996 bylo pod jejich vedením založeno consorcium *UML Partners*, které v roce 1997 dokončilo práce na specifikacích jazyka a návrh UML 1.0 předložilo mezinárodnímu consorciu *Object Management Group* (OMG) ke standardizaci. Později v témže roce dodalo *UML Partners* první plnohodnotně specifikovanou verzi, UML 1.1, které bylo skupinou OMG přijato a jazyk UML počínaje rokem 1997 spadá pod její správu. [2]

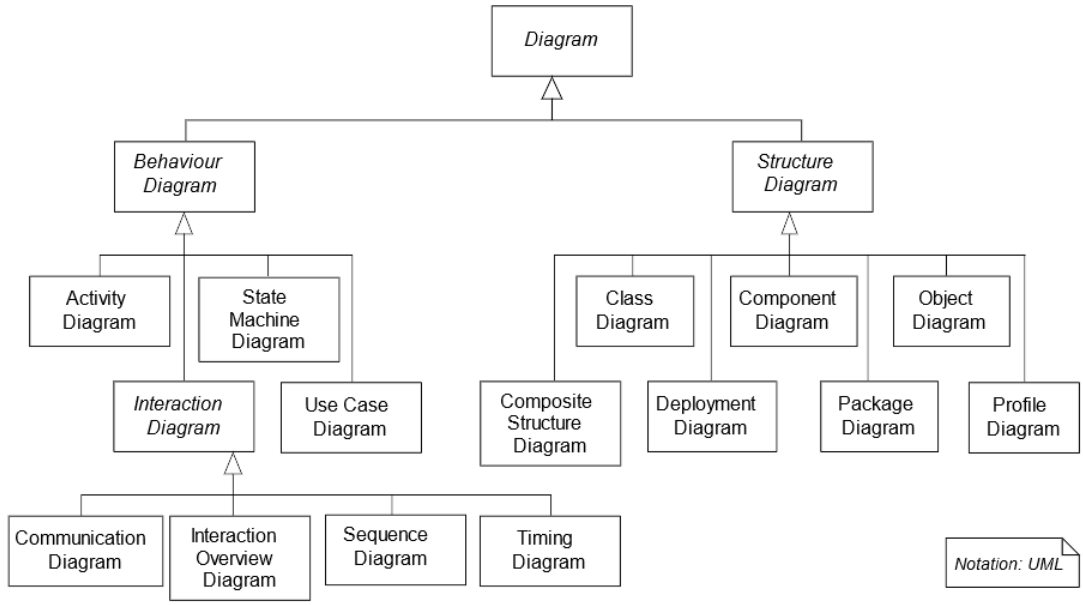
Verze 2.0 byla představena v roce 2005 a současná verze UML 2.5 je aktuální od roku 2015. [3]

3.1.2 Členění UML

UML 2 obsahuje velké množství diagramů, které lze rozdělit na dvě podkategorie: strukturální a behaviorální [4], jak lze vidět na Obr. 1.

Strukturální diagramy zachycují statickou strukturu modelovaného systému a používají se při dokumentaci architektury systému. Příkladem může být například diagram balíků (*package diagram*), který modeluje závislostní vztahy mezi jednotlivými balíky v systému.

Behaviorální diagramy umožňují zaznamenat činnosti, které v systému probíhají; používají se pro popis funkcionality systému. *Use-case* diagram například modeluje varianty interakce, které mohou probíhat mezi systémem a různými typy uživatelů.



Obr. 1: Rozdělení UML diagramů na skupiny [5]

3.1.3 Prvky podporující rozšiřitelnost

UML obsahuje i prvky podporující rozšíření tohoto diagramu nad rámec původně plánované funkcionality. Tyto prvky představují velice volné rozhraní, jehož specifickým použitím lze určit, o jakou přidanou funkcionalitu se jedná. Těmito prvky jsou omezení, tagy (kterými se v této práci nebudeme dále zabývat) a tzv. stereotypy.

Stereotyp slouží k specifikací prvků nad rámec původního UML. Jedná se o identifikátor vložený mezi dvojice ostrých závorek (« », případně << >>). Zobrazuje se nad názvem daného prvku. Například použitím stereotypu «*getter*» nad názvem metody v diagramu tříd stanovujeme, že se jedná o metodu typu *getter* (metody sloužící k získání aktuálních hodnot privátních proměnných). [4] [6]

3.1.4 Kritika

Přijetí skupinou OMG učinilo z UML standard, jinými slovy se jedná o jedinou notaci, u které se dá očekávat, že jí budou rozumět další programátoři napříč IT spektrem. Pro konkrétní případy lze jako alternativy využít jiné typy diagramů, např.: *Data-flow diagram, Control flow diagram, Vennův diagram, Petriho sítě,* nebo *E -R diagram*, který bude rozebrán v kapitole 3.2. Tyto ale nejsou tolik rozšířeny, ani nejsou natolik univerzální, aby se o nich dalo uvažovat jako o přímé konkurenci monopolu UML.

Tento monopol, který UML na poli softwarového inženýrství má, se však nesetkává vždy s pozitivními ohlasy. Nejznámější kritika UML pochází z úst jednoho z původních autorů UML, Ivara Jacobsona: *"UML se stalo příliš komplexním a nemotorným. Na 80% veškerého softwaru potřebujeme pouze 20% UML."* [7]

Další kritika se snáší na přílišné užívání UML i v situacích, kdy to není za potřebí. Tento problém adresuje další z autorů UML, Grady Brooch: *"Člověk by měl používat grafickou notaci pouze pro ty věci v kódu, které nejsou snadno zdůvodnitelné."* [7]

3.2 Diagram tříd

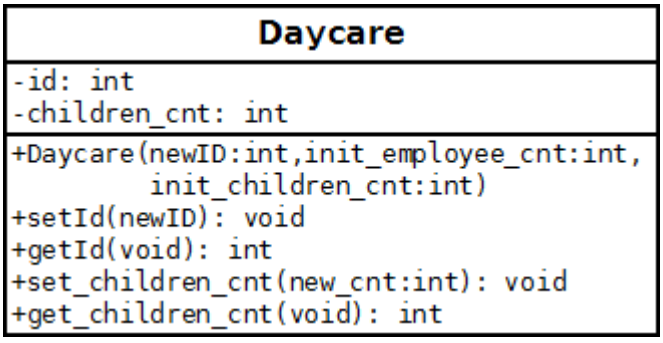
Diragram tříd (Class diagram) je jedním ze strukturálních diagramů jazyka UML. Používá se pro popis struktury tříd systému. Jsou v něm zaneseny jednotlivé třídy (jejich názvy, atributy a metody) a vzájemné vztahy mezi těmito třídami.

Jedná se o diagram na úrovni implementace: model musí být úplný, musí obsahovat veškeré elementy a vztahy. Jedná se o grafickou reprezentaci samotného kódu, tedy tříd tak, jak je popisuje OOP (kapitola 4.1). [8]

3.2.1 Třída

Třída je v diagramu tříd zobrazena jako ohraničený box, který je čarami rozdělen na tři části Obr. 2:

- **Název** - horní třetina boxu, centrovaný, tučně zobrazený text.
- **Atributy** - prostřední třetina boxu, doleva zarovnaný obyčejný text.
- **Metody** - dolní třetina boxu, doleva zarovnaný obyčejný text.



Obr. 2: Příklad notace třídy v diagramu tříd

Protože atributy v objektových jazycích mohou mít nastavenou implicitní hodnotu, (totéž platí i pro parametry metod), diagram tříd rovněž umožňuje její nastavení.

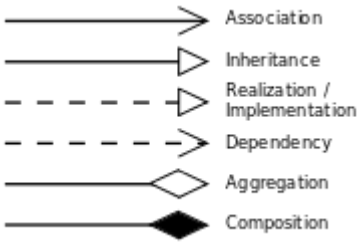
Názvům atributů a metod může být předřazen příznak viditelnosti (*access modifier*), který stejně jako v mnoha objektově orientovaných jazycích určuje rozsah dostupnosti daného elementu dalším třídám. [8]

+	*Public*
-	*Private*
#	*Protected*
/	*Derived*
~	*Package*

Tab. 1: Příznaky viditelnosti v diagramu tříd

3.2.2 Vazby

Grafická notace jednotlivých typů vazeb, které mezi jednotlivými třídami mohou nastat, je znázorněna na Obr. 3.



Obr. 3: Grafická notace vazeb mezi třídami [9]

Association *(*asociace*)* je vztah značen jednoduchou spojitou čarou. Může propojovat libovolné množství tříd (včetně jedné a téže třídy) a to v libovolné kombinaci směrů (směr je značen jednoduchou, čarovou šipkou). Nejčastěji používanou variantou je varianta binární. Všem zúčastněným třídám je při jejich konci asociačního vztahu přiřazena násobnost určující množství, kolikrát daná třída ve vztahu figuruje (např. třída Cestující může být v asociaci s třídou Autobus zastoupena *0* až n-krát, značeno *0..n*). Asociace může být pojmenovaná, rovněž jednotlivé participující třídy mohou mít přiřazeny název své role. [8]

Inheritance (dědičnost), také označovaná jako ***generalization*** *(*generalizace*)* je vztah modelující dědičnost tříd tak, jak je známá z OOP (více v kapitole 4.1.1). Značí se souvislou čarou zakončenou trojúhelníkem bez výplně směřující k rodičovské třídě.

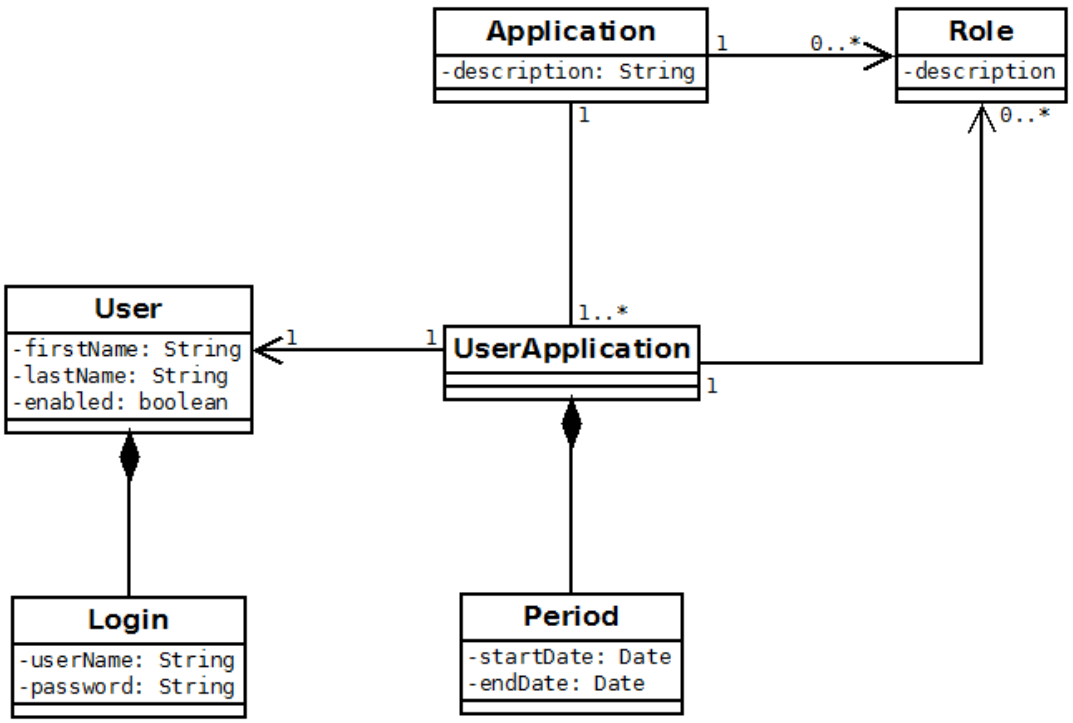
Realization (realizace) značí vztah mezi třídou a rozhraním, jehož metody třída implementuje. Značí se přerušovanou čarou zakončenou nevyplněným trojúhelníkem směřujícím k rozhraní.

Dependency (závilost) je slabším, méně závazným typem vazby mezi třídami. Nezávislá třída ve vztahu je používána třídou závislou buď jako parametr metody nebo lokální proměnná. (Na rozdíl od asociace, kde je závislá třída atributem nezávislé třídy.) Značí se přerušovanou čarou s šipkou směřující k nezávislé třídě. [8]

Aggregation *(*agregace*)* je specializovaný případ asociace, kde nadřazená třída, "vlastník", vlastní podřízenou třídu a tato nemůže být vlastněna jiným vlastníkem. Dojde-li však ke smazání vlastníka, podřízená třída nepřestává existovat. Značí se plnou čarou zakončenou nevyplněným kosočtvercem směřujícím k nadřazené třídě. [8]

Composition *(*kompozice*)* je rovněž specializovaný případ asociace, jedná se o silnější vztah než agregace: podřízená třída při zrušení vlastníka rovněž přestává existovat. Značí se plnou čarou zakončenou vyplněným kosočtvercem směřujícím k nadřazené třídě. [8]

Na Obr. 4 je jako příklad uveden jednoduchý diagram tříd s použitými vztahy kompozice a jednosměrné a dvousměrné binární asociace.



Obr. 4: Příklad jednoduchého diagramu tříd

3.3 Relační databázový model

Relační databázový model je přístup k modelování dat, který je založen na teorii množin a predikátové logice prvního řádu. Databáze využívající tohoto principu se nazývají **relační databáze** a ke své implementaci nejčastěji využívají jazyk SQL (kterým se budeme blíže zabývat v kapitole 4.2).

3.3.1 Historie

Relační model byl poprvé představen Edgarem F. Coddem v letech 1969-70 ve dvou článcích: "*Redundancy, and Consistency of Relations Stored in Large Data Banks*" [10] a "*A Relational Model of Data for Large Shared Data Banks*" [11]. Jednalo se o první formálně popsaný databázový model v kontrastu s již existujícím hierarchickým a síťovým modelem.

3.3.2 Princip

V tomto modelu jsou data reprezentována jako uspořádané n-tice sdružené do **relací** (obvykle pojmenovaných). Relace se nejčastěji chápe jako **tabulka**, n-ticím, které tato sdružuje se pak říká **řádky** tabulky (lze se také setkat s termínem "záznam"). Pořadí řádků není z pohledu relace relevantní. Jednotlivé prvky n-tic představují **sloupce** tabulky a každý sloupec má pevně stanovený název, datový typ, příp. rozsah či další, tzv. integritní omezení. Data spadající do jednotlivých sloupců jsou striktně vázaná omezeními těchto sloupců. [11]

Primární a cizí klíč

Klíč je jedním z nejčastěji používaných omezení. Hodnoty v sloupci s tímto omezením musí obsahovat data unikátní v rámci celého sloupce. Klíče se používají k jednoznačnému určení řádků tabulky, mohou být jednoduché (zahrnují pouze jeden sloupec), nebo složené (unikátní kombinace hodnot ve více sloupcích).

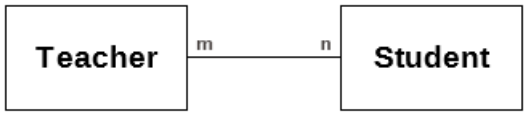
Klíč vlastní dané tabulce se nazývá **primární klíč**, tabulka však může obsahovat sloupce odkazující na jiné tabulky. V těchto sloupcích jsou pak uloženy hodnoty sloupců (nejčastěji primárních klíčů) referovaných tabulek. Takový sloupec se pak nazývá **cizí klíč**. V závislosti na povaze odkazovaného primárního klíče může cizí klíč zahrnovat jeden nebo více sloupců. Je žádoucí (většinou explicitně vyžadované), aby každá tabulka měla primární klíč.

Kardinalita

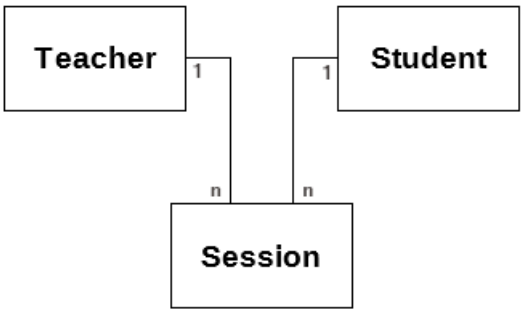
Při modelování databází se spojení mezi tabulkami standardně označují kardinalitou. Kardinalita určuje, kolik objektů/entit reprezentovaných tabulkami ve vztahu, se tohoto vztahu může účastnit. Kardinalita existuje ve dvou základních derivacích: *one-to-many* a *many-to-many***.**

Příkladem *one-to-many* (*1..n*) může být vazba mezi tabulkami Matka a Dítě. Matka je jen jedna, ale může mít více dětí. Zatímco každé dítě může mít jen jednu matku. Této vazbě by odpovídal sloupec s cizím klíčem v tabulce Dítě odkazujícím na záznam v tabulce Matka. Speciálním případem *one-to-many* je vazba *one-to-one* (*1..1*).

Variantě *many-to-many* (*m..n*) by naopak odpovídala vazba mezi tabulkami Učitel a Student. Učitel může učit více studentů, ale každý student může být vyučován více učiteli. Tento typ ale vazby není v databázových systémech realizovatelný, bývá tedy modifikován vložením třetí tabulky, která nese záznamy o jednotlivých variantách vztahu. Efektivně tak z *many-to-many* vztahu učiní dva vztahy *one-to-many* jak demonstrují Obr. 5 a Obr. 6.



Obr. 5: Příklad m..n vazby



Obr. 6: Modifikace m..n vazby vložením třetí tabulky

3.3.3 Příklad relačního modelu

Na Obr. 7 je uveden příklad jednoduché databáze o dvou tabulkách. Horní tabulka, nazvěme ji Owner (Majitel**)**, obsahuje záznamy o jednotlivých majitelích psů. Má čtyři sloupce: ID majitele (které je primárním klíčem tabulky), jméno, pohlaví a zaměstnání. První záznam například nese data o majitelce Jane Doe, která je žena a je již v důchodu.

Druhá tabulka, kterou nazveme Dog (Pes**)**, slouží k evidenci psů. Obsahuje pět sloupců, ID psa (primární klíč), rasu, jméno a pohlaví. Poslední sloupec je cizí klíč, který odkazuje na tabulku Owner. Hodnota v tomto sloupci odpovídá ID majitele, který vlastní psa, jemuž náleží daný záznam. Zajímá-li nás například, kdo vlastní psa jménem Ares, použijeme ID majitele ve sloupci s cizím klíčem (tj. id = 3) a následně se podíváme do tabulky Owner, kterému majiteli dané ID náleží. Zjistíme, že psa Area vlastní majitelka Tina Smith.

Rovněž můžeme zjistit, jestli tato majitelka nevlastní ještě další psy, a to vypsáním všech psů, jejichž ID majitele odpovídá ID Tiny Smith. Databáze nás informuje, že Tina Smith vlastní dva psy, Area a Lunu.

OwnerID	Name	Gender	Occuption
1	Jane Doe	Female	Retired
2	Jonathan Archer	Male	Starship captain
3	Tina Smith	Female	Artist

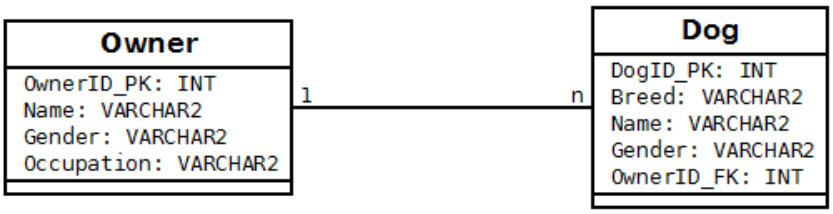
DogID	Breed	Name	Gender	OwnerID
1	Akita	Luna	Female	3
2	Beagle	Porthos	Male	2
3	Boxer	Ares	Male	3
4	Poodle	Mrs. Fluffy	Female	1

Obr. 7: Příklad jednoduché relační databáze

3.3.4 Modelování relačního databázového modelu pomocí UML

Při bližším pohledu na vizualizaci relací z obrázku 7 je možné vypozorovat určitou podobnost s diagramem tříd. Oba tyto modely se skládají z elementů znázorněných boxy, které jsou propojeny vazbami s popsanou četností. V obou případech tyto boxy (třída, tabulka) obsahují další sub-elementy (atributy/metody, sloupce). Díky této podobnosti se diagram tříd používá pro modelování relačních databázových modelů. (Z dalších prvků UML se používají komponenty a balíky pro modelování vztahů mezi databázemi a schématy, tyto modely jsou však nad rámec této práce.)

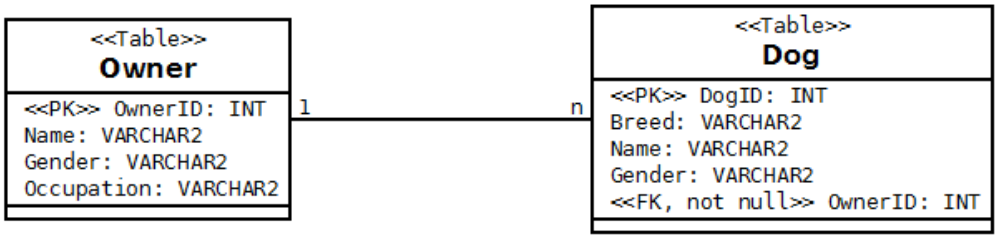
Ve své nejjednodušší podobě probíhá převod relačního modelu na diagram tříd takto: Tabulky modelujeme jako třídy, název tabulky se stane názvem třídy. Sloupce tabulky chápeme jako atributy, jejich názvy odpovídají názvům atributů, datový typ pak odpovídá datovým typům atributů. Primárnímu klíči může být pro přehlednost předřazen modifikátor PK. Vazby mezi třídami se modelují jako asociace s četností odpovídající kardinalitě vazby mezi tabulkami. Modifikátory viditelnosti se buď nepoužívají, nebo (není-li možné je vynechat) se neberou v potaz. [12] (Výjimku tvoří modifikátor *protected* (#), který může být použit k označení primárního klíče.) Převod tabulek z obrázku 7 na UML diagram je ukázán na Obr. 8.



Obr. 8: Tabulky Majitel a Pes za použití diagramu tříd

Tato notace bývá často rozšiřována pomocí stereotypů (kapitola 3.1.3). Názvům tabulek bývá předřazen stereotyp «*Table*», primární či cizí klíče mají stereotypy «*PK*», resp. «*FK*». Obdobně i integritní omezení daného atributu může být určeno stereotypem, např.: «*nullable*», «*unique*». Stereotypy je možné uvnitř ostrých závorek řetězit, vztahuje-li se na daný atribut více než jeden, např.: «*FK, nullable*».

Používání stereotypů umožňuje přesněji specifikovat vlastnosti jednotlivých UML prvků tak, aby co nejvíce odpovídaly cílové doméně, tedy zamýšlenému databázovému modelu. Díky nim lze využít metody třídy pro procedury spojené s danou tabulkou, tedy označit tyto metody jako «*trigger*», «*index*» atp. Na obrázku x je uveden jednoduchý příklad tabulky Učitel vymodelované pomocí třídy z diagramu tříd za použití stereotypů.



Obr. 9: Příklad použití databázových stereotypů

Nevýhoda tohoto přístupu vyplývá z podstaty stereotypů: jedná se o variabilní rozhraní pro přizpůsobení UML diagramů záměrům, pro které tyto nebyly navrženy. Neexistuje tedy žádný UML standard, který by přesně definoval, jaký formát mají stereotypy používané pro modelování databází používat. Ačkoliv tedy existuje jakýsi rámcově shodný přístup k identifikátorům, které se v databázových stereotypech používají, výsledné pojetí je zcela na rozhodnutí experta. Liší se velikosti prvního písmene, použití mezer nebo podtržítek, obecný slovník («*not null*» vs. «*nullable*») i umístění («*PK*» u atributu nebo u procedury?).

Různá diagramová prostředí mohou mít zavedeny tzv. **profily** (popř. umožňovat uživateli vytvořit vlastní) které pro různé domény stanovují použitelné stereotypy. [6] Tyto se však rovněž napříč prostředími liší.

Obecně však problém absence jednotného přístupu zůstává i při nepoužívání stereotypů. Narážíme zde na fakt, že v UML zatím stále neexistuje diagram, který by byl přímo uzpůsobený pro databázové modely. K modelování databází je sice často používáno právě UML a diagramu tříd (případně diagramu komponent pro obecnější schéma), jak však přistupovat k primárnímu klíči? Explicitně ho zapsat, nebo implicitně uvažovat inkrementující celé číslo? Explicitně zapisovat cizí klíče, nebo se spokojit s implicitním deklarováním jejich existence pomocí existujících vazeb mezi tabulkami a jejich kardinalitami? Přístupy se liší expert od experta a stejně jako u stereotypů lze říci, že si formát každý přizpůsobí svým osobním potřebám a preferencím.

4 TYPY CÍLOVÝCH PROGRAMOVACÍCH JAZYKŮ

V kapitole 3 jsme se seznámili s diagramy vstupujícími do procesu generování kódu z modelů. Diagram tříd a relační databázový model (modelovaný za použití diagramu tříd) oba představují modely na implementační úrovni. Jejich struktura je tedy přenositelná na cílový programovací jazyk.

Každý z těchto diagramů popisuje jinou doménu, tedy i typ cílových jazyků pro tyto diagramy se liší. Z diagramu tříd generujeme objektově, především tedy třídně orientované jazyky (kapitola 4.1), zatímco relační databázový model vede na dialekty jazyka SQL (kapitola 4.2).

4.1 Objektově orientované programování

Objektově orientované programování (*object-oriented programming*, OOP) je programovací paradigma, které se od svého předchůdce, procedurálního paradigmatu, liší v přistupování ke kódu jako k množině vzájemně komunikujících objektů.

Objekty jsou chápány jako kódové struktury reprezentující nějaké fyzické či logické entity reálného světa. Jejich charakteristickým rysem je určitá forma **zapouzdření** *(encapsulation)* interních dat tohoto objektu a komunikace s dalšími objekty (nebo se sebou samým) pomocí zasílání tzv. zpráv. Data objektu se obecně označují jako **atributy**, procedury jejichž volání a vykonávání se chápe jako zasílání zpráv, se nazývají **metody**. [13]

4.1.1 Dědičnost

Dědičnost je charakteristickým rysem většiny objektových jazyků. Jedná se o způsob sdílení/předávání vlastností (atributů, metod) mezi objekty za účelem snadnější znovupoužitelnosti kódu a zpřehlednění.

Princip dědičnosti spočívá v umožnění specifičtějším objektům, aby používaly vlastnosti objektů obecnějších. Pro příklad si uvedeme dva objekty: Pes a Ares. Objekt Pes může být relevantní v části kódu, kde pracujeme s druhy zvířat, kde nás zajímají jen obecné vlastnosti těchto druhů (název, počet nohou). V jiné části kódu už nás však zajímají konkrétní domácí mazlíčci, u kterých chceme kromě jejich druhu vědět i jméno, pohlaví atp. U obou objektů by nás však zajímal název druhu, oběma objektům bychom chtěli být schopni zaslat zprávu "vydej zvuk" a obdržet "haf". Bez dědičnosti bychom museli v kódu napsat každý z těchto objektů zvlášť a některé části jejich kódu by byly navzájem duplicitní.

Máme-li však k dispozici dědičnost, můžeme objektu Ares stanovit nadřazený, rodičovský objekt Pes, jehož vlastnosti bude Ares dědit. Objekt Ares pak jistým způsobem převezme kód objektu Pes a dodá k němu svoji rozšířenou funkcionalitu.

Budeme-li pak potřebovat zjistit, jaký zvuk mazlíček Ares vydává, v jeho objektu se odvoláme na metodu přejatou od rodičovského objektu.

K dědičnosti se v OOP přistupuje dvěma způsoby: pomocí prototypů, nebo tříd. Prototypování řeší dědičnost klonováním rodičovského objektu (prototypu), dále se jím však v této práci nebudeme zabývat. Třídy si blíže představíme v následující podkapitole.

4.1.2 Třídně orientované jazyky

Třída (*Class*) v OOP představuje kostru; vzor, na jehož základě jsou objekty vytvářeny. Jazyky užívající konceptu tříd můžeme nazvat **třídně orientované jazyky** (C*lass-oriented languages*).

Každá třída má název a sadu atributů a metod. Objekty odvozené od nějaké třídy jsou chápány jako **instance** této třídy. Lze bezpečně očekávat, že se všemi instancemi téže třídy lze komunikovat stejným způsobem (shodné názvy atributů a zprávy, kterým tyto instance rozumí). Rovněž jakékoliv změny v kódu třídy se projeví na všech jejích instancích. Třídu lze tedy chápat jako šablonu, na jejímž základě jsou vytvářeny instance.

Speciálním případem jsou **abstraktní třídy**, které jsou explicitně navrženy jako rozhraní. Nelze na jejich základě vytvářet instance, protože obsahují tzv. **abstraktní metody**, tj. metody deklarované, ale nedefinované. Od těchto tříd lze pouze dědit a dědící třída musí abstraktní metody svého rodiče implementovat, jinak je sama chápána jako abstraktní.

Ve třídách se setkáme se dvěma typy atributů a metod. Instanční atributy jsou právě ty atributy, které se vytváří pro každou instanci, tyto je pak plní vlastními hodnotami. Třída však může mít i tzv. statické atributy, které se dají chápat jako globální atributy pro všechny instance této třídy. Instanční metody jsou metody náležící jednotlivým instancím, mají přístup jak k instančním, tak statickým proměnným. Statické metody náleží třídě jako celku, přistupovat mohou tedy pouze ke statickým atributům.

Některé jazyky umožňují označit instanční metody za konstantní (klíčové slovo `const`). Konstantní metoda je taková metoda, která nijak nemodifikuje data instance.

Dědičnost u tříd

Dědičnost u tohoto přístupu probíhá právě na úrovni tříd. Třída může dědit od jiné třídy (případně od více tříd) a sama obsahuje pouze ty atributy a metody, kterými rozšiřuje své předky. K těmto prvkům se pak v jednotlivých instancích potomka přistupuje explicitně jako k prvkům rodičovských tříd.

Modifikátory přístupu

Hierarchie vzniklá dědičností mezi třídami umožnila zavedení tzv. **modifikátorů přístupu** (*access modifiers*), které jsou u třídně orientovaných jazyků prostředkem pro dosažení zapouzdření. Atributy i metody dané třídy mají stanoveny své modifikátory, které určují, jaké objekty mají k těmto přístup.

Privátní (*private*) - k prvkům může přistupovat pouze jejich vlastní instance.

Chráněné *(protected)* - k prvkům mohou přitupovat instance dané třídy a všech jejích potomků (podtříd).

Veřejné *(public)* - prvky jsou volně přístupné napříč celým kódem.

4.1.3 Objektově orientované jazyky

Programovací jazyky, které používají paradigma objektově orientovaného programování se dají rozdělit do dvou hlavních skupin. Čistě objektové jazyky jsou takové jazyky, v nichž jsou všechny prvky jazyka chápány jako objekty. Takovými jazyky jsou např. Python, Smalltalk, Ruby, nebo Self. Lze sem zařadit rovněž JavaScript, který je nejznámějším zástupcem prototypových jazyků.

Druhou logickou skupinou OO jazyků jsou jazyky, které sice podporují objektové paradigma, některé jejich prvky však nesplňují ideu čistě objektových jazyků (např. jazyk Java, kde primitivní datové typy nejsou objekty). Může se jednat o jazyky původně procedurální, do nichž byl objektově orientovaný přístup integrován později (ať už jako modifikace původního jazyka, nebo tímto rozšířením vznikl jazyk nový). Jsou to např. C ++, C #, Java, PHP, Pascal, Fortran nebo Perl.

4.2 Jazyk SQL

SQL (*Structured Query Language*) je strukturovaný dotazovací jazyk navržený pro správu dat uložených v relačních databázových systémech.

Pod původním označením SEQUEL (Structured English Query Language) byl tento jazyk v roce 1974 představen Donaldem D. Chamberlinem a Raymondem F. Boycem z firmy IBM jako implementace Coddova relačního databázového modelu (kapitola 3.3). Později byl přejmenován z licenčních důvodů na SQL (značka SEQUEL již byla registrovaná leteckou společností *Hawker Siddeley*). Od roku 1987 je SQL standardem pod mezinárodní standardizační skupinou ISO (*International Organization for Standardization*). [14]

SQL je obecně považován za jazyk **deklarativní**, tj. jazyk, kde deklarujeme, jaké funckionality chceme docílit, aniž bychom se museli zabývat způsobem, jakým má být této funckionality docíleno. Pro jednotlivé databázové systémy (SŘBD, *systémy řízení báze dat*) však nad SQL existují procedurální nadstavby.

4.2.1 Příkaz `CREATE TABLE`

Jazyk SQL se skládá z několika jazykových prvků, které umožňují širokou škálu manipulace s daty. Tyto však nejsou relevantní pro tuto práci a nebudeme se jimi zde zabývat. Pro problematiku převodu UML diagramů na SQL nám postačí zaměřit se na definici základní struktury databáze, tj. tabulek a vztahů mezi nimi.

Pro vytvoření tabulky se v SQL používá příkaz `CREATE TABLE`. V příkazu uvádíme jméno tabulky a definujeme její sloupce, tedy pro každý z nich udáváme název,

datový typ (může zahrnovat rozsah), případně se zde mohou uvést některá ze základních integritních omezení (např. `PRIMARY KEY`, `DEFAULT`, `NOT NULL`). Tato se vztahují výhradně na daný sloupec. Nejsou-li součástí definic sloupců, uvádí se pak integritní omezení separátně za těmito definicemi. Některá integritní omezení (obecně taková, která zahrnují více než jeden sloupec) nelze uvést jinde (např. složené cizí a primární klíče).

Následující jednoduchý příklad ukazuje tvorbu tabulek Owner a Dog, kterými jsme se zabývali v kapitole 3.3.3.

```
CREATE TABLE Owner (
   ID INT NOT NULL,
   Name VARCHAR(20)NOT NULL,
   Gender VARCHAR(20),
   Occupation VARCHAR(30),
   PRIMARY KEY (ID)
);
CREATE TABLE Dog (
   ID   INT NOT NULL,
   Breed VARCHAR(50) NOT NULL,
   Name VARCHAR(50) NOT NULL,
   Gender VARCHAR(20),
   OwnerID INT,
   PRIMARY KEY (ID)
   FOREIGN KEY (OwnerID) REFERENCES Owner(ID)
);
```

4.2.2 Databázové systémy užívající SQL

Ačkoliv je SQL chápáno jako jeden jazyk, jeho syntax se napříč různými databázovými systémy mírně liší, což je odvislé od interní implementace těchto systémů. Z pohledu uživatele se často jedná o drobné syntaktické rozdíly (např. přítomnost závorek v definici primárních a cizích klíčů), výraznějším rozdílem mohou být např. datové typy, které jednotlivé SŘBD nabízí nad rámec standardu ISO SQL. Přenositelnost mezi těmito systémy je tedy možná jen do určité míry, obvykle v nejzákladnějších konstrukcích, které spadají právě pod ISO standard.

Mezi nejznámější SŘBD používající jazyk SQL patří Oracle, MySQL, PostgreSQL, Microsoft SQL Server nebo SQLite.

5 ANALÝZA EXISTUJÍCÍCH ŘEŠENÍ NA POLI PŘEVODU DIAGRAMŮ UML NA KÓD

5.1 Komerční řešení

5.1.1 Visual Paradigm

Visual Paradigm je software od stejnojmenné společnosti *Visual Paradigm International*. Jedná se o nástroj pro vývojáře, nabízí prostředky pro modelování a správu vývojového procesu, modelování UML je jednou z jeho funkcí. Existuje jako samostatná aplikace, ale k dispozici jsou i zásuvné moduly do známých IDE: Eclipse, NETBeans, Visual Studio.

Tento software lze zakoupit jednorázově, nebo platit měsíční předplatné. Cena variant podporujících převod UML na kód se pohybuje v řádech stovek dolarů za jednorázovou koupi a desítek dolarů za měsíční předplatné.

Visual Paradigm umožňuje převádět diagram tříd na jazyky Java a C ++ včetně opačného procesu, převodu kódu na diagram. Poskytuje i kontrolu a korekci datových typů.

5.1.2 UModel 2017

UModel 2017 je grafický modelovací nástroj od společnosti *Altova*. Umožňuje integraci do vývojových prostředí Eclipse a Visual studio. Jednorázová cena za produkt se pohybuje v řádech stovek dolarů.

UModel umožňuje převod třídních diagramů na jazyky Java, C # a Visual Basic .NET (s možností volby mezi několika verzemi každého jazyka) a rovněž podporuje opačný proces převodu jazyka na kód. Kód a diagram mohou být synchronizovány (tzv. *round-trip engeneering*). Při převodu diagramu na kód dochází k syntaktické kontrole. Podobně jako Visual Paradigm zde dochází také ke kontrole datových typů, pro jednotlivé jazyky pak UModel podporuje některé prvky specifické pro daný jazyk, např. podpora rozhraní Enumeration v jazyce Java.

Nástroj rovněž nabízí modelování relačních databází pomocí rozšířeného diagramu tříd (kapitola 3.1.3) a jejich převod do SQL dialektů následujících SŘBD: Microsoft SQL Server, Oracle, MySQL, PostgreSQL, IBM DB2, Sybase a Microsoft Access, pro většinu z těchto rovněž nabízí výběr z více verzí. Je zde umožňuje i připojení ke kterémukoliv z výše vypsaných SŘBD a převod existujících databází do UML.

5.1.3 Enterprise Architect

Komplexní modelovací prostředí firmy *Sparx Systems* s možnou integrací do Eclipse, Visual Studio, DOORS a Visio. Umožňuje modelování ve více doménách (kromě vývoje aplikací také automatizace, modelování databází, simulace). Cena se pohybuje v řádově ve stovkách dolarů za licenci.

Enterprise Architect nabízí generování kódu do množství OO jazyků (např. C ++, C #, Java, Delphi, PHP, VHDL, Python), uživateli je navíc umožněno definovat jazyk vlastní. Nástroj rovněž podporuje převod do SQL dialektů pro Oracle, MySQL, MS SQL Server, PostgreSQL, DB2, MS Access a Firebird.

5.1.4 BOUML

BOUML je grafický nástroj pro modelování UML, jehož autorem je francouzský vývojář Bruno Pagès. Cena se pohybuje v řádech desítek euro za roční licenci, za neomezenou licenci se platí řádově stovky euro.

BOUML umožňuje rozšiřování pomocí externích modulů, tzv. *plug-outs*, které mohou dodávat třetí strany. Nejvýznamnější plug-outy, které jsou součástí BOUML, poskytují právě generování kódu z diagramů tříd. Převod je podporován pro následující OO jazyky: C ++, Java, Idl, PHP a Python. BOUML rovněž umožňuje generování kódu pro MySQL.

5.1.5 Yatta UML Lab

Jedná se o modelovací prostředí od německé firmy *Yatta Solutions GmbH*, které se specializuje na propojení a synchronizaci modelů systému s vývojem kódu. Obdobně jako UModel 2017 nabízí funkci *round-trip engeneering*. Cena produktu na jeden rok se pohybuje v řádech stovek euro.

UML Lab se úzce zaměřuje na jazyk *Java*, převod do dalších OO jazyků není možný. Rovněž se nejedná o prostředí podporující vývoj databází.

5.1.6 StarUML 2

Původně Open Source projekt, který byl ve své době jedním z nejpoužívanějších volně dostupných diagramových prostředí. V roce 2005 se však vývoj zastavil a znovu se na něm pod názvem StarUML2 začalo pracovat o pět let později, tentokrát však jako na produktu s placenou licencí. Cena se pohybuje v řádech desítek dolarů.

StarUML2 podporuje generování kódu v jazycích Java, C ++ a C #. Umožňuje však třetím stranám přidávat vlastní rozšíření, která jsou dále k dispozici zdarma. Mezi těmito rozšířeními lze v současné chvíli najít podporu pro převod do dalších jazyků: Python, Ruby, JavaScript, PHP, Ruby a PostgreSQL.

5.2 Volně dostupná řešení

5.2.1 ArgoUML

Open Source diagramové prostředí, které plně běží na *Java Virtual Machine*, je tedy dostupné napříč všemi platformami podporujícími *Javu*.

V rámci ArgoUML existuje množství subprojektů, z nichž některé se zabývají rozšířeními pro převod UML do konkrétních jazyků. Podporovanými jazyky jsou: Java, C ++, C #, PHP, Ruby a SQL. Protože se jedná o nezávisle vyvíjené moduly, nabízená

funkcionalita se pro každý jazyk liší. Pro Ruby je např. možné pouze generovat kód z diagramu tříd, C # modul umožňuje i opačný proces.

Pro vývoj databází existuje modul DB-UML, který umožňuje tvořit relační databázový model pomocí variace na diagram tříd, specializované pro tento účel.

5.2.2 Moduly pro Eclipse

Eclipse je nejznámějším vývojovým prostředím pro jazyk Java. Jedná se o Open Source projekt, jehož původní kód byl roku 2001 zveřejněn firmou IBM. Podobně jako již zmíněná Open Source řešení se silně zaměřuje na možnost dynamicky přidávat další funkcionalitu pomocí zásuvných modulů. V současné době Eclipse nabízí širokou škálu modulů, které nejen rozšiřují jeho funkcionalitu v oblast vývoje v Javě, ale přidávají podporu pro množství dalších programovacích jazyků, z těch známějších můžeme jmenovat např. C, C ++, JavaScript, PHP, Python, Prolog, Perl či Ruby.

Existuje více modulů, které přidávají možnost modelovat v UML, a to jak komerční varianty (např. již zmíněné Visual Paradigm a UModel), tak varianty volně dostupné. Některé z těchto modelů přímo poskytují možnost převodu na kód v učitém jazyce (PyUML, Papyrus), případně existují další moduly, které je o tuto funkcionalitu rozšiřují. Papyrus má například vestavěné generování kódu v Javě a C ++, existují však moduly přidávající převod do Ruby.

5.3 Shrnutí

Existuje množství řešení umožňujících převod UML do některého programovacího jazyka, tato kapitola se zabývala významnějšími z nich.

Častěji je generování podporováno pro OO jazyky, z těchto nejčastěji Java a C ++. Generování SQL je vesměs doménou dražších komerčních řešení. Tato poskytují komplexní prostředí s širokým množstvím funkcí v doméně generování kódu z UML, včetně syntaktické kontroly či žádaného *round-trip engeneering*. Tyto jsou však často součástí dražších Enterprise či Professional edicí. Pouze Enterprise Architect umožňuje určitou možnost uživatelský přívětivého přidávání vlastního jazyka. Dosažení vlastních změn je u dalších komerčních řešení nereálné.

Open-source prostředí jsou závislá na jednotlivých modulech, které však mají kolísavou kvalitu. Množství dostupných modulů je obecně žádoucí, pokud však tyto řeší stejnou doménu, může být pro vývojáře únavné mezi nimi vybírat. V Eclipse je často generování kódu v konkrétních jazycích nadstavbou nad již existujícími moduly pro UML, čímž se nepřehlednost při výběru zvyšuje. Díky principu Open Source má uživatel možnost zasáhnout do vývoje, případně přidat vlastní jazyk, jedná se však o časově náročnou záležitost, protože prostředí zdrojových UML diagramů jsou poměrně robustní.

Všechny nástroje, kterými se tato kapitola zabývá, jsou grafické povahy. Neumožňují ovládat generování kódu pomocí příkazové řádky.

6 NÁVRH VLASTNÍHO NÁSTROJE PRO GENEROVÁNÍ KÓDU Z UML

V kapitole 5 jsme se seznámili s několika řešeními, která umožňují více či méně komplexní generování kódu z diagramu tříd. Shrnuli jsme si společné vlastnosti těchto nástrojů a na základě těchto poznatků nyní můžeme navrhnout vlastní nástroj. Cílem této práce je, aby navržený nástroj zapadal do rámce potřeb uživatele, ale zároveň přinášel do problematiky nový přístup. Nástroj bude umožňovat převod diagramů vytvořených v některém z volně dostupných nástrojů, které samy možnost generování kódu nenabízí.

Zvoleným nástrojem je diagramové prostředí Dia, které je intuitivní, umožňuje tvořit relační databázová schémata s pevně danou strukturou a zároveň jej lze ovládat pomocí dostupných modulů pro jazyk Python.

6.1 Diagramové prostředí Dia

Dia je volně dostupné, open-source diagramové prostředí. V základním grafickém módu se jedná o SDI rozhraní (*Single-Document Interface* - jedno okno zobrazí pouze jeden dokument), tvořit diagramy lze však do určité míry pomocí jazyka Python, pro jehož skripty má Dia zabudovanou podporu.

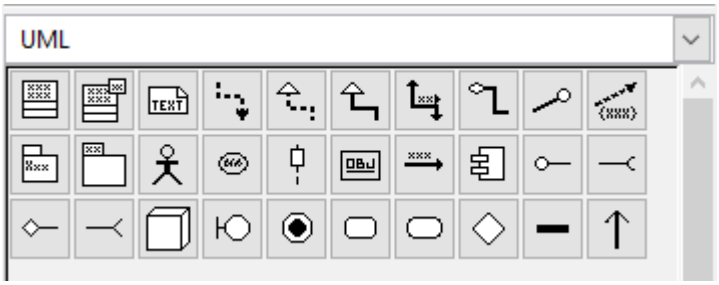
Dia nabízí modelování diagramů napříč mnoha doménami (např. stavebnictví, elektrické obvody, chemická schémata, sítě CISCO). V doméně softwarového inženýrství lze Dia využít pro tvorbu E -R diagramů, vybraných UML modelů či v úvodu zmíněných schémat relačních databází.

Dia lze považovat za odlehčené prostředí, ve kterém se snadno orientuje a o němž lze tvrdit, že ve svých doménách nabízí nezbytné minimum a nezatěžuje uživatele přílišným množstvím dodatečných voleb a nastavení. Nedochází zde k žádné kontrole správnosti použitých prvků dle příslušných standardů, uživatel má tedy volnou ruku a správnost diagramů či údajů v nich uvedených (např. datové typy) je zcela v jeho režii.

6.1.1 Mód UML a jeho použití při modelování diagramu tříd

Dia mód pro modelování UML nabízí panel s prvky několika UML diagramů, jmenovitě diagram balíčků, diagram případů použití, diagram objektů, diagram komponent a především diagram tříd.

Prvky relevantní pro diagram tříd se nacházejí na prvním řádku panelu, zcela vlevo prvek třída, následovaný třídou šablony a poznámkou (tyto nejsou v rámci této práce zpracovávány), zbývající prvky reprezentují vazby mezi třídami, jak jsou popsány v kapitole 3.2.2. Vlastnosti jednotlivých tříd lze specifikovat v okně vlastností třídy, které je přístupné intuitivně rozkliknutím dané třídy.



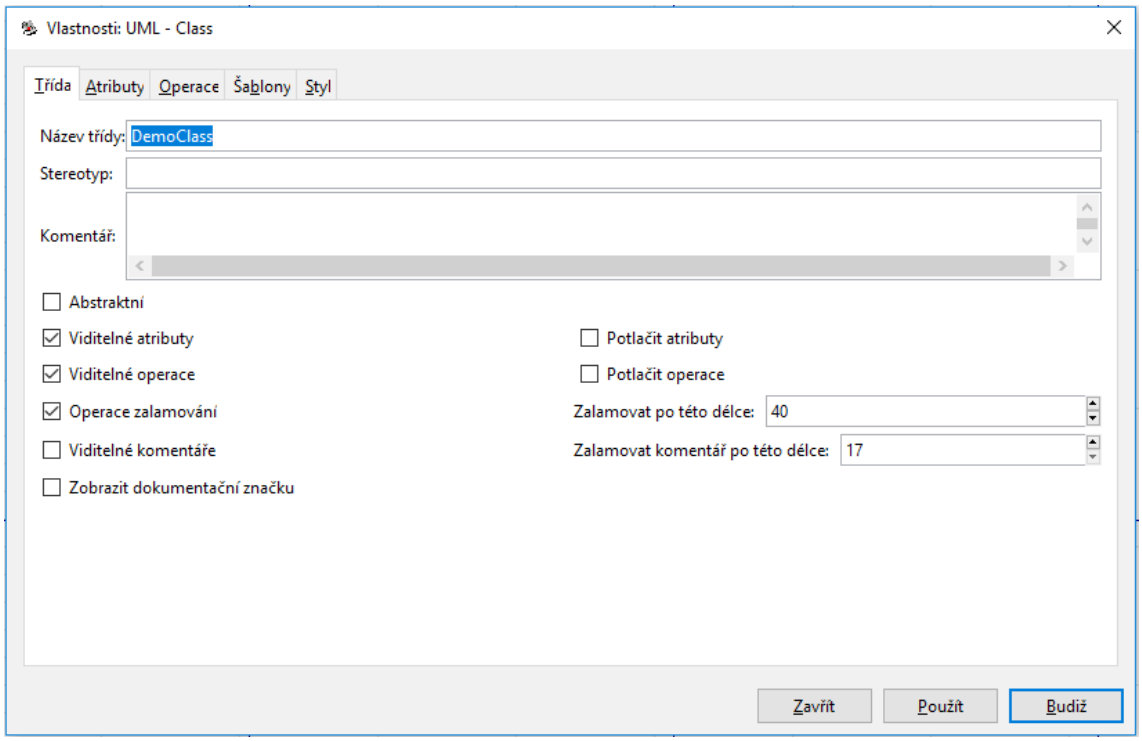
Obr. 10: Náhled na panel s UML prvky

V liště Třída (Obr. 11) lze definovat jméno třídy, stereotyp, nastavit příznak abstraktní třídy, dále se zde nacházejí možnosti úpravy vzhledu prvku.

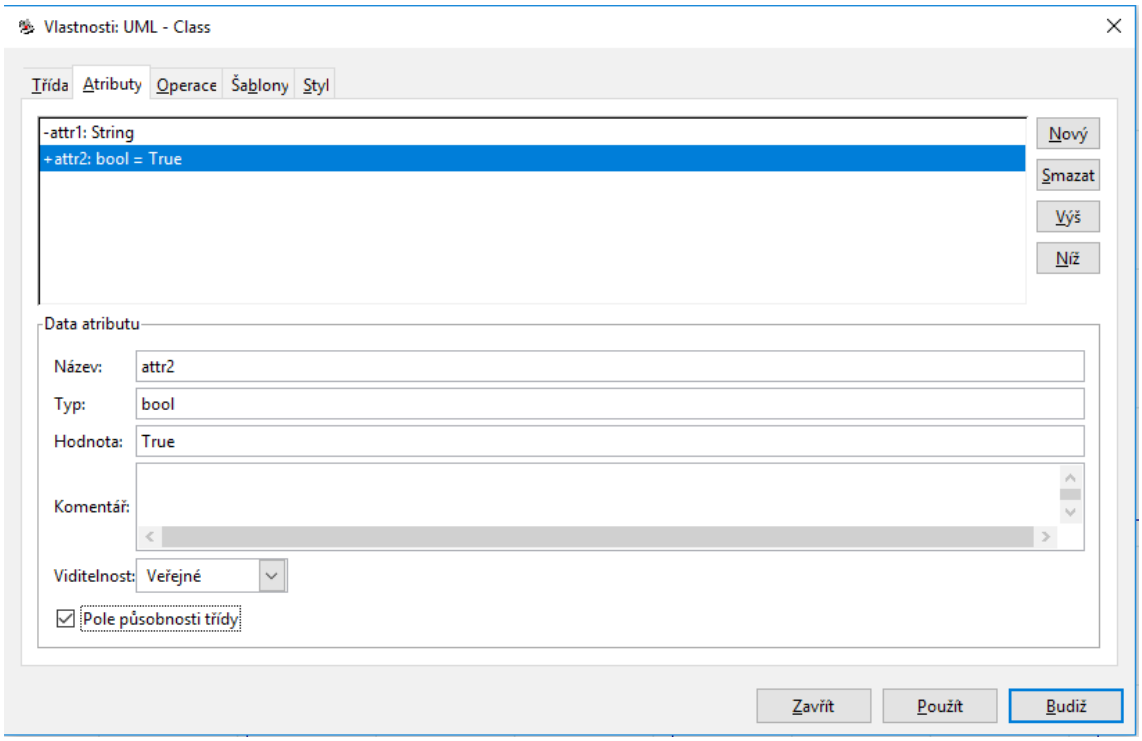
Lišta Atributy (Obr. 12) umožnuje přidávat a specifikovat jednotlivé atributy třídy. Každému atributu lze nastavit název, datový typ, možnou počáteční hodnotu, modifikátor přístupu (viditelnost), případně příznak "Pole působnosti třídy" (*class scope*), který je častěji znám jako příznak statické proměnné.

V liště Operace (Obr. 13) pak lze obdobně nastavit metody třídy včetně jednotlivých parametrů. Metodám může být navíc předřazen stereotyp, typ dědičnosti (abstraktní/virtuální/listová) a kromě příznaku statické metody je možné nastavit i příznak Dotaz (*Query*) značící konstantní metodu.

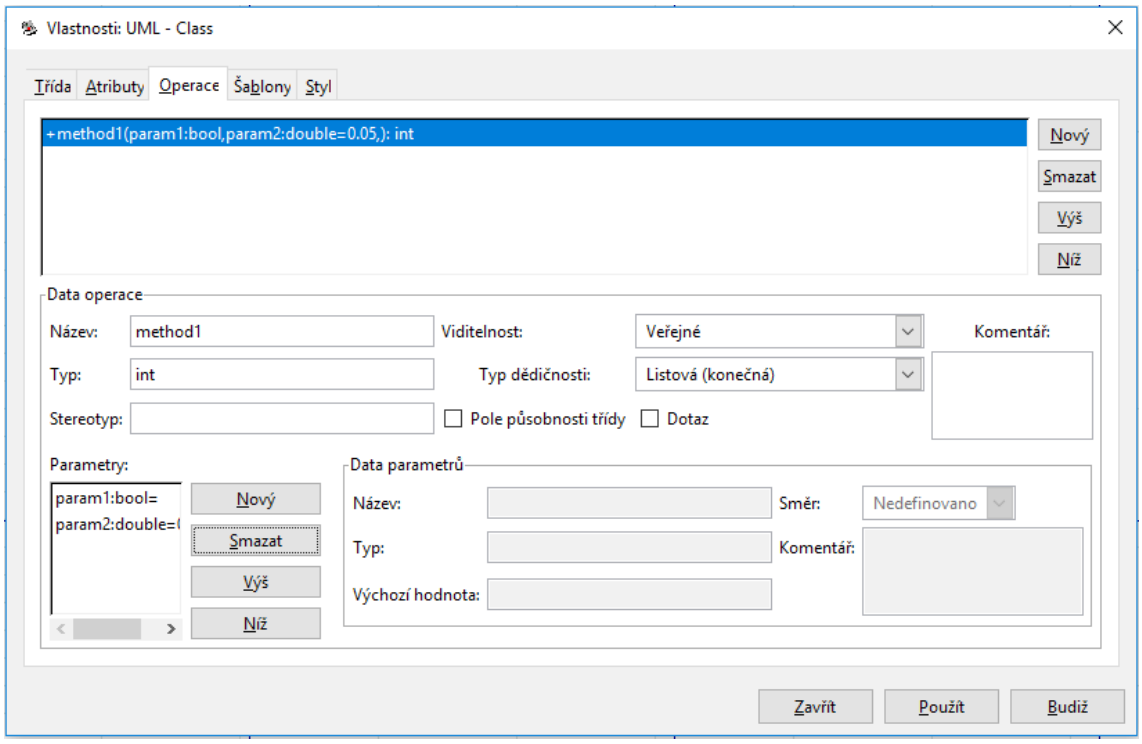
Lišta šablony obsahuje nastavení nad rámec této práce, Styl pak nabízí kosmetické úpravy zobrazení dané třídy.



Obr. 11: Vlastnosti prvku Třída - panel Třída



Obr. 12: Vlastnosti prvku Třída - panel Atributy



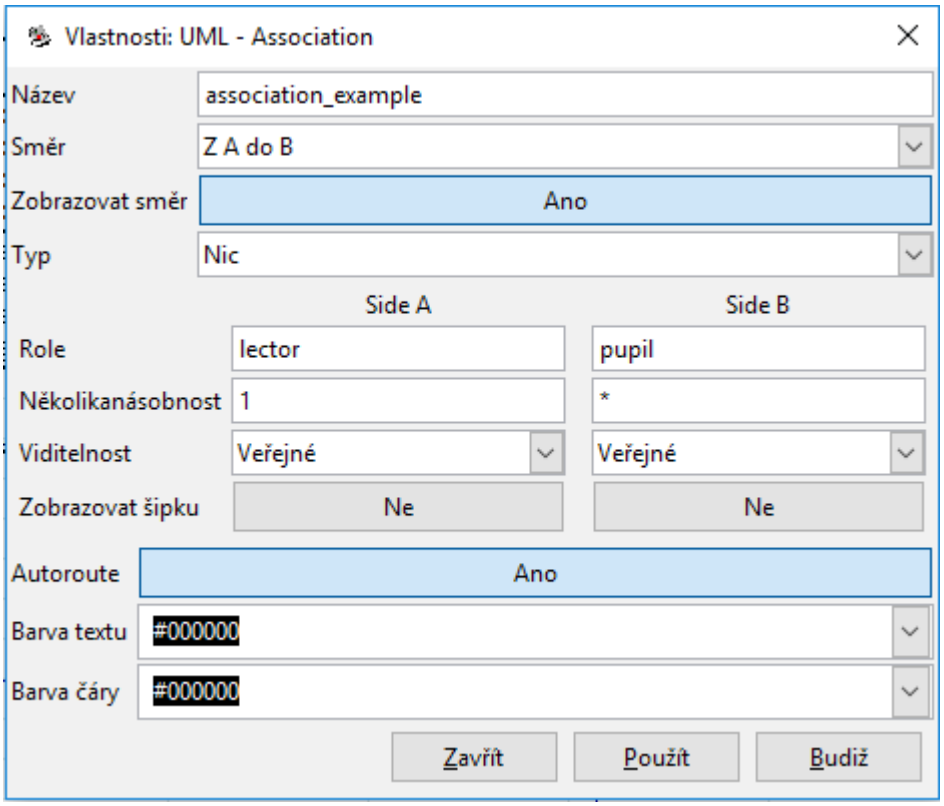
Obr. 13: Vlastnosti prvku Třída - panel Operace

Nabídku vlastností mají i jednotlivé vztahy mezi třídami. Vztahům typu generalizace (dědičnost), realizace a závislost lze měnit pouze kosmetické vlastnosti.

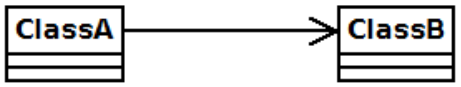
Agregace a kompozice jsou chápány jako varianty asociace, jejíž panel vlastností je možno vidět na Obr. 14.

Asociace lze v Dia modelovat pouze jako binární (případných n-árních variant lze docílit vhodnou kombinací více binárních). V možnostech jim lze nastavit název, směr a typ (Nic - žádný, Agregace, Kompozice). Role účastníků vztahu lze rovněž pojmenovat, nastavit jim četnost a viditelnost.

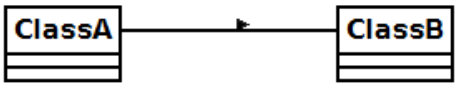
Určit směr asociace lze zobrazením šipek u jednotlivých účastníků (Obr. 15Obr. 15). Šipka obecného směru (Obr. 16) se používá pro určení směru čtení názvu asociace (např. v asociaci nazvané „je majitelem“ povede šipka od majitele k psovi). Zároveň se však slouží k určení pořadí konců asociace. [15] Dia však pořadí určuje jiným způsobem (viz 6.1.3).



Obr. 14 Panel vlastností Asociace



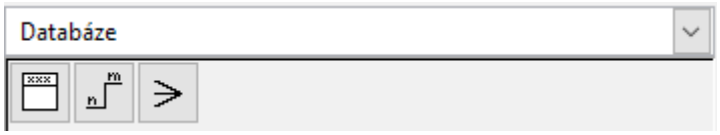
Obr. 15: A do B pomocí zobrazené šipky



Obr. 16: Směr čtení z A do B

6.1.2 Mód Database

Prostředí Dia, podobně jako ArgoUML, umožňuje modelování databázových schémat pomocí specializovaného diagramu. Tento je volně odvozen od diagramu tříd, ze kterého byly odstraněny vlastnosti nerelevantní pro relační databázové modely (i případná potřeba identifikovat jeho prvky pomocí stereotypů). Jak je patrné z Obr. 17, mód Databáze obsahuje pouze tři prvky (zleva doprava): tabulka, vztah a složený atribut.

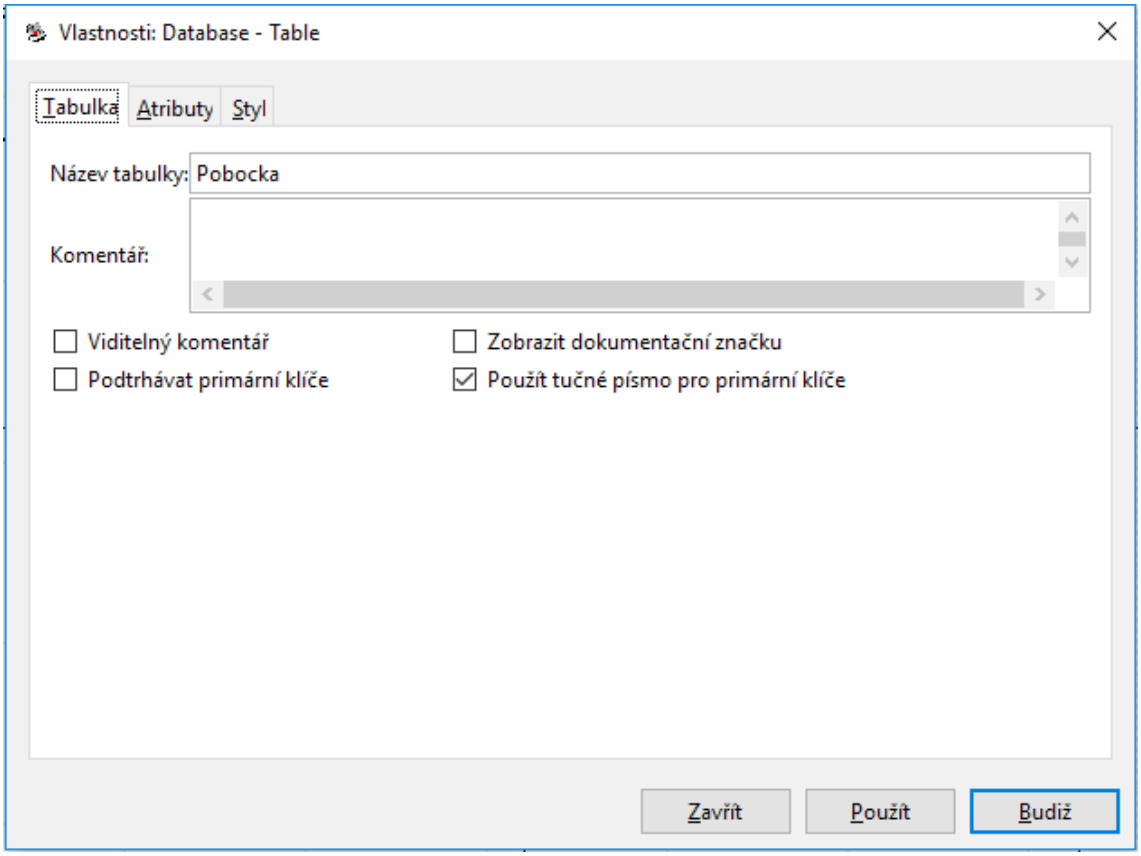


Obr. 17: Náhled na panel Databáze

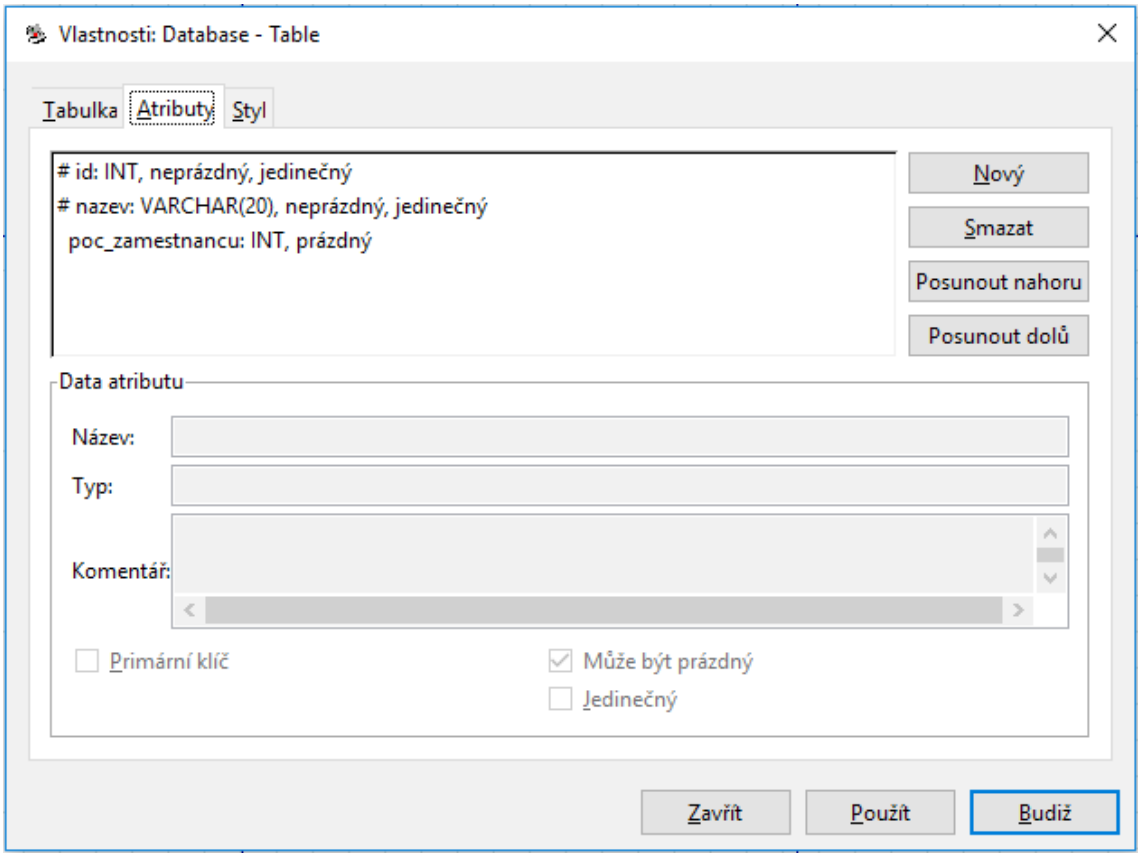
Prvku tabulka lze analogicky k třídě v UML nastavit obecné vlastnosti a přidat sloupce - atributy (viz Obr. 18 a Obr. 19). Jak je patrné z těchto obrázků, databázové tabulky jsou v množství nastavitelných vlastností minimalistické.

Vztahu mezi tabulkami lze přiřadit pouze kardinalitu jednotlivých účastníků. Použití složeného atributu není v dokumentaci Dia upřesněno a ani se ho v rámci této práce nepodařilo zprovoznit, nebude tedy uvažován.

Jak lze vidět na Obr. 19, Dia v tomto módu neumožňuje zadat sloupcům přednastavené `DEFAULT` hodnoty.



Obr. 18: Vlastnosti prvku Tabulka - panel Tabulka



Obr. 19: Vlastnosti prvku Tabulka - panel Atributy

6.1.3 XML reprezentace diagramů

Diagramy vymodelované v prostředí Dia jsou ukládány ve značkovacím jazyce Dia XML, standardně komprimované pomocí GZip (komprimaci lze vypnout).

Strukturu diagramu lze rozdělit na dvě části. Podstrom značky `diagramdata` obsahuje pouze údaje pro vizuální nastavení pracovního plátna. Data samotného diagramu se pak nachází v druhé části, podstromu značky `layer`.

Prvky diagramů jsou ukládány pod značkami `object`, jejichž atribut `type` určuje, o který prvek z kterého modelovacího módu se jedná. Např. *UML - Class*, *UML - Association*, *Database - Table*, *Database - Reference*. Vlastnosti těchto prvků jsou uloženy pod značkami `attribute` a jejich konkrétní povaha je dána řetězcem atributu `name`.

Následující příklad obsahuje ukázku jednoduché tabulky v Dia XML, která se nazývá Zamestnanec a obsahuje pouze jeden sloupec **id** datového typu INT, který je primárním klíčem. (Pro přehlednost byla vynechána data pro vizualizaci).

```
      <dia:object type="Database - Table" version="0 " id="O1">
      <dia:attribute name="name">
        <dia:string>#Zamestnanec#</dia:string>
      </dia:attribute>
      <dia:attribute name="comment">
```

```
        <dia:string>##</dia:string>
      </dia:attribute>
      <dia:attribute name="attributes">
        <dia:composite type="table_attribute">
          <dia:attribute name="name">
            <dia:string>#id#</dia:string>
          </dia:attribute>
          <dia:attribute name="type">
            <dia:string>#INT#</dia:string>
          </dia:attribute>
          <dia:attribute name="comment">
            <dia:string>##</dia:string>
          </dia:attribute>
          <dia:attribute name="primary_key">
            <dia:boolean val="true"/>
          </dia:attribute>
          <dia:attribute name="nullable">
            <dia:boolean val="false"/>
          </dia:attribute>
          <dia:attribute name="unique">
            <dia:boolean val="true"/>
          </dia:attribute>
        </dia:composite>
      </dia:attribute>
      </dia:object>
```

Jak lze z příkladu vypozorovat, data variabilního množství (atributy u databází a tříd, rovněž metody a parametry metod u tříd) jsou navíc subelementy značky `composite`.

Data na listové úrovni jsou vedena pod dvěma typy značek: `boolean` pro dvouhodnotové příznaky a `string` pro textová data. Řetězce jsou uzavřeny mezi dva znaky #, nejedná se však o výlučné zarážky; další výskyty tohoto znaku v řetězcích nejsou omezeny.

Role

Jedinečné ID je přiřazeno každému modelovanému prvku (tabulka v příkladu má *id = 01*). Prvky vztahů (dědičnost, asociace atp.) pak pomocí ID tabulek/tříd stanovují příslušnost jednotlivým rolím ve vztahu. **Roli A** lze u směrovaných vztahů (dědičnost, agregace, realizace atp.) chápat jako nadřazenou (*master*), **role B** je zde třída závislá (*slave).*

U implicitně nesměrovaných vztahů (klasická asociace) role představují interní rozdělení účastníků, aby jim bylo možné přiřazovat četnost, případně názvy rolí. Z hlediska interní reprezentace vždy platí, že role A je přiřazena tabulce, na jejímž příchytném bodě je vykreslování čáry vztahu započato.

Toto pojetí je problematické, protože prostředí Dia se zároveň snaží přidělit těmto rolím odpovídající šipku obecného směru (která byly popsána v kapitole 6.1.1). Zde však byla v rámci této práce v Dia nalezena chyba. Počáteční stav směru je vždy označen jako **A do B**, zobrazená šipka však vždy na počátku ukazuje zleva doprava a to i v případě, že třída A se nachází vpravo od třídy B. V takovém případě pak dochází k nekonzistenci mezi názvy rolí a popsaným směrem. Protože však šipka obecného směru slouží jen

k orientaci při čtení názvu asociace, nejedná se o problém, který by ovlivnil činnost navrhovaného nástroje.

Role nejsou přiřazeny jednotlivým prvkům explicitně. Rozdělení je odvozeno z atributů značek `dia:connection`. Atribut `handle` s hodnotou *0* značí počátek vztahové čáry, tedy roli A. Hodnota *1*značí koncový prvek, tedy roli B.

```
<dia:connections>
  <dia:connection handle="0 " to="O3" connection="13"/>
  <dia:connection handle="1" to="O1" connection="16"/>
</dia:connections>
```

6.2 Návrh nástroje Dia2Code

Na základě zvoleného prostředí pro tvorbu zdrojových diagramů a povaze zadání této práce, byl pro výsledný program zvolen název **Dia2Code**.

Bude se jednat o nástroj ovládaný z unixové příkazové řádky, který na vstupu přijme soubor v Dia XML a dle typu obsaženého diagramu (diagram tříd, databázový model) provede konverzi těchto dat na kód ve zvoleném cílovém jazyce.

Činnost Dia2Code bude možno rozdělit na dvě poloviny: **parser**, který bude obstarávat parsování dodaného XML a uložení získaných dat do navržených struktur, a **generátor**, který bude následně data z těchto struktur používat pro generování řetězců ve zvoleném jazyce.

6.2.1 Volba programovacího jazyka

Za jazyk pro vytvoření *Dia2Code* byl zvolen skriptovací jazyk *Python*. Vzhledem k již zmíněné podpoře ovládání Dia pomocí existujících modulů v *Pythonu* byla tato volba považována za vhodnou. Zde navržený nástroj pak bude moci být chápán jako rozšíření této skupiny modulů.

Python v současné době koexistuje ve dvou verzích, Python2 a Python3. S ohledem na budoucnost jazyka bude v Dia2Code používán Python3.

6.2.2 Zpracování vstupních diagramů ve formátu Dia XML

Při návrhu parseru je nutné uvažovat, že vstupní XML soubor může být dodán jak v komprimované, tak nekomprimované formě. Nástroj musí dokázat rozpoznat, jaký typ vstupního souboru byl dodán a případně ho dekomprimovat.

Python nabízí několik knihoven pro zpracování jazyka XML. V této práci bude použita knihovna `ElementTree`, přednostně její varianta `cElementTree`. Obě tyto varianty implementují shodné API, `ElementTree` je implementována čistě v Pythonu, zatímco `cElementTree` je rychlejší a méně paměťově náročná implementace v jazyce C. [16]

Protože typy vstupních diagramů pro Dia2Code jsou ve svých prvcích poměrně distinktivní, bude pro každý z těchto vytvořen specializovaný parser, aby bylo docíleno

rychlejšího zpracování. Tyto parsery budou dědit od společného předka. Obdobným způsobem budou moci být přidány parsery pro další typy Dia diagramů.

6.2.3 Návrh struktur pro uchování získaných dat

Data získaná ze zdrojového Dia XML bude nutné uložit do interních struktur programu. Pro tyto účely budou pro stěžejní prvky diagramů (tj. prvky mající vlastní nastavení) vytvořeny třídy, které budou analogicky ke zdrojovému diagramu uchovávat informace o jednotlivých prvcích. Informace o zbývajících, méně komplexních prvcích diagramů budou součástí těchto tříd (např. dědičnost, závislost).

Pro diagram tříd budou vytvořeny datové třídy `Class`, `Attribute`, `Method` a `Parameter`. Jednodušší, databázová schémata budou uchovávána pomocí tříd `Table` a `DbAttribute` (pozn.: třídy reprezentující atributy budou jedinečné pro každý typ diagramů, protože i jejich data se výrazně liší).

6.2.4 Návrh modulů pro generování kódu s ohledem na žádanou podporu přidání dalších jazyků

Jednou z význačných vlastností Dia2Code má být dle zadání modularita navržena tak, aby bylo umožněno přidávání dalších jazyků. Pro dosažení tohoto cíle byla zvolena strategie dvojího generátoru.

Modul (třída) centrálního generátoru bude mít na starosti koordinaci dat a tvorbu výsledného souboru. Podobně jako parser, i generátor bude existovat ve dvou specializovaných variantách, dědících od stejného předka.

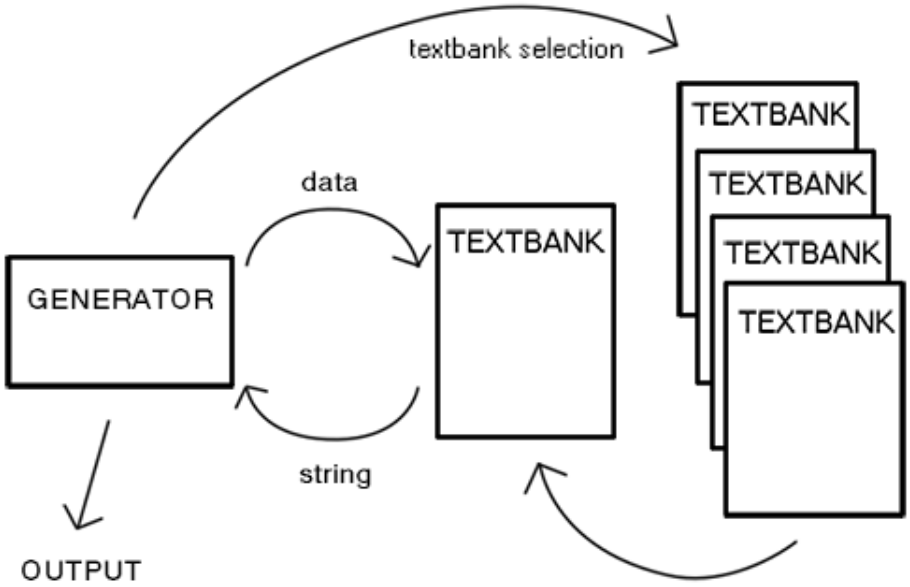
Moduly pro jednotlivé jazyky bude možno chápat jako zapouzdřené sub-generátory s jednotným API, jejichž úkolem bude přijímat dodaná dílčí data a vracet řetězce cílového jazyka, v nichž již budou tato data zaintegrovaná. Tyto jazykové moduly budou v rámci této práce nazývány termínem **textbanky**.

Centrální generátor tak pouze na počátku iniciuje textbanku uživatelem zvoleného jazyka, samotné generování z hlediska tohoto modulu bude probíhat jednotně.

Třídy textbank budou logicky rozděleny na část datovou (konstrukce v cílovém jazyce) a část generující (metody a atributy realizující generování + API metody pro komunikaci s centrálním generátorem).

Přidávání dalších jazyků bude umožněno právě díky tomuto jednotnému API jazykových textbank. Uživatel může k přidání dalšího jazyka využít jako mustr textbanky, které budou součásti Dia2Code (a využít vzájemnou podobnost OO jazyků, příp. SQL dialektů), může však celý algoritmus implementovat zcela dle vlastního uvážení. Obdobným způsobem bude možné přidat vlastní textbanky k již vypracovaným jazykům - uživatel může výstup změnit tak, aby více vyhovoval jeho potřebám, přidat další funkcionalitu atp.

Integrace textbank do zbytky programu bude probíhat jednoduše přidáním názvu texbanky do seznamu jazyků rozpoznávaných generátorem (a pochopitelně přidáním této textbanky do adresářové struktury programu).



Obr. 20: Schéma komunikace centrálního generátoru s textbankami

6.2.5 Výběr cílových jazyků

Dia2Code bude zpracovávat dva typy vstupních diagramů a pro každý z těchto bude nabízet tři cílové jazyky.

Záměrem této práce je ve své základní formě oslovit co nejvíce uživatelů. Při výběru cílových jazyků bude tedy dána přednost jazykům rozšířeným před jazyky neobvyklými (tyto budou chápány jako jedna z mnoha cest pro rozšíření funkcionality programu).

Jak vyplynulo z analýzy existujících řešení v kapitole 5.3, při generování kódu z diagramu tříd jsou nejčastěji poskytované cílové jazyky Java a C ++. Podle statistiky *IEEE Spectrum* [17], online magazínu organizace IEEE, jsou mezi pěti nejoblíbenějšími programovacími jazyky tři objektově orientované, a to právě Java, C ++ a také Python. Tyto tedy budou zvoleny jako cílové jazyky i v této práci.

Co se týče nejpoužívanějších databázových systému, bylo čerpáno ze statistik iniciativy DB-Engines [18]. První dvě pozice jsou dlouhodobě stabilní, jedná se o SŘBD Oracle a MySQL, tyto budou použity i v Dia2Code. Dále pak (navzdory čtvrté pozici) PostgreSQL, jehož popularita dle statistiky roste. PostgreSQL byl rovněž zvolen kvůli možné návaznosti na bakalářskou práci autorky této diplomové práce a skutečnosti, že se jedná o Open source.

7 IMPLEMENTCE NAVRŽENÉHO ŘEŠENÍ

Tato kapitola se bude zabývat vlastní implementací řešení navrženého v kapitole 6. Čtenáři bude nejprve představeno základní členění zdrojových souborů programu do adresářové hierarchie spolu se stručným nástinem, jaké moduly se v kterém adresáři nachází. Následující podkapitoly se budou postupně zabývat konkrétním řešením stěžejních logických částí kódu tak, jak byly představeny v návrhu.

Součástí této kapitoly je rovněž seznámení s ovládáním nástroje pomocí parametrů příkazové řádky a popis činnosti modulu pro řešení chyb včetně návratových hodnot programu.

7.1 Adresářová struktura

Uložení zdrojových kódů programu je následující: za kořenový adresář je považován adresář `/src/dia2code`. Zde se nachází hlavní, spustitelný modul `dia2code.py`, modul pro řešení chyb `error_handler.py` a hlavní adresářová struktura programu.

`classd`	Zdrojové kódy pro zpracování diagramu tříd. Jejich názvům je předřazena předpona "`cls_`". Nachází se zde specializované parser a generátor, dále pak třídy pro uložení dat z Dia XML.
`config`	Moduly pro konfiguraci a zpracování argumentů z příkazové řádky.
`databased`	Analogicky k adresáři `classd`. Zpracování databázových schémat. Předpona souborů "`db_`".
`factory`	Třídy s továrními metodami pro volbu parseru a generátoru.
`parents`	Abstraktní třídy, od kterých dědí specializované parsery, generátory a textbanky.
`textbanks`	Rozdělen na podadresáře `classd` a `databased` obsahuje textbanky cílových jazyků.

Tab. 2: Základní adresářová struktura dia2code

7.2 Parametry příkazové řádky

Nástroj Dia2Code je zcela ovládán pomocí příkazové řádky. Jeho činnost je upravována dvěma pozičními a pěti volitelnými parametry. Přijetí těchto je prováděno pomocí standardní Python knihovny `argparse`, která kromě definování požadovaných parametrů umožňuje i snadné generování a výpis nápovědy. Nastavení parametrů pomocí argparse je řešeno v modulu `/config/configure.py`.

Použití programu je následující:

```
dia2code.py mode src [-h][-dst DESTINATION][-l LANGUAGE]
[-n NAME][-p PRINT]
```

7.2.1 Poziční parametry

Poziční parametry je nutné při spuštění programu zadat a, jak jejich název napovídá, je tak nutno učinit ve stanoveném pořadí, protože jim nepředchází specifický přepínač.

Pro činnost Dia2Code byly stanoveny dva parametry, jejichž přítomnost je ke správné činnosti programu nutná:

`mode`	Poziční parametr, který určuje, v jakém převodním módu bude program pracovat. Je zadáván jedním znakem, od volitelných přepínačů se liší absencí pomlčky. Lze jej zadat dvěma způsoby: znak "`c `" určuje převod diagramu tříd na kód v OO jazyce, "`d `" pak značí převod databázového schématu na dialekt SQL.
`src`	Cesta k souboru se zdrojovým diagramem v Dia SQL.

Tab. 3: Parametry příkazové řádky

7.2.2 Volitelné parametry

Volitelné parametry umožňují blíže specifikovat činnost programu. Jejich přítomnost však není vyžadovaná, protože program má přednastavené chování v případě jejich absence.

Tyto parametry lze při spuštění programu umístit libovolně, protože jim předchází unikátní přepínač, který jasně stanovuje povahu následujícího parametru.

Dia2Code podporuje čtyři volitelné parametry plus parametr pro vypsání nápovědy, který má specifickou povahu a při jeho použití není vyžadována přítomnost pozičních parametrů.

`-h, --help`	Vyvolá vypsání nápovědy a ukončí program. Nápověda obsahuje stručný úvod k programu a výpis parametrů.
`-dst DESTINATION, --destination DESTINATION`	Tento přepínač umožňuje specifikovat umístění, na které bude uložen výsledný soubor/soubory. DESTINATION přijímá cestu k cílové složce. Není-li použit, výsledný soubor je uložen do aktuální složky.
`-l LANGUAGE, --language LANGUAGE`	Volba cílového jazyka. Vázáno na poziční parametr `mode`: podporované jazyky se liší pro různé typy zdrojových diagramů. Není-li tento parametr použit, implicitně se generuje kód v jazyce MySQL, resp. C ++.
`-n NAME , --name NAME`	Umožňuje explicitně zvolit název výstupního souboru. Je-li zvoleno ukládání do více souborů, bude tento název tvořit kořen názvů těchto souborů. Není-li tento parametr použit, použije se název zdrojového souboru. **Pozn:** Jméno je zadáváno bez přípony, tato je dodána programem.
`-p PRINT --print PRINT`	Specifikuje kam a jak má být výsledný soubor vypsán. Přijímá tři varianty: "`t `" pro tisk na terminál, "`f `" pro tisk do souboru, "`ff`" pro tisk do více souborů (aplikovatelné pouze pro diagram tříd, jinak je chápán jako "`f `"). Není-li specifikováno, implicitně se uvažuje nastavení "`f `".

Tab. 4: Volitelné parametry řádky

7.2.3 Kontrola validity zadaných parametrů

Parametry příkazové řádky přijaté při spuštění programu prochází kontrolou v modulu `/configure/args_checker.py`. Pro každý možný parametr zde existuje "checker" funkce, které je jako argument zaslán obsah daného parametru ve formátu String a to i v případě že tento parametr zadán nebyl (volitelné parametry) - což je reprezentováno prázdným řetězcem.

Jednotlivé funkce pak kontrolují správnost dodaného parametru, případně jej upraví pro další použití (např. unifikace formátu cesty k cílové složce). Není-li volitelný parametr zadán, dochází zde k nahrazení prázdného řetězce implicitním nastavením. Kontrolní funkce pak tento parametr opět vrací jako návratovou hodnotu.

Je vhodné zde zmínit funkci `checkLanguage)`, která při kontrole zadaného cílového jazyka přistupuje ke slovníku podporovaných jazyků. Tento se jako třídní proměnná nachází ve třídě daného specializovaného generátoru (více v kapitole 7.5).

Není-li některý z dodaných parametrů validní, je vypsána příslušná chybová hláška a program je ukončen. Více o modulu zpracování chyb bude vysvětleno v kapitole 7.7.

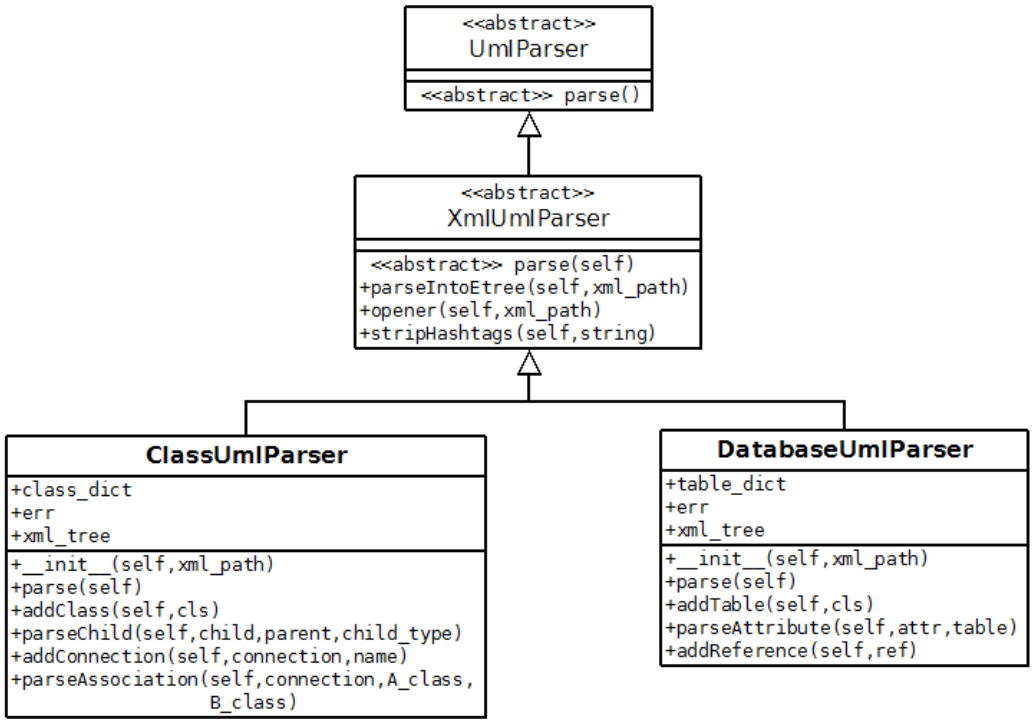
7.3 Parser

Jak bylo stanoveno v návrhu, parsování vstupního DiaXML bylo implementováno jako hierarchie tříd, která má na listové úrovni parsery specializující se na parsování konkrétních typů diagramů. Tato hierarchie je znázorněna na fragmentu diagramu tříd (Obr. 21).

Kořenovou třídou této hierarchie je abstraktní třída UmlParser, která obsahuje pouze abstraktní metodu `parse()`. Tato metoda představuje rozhraní pro komunikaci se všemi parsery, je volána z procedury `main()`. Protože je maximálně obecná, mohou od ní dědit i jiné typy parserů, než jaké jsou implementovány v rámci této práce (např. jako rozšíření).

Třídu UmlParser dále specializuje třída XmlUmlParser, která je rovněž abstraktní, obsahuje však již metody nutné pro otevírání a parsování Dia XML. Tato třída již pevně určuje povahu přijímaných diagramů.

Od třídy XmlUmlParser dědí již konečné, plně specializované parsery: DatabaseUmlParser a ClassUmlParser.



Obr. 21: Hierarchie tříd parseru

7.3.1 Třída `XmlUmlParser`

Tato třída obsahuje kromě abstraktní metody `parse()` tři implementované metody, které poskytují operace společné pro všechny parsery zpracovávající některý diagram z prostředí Dia.

Metoda `parseIntoEtree()` používá standardní knihovnu ElementTree ke zpracování zdrojového XML do stromové struktury, kterou následně vrací.

K otevření souboru slouží metoda `opener()`, která pomocí prvních bytů souboru určí, zdali byl komprimován pomocí gzip (sekvence 1F 8B) . Pokud ano, soubor je čten pomocí vestavěné knihovny gzip.

Metoda `stripHashtags()` poskytuje jednoduchou operaci odstranění znaků #, které ohraničují řetězec XML elementu `string`. Zpracování veškerých dat z těchto elementů nejprve prochází touto metodou.

7.3.2 Třída `ClassUmlParser`

Třída specializovaného parseru pro zpracování diagramu tříd. Při inicializaci používá k otevření zdrojového souboru metody své rodičovské třídy `XmlUmlParser`, vrácenou stromovou strukturu ukládá do atributu `self.xml_tree`. V atributu `self.class_dict` si vede slovník dosud vytvořených a naplněných instancí třídy

Class tak, že indexem k těmto třídám je jejich ID, které jim bylo přiřazeno v prostředí Dia. Jádrem této třídy je implementace metody `parse()`.

Metoda `parse()`

Tato metoda řídí průběh parsování dat ze struktury uložené v atributu `self.xml_tree`. Metodu lze rozdělit na tři části.

V první části jsou v cyklu nacházeny elementy stromu odpovídající značce `object` s typem "UML - Class". Pro každý takový element je volána metoda `add_class()`, která zajišťuje plnění slovníku `self.class_dict`.

V druhé části dochází ke kontrole tohoto slovníku. Je-li po průchodu prvním cyklem stále prázdný, značí to, že zdrojový soubor nelze považovat za kompatibilní s tímto módem, protože neobsahuje žádný stěžejní prvek Dia diagramu tříd, tj. třídy. V takovém případě je program ukončen s chybovým výpisem.

Ve třetí části je opět v cyklu procházen zdrojový strom, tentokrát jsou z něj však vybírány prvky reprezentující vztahy mezi třídami. Konkrétně tedy značky typu `object` s typy "UML - Association", "UML - Generalization", "UML - Dependency" a "UML - Realizes". Pro každý nález volá metody `addConnection()` s patřičnými parametry.

Po ukončení druhého cyklu je celý diagram zpracován a data uložena v `self.class_dict`.

Metoda `add_class()`

V rámci této metody dochází k instanciaci objektu třídy `Class`, jehož atributy jsou dále plněny. Jako parametr je přijímán podstrom nesoucí data k jedné třídě zdrojového diagramu.

Metoda nejdříve zpracovává obecná data vztahující se k dané třídě: jméno, stereotyp, komentář a příznak, že se jedná o abstraktní třídu.

V druhé části jsou dvěma cykly zpracovány nejdříve atributy, následně metody dané třídy. Pro každý z těchto je volána metoda `parseChild()` s vhodnými parametry.

Po naplnění nové instance je tato přidána do slovníku `self.class_dict`.

Metoda `parseChild()`

Metoda zajišťující uložení vlastností variabilních prvků třídy (atributy, metody, parametry metod). Jako argumenty přijímá zpracovávaný element, jeho typ a rodičovskou třídu.

Jak bylo rozebíráno v popisu struktury Dia XML, faktická (ne kosmetická) data těchto prvků jsou uložena pod značkami `boolean` a `string`. Metoda `parseChild()` tedy prochází všechny potomky dodaného elementu a odpovídá-li jejich název některému z výše uvedených, ukládá jeho data do lokálního slovníku `attr_dict` tak, že indexem je název tohoto elementu (např.: `attr_dict["abstract"] = True`).

Teprve po naplnění `attr_dict` dochází k určení, zdali se jedná o vlastnosti atributu nebo metody, na jehož základě je instanciován objekt vhodné třídy. Slovník `attr_dict` je tomuto objektu předán k inicializaci.

Parametry metod jsou zpracovány odlišně, protože se jedná o variabilní elementy vnořené do dalšího variabilního elementu (metody). Při nalezení elementu `composite`, který obsahuje argumenty metody, tedy dochází k opětovnému volání `parseChild()` pro tyto argumenty (rekurze první úrovně). Vytvořené instance třídy Parameter jsou ukládány do seznamu a následně použity k instanciaci mateřské metody.

Metoda `addConnection()`

Tato metoda provádí analýzu elementu vztahu, který dostává parametrem. Dochází k určení rolí A a B , dále ID tříd, kterým tyto role náleží. Podle povahy konkrétního vztahu (dědičnost, závislost, realizace) pak dochází k aktualizaci patřičných příznaků u těchto tříd.

V případě, že se jedná o asociaci, která nese komplexnější informace, dochází k předání jejího dalšího zpracování metodě `parseAssociation()`.

Spojení mezi třídami v diagramu nemusí být řádně uzavřeno (ačkoliv vizuálně se jeví v pořádku). Pokud je tento stav detekován, program je ukončen s chybovým hlášením.

Metoda `parseAssociation()`

V této metodě dochází ke zpracování informací, které nese element vztahu asociace. Vytvořená instance třídy Association je plněna svým názvem, směrem, případnou specifikací asociace (agregace, kompozice) a vlastnostmi jednotlivých rolí (název, četnost, zobrazená šipka).

Dochází také ke stanovení směru asociace na základě zobrazených šipek u jednotlivých rolí. Zjištěný směr (který stanovuje vzájemnou viditelnost mezi účastnickými třídami) je pak použit pro vhodné přidání této nové instance do seznamů asociací patřičných tříd. Nejsou-li šipky zobrazeny, asociace je považována za obousměrnou (tj. obě třídy o sobě navzájem ví).

7.3.3 Třída `DatabaseUmlParser`

Analogická třída ke `ClassUmlParser`, ovšem výrazně zjednodušená, protože i databázová schémata vymodelovaná v prostředí Dia jsou méně komplexní než diagram tříd.

Metoda `parse()`

Obdobně jako její třídní protějšek, i tato metoda má tři logické části.

V první části dochází k vyhledávání stěžejních prvků diagramu (zde tabulky), instanciaci objektů třídy Tabulka, jejich plnění pomocí metody `addTable()` a následné přidávání naplněných instancí do slovníku `self.table_dict`. Nebyla-li nalezena žádná tabulka, program je ukončen s chybovým výpisem.

V dalším cyklu jsou nacházeny reference, které jsou předávány metodě `addReference()`.

Metoda `addTable()`

Tato metoda prochází podstrom dodané tabulky a získanými daty plní novou instanci třídy Table. Nejprve dochází k nalezení obecných vlastností tabulky (zde pouze jméno a komentář), následně jsou v dalším cyklu řešeny atributy tabulky metodou `parseAttribute()`.

Pokud je nově zpracovaný atribut primárním klíčem, je přidán do seznamu primárních klíčů tabulky.

Metoda `parseAttribute()`

Zjednodušená analogie k metodě `parseChild()`. Tabulky obsahují pouze jeden typ variabilních elementů, a to jsou atributy (sloupce). Zpracování vlastností těchto je tedy přímočaré, není nutné rozlišovat, kterou třídu instanciovat, případně řešit vnořené variabilní elementy jako u `parseChild()`.

Metoda `parseAttribute()` ukládá data pod značkami `boolean` a `string` do lokálního slovníku `attr_dict`, tento pak předává nové instanci třídy Attribute při inicializaci (pozn. jedná se o třídu db_attribute.Attribute).

Metoda `addReference()`

Vztah reference mezi třídami je implicitně směrovaný (jedná se o vztah 1:n). Je tedy nutné určit role A a B , aby bylo možné stanovit, která třída odkazuje na kterou.

Metoda `addReference()` u odkazující třídy (kardinalita n, role B) volá její metodu `addForeignKey()` a předává jí parametrem třídu A. U odkazované tabulky (kardinalita 1, role A) je volána metoda `addSlave()`.

7.4 Třídy pro uchování dat

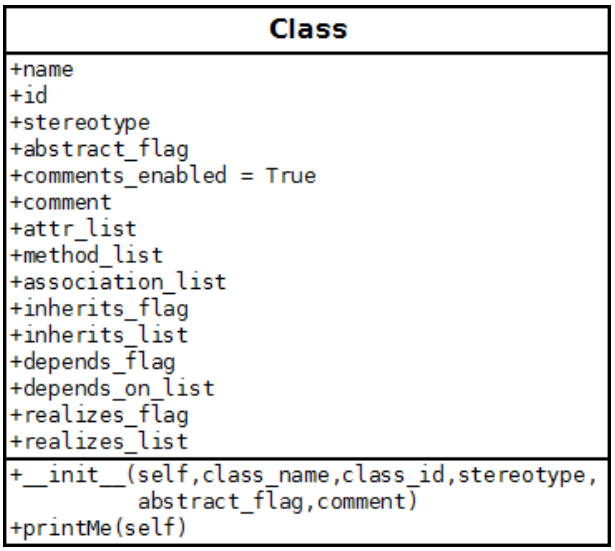
Pro konkrétní typ diagramu vždy existují jeho vlastní datové třídy. Jsou uloženy v adresářích `/classd`, resp. `/databased` a jejich náležitost k určitému módu je rovněž dána předponou jejich zdrojového souboru: "`cls_`" pro diagram tříd, "`db_`" pro databázový model.

V této kapitole budou nejprve představeny datové třídy diagramu tříd, následně datové třídy databázového modelu.

7.4.1 Třída `Class`

Třída `cls_class.Class` (dále jen `Class`) je primární třída pro uchování dat z diagramu tříd. Při inicializaci je plněna jen částečně a to svými obecnými vlastnostmi (jméno, ID, stereotyp, komentář, příznak abstraktní třídy), které jsou při vytváření objektů této třídy již známé (jak vyplývá ze struktury Dia XML a algoritmu parsovacích metod třídy `ClassUmlParser`.)

Stěžejními prvky každé třídy jsou její atributy, metody a vztahy, kterých se účastní. Pro jednoduché vztahy (dědičnost, závislost, realizace) ve třídě `Class` existují dvojice atributů příznak + seznam tříd druhé strany. Atributy, metody a asociace jsou jako instance svých tříd ukládány do příslušných seznamů.



Obr. 22: Třída Class

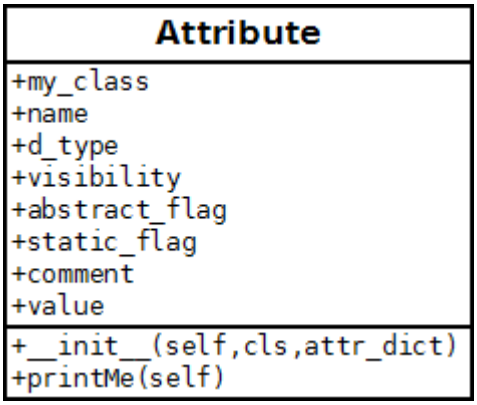
7.4.2 Třídy `Atribute`, `Method` a `Parameter`

Třídy pro ukládání dílčích dat třídy `Class` jsou zcela plněny při inicializaci. Jako parametr je jim předáván `attr_dict`, slovník atributů daného prvku ze zdrojového diagramu. Těmito jsou pak naplněny jim ekvivalentní atributy nově instanciovaného objektu.

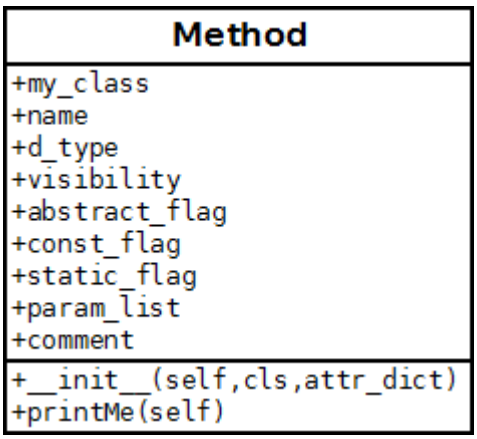
Jak bylo vysvětleno u metody `parseChild()`, zpracování všech parametrů metody (tedy instanciace a naplnění objektů třídy `Parameter`) dochází před instanciací samotné metody, seznam parametrů je tedy již součástí dodaného `attr_dict`.

Třídy `Attribute` a `Method` přijímají ještě další parametr, objekt `Class` mateřské třídy. Třída `Parameter` má na tomto místě podtržítko, které v Pythonu značí zanedbatelný atribut. Toto je v důsledku zpracovávání všech těchto prvků (atributy, metody i parametry) pomocí téže metody (`parseChild()`), která pracuje s proměnnou

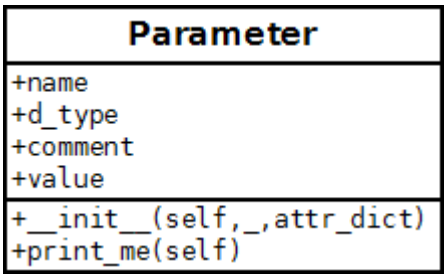
`parent`, ačkoliv ve třídě `Parametr` se nadřazený objekt neuvažuje, protože v momentě inicializace parametru ještě není znám (viz výše).



Obr. 23: Třída Attribute



Obr. 24: Třída Method

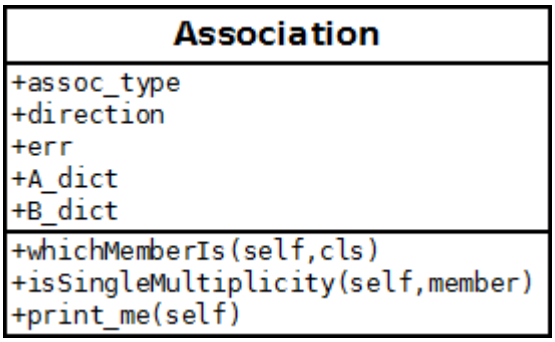


Obr. 25: Třída Parameter

7.4.3 Třída `Association`

Třída `Attribute` je podobně jako třída `Class` plněna až po inicializaci. Stěžejními atributy jsou slovníky `self.A _dict` a `self.B _dict`, které obsahují indexy "class", "role", "multiplicity", "visibility" a "arrow_visible", připravené pro uložení informací týkajících se jednotlivých rolí.

`Association` narozdíl od ostatních datových tříd rozebraných v předchozích podkapitolách obsahuje i několik metod (vyjma testovací `print_me()`). Ve vztahu asociace může při modelování dojít k problémům, které musí být ošetřeny, aby mohl generátor následně správně pracovat.



Obr. 26: Třída Association

Metoda `isSingleMultiplicity()`

Tato metoda je používána generátorem k určení, jestli v dané třídě (dodané parametrem) budou instance třídy z opačné strany vztahu uchovávány v jedno hodnotové proměnné, nebo ve vektoru/seznamu. Musí tedy dojít ke kontrole uvedené četnosti na protilehlé straně vztahu, než na které se nachází dodaná třída.

Je-li četnost popsaná číslem 1, případně není nastavena vůbec, je to považováno za četnost typu "single", která je značena návratovou hodnotou 0. Jakákoliv jiná syntakticky správná četnost je považována za "variable" a metoda při jejím zjištění vrací 1.

Syntaktická kontrola četnosti se odvíjí od standardního formátu zadávání četnosti. Příklady povolených formátů jsou uvedeny v Tab. 5. Variabilní množství je udáváno pouze znakem "*", další alternativy (n, m..n) nejsou povoleny.

1
5
*
0..10
1..*

Tab. 5: Příklady povolených formátů četnosti

7.4.4 Třída `Table`

Třída sestává především z několika seznamů, které rozřazují atributy tabulky jejich rolím. Nachází se zde obecný seznam atributů `self.attr_list`, dále `self.p _key` pro primární klíč tabulky (pokud PK není složený, obsahuje tento seznam jediný prvek) a `self.f _key_attr_list` obsahující seznamy atributů, které jsou cizími klíči. Pro oboustrannou provázanost tabulek (pro přehlednost a kvůli testování) se zde nachází i seznam tabulek, které na tuto tabulku odkazují, `self.referenced_by_list`.

Třída obsahuje dvě metody, které zapouzdřují akce potřebné při plnění některých z těchto seznamů.

Metoda `addForeignKey()`

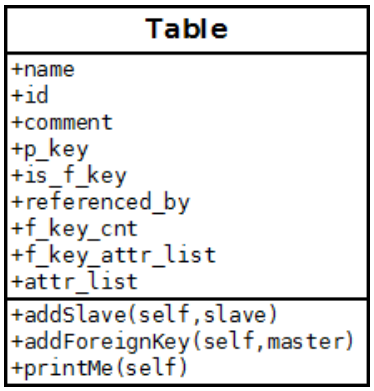
Jako parametr přijímá odkazovanou tabulku. Její seznam `p _key` přidá do svého seznamu `f _key_attr_list` a inkrementuje počítadlo cizích klíčů.

Metoda `addSlave()`

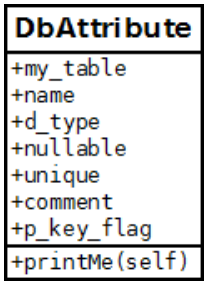
Tabulku přijatou v parametru ukládá do `self.referenced_by_list` a inkrementuje počítadlo.

7.4.5 Třída `DbAttribute`

Jednoduchá třída obsahující kromě jména a datového typu sérii příznaků pro jednotlivá integritní omezení.



Obr. 27: Třída Table

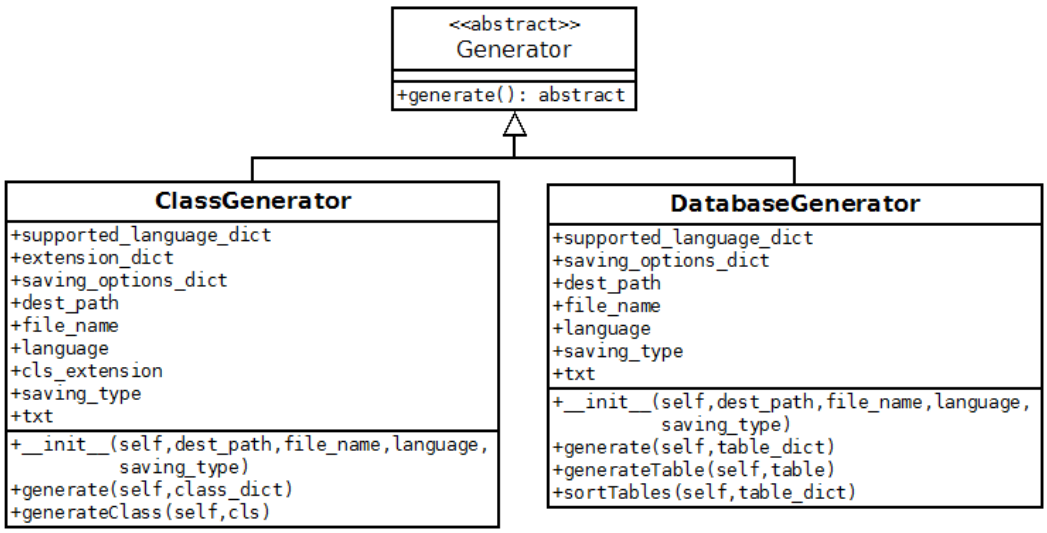


Obr. 28: Třída DbAttribute

7.5 Generátor

Podobně jako parser, je i generátor implementován ve dvou specializacích, jedna pro každý převodní mód. Společným předkem těchto specializací je abstraktní třída `Generator` obsahující pouze jednu, abstraktní metodu `generate()`, kterou je generování spouštěno z hlavního modulu.

Na základě návrhu byly generátory implementovány tak, aby poskytovaly manipulaci s uloženými daty a koordinovaly výpis na výstup. Stěžejní činností generátoru je jeho komunikace se zvolenou textbankou, která generuje řetězce cílového jazyka.



Obr. 29: Hierarchie tříd generátoru

7.5.1 Třída `ClassGenerator`

Třída specializovaného generátoru pro generování OO jazyků. Implementuje metodu `generate()`. Nastavení činnosti generátoru probíhá výběrem uživatelem zvolených variant ze tří slovníků povolených možností.

Slovník `supported_language_dict` slouží k volbě jazyka. Pod názvem jednotlivých jazyků jsou ve formě řetězců uloženy konstruktory jednotlivých textbank. Stejné klíče používá i druhý slovník, `extension_dict`, který na základě jazyka umožňuje výběr vhodné souborové přípony. Třetí slovník, `saving_options_dict` pouze umožňuje transformovat hodnoty vstupního přepínače pro volbu výstupu (tj. `"t "`, `"f "`, `"ff"`) na jejich slovní ekvivalenty pro lepší čitelnost kódu.

Při inicializaci této třídy dochází na základě parametrů k instanciaci textbanky zvoleného jazyka do atributu `self.txt` za použití vestavěné python funkce `eval()`, která vykoná obsah řetězce, v tomto případě konstruktoru ze `supported_language_dict`.

Metoda `generate()`

Převod na OO jazyky umožňuje tři způsoby výstupu: na terminál, do souboru nebo jeden soubor na třídu. Každá z těchto variant je v `generate()` reprezentována cyklem, který vhodným způsobem volá metodu `generateClass()`.

Pokud byl zvolen výpis do souboru, dochází k vytvoření tohoto souboru (na základě parametrů zadaných při spuštění programu), včetně vhodné přípony. Do něj jsou pak zapisovány výstupy ze všech iterací volání metody `generateClass()`. Podobně dochází k výpisu na terminál.

Mají-li být generované třídy ukládány každá do svého souboru, dochází k vytváření nového souboru v každé iteraci. Název jednotlivých souborů je složen z kořenového názvu, jak byl předán parametry při spuštění programu, a názvu dané třídy oddělených podtržítkem, jak je uvedeno na následujícím příkladu.

```
PATH/filename_classname.extension
```

Metoda `generateClass()`

V této metodě dochází ke komunikaci s textbankou, které jsou postupně předávána data z třídy předané v parametru. Nachází se zde tři cykly pro postupné zpracování atributů, metod a asociací. Po zpracování všech dat `generateClass()` přijímá od textbanky výsledný řetězec s danou třídou ve vybraném jazyce, který vrací návratovou hodnotou.

7.5.2 Třída `DatabaseGenerator`

Strukturou podobná třídě `ClassGenerator`. Protože soubory s jazykem SQL, resp. jeho dialekty, nemají pevně definované přípony, `DatabaseGenerator` případnému novému souboru přiřazuje příponu s názvem cílového SŘBD, `extension_dict` zde tedy není potřeba.

Analogicky ke ClassGenerator se zde nachází metody `generate()` a `generateTable()`. `Generate()` řídí generování na terminál/výstup. Seznam tříd nejdříve seřadí metodou `sortTables()` a následně pro jednotlivé zpracované třídy `Table` volá `generateTable()`, která komunikuje s textbankou a nejprve v cyklu prochází atributy dané tabulky, následně její cizí klíče.

Metoda `sortTables()`

Tato metoda má za úkol přetřídit seznam zpracovaných tabulek do vhodného pořadí. Toto pořadí je odvozeno od množství tabulek, na které daná tabulka odkazuje. Tabulky bez odkazů jsou řazeny na první místa, tabulky odkazující na již zařazené tabulky jsou řazeny za ně atp. Důvodem je, že některé SŘBD vyžadují, aby odkazované tabulky při vytváření cizího klíče již existovali. Tato metoda zaručí, že tabulky budou vytvářeny v pořadí, aby byla tato podmínka splněn.

Dochází k vytvoření nového seznamu tabulek, který je postupně plněn. Jádrem algoritmu je cyklus, který prochází původním seznamem tabulek tak dlouho, dokud není prázdný. U každé tabulky dochází ke kontrole odkazovaných tabulek. Neodkazuje-li tabulka na žádnou další tabulku, je přesunuta z původního seznamu do nového. Obsahuje-li zpracovávaná tabulka cizí klíče, je kontrolována přítomnost odkazovaných tabulek v původním seznamu. Nenachází-li se v tomto seznamu ani jedna (tzn. již byly zpracovány a přesunuty do nového seznamu), může být přesunuta i tabulka, která na ně odkazuje.

Tento cyklus se sám nachází v dalším cyklu, který je ukončen buď kontrolou prázdnosti původního seznamu, nebo dosažením maximálního povoleného počtu iterací. Počet iterací byl v rámci této práce nastaven na 100, tj. umožňuje vzájemné zřetězení odkazů až 99 tabulek. Jeho hlavním záměrem je však prevence cyklických referencí.

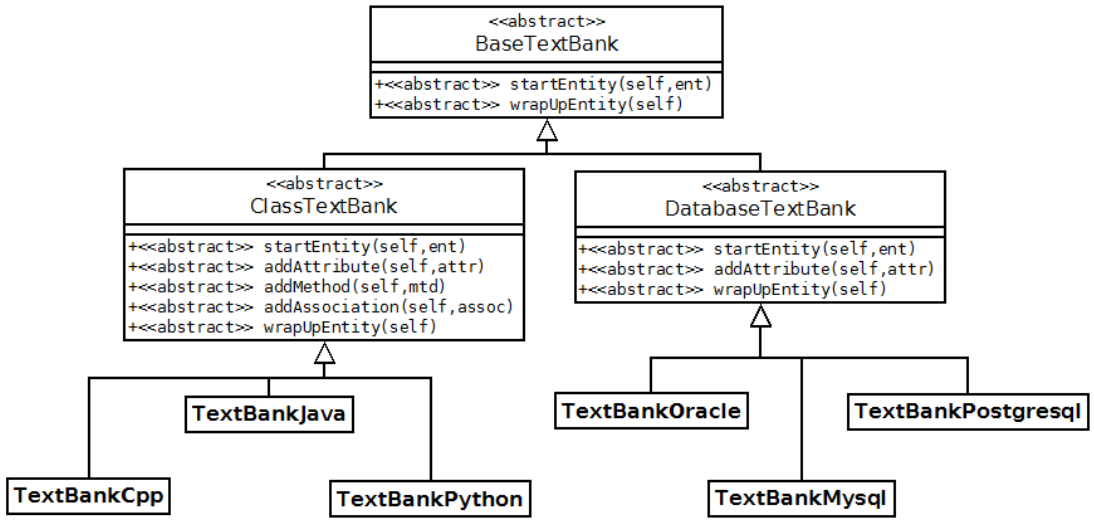
Dojde-li k vypršení povoleného limitu, program je ukončen s chybovým výpisem.

7.6 Třídy textbank

Textbanky byly navrženy tak, aby poskytovaly jednotné rozhraní pro komunikaci s generátorem, zároveň však mají zapouzdřovat veškeré individuální znaky daného jazyka. Aby byly dodrženy tyto vlastnosti, byla i zde vytvořena hierarchie tříd, která zároveň nese strukturální podobnost s hierarchií tříd generátoru, se kterými jsou textbanky úzce provázané.

Metody `startEntity()` a `wrapUpEntity()` jsou společné všem textbankám, jsou tedy jako abstraktní metody součástí společného předka, třídy `BaseTextBank`. Třídy `ClassTextBank` a `DatabaseTextbank` k těmto přidávají další abstraktní metody doplňující komunikaci s generátorem v daném módu. U módu `class` se jedná o metody `addAttribute()`, `addMethod()` a `addAssociation()`, u módu `database` pak pouze `addAttribute()`.

Jednotlivé jazykové textbanky již pak toto rozhraní implementují a přidávají vlastní metody pro manipulaci a vytváření požadovaných řetězců, jak bude dále vysvětleno.



Obr. 30: Hierarchie tříd textbank

7.6.1 Charakteristika textbanky

Ačkoliv jsou textbanky implementovány jako třídy určené k instanciaci (obsahují instanční proměnné a metody), jejich použití leží na rozhraní mezi instančními a statickými třídami. Pro každé použití Dia2Code dochází v generátoru k vytvoření instance textbanky zvoleného jazyka, aby tato mohla být uložena do proměnné pro další, jednotnou manipulaci, ať už byla zvolena jakákoliv textbanka.

S touto instancí je však dále nakládáno, jako by se jednalo o statickou třídu. Nedochází tedy k několikanásobné instanciaci této textbanky, ale k vyprazdňování a naplňování atributů jediné, původní instance. Takto je implementačně vyřešen návrh, který textbanky chápe jako jakési volitelné rozšíření centrálního generátoru; více instancí by zde tedy nebylo vhodné.

Textbanka implementuje proces generování řetězce jedné stěžejní struktury ve zvoleném módu: v módu `class` se jedná o třídu, v módu `database` o tabulku.

7.6.2 Struktura textbanky

Atributy textbanky lze logicky rozdělit na dvě části. První část, tzv. "řetězcové" atributy slouží k ukládání mezivýsledků. Generátor postupně volá jednotlivé metody textbanky s různými daty (atributy, metody, parametry metod). Nejedná se tedy o jednorázový proces; vygenerované řetězce musí být uchovávány, dokud nebude generování ukončeno a všechny řetězce nebudou sloučeny do finální, navracené formy. Příkladem řetězcového atributu může být například `self.private_mtd_string`, který se nachází v textbance jazyka C ++. K tomuto atributu jsou konkatenovány řetězce deklarací metod nastavených jako soukromé. Při tvorbě finálního řetězce třídy pak budou moci být tyto metody umístěny pod modifikátor přístupu `private`.

Druhým typem atributů jsou tzv. "vzorové" atributy. Využívají formátovaných řetězců jazyka Python, které umožňují jednoduše vkládat proměnné hodnoty na stanovená místa v pevně daném řetězci. Tato místa jsou v řetězci stanovena přímo sousedícími složenými závorkami "{}".

Každá textbanka má sadu vlastních vzorových atributů, které odpovídají konstrukcím daného jazyka pro definici metody, deklaraci atributu, řádkový komentář atp. Tyto atributy jsou pak používány jako vzor při generování dílčích řetězců: jako proměnné jsou vhodně dosazována data předávána centrálním generátorem. Každý vzorový atribut je v kódu důsledně okomentován včetně jednoznačného určení, jaký typ dat má být vložen na kterou proměnnou pozici. Následující příklad demonstruje vzor pro tvorbu řetězce atributu bez počáteční hodnoty v jazyce C ++ (hranaté závorky značí nepovinnou hodnotu):

```
# ATTRIBUTE PATTERN - for attributes without default value
# - "{[static]}{data_type} {attr_name};{line_comment}\n"
self.attribute_format = "{}{} {};{}\n"
```

7.6.3 Princip činnosti textbanky

Jak bylo nastíněno v předchozích kapitolách, jediná instance dané textbanky zajišťuje generování všech struktur daného jazyka. Každá struktura (třída/tabulka) je však textbankou generována zvlášť a zpracování následující struktury nelze započít dřív, než je dokončeno vygenerování struktury předchozí.

Správný chod textbanky je řízen z centrálního generátoru pomocí metod společného rozhraní všech textbank. Zbývající, vlastní metody jednotlivých textbank zde nebudou detailně rozebírány, protože slouží pro potřeby generování konkrétního jazyka a svou podstatou nejsou relevantní pro obecný pohled na činnost textbank.

Metoda `startEntity()`

Tuto metodu lze chápat jako opětovnou inicializaci textbanky pro práci s novou strukturou. Touto strukturou je nahrazena předchozí struktura v atributu `self.cls` (mód `class`), případně `self.table` (mód `database`). Dochází zde k uvedení řetězcových atributů do počátečního stavu (tedy k jejich nastavení na prázdné řetězce ""). V databázových textbankách zde zároveň dochází k tvorbě a uložení řetězce primárního klíče.

Add-metody

Metody volané pro jednotlivé variabilní prvky dané struktury (`addMethod()`, `addAssociation()`,`addTable()`,`addAttribute()`,`dForeignKey()`). Dochází zde ke generování řetězce (případně více řetězců), které se daného elementu týkají. Tyto řetězce jsou přidávány do vhodných řetězcových atributů.

V metodě `addAssociation()` dochází k "překladu" dané asociace na atribut třídy. Zde je volána metoda třídy `Association`, `isSingleMultiplicity()`, aby došlo k určení, jestli má vytvářený atribut být jednohodnotová proměnná, nebo jestli se má jednat o strukturu schopnou uložit více hodnot. Název nově vytvořeného atributu je odvozen od názvu odkazované třídy, názvu její role (je-li tento zadán) a slova "*association*". Protože počet asociací mezi dvěma tabulkami může být větší než jedna, a ani jedna z těchto asociací nemusí mít pojmenované role, vede si `ClassGenerator` interní počítadlo, které připojuje ke generovanému názvu, pokud není potřebná role zadána. Duplicita názvů rolí kontrolována není; integrita identifikátorů zadávaných uživatelem je zcela v uživatelově režii.

Formáty generovaných názvů pro proměnné asociací jsou tedy následující:

- `[role-name]_[other-class-name]_[association]`
- `[other-class-name]_[association + number]`

Podobně metoda `addForeignKey()` vytváří nové sloupce, které budou sloužit k odkazování na sloupce primárního klíče odkazované tabulky. Generátor si zde rovněž vede počítadlo cizích klíčů a pro každý zpracovávaný klíč dodává k názvům nových sloupců současnou hodnotu počítadla. Je tak zajištěna unikátnost názvů vytvořených sloupců i v případě, že existuje více cizích klíčů odkazujících na tutéž tabulku. Aby byla umožněna lepší manipulace s těmito klíči, je obdobně generováno i unikátní jméno celého cizího klíče, ve tvaru zobrazeném v následujícím příkladu:

- `[fk + number]__[this-table]_[referenced-table]`

Metoda `wrapUpEntity()`

Tato metoda je volána po dokončení generování dílčích řetězců pomocí add-metod. Všechny uložené řetězce jsou zde skládány dohromady, až vznikne jeden výsledný řetězec s celou strukturou v daném jazyce. Tento je vracen generátoru návratovou hodnotou.

7.6.4 Význačné vlastnosti jednotlivých implementovaných textbank

Protože jednotlivé OO jazyky, příp. SQL dialekty se od sebe navzájem liší, i vlastní řešení některých syntaktických prvků je napříč textbankami rozdílné. V této kapitole budou zmíněny nejdůležitější charakteristické rysy jednotlivých jazyků a jejich řešení v příslušné textbance.

Textbanka jazyka C ++

- Jazyk C ++ je z hlediska struktury tříd poměrně specifický. Deklarace třídy (včetně atributů a metod) v kódu existuje separátně od samotné implementace jednotlivých metod. Proto i v této textbance je generování metod řešeno ve dvou krocích. Dochází k vytvoření řetězce deklarace metody, který je později umístěn do deklarační části třídy. Rovněž je vytvořen řetězec definice metody, tedy hlavička a tělo připravené pro vepsání samotného kódu. Definice jsou ukládány zvlášť a ve `wrapUpEntity()` jsou řazeny až za tělo deklarace třídy.
- V C ++ dochází v deklarační části třídy k dělení atributů a metod pod jednotlivé modifikátory přístupu. Tímto se liší např. od Javy, kde je každému z těchto modifikátor přímo předřazen, mohou se tedy nacházet v libovolném pořadí. Zde musí dojít ke třídění atributů a metod, aby mohly být vhodně umístěny pod daný modifikátor. Protože atributy mohou být přidávány přímo v textbance (atributy vytvářené pro reprezentaci asociace), jsou atributy a metody s daným přístupem ukládány zvlášť, aby mohly být pod modifikátor pro větší přehlednost vloženy nepomíchané.

- Protože v C ++ je abstraktní třída dána vlastnictvím alespoň jedné čisté virtuální metody (`pure virtual` metody) [19], příznak abstraktní třídy je v této textbance ignorován. Abstraktní metody jsou tedy v této třídě generovány jako `pure virtual`, aby zároveň označily za abstraktní svou třídu.
- Atributy reprezentující asociaci s vícečetnou či proměnnou četností jsou generovány jako `std::vector`.
- Třídy v C ++ bývají standardně uloženy ve dvou souborech (deklarace v hlavičkovém `.h` souboru, implementace v.`cpp` souboru). Způsob komunikace generátoru s textbankou v současné chvíli uložení do dvou souborů nepodporuje (došlo by k narušení univerzálnosti centrálního generátoru testem na tento konkrétní jazyk). Vyřešení této záležitosti může být chápáno jako možné rozšíření Dia2Code.
- Přednastavené hodnoty parametrů mohou být zapsány pouze v deklaraci [19]. Obecně je volena deklarace, protože bývá v hlavičkových souborech, které jsou uživatelům k dispozici. Stejně postupuje i tato textbanka. Parametry s nastavenými hodnotami již nesmí být následovány parametry bez nastavených hodnot, dochází zde tedy k třídění parametrů do dvou řetězcových atributů. (Tyto musí být znovu nastaveny na prázdný řetězec při každém spuštění `getParameters()`).
- Dojde-li ke kolizi příznaků `static` a `abstract` u metody, je dána přednost příznaku abstract.

Textbanka jazyka Java

- Komentáře k jednotlivým metodám a jejich parametrům jsou v této textbance generovány tak, aby byl dodržen formát automaticky generované Java dokumentace, tzv. *javadoc*. Parametrům tedy předchází tag `@param` a za seznam parametrů je rovněž generován řádek s návratovým datovým typem metody s tagem `@return`.
- Java ve své základní struktuře nepodporuje přednastavené parametry metod. Je-li tedy některý parametr přednastaven, tato hodnota není ve výsledném kódu použita.
- Podobně jako u C ++ je při kolizi dána přednost příznaku `abstract` před příznakem `static`.
- V Javě neexistují konstantní metody. Má-li metoda nastavena tento příznak, nebude brán v úvahu.
- Atributy reprezentující asociaci s vícečetnou či proměnnou četností zde mají datový typ `Vector`.
- Java třídám předřazuje modifikátor přístupu, v Dia však není možnost tyto třídám nastavit. Všechny generované třídy budou automaticky považovány za

veřejné – public. (Takto označené třídy se v Javě musí nacházet v oddělených souborech [20], v Dia2Code se však volba tisku všech tříd do společného souboru považuje za mezikrok pro další, např. automatizované zpracování.)

Textbanka jazyka Python

- Proměnné v Pythonu nejsou svázané s konkrétním datovým typem, protože vše v tomto jazyce se chápe jako objekt. [21] Proto jsou případné datové typy ze zdrojového diagramu při generování vypuštěny. (Pozn.: Python3 od verze 3.5 pomocí modulu `typing` umožňuje vkládat anotace datových typů, tyto však v základní verzi Dia2Code nejsou používány kvůli potřebě importu tohoto modulu.)
- Statické třídní atributy v Pythonu nejsou určeny modifikátorem jako např. v C ++ a Javě, ale jejich umístěním v těle třídy, mimo kteroukoliv metodu. Naopak, instanciální atributy musí být vytvořeny v těle některé z metod této třídy, obvykle v konstruktoru. V této textbance jsou tedy statické atributy umístěny bezprostředně za hlavičku třídy, instanciální atributy jsou vloženy do metody konstruktoru `__init__()`. Není-li tato součástí třídy ve zdrojovém diagramu, dojde k jejímu vytvoření.
- Python nenabízí modifikátor určující abstraktní třídu a abstraktní metodu. Této funkcionality je dosaženo použitím třídy `ABCMeta` a dekorátoru `@abstractmethod` z knihovny `abc`. Je-li tedy zpracovávána abstraktní třída, je jí předřazen import výše zmíněných. V této třídě dojde k vytvoření statické proměnné `__metaclass_ = ABCMeta` a každé abstraktní metodě je předřazen dekorátor `@abstractmethod`. Toto použití modulu `abc` je možné od verze Python 3.4 dále, jeho použitím v této textbance tedy zároveň dochází ke stanovení kompatibility vygenerovaného kódu s těmito verzemi (jedná se o jediný generovaný prvek, který se napříč verzemi liší). Aby mohly být prvky knihovny `abc` použity, musí být přítomen import těchto prvků. Protože třídy mohou být ukládány jednotlivě, každá do svého souboru, je tento import generován nad každou abstraktní třídou. Při uložení více tříd v jednom souboru tedy může dojít k duplicitním importům, tyto však v jazyce Python nejsou kolizní.
- Přednastavené hodnoty parametrů metod jsou v Pythonu pevně určeny již při spuštění, ne až při volání dané metody. Pokud jsou tedy parametrům přednastavené jiné než primitivní datové typy, může dojít nezamýšlenému chování výsledného kódu. Z podobných důvodů nemohou být přednastavenými hodnotami instanciální atributy (začínající „`self.`“), protože daná instance ve chvíli stanovování přednastavené hodnoty ještě neexistuje. Tato textbanka ve své současné podobě neposkytuje žádnou kontrolu ani generování ošetřujícího kódu do dané metody. Počítá se se

znalostí uživatele dané problematiky a jeho vlastním zájmu vkládat do diagramu validní data.

- Podobně jako v Javě zde neexistují konstantní metody. Má-li metoda nastavena tento příznak, nebude brán v úvahu.
- Komentáře k jednotlivým metodám a třídám jsou umístěny pod jejich hlavičky ve formátu víceřádkového komentáře, aby bylo možné z nich generovat Python dokumentaci, tzv. *docstring*. [22]
- Atributy reprezentující asociaci s vícečetnou či proměnnou četností jsou v této textbance generovány jako seznam: `[]`.

Textbanka dialektu MySQL

- MySQL umožňuje přidávat komentáře přímo k jednotlivým sloupcům tabulky pomocí klíčového slova `COMMENT`. [23] V této textbance jsou případné komentáře jednotlivých sloupců (atributů) řešeny tímto způsobem.
- Primární klíč lze v MySQL nastavit více způsoby. Řádek s jeho definicí může začínat klíčovým slovem `CONSTRAINT`, tato varianta však není v této textbance použita a primární klíč je definován pouze pomocí `PRIMARY KEY (row_names)`. Analogicky je zacházeno s omezením `UNIQUE` a `FOREIGN KEY`, kde druhý zmíněný je navíc pojmenován, jak bylo vysvětleno u metody `addForeignKey()`. K volbě tohoto formátu došlo kvůli jeho přehlednosti.
- Ačkoliv Dia neumožnuje nastavit sloupcům omezení `DEFAULT` (viz kapitola 6.1.2), způsob přijímání dat ze zdrojového diagramu (řetězec na dané pozici je přijat tak, jak je) poskytuje uživateli možnost vepsat toto omezení za datový typ, např.: `INT DEFAULT 0`. Toto platí obecně pro všechny SQL dialekty a zmíněný postup lze aplikovat i na další omezení, jež se mohou nacházet v rámci příkazu pro tvorbu sloupce (např. `CHECK`).

Textbanka dialektu Oracle

- Na rozdíl od MySQL je v Oracle nutné definovat primární klíče, cizí klíče i omezení `UNIQUE` jako pojmenovaná integritní omezení `CONSTRAINT`. Dochází zde tedy ke generování názvu i u `PRIMARY KEY` a `UNIQUE`. Těmto tedy rovněž musí být generována unikátní jména.

Textbanka dialektu PostgreSQL

- V PostgreSQL je možné vytvořit primární klíče, cizí klíče i omezení `UNIQUE` podobně jako v MySQL (bez nutnosti klíčového slova `CONSTRAINT`). [24]

U cizího klíče však tato varianta neumožňuje tento klíč pojmenovat. Aby byla dodržena konzistence s předchozími textbankami, kde jsou cizím klíčům přidělovány i unikátní názvy, musí zde tedy být použita varianta `"CONSTRAINT name FOREIGN KEY..."`.

7.7 Modul zpracování chyb

Pro přehlednější řešení chybových situací byl pro Dia2Code naimplementován modul `error_handler` nacházející se v kořenové složce programu, který obsahuje stejnojmennou třídu `ErrorHandler`. Atributy této třídy tvoří dva slovníky, první z nich, `self.exit_code`, obsahuje chybové návratové hodnoty indexované řetězcem s názvem části programu, kde k chybě došlo (viz Tab. 6). Druhý slovník, `self.err_dict`, je naplněn řetězci s chybovými hlášeními pro konkrétní chyby. Indexem k těmto hláškám je řetězec identifikující danou chybu.

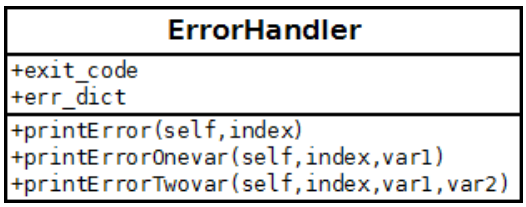
Chybová hlášení mohou být trojího typu: klasický řetězec, formátovaný řetězec s jednou proměnnou a formátovaný řetězec se dvěma proměnnými. Pro každý z těchto typů v třídě `ErrorHandler` existuje metoda, která zajišťuje jeho správné zformátování a vytisknutí.

K formátování výstupu je použita vestavěná knihovna `textwrap`, která chybová hlášení zarovnává na 80 znaků na řádek.

Hlášení jsou tisknuta na standardní chybový výstup a program je ukončen s vhodnou návratovou hodnotou ze slovníku `self.err_dict`.

Typ chyby	Návratová hodnota	Popis
`Parameter`	1	Chybně zadaný parametr (chybná cesta k souboru, soubor nelze otevřít, zvolen nepodporovaný jazyk, zdroj není xml...)
`Xml`	2	Poškozený xml soubor (chybí esenciální elementy)
`Diagram`	3	Vlastní chyby v modelu (nepropojené vztahy)
`Parser`	4	Chyba nalezená při běhu parseru
`Generator`	5	Chyba nalezená při běhu generátoru

Tab. 6: Chybové návratové kódy dia2code



Obr. 31: Třída ErrorHandler

8 DEMONSTRACE ČINNOSTI

V této kapitole bude na příkladech předvedena základní funkcionalita implementovaného Dia2Code. Příklady budou voleny jednoduché, aby nedošlo k zahlcení čtenáře stránkami kódu.

Analogicky k předchozím kapitolám bude nejprve představena činnost módu `class`, následně činnost módu `database`.

8.1 Mód `class`

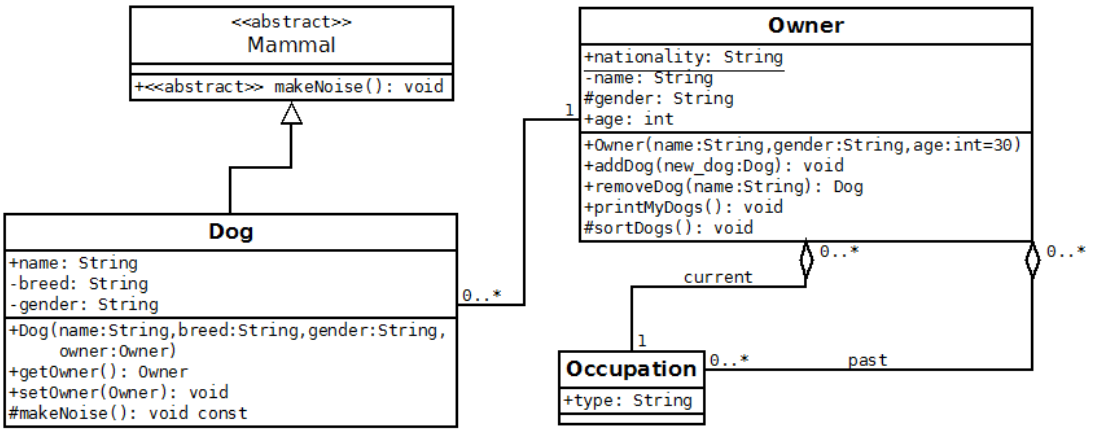
Demonstrační diagram tříd volně interpretuje příklad majitele a psa, který již byl v této práci několikrát použit. Jak je z modelu (Obr. 1) patrné, větší důraz byl kladen na názornost než použitelnost výsledného kódu. Modifikátory přístupu u jednotlivých atributů a metod jsou náhodné, aby došlo k použití všech variant. Z rozdílnosti cílových jazyků je zjevné, že model může být optimalizován pouze pro jeden z nich (datové typy, konstruktory), v tomto případě se jedná o C ++.

Třídě `Dog` byla navíc přidána rodičovská třída `Mammal` (savec) s jedinou, abstraktní metodou `makeNoise()`. Implementace této metody ve třídě `Dog` byla označena za konstantní.

Třídě `Owner` byl přidán statický atribut `nationality` (národnost). Poslednímu parametru konstruktoru `Owner()` je přiřazena implicitní hodnota „`null`“. Tato třída rovněž dvakrát agreguje třídu `Occupation` (zaměstnání). Současné zaměstnání může mít majitel pouze jedno, zatímco předešlých zaměstnání může být více.

Většině prvků byly zároveň přiřazeny komentáře, aby mohl být demonstrován jejich formát ve výsledném kódu.

Výstupy v jednotlivých jazycích se nachází v přílohách A, B a C. (Pozn.: Některé komentáře byly pro přehlednost v této práci ručně zalomeny.)



Obr. 32: Demonstrační diagram tříd

Jazyk C ++

- Atributy a metody jsou roztříděny pod jednotlivé modifikátory přístupu.
- Definice metod jsou uloženy mimo těla svých tříd.
- Abstraktnost třídy `Mammal` je určena metodou `makeNoise()`, která je dle syntaxe jazyka C ++ typu `void` a je ryze virtuální.
- Metodám s nezadanými parametry je automatický dodán parametr `void`.

Java

- Komentáře metod, především jejich parametrů jsou ve formátu *javadoc*; má-li metoda návratovou hodnotu, je přidán i tag `@return`.
- Pro vektory reprezentující vícečetné asociace je rovnou volán vhodný konstruktor.
- Všem třídám je automaticky přidělen modifikátor `public`.

Python

- Protože metody v diagramu z důvodu jeho specializace na C ++ neobsahují parametr `self`, jsou generovány bez něj. Nejedná se tedy o instanční metody (jak bylo zamýšleno), ale o metody třídní.
- Abstraktní třídě `Mammal` je předřazen import z knihovny `abc`. V jejím těle je vytvořená požadovaná statická proměnná `__metaclass__ = ABCMeta`. Samotné abstraktní metodě je předřazen dekorátor `@abstractmethod` a v jejím těle se nachází dle dokumentace příkaz `pass`.
- Přestože se v diagramu pro třídy Dog a Owner nachází konstruktory, jejich syntax odpovídá konstruktorům C ++ a Javy, ne však jazyka Python. Jak bylo vysvětleno v kapitole 7.6.4, v Python textbance musí dojít k vytvoření konstruktoru (není-li přítomen), aby bylo kam vložit instanciální atributy. V tomto konkrétním případě tedy v kódu existují metody `Owner()` a `Dog()` s komentářem, že se jedná o konstruktor (ačkoliv tomu tak není), a skutečné, vygenerované konstruktory `__init__()` pro tyto třídy.

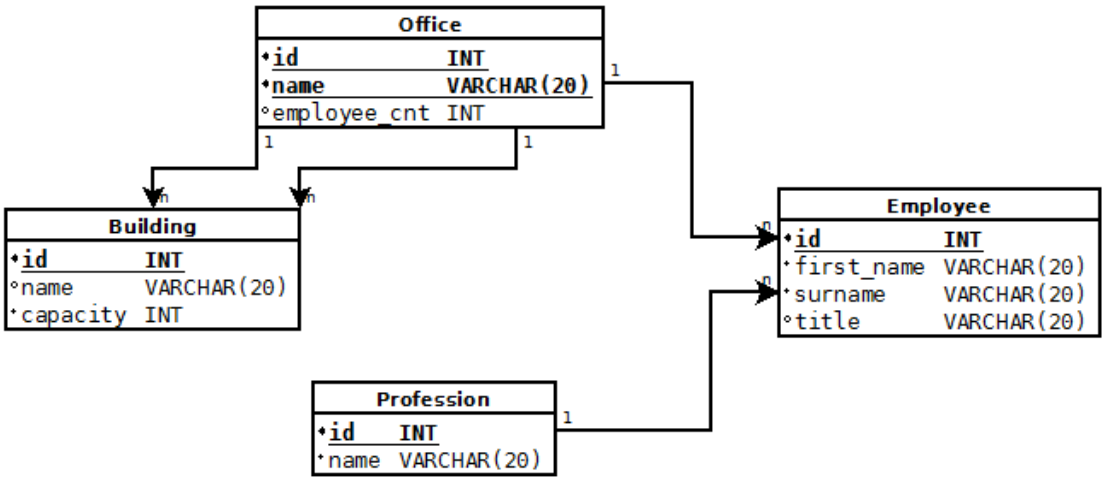
8.2 Mód database

Pro demonstraci činnosti módu `database` byla namodelována jednoduchá relační databáze (Obr. 33) ukládající informace o pobočkách firmy. Drobné piktogramy před názvy sloupců pomáhají vizuálně určit některá integritní omezení daných sloupců. Plný kosočtverec značí primární klíč, prázdný kroužek příznak `NULLABLE` a plný bod naopak `NOT NULL`. Sloupec `name` v tabulce `Profession` má navíc přiřazeno omezení `UNIQUE`. Tabulka `Office` má složený primární klíč a dvěma referencemi na ni odkazuje tabulka `Building`.

Výstupy Dia2Code pro tento vstupní diagram se nachází v přílohách D , E a F. Reference z diagramu byly převedeny na nové sloupce sloužící jako cizí klíče. Na vygenerovaných názvech je patrné číslování, které má zajistit unikátnost názvů sloupců i samotného integritního omezení v případě více referencí na tutéž tabulku.

Na výstupu v dialektu PostgreSQL si lze všimnout, že pořadí prvních dvou tabulek (`Office` a `Profession`) se liší od MySQL a Oracle. Tyto tabulky jsou při řazení tabulek na stejné úrovni (žádná nemá cizí klíč), k jejich zpracování však z povahy struktury slovník, ve které jsou uloženy, může dojít v náhodném pořadí. (Položky slovníku nebývají ukládány – a procházeny - vždy v tomtéž jistém pořadí [22].) Z toho vyplývá, že pořadí tabulek stejné úrovně bude napříč jednotlivými průchody programu vždy náhodné.

Výstup módu `database` lze na rozdíl od módu `class` nazvat **úplným** ve smyslu jeho okamžité použitelnosti v daném databázovém systému.



Obr. 33: Demonstrační relační databáze

9 ZHODNOCENÍ A NÁVRH ROZŠÍŘENÍ

Dia2Code je ve své základní formě, ve které byl implementován pro tuto práci, odlehčeným programem pro konverzi Dia diagramů na cílové jazyky. V této podobě jen v malé míře kontroluje, popř. opravuje chyby uživatele (opravuje např. pořadí parametrů bez a s přednastavenými hodnotami).

Základní syntaktické struktury generované pomocí Dia2Code byly testovány způsobem vhodným pro typ cílového jazyka. Syntaktická správnost výsledku však značně závisí na pozornosti uživatele při modelování diagramu a jeho znalosti cílového jazyka. Toto pojetí však zároveň poskytuje uživateli značnou volnost a při vhodném použití jednotlivých prvků může být dosaženo i funkcionality nad rámec možností modelů z prostředí Dia (např. `DEFAULT` hodnoty v SQL, viz kapitola 7.6.4).

Prostor pro rozšíření Dia2Code je značný. Lze přidat další jazyky do báze textbank nebo přidat moduly hlubší syntaktické kontroly dat z diagramu. Bylo by např. možné představit rozšířenou textbanku jazyka Python, která využívá anotace datových typů (jak bylo zmíněno v kapitole 7.6.4). Relačnímu databázovému modelu by mohl být zaveden vlastní Dia2Code profil stereotypů, jejichž použitím by následně došlo k rozšíření rozpoznávaných modelových dat například o triggery, omezení `CHECK` apod. Rovněž je možné se zaměřit na vizuální stránku výstupů, např. formátovat jednořádkové komentáře elementů tak, aby začínaly na téže pozici, odsazovat každý řádek víceřádkového komentáře apod.

Pro větší přehled nad činností programu by bylo žádoucí integrovat modul `logging`. Uživateli by navíc mohla být parametry dána větší kontrola nad jednotlivými postupy při generování, např. přeje-li si do kódu generovat komentáře (nyní automaticky), zpracovávat stereotypy (v případě jejich implementace) či ošetřovat syntaktické chyby. Tyto nástavby by však měly být volitelné, aby byla uživateli ponechána volnost původní myšlenky tohoto programu.

Bylo by také možné zavést další vrstvu mezi centrálním generátorem a textbankami, určité individuální předzpracování, které by mohlo zprostředkovat například právě výše zmíněné syntaktické kontroly či opravy. Umožnilo by ale např. i generování více souborů pro jednu třídu u jazyka C++ (hlavičkový `.h` a implementační `.cpp`).

Z obecnějšího hlediska by bylo možné rozšířit typy diagramů přijímané Dia2Code o další diagramy nabízené prostředím Dia (např. E-R diagram). Program je strukturován tak, aby bylo možné tyto poměrně jednoduše přidávat.

10 ZÁVĚR

Tato práce se zabývala problematikou převodu diagramů na kód se zaměřením na diagram tříd z UML. Pomocí diagramu tříd jsou modelovány programy v OO jazycích, lze je ovšem použít i pro modelování relačního databázového modelu.

V první části (kapitoly 3 a 4) se práce zaměřila na představení výše zmíněných (UML, diagram tříd, relační databázový model). Rovněž zde byl čtenář seznámen s typy cílových jazyků při převodu diagramu tříd na kód: OO jazyky a dialekty jazyka SQL (pro databázové modely).

V druhé části (kapitola 5) byla zanalyzována existující řešení na poli převodu UML na kód, komerční i volně dostupná, včetně shrnutí jejich společných rysů.

Na základě této analýzy byl v kapitole 6 navržen vlastní nástroj pro převod diagramů na kód. Za zdrojové bylo zvoleno volně dostupné modelovací prostředí Dia, které nabízí módy pro UML, včetně diagramu tříd, i mód pro databáze, kde se pracuje se zjednodušeným, diagramem tříd specializovaným na relační databázové modely.

Navržený nástroj byl nazván Dia2Code. Na rozdíl od analyzovaných řešení, která používají grafické uživatelské rozhraní, je tento nástroj ovládán pomocí příkazové řádky. Pracuje ve dvou režimech pro dva typy vstupních diagramů (mód `class` a mód `database`). Činnost nástroje lze rozdělit na dvě logické části: parser a generátor.

Parser zajišťuje zpracování zdrojového diagramu (uloženého ve formátu Dia XML) a uložení získaných dat do pomocných datových struktur (tříd). Tyto struktury jsou následně přijaty generátorem, který na základě uložených dat zprostředkuje vygenerování struktur v cílovém jazyce.

Stěžejním prvkem tohoto nástroje je idea tzv. **textbank**. Textbanka je třída zapouzdřující generování individuálních řetězců v daném jazyce. Jazyk je nástrojem podporován, pakliže se v něm nachází textbanka tohoto jazyka. Na základě uživatelovy volby je pak iniciována konkrétní textbanka, pomocí které generátor docílí vygenerování výsledného kódu. Modularita textbank a jejich společné rozhraní pro komunikaci s centrálním generátorem umožňují do Dia2Code jednoduše zavádět další jazyky, případně modifikace již existujících jazyků.

Kapitola 7 se následně zabývá samotnou implementací navrženého nástroje, jehož činnost je demonstrována v kapitole 8.

Výsledný Dia2Code je z hlediska použití jednoduchý a intuitivní, podobně jako zvolené zdrojové prostředí Dia. Nezatěžuje uživatele množstvím jednotlivých nastavení, ani robustním ovládáním. Uživateli je ponechána značná volnost ve způsobu tvorby zdrojového diagramu a zanášených dat. Součástí nástroje je i šest základních textbank, tři pro mód `class` (jazyky C ++, Java a Python) a tři pro mód `database` (SQL dialekty pro MySQL, Oracle a PostgreSQL). Jednotlivé textbanky zajišťují vhodné přizpůsobení zdrojových dat individuálním rysům svých jazyků, lze je navíc použít jako vzor pro tvorbu textbank dalších jazyků.

S ohledem na analyzovaná existující řešení lze říci, že Dia2Code nabízí nový pohled na známou problematiku. Díky svému otevřenému kódu (licence GNU GPL) a systému textbank umožňuje prakticky neomezené přizpůsobení činnosti programu vlastním potřebám, což jistě osloví nemálo uživatelů.

BIBLIOGRAFIE

[1] I v roce 2016 bude nedostatek IT odborníků. In: *Business IT: Informační technologie pro profesionály* [online]. b.r. [cit. 2017-05-17]. Dostupné z : http://www.businessit.cz/cz/i -v -roce-2016-bude-nedostatek-it-odborniku.php

[2] BOOCH, Grady, James RUMBAUGH a Ivar JACOBSON. *Unified Modeling Language User Guide*. 2nd ed. Boston: Addison-Wesley Professional, 2005. ISBN 978-0 -321-26797-9.

[3] UML. In: *UML* [online]. Needham(Massachusetts): Object Management Group, c1997-2016 [cit. 2017-05-11]. Dostupné z : http://www.omg.org/spec/UML/

[4] *OMG Unified Modeling Language (OMG UML), Superstructure*. Ver. 2.4.1. Needham(Massachusetts): Object Management Group, 2010.

[5] DERFEL73, a PMERSON. File:UML_diagrams_overview.svg. In: *Commons.wikimedia.org* [online]. Public domain, b.r. [cit. 2017-05-16]. Dostupné z : https://commons.wikimedia.org/wiki/File:UML_diagrams_overview.svg

[6] ARLOW, Jim a Illa NEUSTADT. *UML 2 a unifikovaný proces vývoje aplikací: objektově orientovaná analýza a návrh prakticky*. 2. aktualiz. a dopl. vyd. Brno: Computer Press, 2007. ISBN 978-80-2 -1-1503-9.

[7] What do their creators think of UML now?. *Modeling Languages* [online]. Castelldefels: Jordi Cabot, 2016 [cit. 2017-05-11]. Dostupné z : http://modeling-languages.com/uml-opinions-creators/

[8] FOWLER, Martin. *Destilované UML*. 1.vyd. Praha: Grada, 2009. ISBN 978-80-247-2062-3.

[9] YANPAS, . File:Uml classes en.svg. In: *Commons.wikimedia.org* [online]. Creative Commons Attribution-Share Alike 4.0 International licence, b.r. [cit. 2017-05-16]. Dostupné z : https://commons.wikimedia.org/wiki/File:Uml_classes_en.svg

[10] CODD, Edgar F. *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*. Research Report. IBM, 1969.

[11] CODD, Edgar. A relational model of data for large shared data banks. *Communications of the ACM*. 1970, **13**(6), 377-387. DOI: 10.1145/362384.362685.

[12] Database Modelling in UML. In: *UML tools for software developing and modelling* [online]. Victoria: Sparx Systems, c2000-2017 [cit. 2017-05-12]. Dostupné z :

http://www.sparxsystems.com.au/downloads/whitepapers/Database_Modeling_In_UML.pdf

[13] PECINOVSKÝ, Rudolf. *OOP - Learn Object Oriented Thinking & Programming* [online]. 1st ed. Řepín: Tomáš Bruckner, 2013 [cit. 2017-05-11]. ISBN 978-80-904661-9 -7. Dostupné z : http://files.bruckner.cz/be2a5b2104bf393da7092a4200903cc0/PecinovskyOOP.pdf

[14] STEPHENS, Ryan, Ron PLEW a Arie JONES. *Naučte se SQL za 28 dní*. 1. vyd. Brno: Computer Press, 2012. ISBN 978-80-251-2700-1.

[15] UML Association. *UML Diagrams* [online]. b.r. [cit. 2017-05-17]. Dostupné z : http://www.uml-diagrams.org/association.html

[16] Processing XML in Python with Element Tree. In: *Http://eli.thegreenplace.net/2012/03/15/processing-xml-in-python-with-elementtree* [online]. Bendersky, c2003-2017 [cit. 2017-05-12]. Dostupné z : http://eli.thegreenplace.net/2012/03/15/processing-xml-in-python-with-elementtree

[17] The 2016 Top Programming Languages. *IEEE Spectrum* [online]. New York: IEEE, 2016 [cit. 2017-05-11]. Dostupné z : http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[18] *DB-Engines* [online]. Vienna: solid IT, 2017 [cit. 2017-05-11]. Dostupné z : https://db-engines.com/en/

[19] ECKEL, Bruce. *Myslíme v jazyce C ++*. 1. vyd. Praha: Grada, 2000. ISBN 80-247-90092.

[20] HEROUT, Pavel. *Učebnice jazyka Java*. 1. vyd. České Budějovice: KNOPP, 2000. ISBN 80-7232-115-3.

[21] PILGRIM, Mark. *Dive into Python 3.* 1st. ed. New York: Apress, 2009. ISBN 978-1430224150.

[22] LUTZ, Mark. *Learning Python*. 4th ed. Sebastopol: O'Reilly, 2009. ISBN 978-0 -596-15806-4.

[23] MySQL 5.7 Reference Manual: Including MySQL NDB Cluster 7.5 and NDB Cluster 7.6. In: *MySQL* [online]. Redwood Shores: Oracle Corporation, 2017 [cit. 2017-05-12]. Dostupné z : https://downloads.mysql.com/docs/refman-5.7 -en.pdf

[24] PostgreSQL 9.6.3 Documentation. In: *PostgreSQL: The world's most advanced open source database* [online]. Wolfville, Nova Scotia: The PostgreSQL Global Development Group, c1996-2015 [cit. 2017-05-12]. Dostupné z :

https://www.postgresql.org/files/documentation/pdf/9.6 /postgresql-9.6 - A4.pdf

SEZNAM OBRÁZKŮ A TABULEK

Obr. 1: Rozdělení UML diagramů na skupiny [5] 20
Obr. 2: Příklad notace třídy v diagramu tříd 21
Obr. 3: Grafická notace vazeb mezi třídami [9] 22
Obr. 4: Příklad jednoduchého diagramu tříd 23
Obr. 5: Příklad m..n vazby 25
Obr. 6: Modifikace m..n vazby vložením třetí tabulky 25
Obr. 7: Příklad jednoduché relační databáze 25
Obr. 8: Tabulky Majitel a Pes za použití diagramu tříd 26
Obr. 9: Příklad použití databázových stereotypů 27
Obr. 10: Náhled na panel s UML prvky 38
Obr. 11: Vlastnosti prvku Třída - panel Třída 38
Obr. 12: Vlastnosti prvku Třída - panel Atributy 39
Obr. 13: Vlastnosti prvku Třída - panel Operace 39
Obr. 14 Panel vlastností Asociace 40
Obr. 15: A do B pomocí zobrazené šipky 40
Obr. 16: Směr čtení z A do B 40
Obr. 17: Náhled na panel Databáze 41
Obr. 18: Vlastnosti prvku Tabulka - panel Tabulka 41
Obr. 19: Vlastnosti prvku Tabulka - panel Atributy 42
Obr. 20: Schéma komunikace centrálního generátoru s textbankami 46
Obr. 21: Hierarchie tříd parseru 50
Obr. 22: Třída Class 54
Obr. 23: Třída Attribute 55
Obr. 24: Třída Method 55
Obr. 25: Třída Parameter 55
Obr. 26: Třída Association 56
Obr. 27: Třída Table 57
Obr. 28: Třída DbAttribute 57
Obr. 29: Hierarchie tříd generátoru 58
Obr. 30: Hierarchie tříd textbank 60
Obr. 31: Třída ErrorHandler 67
Obr. 32: Demonstrační diagram tříd 69
Obr. 33: Demonstrační relační databáze 71

Tab. 1: Příznaky viditelnosti v diagramu tříd 22
Tab. 2 : Základní adresářová struktura dia2code 47
Tab. 3 : Parametry příkazové řádky 48
Tab. 4 : Volitelné parametry řádky 48
Tab. 5 : Příklady povolených formátů četnosti 56
Tab. 6 : Chybové návratové kódy dia2code 67

PŘÍLOHA A: VÝSTUP MÓDU CLASS V JAZYCE C ++

```
class Mammal {
/* This is an abstract class representing Mammals.
It can make a noise. */

    public:
        virtual void makeNoise(void) = 0;     // Abstract/Virtual
                                                 (depends) method of
                                                  making noise
};

void Mammal::makeNoise(void) {

// YOUR CODE HERE

}

class Dog: public Mammal {
/* Class representing Dogs. Inherits from Mammal. */

    private:
        String breed;         // Dog's breed
        String gender;         // Dog's gender
        Owner 1_Owner_association;

    protected:
        void makeNoise(void) const;     // Implementation of abstract
                                           makeNoise() method inherited
                                           from Mammal


    public:
        String name;         // Dog's name
        Dog(String name,String breed,String gender,Owner owner);
                                             // Constructor method
        Owner getOwner(void);     // Owner getter
        void setOwner(Owner new_owner);     // Owner setter

};

Dog::Dog(String name,String breed,String gender,Owner owner) {
/* String name - New Dog's name
String breed - New Dog's breed
String gender - New dog's gender
Owner owner - New dog's initial owner
 */
```

```
// YOUR CODE HERE

}

Owner Dog::getOwner(void) {

// YOUR CODE HERE

}

void Dog::setOwner(Owner) {
/*  Owner - New owner to be set
 */

// YOUR CODE HERE

}

void Dog::makeNoise(void) {

// YOUR CODE HERE

}

class Owner {
/* Class representing Dog Owners.  */

    private:
        String name;        // Owner's name
        std::vector<Dog> Dog_association1;
        Occupation Occupation_association2;
        std::vector<Occupation> Occupation_association3;

    protected:
        String gender;        // Owner's gender
        void sortDogs(void);    // Sorts dogs alphabetically.

    public:
        static String nationality;        // A static attribute for
                                             Owner's nationality
        int age;        // Owner's age
        Owner(String name,String gender,int age = 30);
                                                  //Constructor method
        void addDog(Dog new_dog);    // Adds a Dog to Owner's pack of
                                        dogs
        Dog removeDog(String name);    // This method removed the dog
                                          of given name from Owner's
                                          pack of dogs
        void printMyDogs(void);    // Prints all Owner's dogs by name
};
```

```
Owner::Owner(String name,String gender,int age) {
/* String name - New Owner's name
String gender - New Owner's gender
int age - New owner's age
 */

// YOUR CODE HERE

}

void Owner::addDog(Dog new_dog) {
/* Dog new_dog - New Dog to be added
 */

// YOUR CODE HERE

}

Dog Owner::removeDog(String name) {
/* String name - Name of the dog to be removed
 */

// YOUR CODE HERE

}

void Owner::printMyDogs(void) {

// YOUR CODE HERE

}

void Owner::sortDogs(void) {

// YOUR CODE HERE

}

class Occupation {
/* Class representing Owner's both previous and current occupations.
*/

    public:
        String type;        // Type of occupation

};
```

PŘÍLOHA B: VÝSTUP MÓDU CLASS V JAZYCE JAVA

```
/**
 * This is an abstract class representing Mammals.
It can make a noise.
 */
abstract public class Mammal {

    /**
     * Abstract/Virtual (depends) method of making noise
     */
    public void makeNoise() {

    // YOUR CODE HERE

    }
}

/**
 * Class representing Dogs. Inherits from Mammal.
 */
public class Dog extends Mammal {

    public String name;         // Dog's name
    private String breed;         // Dog's breed
    private String gender;         // Dog's gender
    private Owner Owner_association1;

    /**
     * Constructor method
     * @param name New Dog's name
     * @param breed New Dog's breed
     * @param gender New dog's gender
     * @param owner New dog's initial owner
     * @return
     */
    public Dog(String name,String breed,String gender,Owner owner) {

    // YOUR CODE HERE

    }

    /**
     * Owner getter
     * @return Owner
     */
    public Owner getOwner() {
```

```
    // YOUR CODE HERE

    }

    /**
     * Owner setter
     * @param Owner New owner to be set
     */
    public void setOwner(Owner new_owner) {

    // YOUR CODE HERE

    }

    /**
     * Implementation of abstract makeNoise() method inherited from
Mammal
     */
    protected void makeNoise() {

    // YOUR CODE HERE

    }
}


/**
 * Class representing Dog Owners.
 */
public class Owner {

    public static String nationality;        // A static attribute for
                                                Owner's nationality
    private String name;         // Owner's name
    protected String gender;         // Owner's gender
    public int age;         // Owner's age
    private Vector<Dog> Dog_association1 = new Vector<Dog>();
    private Occupation Occupation_association2;
    private   Vector<Occupation>   Occupation_association3   =   new
Vector<Occupation>();

    /**
     * Constructor method
     * @param name New Owner's name
     * @param gender New Owner's gender
     * @param age New owner's age
     */
    public Owner(String name,String gender,int age) {

    // YOUR CODE HERE
```

```
    }

    /**
     * Adds a Dog to Owner's pack of dogs
     * @param new_dog New Dog to be added
     */
    public void addDog(Dog new_dog) {

    // YOUR CODE HERE

    }

    /**
     * This method removed the dog of given name from Owner's pack of
dogs
     * @param name Name of the dog to be removed
     * @return Dog
     */
    public Dog removeDog(String name) {

    // YOUR CODE HERE

    }

    /**
     * Prints all Owner's dogs by name
     */
    public void printMyDogs() {

    // YOUR CODE HERE

    }

    /**
     * Sorts dogs alphabetically.
     */
    protected void sortDogs() {

    // YOUR CODE HERE

    }
}

/**
 * Class representing Owner's both previous and current occupations.
 */
public class Occupation {

    public String type;         // Type of occupation
}
```

PŘÍLOHA C: VÝSTUP MÓDU CLASS V JAZYCE PYTHON

```
from abc import ABCMeta, abstractmethod

class Mammal:
    """
    This is an abstract class representing Mammals.
It can make a noise.
    """
    __metaclass__ = ABCMeta

    @abstractmethod
    def makeNoise():
        """
        Abstract/Virtual (depends) method of making noise
        """
        pass


class Dog(Mammal):
    """
    Class representing Dogs. Inherits from Mammal.
    """
    def __init__(self):

        self.name = None         #Dog's name
        self.breed = None         #Dog's breed
        self.gender = None         #Dog's gender
        self.Owner_association1 = None


    def Dog(name,breed,gender,owner):
        """
        Constructor method
        name - New Dog's name
        breed - New Dog's breed
        gender - New dog's gender
        owner - New dog's initial owner
        """

        # YOUR CODE HERE


    def getOwner():
        """
        Owner getter
        """
```

```
        # YOUR CODE HERE



    def setOwner(new_owner):
        """
        Owner setter
        new_owner - New owner to be set
        """

        # YOUR CODE HERE


    def makeNoise():
        """
        Implementation of abstract makeNoise() method inherited from
Mammal
        """

        # YOUR CODE HERE


class Owner:
    """
    Class representing Dog Owners.
    """
    nationality = None         #A static attribute for Owner's
                               nationality
    def __init__(self):

        self.name = None         #Owner's name
        self.gender = None         #Owner's gender
        self.age = None         #Owner's age
        self.Dog_association1 = []
        self.Occupation_association2 = None
        self.Occupation_association3 = []


    def Owner(name,gender,age = 30):
        """
        Constructor method
        name - New Owner's name
        gender - New Owner's gender
        age - New owner's age
        """

        # YOUR CODE HERE


    def addDog(new_dog):
```

```
        """
        Adds a Dog to Owner's pack of dogs
        new_dog - New Dog to be added
        """

        # YOUR CODE HERE


    def removeDog(name):
        """
        This method removed the dog of given name from Owner's pack of
dogs
        name - Name of the dog to be removed
        """

        # YOUR CODE HERE


    def printMyDogs():
        """
        Prints all Owner's dogs by name
        """

        # YOUR CODE HERE


    def sortDogs():
        """
        Sorts dogs alphabetically.
        """

        # YOUR CODE HERE


class Occupation:
    """
    Class representing Owner's both previous and current occupations.
    """
    def __init__(self):

        self.type = None        #Type of occupation
```

PŘÍLOHA D: VÝSTUP MÓDU DATABASE V DIALEKTU MYSQL

```
CREATE TABLE Office (
/*
Represents individual offices
*/
    id INT NOT NULL COMMENT 'Unique ID',
    name VARCHAR(20) NOT NULL COMMENT 'Name of the office',
    employee_cnt INT COMMENT 'How many employees work here',
    PRIMARY KEY (id,name)
);

CREATE TABLE Profession (
/*
Represents existing professions in the company
*/
    id INT NOT NULL COMMENT 'Unique ID',
    name VARCHAR(20) NOT NULL COMMENT 'Name of the profession',
    UNIQUE (name),
    PRIMARY KEY (id)
);

CREATE TABLE Building (
/*
Represents individual buildings used byt the company
*/
    id INT NOT NULL COMMENT 'Unique ID',
    name VARCHAR(20) COMMENT 'Name of the building',
    capacity INT NOT NULL COMMENT 'How many employees can work
here',
    Office_id_fk1 INT,
    Office_name_fk1 VARCHAR(20),
    Office_id_fk2 INT,
    Office_name_fk1 VARCHAR(20),
    FOREIGN KEY fk1__Building_Office (Office_id_fk1,Office_name_fk1)
REFERENCES Office(id,name),
    FOREIGN KEY fk2__Building_Office (Office_id_fk2,Office_name_fk2)
REFERENCES Office(id,name),
    PRIMARY KEY (id)
);

CREATE TABLE Employee (
/*
Representing people working for the company
*/
    id INT NOT NULL COMMENT 'Unique ID',
```

```
    first_name VARCHAR(20) NOT NULL COMMENT 'First name of the
person',
    surname VARCHAR(20) NOT NULL COMMENT 'Surname of the person',
    title VARCHAR(20) COMMENT 'Title if they have any',
    Office_id_fk1 INT,
    Office_name_fk1 VARCHAR(20),
    Profession_id_fk2 INT,
    FOREIGN KEY fk1__Employee_Office (Office_id_fk1,Office_name_fk1)
REFERENCES Office(id,name),
    FOREIGN KEY fk2__Employee_Profession (Profession_id_fk2)
REFERENCES Profession(id),
    PRIMARY KEY (id)
);
```

PŘÍLOHA E: VÝSTUP MÓDU DATABASE V DIALEKTU ORACLE

```
CREATE TABLE Office (
/*
Represents individual offices
*/
    id INT NOT NULL,        --Unique ID
    name VARCHAR(20) NOT NULL,        --Name of the office
    employee_cnt INT,        --How many employees work here
    CONSTRAINT pk__id_name PRIMARY KEY (id,name)
);

CREATE TABLE Profession (
/*
Represents existing professions in the company
*/
    id INT NOT NULL,        --Unique ID
    name VARCHAR(20) NOT NULL,        --Name of the profession
    CONSTRAINT name_unique UNIQUE (name),
    CONSTRAINT pk__id PRIMARY KEY (id)
);

CREATE TABLE Building (
/*
Represents individual buildings used byt the company
*/
    id INT NOT NULL,        --Unique ID
    name VARCHAR(20),        --Name of the building
    capacity INT NOT NULL,        --How many employees can work here
    Office_id_fk1 INT,
    Office_name_fk1 VARCHAR(20),
    Office_id_fk2 INT,
    Office_name_fk2 VARCHAR(20),

    CONSTRAINT fk1__Building_Office FOREIGN KEY
(Office_id_fk1,Office_name_fk1) REFERENCES Office(id,name),
    CONSTRAINT fk2__Building_Office FOREIGN KEY
(Office_id_fk2,Office_name_fk2) REFERENCES Office(id,name),
    CONSTRAINT pk__id PRIMARY KEY (id)
);

CREATE TABLE Employee (
/*
Representing people working for the company
*/
    id INT NOT NULL,        --Unique ID
    first_name VARCHAR(20) NOT NULL,        --First name of the
```

```
                                                person
    surname VARCHAR(20) NOT NULL,         --Surname of the person
    title VARCHAR(20),         --Title if they have any
    Office_id_fk1 INT,
    Office_name_fk1 VARCHAR(20),
    Profession_id_fk2 INT,
    CONSTRAINT fk1__Employee_Office FOREIGN KEY
(Office_id_fk1,Office_name_fk1) REFERENCES Office(id,name),
    CONSTRAINT fk2__Employee_Profession FOREIGN KEY
(Profession_id_fk2) REFERENCES Profession(id),
    CONSTRAINT pk__id PRIMARY KEY (id)
);
```

PŘÍLOHA F: VÝSTUP MÓDU DATABASE V DIALEKTU MYSQL

```
CREATE TABLE Profession (
/*
 *Represents existing professions in the company
 */
    id INT NOT NULL,        -- Unique ID
    name VARCHAR(20) NOT NULL,        -- Name of the profession
    UNIQUE (name)
    PRIMARY KEY (id)
);

CREATE TABLE Office (
/*
 *Represents individual offices
 */
    id INT NOT NULL,        -- Unique ID
    name VARCHAR(20) NOT NULL,        -- Name of the office
    employee_cnt INT ,        -- How many employees work here
    PRIMARY KEY (id,name)
);

CREATE TABLE Employee (
/*
 *Representing people working for the company
 */
    id INT NOT NULL,        -- Unique ID
    first_name VARCHAR(20) NOT NULL,        -- First name of the
                                               person
    surname VARCHAR(20) NOT NULL,        -- Surname of the person
    title VARCHAR(20) ,        -- Title if they have any
    Office_id_fk1 INT,
    Office_name_fk1 VARCHAR(20),
    Profession_id_fk2 INT,
    CONSTRAINT fk1__Employee_Office FOREIGN KEY
(Office_id_fk1,Office_name_fk1) REFERENCES Office(id,name),
    CONSTRAINT fk2__Employee_Profession FOREIGN KEY
(Profession_id_fk2) REFERENCES Profession(id),
    PRIMARY KEY (id)
);

CREATE TABLE Building (
/*
 *Represents individual buildings used byt the company
 */
    id INT NOT NULL,        -- Unique ID
    name VARCHAR(20) ,        -- Name of the building
```

```
    capacity INT NOT NULL,          -- How many employees can work
                                       here
    Office_id_fk1 INT,
    Office_name_fk1 VARCHAR(20),
    Office_id_fk2 INT,
    Office_name_fk2 VARCHAR(20),
    CONSTRAINT fk1__Building_Office FOREIGN KEY
(Office_id_fk1,Office_name_fk1) REFERENCES Office(id,name),
    PRIMARY KEY (id)
);
```

PŘÍLOHA G: INSTALACE A DEMONSTRACE DIA2CODE

REPOZITÁŘ DIA2CODE
https://github.com/Rihorama/dia2code.git

CENTOS 7 (RHEL7,OL7,SL7)

```
#sudo nebo jako root
sudo yum -y update
sudo yum -y install epel-release
sudo yum -y install python34 git
#jako obycejny uzivatel
git clone https://github.com/Rihorama/dia2code.git
```

FEDORA 25 (Workstation)

```
#jako obycejny uzivatel
git clone https://github.com/Rihorama/dia2code.git
```

UBUNTU 16.04 LTS

```
sudo apt-get install git
git clone https://github.com/Rihorama/dia2code.git
```

PŘÍKLADY:

```
cd dia2code/src/dia2code/
#priklad jak pouzit demo
./dia2code.py c../demo/Class_demonstration_EN.dia -p f -l c ++
./dia2code.py c../demo/Class_demonstration_EN.dia -p f -l java
./dia2code.py c../demo/Class_demonstration_EN.dia -p f -l python
./dia2code.py   d../demo/Database_demonstration_EN.dia   -p f -
l mysql
./dia2code.py   d../demo/Database_demonstration_EN.dia   -p f -
l oracle
./dia2code.py   d../demo/Database_demonstration_EN.dia   -p f -
l postgresql
```