Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Modelovací software podporující model tříd v UML

Ladislav Böhm

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Katedra informačního inženýrství

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Ladislav Böhm

Informatika

Název práce

Modelovací software podporující model tříd v UML.

Název anglicky

Modelling software supporting class model in UML.

Cíle práce

Naprogramujte a otestujte malý CASE nástroj, který bude umět namalovat zjednodušený model tříd podle UML a z něj vygenerovat základ kódu ve vybraném objektovém programovacím jazyku a také v SQL.

Metodika

Využití metod řízení životního cyklu tvorby softwaru. Obvyklé standardy softwarového inženýrství včetně dokumentace.

Doporučený rozsah práce

30-50 stran

Doporučené zdroje informací

AMBLER, Scott. Data modeling. In: Agile Data [online]. 2003 [cit. 2014-04-07]. Dostupné z: http://www.agiledata.org/essays/umlDataModelingProfile.html

FOWLER, Martin. Destilované UML. 1. vyd. Praha: Grada, 2009, 173 s. Knihovna programátora (Grada). ISBN 978-80-247-2062-3

FREEMAN, Eric a Elisabeth FREEMAN. Head first design patterns. 1st ed. Sebastopol: O´Reilly, 2004, xxxvi, 638 s. ISBN 978-0-596-00712-6

NAGEL, Christian, Bill EVJEN, Jay GLYN, Karli WATSON a Morgan SKINNER. Professional C# 2012 and .Net 4.5. Indianapolis, IN: Wiley, c2013, lvii, 1523 p. Programmer to programmer. ISBN 11-183-1442-5

POKORNÝ, Jaroslav a Michal VALENTA. Databázové systémy. 1. vyd. Praha: České vysoké učení technické v Praze, 2013, 265 s. ISBN 978-80-01-05212-9

Předběžný termín obhajoby

2015/06 (červen)

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Elektronicky schváleno dne 10. 11. 2014

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 10. 11. 2014

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 07. 03. 2015

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Modelovací Software podporující model tříd v UML" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3. 2015 _____

Poděkování

Děkuji tímto doc. Ing. Vojtěchu Merunkovi, Ph.D za cenné rady a připomínky týkající se vhodného návrhu a zpracování této práce, které vedly k jejímu zdárnému zpracování.

Modelovací software podporující model tříd v UML

Modelling software supporting class model in UML

Souhrn

Datové modelování je součástí softwarového inženýrství. Je to technika, kterou je možno analyzovat a popsat data a vazby mezi nimi. Umožňuje tak vytvořit model bez znalosti konkrétního programovacího jazyka či databázové technologie. Jeden z možných přístupů k datovému modelování pochází od Scotta Amblera, odborníka na objektově orientovaný přístup a agilní metody. Zmíněný přístup, který respektuje standard UML diagramu tříd, je v práci implementován do zjednodušené podoby CASE nástroje. Z vytvořeného modelu také umožňuje generovat základ kódu v několika objektově orientovaných jazycích a SQL.

Summary

Data modeling is a part of software engineering. It is a technique, which allows us to analyze and describe data and relationships between them. It allows us to create a model without knowledge of specific programming language or database technology. One possible approach to data modelling is presented by Scott Ambler, an expert on object-oriented development and agile methods. Previously mentioned approach, which follows UML standard of class diagram, is implemented in this thesis into simplified form of CASE software. The software also allows to generate code in selected object-oriented language and SQL.

Klíčová slova: UML, CASE, OOP, SQL, C#, SmallTalk, Java, MVVM, diagram tříd, návrhové vzory, agilní metody, unit test

Keywords: UML, CASE, OOP, SQL, C#, SmallTalk, Java, MVVM, class diagram, design patterns, agile methods, unit test

Obsah

Part I
Úvod

Vývoj softwaru dnes není zdaleka jen záležitostí jeho implementace v konkrétním programovacím jazyce. Samotné implementaci předchází několik fází analýz a plánování, které ovlivňují jeho budoucí podobu a v jistých případech mohou tvorbu projektu i předčasně ukončit. Jednou z takových metod je datové modelování, které umožňuje vystihnout povahu potřebných datových struktur a vazeb mezi nimi. Modely se tvoří z požadavků klienta a upravují se v průběhu jejich tvorby v závislosti na aplikované metodice vývoje.

Modelovat lze samozřejmě pomocí běžných psacích potřeb na papír či tabuli, které jsou mnohými autory při aplikaci agilních metodik upřednostňovány. U rozsáhlých modelů je ovšem vhodnější využít některý z dostupných CASE nástrojů a mít tak možnost modelovat rozměrné a snáze udržovatelné modely. Komerční CASE nástroje mají mnoho funkcí, což ale často znamená, že se odchylují od čistého datového modelování a dělají ústupky konkrétním databázovým technologiím, nebo programovacím jazykům. Modely poté ztrácí svou vlastnost abstrakce, jsou nepřenositelné, složité a nesrozumitelné pro lidi bez znalosti dané technologie. Z těchto důvodů se velká část programátorů odklání od používání takových programů a volí raději tradiční metody, nebo nemodelují vůbec.

Výhody modelování jsou však mnohými odborníky považovány za nesporné a prospěšné i v týmech zastávající agilní přístupy, které jsou v dnešní době velmi často aplikovány.

Part II
Cíl práce a použitá metodika

1 Cíl práce

Cílem práce je popsat vývoj objektového přístupu ke tvorbě softwaru, zejména techniky datového modelování. Poskytnout přehled o aktuálně využívaných metodikách řízení vývoje se zaměření na agilní metodiky a ukázat současné možnosti modelování pomocí diagramu tříd ve standardu UML.

Stěžejní část práce je věnována implementaci výše jmenovaných technik do zjednodušené podoby CASE nástroje. Práci provází zdokumentovaný postup jeho vývoje a umožní tak čtenáři vidět praktickou implementaci teoreticky vymezených témat. Aplikací dekompozice a komponentového přístupu k tvorbě softwaru je umožněna jeho možná budoucí rozšiřitelnost či využití částí jeho knihoven v dalších, zcela nezávislých projektech.

Cílem programu v jeho finální podobě není konkurenceschopnost současným CASE nástrojům. Program slouží k demonstraci jednoho z možných přístupů k datovému modelování při zachování notace UML diagramu tříd a prokáže to funkcí generování kódu do několika vybraných objektově orientovaných jazyků a SQL.

2 Struktura práce a metodika

Práce je strukturována na teoretickou část, která má za úkol vysvětlit vznik a vývoj použitých technologií, včetně jejich současného stavu a uplatnění. Větší část bude následně věnována postupu při vývoji programu a vybraným detailům z jeho implementace. Vývoj softwaru byl průběžně konzultován a následně upravován dle doporučení vedoucího bakalářské práce.

Práce bude provázena obrázky diagramů užitých technologií a dodaného programu, včetně vybraných částí kódu.

Software byl naprogramován ve vývojovém prostředí Microsoft Visual Studio Ultimate 2013 v jazyce C#, na platformě WPF. Využit byl framework .NET 4.5, knihovna Json.NET od Jamese Newton-Kinga a verzovací systém Team Foundation Server Version Control. Práce byla napsána technologií LaTex v programu LyX.

Zdrojem pro vypracování práce byly odborné publikace uvedené v seznamu literatury a přednášky doc. Ing. Vojtěcha Merunky, Ph.D k předmětu *Datové a znalostní modelování.*

Part III
Metodiky vývoje softwaru

Problematika řízení vývoje softwaru jako procesu se řešila již v období procedurálních jazyků. Potřeba metodiky plyne z potřeb zadavatelů sledovat průběh vývoje softwaru, analytiků a manažerů navrhnout plán financování a očekávaného dokončení projektu. Hlavním rozdílem mezi metodikami rigorózními a agilními je v pohledu na členění procesu vývoje na menší celky. [Fowler, 2009]

Z obrázku 1, je možné vidět důležitost správné volby metodiky [1]. Kritérium pro zahrnutí do kategorie **successful** bylo dodání hotového řešení a splnění všech požadavků tak, aby bylo pro zadavatele přijatelné. Kategorie **challenged** obsahuje projekty, které byly dodány, ale například jejich kvalita nebyla na požadované úrovni. Do **challenged** skupiny spadají i projekty, jejichž kvalita byla na požadované úrovni, ale například byl výrazně překročen rozpočet. V poslední skupině **failed** se nachází projekty, které nebyly dodány zadavateli.

Tradičními metodikami jsou označovány rigorózní metodiky, jako je například vodopádový model, popsaný v kapitole 3.1.

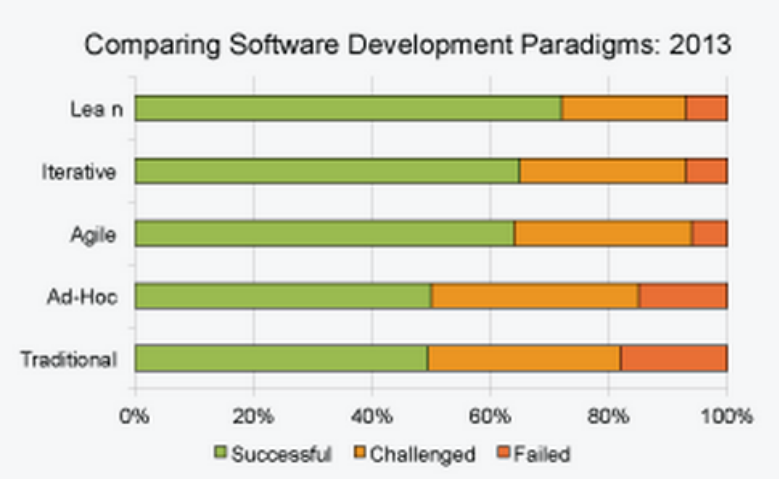

Figure 1: Úspěšnost projektů v závislosti na použité metodice (Zdroj: www.scottambler.com)

3 Rigorózní metody

Prvními uplatňovanými metodikami byly rigorózní metody, které se vyznačují především statickým pohledem na řízení vývoje. Rigorózní metodiky předpokládají, že se specifikace požadavků během vývoje nebudou měnit. Zadavatel v podobě budoucího

[1] Data uvádí Scott Ambler jako výsledek jeho vlastní studie z období listopadu a prosince roku 2013 o úspěšnosti IT projektů.

uživatele ovšem není ten, kdo by byl schopen přesné technické analýzy a specifikace všech požadavků, na které bude při vývoji muset být brán zřetel. Změny v požadavcích navíc nemusí nutně „způsobit" jen zadavatel, protože během vývoje softwaru může dojít například k legislativním změnám, které nebylo možné očekávat, ani přímo ovlivnit.

Rigorózní metody se považují za nevhodné pro současné projekty, jak je uvedeno v [Buchalcevová, 2005, Fowler, 2009], přesto se jimi při vývoje stále řídí mnoho vývojářských týmů. Ani Martin Fowler v [Fowler, 2009] důrazně nedoporučuje aplikaci čistě rigorózních metodik na současné projekty a uvádí pojem **pseudoiterativní** vývoj, který označuje týmy vyvíjející vodopádovými styly, ale považují je za styly iterativní. Jako obecné příznaky tohoto pojmu uvádí následující tvrzení:

- „My používáme jednu iteraci analýzy následovanou dvěma iteracemi návrhu."
- „Kód této iterace obsahuje spoustu chyb - ale na konci vývoje je odstraníme."

3.1 Vodopádový model

Vodopádový model je jedním z tradičních přístupů k řízení procesu, ve kterém se jednotlivé fáze životního cyklu prochází sekvenčně. Fáze životního cyklu jsou zobrazeny na obrázku 2, kdy se počet a složení fází vodopádového modelu mohou lišit, vždy však zůstává jeho sekvenční vlastnost, která udává, že lze začít fázi pouze v případě, že byla kompletně ukončena fáze předchozí. Stejný princip nastává při odhalení chyby či změně požadavku v nějaké z předchozích fází, což znamená, že je nutné postupovat sekvenčně až k fázi, která je změnou postižena.

Snahou analytiků při použití vodopádových modelů tedy je anticipovat všechny požadavky a co nejpřesněji je analyzovat v počátečních fázích vývoje. [Buchalcevová, 2005]

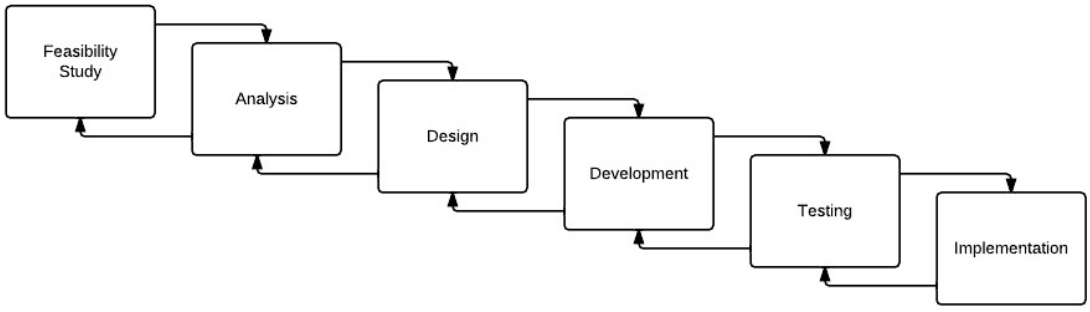

Figure 2: Fáze životního cyklu vodopádového modelu (Zdroj: www.gypthecat.com)

Práce uplatňuje agilní přístupy, proto zde nejsou detailně popsány jednotlivé rigorózní metodiky. Ucelený přehled agilních i rigorózních metodik může čtenář nalézt v [Buchalcevová, 2005].

4 Agilní metody

Realita vývoje, která představuje změny požadavků v jeho průběhu přiměla v roce 2001 skupinu 17 metodologů k vytvoření Agile Software Development Alliance

(http://agilealliance.org). Odborníci pocházeli z různých profesních odvětví, přesto se shodli na obecném postupu vývoje, jehož hlavní myšlenky jsou obsaženy v textu **Manifesto of Agile Software Development**, který je dostupný na adrese

http://www.agilemanifesto.org/ [Ambler, 2004]. Čtyři základní principy jsou následující:

1. **Jednotlivci a interakce** před procesy a nástroji
2. **Fungující software** před vyčerpávající dokumentací
3. **Spolupráce se zákazníkem** před vyjednáváním o smlouvě
4. **Reagování na změny** před dodržováním plánu

Jednotlivé principy jsou úmyslně strukturovány na pravou a levou část, z nichž jsou obě důležité. Levá strana je ovšem v agilních metodikách upřednostňována.

Redukované srovnání agilního a rigorózního přístupu je v tabulce 1, které je převzato z publikace [Buchalcevová, 2005]. Tabulka reflektuje odlišné pohledy metodik na hlediska, která vývoj softwaru provází. Uchýlením k jedné konkrétní metodice však nemusí znamenat nutně dodržování všech zmíněných principů, je například možné zahrnout do rigorózně vedeného procesu určitou agilní metodiku (nebo její část). Scott Ambler v [Ambler, 2004] uvádí příklad, kdy zahrnul agilní modelování do projektů vedených metodikou **Rational Unified Process**.

Autor práce podotýká, že se autoři publikací v uvedeném srovnání liší, například Scott Ambler v [Ambler, 2004] upřednostňuje agilní metody pro týmy všech velikostí. Martin Fowler [2] v [Fowler, 2009] obecně doporučuje aplikování iterativního vývoje, ale s možným přizpůsobením pro konkrétní projekt a situaci.

[2] Martin Fowler se jako jeden ze 17 metodologů podílel na vzniku „agilní aliance" a z ní plynoucího manifestu agilního softwarového vývoje.

Hledisko	Rigorózní metodiky	Agilní metodiky
náplň metodiky	procesy, zaměřují se na explicitní znalost a pohlíží na lidi jako na sekundární faktor	praktiky, zaměřují se na „tacit" znalosti, chápou lidi jako klíčové faktory úspěchu
podrobnost metodiky	procesy a činnosti jsou popsány velmi podrobně	definovány tzv. sotva dostatečná metodiky, která se zaměřuje na činnosti, které vytvářejí hodnotu a eliminuje činnosti, které hodnotu nepřinášejí
kvalita	zaměření na kvalitu procesů a předpoklad, že kvalitní procesy povedou ke kvalitnímu výsledku	zaměření na hodnotu pro zákazníka a vysokou kvalitu produktu
předvídatelnost	předpokládá předvídatelnost budoucnosti, důraz na anticipaci	předpokládá nepředvídatelnost budoucnosti, důraz na adaptaci na změny (přírůstkové shromažďování požadavků, plánování pro iteraci)
změny	změny podléhají řízení změn a je snaha změny minimalizovat	snaha změny umožnit a využít je, umožňují zákazníkům přehodnotit své požadavky s ohledem na nové znalosti
participace zákazníka na projektu	jen v počátečních a koncových fázích, pod podpisu dokumentu specifikace požadavků řízení přebírá tým technologických pracovníků	přesun nositele řízení z týmu na zákazníka, zákazník je řídícím subjektem během celého projektu, při každé iteraci zákazník může měnit priority funkcí
kvalifikace lidí	stačí standardní jedinci	důraz na schopnosti, znalosti a dovednosti lidí
dokumentace	rozsáhlá dokumentace	podstatná není dokumentace, ale pochopení
modelování	velký důraz na modelování, zejména modelování předem - big design up front, potom se zmrazí požadavky	agilní modelování, při modelování nejde o model jako takový, ale o akt modelování, smyslem modelování je komunikace

Table 1: Srovnání agilního a rigorózního přístupu [Buchalcevová, 2005]

4.1 Agilní modelování

Modelování je jednou ze základních praktik, které by měly provázet agilní i rigorózní vývoj. Metodika agilního modelování pochází od Scotta Amblera, který definuje jeho tři hlavní cíle:

- Definovat kolekci hodnot, principů a zásad pro efektivní modelování
- Popsat, jak aplikovat modelovací techniky na týmy řídící se agilními vývojovými

procesy

- Umožnit aplikaci „téměř agilního" přístupu do softwarového vývoje, zejména u týmů, které se řídí přístup RUP, nebo EUP

Agilní modelování neznamená pouze vytváření vizuálních diagramů a složitých dokumentů, zahrnuty jsou i CRC [3] karty a textové popisy podnikových směrnic. K modelování není nutné používat CASE nástroje, užívány jsou běžné tabule, papír, nebo lepící bločky. Metodika vychází z extrémního programování od Kenta Becka a přebírá mnohé z jejích principů, modelovány tak jsou jen ty problémy, které jsou opravdu třeba a jen tak rozsáhle, aby dostačovaly (nikoli snaha o úplný popis, nebo dokumentaci).

Z předchozích tvrzení plyne, že modely netvoří dokumentaci softwaru, ale pomáhají k jeho pochopení a porozumění mezi vývojáři. Investice zákazníka se tak dostávají přímo do produktu a ne do vytváření rozsáhlých dokumentací a modelů. [Ambler, 2004]

5 Datové modelování

Datové modelování není přímo metodikou vývoje, je to nástroj, který provází rigorózní i agilní metodiky. Soustředí se na vymezení datových struktur ve zkoumané doméně a vazeb mezi nimi. V datových modelech nejsou přítomny žádné prvky softwarových komponent, nezachycují chování dat ani jejich konkrétní reprezentaci v úložišti dat. [Merunka, 2008] Datové modely lze rozdělit na **konceptuální**, **logické** a **fyzické**.

Model	Význam
Konceptuální	Konceptuální modely (nazývané také doménové) slouží ke komunikaci se zadavatelem projektu. Jsou vysoce abstraktní a užívají se zejména v počátcích vývoje.
Logický	Logické modely specifikují doménovou strukturu, kterou zobrazují pomocí entit, datových atributů (nikoli však datové typy) a vztahů mezi nimi. Logické datové modely se v agilních projektech příliš nevyužívají.
Fyzický	Fyzický datový model definuje vnitřní datové schéma databáze popisem tabulek, jejich sloupců, datových typů a vzájemných vztahů. Fyzický model je aplikován v programu vyvinutém v rámci této práce.

Table 2: Druhy datových modelů [Ambler, 2003c]

Linie mezi logickým a fyzickým modelem bývá velmi tenká, protože oba typy modelu popisují stejnou datovou strukturu. Liší se však zejména svým účelem, kdy logický datový model může být konzultován se zákazníkem či uživatelem, ale fyzický model slouží především ke komunikaci mezi vývojáři. Míra detailu je značně vyšší u fyzických modelů, které mohou zobrazovat i implementační detaily, jako asociativní tabulky, nutné v relačně databázových systémech. [Ambler, 2003c]

[3] Class Responsibility Collaborator je označení pro techniku, jak zachytit odpovědnosti a spolupráci třídy s ostatními třídami v systému. CRC technika nachází široké uplatnění například v eXtrémním programování. [Ambler, 2004]

Part IV
UML

UML je zkratkou pro Unified Modeling Language, což je modelovací jazyk umožňující vizuálně popsat softwarové systémy. Jedná se o soustavu mnoha typů diagramů, pomocí nichž lze vyjádřit podstatu, fungování či strukturu softwaru, nebo jeho části. Důležitou vlastností UML oproti konkurenčním modelovacím jazykům je jeho dostatečná abstrakce tak, že lze vytvářet diagramy nezávisle na finálně použitém programovacím jazyce. Diagramy tak mají vlastnost přenositelnosti například při změně platformy, nebo přechodu na jinou technologii v pozdní fázi vývoje. Dále diagramy usnadňují komunikaci mezi členy týmu, kdy spolu mohou prostředníctvím diagramu komunikovat lidé s rozdílným profesním zaměřením.

Použití UML lze rozdělit do skupiny **konceptuálního** a **softwarového** modelování. V softwarovém modelování jsou prvky modelu přímo vázány na prvky softwarového systému. Model tak, stále však abstraktně, popisuje technickou strukturu systému a z uvedeného rozdělení je dle [Fowler, 2009] častější. Konceptuální model se naopak zaměřuje na popis zkoumané domény a nevěnuje přílišnou pozornost elementům softwaru, jeho účelem je porozumění a diskuse nad zkoumanou problematikou.

O vznik a rozvoj UML se stará organizace OMG (Object Management Group), která první verzi UML vydala v roce 1997. Obsahem první verze byl sourhn diagramů, které vycházely z již existujících návrhů a jejím hlavním úkolem bylo sjednotit tyto diagramy do notace, kterou bude možné používat globálně. V současnosti je jazyk UML ve verzi 2.4.1, která přinesla revizi stávajících diagramů a zavedla nové typy, jako například **diagram časování**, **diagram balíčků**, nebo **diagram interakcí**. Práce užívá UML vždy ve verzi 2.0 (výjimkou je zahrnutí agregačního vztahu).

Zhodnocení současného stavu UML podle [Ambler, 2004] je zobrazeno v tabulce 3. Nedostatek nekompletnosti v podobě absence notace pro datové modelování je částečně možné nahradit zavedením profilu, který je popsán v kapitole 8.

Výhody	Nevýhody
Přijato širokou veřejností	Modelovací nástroje často nepodporují plně standard UML, nebo notaci neimplementují správně.
Rozsáhlá podpora nástrojů	Zatím nekompletní (chybí např. notace pro modelování UI a datové modelování)
Konzistentní notace modelů	Kompletní notace je příliš složitá. Pro většinu případů postačuje malá podmnožina notace.

Table 3: Zhodnocení UML

6 Způsoby užití

Užití UML diagramů, jak je popsáno v [Fowler, 2009], lze rozdělit do tří kategorií.

6.1 Náčrtek

Náčrtek označuje Martin Fowler v [Fowler, 2009] za nejpoužívanější styl zápisu. Náčrtek lze použít jak jako dopředné inženýrství, kdy vytvoření modelu předchází napsání kódu, tak jako zpětné inženýrství, kde model vzniká z existujícího kódu a slouží tak k jeho lepšímu pochopení. Základní vlastností náčrtku je vybrání pouze důležitých částí modelovaného systému. Při vytváření diagramu zpětným inženýrstvím tak zahrneme do diagramu pouze kritickou část kódu, kterou chceme vysvětlit. Tvorba náčrtku by měla být rychlá a často se k jeho tvorbě užívá tabule místo sofistikovaných softwarových nástrojů. Jejich cílem tak je především umožnit rychlou a snadnou komunikaci v rámci týmu vývojářů.

6.2 Detailní návrh

Detailní návrh, v anglickém jazyce „blueprint", je na rozdíl od náčrtku přesný a detailní popis systému. Při dopředném inženýrství vytvoří analytik diagram, který bude programátor implementovat. Tvorba takto detailního diagramu vyžaduje zkušeného vývojáře či analytika, který bývá často i vedoucím celého týmu vývojářů [Fowler, 2009] a užití CASE nástroje. Návrh nemusí nutně obsahovat kompletně celý popisovaný systém, lze do něj zahrnout pouze jeho část, ovšem popsat ji kompletně a dodržet vysokou míru detailnosti. Práce uplatňuje vlastnosti agilních metodik, což znamená, že tvorba detailních návrhu není primárním cílem vyvíjeného softwaru. Důraz je naopak kladen na jednoduchost a rychlost jejich tvorby.

6.3 Programovací jazyk

Využití UML jako programovacího jazyka je úzce spjato s CASE nástroji, které umožňují v různých formách generovat kód z diagramu. V takovém stavu se UML diagramy stávají zároveň zdrojovým kódem softwaru, proto již nemá smysl hovořit o dopředném či zpětném inženýrství. S použitím UML jako programovacího jazyka souvisí přístup Model Driven Architecture, který pochází od společnosti OMG.

7 Diagram tříd

Diagram tříd je nejpoužívanějším diagramem z UML, umožňuje popsat strukturu a vztahy mezi objekty ve vytvářeném systému. Popisuje jejich datové složky a operace, které vykonávají včetně možnosti aplikace omezujících podmínek.

Třída

Hlavní komponenta diagramu tříd je třída. Je značena obdélníkem rozděleným na tři části, které obsahují název třídy, atributy a operace, které vykonává. Atribut má povinný pouze svůj název, lze ho však uvést s modifikátorem přístupnosti pomocí znaménka „+" pro veřejné atributy a „-" pro privátní. Po názvu atributu může následuje dvojtečka

a jeho datový typ včetně nepovinného omezení. Atributy lze zapisovat i pomocí asociací, které umožňují využít kardinalit na obou koncích asociace a názvu atributu, který se zapíše na konec asociace. Vlastníkem atributu je zdrojová třída.

Obdobně se zapisují operace, které se skládají z modifikátoru přístupnosti, názvu operace, parametrů a návratové hodnoty.[Fowler, 2009]

Vztahy

Závislosti tříd se v UML vyjadřují pomocí vztahů, z nichž má každý podobu plné či přerušované čáry ze zdrojové třídy do cílové a mohou být zakončeny určitým obrazcem. Začátek i konec vztahu může být označen kardinalitami, uprostřed je možné uvést slovní popis vztahu včetně šipky ve směru jeho čtení.

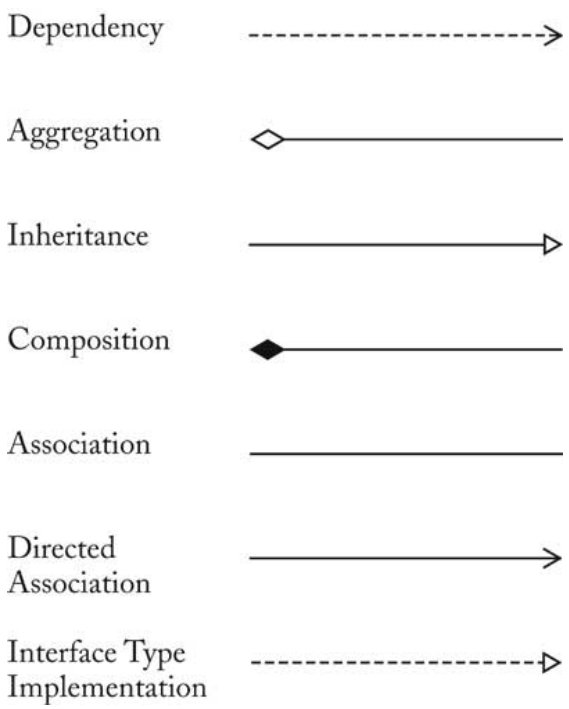

Figure 3: Notace UML vztahů (Zdroj: www.sable.mcgill.ca)

Vztah	Anglický termín	Význam
Asociace	Association	Nejobecnější vyjádření vztahu mezi dvěma třídami. Vyjadřuje pouze určitou závislost zdrojového objektu na cílovém.
Agregace	Aggregation	Označuje vztah, kdy je cílový objekt **součástí** zdrojového objektu. Příkladem agregace je motor, který je součástí automobilu. V UML 2.0 není agregace podporována, autor ji ovšem z důvodu dodržení profilu zahrnul do práce a výsledného projektu.
Kompozice	Composition	Silnější vztah, než u agregace. Existence cílového objektu je podmíněna existencí zdrojového objektu. Příklad kompozice je bod, který tvoří střed kružnice. Se zánikem kružnice zaniká i existence jejího středového bodu.
Závislost	Dependency	Vztah závislosti nastává, jak píše Martin Fowler v [Fowler, 2009], v momentě, kdy změna v cílovém objektu způsobí potřebu změnit i objekt zdrojový.
Dědičnost	Inheritance	Pomocí dědičnosti můžeme určit, že zdrojový objekt je podtypem cílového objektu.

Table 4: Význam jednotlivých UML vztahů

8 Profil

K modifikaci UML diagramů slouží profily. Pomocí profilu můžeme do UML prvky pouze přidávat, aby byla zajištěna funkčnost stávajících diagramů. Profilem lze podrobněji přizpůsobit UML na cílově vyvíjenou platformu. [Buchalcevová, 2005]

Modifikace probíhá s využítím stereotypů, které se značí názvem stereotypu uzavřeným ve dvojitých ostrých závorkách. Práce využívá profil vytvořený Scottem W. Amblerem, jehož detailní specifikaci lze nalézt v [Ambler, 2003a].

9 CASE nástroje

Computer Aided Software Engineering je označení pro nástroje, které usnadňují činnosti softwarového inženýrství. CASE nástroje se nejčastěji skládají z grafického UI, podporuje tvorbu modelů, případně kontrolu jejich validity a generátory zdrojových kódů. Práce má za cíl vyvinout software, který lze označit jako jednoduchý CASE nástroj.

Part V
Objektově orientovaný přístup

10 Vývoj objektově orientovaných jazyků

Přístup k vývoji softwaru se od počátků ve 20. století velmi změnil. V období užívání procedurálních jazyků byl vývoj softwaru ovlivněn jinými faktory, než dnes. Vývojáři nebyli pod takovým časovým tlakem ze strany zadavatelů, software nedosahoval takové velikosti a složitosti, jako dnes. V 90. letech 20. století však potřeba po složitějších programech a růst výpočetního výkonu tehdejších počítačů způsobila, že se začaly prosazovat první objektově orientované jazyky. Čistě objektově orientované jazyky, mezi které řadíme například jazyk Smalltalk, Simula nebo LISP, byly sice dostupné již několik let [4], jelikož se jedná o jazyky s mnohem vyšší úrovní abstrakce, jejich výkon nebyl na tehdejších počítačích dostatečný. [Merunka, 2008]

Z toho důvodu vznikly smíšené objektové jazyky 12, které jsou dnes stále nejpoužívanější. Do této kategorie se řadí jazyky jako je Java, C# nebo C++. Smíšené jazyky si z původní filozofie OOP vzaly jen část vlastností a ostatní doplnily o vlastnosti známé z procedurálních jazyků.

11 Čistě objektový přístup

Základní filozofie čistě objektově orientovaného přístupu (dále jen OOP) je na první pohled velmi jednoduchá a je založena na třech základních prvcích, ze kterých se celý objektový systém skládá. Chování systému pak určují vazby a operace (zasílání zpráv) mezi takovými prvky. Základními prvky jsou:

11.1 Objekt

Objekt si lze představit jako zapouzdření vlastností a chování určitého prvku reálného světa. Příkladem objektů mohou být živé bytosti, předměty ale i abstraktní pojmy, se kterými potřebujeme jistým způsobem pracovat. Je důležité poznamenat, že podoba objektu závisí situaci, ve které ho vytváříme. Jinak například bude vypadat objekt psa, který budeme popisován do evidence ošetřených zvířat u veterináře a jinak pro soutěž agility. Je tedy chybou hledat univerzální popis konkrétního objektu, vždy je třeba objekt upravit dle aktuálních potřeb. [Merunka, 2008]

11.2 Zpráva

Zpráva je způsob, kterým objekt reaguje na požadavky jiných objektů. Mohou sloužit k získávání informací či nastavování údajů o objektu. Získávání dat probíhá způsobem, kdy zašleme objektu zprávu, jejímž výsledkem je požadovaná informace. K nastavení dat předáme zprávě parametr, který obsahuje hodnotu, jež chceme objektu nastavit.

[4] Smalltalk vznikl ve stejném roce, jako procedurální jazyk C.

11.3 Metoda

Objekt může ovšem provádět mnohem komplexnější operace, než pouze vybírání a nastavování dat. Může se jednat o složité algoritmické výpočty, nebo samostatné programy. Zprávy, které spouští takové operace, nazýváme metody.

12 Smíšený přístup

Smíšené programovací jazyky, jak již bylo zmíněno výše, vznikly z potřeby zahrnutí alespoň části z objektového paradigmatu a zároveň zachování vysokého výkonu, na který byly vývojáři zvyklý z jazyků procedurálních. Popularita smíšeného přístupu rychle vzrostla a dodnes je tento přístup nejpoužívanější. Inspirace smíšených jazyků u procedurálních předků ovšem způsobila, že si s sebou jazyky přinesly mnoho neduhů, které se ani s postupem času nepodařilo odstranit. Proto se dodnes ve smíšených jazycích setkáváme například s těmito vlastnostmi:

- Klíčová slova
- Modifikátory přístupnosti
- Rozhraní
- Výčtové typy
- Omezený polymorfismus

Mezi výhody smíšených programovací jazyků bezesporu patří, byť nepřímo, široká nabídka dostupných IDE [5], školení, dokumentace a materiálů, které výrazně usnadňují a zlevňují vývoj. Populární smíšené jazyky, jako je C# a Java, jsou udržovány a vyvíjeny velkými korporacemi, které zajišťují širokou podporu knihoven a nativního GUI [6], které lze na podporovaných platformách využít. Výše zmíněné výhody stále drží smíšené jazyky na vrcholu popularity a je otázkou kdy a zda se to někdy změní. [Merunka, 2008]

13 Návrhové vzory

Nedostatky současných programovacích jazyků a stále se opakující problémy, které musí vývojáři řešit, vedly ke vzniku návrhových vzorů. Návrhové vzory tvoří skupinu doporučených postupů, jak řešit určitý problém. Lze je implementovat v jakémkoli objektovém jazyce a jejich chování přizpůsobit vlastnostem použitého jazyka. Nejpoužívanějších návrhových vzorů dnes existuje více než 50. [Ambler, 2003a]

[5] Integrated Development Environment je software, ve kterém programátor píše a kompiluje kód. V dnešní době mají IDE podporu doplňování kódu, refactoringu, pluginů a dalších funkcí.

[6] Graphic User Interface je označení pro uživatelské rozhraní, kterým probíhá komunikace mezi uživatelem a programem.

Za průkopníky návrhových vzorů jsou považováni autoři publikace Design Patterns: Element of Reusable Object Oriented Software [7] [Erich Gamma, 1995], kteří návrhové vzory rozdělili do třech základních kategorií podle jejich účelu:

Strukturální vzory se starají o skládání a vytváření hierarchií tříd a objektů, což vede k lepší přehlednosti a udržitelnosti systému.

Tvořivé vzory zajišťují vytváření správných objektů, volbu správného postupu vytvoření a často až za běhu programu.

Behaviorální návrhové vzory řídí chování objektů a jejich spolupráci v rámci systému.

Takové rozdělení ale není podle [**?**] dostatečné, proto vzniklo několik dalších, jako například *POSA*, nebo *A Pattern Language*. Následující vývoj návrhových vzorů se ubíral a stále ubírá například problematikami vícevláknového zpracování, vývoje podnikových systémů, zpracování dat či prezentačními modely.

Praktická část práce implementuje několik návrhových vzorů, z nichž zde budou popsány ty nejdůležitější.

13.1 Decorator

Prvním vzorem, který je v práci implementován, je dekorátor. Dekorátor spadá do kategorie strukturálních vzorů a slouží k dynamickému přidávání a odebírání vlastností objektu beze změny jeho původního chování. Nabízí alternativu k dědičnosti, která neumožňuje jejich dynamické přidávání, protože se aplikuje na třídu, nikoli na objekt.

Principem dekorátoru je vytvoření abstraktní třídy **Decorator**, která bude dědit od stejného potomka, jako třída, kterou chceme dekorovat. Od třídy **Decorator** lze dědičností vytvořit konkrétní dekorátory, které budou obsahovat dodatečné vlastnosti. Jelikož třída **Decorator** obsahuje i původní dekorovaný objekt, zpravidla přijímaný skrze konstruktor, chování původního objektu bude zachováno a konkrétní implementace dekorátorů přidají potřebné vlastnosti. [Erich Gamma, 1995]

[7] Čtyřem autorům publikace se začalo říkat Gang of Four, podle čehož se přezdívá i skupině vzorů, které publikovali - GOF.

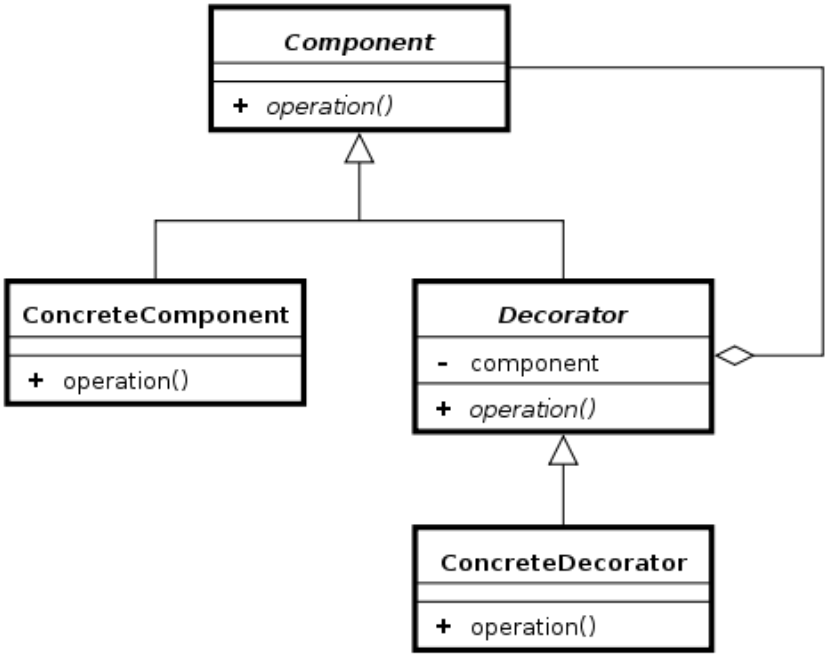

Figure 4: Diagram návrhového vzoru Dekorátor (Zdroj: en.wikipedia.org)

13.2 Strategy

Strategie je zástupcem behaviorálních návrhových vzorů a umožňuje za běhu programu měnit algoritmus za různé jeho implementace.

Funguje na jednoduchém principu, kterým je abstrakce protokolu [8] algoritmu do rozhraní (v čistě objektových jazycích do třídy). Konkrétní algoritmy musí implementovat předešlou abstrakci a jsou následně nastaveny třídě skrze konstruktor, metodu, nebo obojí. Třída využívající algoritmus nepotřebuje informaci o aktuálně využívaném algoritmu a je možno ho za běhu programu měnit. [Erich Gamma, 1995]

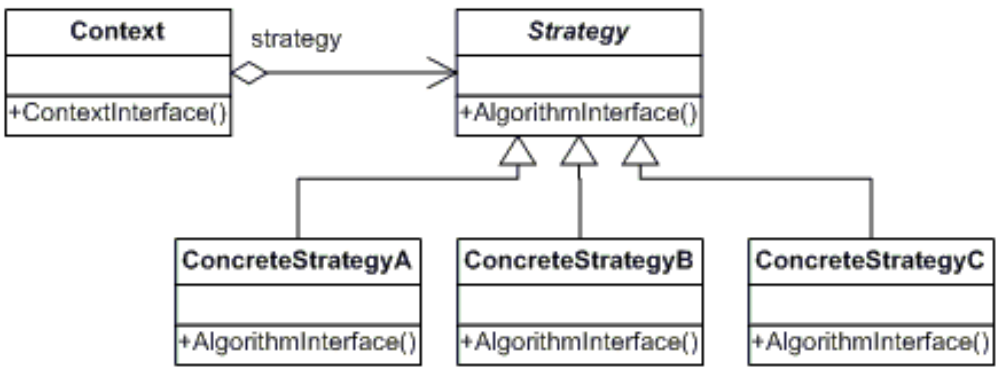

Figure 5: Diagram návrhového vzoru Strategie (Zdroj: www.dofactory.com)

[8] Protokol je množinou všech zpráv, které je možné objektu zaslat. [Merunka, 2008]

13.3 Mediator

Mediator patří do kategorie behaviorálních vzorů a zprostředkovává komunikaci mezi objekty, které spolu nejsou přímo svázané [9]. Objekty spolu mohou komunikovat skrze zprávy, více v kapitole 11.2, poté je ale nutné udržovat u odesílatelů instance konkrétních objektů, se kterými mají komunikovat. Vzniku těchto vazeb zamezí návrhový vzor Mediator, který je vložen jako prostředník mezi komunikující objekty. Mediator je implementován v praktické části 18.3.1.

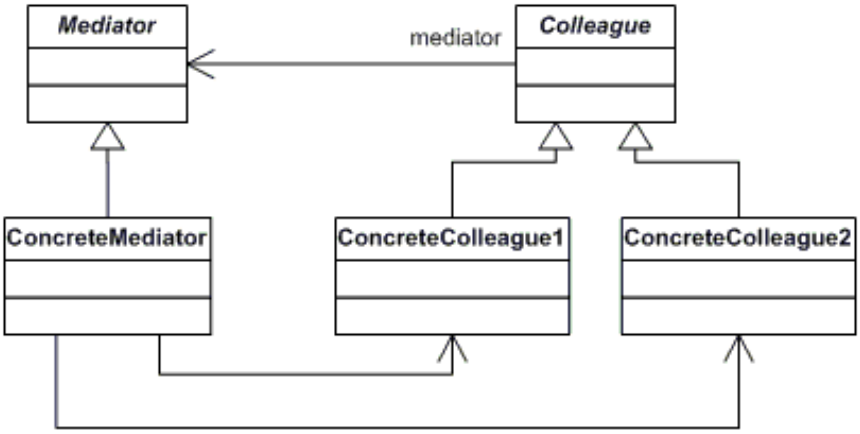

Figure 6: Diagram návrhového vzoru Mediator (Zdroj: www.dofactory.com)

13.4 Command

Účelem návrhového vzoru Command je zapouzdřit operaci, kterou jde poté spouštět z mnoha na sobě nezávislých objektů. Vzor Command se implementuje pomocí rozhraní, které definuje metodu *Execute()* (lze využít i parametr, viz 18.3.2). Třídy, které takové rozhraní implementují poté rozhodují o tom, která metoda se při zavolání spustí. Implementací Command vzoru je mnoho, někteří ho implementují i s funkcí, kdy se lze vracet k předchozím krokům [Freeman – Freeman, 2004] či metodou *CanExecute()*, která podle parametru určí, zda se má operace spustit.

[9] V OOP terminologii se tomuto jevu říká „tight coupling".

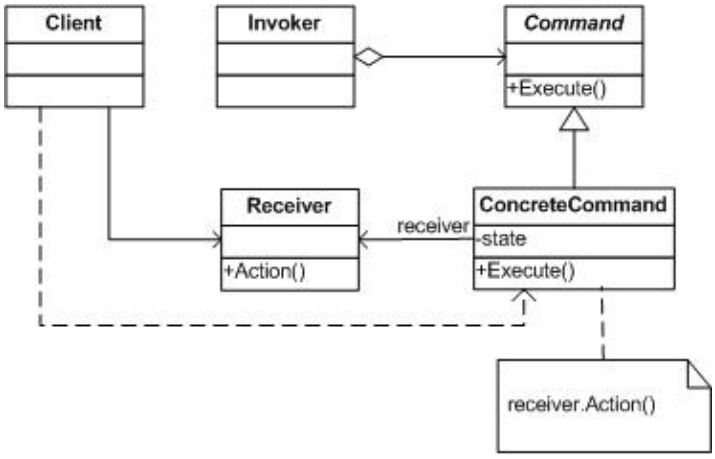

Figure 7: Diagram návrhového vzoru Command (Zdroj: www.dofactory.com)

14 Model View ViewModel

Model View ViewModel (dále jen MVVM) je architektonický vzor, nespadá tak do předchozí kategorie návrhových vzorů. Architektonické vzory se od návrhových vzorů liší svou větší obecností. Uvádějí doporučený postup pro strukturování celého projektu, na rozdíl od návrhových vzorů, které řeší implementaci konkrétních problémů. Modelovací software v této práci byl vyvinut na platformě WPF, kde je standardně užívaným architektonickým vzorem právě MVVM. MVVM rozděluje aplikaci na tři hlavní části, které jsou ilustrovány na níže uvedeném obrázku 8. Základním principem je to, že jednotlivé vrstvy musí být odděleny natolik, aby o sobě navzájem vůbec nevěděly. To umožňuje snazší testovatelnost (možnost vrstvy testovat samostatně) a udržitelnost (změny v jedné vrstvě by měly jen minimálně ovlivnit chod jiné vrstvy). [Blaine Wastell, 2014]

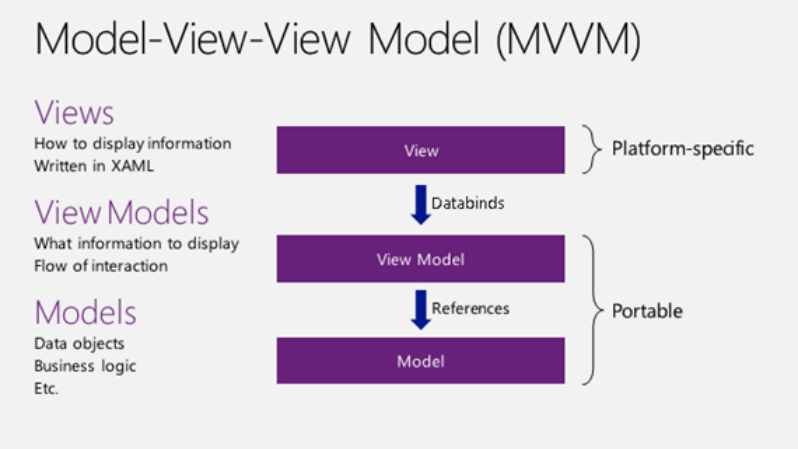

Figure 8: Komunikace vrstev MVVM (Zdroj: www.msdn.microsoft.com)

14.1 Model

Vrstva modelu je vrstvou, kde se uchovávají data, která se poté propagují do uživatelské rozhraní. Vrstvu mnozí označují za *Domain Model* [10], což znamená, že obsahuje třídy, které mapují data uložená v databázi (entity) a logiku, kterou se data z databáze načítají a ukládají. V rozsáhlejších projektech bývá vrstva rozdělena na několik podvrstev, čímž se docílí ještě vyšší dekomponovanosti aplikace.

14.2 View

Pod touto vrstvou se nachází veškeré uživatelské rozhraní aplikace. Uplatňuje se zde technika **data binding**, která umožňuje přímo navázat vlastnosti z uživatelského rozhraní na vlastnosti z vrstvy ViewModelu. Jako jediná vrstva je platformě závislá, což znamená, že při správném návrhu aplikace je možno pro různá zařízení (desktop, mobilní telefon, tablet atd.) ponechat ostatní vrstvy nedotčené a změny provádět pouze zde.

14.3 View Model

ViewModel je nejkontroverznější vrstvou celého vzoru. Je prostředníkem, skrze kterého komunikují dvě předchozí vrstvy. View Model obsahuje konkrétní modely a propaguje jejich data do uživatelského rozhraní. Notifikuje uživatelské rozhraní o změnách, které v datech nastaly, což je v .NET frameworku realizováno rozhraním **INotifyPropertyChanged**, které vychází z návrhového vzoru Observer. Spouštění operací se ve View Modelu realizuje vzorem Command, který zajišťuje dostatečnou abstrakci, platformní nezávislost a testovatelnost, viz 20.3.

14.4 Srovnání

Nabízí se srovnání s ostatními architektonickými vzory, které bývají často zaměňovány a spojovány. Srovnány budou vzory MVC, MVVM a MVP, jejichž hlavní rozdíl tkví ve třetí zprostředkující vrstvě, která jsou podrobněji vymezeny níže.

14.4.1 Controller

Controller rozhoduje o tom, která část UI bude zobrazena. Příkladem může být hlavní stránka webové aplikace, kterou může obsluhovat **HomePageController** a ten zprvu zobrazí úvodní stránku (z vrstvy View). Reaguje na události z UI a vrací příslušné prvky uživatelského rozhraní. V Controlleru by se neměly provádět žádné logické operace, vyjma rozhodovací logiky vracení UI. Klíčové je to, že obsluhuje více různých UI a neobsahuje přímo data uživatelského rozhraní. [Fowler, c2003]

14.4.2 View Model

View Model obsahuje data, která jsou propagována do uživatelského rozhraní. K propagaci potřebuje techniku **data binding** a implementaci notifikace o změnách dat, která obsahuje. Praxí je, že jeden ViewModel obsluhuje jednu část uživatelského rozhraní

[10] Více o doménovém modelu v [Fowler, c2003]

(například jeden formulář, nebo okno aplikace), ale není to podmínkou. V aplikacích, které kladou velký důraz na bohaté uživatelské rozhraní se v jedné části uživatelského rozhraní může vyskytovat i několik View Modelů. Takových případem je modelovací software vyvinutý v této práci.

14.4.3 Presenter

Presenter je velmi podobný principu View Modelu a používá se zejména tam, kde není dostupný **data binding** a není možné se přímo spojit s uživatelským rozhraním. Přímé spojení je nahrazeno rozhraním, které definuje operace, které bude View provádět. Rozhraní je následně implementováno [11] v příslušného UI celku. Uplatnění nachází například u technologie Windows Forms, nebo v php frameworku Nette. Nevýhodou Presenteru je nutná přítomnost abstrakce skrze rozhraní, což s sebou přináší více kódu, který je třeba v aplikaci udržovat.

Part VI
Vývoj programu

15 Struktura projektu

V této kapitole se nachází stručný popis a záměr, který vedl k vytvoření struktury, která je prezentována na obrázku 9. Projekt se řídí konvencí pojmenování, která zahrnuje název projektu **UmlDiagramDesigner** oddělený tečkou a následovaný názvem konkrétní části projektu. Vyjma projektu Client názvy projektů odpovídají použitým jmenným prostorům [12].

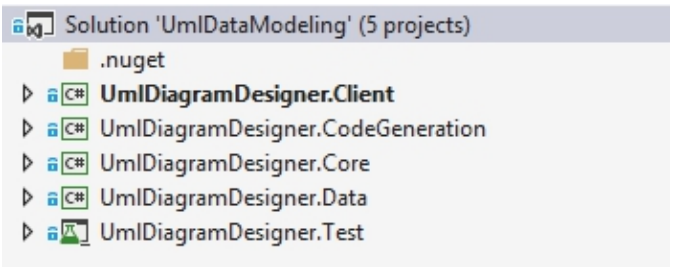

Figure 9: Struktura projektu ve Visual Studiu 2013 (Zdroj: vlastní)

Core

Knihovna Core byla zavedena do projektu až v pozdní fázi technikou refaktoringu, kterou autor abstrahoval části kódu použité v ostatních projektech. Core knihovnu referencují všechny ostatní projekty, protože obsahuje základní výčtové typy, abstrakci grafu a potřebné algoritmy.

[11] Ve Windows Forms technologii rozhraní implementuje code-behind třída příslušného UI.
[12] Projekt Client využívá kořenový jmenný prostor UmlDiagramDesigner

Data

Data projekt je knihovna, která slouží jako prostředník mezi diagramy z klientské aplikace a generováním kódu. Diagram z View Modelů 14.3 je mapován na třídy v této knihovně. Z namapovaných objektů je následně generován kód do všech podporovaných jazyků. Je tím docíleno abstrakce generování kódu na konkrétní vizuální podobě diagramu, protože třídy v projektu Data neobsahují žádné informace o tom, jak diagram vypadá.

CodeGeneration

Z názvu zřejmý cíl projektu CodeGeneration je generování kódu z modelu. Důležitým faktem je, že projekt referencuje pouze projekty Core a Data, nikoli projekt Client. Testovat generování kódu je tedy možné zcela bez přítomnosti klientské aplikace. Knihovny Data, Core a CodeGeneration se dají využívat zcela nezávisle na ostatních projektech a je tak možné je zahrnout do jiných prací či programů.

Client

Samotná aplikace je obsažena v projektu Client. Aplikace je vytvořena na platformě WPF a projekt obsahuje všechny prvky uživatelského rozhraní a logiky, kterou provádí.

Test

Projekt Test je vytvořen z šablony Unit Test Project, což je knihovna, která má automaticky přidané reference na knihovny testovacího frameworku UnitTestFramework. Zde probíhají odděleně testy všech dílčích projektů.

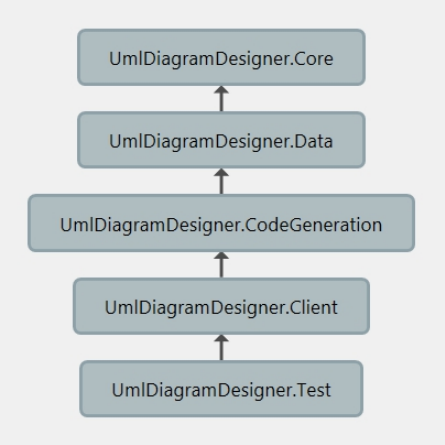

Figure 10: Závislosti jednotlivých projektů. (Zdroj: vlastní)

16 Rozbor programu

16.1 Uživatelské rozhraní

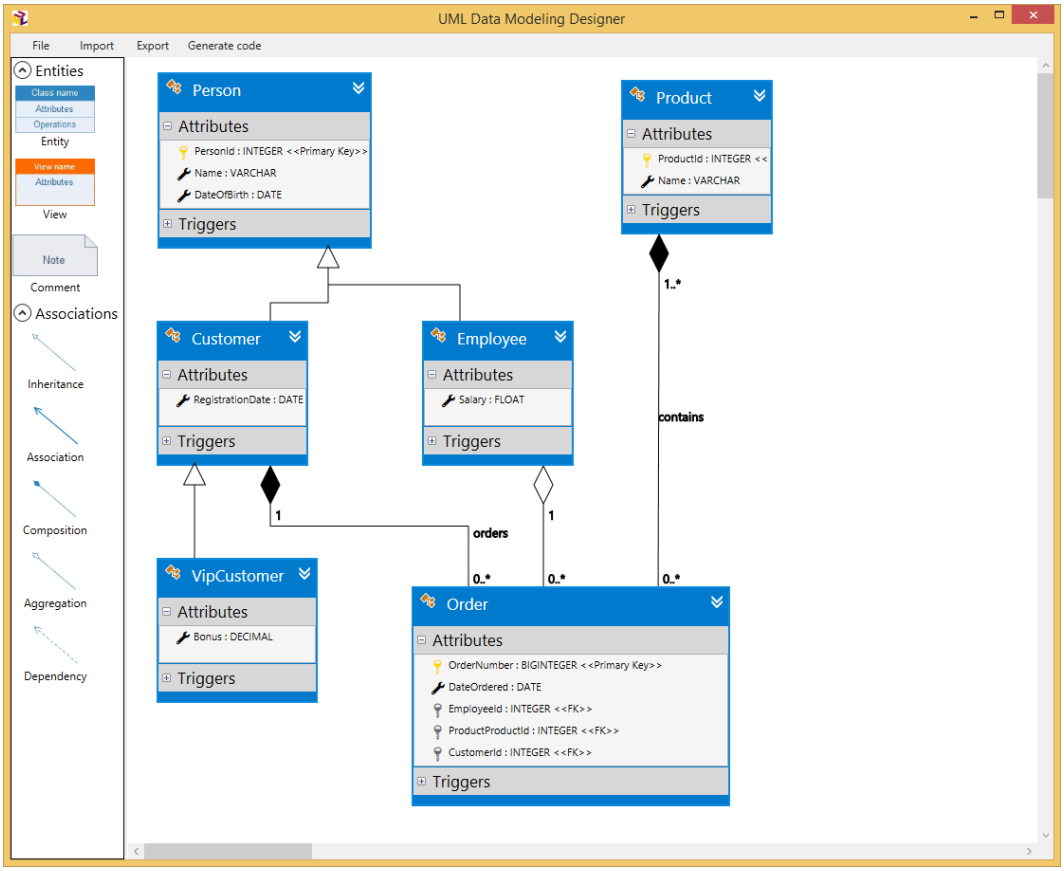

Figure 11: Náhled UI hlavní obrazovky (Zdroj: vlastní)

16.2 Popis funkcí

Settings

Volba *Settings* se nachází na navigační liště v záložce *File* a obsahuje několik vizuálních nastavení, které se automaticky v reálném čase při změně aplikují na právě otevřený model. Všechna nastavení a rozpoložení programu jsou automaticky ukládána do lokálního nastavení v adresáři aktuálně přihlášeného uživatele. K načtení a editaci nastavení je využita technika **data binding** a propojení skrze třídu *BindableSettings*, která dědí od třídy *Binding*.

Toolbox

Toolbox, neboli panel s prvky, které lze umisťovat na diagram, je velmi důležitou komponentou v programu. Toolbox rozhoduje o tom, který prvek bude na diagramu vytvořen a notifikuje o tom plochu diagramu. Všechny prvky, které jsou z toolboxu vybrány nesou pouze informaci o názvu, obrázku, popisu a datovém typu, který se má na ploše vytvořit. O tvorbu objektu na ploše se stará třída *DiagramItemFactory*, která

z datového typu pomocí reflexe zavolá konstruktor bez parametrů, čímž získá instanci požadovaného prvku.

Canvas

Canvas v programu reprezentuje celou plochu, na kterou může uživatel umisťovat prvky modelu. Mezi jeho nejdůležitější části patří kolekce všech objektů, které obsahuje a rozměry. Rozměry plochy jsou poté užívány k validaci pohybu prvků, aby se zamezilo tomu, že uživatel programu posune prvek na místo, odkud jej nebude moci dostat zpět.

Míra složitosti a komplikace při implementaci přibližování v kombinaci s výše zmíněnou validací způsobily, že není dostupná funkce přiblížení modelu. Jako náhrada této funkce slouží posouvací lišty, pomocí nichž je možné vytvářet i modely větších rozměrů.

17 Core knihovna

Zcela nezávislé komponenty aplikace byly umístěny do projektu UmlDiagramDesigner.Core, který představuje knihovnu, již si referencují všechny ostatní projekty. Obsah tak tvoří především abstrakní položky, jako jsou rozhraní, které jsou implementovány v konkrétních projektech. Další významnou částí Core knihovny jsou výčtové typy, které musí být shodné ve všech částech aplikace a třída *Messenger*, popsaná v kapitole 18.3.1.

Extrahování částí kódu aplikace do společného projektu usnadňuje budoucí úpravy a snižuje riziko vzniku nekonzistencí, protože lze úpravy provádět z jednoho místa aplikace. Další podstatnou výhodou separace do samostatné knihovny je ta, že při nasazení programu u klientů lze aktualizaci knihovny provést bez nutnosti kompilace celého programu. Pokud nedojde ke změně pojmenování tříd, metod či jejich přesunu do jiných jmenných prostorů, lze aplikaci aktualizovat pouhým nahrazením příslušné knihovny.

Algoritmus 1 Rozhraní definované v Core knihovně

```
1  public interface IDialogService
2  {
3      void Display(string message);
4      void Display(Exception exception);
5      void Display(Exception exception, string additionalInfo);
6      bool Ask(string question);
7  }
8
9  public interface ICycleDetector
10 {
11     IReadOnlyCollection<IList<IIndexedVertex>> DetectCycles(
           IIndexedGraph graph);
12
13     IReadOnlyCollection<IList<IIndexedVertex>> DetectCycles<T>(
           IIndexedGraph graph) where T : class;
14 }
```

18 Klientská aplikace

18.1 View

Jmenný prostor *UmlDiagramDesigner.View* obsahuje dle definice architektonického vzoru MVVM z kapitoly 14 jednotlivé soubory s uživatelským rozhraním aplikace. Platforma WPF umožňuje seskupovat celky uživatelských rozhraní do třídy zvané **UserControl**, která poté umožní znovupoužití těchto celků na více místech v aplikaci. Takový celek tvoří i Toolbox a Canvas, které jsou v práci popsány níže.

18.1.1 Toolbox

Úkolem Toolboxu je zobrazit položky, které bude mít uživatel k dispozici pro tvorbu modelu. Implementace se skládá ze dvou objektů typu *ItemsControl* (jeden pro entity a druhý pro asociace), které mají pouze za úkol zobrazit kolekci objektů. Obě kolekce jsou umístěny v objektech *Expander*, což umožňuje jejich složení a rozložení dle potřeby. Typ *ItemsControl* má automatickou implementaci pro posouvání kolečkem myši s využitím *ScrollView*. Vizuální reprezentace konkrétních položek je nastavena stylem **ToolboxItemsControlStyle**, který se nachází v separátním souboru.

Datová kolekce objektů je nastavena do vlastnosti **ItemsSource** na kolekci *DiagramItems*. Poslední důležitou definicí u Toolboxu je spouštění Commandu při zachycení události *PreviewMouseLeftButtonDown*, pomocí které jde zajišťovat přetažení objektů z Toolboxu na plochu s modelem.

Algoritmus 2 Toolbox implementace v XAML

```
<Expander
     Style="{StaticResource ToolboxExpanderStyle}"
             Header="Entities"
             IsExpanded="{models:BindableSettings
             ToolboxAssociationsExpanded}">
    <ItemsControl ItemsSource="{Binding DiagramItems}"
                  Style="{StaticResource
                  ToolboxItemsControlStyle}"
                  Name="ToolboxItemsControl"
                  UseLayoutRounding="True"
                  SnapsToDevicePixels="True">
        <ItemsControl.ItemsPanel>
          <ItemsPanelTemplate>
           <WrapPanel ItemHeight="77" ItemWidth="90"/>
          </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
        <i:Interaction.Triggers>
            <i:EventTrigger EventName="
                  PreviewMouseLeftButtonDown">
                <i:InvokeCommandAction
                  Command="{Binding StartDragCommand}"
                  CommandParameter="{Binding
                  ElementName=ToolboxItemsControl}"/>
            </i:EventTrigger>
        </i:Interaction.Triggers>
    </ItemsControl>
</Expander>
<Expander
      Style="{StaticResource ToolboxExpanderStyle}"
          Header="Associations"
          IsExpanded="{models:BindableSettings
                  ToolboxEntitiesExpanded}">
    <ItemsControl ItemsSource="{Binding
                                DiagramRelationships}"
                  Style="{StaticResource
                        ToolboxItemsControlStyle}">
        <ItemsControl.ItemsPanel>
          <ItemsPanelTemplate>
           <WrapPanel ItemHeight="77" ItemWidth="90"/>
          </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
    <!--kód byl z důvodu jeho délky zkrácen-->
```

18.1.2 Canvas

Uživatel od UI CASE nástroje dnes vyžaduje mnoho specifik, které je třeba dodržet. Autor zhodnotil stav ovladatelnosti diagramu v programu v následující tabulce.

Vlastnost	Stav	Způsob implementace
Manipulace s prvky v podobě posunu po diagramu	Plná podpora	Výběrem objektu a tahem myší
Výběr více objektů	Plná podpora	Tahem myší na volné ploše
Hromadná manipulace s objekty	Částečná podpora	Pouze jejich mazání
Manipulace s hranami asociací, které propojují prvky modelu	Částečná podpora	Pouze po hranách propojených objektů
Seskupování objektů	Není implementováno	Není implementováno
Přítomnost klávesových zkratek	Částečná podpora	Pouze pro operace mazání objektů a atributů

Table 5: Stav interakce s modelem

Z tabulky 5 vyplývá, že ne všechny funkce jsou v programu přítomny a zcela plně podporovány. Přesto je autorův subjektivní názor, že je program snadno a efektivně ovladatelný i uživatelem bez hlubší znalosti CASE nástrojů a jiných technicky zaměřených programů. Všechny vlastnosti zmíněné v tabulce 5 jsou alespoň z části implementovány na prvku Canvas.

Canvas, v českém jazyce přeloženo jako plátno, je plocha, kde uživatel v programu vytváří model. K jeho implementaci byl použit, stejně jako u prvku Toolbox, objekt *ItemsControl*, který narozdíl od objektu *Canvas* ve WPF umožňuje pomocí data binding zobrazovat kolekci objektů. Jako zobrazovací panel prvku *ItemsControl* byl zvolen objekt *Canvas*, který poskytuje možnost pozicovat objekty v dvourozměrné soustavě souřadnic.

Algoritmus 3 Canvas implementace v XAML

```
<ScrollViewer VerticalScrollBarVisibility="Auto"
              HorizontalScrollBarVisibility="Auto">
    <Grid>
        <Border BorderBrush="Black"
            BorderThickness="1">
            <ItemsControl Name="DiagramItemsControl"
                    ItemsSource="{Binding Items}"
                    Background="White"
  ItemContainerStyleSelector="{x:Static styleSelectors
     :DiagramItemStyleSelector.Instance}"
  PreviewMouseLeftButtonUp="
     DiagramCanvasPreviewMouseLeftButtonUp"
  PreviewMouseMove="
     DiagramItemsControl_OnPreviewMouseMove"
  PreviewMouseDown="
     DiagramItemsControl_OnPreviewMouseDown"
                    SnapsToDevicePixels="True">

          <ItemsControl.ItemsPanel>
                <ItemsPanelTemplate>
                  <Canvas Width="5000" Height="5000"/>
                </ItemsPanelTemplate>
          </ItemsControl.ItemsPanel>

          <ItemsControl.InputBindings>
             <KeyBinding Key="Delete" Command="{Binding
                 DeleteSelectedItemsCommand}"/>
          </ItemsControl.InputBindings>
        </ItemsControl>
      </Border>
    </Grid>
</ScrollViewer>
```

Celý prvek je umístěn v objektu *ScrollViewer*, který zajišťuje možnost automatického zobrazení posuvných lišt v případě, že objekt přesáhne velikost svého předka. Taková vlastnost je využita prakticky vždy, protože je nastavena pevná výška i šířka plochy na 5000 DIP [13].

Detailní interakce s prvky na plátně vyžaduje obsluhu mnoha událostí, které byly z důvodu lepší čitelnosti a přehlednosti implementovány v code-behindu třídy *DiagramCanvas* a nikoli pomocí Command objektů, jak by tomu správně v MVVM vzoru mělo

[13] Device Independent Pixel je měrná jednotka užívaná v platformě WPF, která je vypočítávána z hustoty pixelů na palec. Umožňuje tak částečně zachovat velikost prvků na zařízeních s různým rozlišením. V současnosti se tato technika uplatňuje v mobilním vývoji, kde je široká různorodost zařízení.

být. Nejdůležitější nastavené vlastnosti jsou **ItemsSource**, která obsahuje kolekci zobrazených objektů, a **ItemContainerStyleSelector**, která určuje, jaký styl mají přidané objekty dostat. Nastaveným stylem je poté určen konečný vzhled objektu.

Algoritmus 4 Výběr příslušného stylu v třídě DiagramItemStyleSelector

```
1  public override Style SelectStyle(object item, DependencyObject
       container)
2      {
3          if (DiagramItemsResourceDictionary == null) return null
               ;
4
5          var diagramCanvas = ItemsControl.
               ItemsControlFromItemContainer(container);
6          if (diagramCanvas == null) return null;
7
8          var diagramItem = item as DiagramItemViewModelBase;
9
10         if(diagramItem != null &&
               DiagramItemsResourceDictionary.Contains(diagramItem.
               GetResourceStyleName()))
11           return (Style)DiagramItemsResourceDictionary[
                 diagramItem.GetResourceStyleName()];
12
13         var diagramRelationship = item as
               RelationshipViewModelBase;
14
15         if (diagramRelationship != null &&
               DiagramRelationshipsResourceDictionary.Contains(
               diagramRelationship.GetResourceStyleName()))
16           return (Style) DiagramRelationshipsResourceDictionary
                 [diagramRelationship.GetResourceStyleName()];
17
18         return null;
19     }
```

Každý prvek, který lze přiřadit na diagram musí dědit od abstraktní třídy *DiagramEntityViewModelBase*, která definuje jedinou abstraktní metodu *GetResourceStyleName*. Této vlastnosti je využito při výběru vhodného stylu, který je dostupný v algoritmu 4, kde je podle názvu stylu hledán příslušný prvek z konkrétníko slovníku.

18.2 View Model

V kapitole 14 je zmíněno, že obvykle pro jeden view existuje jeden view model. V práci se tato zvyklost mnohokrát nedodržuje zejména díky složitosti UI, které v programu je. Ve skutečnosti se každý objekt na modelu může skládat z mnoha dalších view modelů, příkladem může být například *EntityDiagramItemViewModel*, který obsahuje kolekci view modelů typu *RelationshipViewModelBase, AttributeViewModel* a *TriggerViewModel*. Celá plocha programu samozřejmě obsahuje kolekci entit a asociací, které tvoří výsledný model.

Náhled diagramu tříd všech aktuálně implementovaných typů objektů a asociací je zobrazen na obrázku 12. O majoritní díl práce se starají abstraktní třídy *DiagramItemViewModelBase*, od které dědí všechny prvky diagramu, a *RelationshipViewModelBase*, od které dědí všechny asociace. Cílem zmíněných tříd je abstrahovat maximální díl práce, který mají všichni jejich potomci stejný.

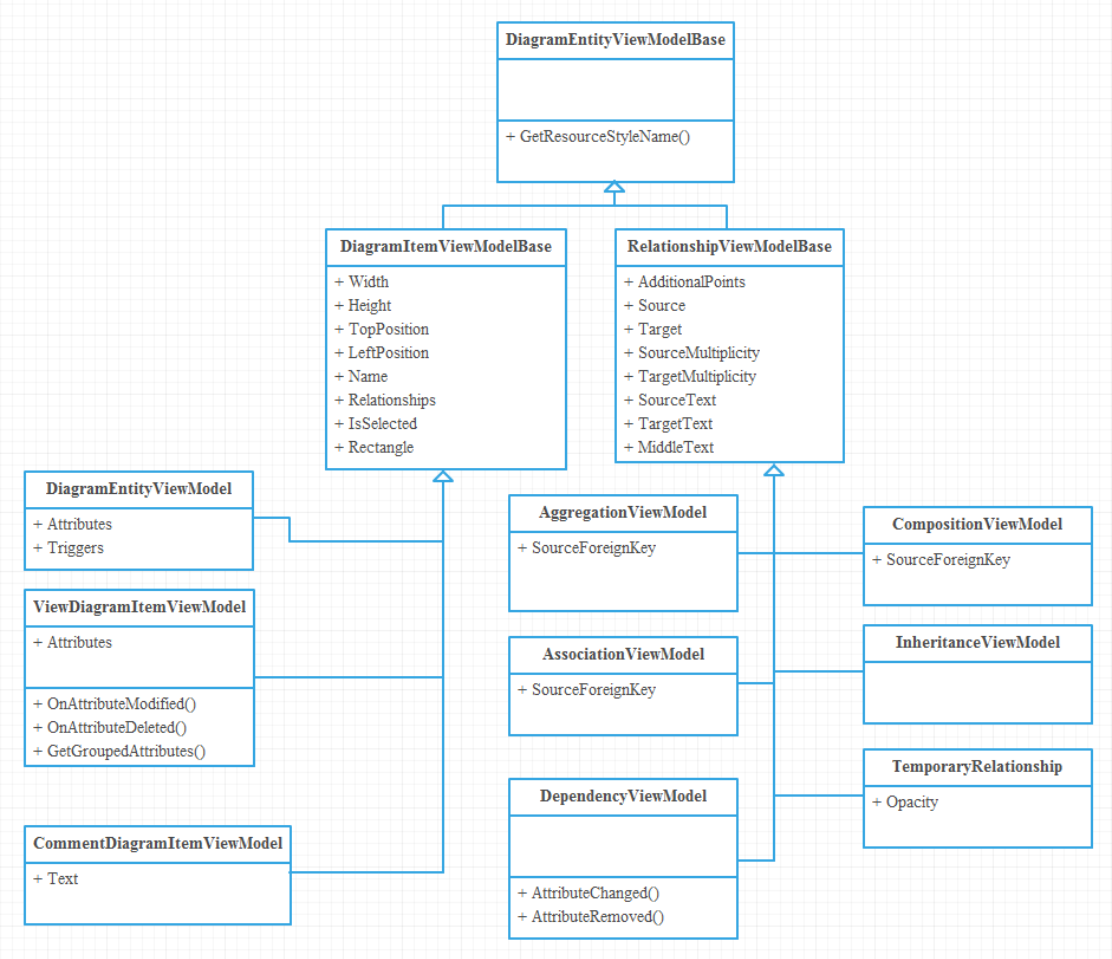

Figure 12: Zjednodušený diagram tříd objektů a asociací v programu (Zdroj: vlastní)

18.2.1 DiagramItemViewModelBase

Třída, od které musí dědit všechny objekty, které se vyskytují na modelu, je v práci nazvána *DiagramItemViewModelBase*. Skrze vlastnosti poskytuje informace o pozici objektu, zda je objekt vybrán, jeho rozměry a jiná data o vizuální podobě. Třída dále implementuje rozhraní:

ISelectable

- umožňuje definovat, zda lze objekt na diagramu označit (vybrat)

IIndexedVertex

- zajišťuje spolupráci s třídou *TarjansAlgorithm*

IPointProvider

- slouží pro získání vhodných bodů na objektu, na které mohou být napojeny asociace

Spouštěním události **ItemMoved** je zajištěno, že při posunu objektu dojde k překreslení všech hran, které jsou k objektu připojené, nebo z něj vychází. Třída také musí udržovat stále aktuální vlastnost **Rectangle**, která určuje obdélník ohraničující objekt. Na tento obdélník jsou následně napojovány všechny asociace, z čehož plyne vlastnost programu, že jsou aktuálně podporovány pouze obdélníkové objekty.

View model	Účel a vlastnosti
EntityDiagramItemViewModel	Reprezentace třídy z diagramu tříd UML. Obsahuje svůj název, atributy a triggery. Označen atributem *Associatable* s parametry všech asociací, vyjma závislosti.
ViewDiagramItemViewModel	Označuje databázový pohled. Obsahuje svůj název a kolekci atributů ze závislých tříd. Označen atributem *Associatable* s parametrem závislosti.
CommentDiagramItemViewModel	Komentář, který obsahuje pouze datovou složku s jeho textem a datem vytvoření. Neoznačen atributem *Associatable*, takže není možné ho s žádným jiným objektem asociovat.

Table 6: View modely objektů

18.2.2 RelationshipViewModelBase

Asociace propojující jednotlivé objekty mají svého předka ve třídě *RelationshipViewModelBase*, od kterého musí všechny dědit. Obdobně jako u *DiagramItemViewModelBase* je i zde mnoho vizuálních vlasností, zejména všechny souřadnice, na kterých asociace leží, textový popis, kardinality, objekty, které propojuje a další. Metoda *OpenEditWindow*, která otevře editační formulář, má implementaci prázdnou, je ovšem označena klíčovým slovem **virtual**, čímž je možné logiku doplnit do všech tříd, u nichž chceme umožnit určitou formu editace. Konkrétní implementace metody je tak naprogramována u tříd *AggregationViewModel, AssociationViewModel a CompositionViewModel*.

View model	Účel a vlastnosti
AggregationViewModel	UML agregace. Označena atributem *AllowMultiple* pro možnost aplikování více instancí na jeden objekt. Implementuje *IForeignKeyRelationship*, protože tvoří cizí klíč.
AssociationViewModel	UML asociace. Označena atributem *AllowMultiple* pro možnost aplikování více instancí na jeden objekt. Implementuje *IForeignKeyRelationship*, protože tvoří cizí klíč.
CompositionViewModel	UML kompozice. Označena atributem *AllowMultiple* pro možnost aplikování více instancí na jeden objekt. Implementuje *IForeignKeyRelationship*, protože tvoří cizí klíč.
DependencyViewModel	UML závislost. Označena atributy *AllowMultiple* a *RequiresConstraint* s parametry primárního klíče a unique.
InheritanceViewModel	UML dědičnost. Označena atributy *MultiplicityNotSupported*, čímž je zamezeno aplikaci kardinalit a *ValidateDuplicateAttributes*, pro aplikaci validace na duplicitní atributy v hierarchii dědičnosti. Vlastnost **IsConnectable** nastavena na hodnotu **true**, aby docházelo ke skládání.
TemporaryRelationship	Konkrétní dekorátor pro všechny asociace. Upravuje vizuální podobu asociace, která je v procesu vytváření a není ještě rozhodnuto, zda bude vytvořena.

Table 7: View modely asociací

18.2.3 CanvasViewModel

Diagram modelu, který uživatel programu vytváří obsluhuje view model *CanvasViewModel*. Jsou v něm uloženy všechny objekty a asociace diagram v jedné kolekci **Items** v podobě abstraktního typu *DiagramEntityViewModelBase*. K výběru objektů z této kolekce je použita LINQ [14] metoda *OfType<T>*, která umožní filtrovat kolekci pouze na učitý typ, například *RelationshipViewModelBase*, čímž dojde k výběru pouze asociací.

CanvasViewModel deleguje vytváření objektů a asociací na příslušné objekty, předává asociace k validaci a zobrazuje zprávy uživateli při výskytu chyb a výjimek. Při výběru objektu na modelu přesouvá daný objekt do popředí pomocí jeho vlastnosti **ZIndex** a registruje se k příjmu zpráv o smazání objektů z diagramu. Canvas je nutné notifikovat, pokud dojde ke smazání některé z entit či asociací z modelu. K tomu slouží třída *Messenger*, do níž se *CanvasViewModel* registruje ve vloženém algoritmu 5.

[14]Language Integrated Query je jazyk dostupný v C# od verze 3.0, který slouží k dotazování nad kolekcemi dat. Syntaxi lze využít způsobem podobným jazyku SQL, nebo pomocí lamba výrazů.

Algoritmus 5 Registrace do Messengeru v CanvasViewModel

```
Messenger.Register<DeletedEntityDTO>(this, toDelete =>
    DeleteEntity(toDelete.EntityToDelete));

Messenger.Register<DeletedRelationshipDTO>(this, toDelete =>
    DeleteRelationship(toDelete.Relationship));
```

Implementace rozhraní **IIndexedGraph** umožňuje aplikaci Tarjanova algoritmu na diagram. Realizace výběru vrcholů grafu je v algoritmu 6.

Algoritmus 6 Implementace IIndexedGraph u CanvasViewModel

```
public IEnumerable<IIndexedVertex> GetVertices()
{
    return Items.OfType<DiagramItemViewModelBase>();
}

IEnumerable<IIndexedVertex> IIndexedGraph.GetVertices()
{
    return Items.OfType<DiagramItemViewModelBase>();
}
```

18.3 Model

V následujících kapitolách bylo do práce zahrnuto několik tříd, které implementují návrhový vzor, nebo byly jiným způsobem ztěžejní pro správný návrh a fungování aplikace.

18.3.1 Messenger

Třída Messenger je modifikovanou implementací návrhového vzoru Mediator. Inspiraci poskytla implementace stejnojmenné třídy v knihovně **Mvvm Light Toolkit**, která však nebyla do projektu zahrnuta. Podoba třídy je zjednodušena natolik, aby poskytovala opravdu jen jeden účel - zprostředkovat komunikaci pomocí zasílání zpráv na sobě nezávislým objektům.

Algoritmus 7 Ukázka komunikace s využítím třídy Messenger

```
internal class Sender
{
    public void SendMessage()
    {
        Messenger.Send("Data zpravy");
    }
}

internal class Reciever
{
    public void RegisterMessage()
    {
        Messenger.Register<string>(this, message =>
        {
            //zde lze vykonat libovolný kód s přijatou zprávou
               typu string
        });
    }
}
```

Z ukázkového kódu v algoritmu 7 je důležité zejména to, že třída *Sender* nemá žádnou informaci o existenci třídy *Receiver*. Pokud třída *Reciever* neexistuje a neexistuje ani žádná jiná třída, která by se zaregistrovala k přijímání určité zprávy, nebude po volání metody Send spuštěna žádná akce na straně příjemce, protože žádný není. K registraci je třeba uvést datový typ zprávy, ke které se příjemce registruje a také instanci, podle které je identifikován při odregistrování. Třída *Messenger* je implementována jako návrhový vzor Singleton [15] a její kód je následující:

[15] Více o návrhovém vzoru Singleton v [Erich Gamma, 1995] a [Freeman – Freeman, 2004].

Algoritmus 8 Implementace třídy Messenger

```
1  public static class Messenger
2  {
3      private static readonly Dictionary<Type, IList<
           ReceiverAction>> RegisteredRecipients;
4
5      static Messenger()
6      {
7          RegisteredRecipients = new Dictionary<Type, IList<
               ReceiverAction>>();
8      }
9
10     public static void Register<T>(object receiver, Action<T>
           action)
11     {
12         IList<ReceiverAction> value;
13         if (RegisteredRecipients.TryGetValue(typeof(T), out
               value))
14         {
15             value.Add(new ReceiverAction(receiver, o => action
                   ((T)o)));
16         }
17         else
18         {
19             var recipients = new List<ReceiverAction>() { new
                   ReceiverAction(receiver, o => action((T)o)) };
20             RegisteredRecipients.Add(typeof(T), recipients);
21         }
22     }
23
24     public static void Unregister<T>(object receiver, Action<T>
           action)
25     {
26         IList<ReceiverAction> value;
27         if (RegisteredRecipients.TryGetValue(typeof(T), out
               value))
28         {
29             for (int i = 0; i < value.Count; i++)
30             {
31                 value.RemoveAt(i);
32             }
33         }
34     }
35
36     public static void Send<T>(T message)
37     {
38         IList<ReceiverAction> value;
39         if (RegisteredRecipients.TryGetValue(typeof(T), out
               value))
40         {
41             foreach (var receiverAction in value)
42             {
43                 receiverAction.Action.Invoke(message);
44             }
45         }
46     }
47 }
```

18.3.2 GenericCommand

Rozhraní *ICommand* je obsaženo v .NET Frameworku a slouží k definici tříd, které zapouzdřují určité chování. Platforma WPF umožňuje přímo navazovat konkrétní command objekty na události vyvolané z uživatelského rozhraní. Přímou komunikací se programátor může ve velké míře vyhnout použití code-behind části kódu, která svazuje chování aplikace s konkrétními prvky uživatelského rozhraní (lze řešit pomocí architektury Model-View-Presenter, srovnání v kapitole 14.4). Zapouzdření logiky do command objektů umožňuje jejich snadnou testovatelnost bez přítomnosti UI.

V projektu implementují rozhraní *ICommand* dvě třídy, jedna bez parametru a druhá s generickým parametrem. Pro jejich použití není nutné vytváření konkrétních command tříd tak, jak to uvádí diagram návrhového vzoru Command na obrázku 7, stačí instanci předat při spouštění metody Execute delegát metody, která má být provedena. Takový postup je v projektu kombinován s vlastností jazyka C# - anonymních metod.

Algoritmus 9 Implementace rozhraní ICommand

```
1  public class GenericCommand<T> : ICommand
2  {
3      private readonly Action<T> _action;
4      private readonly bool _canExecute;
5
6      public GenericCommand(Action<T> action,
7                            bool canExecute = true)
8      {
9          _action = action;
10         _canExecute = canExecute;
11     }
12
13     public bool CanExecute(object parameter)
14     {
15         //zde lze implementovat libovolnou logiku
16         //pro vyhodnocení, zda se má Command spustit
17         return _canExecute;
18     }
19
20     public void Execute(object parameter)
21     {
22         _action((T) parameter);
23     }
24
25     private void RaiseCanExecuteChangedEvent()
26     {
27         if (CanExecuteChanged != null)
28             CanExecuteChanged(this, new EventArgs());
29     }
30
31     public event EventHandler CanExecuteChanged;
32 }
```

18.3.3 Depth First Search

Prohledávání do hloubky je jedním ze způsobů, jak lze algoritmicky projít graf. Implementace Depth First Search (dále jen DFS) byla nutná z důvodu generování kódu a validace modelu, což je popsáno v následujících kapitolách. Průchod grafu pomocí DFS je možné aplikovat na kterýkoli z jeho vrcholů, kdy se počáteční vrchol označí jako otevřený a rekurzivně dochází k volání DFS průchodu u všech jeho sousedních vrcholů. Když dojde k návratu do již prohledaného vrcholu, je uzavřen a návratová hodnota algoritmu je postupný sled vrcholů, které algoritmus zpracoval. Rekurzivním voláním tak dojde k průchodu všech větví grafu do libovolné hloubky. [Mička, 2010a]

Samotná implementace DFS byla zahrnuta do programu v pozdní fázi vývoje a užito tak bylo rozhraní, které implementovaly všechny objekty, které se na diagramu mohou vyskytovat. Pro vhodnější zavedení Tarjanova algoritmu, který je popsán v následující kapitole 18.3.4, bylo rozhraní rozděleno na abstraktnější a konkrétnější formu. Abstraktní forma byla nazvána *IVertex* a konkrétní, kterou využívá Tarjanův algoritmus, je nazvána *IIndexedVertex* a dědí od předchozího rozhraní.

Algoritmus 10 Rozhraní IVertex

```
1  public interface IVertex
2  {
3      IEnumerable<IVertex> GetNeighbours(bool revert = false);
4      IEnumerable<IVertex> GetNeighbours<T>
5        (bool revert = false) where T : class;
6  }
7
8  public interface IIndexedVertex : IVertex
9  {
10     int Index { get; set; }
11     int LowLink { get; set; }
12     IEnumerable<IIndexedVertex> GetIndexedNeighbours(bool
           revert = false);
13     IEnumerable<IIndexedVertex> GetIndexedNeighbours<T>
14         (bool revert = false) where T : class;
15 }
```

Princip prohledávání

Počáteční vrchol se v algoritmu vloží na první místo zásobníku. Následuje průchod všech prvků v zásobníku, kdy se vrchní vrchol ze zásobníku vždy odebere a vloží do kolekce již navštívených vrcholů. Pokud ještě prvek nebyl navštíven je vrácen konstrukcí *yield return*, která umožňuje výslednou návratovou hodnotu za běhu algoritmu dále modifikovat, například uplatněním filtrace nebo řazení. Všechny sousední prvky jsou také vloženy do zásobníku a dojde k jejich postupnému průchodu v následujících iteracích cyklu.

K zachycení všech již navštívených vrcholů je použita kolekce *HashSet<IVertex>*, která neumožňuje přidávat duplicitní položky. Návratovou hodnotou metody *Add(IVertex)* je boolean hodnota, která určuje, zda byl prvek do kolekce přidán, či nikoli.

Pro implementaci DFS algoritmu byla zvolena extension metoda [16] s názvem *DepthFirstTraversal* a její generická verze *DepthFirstTraversal<T>*.

Algoritmus 11 Extension metoda DepthFirstTraversal<T>

```
1  public static IEnumerable<IVertex> DepthFirstTraversal<T>(
2      this IVertex start, bool revert = false) where T : class
3  {
4      var visited = new HashSet<IVertex>();
5      var stack = new Stack<IVertex>();
6
7      stack.Push(start);
8
9      while (stack.Count != 0)
10     {
11         var current = stack.Pop();
12
13         if (!visited.Add(current)) continue;
14
15         yield return current;
16
17         var neighbours = current.GetNeighbours<T>(revert)
                                   .Where(n => !visited.Contains(n));
18
19         foreach (var neighbour in neighbours)
20             stack.Push(neighbour);
21     }
22 }
```

18.3.4 Tarjanův algoritmus

Problematika modelování s sebou přináší nutnost validace, protože pokud by model validován nebyl, nebylo by možné z něj úspěšně generovat kód. Vlastnost dědičnosti přináší mnohé výhody, zejména v oblasti znovupoužitelnosti a polymorfismu. Při neuváženém použití však může způsobit nefunkčnost celého systému, jedním z takových případů by bylo užití tzv. cyklické dědičnosti [17]. Cyklická dědičnost nastane v případě, že potomek třídy bude dědit od kteréhokoli z jejích nadtypů, včetně třídy samotné. Takový jev se vyskytuje v teorii grafů u orientovaného grafu pod názvem *cyklus*, nebo *kružnice*.

Cyklus

Cyklus v orientovaném grafu lze definovat jako cestu v grafu, jejíž začátek i konec je shodný. Každý uzel v cestě může incidovat maximálně se dvěma hranami a všechny hrany v cestě musí být jedinečné.

[16] Extension metoda je vlastnost jazyka C# od jeho 3. verze. Umožňuje doplnit metodu do tříd, či rozhraní .NET frameworku, ke kterým nemáme přímo zdrojové kódy. Lze užít i pro implementaci metod pro rozhraní.

[17] Byť například jazyk C++ s cyklickou dědičností pracovat umí.

Silná komponenta Silná komponenta se v orientovaném grafu vyskytuje tehdy, pokud se z každého vrcholu silné komponenty jde dostat do jakéhokoli jiného vrcholu dané komponenty. Takovou definici splňuje i výše popsaný cyklus.

Jelikož je možné chápat UML diagram tříd jako orientovaný graf, jehož vrcholy jsou třídy a hrany jednotlivé asociace, byl pro detekci cyklické dědičnosti použit Tarjanů algoritmus. V počátku vývoje autor práce chování potřebné pro detekci cyklu neimplementoval, bylo potřeba ho zavést už do rozpracovaného řešení. Dosaženo toho bylo zavedením DFS, jehož implementace je dostupná v 18.3.3. Samotný algoritmus, jehož kompletní chování je popsáno v [Mička, 2010b], je implementován následujícím způsobem.

Algoritmus 12 Tarjanův algoritmus

```
1  private void StrongConnect(IIndexedVertex vertex, IList<IList<
       IIndexedVertex>> stronglyConnected)
2  {
3      vertex.Index = _index;
4      vertex.LowLink = _index;
5      _index++;
6      _stack.Push(vertex);
7
8      foreach (var neighbour in vertex.GetIndexedNeighbours())
9      {
10         if (neighbour.Index < 0){
11             StrongConnect(neighbour, stronglyConnected);
12             vertex.LowLink = Math.Min(vertex.LowLink, neighbour
                   .LowLink);
13         }
14         else if (_stack.Contains(neighbour)){
15             vertex.LowLink = Math.Min(vertex.LowLink, neighbour
                   .Index);
16         }
17     }
18
19     if (vertex.LowLink == vertex.Index){
20         var stronglyConnectedComponent = new List<
               IIndexedVertex>();
21         IIndexedVertex tempVertex;
22         do{
23             tempVertex = _stack.Pop();
24             stronglyConnectedComponent.Add(tempVertex);
25         }
26         while (tempVertex != null && !Equals(vertex, tempVertex
               ));
27         stronglyConnected.Add(stronglyConnectedComponent);
28     }
29 }
30
31 public IReadOnlyCollection<IList<IIndexedVertex>> DetectCycles(
       IIndexedGraph graph)
32 {
33     IList<IList<IIndexedVertex>> stronglyConnected = new List<
           IList<IIndexedVertex>>();
34
35     _stack = new Stack<IIndexedVertex>();
36     _index = 0;
37     ResetVertexIndexes(graph);
38
39     foreach (var vertex in graph.GetVertices()){
40         if (vertex.Index < 0){
41             StrongConnect(vertex, stronglyConnected);
42         }
43     }
44
45     return new ReadOnlyCollection<IList<IIndexedVertex>>(
           OmitIndependentComponents(stronglyConnected));
46 }
```

Kód celé třídy je příliš rozsáhlý, takže jsou zde uvedeny pouze dvě nejdůležitější metody. Metoda *DetectCycles<T>* projde v cyklu všechny vrcholy grafu na pomocí metody *StrongConnect<T>* najde jejich silnou komponentu. Po nalezení silných komponent se z důvodů snazší interpretace výsledků odstraní ty komponenty, které obsahují pouze jeden vrchol. Takové vrcholy nejsou cyklem a pro výsledek jsou irelevantní.

V metodě *StrongConnect<T>* dojde k přidání vrcholu na zásobník, průchodu všech jeho sousedících vrcholů a zjištění, zda už nějaký z nich byl „zkoumán" metodou *StrongConnect<T>*. V případě, že se vlastnost LowLink rovná vlastnosti Index, dojde k uzavření silné komponenty a přidání všech jejích vrcholů do výsledné kolekce. Implementace využívá rekurze, kdy se opětovně volá metoda s parametry aktuálně prohledávaného vrcholu a silné komponenty s aktuálním výčtem jejích vrcholů.

Generický parametr zde označuje typ hrany pro který chceme silné komponenty najít. UML diagram tříd je po transformaci do orientovaného grafu multigrafem, protože může obsahovat více typů hran (asociací). Třída *TarjansAlgorithm* obsahuje i verzi bez generického parametru, která umožňuje najít cykly v grafu bez ohledu na to, jakým typem hrany jsou vrcholy spojeny.

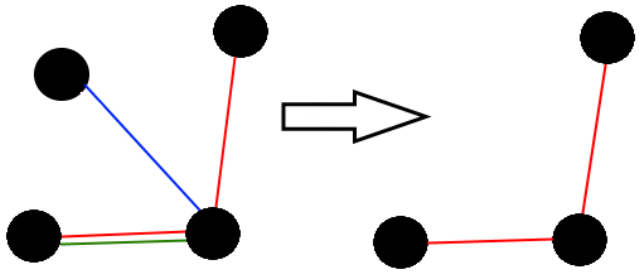

Figure 13: Transformace multigrafu (Zdroj: vlastní)

Transformace multigrafu, která nastane při použití generické verze Tarjanova algoritmu je zobrazena na obrázku 13. Pro ilustrativní účely jsou hrany rozlišeny pouze barevně a generickým parametrem v případě této transformace je červená hrana. Je zřejmé, že nedošlo pouze k odstranění ostatních hran, ale také vrcholu, který nyní nesousedí s žádným vrcholem genericky zvolenou hranou. V programu je takový parametr využit právě pro vymezení vztahů typu dědičnost, pro které je třeba cykly najít.

19 Generování kódu

Stěžejní funkcí programu je možnost generování kódu. Generování je možné provádět z libovolného modelu, protože jeho validita je zaručena již při jeho vytváření. Majoritní část dnešního softwaru, zejména v podnikové sféře, vyžaduje provázanost s úložištěm dat. Na tuto skutečnost byl brán zřetel při výběru jazyků, do kterých je model generován. Při volbě jazyků byla zohledněna jejich popularita a vlastnosti, zejména z pohledu objektově orientovaného přístupu. Problematika volby SŘBD mezi objektovými a relačními databázemi v této práci není zpracována a bylo tak upřednostněno široké rozšíření relačně databázových technologií.

V aktuální verzi software umožňuje generovat kód do následujících jazyků:

- C#
- Java
- Smalltalk
- SQL

19.1 Objektově orientované jazyky

Pro generování kódu byl zaveden do programu návrhový vzor Strategie, který umožňuje abstrahovat implementaci algoritmu a dynamicky tak měnit jeho konkrétní podobu. První abstrakcí nad všemi jazyky je abstraktní třída *GeneratorBase*, která obsahuje pouze jednu veřejnou metodu *GenerateCode* s návratovou hodnotou vygenerovaného kódu. Třída vnitřně hrubě definuje strukturu vygenerovaného kódu, který rozděluje na pouze na úvodní část, obsah kódu a závěrečnou část. Úvodní a závěrečná část obsahují pouze zakomentované části s doplňujícími informacemi o vygenerovaném kódu.

Druhou abstrakcí nad konkrétními jazyky je třída *ObjectOrientLanguageGenerator*, která je také označena klíčovým slovem **abstract** [18] a definuje podrobněji strukturu vygenerovaného kódu pro objektové jazyky. Důležitým faktem je označení všech metod, které generují kód, klíčovým slovem **virtual**, což umožňuje v případě potřeby změnit chování metody v konkrétních třídách. V algoritmu 13 lze vidět implementaci generování jmenných prostorů pro objektově orientované jazyky. V případě, že je třeba generovat kód do jazyka, kde jmenné prostory importovány být nemusí (takovým jazykem je například Smalltalk), stačí v konkrétním generátoru metodu označit klíčovým slovem **override** a nechat její implementaci prázdnou.

[18] Od tříd označených klíčovým slovem abstract nelze vytvářet instance.

Algoritmus 13 Generování jmenných prostorů

```
protected virtual string GenerateNamespaceImports()
{
    var sb = new StringBuilder();

    string keyword;

    if (LanguageProperties.Features.TryGetValue(LanguageFeature
        .NamespaceImport, out keyword)){
         string endLine;
         if (LanguageProperties.Features.TryGetValue(
             LanguageFeature.EndLineCharacter, out endLine)){
           foreach (var ns in RequiredNamespaces)
           {
                sb.AppendLine(string.Format("{0} {1}{2}",keyword,
                    ns, endLine));
           }
      }
    }
    return sb.ToString();
}
```

Vhodnou demonstrací při transformaci modelu do objektového kódu je zohlednění UML kompozic, které značí, že cílová objekt nemůže existovat bez zdrojového objektu. Takový vztah byl do vygenerovaného kódu zahrnut v podobě parametrů konstruktoru, kam jsou umístěny všechny objekty, na kterých zdrojový objekt závisí.

Algoritmus 14 Generování parametrů konstruktoru

```
1  private string GenerateConstructor(Class @class)
2  {
3      string publicAccessibility;
4      LanguageProperties.Accessibilities.TryGetValue(
           Accessibility.Public, out publicAccessibility);
5
6      if (@class.Compositions.Count == 0)
7      {
8          //vygenerovat prázdný konstruktor
9          return string.Format("{0} {1}() {2}{3}",
10       publicAccessibility ?? string.Empty,
11       @class.Name,
12             GenerateStartBlock(),
13       GenerateEndBlock());
14     }
15
16     var sb = new StringBuilder();
17     sb.AppendLine(string.Format("{0} {1}(",
18     publicAccessibility ?? string.Empty,
19     @class.Name));
20
21     foreach (var composition in @class.Compositions)
22     {
23         //přidat povinné parametry do konstruktoru
24         sb.Append(string.Format("{0} {1},", LanguageProperties.
               GenerateNavPropertyDataType(composition),
25             DetermineNavPropertyName(composition).ToCamelCase()
26         ));
27     }
28
29     sb.Remove(sb.Length - 1,1);//smazat poslední ','
30     sb.Append(string.Format(") \n {0}", GenerateStartBlock()));
31     sb.Append(GenerateConstructorInitializations(@class.
           Compositions).InsertTabsAtStartOfEachLine());
32     sb.AppendLine(GenerateEndBlock());
33
34     return sb.ToString();
35 }
```

19.2 SQL

Přístupů, jak transformovat datový model do relačně databázového modelu, potažmo skriptu, je mnoho. V této práci byla zvolena metodika popsána Scottem Amblerem v [Ambler, 2003b]. Konverze atributů tříd z modelu je realizována pomocí atributu (sloupce) v tabulce a příslušného datového typu, který je dynamicky zvolen během konverze. Pro název tabulky je použit název třídy z modelu a název asociativní tabulky, které jsou nutné pro realizaci **many to many** kardinality, je tvořen sloučením dvou jmen asociovaných tabulek.

Názvy všech atributů a tabulek jsou ošetřeny proti použití klíčových slov registrovaných SŘBD (aktuálně je podporován pouze SQL Server od společnosti Microsoft).

Realizace kardinalit vztahů asociace, kompozice, nebo agregace jsou následující:

1:1 říká, že zdrojový objekt obsahuje vždy pouze jeden objekt cílového typu. V transformaci je zohledněn jako cizí klíč odkazující na primární klíč cílové tabulky.

1:M charakterizuje možný několikanásobný výskyt cílového objektu ve zdrojovém. V transformaci je generován jako cizí klíč s referencí na primární klíč cílové tabulky.

M:N je nejproblematičtější kardinalitou k realizování v relačně databázových systémech. V relačně databázových systémech vyžaduje existenci asociativní tabulky, která obsahuje dva sloupce, které jsou cizími klíči pro obě asociované tabulky. Zároveň oba atributy tvoří složený primární klíč.

Dědičnost

Relačně databázové systémy neobsahují podporu dědičnosti, což znamená nutnost její realizace jiným způsobem. Strategie, které lze využít pro efektivní zpracování, se mohou v jednotlivých projektech lišit a není možné striktně vymezit jedinou správnou. Detailní popis doporučovaných postupů pro transformaci datového modelu, který obsahuje dědičnosti, na model relačně databázový je podrobně popsán v [Ambler, 2003b]. Autor zde zmíní pouze výhody a nevýhody jednotlivých postupů v tabulce 8.

Strategie	Výhody	Nevýhody
Tabulka pro každou hierarchii	Nejjednodušší. Podpora polymorfismu. Rychlý přístup k datům. Snadné generování reportů.	Nevhodné pro velké hierarchie. Komplikované určování konkrétních typů. Pevné svázání všech tříd. Změna v jedné třídě může ovlivnit ostatní třídy.
Tabulka pro každou konkrétní třídu	Snadné generování reportů (pro konkrétní třídu z jedné tabulky). Rychlý přístup k datům jedné třídy.	Při modifikaci tabulky nadtypu je nutné modifikovat tabulky všech podtypů. Nevhodné pro struktury, kde objekt může zastávat více typů v hierarchii (ztráta integrity)
Tabulka pro každou třídu	Snadná realizace (jedna třída je rovna jedné tabulce). Velmi dobrá podpora polymorfismu. Snadná modifikace tříd (pouze změna konkrétní tabulky). Data rostou pouze s nárustem objektů.	Databáze obsahuje mnoho tabulek. Potenciálně pomalejší přístup k datům (propojení mnoha tabulek). Obtížné generování reportů (lze řešit přidáním pohledů).

Table 8: Mapování dědičnosti do relačně databázových systémů

Autor v práci zvolil a implementoval třetí zmíněný postup z tabulky 8, který je dostatečně flexibilní ke změnám, což je preferovaný postup při užití agilních metodik. Z ilustračních důvodů autor přiložil ukázkovou transformaci datového modelu a diagramu z vytvořené databáze.

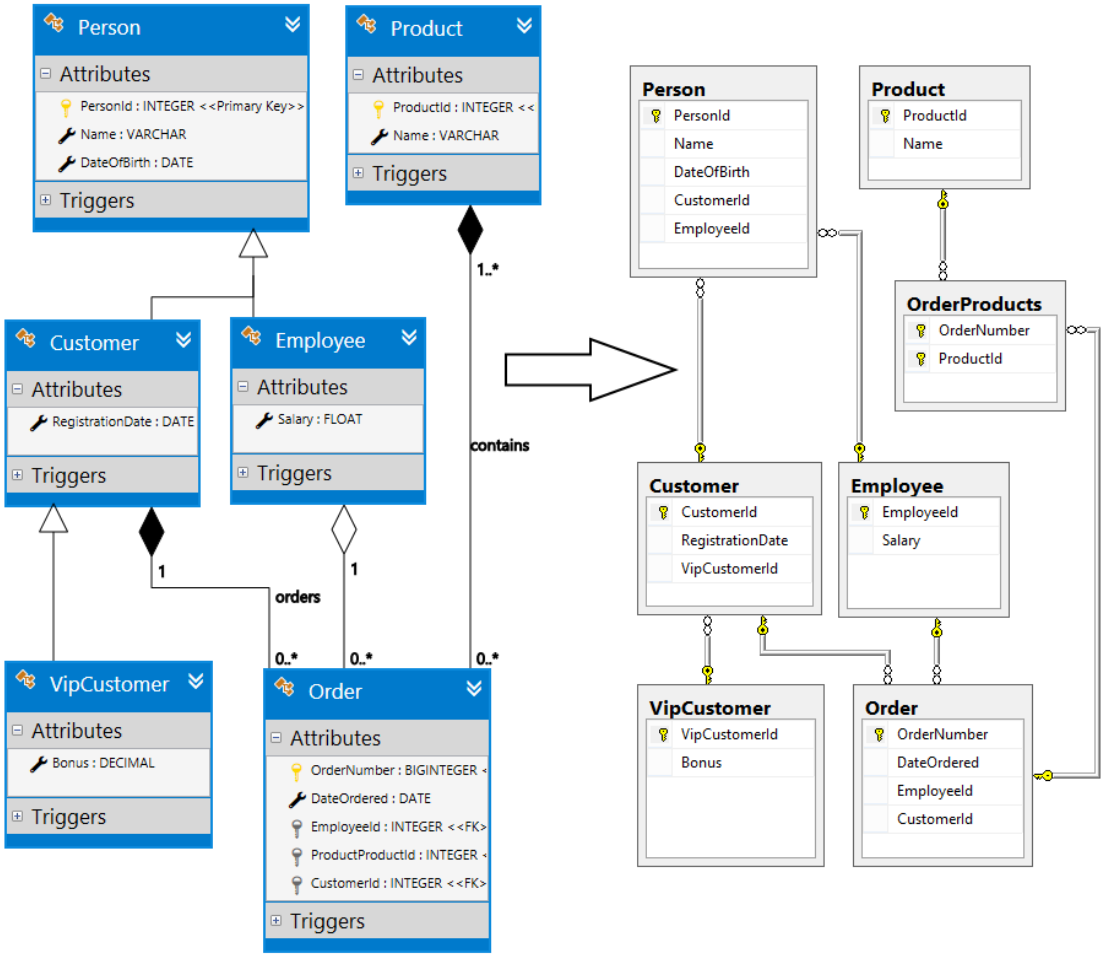

Figure 14: Transformace modelu (Zdroj: vlastní)

Z obrázku 14 je zřejmé, že došlo k přidání tabulky *OrderProducts*, která slouží jako asociační tabulka mezi tabulkami *Order* a *Product*. Strategie pro mapování dědičnosti je vidět na tabulkách *Employee* a *Customer*, kde jsou obsaženy pouze konkrétní atributy. Pokud je například osoba zákazníkem, přidá se záznam do tabulky *Customer* s datem registrace a do tabulky *Person* ostatní data, spolu s cizím klíčem, který bude odkazovat na předchozí záznam z tabulky *Customer*. Pokud by se zákazník později stal zároveň zaměstnancem firmy, stačilo by přidat záznam do tabulky *Employee* a aktualizovat záznam z tabulky *Person* o cizí klíč na nově přidaný záznam do tabulky *Employee*.

20 Testování

Testování aplikací je v dnešní době nedílnout součástí jejich vývoje, tuto práci nevyjímaje. Testování probíhalo v průběhu vývoje dle filozofie agilních přístupů, umístěno bylo ovšem až na konec práce, aby čtenář měl kvalitní přehled o tom, jak program funguje a proč byly vybrané komponenty testovány. V praxi se testy dělí na několik skupin a jsou vývojové metodiky, které testům přímo podmiňují samotný vývoj [19].

[19]Takovou metodikou je například Test Driven Development, kde testy předchází samotný kód.

Důležitým konceptem při testování je tzv. **Cost of Change**, který udává relativní cenu, kterou bude stát odhalení chyby v určité fázi vývoje. Zejména při použití vodopádových metodik je pozdní odhalení chyby a její následná oprava velmi nákladným a časově náročným procesem. Z obrázku 15 vyplývá, že agilní metodiky reagují na změny lépe a jejich „cena za změnu" je v pozdních fázích vývoje výrazně nižší.

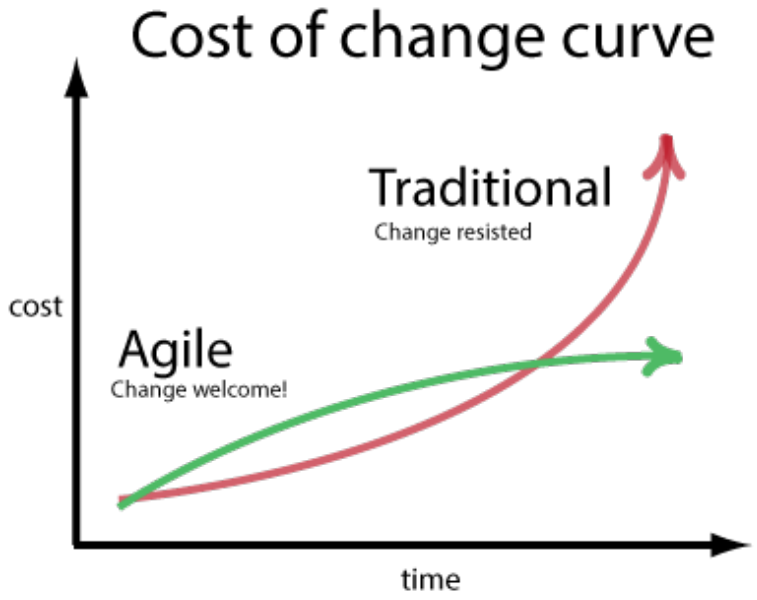

Figure 15: Vývoj cost of change u rozdílných metodik (Zdroj: www.agilenutshell.com)

20.1 Full Lifecycle Object-Oriented Testing (FLOOT)

> „*Full Lifecycle Object-Oriented Testing metodika je kolekcí testovacích a validačních technik pro ověřování a validování objektově orientovaného softwaru.*" - Scott W. Ambler

Výše uvedenou definici FLOOT autor zmiňuje v knize Object Primer [Ambler, 2004]. Soubor FLOOT čítá přes 30 technik, kterými je možné software v průběhu vývoje testovat. Aplikace testů neprobíhá pouze na zdrojový kód, ale i na uživatelské rozhraní, doménový model a jiné aspekty. V práci se však autor zaměřil pouze na testování kritické části zdrojového kódu aplikace. Mnohé z FLOOT technik, jako například **code inspection** a **boundary value**, jsou navíc automatizovány a dostupné přímo ve vývojových prostředích.

20.2 Unit test

Pro testování algoritmů, vlastních kolekcí a jiných samostatně izolovatelných jednotek slouží unit testy. V případě, že má kód nějaké závislosti (například na použité databázi), je nutné je nejprve izolovat. V .NET frameworku lze pro unit testy využít mnoho dostupných frameworků, mezi nejpoužívanější autor zařadil frameworky **xUnit**, **NUnit** a **MS Test**.

V této práci byl zvolen framework MS Test od společnosti Microsoft. Jeho funkce jsou plně dostačující a nepřináší do práce závislost na knihovnách třetích stran. Vlastnosti unit testů jsou následující:

1. Nezávislé na sobě
2. Nezávislé na pořadí, ve kterém jsou spouštěny (samostatné jednotky)
3. Paralelizovatelné (opět díky své nezávislosti)
4. Izolace od případných závislostí (řešením je zavedení Mockovacích tříd)

Mockování

Mock objekty slouží v objektově orientovaném programování k ostínění závislostí. Mock objekt je třída, která má totožný protokol, jako reálná třída, ale její implementace je zcela odlišná.

Příkladem pro použití mockovací třídy je situace, kdy bychom chtěli testovat kód, který závisí na existenci databáze a určitých dat. V produkčním prostředí není možné provádět testy na reálných datech, ovšem při zavedení mock objektu můžeme odstínit reálnou databázi a testy provést na jiné databázi určené pro účely testování. Díky vlastnosti stejného protokolu mock objektu, jako má reálná třída, chování kódu zůstane beze změny.

Vytváření mock objektů je možné dělat ručním napsáním příslušných tříd, nebo využít některý z dostupných frameworků. Mezi nejpoužívanější mockovací frameworky v době psaní práce autor zařadil **Moq**, **Rhino Mocks**, **Microsoft Fakes**.

Testy psané v této práci nebyly natolik rozsáhlé, proto autor zvolil ruční psaní mockovacích tříd. Všechny autorem napsané testy lze nalézt v projektu **UmlDiagramDesigner.Test**.

20.3 Tarjanův algoritmus

Tarjanův algoritmus lze vhodně testovat pomocí unit testů, protože se jedná o izolovaný algoritmus, jehož jediné závislosti jsou objekty tvořící diagram. Pro takové objekty byly ve jmenném prostoru *UmlDiagramDesigner.Test.Mocks* vytvořeny mockovací třídy. Pro vytvoření testu je nutné třídu označit atributem **TestClass** a všechny metody, které mají být interpretovány jako testovací, označit atributem **TestMethod**.

Konvence pojmenování pro metody v testu Tarjanova algoritmu byla zvolena následující: *PočetVrcholů_PočetHran_OčekávanýPočetCyklů*. Ze spuštěného testu je poté na první pohled zřejmé, jaký graf je vytvořen a jaký by měl být výsledek. Doplňující popis je dostupný v XML komentáři u každého testu.

Algoritmus 15 Část testu Tarjanova algoritmu

```
[TestMethod]
public void TwoVertices_TwoDifferentConnections_SingleCycle()
{
    var vertexA = new IndexedVertexMock();
    var vertexB = new IndexedVertexMock();

    vertexA.Neighbours.Add(new Tuple<IIndexedVertex,
        RelationshipViewModelBase>(vertexB, new
        InheritanceRelationshipMock()));
    vertexB.Neighbours.Add(new Tuple<IIndexedVertex,
        RelationshipViewModelBase>(vertexA, new
        CompositionRelationshipMock()));

    IIndexedGraph graph = new IndexedGraphMock { Vertices = new
         List<IIndexedVertex> { vertexA, vertexB }};

    var cycles = _tarjan.DetectCycles(graph);

    Assert.IsNotNull(cycles, "Empty non-null collection should
        have been returned.");
    Assert.AreEqual(1, cycles.Count, "There should be a single
        cycle between vertexA and vertexB.");
    Assert.IsTrue(cycles.First().Contains(vertexA), "Cycle
        should contain vertexA");
    Assert.IsTrue(cycles.First().Contains(vertexB), "Cycle
        should contain vertexB");
}

[TestMethod]
public void TwoVertices_TwoDifferentConnections_NoCycle()
{
    var vertexA = new IndexedVertexMock();
    var vertexB = new IndexedVertexMock();

    vertexA.Neighbours.Add(new Tuple<IIndexedVertex,
        RelationshipViewModelBase>(vertexB, new
        InheritanceRelationshipMock()));

    vertexB.Neighbours.Add(new Tuple<IIndexedVertex,
        RelationshipViewModelBase>(vertexA, new
        CompositionRelationshipMock()));

    IIndexedGraph graph = new IndexedGraphMock { Vertices = new
         List<IIndexedVertex> { vertexA, vertexB } };

    var cycles = _tarjan.DetectCycles<
        InheritanceRelationshipMock>(graph);

    Assert.IsNotNull(cycles, "Empty non-null collection should
        have been returned.");
    Assert.AreEqual(0, cycles.Count,
"There shouldnt be any cycles between vertexA and vertexB
    because of different type.");
}
```

Part VII
Závěr

V datovém modelování si většina vývojářů představí některou z notací E-R diagramu. Takový pohled ovšem značně omezuje jejich uplatnitelnost, protože jsou použitelné pouze v relačně databázových systémech. Výhodou popsaného a implementovaného přístupu k datovému modelu podporující standard UML za využití profilu od Scotta Amblera je vyšší platformní nezávislost na databázové technologii i použitém programovacím jazyce. Další nezanedbatelnou výhodou modelů dodržující standard UML je, že byl kladně přijat komunitou vývojářů i analytiků, což usnadňuje jeden z hlavních cílů modelů - vzájemnou komunikaci.

Teoretická část popisuje rozdílný pohled na řízení vývoje softwaru, důležitost volby vhodné metodiky a shrnuje vlastnosti obou skupin v tabulce 1. V dalších kapitolách jsou popsány principy modelování se zaměřením na modely s notací UML, především diagramu tříd. V závěrečné části jsou vymezeny metody objektově orientované tvorby softwaru, včetně uvedení návrhových a architektonických vzorů, které byly v praktické části práce implementovány.

Praktická část provází čtenáře strukturou programu a jeho vývoje od základních komponent a principu jejich fungování, po vybrané konkrétní detaily z jejich implementací. Zdrojový kód programu obsahuje přes 20 tisíc řádků kódu, proto byly zahrnuty pouze ukázky implementací navazující na termíny z teoretické části a ty, které závažným způsobem ovlivňují funkcionalitu programu.

Autor se domnívá, že výsledná podoba programu splňuje zadání práce ve smyslu dodržení standardu UML, implementace doporučeného profilu a umožňuje vytvářet diagramy včetně generování kódu do objektových jazyků a SQL. Nicméně autor podotýká, že manipulace s diagramem není bezchybná a zejména chybí možnost volného rozmístění asociačních vztahů.

Při vývoji autor dbal na použití objektově orientovaných principů, což je do jisté míry prokázáno užitím vybraných návrhových a architektonických vzorů, které umožňují projekt dekomponovat a distribuovat v samostatných komponentách v podobě knihoven. Ve zmíněných rozšiřitelných knihovnách a výsledné aplikaci spatřuje autor hlavní přínosy této práce. Program může sloužit jako podpora při výuce předmětů souvisejících se softwarovým inženýrstvím či přímo datovým modelováním, nebo v malých vývojových týmech, které spoléhají na agilní vývoj, nepožadují detailní funkce a modely užívají zejména v podobě náčrtků.

Part VIII
Seznam použitých zdrojů

Seznam literatury

AMBLER, S. UML Data modeling profile, 2003a. Dostupné z: <http://www.agiledata.org/essays/umlDataModelingProfile.html>.

AMBLER, S. Mapping Objects to Relational Databases, 2003b. Dostupné z: <http://www.agiledata.org/essays/mappingObjects.html>.

AMBLER, S. Data modeling, 2003c. Dostupné z: <http://www.agiledata.org/essays/agileDataModeling.html>.

AMBLER, S. W. *The Object Primer*. Cambridge University Press, 3rd ed. edition, 2004.

BLAINE WASTELL, N. D. R. S. R. C. F. C. *Developer's Guide to Microsoft Prism Library 5.0 for WPF*. Microsoft Corporation, 5th version edition, 2014.

BUCHALCEVOVÁ, A. *Metodiky vývoje a údržby informačních systémů*. Grada, 1. vyd. edition, 2005.

ERICH GAMMA, J. V. R. H. R. J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

FOWLER, M. *Destilované UML*. Grada, 1. vyd. edition, 2009.

FOWLER, M. *Patterns of enterprise application architecture*. Addison-Wesley, c2003.

FREEMAN, E. – FREEMAN, E. *Head first design patterns*. OÂ´Reilly, 1st ed. edition, 2004.

MERUNKA, V. *Objektové modelování*. Alfa Nakladatelství, 1. vyd. edition, 2008.

MIČKA, P. Depth First Search, 2010a. Dostupné z: <http://www.algoritmy.net/article/1378/Prohledavani-do-hloubky>.

MIČKA, P. Tarjanův algoritmus, 2010b. Dostupné z: <http://www.algoritmy.net/article/1515/Tarjanuv-algoritmus>.

Seznam obrázků

Seznam tabulek

Seznam algoritmů