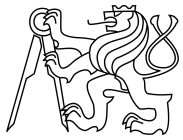
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Vizualizace instancí OntoUML modelů
Student:	Bc. Matúš Vološin
Vedoucí:	Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Navrhněte a implementujte systém pro vizualizaci instancí OntoUML modelů. Řešení postavte na platformě DynaCASE. Jádrem práce je návrh a implementace metamodelu a logiky těchto vizualizací. Uživatel by měl být schopen pro určitý OntoUML model vytvářet vlastní instance s tím, že systém mu bude nabízet validní varianty a hlídat, aby nevznikla nevalidní instance. Součástí řešení je i návrh a implementace modelování integritních omezení instancí. Výsledné uživatelské rozhraní může být minimalistické, ale funkční. Výsledné řešení důkladně otestujte a demonstrujte na netriviálním příkladu.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 13. listopadu 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práca

Vizualizace instancí OntoUML modelů

Bc. Matúš Vološin

Vedúci práce: Ing. Robert Pergl, Ph.D.

8. mája 2016

Poďakovanie

Chcel by som týmto veľmi poďakovať svojmu vedúcemu Ing. Robertovi Perglovi, Ph.D., pretože bez jeho vedenia a podpory by táto práca nikdy nevznikla. Taktiež by som chcel poďakovať hlavnému vývojárovi frameworku DynaCASE Petrovi Uhnákovi, za jeho pomoc a podporu pri práci s týmto frameworkom.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 8. mája 2016 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na túto prácu

Vološin, Matúš. *Vizualizace instancí OntoUML modelů.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práca si kladie za cieľ implementáciu dvoch nových editorov na modelovanie OntoUML diagramov a OntoUML inštančných diagramov. Riešenie bude postavené na frameworku DynaCASE, ktorý je implementovaný v jazyku Smalltalk a virtuálnom stroji Pharo. Editory budú schopné vytvorené diagramy zobrazovať a validovať. OntoUML diagram bude validovaný podľa platných syntaktických pravidiel a inštancný diagram podľa napojeného OntoUML diagramu. Implementovaná bude aj podpora overovania integritných obmedzení na úrovní inštancií. Validácia a niektoré ďalšie dôležité funkcie budú demonštrované na príkladoch vytvorených priamo vo virtuálnom stroji Pharo.

Klíčová slova OntoUML, UML, ontológia, DynaCASE, Pharo, Smalltalk, metamodel OntoUML, inštančné modelovanie, integritné obmedzenia, validácia diagramu

Abstract

Main goal of this thesis is to implement two new editors for modeling OntoUML diagrams and OntoUML instance diagrams. The solution will be based on DynaCASE framework that is implemented in Smalltalk language and virtual machine Pharo. OntoUML diagram will be validated according to OntoUML syntax rules and instance diagram according to connected OntoUML diagram. Implementation will include verification of the integrity constraints at instances. Validation and some other important features will be demonstrated in the examples created directly in a virtual machine Pharo.

Keywords OntoUML, UML, ontology, DynaCASE, Pharo, Smalltalk, metamodel OntoUML, instance level modeling, integrity constraints, diagram validation

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Každý človek má iba odmedzenú predstavivosť a abstraktné myslenie, preto často potrebuje grafickú (vizuálnu) alebo textovú pomoc. Každý analytický OntoUML model alebo triedny UML model reprezentuje iba určitú predstavu, definíciu možného sveta, ktorý zatiaľ neexistuje. Vznik sveta definovaného takýmto modelom chápeme ako vznik inštancií elementov definovaných v tomto modeli.

Takáto abstraktná definícia je však pre mnohých ľudí príliš zložitá na pochopenie, a preto je potrebné vzniknuté inštancie vizualizovať.

Cieľ práce

Hlavným cieľom tejto práce je navrhnúť, impelementovať a otestovať komponentu pre vizuálizáciu instancií OntoUML modelov. Hlavná časť práce sa bude venovať pozadiu tejto funkcionality, tzv. backendu. Tento backend bude reprezentovať OntoUML model v pamäti, validovať syntakticku správnosť modelu, vytvárať a ukladať instancie elementov modelu. Súčasťou funkcionality bude aj vytváranie a validácia integritných obmedzení OntoUML modelu. Nad týmto backendom bude vytvorené aj grafické GUI. Toto GUI bude jednoduché a vytvorené ako komponenta frameworku DynaCase vyvíjaného na FIT ČVUT.

Štruktúra práce

V prvej kapitole bude venovaná pozornosť samotnému jazyku OntoUML. Tomuto jazyku som sa venoval už vo svojeje bakalárskej práci [1], a preto bude táto práca zameraná hlavne na veci, ktoré boli v bakalárskej práci urobené chybne alebo nedostatočne vysvetlené. Podľa týchto zmien bude navrhnutý a implementovaný nový OntoUML metamodel v Smalltalku.

Po uložení OntoUML modelu v pamäti bude navrhnutý model na zachytenie inštancií OntoUML modelu. Model inštancí bude obsahovať aj pridávanie a kontrolu integritných obmedzení. Týmto bude dokončený tzv. backend.

Po dokončení backendu bude vytvorená DynaCase komponenta na vizualizáciu týchto modelovaných inštancií. Ako posledná časť bude vytvorená veľká sada ako unit testov, tak aj netriviálny príklad použitia tohto riešenia. Pomocou výslednej aplikácie bude užívateľ môcť namodelovať OntoUML model a následne začať vytvárať inštancie elementov tohto modelu. Všetko vo vizuálnej podobe pomocou frameworku DynaCASE a virtuálneho stroja Pharo.

Kapitola 1

Jazyk OntoUML

V tejto prvej kapitole sa budeme venovať modelovaciemu jazyku OntoUML. Tento jazyk bol podrobne popísaný už v predošlej práci s názvom Transformácia OntoUML do Smalltalku [1]. Táto práca však obsahovala viacero chýb a nepresností, ktoré budu opravené. Podobne niektoré definície v špecifikácii OntoUML boli prepracované, a preto je ich potrebné znova napísať. Dôraz bude kladený tiež na praktické ukážky modelov, ktoré boli sčasti vytvárané podľa Foundational Ontology Patterns (FOPs) [2].

OntoUML je profil Unified Modeling Language (UML). Teória OntoUML je postavená na Unified Foundational Ontology (UFO). UFO definoval Giancarlo Guizzardi vo svojej Ph.D práci [3] už v roku 2005. V poslednom období vývoj napreduje pomerne rýchlo. Celé univerzum v UFO-A [4] je rozdelené na dve časti.

1. Monadic - reprezentuje elementy, obrázok 1.1
2. Relation - reprezentuje vzťahy medzi elementami, obrázok 1.2

1.1 Vlastnosti elementov

Ešte predtým ako začneme definovať jednotlivé OntoUML elementy, je potrebné popísať základné vlastnosti, ktoré budú spomínané v definíciách.

1.1.1 Definície

- Existenčná závislosť(Existential Dependence) - Inštancia x je existenčne závislá na inštancii y práve vtedy, ak musí nutne existovať y stále ak existuje x.[5]
- Rigidita - Typ T je rigídny pre každú inštanciu x, práve vtedy ak x je nutne (v modálnom slova zmysle) inštanciou T. Inak povedané, ak x je

inštanciou T vo svete w, tak x musí byť inštanciou T v každom možnom svete w'.[5]

- Anti-Rigidita - Typ T je anti-rigídny pre každú inštanciu x, práve vtedy, keď je možné (v modálnom slova zmysle), že x nemusí byť inštanciou T. Inak povedané, ak je x inštanci T v nejakom svete w, tak je možné, že existuje nejaký svet w', kde x nie je inštanciou T.[5]
- Relačná závislosť(Relational Dependence) - Typ T je relačne závislý na inom type P cez reláciu R práve vtedy, ak pre každú inštanciu x typu T existuje inštancia y typu P taká, že x a y sú v relácii R.[5]
- Generická závislosť (Generic Dependance) - Inštancia y je genericky závislá na type T práve vtedy, ak je pre existenciu inštancie y nutné, aby existovala aj inštancia T.[5]

1.1.2 Prehľad vlastností

Všetky základné vlastnosti, identita, rigidita, existenčná, resp. relačná závislosť elementu sú prehľadne spísané v tabuľke 1.1. Význam meta-atribútov v tabuľke:

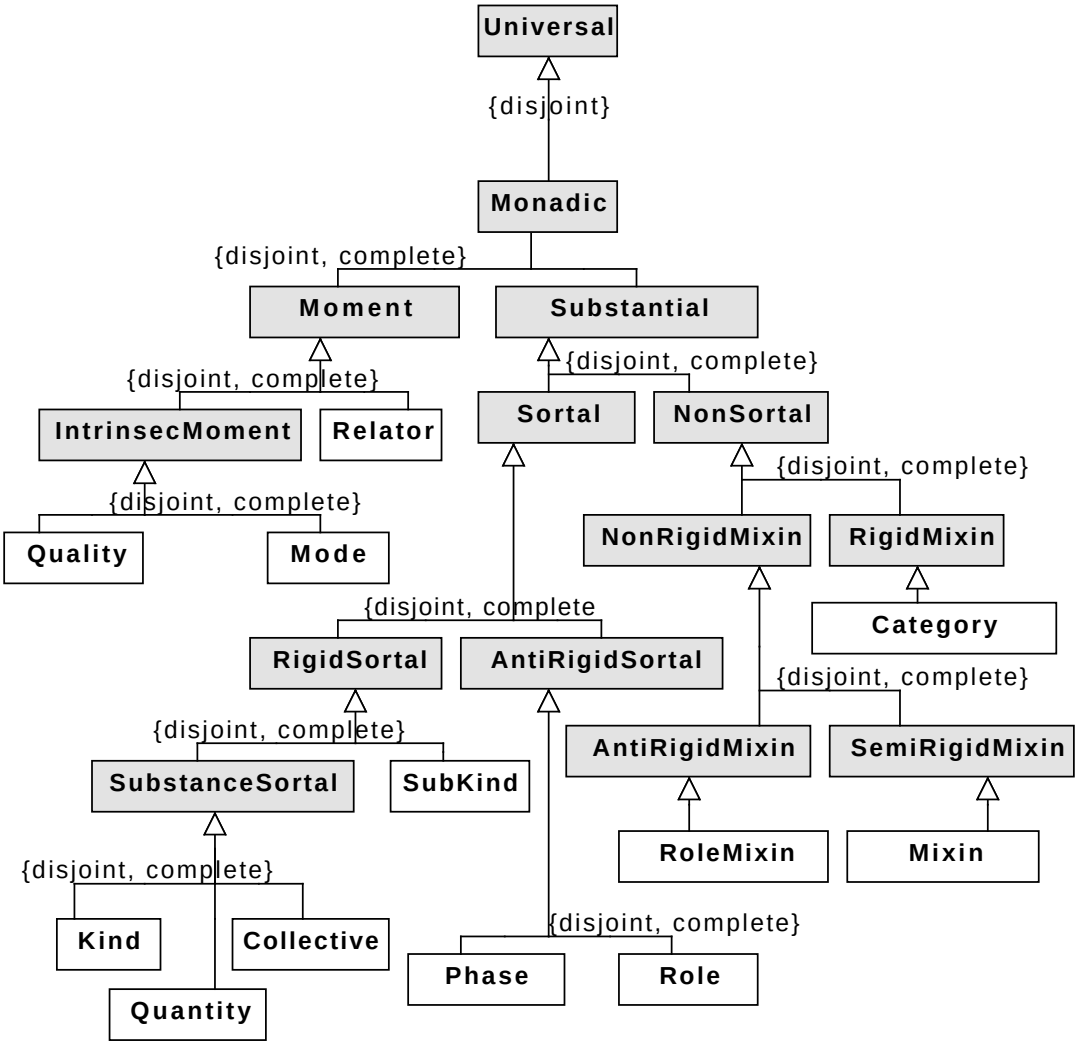
- I identita elementu,
- O môže poskytovať identitu,
- +R element je rigídny,
- -R element je anti-rigídny,
- ~R element je semi-rigídny.

1.2 Sortal

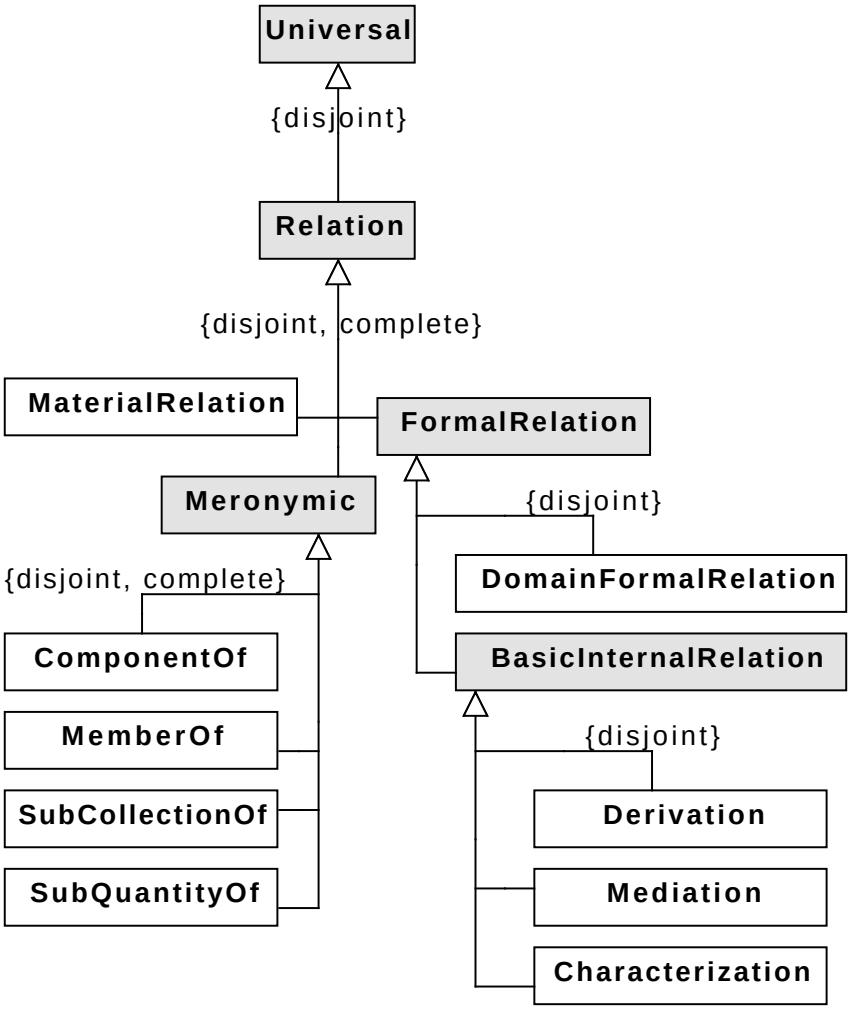
Základným spoločným znakom všetkých sortálov je to, že ich inštancie majú identitu. Túto identitu buď majú vlastnú (napr. «kind») alebo ju nejakým spôsobom zdedia (napr. «subkind»).

1.2.1 Kind

Stereotyp «kind» reprezentuje sortal, ktorého inštancie sú funkčné komplexy. Inými slovami jeho instancie sú prirodzené druhy (napr. Person, Lake, River) alebo veci (napr. House, Car, Bus). «kind» je rigídny, má vlastnú identitu a túto identitu dokáže poskytovať aj ďalej. Dokáže existovať nezávisle [6]. Príklad na obrázku 1.3.



Obr. 1.1: Vyňatok z ontologicky presného OntoUML metamodelu týkajúci sa elementov podľa [6]



Obr. 1.2: Vyňatok z ontologicky presného OntoUML metamodelu týkajúci sa vzťahov medzi elementami podľa [6]

Tabuľka 1.1: Prehľad vlastností OntoUML elementov

	I	O	+R	-R	~R	Relačne závislý	Existenčne závislý
«kind»	✓	✓	✓				
«subkind»		✓	✓				
«role»		✓		✓		✓	
«phase»		✓		✓			
«collective»	✓	✓	✓				
«quantity»	✓	✓	✓				
«category»			✓				
«rolemixin»				✓		✓	
«mixin»					✓		
«relator»							✓
«mode»							✓
«quality»							✓

1.2.2 SubKind

Stereotyp «subkind» reprezentuje rigídny sortal. Nemá vlastnú identitu, ale musí ju dostať (zdediť) od nejakého sortalu, ktorý poskytuje identitu (napr. «kind»). Identitu nedokáže poskytovať ďalej [6]. Príklad na obrázku 1.3.

1.2.3 Role

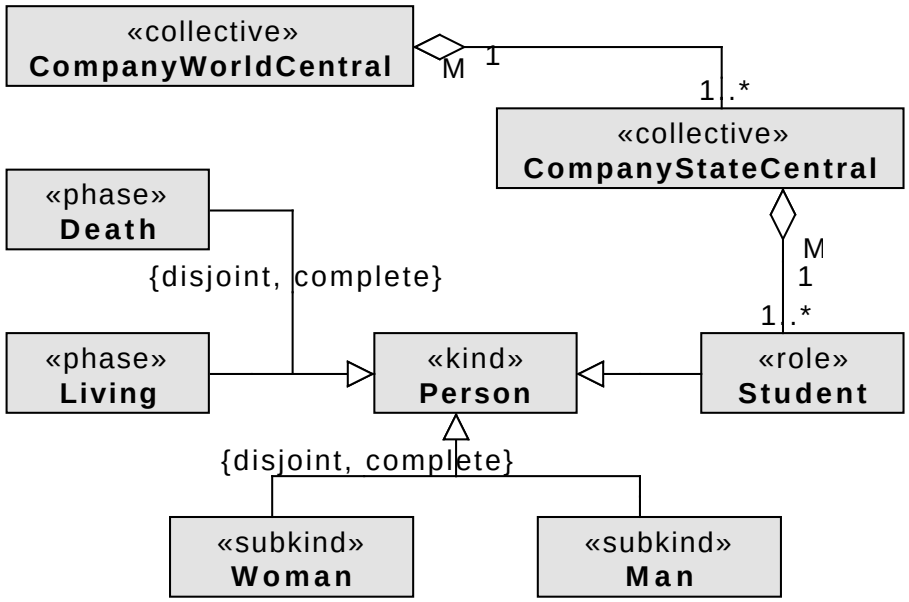
Stereotyp «role» patrí medzi anti-rigídne sortály, je relačne závislý a bližšie nám špecifikuje rolu nejakého objektu v realite (napr. «kind» Person môže mať «role» Student a Teacher). Podobne ako «subkind» nemá vlastnú identitu, ale musí ju zdediť od objektu, na ktorom je relačne závislý [6]. Príklad na obrázku 1.3.

1.2.4 Phase

Stereotyp «phase» patrí medzi anti-rigídne sortály a charakterizuje nám určitú vlastnosť objektu po určitú dobu, pokiaľ táto vlastnosť platí (napr. «kind» Person môže mať dve «phase» Living, Death a každá z týchto «phase» charakterizuje človeka určitý čas v realite). GeneralizationSet (množina nadtypov) pre každú «phase» je stále disjoint a complete [6]. Príklad na obrázku 1.3.

1.2.5 Collective

Stereotyp «collective» reprezentuje sortál, ktorého inštancie sú kolekcie objektov zjednotené nejakým zjednocujúcim vzťahom (napr. v «collective» Company sú jednotlivé «role» Employee a spoločný vzťah je, že pracujú v tejto



Obr. 1.3: Príklad použitia entít typu «kind», «subkind», «role», «phase», «collective»

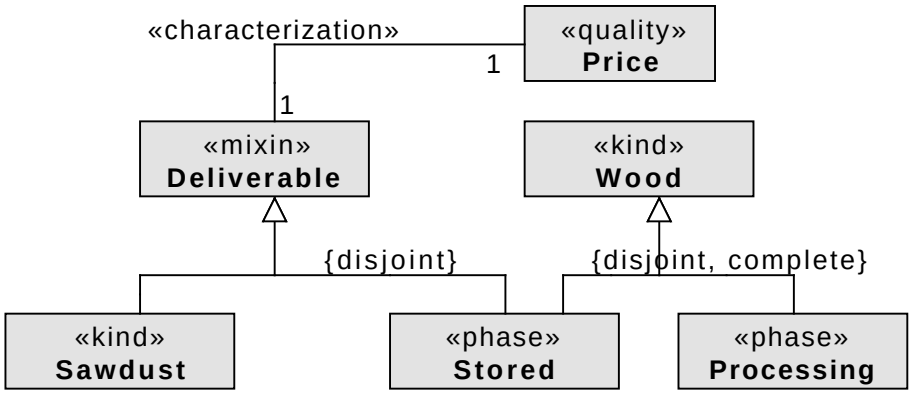
spoločnosti). Všetky «collective» môžu byť rozšíriteľné, t.j. majú rozšíriteľný princíp identity. Rozširiteľnosť znamená, že ak inštancia «collective» stratí nejakých členov, priberie nových alebo nahradí aktuálneho člena iným, tak celý «collective» nezmení svoju identitu (napr. ak z inštancie «collective» Company odoberieme nejakého «role» Employee, tak identita tejto «collective» Company ostane zachovaná a zmení sa iba jej kolekcia zamestnancov) [6]. Príklad na obrázku 1.3.

1.2.6 Quantity

Posledným zástupcom sortálov je stereotyp «quantity». Inštancie «quantity» nám nejakým spôsobom reprezentujú kvantitu nejakého objektu. Každá «quantity» reprezentuje maximálne spojitý objekt, ktorý nie je podčasťou nejakej väčšej časti rovnakého typu (napr. «quantity» Water v «kind» Bottle zaberá celý vymedzený priestor a tekutina je spojitá) [6]. Príklad na obrázku 1.6.

1.3 Non Sortal

Všetky non sortály, narozdiel od sortálov, nemajú vlastnú identitu vo svete a ani sa k nej nijakým spôsobom nemôžu dostať. Reprezentujú iba akési abs-



Obr. 1.4: Príklad použitia entity «mixin», «quality»

traktné vlastnosti, ktoré sú spoločné pre viac disjunktných typov [6].

1.3.1 Category

Stereotyp «category» je rigídny, nemá vlastnú identitu a dokáže existovať sám, t.j. je nezávislý. «category» reprezentuje spoločné vlastnosti objektov typu «kind», ktoré tvoria disjunktnú množinu nadtypov GeneralizationSet (napr. «category» PhysicalObject reprezentuje objekty v reálnom svete «kind» Car, Bus, House atď.) [6]. Príklad na obrázku 1.5.

1.3.2 RoleMixin

Stereotyp «rolemixin» je anti-rigídny, nemá žiadnu identitu a je relačne závislý. Reprezentuje spoločné vlastnosti objektov typu «role» (napr. v «kind» Shop nakupujú rôzny «rolemixin» Customer a tento «rolemixin» obsahuje spoločné vlastnosti «role» PhysicalCustomer a «role» CorporateCustomer) [6]. Príklad na obrázku 1.5.

1.3.3 Mixin

Stereotyp «mixin» reprezentuje spoločné vlastnosti, ktoré sú pre niektoré objekty povinné a pre iné nepovinné, až náhodné (napr. «mixin» Seatable je pre «kind» Chair charakteristický, povinný ale, pre «phase» SolidCreate iba nepovinný, pretože «phase» SolidCreate je primárne určený k inému účelu) [6]. Príklad na obrázku 1.3.

1.4 Moment

Všetky momenty, niekedy nazývané aj aspekty, sú vždy existenčne závislé na nejakej inej entite, objekte. Toto je podstatný rozdiel oproti sortálom a non sortálom, ktoré sú existenčne nezávislé.

1.4.1 Relator

«relator» je stereotyp, ktorý je existenčne závislý na minimálne dvoch rozdielnych entitách. Týmto entitám sprostredkúvava «material» reláciu. K entitám, na ktorých je existenčne závislý, sa pripája cez «mediation» reláciu (napr. «relator» Wedding reprezentuje vzťah medzi dvoma entitami «role» Husand a Wife) [6]. Príklad na obrázku 1.5.

1.4.2 Mode

Stereotyp «mode» reprezentuje nejakú nemerateľnú vlastnosť objektu narozdiel od «quality». «mode» je stále napojený na charakterizovaný objekt cez «characterization» 1.5.2 reláciu a je existenčne závislý na práve jednej entite (napr. «kind» Person má nejaké «mode» Symptom) [6]. Príklad na obrázku 1.5.

1.4.3 Quality

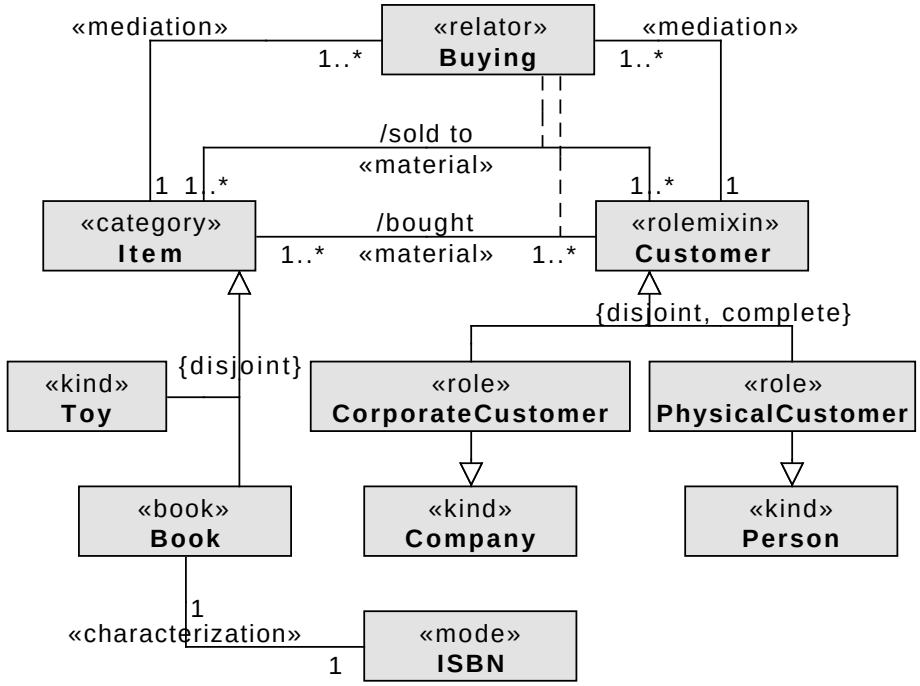
Stereotyp «quality», podobne ako «mode», reprezentuje nejakú vlastnosť objektu. Špeciálne pre «quality» je, že reprezentuje nejakú merateľnú vlastnosť objektu. Stále je napojený na charakterizovaný objekt cez reláciu «characterization» 1.5.2. «quality» často degraduje iba na atribúty charakterizovaného objektu (napr. «kind» Car je charakterizovaný svojou «quality» Color). Z čisto objektového hľadiska volíme samostatný objekt pre «quality» iba v prípade, že potrebujeme nejaké špecifické správanie [6]. Príklad na obrázku 1.3.

1.5 Relationships

Vzťahy, relácie medzi entitami sú reprezentované tzv. asociáciami. Relácia je matematický pojem, ktorý nám reprezentuje vzťah medzi dvoma inštanciami tried. Pri každej relácii budú spomenuté aj pravidlá, ktoré musí spĺňať a ktoré sú implementované pri syntaktickej kontrole modelu.

1.5.1 Mediation

Relácia «mediation» je typ formálnej relácie. Vyjadruje vzťah medzi entitou «relator» a príslušným objektom, na ktorom je «relator» existenčne závislý. Na oboch konch musí mať «mediation» multiplicitu aspoň 1 [6]. Príklad na obrázku 1.5.



Obr. 1.5: Príklad použitia entít typu «relator», «category», «rolemixin», «mode»

1.5.2 Characterization

Relácia «characterization» je typ formálnej relácie, ktorá reprezentuje vzťah medzi «quality», «mode» a príslušnou charakterizovanou entitou. Koniec relácie napojený na charakterizovaný objekt musí mať multiplicitu práve 1 a koniec napojený na «quality» alebo «mode» musí mať multiplicitu aspoň 1 [6]. Príklad na obrázku 1.3.

1.5.3 Material

Relácia «material» spája entity, ktorých spoločný vzťah je reprezentovaný cez «relator». Každá «material» relácia musí byť napojená na práve jednu «derivation» 1.5.5. Multiplicity «material» relácie sú derivované (prevzaté) od «mediation» relácie, ktorou je entita napojená na príslušný «relator». Keďže «mediation» relácia má minimálne multiplicity na oboch koncoch aspoň 1, tak aj «material» relácia má minimálne multiplicity aspoň 1 [6]. Príklad na obrázku 1.5.

1.5.4 Formal

Relácie «formal» sú vždy odvodené relácie a spájajú elementy priamo. Na značenie používame UML symbolickú reprezentáciu (/). Väčšionu vieme «formal» relácie nejako matematicky spočítať podľa ostatných elementov v modeli (napr. «kind» Person má «formal» reláciu pomenovaná /olderThan) [6].

1.5.5 Derivation

Relácia «derivation» je typ formálnej relácie. Reprezentuje vzťah medzi «relator» a príslušnou «material» reláciou. Tento špeciálny typ relácie nám ukazuje, ako môže byť inštancia «material» relácie odvodená (derivovaná) z inštancie «mediation» relácie. Veľmi zjednodušene povedané, relácia «derivation» pomenúva «material» reláciu [7]. Koniec relácie napojený na «relator» musí mať multiplicitu práve 1 a druhý koniec napojený na «material» reláciu musí mať multiplicitu aspoň 1 [6]. Príklad na obrázku 1.5, prerušovanou čiarou.

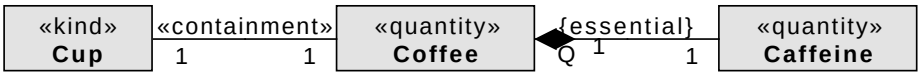
1.5.6 Generalization

Špeciálnym typom relácie je generalization (vzťah nadtyp, podtyp). V tabuľkách 1.2, 1.3, 1.4 sú všetky možnosti dvojíc elementov, medzi ktorými je možné viesť generalization reláciu. Stĺpec označuje podtyp a v riadkoch sú uvedené všetky jeho možné nadtypy (napr. stereotyp «kind» môže mať ako nadtyp «category» alebo «mixin»). Všetky generalization relácie združujeme to tzv. množiny nadtypov (generalization set), ktorá môže mať vlastnosti disjoint (všetky prvky množiny sú navzájom disjunktné) alebo complete (množina obsahuje všetky možné podtypy v danom svete).

1.6 Part-Whole

Posledným typom relácií sú tzv. part-whole relácie. Všetky part-whole relácie môžu mať špecifické vlastnosti:

- Esentiálna časť (essential parthood) - Inštancia x je esentiálnou časťou inštance y, ak y je existenčne závisle na x a inštancia x je nevyhnutne Časť inštancie y.[8]
- Neoddeliteľná časť (inseparable parthood) - Inštancia x je neoddeliteľnou časťou inštance y, ak x je existenčne závislé na y a x je nevyhnutne Časť inštancie y.[8]
- Povinná časť (mandatory parthood) - Inštancia x je povinnou časťou inštancie y, ak y je genericky závislé na type T, ktoré je inštanciou x a inštance x je nevyhnute Časť inštancie typu T.[8]



Obr. 1.6: Príklad použitia entity «quantity» a jej relácií

- Povinný celok (mandatory whole) - Inštancia y je povinný celok pre inú inštanciu x, ak x je genericky závislé na type T, ktoré je insštanciou y a inštancia x je nevyhnute Časťou inštancie typu T.[8]

1.6.1 ComponentOf

Relácia «componentof» je typ part-whole (časť-celok) relácie. Pomocou «componentof» relácie sú časti naviazané k funkčnému celku (napr. ruka je časťou môjho tela). Funkčný celok sa skladá z rôznych častí, ktoré majú rôzne úlohy a prispievajú k funkčnosti daného celku. Toto je podstatný rozdiel medzi funkčným celkom a «collective», v ktorom majú všetky časti rovnakú úlohu, t.j. sú ekvivalentné. Relácia «componentof» musí byť na oboch koncoch napojená na inštanciu funkčného celku [6].

1.6.2 MemberOf

Relácia «memberof» je typom part-whole (časť-celok) relácie medzi prvkom z «collective» (členom) a samotným «collective» alebo medzi dvoma «collective», kde jeden vystupuje ako celok a druhý ako časť, tzv. podkolektív (napr. «kind» Tree je súčasť «collective» Forest) [6]. Príklad na obrázku 1.3.

1.6.3 SubCollectionOf

Relácia «subcollectionof» je typom part-whole (časť-celok) relácie medzi dvoma entitami typu «collective» (napr. «collective» University má prvky «collective» Faculty). Každý podkolektív však musí byť zjednotený cez inú zjednocujúcu reláciu ako jeho nadkolektív, pretože každý «collective» je maximálna množina cez danú zjednocujúcu reláciu a tým pádom nemôže mať podkolektív rovnakého druhu [6]. Príklad na obrázku 1.3.

1.6.4 SubQuantityOf

Relácia «subquantityof» je typom part-whole (časť-celok) relácie medzi dvoma «quality» entitami. Multiplicita na konci relácie pripojená k časti musí byť stále práve 1 [6]. Príklad na obrázku 1.6.

Tabuľka 1.2: Tabuľka špecializácií v OntoUML, 1. časť [6]

	«kind»	«quantity»	«collective»	«subkind»
«kind»				✓
«quantity»				✓
«collective»				✓
«subkind»				✓
«role»				
«phase»				
«category»	✓	✓	✓	✓
«mixin»	✓	✓	✓	✓
«rolemixin»				
«relator»				
«mode»				
«quality»				

Tabuľka 1.3: Tabuľka špecializácií v OntoUML, 2. časť [6]

	«role»	«phase»	«category»	«rolemixin»
«kind»	✓	✓		
«quantity»	✓	✓		
«collective»	✓	✓		
«subkind»	✓	✓		
«role»	✓	✓		
«phase»	✓	✓		
«category»			✓	✓
«mixin»	✓	✓	✓	✓
«rolemixin»	✓	✓		✓
«relator»	✓	✓		
«mode»				
«quality»				

Tabuľka 1.4: Tabuľka špecializácií v OntoUML, 3. časť [6]

	«mixin»	«relator»	«mode»	«quality»
«kind»				
«quantity»				
«collective»				
«subkind»				
«role»				
«phase»				
«category»	✓			
«mixin»	✓			
«rolemixin»				
«relator»		✓		
«mode»			✓	
«quality»				✓

Kapitola 2

OntoUML komponenta pre DynaCASE

V tejto kapitole na začiatku stručne popíšeme framework DynaCASE a následne vytvoríme novú komponentu pre DynaCASE na prácu s OntoUML diagramami. Okrem vytvorenia a zobrazovania diagramu bude komponenta vykonávať syntaktickú validáciu modelu podľa platných OntoUML pravidiel. Popíšeme detailne návrh metamodelu a jeho použitie. Ako posledné rozoberieme samotnú komponentu na ovládanie zobrazovaných objektov v DynaCASE canvase (controllers).

2.1 Modelovací framework DynaCase

Dynamic Computer Aided Software Engineering Tool alebo skrátene DynaCASE je modelovacia platforma vyvinutá a spravovaná v dynamickom prostredí Pharo [9] a napísaná v jazyku Smalltalk. Nástroj je vyvíjaný v rámci vedecko-výskumnej skupiny Centrum pre Konceptuálne Modelovanie a Implementácie (CCMi). Základnou myšlienkou tejto platformy je jednoduchá rozšíriteľnosť o nové notácie, algoritmy a prácu s modelmi [10]. V súčasnej dobe nástroj podporuje notácie:

- BORM - Business Objects Relation Modeling Object-Relation Diagrams,
- DEMO - Design amd Engineering Methodology for Organizations,
- UML Class Diagrams,
- FSM - Finite State Machines.

Aby bolo možné vizualizovať inštancie OntoUML diagramu v DynaCASE, je potrebné vytvoriť novú komponentu, v ktorej bude možné namodelovať nový OntoUML diagram a nejakým spôsobom ho reprezentovať v pamäti.

2.2 Metamodel

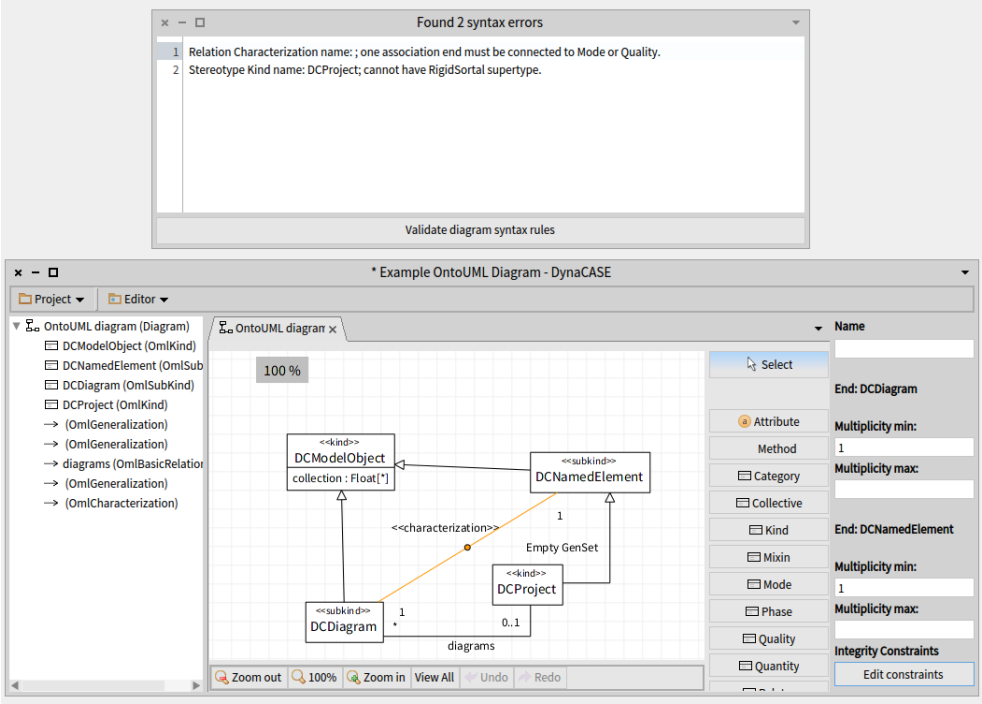
Ešte predtým ako vytvoríme a zaregistrujeme novú DynaCASE komponentu, je potrebné navrhnúť metamodel. OntoUM Lometamodel obsahuje: ontologické elementy popísané v UFO a integritné obmedzenia pokrývajúce možné väzby, ktoré môžu vzniknúť medzi elementami [11]. Existujú viaceré verzie implementácie OntoUML metamodelu, s ktorých sa pri návrhu dá vychádzať. Návrh bude vychádzať s oficiálneho ontologického modelu v OntoUML špecifikácií [6] od skupiny Ontology and Conceptual Modeling Research Group (nemo), ktorá vyvíja a spravuje tento modelovací jazyk. Toto je už druhá verzia tohto metamodelu pre rovnaký účel. V predošlej verzii boli problémy s prílišnou všeobecnosťou a zlou integráciou priamo na DynaCASE. Starý metamodel nevyužíval preddefinované objekty DynaCASE (napr. `DCNamedElement`) a následne vznikali problémy pri previazaní elementov alebo mazaní. Metamodel je rozdelený na tri logické časti:

1. jadro metamodelu na obrázku 2.3,
2. entity OntoUML na obrázku 2.4,
3. relácie medzi entitami na obrázku 2.5.

Každá trieda metamodelu má prefix „Oml“, pretože v prostredí Pharo sú všetky triedy globálne prístupné a aby sme nejakú už existujúcu triedu nechtiac neprepísali, musíme používať prefix.

Jadro metamodelu je tvorené základnými triedami, ktoré reprezentujú samotný diagram, pridávajú, mažu elementy z modelu a komunikujú s DynaCASE. Jádro obsahuje tieto triedy:

- OmlDiagram,
- OmlElement,
- OmlElementClass,
- OmlRelation,
- OmlGeneralizationSet,
- OmlIntegrityConstraint,
- FAMIXAttribute,
- FAMIXMethod.

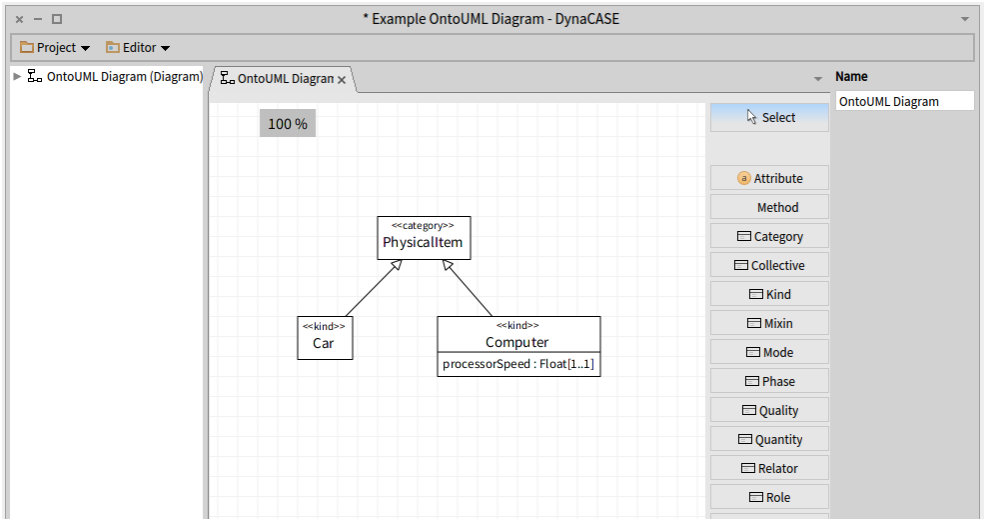


Obr. 2.1: Ukážka validačného okna

Trieda OmlDiagram slúži ako hlavná trieda celého metamodelu. Drží kolekciu elementov (entity aj relácie), pridáva a odoberá elementy. Je potomkom generickej DynaCASE triedy `DCDiagram`. Trieda `OmlElement` reprezentuje OntoUML element v diagrame a je potomkom triedy, ktorá v DynaCASE tvorí základ každého modelového elementu `DCNamedElement`. V tejto triede začína syntaktická kontrola elementu v modeli a tiež obsahuje kontrolu validnosti mena pre Smalltalk. Triedy `OmlElementClass` a `OmlRelation` reprezentujú OntoUML entitu, resp. reláciu a implementujú ďalšie syntaktické pravidlá špecifické pre každý element.

Špecifikom OntoUML sú tzv. Generalization Set alebo množiny nadtypov. Každý element môže obsahovať niekoľko množín nadtypov, ktoré sú reprezentované pomocou triedy `OmlGeneralizationSet`.

V základe sa predpokladá, že tento metamodel nebude používaný mimo DynaCASE, avšak aj napriek tomu dokáže fungovať samostatne. Jediná závislosť je na triedach `FAMIXAttribute` a `FAMIXMethod`. OntoUML atribúty a metódy majú rovnaké správanie ako v UML, preto boli tieto triedy prevzaté od DynaCASE komponenty s názvom UML Class Diagram a znovupoužité, aby sa čo najviac dodržalo pravidlo DRY (Don't repeat yourself).



Obr. 2.2: Diagram z ukážkového kódu namodelovaný v DynaCASE

2.2.1 Syntaktická validácia diagramu

Táto validácia nie je vo všeobecnosti jednoduchá. Predošlá práca [1] zachycovala všetky dostupné OntoUML pravidlá z práce [5]. Tieto pravidlá boli následne implementované do navrhnutého metamodelu. Oproti predošlej verzii metamodelu bola validácia vylepšená o reportovanie. V predošlej verzii bola výsledkom validácie iba boolean hodnota true resp. false. Avšak v novej verzii má každé porušenie OntoUML syntaxe svoju chybovú hlášku a užívateľ následne môže interaktívne sledovať, ako prí úpravach diagramu tieto chyby vznikajú alebo zanikajú. Na obrázku 2.1 je ukážka ako vyzerá okno, ktoré zobrazuje syntaktické chyby k diagramu podním. V tomto prípade boli nájdené dve chyby, prvou je, že «kind» nemôže mať ako nadtyp nejaký rigídny sortál a druhou je, že relácia «characterization» musí byť napojená na «quality».

2.2.2 Integritné obmedzenia

Každý OntoUML element alebo relácia môže mať definované vlastné integritné obmedzenia. Tieto integritné obmedzenia sú uložené ako Smalltalkový blok, ktorý vráti boolean (true/false) hodnotu. Tento blok kódu je vždy uložený v triede s názvom `OmlIntegrityConstraint`. Vstupom do tohto bloku je konkrétna inštancia, ktorá je naviazaná na daný OntoUML element alebo reláciu.

2.2.3 Ukážka práce s metamodelom

Ukážka ako pracovať s modelom na jednoduchom príklade:

```
diagram := OmlDiagram named: 'New OntoUML Diagram'.

OmlCategory
    name:       'PhysicalItem'
    diagram:    diagram.
OmlKind
    name:       'Car'
    diagram:    diagram.
OmlKind
    name:       'Computer'
    diagram:    diagram.

attribute := FAMIXAttribute
                named:          'processorSpeed'
                type:           (FAMIXClass named: 'Float')
                multiplicity: (DCFAMIXMultiplicity one).
(metamodel at: 'Computer') addAttribute: attribute.

genSet := OmlGeneralizationSet new disjoint: true.

OmlGeneralization
    parent:             category
    child:              car
    generalizationSet:  genSet.

OmlGeneralization
    parent:             category
    child:              computer
    generalizationSet:  genSet.
```

2.3 Diagram

Diagram v DynaCASE ja ovládaný triedami s postfixom Controller [12]. Každý modelový element má svoj vlastný kontroler. Kontrolery pre OntoUML:

- DCOmlDiagramController,
- DCOmlAttributeContoller,
- DCOmlMethodController,

- DCOmlInheritanceController,
- DCOmlElementClassController,
- DCOmlRelationController.

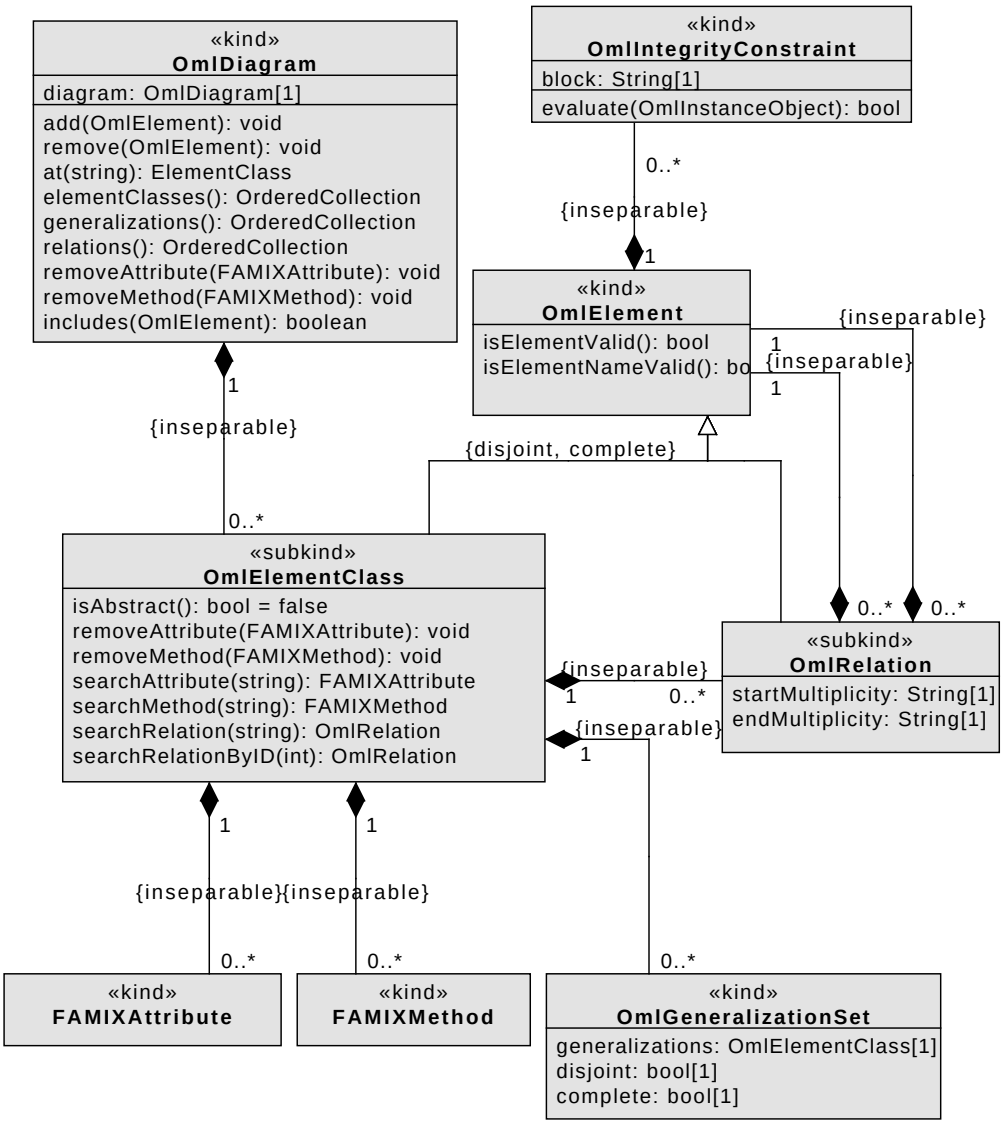
Kontrolery z väčšej časti názvom kopírujú svoje modelové elementy. Rozdiel je hlavne pri `DCOmlInheritanceController`. Tento rozdiel oproti modelu vznikol spôsobom impelementácie. OntoUML je iba profil UML, tým pádom používa niektoré rovnaké tvary a čiary. V tomto prípade je implementácia urobená tak, že každý OntoUML kontroler je potomkom nejakého kontrolera pre UML Class Diagram, používa niektoré jeho implementácie a prepisuje iba tie nevyhnutné. Taktiež používa UML tvary pre triedu, atribúty, metódy, dedičnosť a základnú asociáciu. Na obrázku 2.2 je v DynaCASE namodelovaný jednoduchý diagram podľa ukážkového kódu.

Pre ručné otvorenie nového okna OntoUML pluginu treba spustiť nasledujúci kód:

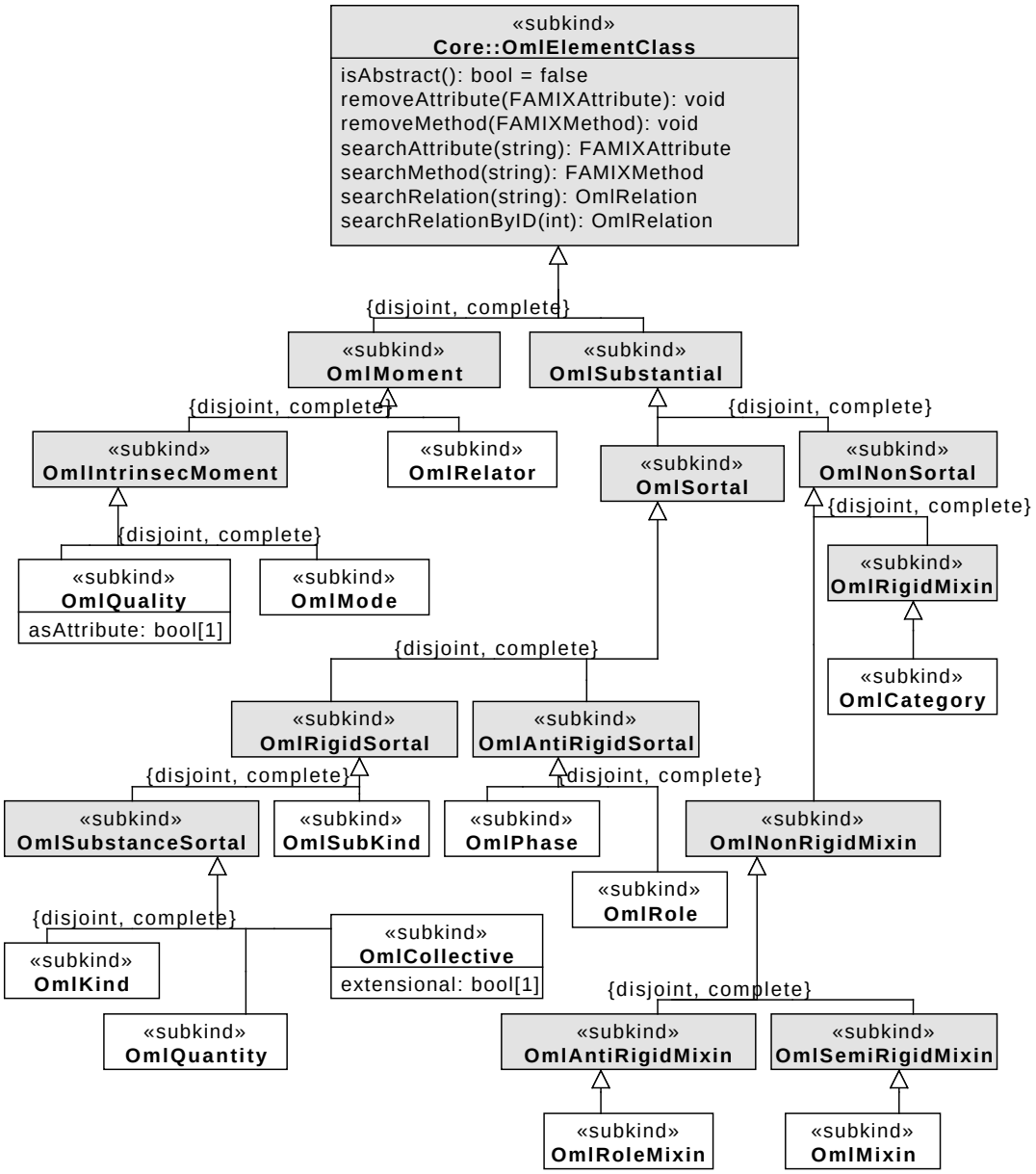
```
project := DCProject new
            projectName: 'Example OntoUML Diagram'.
diagram := self createExampleDiagram.
project addModel: diagram.

DCWorkbench openProject: project
```

2.4 Repozitár

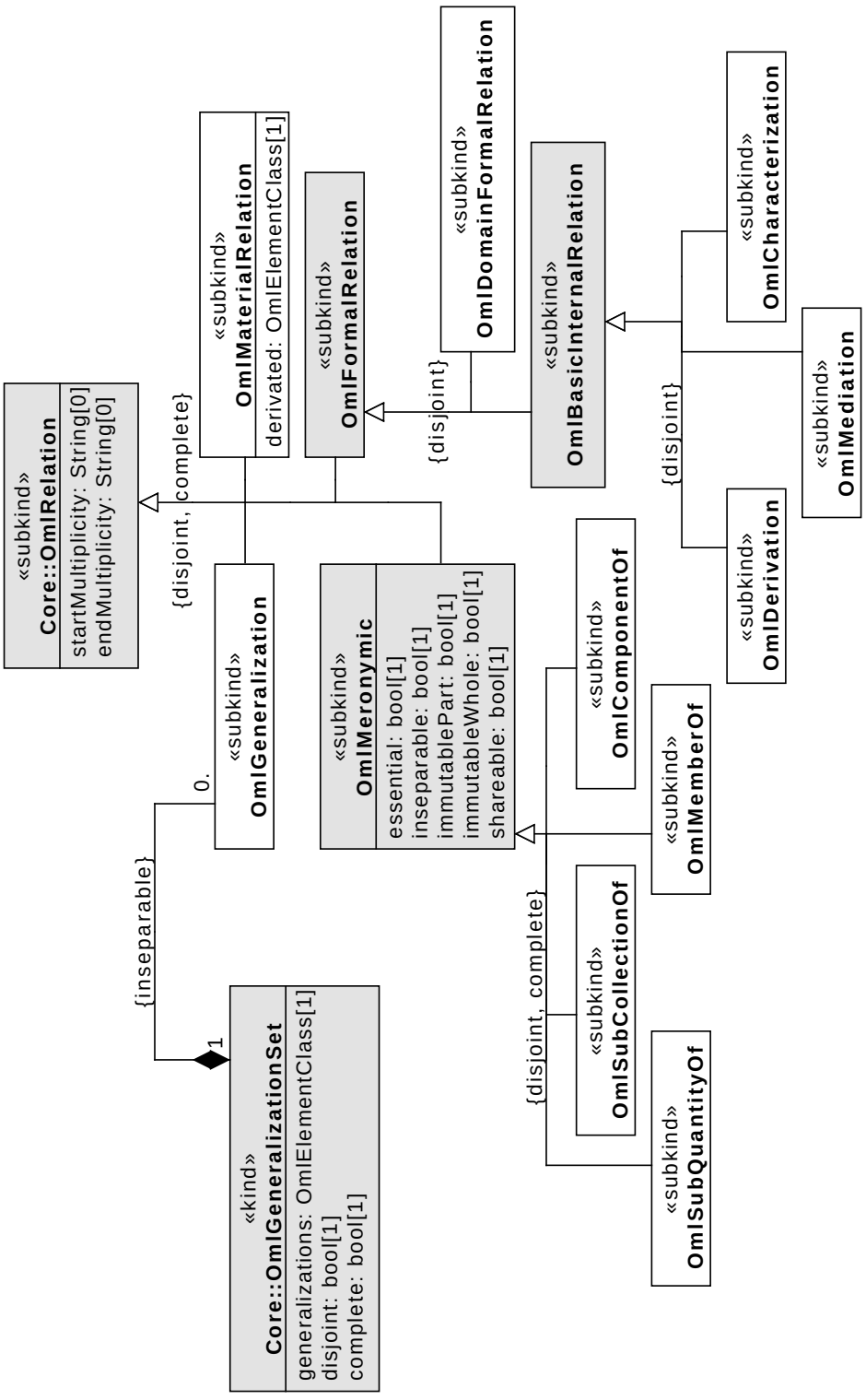
Stále najaktuálnejšiu verziu OntoUML pluginu je možné získať priamo z repozitáru [13] v zdrojovej forme. Plugin bol taktiež začlenený do tzv. all-in-one(alpha build) samotného DynaCASE. Pre plnú funkčnosť a vyskúšanie pluginu stačí stiahnuť priamo build zo stránok DynaCASE [12].



Obr. 2.3: Jádro implementácie metamodelu OntoUML



Obr. 2.4: Časť OntoUML metamodelu, implementácia entít



Obr. 2.5: Časť OntoUML metamodelu, implementácia relácií

Kapitola 3

Instance-Level Modelling

Konceptuálne modelovanie úzko súvisí s výslednou kvalitou softwarového produktu a správny konceptuálny model je stále veľkou výzvou aj pre odborníka v danej oblasti. Analytik musí správne formalizovať všetky koncepty modelovanej domény [14]. V tejto kapitole budeme navrhovať postup, ktorý má za úlohu uľahčiť process formalizovania domény v OntoUML, modelovaním priamo OntoUML inštancií a ich následním validovaním podľa doteraz definovaného konceptuálneho modelu. Tento postup pomôže odhaliť nepresnosti v konceptuálnom modeli včas (napr. predtým ako začne ďalšia fáza vývoja software).

Výsledný navrhnutý spôsob modelovania inštancii OntoUML je mixom z viacerých rôznych prístupov k Instance Level Modelling.

3.1 Kvalita modelu

Instance Level Modelling je časť konceptuálneho modelovania, ktorá pracuje s konkrétnimi objektami namiesto tried a typov. V mnohých ohľadoch je tento prístup k modelovaniu veľmi inovatívny, pretože umožnuje analytikovi iný pohľad na modelovanú doménu, umožňuje vidieť reálne data v kontexte daného modelu [15].

Faktory, ktoré vplývajú na kvalitu konceptuálneho modelu podľa [16] sú:

1. človek - analytik, ktorý vytvára model,
2. modelovaci jazyk - zvolený jazyk, ktorým je model popísaný,
3. modelovací proces - postup vytvárania modelu,
4. model - samotný vytvorený model.

Z tohto zoznamu je jasné, že modelovací proces je jedným z hlavných faktorov, ktoré vplývajú na kvalitu konceptuálneho modelu. „Najlepším riešením

ako môžu netechnickí ľudia, užívatelia rozprávať a opísať doménu je instance-level modelling [17]“. Aby sme následne vedeli, že je konceptualizácia správna, potrebujeme ju nejakým spôsobom validovať.

3.2 Alloy

Jedným z možných riešení ako validovať konceptuálny model je vygenerovať všetky inštancie a následne ich overiť. Na takýto postup môžeme použiť Alloy. Alloy je definovaný ako štrukturálny modelovací jazyk, ktorý vychádza z logiky poriadku pre vyjadrenie komplexných štrukturálnych obmedzení a správania [18]. Tento jazyk je podporovaný v nástroji Alloy Analyzer, ktorý je súčasťou OntoUML Lightweight Editor (OLED) [19], v ktorom je možné inštancie generovať, analyzovať, simulovať a validovať. Model v Alloy sa skladá z logických obmedzení a tieto odmedzenia su zachytené v signatúrach, deklaráciách a tzv. facts. Pri vytváraní inštancie z modelu v Alloy Analyzer základné prvky sú generované zo signatúr a dodržujú všetky logické obmedzenia modelu. Na obrázku 3.1 je ukážka základného modelu reprezentovanom jazykom Alloy. Na tomto príklade je vidieť, že inštancie Person sú generované zo signatúry Person. Každá signatúra generuje vlastnú množinu inštancií.

V Alloy neexistujú žiadne signatúry reprezentujúce vzťahy a relácie medzi objektami [14]. Namiesto toho každá signatúra môže obsahovať pole deklarácií, ktoré reprezentujú vzťahy medzi signatúrami (napr. v ukážkovom modeli 3.1 má signatúra Enrollment položku school a tá reprezentuje vzťah medzi Enrollment a Organization. V každej deklerácii je možné používať multiplicitu na vyjadrenie povinnosti danej relácie. Základné multiplicity sú:

- one - relácia je povinná a max. 1,
- lone - relácia je nepovinná a max. 1, (0..1),
- some - relácia je povinná a max. 1, (1..*),
- set - relácia je nepovinná a max. 1, (0..*).

V ukážkovom modeli 3.1 signatúra Enrollment obsahuje deklaráciu student s multiplicitou one, to znamená, že ku každej Enrollment musí byť priradený práve jeden Person cez student reláciu. Oproti klasickému OntoUML táto multiplicita platí iba v jednom smere. Nikde nie je definované, na koľko Enrollments môže byť naviazaný jeden Person. Facts sú obmedzenia, ktoré sú stále true.

Alloy podporuje aj dedičnosť alebo inak povedané podsignatúry. Mechanizmus vytvárania subsignatúr v Alloy je rovnaký ako špecializácie v konceptuálnom modelovaní. Podsignatúry dedia relácie a obmedzenia svojích nadsignatúr. Napríklad v ukážkovom modeli 3.1 signatúry Man a Woman sú podsignatúrami Person. Označenie podsignatúry je riešené kľúčovým slovom „in“.

```
01.open util/ordering[State] as state
02.open util/relation
03.sig Person{ }
04.sig Man, Woman in Person{ }
05.fact generalization_set{
06.   disj[Man,Woman]
07.   Person = Man+Woman
08.}
09.sig Organization{ }
10.sig Enrollment{
11.   school: one Organization,
12.   student: one Person,
13.   derived_material_relation: student one-> one school,
14.}
15.fun Agent:(Organization+Person){
16.   Organization+Person
17.}
18.sig State{
19.   exists: set univ,
20.   disj Adult, Child, Teenager: set Person:>exists,
21.   disj Deceased, Living: set Person:>exists,
22.   Student: set Person:>exists,
23.   School: set Organization:>exists,
24.   Insurable: set Living+Organization:>exists,
25.   study: set Student some -> some School,
26.}{
27.   exists in ultimate_sortal
28.   all x:exists| x not in this.next.@exists implies x not in this.^next.@exists
29.   Person:>exists= Adult+Child+Teenager
30.   Person:>exists= Deceased+Living
31.   all x:Enrollment:>exists | x.school in Organization:>exists
32.                              and  x.student in Person:>exists
33.   Student = (Enrollment:>exists).student
34.   (Enrollment:>exists).school in School
35.   all x: School | some Enrollment:>exists:>school.x
36.   Insurable = Organization:>exists+Living
37.   study in exists.derived_material_relation
38.   }
```

Obr. 3.1: Ukážka alloy modelu [14]

Všetky signatúry, ktoré nemajú žiadne nadsignatúry sa nazývajú top-level signatúry. Každá inštancia vygenerovaná v Alloy Analyzer patrí k práve jednej top-level signatúre, ale môže patriť k viac subsignatúram.

3.3 Fact-Oriented Modeling

Fact-Oriented Modeling je konceptuálny prístup modelovania, dopytu a transformácie dát, kde všetky „facts“ a pravidlá môžu byť vyjadrené v jazyku, ktorému budú rozumieť aj netechnicky orientovaní ľudia chápajúci danej biznis doméne. Oproti entitne orientovanému modelovaniu (ER modelovanie), objektovo orientovanému modelovaniu (UML), relačnej databáze a značkovacím jazykom (XML) je tento druh modelovania založený na reláciach (unárné, binárne...) a je bezatribútový.

Združovanie informácií do atribútov je už viac implementačne založený postup, ktorý je pre biznis zákazníka nepodstatný a ťažko pochopiteľný pre menej technicky zameraného človeka. Napríklad namiesto atribútov `Person.isSmoker` a `Person.birthCountry` použijeme relácie `Person smokes` alebo `Person was born in Country` [20].

Ďalšou nespornou výhodou tohto prístupu je jeho sémentická stabilita. Napríklad v našom príklade, ak používame atribút `birthCountry` a neskôr sa rozhodneme sledovať ešte veľkosť populácie krajín, musíme premodelovať danú informáciu na reláciu a zmeniť všetky dopyty založené na tomto atribúte [20].

Stále najpopulárnejší fact-oriented prístup je ORM (Object-Role modelling). Toto meno dostal, pretože vyobrazuje svet z pohľadu objektov (entity alebo hodnoty), ktoré reprezentujú určité role (vzťahy) [15].

3.4 Object Constraint Language

Object Constraint Language (OCL) je deklarativný jazyk používaný pre popis pravidiel a obmedzení aplikovateľných na UML modely, ktoré nemôžu byť zachytené priamo v UML notaci. OCL je súčasť UML špecifikácie. OCL výrazy sú zložené zo 4 častí:

1. kontext definujúci nejakú situáciu (časť reality), v ktorej výraz platí,
2. vlastnosť, ktorá reprezentuje nejakú charakteristiku daného kontextu (napr. ak je kontext trieda, tak bude vlastnosť nejaký atribút),
3. operácia, ktorá manipuluje s vlastnousťou (napr. aritmetická operácia, xor atď.),
4. kĺúčové slová v podmienkach (napr. if, else, and, or, atď.).

Jednoduchý OCL výraz vyzerá takto [21]:

```
context TypeName inv:
'this is an OCL expression with stereotype <<invariant>>
in the context of TypeName' = 'another string'
```

Trochu konkrétnejší príklad už so špecifickým kontextom definovaným na začiatku výrazu [21]:

```
context c : Company inv:
  c.numberOfEmployees > 50
```

3.5 Modelovanie inštancií OntoUML

Vo vytvorenom návrhu modelovania inštancií OntoUML bolo, okrem teoretických znalostí, vychádzané z požiadaviek, ktoré boli definované zo strany CCMi (Centrum pre Konceptuálne Modelovanie a Implementace) hlavne od Róberta Pergla, ale tiež z vlastných skúseností pri práci na rôznych konceptuálnych analýzach pre business partnerov centra. Hlavným prípadom použitia (use case), z ktorého návrh vychádza bol:

- analytik sedí so zákazníkom, ktorý potrebuje kompletnú konceptuálnu analýzu spoločnosti. Pri tomto stretnutí zákazník popisuje štruktúry, role, oddelenia a celkovú podobu spoločnosti. Analytik z týchto informácií dokáže priamo na mieste vytvárať inštančný model pomocou nástroja vytváraného v tejto práci.

Tento inštančný model ešte nijako nie je viazaný na konkrétny OntoUML model, ale už zobrazuje skutočnosť, ako to v tejto danej spoločnosti funguje. Z tohto diagramu dokáže analytik rýchlo dostať predstavu o fungovaní spoločnosti, nájsť prípadné nezrovnalosti a hneď ich spolu so zákazníkom priebežne opravovať bez toho, aby ešte čo i len začal vznikať konkrétny OntoUML diagram.

Takýto prístup pomáha znížiť počet schôdzok potrebných k presnému definovaniu zadania a tiež eliminuje prípadné ťažko opraviteľné chyby v diagramoch (napr. je jednoduché opraviť chybu v nejakej inštancii ako rovnakú chybu upravovať v OntoUML, BPMN alebo inom diagrame, ktorý vznikol na základe rovnakej zlej definície).

Samozrejme je možné postupovať aj opačným smerom. Najprv môžeme vytvoriť OntoUML model a k tomuto modelu vytvoriť inštančný model. Tento smer by však mal byť menej častý, pretože prirodzene zákazník nám vie presne opísať inštancie a až potom ich analytik vie presne zachytiť z konceptuálneho hľadiska. Dôležitým prvkom takéhoto previazania diagramov je validácia. OntoUML má pevne dané syntaktické pravidlá, ktoré musí spĺňať každý model. Problém je však pri inštančnej validácii modelu. Táto validácia vychádza

priamo z toho, ako je vytvorený naviazaný OntoUML model. Podrobne je táto validácia opísaná v sekcii 3.5.1.

3.5.1 Validácia inštancií

Validácia inštancií je v celku veľmi zložitá vec. V tejto práci sa budeme venovať validácii inštancií na základe pripojeného OntoUML diagramu. Implementované validačné pravidlá pre inštanciu a podinštanciu:

- inštancia musí byť pripojená k nejakému OntoUML elementu,
- meno inštancie sa musí zhodovať z menom pripojeného OntoUML elementu,
- inštancia musí obsahovať všetky povinné atribúty,
- každá property v inštancii musí byť v zhode s maximálnou a minimálnou multiplicitou pripojeného atribútu,
- inštancia musí obsahovať všetky povinné asociácie,
- počet napojených inštancií cez rovnakú reláciu (napojené na rovnakú OntoUML asociáciu), musí byť v súlade s maximálnou a minimálnou multiplicitou danej OntoUML asociácie,
- inštancia musí obsahovať všetky podinštancie.

Pravidlá pre reláciu:

- relácia musí byť pripojená k nejakému OntoUML elementu,
- meno relácie sa musí zhodovať z menom pripojeného OntoUML elementu,
- začiatok a koniec relácie musí byť napojený na inštancie, ktoré boli vytvorené od OntoUML elementov, ktoré sú na začiatku a konci napojenej OntoUML asociácie.

3.5.2 Príklady diagramov

Celá táto sekcia je venovaná praktickým ukážkam toho, ako funguje popisovaná metóda modelovania inštancií OntoUML v praxi na diagramoch. Tieto ukážkové diagramy slúžia pre užívateľa aj ako predloha. Každý z diagramov zobrazuje modelovanie inštancií od určitých OntoUML elementov.

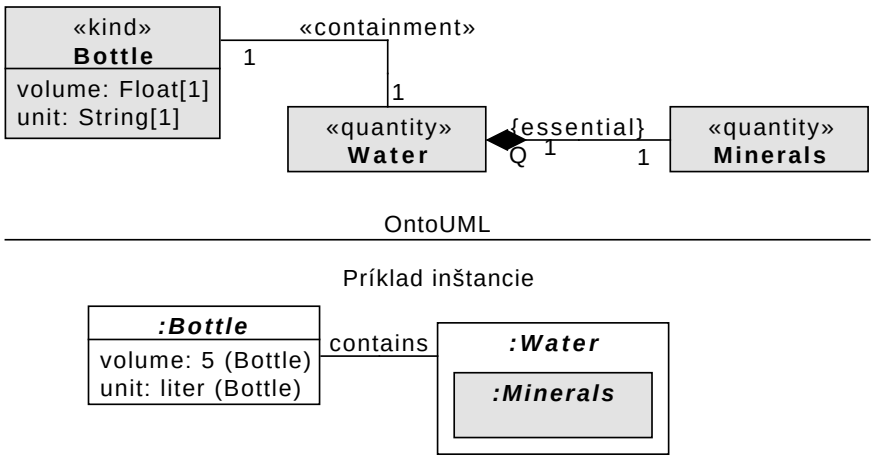
Zoznam OntoUML elementov a diagramov, na ktorých je ukážka vytvorenia ich inštancie:

- «kind» 3.2 3.4 3.5 3.6 3.7 3.8,

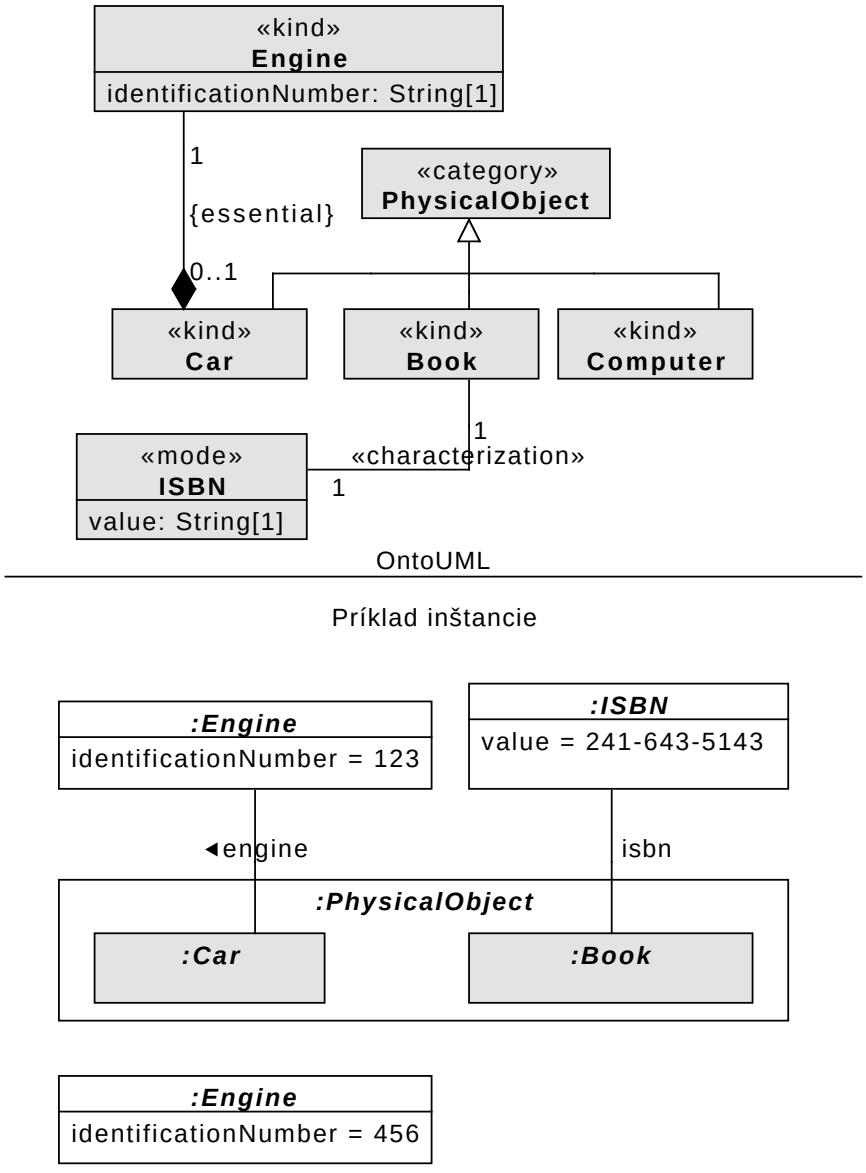
- «subkind» 3.7,
- «role» 3.4 3.6 3.7,
- «phase» 3.5 3.7,
- «collective» 3.6 3.8,
- «quantity» 3.2,
- «category» 3.3,
- «rolemixin» 3.4,
- «mixin» 3.5,
- «relator» 3.7,
- «mode» 3.3,
- «quality» 3.5.

Zoznam OntoUML relácií a diagramov, na ktorých je ukážka vytvorenia ich inštancie:

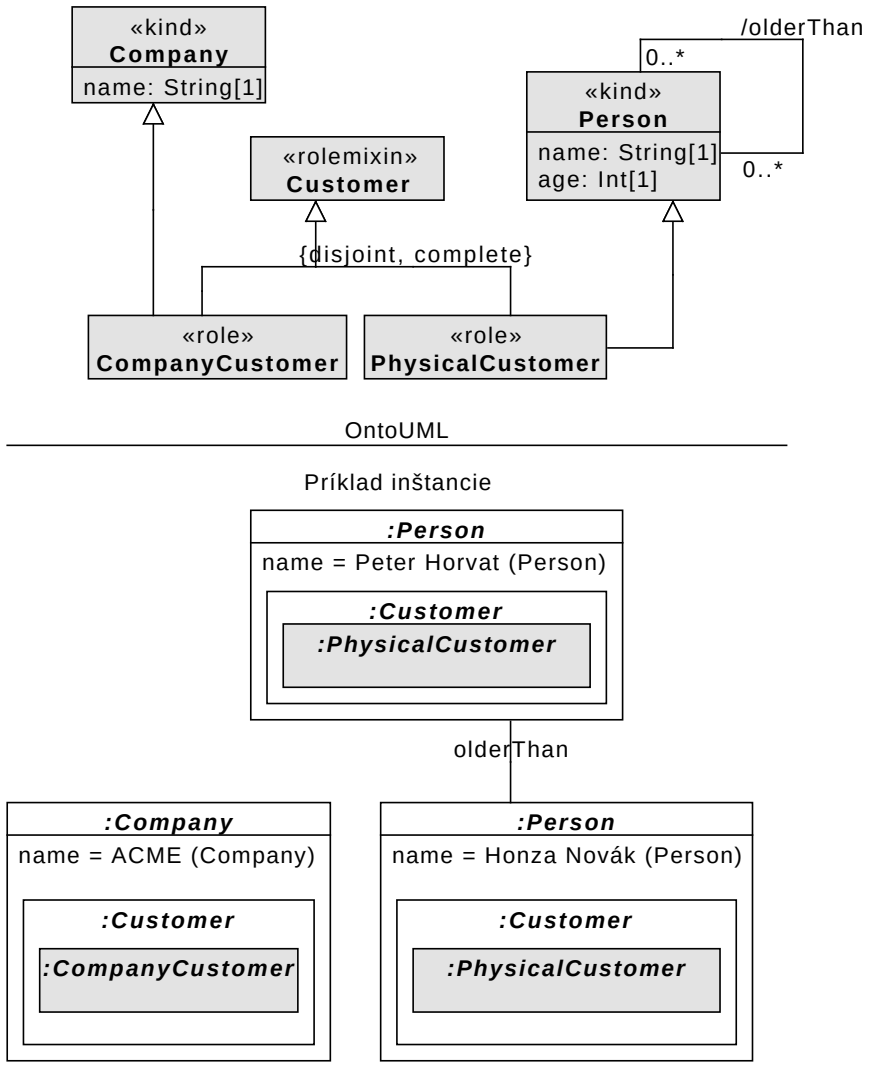
- «mediation» 3.7,
- «characterization» 3.3 3.5,
- «material» 3.7,
- «formal» 3.2 3.4,
- «componentOf» 3.3,
- «memberOf» 3.6 3.8,
- «subCollectionOf» 3.8,
- «subQuantityOf» 3.2,
- «derivation» 3.7.



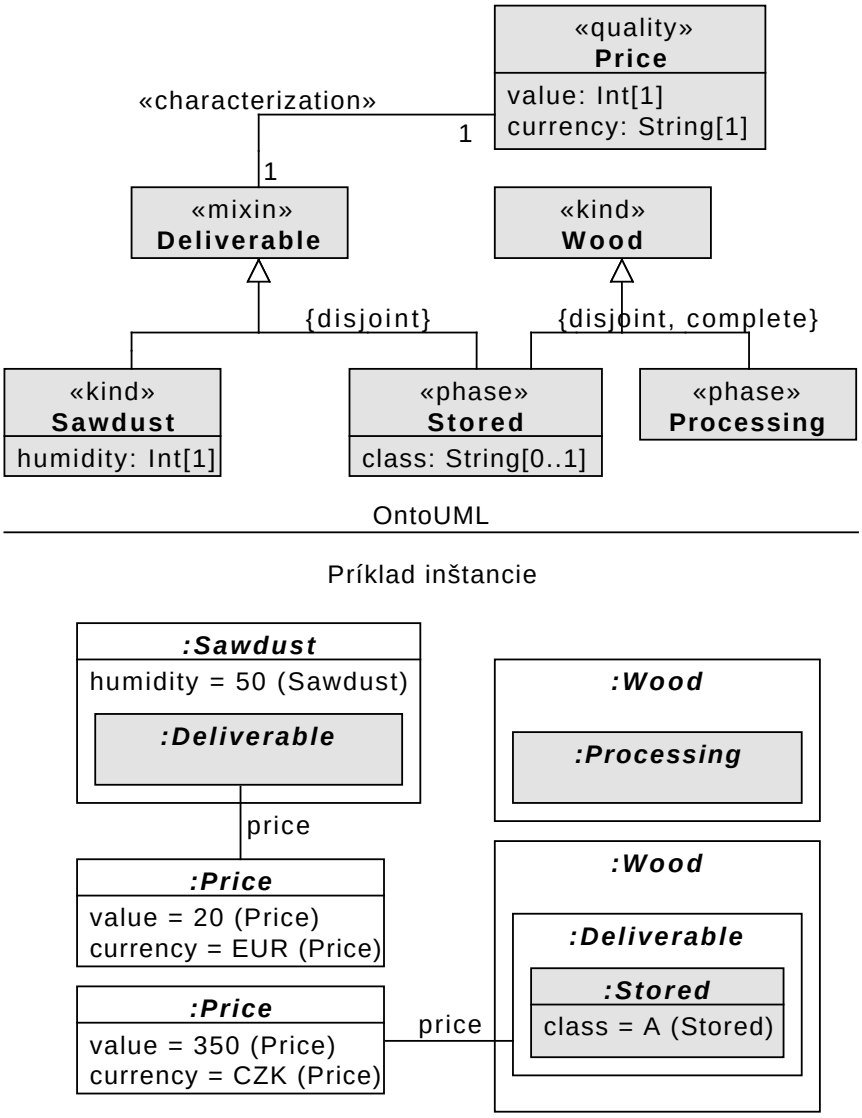
Obr. 3.2: Ukážka inštančného modelovania quantity



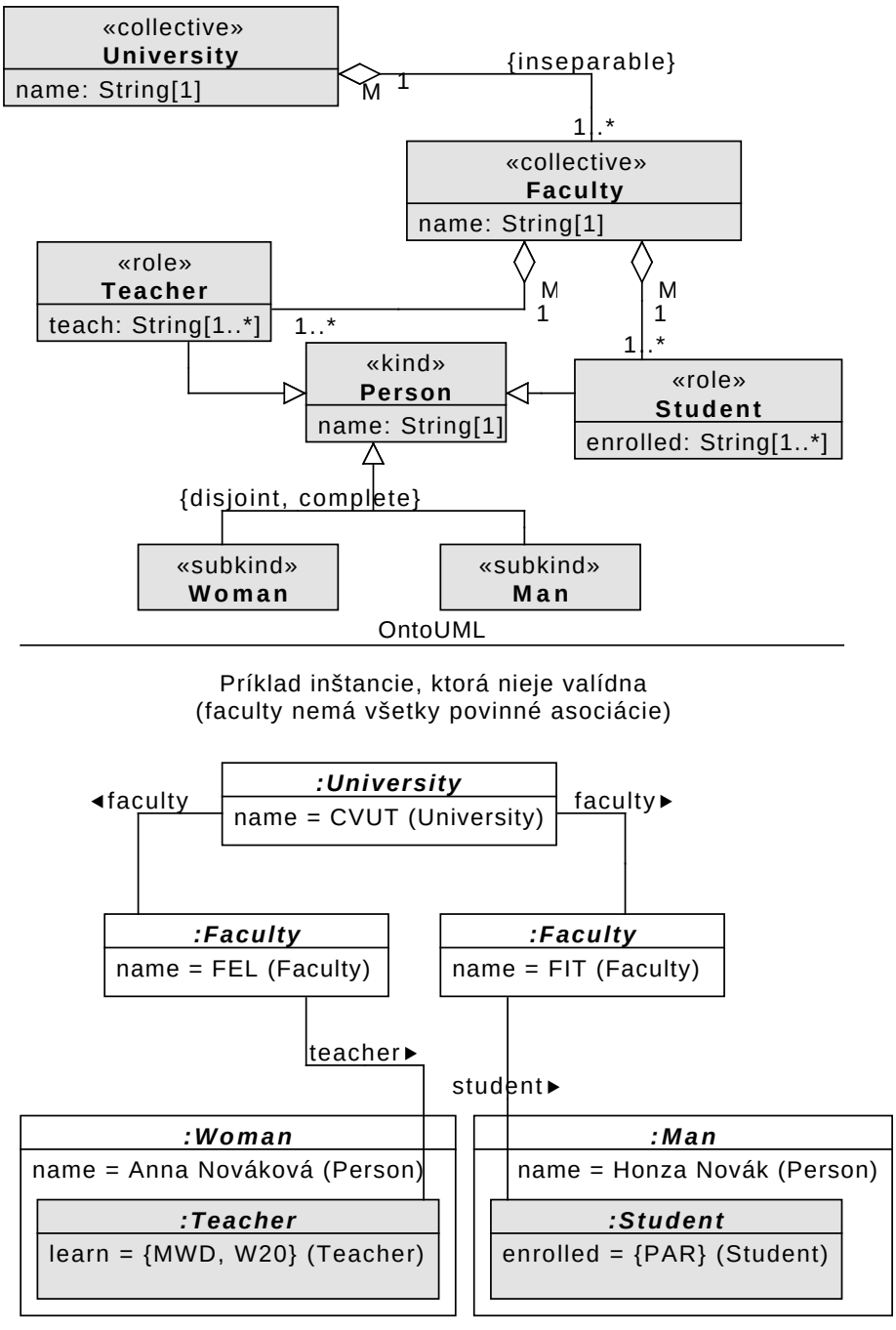
Obr. 3.3: Ukážka inštančného modelovania category



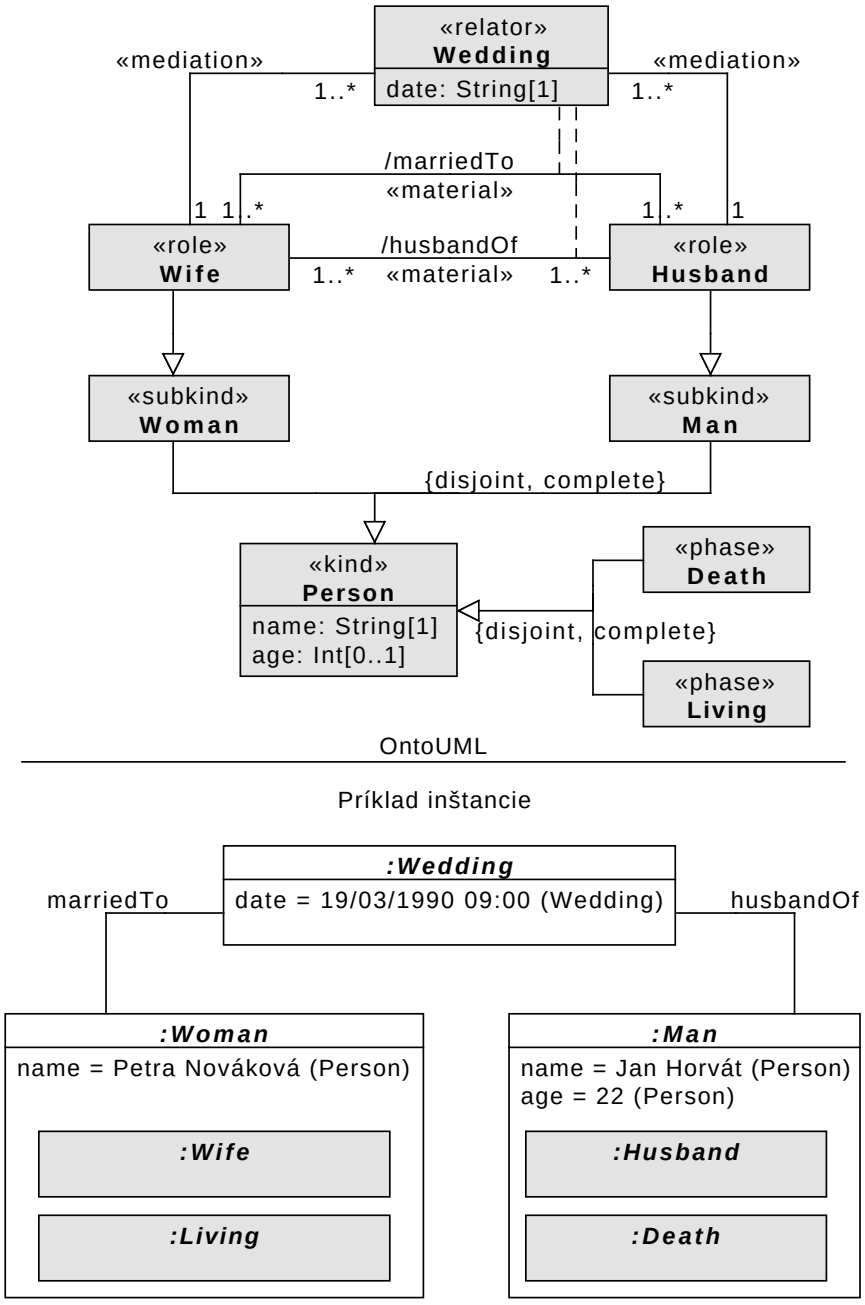
Obr. 3.4: Ukážka inštančného modelovania rolemixin



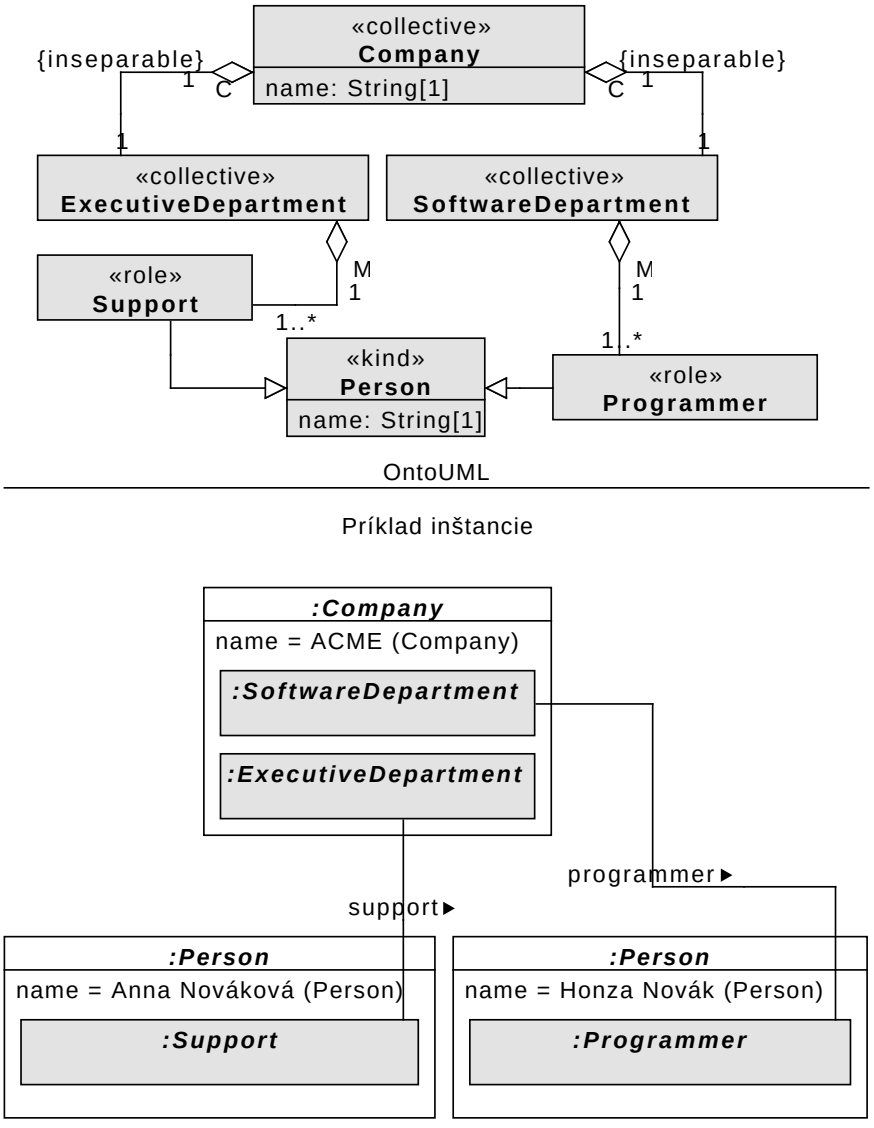
Obr. 3.5: Ukážka inštančného modelovania mixinu a quality



Obr. 3.6: Ukážka inštančného modelovania collective



Obr. 3.7: Ukážka inštančného modelovania relatoru



Obr. 3.8: Ukážka inštančného modelovania relácie SubCollectionOf

Kapitola 4

Komponenta OntoUML inštancií pre DynaCASE

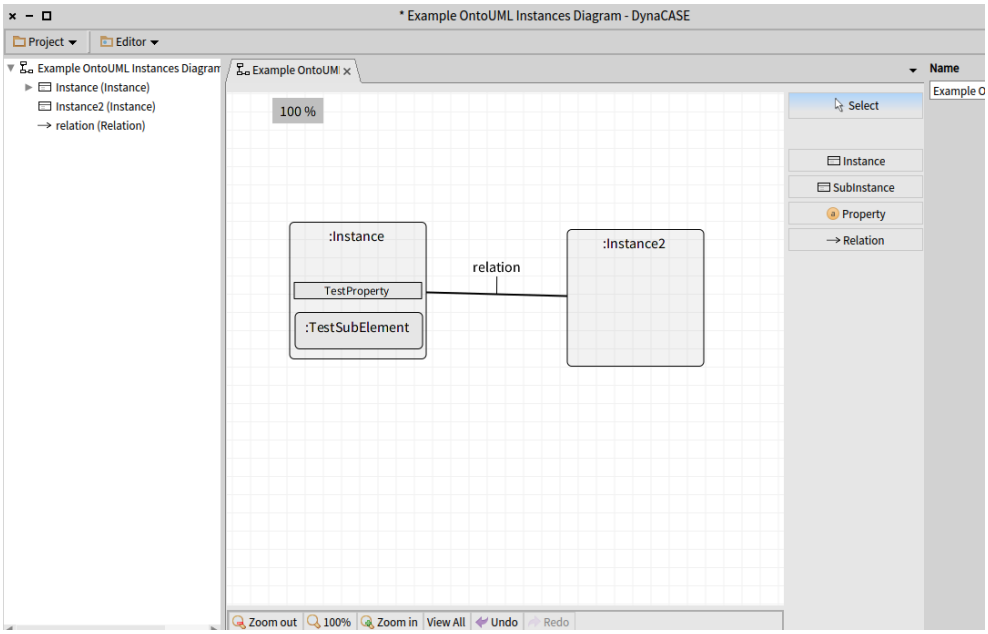
V tejto kapitole vytvorýme novú DynaCASE komponentu pre prácu s inštančným OntoUML diagramom. Táto komponenta bude okrem vytvárania a zobrazovania diagramu vykonávať inštančné validácie a spúšťať integritné obmedzenia. Predstavený bude detailne návrh metamodelu, jeho použitie a fungovanie. Na konci kapitoly, podobne ako v kapitole o OntoUML komponente, popíšeme jednotlivé controllers, ktoré ovládajú prvky na DynaCASE canvase.

4.1 Metamodel

Každá nová DynaCASE komponenta potrebuje ako prvú časť svoj model. Metamodel inštancií OntoUML je narozdiel od metamodelu klasického OntoUML podstatne menej rozsiahly, pretože neobsahuje také množstvo rôznych elementov. Skladá sa so 6 tried:

1. OmlInstancesDiagram,
2. OmlInstancesObject,
3. OmlInstancesProperty,
4. OmlInstancesRelation,
5. OmlInstancesElement,
6. OmlInstancesSubElement.

Každá trieda metamodelu má prefix „OmlInstances“, pretože ako už bolo spomenuté, v prostredí Pharo sú všetky triedy globálne prístupné a ľahko sa môže stať pri viacerých všeobecných triedach, že nejakú nedopatrením prepíšeme.



Obr. 4.1: Inštančný diagram z ukážkového kódu v DynaCASE

Trieda `OmlInstancesDiagram` je hlavná trieda celého modelu. Má v sebe uložené všetky elementy typu `OmlInstancesObject` a `OmlInstancesRelation`. Spúšťa inštančnú validáciu diagramu a rovnako aj vyhodnocuje integritné obmedzenia. Aby všetko plne fungovalo je potrebné, aby v atribúte `omlDiagram` bol odkaz na OntoUML diagram.

Trieda `OmlInstancesObjekt` tvorí základ navrhnutej hierarchie elementov zobrazených na DynaCASE canvase. Pre všetkých svojich potomkov zabezpečuje napojenie na OntoUML element, prípravu reportu z validácie integritných obmedzení a kontrolu napojenia inštančného elementu na OntoUML element.

Triedy `OmlInstancesElement` a `OmlInstancesSubElement` reprezentujú OntoUML inštanciu, resp. podinštanciu. Obidve majú v základe rovnaké vlastnosti a používajú rovnaké metódy. Rozdiel je však hlavne v tom, že podinštancia nedokáže existovať bez inštancie a do canvasu sa nepridáva priamo, ale musíme ju pridať do inej inštancie podobne ako napr. property. Samotná podinštancia sa po pridaní už správa rovnako ako inštancia, taktiež môže obsahovať jej vlastné property, podinštancie a relácie.

Relácia medzi inštanciami je reprezentovaná ako `OmlInstancesRelation`. Jej funkcia a význam sú rovnaké ako v OntoUML diagrame. Vyjadruje vzťah medzi dvoma inštanciami napojenými na konce tejto relácie. Každá relácia musí mať odkaz na nejakú OntoUML asociáciu.

Poslednou triedou v inštančnom metamodeli je `OmlInstancesProperty`. Táto trieda reprezentuje atribúty OntoUML elementu. Stále je napojená na nejaký `FAMIXAttribute` a má nastavenú nejakú hodnotu.

4.1.1 Ukážka práce s metamodelom

Ukážka ako pracovať s inštančným metamodelom na jednoduchom príklade:

```
diagram := OmlInstancesDiagram new.
diagram
    add:
        (OmlInstancesElement new
            name: 'Instance';
            diagram: diagram).
diagram
    add:
        (OmlInstancesElement new
            name: 'Instance2';
            diagram: diagram).
diagram
    add:
        (OmlInstancesElement new
            name: 'Instance3';
            diagram: diagram).

OmlInstancesRelation new
    name: 'Relation';
    start: (diagram at: 'Instance');
    end: (diagram at: 'Instance2').

(diagram at: 'Instance')
    add:
        (OmlInstancesProperty new name: 'TestProperty').

project := DCProject new
   projectName: 'Example OntoUML Instances Diagram'.
project addModel: diagram.
DCWorkbench openProject: project
```

Po spustení tohto kódu sa zobrazí DynaCASE okno s diagramom podľa obrázku 4.1

4.2 Implementácia validácie

Dôležitou funkciou metamodelu je okrem reprezentácie diagramu v pamäti aj validácia. Metamodel inštancií OntoUML spomenutý v predošlej sekcii implementuje všetky pravidlá definované v sekcii 3.5.1. Validácia začína spustením metódy `validation` v triede `OmlInstancesDiagram`. Táto trieda overí napo-

jenie inštančného diagramu na OntoUML model a spustí validáciu všetkých vytvorených inštancií. Validácia inštancie je rozdelená do viacerých krokov:

1. Validácia názvu inštancie.
2. Overenie povinných atribútov - z OntoUML elementu sa zíkajú povinné atribúty a porovnajú sa s kolekciou properties.
3. Overenie počtu property od každého atribútu podľa multiplicity - spočítajú sa všetky property napojené na rovnaký `FAMIXAttribute` a tento počet sa porovná s max., resp. min. multipliciou.
4. Overenie povinných asociácií - z OntoUML elementu sa získajú povinné asociácie a porovnajú sa s kolekciou relácií danej inštancie.
5. Overenie počtu relácií od každej asociácie podľa multiplicít - spočítajú sa všetky relácie napojené na rovnakú OntoUML asociáciu a tento počet sa porovná s max., resp. min. multiplicitou na druhom konci danej asociácie.
6. Overenie, či má daná inštancia všetky podelementy - z OntoUML elementu sa získajú všetky `OmlGeneralizationSet`, ktoré nemajú rigídneho rodiča alebo nemajú rigídnych potomkov. Následne sa overí, či daná inštancia obsahuje podinštanciu, ktorá je napojená na nejakého potomka v danej množine nadtypov.

Ak inštancia poruší niektoré z týchto pravidiel, vytvorí sa chybová hláška, ktorá sa uloží do kolekcie a pokračuje sa na ďalšom kroku. Na konci sú všetky chyby vrátené a zobrazené. Žiaden z týchto krokov neprebehne, ak nie je daná inštancia napojená na nejaký OntoUML element.

Pre spustenie validácie je potrebné kliknúť na položku Editor v toolbare a následne na „Validate Diagram“, resp. „Evaluate Integrity Constraints“.

4.3 Diagram

Podobne ako pri OntoUML komponente aj kontrolery pre inštančný OntoUML model kopírujú svoje modelové triedy:

- OmlInstancesDiagramController,
- OmlInstancesElementController,
- OmlInstancesSubElementController,
- OmlInstancesPropertyController,
- OmlInstancesRelationController.

Inštančný element tvorí tzv. kontajner, do ktorého su vkladané ďalšie prvy a pri pohybe kontajneru sa hýbu aj elementy vložené v ňom. Implementácia vychádza z BORM komponenty v ktorej sa tiež používajú kontajnery pre elementy.

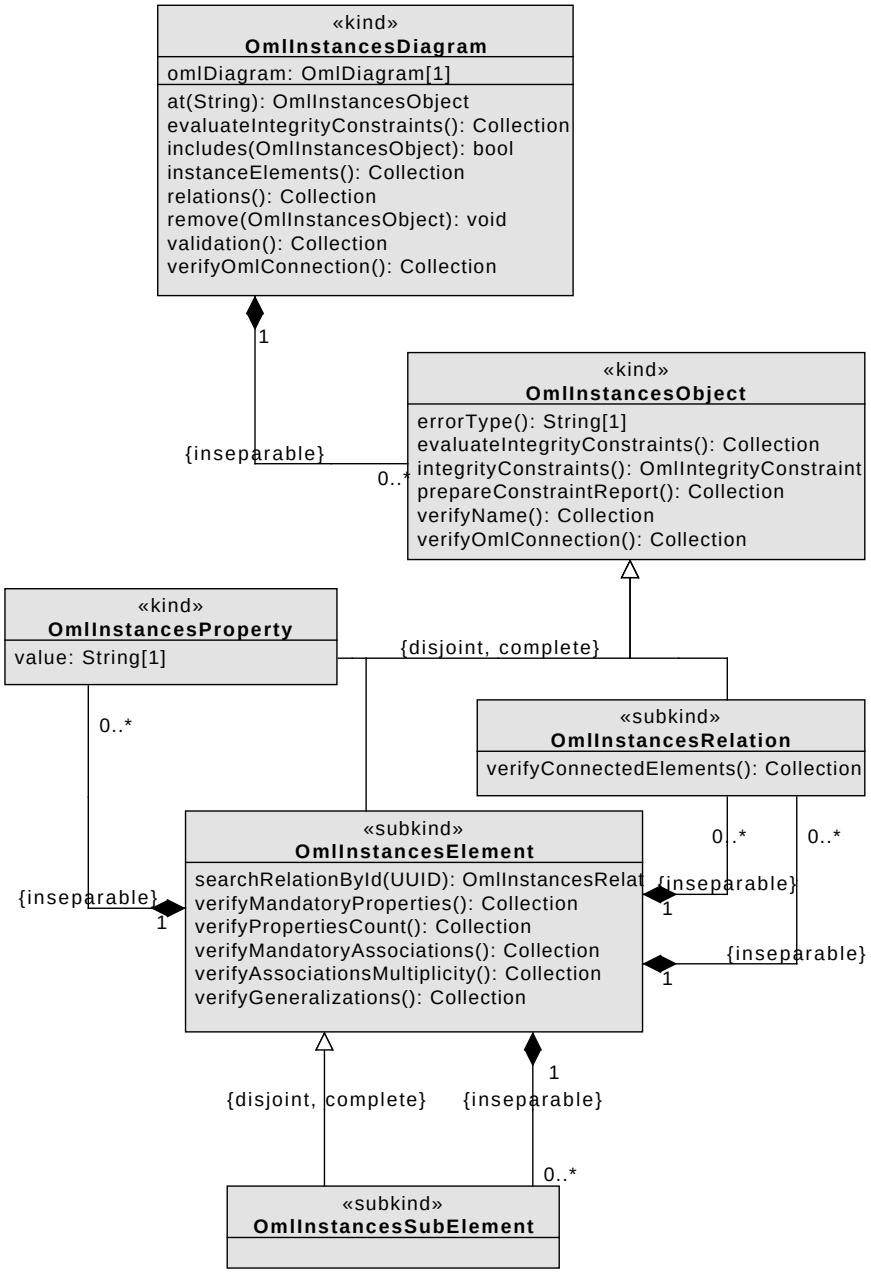
4.3.1 Layout

Podľa práce o layoutingu v DynaCASE [10] bol vytvorený jednoduchý layout pre inštančný diagram. Nachádza sa v triede `OmlInstancesBasicLayout`.

Layout je založený na triedach `RTGridLayout` a `RTVerticalLineLayout`. Prvá z menovaných tried rozloží inštancie po DynaCASE canvase do mriežky a druhá rozloží všetky prvky z inštančného kontajneru vertikálne pod seba. Následne sa ešte upravia automaticky rozmery každého elementu, podľa rozmerov prvkov ktoré obsahuje. Tento layout je využívaný pri otvorení diagramu priamo z ručne vytvoreného modelu 4.1.1 a taktiež sa bude využívať pri budúcom importe diagramu z nejakého textového formátu.

4.4 Repozitár

Stále najaktuálnejšiu verziu OntoUML Inštančného pluginu je možné získať priamo z repozitáru [13] v zdrojovej forme. Plugin bol taktiež začlenený, spoločne s OntoUML pluginom, do tzv. all-in-one(alpha build) samotného DynaCASE. Pre plnú funkčnosť a vyskúšanie pluginu stačí stiahnuť priamo build zo stránok DynaCASE [12].



Obr. 4.2: Metamodel inštančnej komponenty

Kapitola 5

Testovanie

Neoddeliteľnou a dôležitou časťou každého SDLC (software development life cycle) je testovacia fáza, či už v malom alebo vo veľkom. Hlavným cieľom testovacej fáze je overiť, či vytvorená aplikácia funguje správne ako celok a pomáha určiť kvalitu výsledného produktu. Pri testovaní v malom sa používajú tzv. unit testy. Testovanie v malom na unit testoch je užitočné hlavne pri iteratívnej metóde vývoja. Po každej zmene môžeme spustit unit testy a určiť správnoť dokončenej iterácie [22]. Správne vytvorené unit testy dokážu ušetriť veľa času pri hľadaní a opravovaní chýb. Vstupom do testovania vo veľkom je skompilovaná a zabalená aplikácia, ktorá je pripravená na testovanie. Dôležité je počas testovania vo veľkom pozastaviť vývoj a iba testovať na komplexnom príklade [23]. Celkovo bolo vytvorených 94 unit testov. Rozdelenie unit testov:

- OntoUML plugin - 66 testov, z toho 9 testov pre controllery a 57 testov metamodelu,
- OntoUML inštančný plugin - 28 unit testov, všetky testujú inštančný. metamodel.

Vytvorený balíček `DynaCASE-OntoUML` je pokrytý celkovo zo 63% a balíček `DynaCASE-OntoUML-Instances` z 32%. Pokrytie nie je až také veľké a to hlavne z dôvodu množstva jednoduchých gettrov a settrov a taktiež balíčky obsahujú triedy na definovanie nového pluginu v DynaCASE, ktoré nie je potrebné testovať (`OmlInstancesPlugin` a `DCOmlPlugin`. Modely sú pokryté takmer celé.

Dôležitá poznámka: Framwork DynaCASE, rovnako ako aj všetky jeho pluginy, sú a budú vo vývoji aj po dokončení tejto práce, a preto sa počty unit testov, rovnako ako aj pokrytie vytvorených balíčkov testami môže meniť.

5.1 Komponenta vytvárania OntoUML modelu

5.1.1 Unit testy

Pôvodná prvá verzia OntoUML metamodelu, ktorej implementácia vznikla pred dvoma rokmi v práci [1], bola vyvinutá metódou Test-driven development, t.j. testy boli napísané skôr ako samotný metamodel a vývoj prebiehal tak, aby testy prešli. V tejto druhej verzii už táto metóda použítá nebola. Prevzali sa niektoré koncepty zo starého modelu, ktoré sa upravili a následne sa vytvorili unit testy, ktoré sú rozdelené do kategórií:

- OmlDiagramTest - táto trieda obsahuje testy jadra metamodelu. Odoberanie a prístup k elementom, atribútom a metódam, výber asociácií, množín nadtypov, atď.
- OmlElementClassTest - táto trieda obsahuje testy OntoUML pravidiel, ktoré sa viažu k elementom, výber povinných asociácií, atribútov a množín nadtypov.
- OmlElementTest - táto trieda obsahuje všeobecné testy, ktoré sa viažu spoločne k asociáciám aj reláciám.
- OmlGeneralizationSetTest - táto trieda obsahuje testy pre množinu nadtypov.
- OmlRelationTest - táto trieda obsahuje testy OntoUML pravidial, ktoré sa viažu k asociáciám a dedeniu. Taktiež tiež obsahuje testy na overenie fungovania niektorých inverzných väzieb.
- DCOmlControllerTest - testy OntoUML controllerov.

5.1.2 Komplexný príklad

Ako komplexný príklad bol použitý veľký OntoUML diagram, ktorý zobrazuje initialize phase vývoja softwarového produktu. Ide o prevod BORM a BPMN procesných diagramov do OntoUML. Diagram obsahuje 48 tried, dedenie a rôzne druhy asociácií. Diagram je na obrázku 5.1 a všetky jeho role sú na obrázku 5.2. V tomto diagrame bola použitá tzv. modularizácia diagramu. Táto metóda je použítá pre lepšiu prehľadnosť a čitateľnosť základného diagramu. Modularizáciu popísal Dr. Giancarlo Guizzardi v jednej zo svojích prác [24] .Tento diagram bol vytvorený v rámci jedného projektu pre CCMi (Centrum pre Konceptuálne Modelovanie a Implementácie). Informácie o tejto časti SDLC boli čerpané z knihy [22] a článku [25].

5.2 Komponenta vizualizácie inštancií OntoUML

5.2.1 Unit testy

Pre inštančný metamodel bolo vytvorených celkovo 28 unit testov, ktoré boli rozdelené do kategórií kopírujúce modelové triedy:

- OmlInstancesDiagramTest,
- OmlInstancesElementTest,
- OmlInstancesObjectTest,
- OmlInstancesPropertyTest,
- OmlInstancesRelationTest.

5.2.2 Komplexný príklad

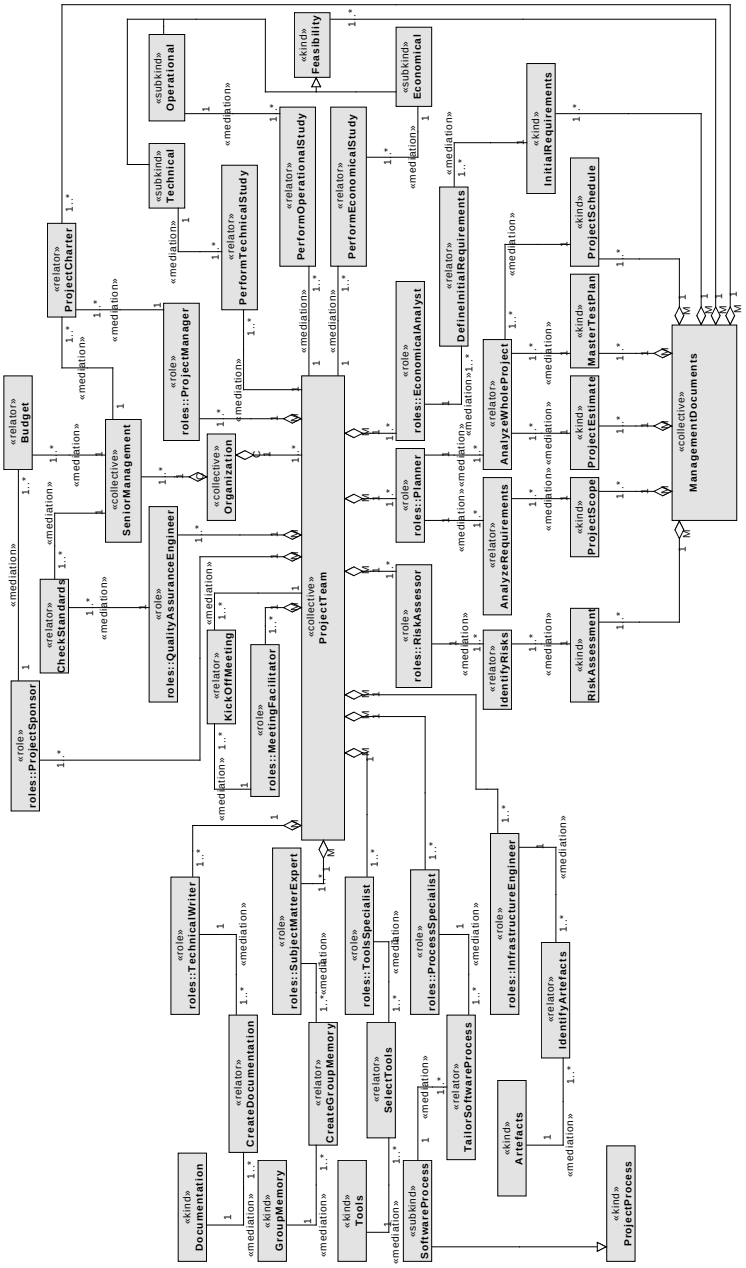
Komplexný príklad inštančného diagramu bol vytvorený z diagramov na obrázkoch 5.1 a 5.2. Pre jeho veľkosť a lepší prehľad bol rozdelený do dvoch častí 5.3, resp. 5.4. Na tomto príklade vidíme, ako môže analytik pri schôdzke s klientom namodelovať inštančnú podobu initiate phase v nejakej spoločnosti. Diagram samozrejme môže obsahovať množstvo vlastností v každom elemente, avšak v tomto prípade bolo pre nedostatok miesta zaznamenaných iba zopár, napr. name pri inštancii Person.

5.3 Implementované ukážky modelov

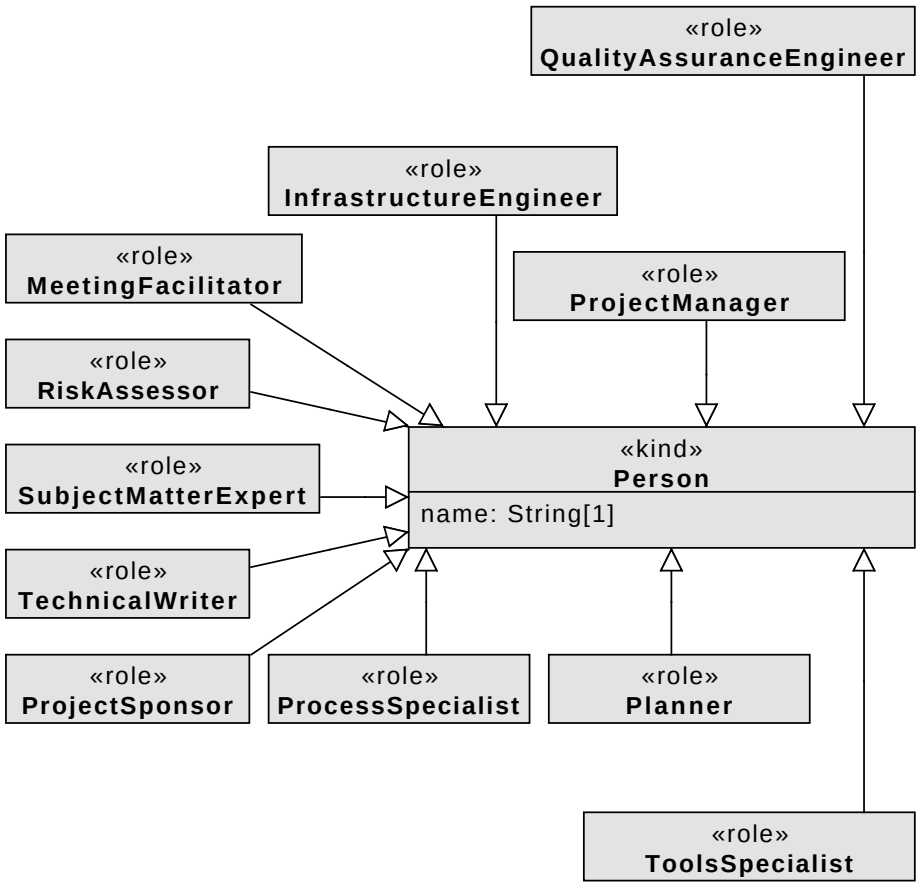
Priamo vo Pharo boli ručne implementované viaceré ukážkové diagramy, aby si každý užívateľ mohol urobiť obraz o tom, ako tieto pluginy fungujú a čo dokážu. Ukážky boli typovo vybrané tak, aby reflektovali niektoré známe problémy prevodu a taktiež, aby bolo zachytených čo najviac rôznych elementov.

Všetky implementované modely pre OntoUML plugin sa nachádzajú v triede `OmlExamples` resp. `OmlBigExample` v triednej časti. Modely pre OntoUML inštančný plugin sa nachádzajú v triede `OmlInstancesExample` taktiež v triednej časti.

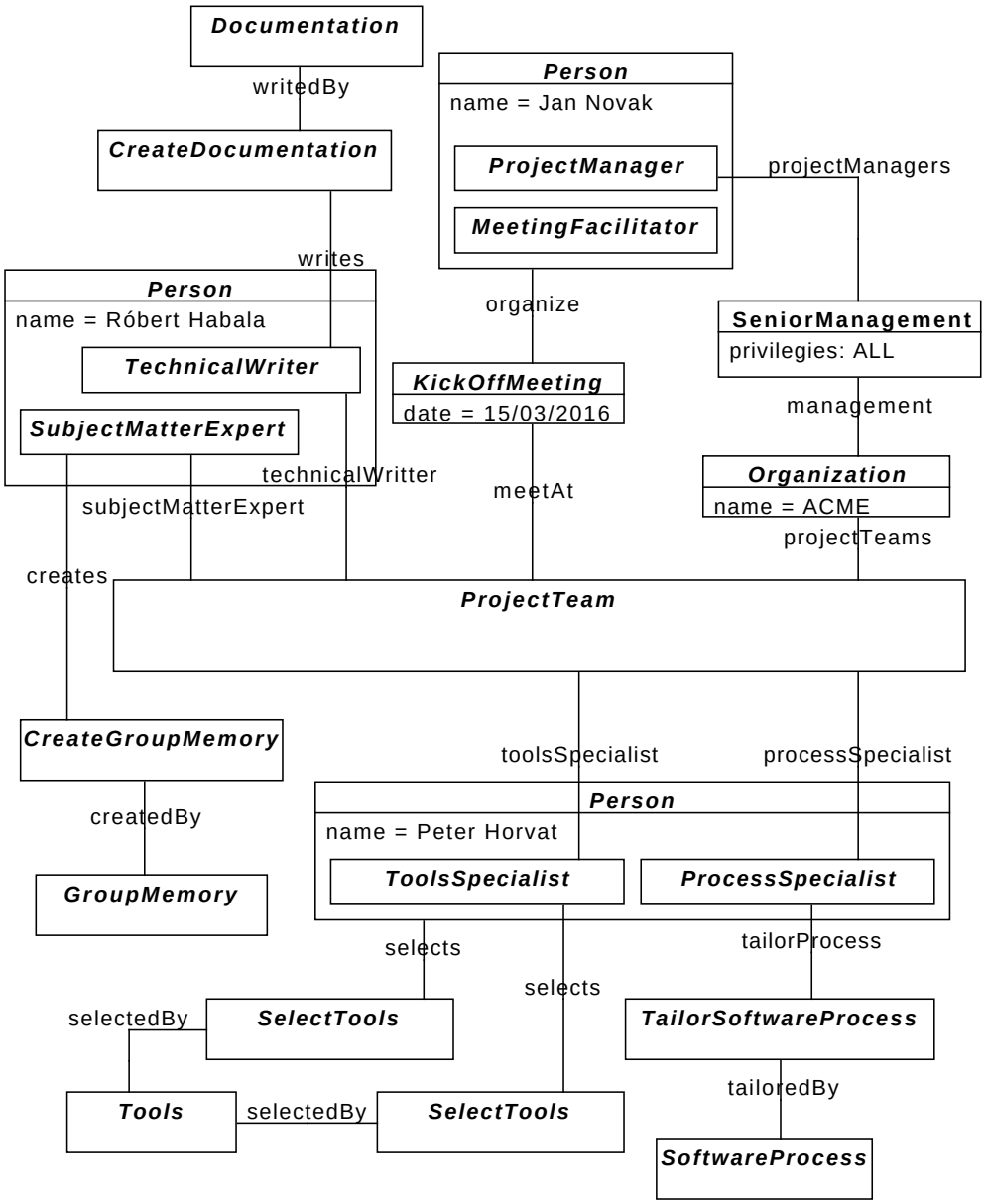
Ukážky sa spúšťajú priamo z Pharo „World Menu“ z časti OntoUML Editor resp. OntoUML Instances Editor.



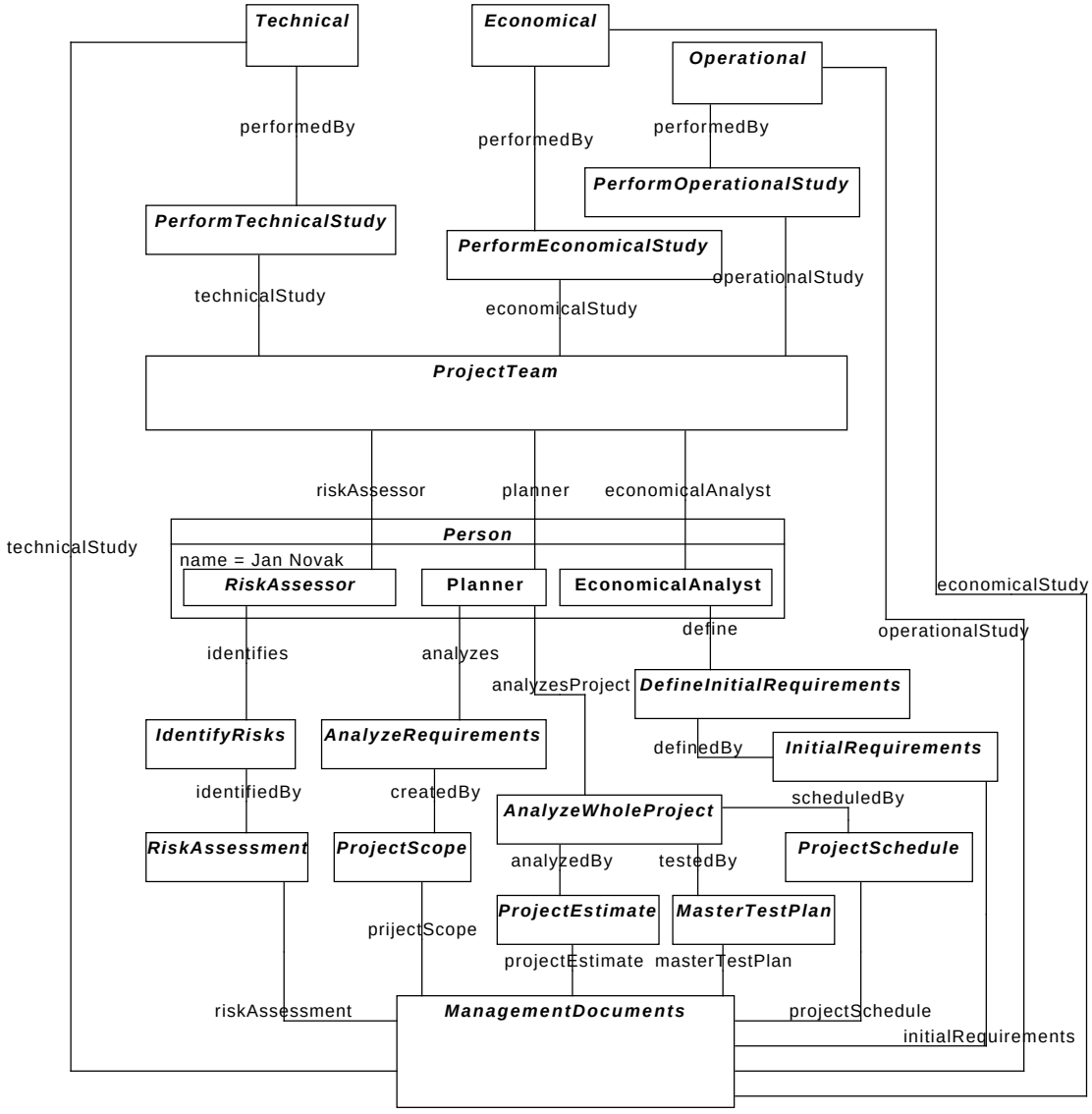
Obr. 5.1: Komplexný diagram použitý pri testovaní vo veľkom



Obr. 5.2: Role používané v diagrame 5.1



Obr. 5.3: Inštančný diagram z 5.1 a 5.2, časť 1



Obr. 5.4: Inštančný diagram z 5.1 a 5.2, časť 2

Záver

Hlavným cieľom tejto práve bolo navrhnúť, implementovať a otestovať nové pluginy do frameworku DynaCASE s názvom OntoUML Editor a OntoUML Instances Editor, pomocou ktorých bude mať užívateľ možnosť vytvárať a validovať OntoUML modely a im pridružené inštančné modely v jednom nástroji. Tieto pluginy by mali byť open source rovnako ako framework DynaCASE, ktorý je vyvíjaný na tejto fakulte.

Prvá časť rešerše práce bola zameraná ako akási dokumentácia OntoUML napísaná pre študenta zrozumiteľným spôsobom, pretože jej cieľom bolo nielen zoznámiť čitateľa s týmto ontologickým prístupom k UML, aby dokázal používať vytvorený plugin, ale aj zaujať študentov, ak majú OntoUML v rámci nejakého predmetu, napr. Objektové, resp. Konceptuálne modelovanie a poskytnúť im materiál k OntoUML v inom ako anglickom jazyku. V druhej časti rešerše sú popísané rôzne spôsoby inštančného modelovania, ich príklady a použitie, napr. Alloy alebo OCL. Následne je podrobne popísaný priamo implementovaný spôsob modelovania OntoUML inštancií aj s viacerými príkladmi.

Najdôležitejšou častou celej práce je samotná implementácia daných pluginov. Výber programovacieho jazyka a platformy bol daný už zo zadania, pretože na vývoj v jazyku Smalltalk a virtuálnom stroji Pharo sa vedecko-výskumná skupina CCMi, v rámci ktorej bola táto práca vytvorená, zameriava a priamo spolupracuje so skupinou vyvíjajúcou Pharo.

Práca bola taktiež dôležitá ako spätná väzba pre hlavného architekta a vývojára frameworku DynaCASE Petra Uhnáka, pretože som s ním mohol počas vývoja priamo komunikovať o tom, čo mi pri vývoji v DynaCASE veľmi pomáha a je dobre impelentované, čo mi chýba a čo naopak by som spravil inak. Našiel som pár bugov v DynaCASE Class Editore, a taktiež boli zapracované niektoré moje požiadavky, napr. automatická obnova formulára po zmene položky.

Ďalší rozvoj

Ako už bolo spomenuté vývoj DynaCASE sa posúva každým mesiacom dopredu a vytvorené pluginy je potrebné udržiavať aktulizované. Samotný DynaCASE je stále ešte len v alfa verzii podobne, ako všetky jeho pluginy. Samozrejme, ani vývoj pluginov dokončením tejto práce nekončí. Ďalším krokom bude perzistancia, t.j. export OntoUML diagram a OntoUML inštančného diagramu do nejakého textového formátu, ako napr. JSON alebo XML. Ďalšou vecou je možnosť vytvorenia «derivation» relácie, pretože táto relácia je trochu špeciálna v tom, že jeden koniec je napojený na inú reláciu a DynaCASE zatiaľ nepodporuje napojenie relácie na reláciu.

Osobný prínos

Vďaka tejto práci som prenikol do témy, ktorá ma už dlhšiu dobu zaujímala a to inštančné modelovanie. Okrem toho som sa tešil na implementáciu novej notácie vo frameworku DynaCASE napísanom v čisto objektovom jazyku Smalltalk. Samozrejme narazil som aj na problémy vo Pharo, pretože DynaCASE funguje iba vo Pharo 5, ktorého stabilná verzia nebola do dokončenia tejto práce vydaná a vývoj prebiehal v jeho vývojovej verzii.

Literatúra

[1] Vološin, M.: *Transformace OntoUML do Smalltalku. Bakalárska práca*. Praha, Česká Republika, Máj 2014. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/volosmat_2014bach.pdf

[2] Ruy, F. B.; Reginato, C. C.; Santos, V. A.; aj.: *Conceptual Modeling: 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings*, kapitola Ontology Engineering by Combining Ontology Patterns. Cham: Springer International Publishing, 2015, ISBN 978-3-319-25264-3, s. 173–186. Dostupné z: http://www.inf.ufes.br/~gguizzardi/ER2015OntologyPatterns.pdf

[3] Guizzardi, G.: *Ontological Foundations for Structural Conceptual Models*. Dizertační práce, University of Twente, Enschede, The Netherlands, October 2005. Dostupné z: http://doc.utwente.nl/50826/1/thesis_Guizzardi.pdf

[4] Wagner, G.; Guizzardi, R. S.; Guizzardi, G.; aj.: *Towards Ontological Foundation for Conceptual Modeling: The Unified Foundational Ontology (UFO) Story*. Applied Ontology, Vol. 10, issues 3-4, IOS Press, 2015. Dostupné z: http://www.inf.ufes.br/~gguizzardi/UFO-Story.pdf

[5] Benevides, A. B.: *A Model-based Graphical Editor for Supporting the Creation, Verification and Validation of OntoUML Conceptual Models*. Diplomová práce, Federal University of Espírito Santo (UFES), Vitória, E.S., Brazil, February 2010. Dostupné z: http://code.google.com/p/ontouml/downloads/detail?name=MSc_thesis_Alessander_Botti_Benevides.pdf

[6] Lucas Bassetti Rodrigues da Fonseca: Ontology Project. 2015. Dostupné z: http://ontology.com.br/ontouml/spec/

[7] Guarino, N.; Guizzardi, G.: *Advanced Information Systems Engineering: 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, kapitola "We Need to Discuss the Relationship": Revisiting Relationships as Modeling Constructs. Cham: Springer International Publishing, 2015, ISBN 978-3-319-19069-3, s. 279–294. Dostupné z: http://www.inf.ufes.br/~gguizzardi/CAISE2015-CR.pdf

[8] Guizzardi, G.: *Ontological Foundations for Conceptual Part-Whole Relations: The Case of Collectives and Their Parts*. Advanced Information Systems Engineering: 23rd International Conference, CAiSE 2011, London, UK, June 20-24, 2011. Proceedings, 2011. Dostupné z: http://www.inf.ufes.br/~gguizzardi/CAISE2011-CR.pdf

[9] Pharo Community: The Pharo CollaborActive Book. 2011. Dostupné z: http://pharo.gemtalksystems.com/

[10] Uhnák, P.: *Layouting of Diagrams in the DynaCASE Tool. Bakalárska práca*. Praha, Česká Republika, Január 2016. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/63194/F8-BP-2016-Uhnak-Peter-thesis.pdf

[11] Sales, T. P.; Guizzardi, G.: *Ontological anti-patterns: empirically uncovered error-prone structures in ontology-driven conceptual models*, ročník 99. 2015, 72 - 104 s., selected Papers from the 33rd International Conference on Conceptual Modeling (ER 2014). Dostupné z: http://www.inf.ufes.br/~gguizzardi/DKE2015-ER2014.pdf

[12] Peter Uhnák: DynaCASE Project, GitHub. 2016. Dostupné z: http://dynacase.github.io/

[13] Matúš Vološin: OntoUML plugin a OntoUML inštančný plugin pre DynaCASE, GitHub repozitár. 2016. Dostupné z: https://github.com/dynacase/ontouml

[14] Braga, B. F. B.; Almeida, J. P. A.; Guizzardi, G.; aj.: *Transforming OntoUML into Alloy: Towards Conceptual Model Validation using a Lightweight Formal Method*. Innovations in Systems and Software Engineering, 2010. Dostupné z: http://nemo.inf.ufes.br/wp-content/papercite-data/pdf/transforming_ontouml_into_alloy__towards_conceptual_model_validation_using_a_lightweight_formal_method_2009.pdf

[15] Pergl, R.; Sales, T. P.; Rybola, Z.: *Enterprise and Organizational Modeling and Simulation: 9th International Workshop, EOMAS 2013, Held at CAiSE 2013, Valencia, Spain, June 17, 2013, Selected Papers*, kapitola Instance-Level Modelling and Simulation Revisited. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-41638-5,

s. 85–100, doi:10.1007/978-3-642-41638-5_6. Dostupné z: https://www.researchgate.net/profile/Tiago_Prince_Sales/publication/260548253_Instance-Level_Modelling_and_Simulation_Revisited/links/5466306b0cf2f5eb18016875.pdf?origin=publication_list

[16] Irit, H.; Pnina, S.: *Variations in Conceptual Modeling: Classification and Ontological Analysis*. Journal of the Association for Information Systems, 2006. Dostupné z: http://aisel.aisnet.org/jais/vol7/iss8/20

[17] Merunka, V.: *Instance-Level Modeling and Simulation Using Lambda-Calculus and Object-Oriented Environments*. Conference: Enterprise and Organizational Modeling and Simulation - 7th International Workshop, 2011, ISBN 978-3-642-24175-8. Dostupné z: https://www.researchgate.net/profile/Vojtech_Merunka/publication/220921349_Instance-Level_Modeling_and_Simulation_Using_Lambda-Calculus_and_Object-Oriented_Environments/links/570e56b008aee76b9dadead9.pdf

[18] Benevides, A. B.; Braga, B. F. B.; Guizzardi, G.; aj.: *Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures*. Journal of Universal Computer Science, Special Issue on Evolving Theories of Conceptual Modeling, 2010. Dostupné z: http://www.jucs.org/jucs_16_20/validating_modal_aspects_of/jucs_16_20_2904_2933_benevides.pdf

[19] Guerson, J.; Sales, T. P.; Guizzardi, G.; aj.: *OntoUML Lightweight Editor: A Model-Based Environment to Build, Evaluate and Implement Reference Ontologies*. 19th IEEE Enterprise Computing Conference (EDOC 2015), Demo Track, Adelaide, Australia, 2015. Dostupné z: http://www.inf.ufes.br/~gguizzardi/edoc2015_camera_ready.pdf

[20] Halpin, T.: *Conceptual Modelling in Information Systems Engineering*, kapitola Fact-Oriented Modeling: Past, Present and Future. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ISBN 978-3-540-72677-7, s. 19–38, doi:10.1007/978-3-540-72677-7_2. Dostupné z: http://dx.doi.org/10.1007/978-3-540-72677-7_2

[21] OMG - Object Management Group: Object Constraint Language. 2016. Dostupné z: http://www.omg.org/spec/OCL/2.4

[22] Ambler, S.: *Process Patterns: Building Large-Scale Systems Using Object Technology*. SIGS: Managing Object Technology, Cambridge University Press, 1998, ISBN 9780521645683. Dostupné z: https://books.google.cz/books?id=qJJk2yEeoZoC

[23] Ambler, S.: *More Process Patterns: Delivering Large-Scale Systems Using Object Technology.* SIGS: Managing Object Technology, Cambridge University Press, 1999, ISBN 9780521652629. Dostupné z: https://books.google.cz/books?id=o_GpsHi8JtQC

[24] Gonçalves, B.; Guizzardi, G.; Filho, J. G. P.: *An Electrocardiogram (ECG) Domain Ontology.* João Pessoa, Brazil: In Proceedings of the Second Brazilian Workshop on Ontologies and Metamodels for Software and Data Engineering, 22nd Brazilian Symposium on Databases (SBBD)/21st Brazilian Symposium on Software Engineering (SBES), 2007. Dostupné z: http://www.inf.ufes.br/~bgoncalves/contents/womsde07.pdf

[25] Falbo, R. A.; Bringuente, A.; Guizzardi, G.: *Using a Foundational Ontology for Reengineering a Software Process Ontology.* Journal of Information and Data Management, 2011. Dostupné z: http://www.inf.ufes.br/~gguizzardi/Using%20a%20Foundational%20Ontology%20for%20Reengineering%20a%20Software%20Process%20Ontology_cameraready%20%281%29.pdf

Dodatok **A**

Zoznam použitých skratiek

UFO Unified Foundational Ontology

NEMO Ontology and Conceptual Modeling Research Group

FSM Finite State Machine

BORM Business Objects Relation Modeling

ORD Object-Relation Diagrams

UML Unified Modeling Language

DynaCASE Dynamic Computer Aided Software Engineering Tool

CCMi Centrum pre Konceptuálne Modelovanie a Implementácie

ORM Object-Role modeling

XML Extensible Markup Language

ER Entity relationship

OCL Object Constraint Language

SDLC Software development life cycle

BPMN Business Process Model and Notation

JSON JavaScript Object Notation

DRY Don't repeat yourself

OLED OntoUML Lightweight Editor

Dodatok **B**

Obsah priloženého CD

- `bin` ..... adresár so spustiteľnou formou implementácie
  - `dynacase-linux` .. virtuálny stroj Pharo pre linux s DynaCASE image
- `src`
  - `image` ..... DynaCASE image obsahujúci všetky komponenty
  - `repository` ..... GitHub repozitár
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
- `text` ..... text práce
  - `DP_Volosin_Matus_2016.pdf` ..... text práce vo formáte PDF