UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Aplikácia OCL v UML profiloch

DIPLOMOVÁ PRÁCA

Rastislav Mlčoch

Študijný odbor: Informatika
Školiteľ: Ing. Miroslav Líška
Bratislava, 2006

Čestné prehlásenie:

Čestne prehlasujem, že som diplomovú prácu vypracoval samostatne pod odborným vedením školiteľa, len s použitím uvedenej literatúry.

Bratislava, máj 2006

……..

Rastislav Mlčoch

Ďakujem svojmu diplomovému vedúcemu Ing. Miroslavovi Líškovi za odborné vedenie, cenné rady a podporu pri písaní práce.
Ďakujem Mgr. Radovanovi Červenkovi za konzultáciu ohľadom OCL podmienok a UML metamodelu.
Ďakujem svojej rodine a priateľom za oporu počas písania tejto diplomovej práce.

Abstrakt

V tejto práci sa zameriavame na unifikáciu analytických modelov v organizácii. Využívame pritom možnosti rozšírenia jazyka UML prostredníctvom UML profilov. Vytvorili sme biznis profil a profil na modelovanie požiadaviek. Použitím týchto profilov prispejeme k tomu, že analytici budú vytvárať modely s uniformnou štruktúrou. Ďalej sa zameriavame na modelovanie biznis stratégie zákazníka. Pre entity definované v spomínaných profiloch, ktoré hrajú dôležitú úlohu v modelovaní stratégie, sme zadefinovali spresňujúce podmienky v jazyku OCL.

Kľúčové slová: UML, UML profily, OCL, unifikácia analytických modelov

OBSAH

Zoznam obrázkov

Zoznam tabuliek

1 Úvod

Narastajúca zložitosť informačných systémov kladie vysoké nároky na prácu softvérových analytikov, architektov a inžinierov. Od počiatočných záujmov o štruktúru a kvalitu zdrojového kódu, softvéroví inžinieri postupne zameriavajú svoju pozornosť na modelovacie aspekty vývoja softvérových systémov. Modely poskytujú abstrakcie systémov, ktoré umožňujú lepšie zvládnutie väčších a komplexnejších aplikácií jednoduchšími spôsobmi, nezávisle od technológií, ktoré ich implementujú.

Všeobecne platí, že čím neskôr v procese vývoja softvéru sa odhalí chyba v špecifikácii, resp. nesprávne pochopenie požiadaviek zákazníka, tým bude vyššia cena za odstránenie tejto chyby. Takisto sa predĺži aj čas na vývoj daného softvéru. Preto je dôležité, aby sa dbalo na kvalitu a presnosť počiatočných fáz vývoja.

UML, vizuálny modelovací jazyk, sa stal jedným z najviac používaných štandardov na špecifikáciu a dokumentáciu informačných systémov. Skutočnosť, že UML je dosť všeobecný jazyk, môže byť nevýhodou pri modelovaní niektorej konkrétnej domény, pre ktorú by boli vhodnejšie viac špecializované jazyky. Jazyk UML poskytuje určité mechanizmy umožňujúce rozšírenie a úpravu jeho syntaxe a sémantiky na prispôsobenie sa danej doméne. Tieto mechanizmy sú zoskupené v UML Profiloch.

Modely vytvorené v UML obvykle nie sú dostatočne presné. Na spresnenie syntaxe a sémantiky UML bol vytvorený štandardný jazyk OCL.

V tejto práci sa snažíme riešiť otázku unifikácie analytických modelov v organizácii.

V druhej kapitole si stanovíme ciele práce. V kapitole 3 sa venujeme súčasným trendom v modelovaní informačných systémov. Popíšeme si jazyk UML a procesy vývoja v analýze. V štvrtej kapitole poukážeme na existujúce problémy spojené s modelovaním. Piata kapitola obsahuje popis možností riešenia spomínaných problémov. Venujeme sa tu UML profilom a jazyku OCL. Napokon v kapitole 6 predstavíme vlastné riešenie uvedených problémov a naplníme ciele práce.

2 Ciele práce

Najprv si uvedieme ciele, ktoré budeme v tejto práci sledovať, a ktoré sa pokúsime touto prácou naplniť. Umožní nám to získať lepšiu predstavu o tom, akým smerom sa bude uberať naše snaženie.

V tejto práci si stanovíme nasledovné ciele:

- Zefektívnenie procesu vývoja softvéru
- Unifikácia modelov v organizácii
- Vytvorenie UML profilu a jeho spresnenie pomocou OCL

2.1 Zefektívnenie procesu vývoja softvéru

Predstavme si ako vyzerá proces vývoja softvéru v nejakej väčšej softvérovej firme. Je viacero prístupov, ktoré hovoria o tom, ako by mal taký proces prebiehať (vodopádový model, inkrementálno-iteratívny, Rational Unified Process, a pod.). Ale v princípe sa vo všetkých prístupoch objavujú určité fázy. Najprv je potrebné zozbierať požiadavky na systém, aby sme boli schopní zadefinovať, čo má systém robiť a čo zákazník od informačného systému očakáva. Keď už sú požiadavky zozbierané a odsúhlasené zákazníkom, môžu analytici vytvoriť prvotnú analýzu systému. Táto analýza potom putuje k návrhárom, ktorí na jej základe vytvoria návrh architektúry systému a ako by mal byť daný systém implementovaný. Programátori potom podľa tohto návrhu postupujú a vytvoria samotnú aplikáciu. Táto sa dostane k testerom na otestovanie, a po úspešnom otestovaní môže byť aplikácia nasadená v praxi.

My sa budeme zaoberať hlavne fázou analýzy. Zameriame sa teda na analytické modely, teda na výsledné produkty práce analytikov. Je jasné, že keď sa nám podarí zefektívniť určitým spôsobom túto fázu, bude to mať dopad na celý vývojový proces, ktorý sa tým následne takisto zefektívni.

2.2 Unifikácia modelov v organizácii

Predstavme si teraz znovu väčšiu softvérovú firmu a pozrime sa bližšie na fázu analýzy. Dvaja rôzni analytici majú prirodzene odlišné chápanie a rozmýšľanie, a odlišné štýly pri príprave analýzy. Preto sa môže stať, že v jednom projekte dostane návrhár analýzu od jedného analytika, a o mesiac pri práci na inom projekte dostane analýzu od druhého analytika, kde však podobné koncepty budú modelované inak v porovnaní s prvým analytikom. Toto môže prácu návrhára podstatne skomplikovať, nehovoriac o tom, že by to mohlo viesť aj k chybám. Keď si už totiž návrhár osvojil určitý štýl, mohlo by sa stať, že by v druhej analýze pochopil určité koncepty v tom duchu, v akom ich používal prvý analytik, aj keď druhý analytik to myslel trochu ináč.

Nejde tu však len o osvojenie si určitého štýlu modelovania. Je totiž veľmi ťažké zlepšiť kvalitu práce a zefektívniť celkový proces, keď sú rôzne výstupy (v tomto prípade v podobe modelov) a neexistuje štandardný prístup. Bez unifikácie je teda veľmi ťažké zefektívniť a skvalitniť vývojový proces.

2.3 Vytvorenie UML profilu a jeho spresnenie pomocou OCL

Vytvorením UML profilu umožníme analytikom používať vo svojich modeloch entity, ktoré priamo v jazyku UML nie sú, ale ktoré analytici potrebujú zachytiť vo svojich modeloch. Tým umožníme, aby mohli rôzni analytici vytvárať rovnaké výstupy pokiaľ ide o štruktúru modelov, a teda poskytneme určitý štandardný prístup. Tým dosiahneme cieľ unifikácie analytických modelov v organizácii.

Spresnením tohto profilu pomocou OCL takisto prispejeme k dosiahnutiu cieľa unifikácie analytických modelov, nakoľko bude možné v modelovacích nástrojoch kontrolovať výstupy analytikov, či zodpovedajú stanoveným podmienkam. Týmto zároveň prispejeme k väčšej kvalite analytických modelov, a teda aj k zefektívneniu procesu vývoja softvéru.

3 Súčasné trendy v softvérovom inžinierstve

Proces vývoja softvéru sa môže líšiť aj v závislosti od veľkosti softvérovej firmy a od veľkosti projektu. Čím väčšia firma a čím väčší projekt, tým viac rolí existuje. Nebolo by ani rozumné, ani efektívne, aby niekoľko sto ľudí pracovalo ako jeden tím. Preto sa obvykle určia podprojekty a ľudia vo firme sa zadelia do tímov. Tým však podstatne narastajú nároky na komunikáciu a výmenu artefaktov medzi týmito tímami. Zároveň sa aj zvyšuje riziko zlej komunikácie a následnej chyby. Preto je tu veľmi citeľná potreba výmeny artefaktov v jednotnej podobe.

Ciele spomenuté v druhej kapitole, sú dlhodobo známe a samozrejme, že bola snaha o ich riešenie. Aj vďaka tomu vznikol jazyk UML (Unified Modeling Language), ktorý sa určitým spôsobom snaží riešiť cieľ unifikácie modelov v organizácii. Tento modelovací jazyk sa snaží zjednotiť dlhoročné skúsenosti s modelovacími technikami a zapracovať súčasné najlepšie softvérové postupy (best practices) do štandardného prístupu k vývoju softvéru. V časti 3.1 si popíšeme najnovšiu verziu jazyka UML, verziu UML 2.0.

Je dôležité si uvedomiť, že UML nie je softvérový proces, ale je používaný v rámci určitého softvérového procesu. UML iba definuje modelovacie elementy používané na popis artefaktov softvérového vývoja, nepopisuje žiaden proces na vytváranie týchto artefaktov. Špecifikácia jazyka UML nedefinuje štandardný proces, ale je zamýšľané používať ho v rámci iteratívneho vývojového procesu. UML však podporuje väčšinu existujúcich objektovo orientovaných vývojových procesov.

V časti 3.2 si popíšeme prístupy k procesu vývoja softvéru. Ale keďže sa v tejto práci zameriavame na fázu analýzy, budeme hovoriť o procese týkajúcom sa práve tejto prvotnej etapy vo vývoji softvéru.

3.1 UML 2.0

Unified Modeling Language (UML) je relatívne otvorený štandard, kontrolovaný otvoreným konzorciom spoločností Object Management Group (OMG). OMG bolo zostavené za účelom vytvárania štandardov, ktoré podporujú interoperabilitu, najmä interoperabilitu objektovo orientovaných systémov. OMG je pravdepodobne známe hlavne vďaka CORBA (Common Object Request Broker Architecture) štandardom. OMG definuje niekoľko modelovacích jazykov, z pomedzi ktorých je UML zrejme najviac rozšírený a používaný.

UML je všeobecne použiteľný vizuálny modelovací jazyk používaný na špecifikovanie, vizualizáciu, konštruovanie a dokumentovanie artefaktov softvérového systému. Umožňuje tvorcom systému vytvárať návrhy, ktoré zobrazujú ich predstavy štandardným, ľahko pochopiteľným spôsobom a poskytuje možnosti na efektívne zdieľanie a komunikáciu týchto predstáv s ostatnými. Zachytáva rozhodnutia ohľadom vytváraných systémov a takisto aj ich celkové pochopenie.

UML zobrazuje informácie o statickej štruktúre a dynamickom správaní systému. Statická štruktúra definuje objekty dôležité pre systém a jeho implementáciu, ako aj vzťahy medzi týmito objektmi. Dynamické správanie popisuje históriu objektov v čase a komunikáciu medzi objektmi za účelom dosiahnutia stanovených cieľov. Modelovanie systému z viacerých odlišných, ale súvisiacich pohľadov umožňuje jeho pochopenie pre rôzne účely.

UML takisto obsahuje organizačné konštrukcie pre usporadúvanie modelov do balíkov (packages) a umožňuje softvérovým tímom rozložiť systémy do menších, ľahšie spracovateľných častí, pochopiť a kontrolovať závislosti medzi týmito balíkmi a manažovať verzie modelov a jeho častí v komplexnom vývojovom prostredí. Obsahuje aj konštrukcie na znázornenie implementačných rozhodnutí a pre organizovanie bežiacich (run-time) elementov do komponentov.

Je potrebné si uvedomiť, že UML je v prvom rade jazyk. Jazyk poskytuje slovnú zásobu a pravidlá pre kombinovanie slov v tejto slovnej zásobe za účelom komunikácie. Modelovací jazyk je jazyk, ktorého slovná zásoba a pravidlá sa zameriavajú na konceptuálnu a fyzickú reprezentáciu systému. Takže UML má ako každý jazyk syntax a sémantiku, t.j. existujú pravidlá týkajúce sa usporiadávania a spájania jednotlivých elementov a takisto pravidlá hovoriace o tom, čo to znamená, keď sú tieto elementy organizované určitým spôsobom.

3.1.1 Použitie jazyka UML

UML môže byť použitý v rôznych oblastiach a dokáže zachytiť a popísať temer čokoľvek od usporiadania firmy, cez biznis procesy až po distribuované podnikové softvéry.

Medzi najbežnejšie spôsoby použitia jazyka UML patria:

- Návrh softvéru
- Popisovanie softvérových alebo biznis procesov
- Znázornenie detailov o systéme týkajúcich sa požiadaviek alebo analýzy
- Dokumentovanie existujúceho systému, procesu alebo organizácie

UML bol úspešne použitý v mnohých doménach, napr.:

- Podnikové informačné systémy
- Bankové, finančné a investičné sektory
- Telekomunikácie
- Zdravotná starostlivosť
- Distribuované systémy
- Vnorené systémy (embedded systems)
- Maloobchodný predaj
- Zásobovanie
- a iné ...

a v rôznych implementačných platformách, napr.:

- .NET
- J2EE
- CORBA

3.1.2 Modelovanie

Modelovanie je prostriedkom na zachytenie ideí, vzťahov, rozhodnutí a požiadaviek v dobre definovanej notácii, ktorá môže byť použitá pre rôzne domény. Model je popis (časti) systému v dobre definovanom jazyku. Dobre definovaný jazyk je jazyk s dobre definovanou formou (syntaxou) a významom (sémantikou), ktorý je vhodný na automatizovanú interpretáciu počítačom.

Modelovanie umožňuje lepšie pochopenie systému. Častokrát je potrebných niekoľko prepojených modelov, aby sme boli schopní systému naozaj porozumieť. Pre softvérové systémy je preto dôležitý taký modelovací jazyk, ktorý poskytuje niekoľko rôznych pohľadov na systém a jeho architektúru, ako aj na jeho vývoj počas životného cyklu vývoja softvéru.

Jazyk UML nie je obmedzený len na modelovanie softvéru. V skutočnosti je dostatočne expresívny, aby dokázal modelovať aj nesoftvérové systémy.

3.1.3 Stručný prehľad histórie

Teraz si v krátkosti zosumarizujeme historický vývin jazyka UML. Tento prehľad je dôležitý, lebo je ťažké pochopiť, kde sa UML dnes nachádza, bez porozumenia histórie toho, ako sa vyvíjal až po súčasnosť.

UML sa v podstate stal štandardom pre modelovanie softvérových aplikácií a jeho popularita narastá aj v modelovaní iných domén. Vznikol v rámci úsilia zjednodušiť a konsolidovať veľký počet objektovo orientovaných vývojových metód, ktoré v tom čase vznikali. Jeho korene sú v troch odlišných metódach: Boochova Metóda, ktorej autorom je Grady Booch, Metóda na modelovanie objektov (Object Modeling Technique), ktorej spoluautorom je James Rumbaugh, a Objectory, autorom ktorej je Ivar Jacobson. Známi ako Traja kamaráti (Three amigos), Booch, Rumbaugh a Jacobson začali v roku 1994 pracovať na niečom, čo sa neskôr stalo prvou oficiálnou verziou jazyka UML.

V roku 1996 OMG vydalo požiadavku na návrhy ohľadom štandardného prístupu k objektovo orientovanému modelovaniu. Booch, Rumbaugh a Jacobson začali spolupracovať s metodológmi a vývojármi z rôznych spoločností, aby vytvorili návrh dostatočne zaujímavý pre členov OMG, ako aj modelovací jazyk, ktorý by bol všeobecne akceptovaný tvorcami CASE modelovacích nástrojov, metodológmi a vývojármi, ktorí by sa v konečnom dôsledku stali jeho používateľmi.

Finálny návrh UML bol predložený OMG v roku 1997 a bol výsledkom spolupráce mnohých ľudí. V tom istom roku bol jazyk UML akceptovaný OMG a vydaný ako UML verzia 1.1. UML odvtedy prešiel niekoľkými zmenami a vylepšeniami až po súčasnú verziu 2.0. Každá revízia sa snažila venovať problémom a nedostatkom, ktoré boli identifikované v predchádzajúcich verziách, čo viedlo k zaujímavému rozširovaniu a zmenšovaniu jazyka. UML 2.0 je zatiaľ najrozsiahlejšia špecifikácia vzhľadom na počet strán (len samotná špecifikácia superštruktúry má vyše 600 strán), ale reprezentuje doteraz najkompaktnejšiu verziu UML.

Medzi niektoré význačné črty najnovšej verzie UML patria:

- Zosúladenie jadra UML s konceptuálnymi modelovacími časťami Meta Object Facility (MOF)
- Existencia a dostupnosť profilov, ktoré umožňujú definovať doménovo a technologicky špecifické rozšírenia UML
- Vylepšená verzia jazyka Object Constraint Language

3.1.4 Špecifikácie

Jazyk UML je definovaný niekoľkými dokumentmi publikovanými OMG. Tieto dokumenty sú dostupné na oficiálnej stránke OMG (http://www.omg.org). UML 2.0 je distribuovaný ako 4 špecifikácie:

- UML Infraštruktúra
- UML Superštruktúra
- Object Constraint Language (OCL)
- Výmena diagramov (Diagram Interchange Specification)

Teraz si stručne popíšeme jednotlivé špecifikácie:

3.1.4.1 UML Infraštruktúra

Infraštruktúra definuje fundamentálne, kľúčové koncepty v UML, ktoré môžu byť použité čiastočne alebo úplne inými špecifikáciami, napr. MOF alebo CWM. Obsahuje iba základné statické koncepty z UML a je orientovaná k popisu štruktúre dát. Infraštruktúra je vlastne metamodel (model popisujúci model), ktorý je použitý na vytvorenie zvyšku UML. Táto špecifikácia zvyčajne nie je používaná koncovým užívateľom UML, ale poskytuje základy pre UML Superštruktúru.

3.1.4.2 UML Superštruktúra

Superštruktúra formálne definuje elementy jazyka UML. V podstate definuje kompletný jazyk UML tak, ako ho poznajú používatelia. Infraštruktúra obsahuje podmnožinu nazvanú kernel, ktorá zahŕňa do dokumentu superštruktúry všetky relevantné časti infraštruktúry. Táto špecifikácia je obvykle používaná tvorcami CASE nástrojov a autormi kníh o UML, aj keď boli už určité snahy spraviť ju čitateľnejšiu aj pre širšiu verejnosť.

3.1.4.3 OCL

OCL špecifikácia definuje jazyk na písanie rôznych obmedzujúcich podmienok a výrazov pre elementy modelu. OCL sa často využíva v situácii, keď prispôsobujeme UML konkrétnej doméne a potrebujeme použiť určité obmedzenia, ale takisto sa používa aj na formálne spresnenie modelov. Viac o OCL si povieme v sekcii 5.2.

3.1.4.4 Výmena diagramov

Táto špecifikácia bola napísaná za účelom poskytnutia spôsobu, ktorý umožní zdieľať UML modely medzi rôznymi modelovacími CASE nástrojmi. Predchádzajúce verzie jazyka UML definovali XML (Extensible Markup Language) schému na zachytenie informácie o použitých elementoch v UML diagrame, ale táto XML schéma neobsahovala informáciu o tom, ako sú tieto elementy v diagrame usporiadané. Na vyriešenie tohto problému bola táto špecifikácia vyvinutá spoločne s mapovaním z novej XML schémy do SVG (Scalable Vector Graphics) reprezentácie.

3.1.5 Metamodelovanie

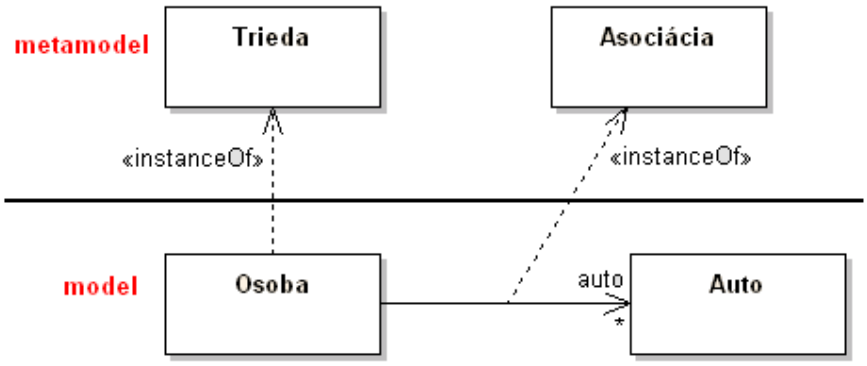
Model je definovaný ako popis systému (alebo jeho časti) napísaný v dobre definovanom jazyku. Mechanizmus definovania a vytvárania takého dobre definovaného modelovaciaho jazyka sa nazýva metamodelovanie.

Pri metamodelovaní rozlišujeme najmä medzi modelmi a metamodelmi. Model definuje, aké elementy môžu existovať v systéme. Metamodel je výsledkom procesu abstrakcie, klasifikácie a zovšeobecňovania problémovej domény modelovacieho jazyka. Teda metamodel je v zásade modelom modelovacieho jazyka a definuje elementy, ktoré môžu byť použité v modeli.

Modelovanie a metamodelovanie sú v skutočnosti identické aktivity, rozdiel je len v interpretácii. Modely sú abstraktné reprezentácie skutočných systémov alebo procesov. A keď proces, ktorý modelujeme, je procesom vytvárania iných modelov, vtedy sa modelovanie stáva metamodelovaním.

Každý element, ktorý môže byť použitý v modeli, je definovaný v metamodeli jazyka. V jazyku UML je možné používať triedy, atribúty, asociácie a iné, pretože v metamodeli jazyka UML sú elementy, ktoré definujú, čo je to trieda, atribút, asociácia a podobne. Keby napr. metatrieda Asociácia nebola zahrnutá v UML metamodeli, nebolo by možné definovať asociáciu ani v UML modeli.

Pri znázornení vzťahu medzi metamodelom a modelom dostaneme dve vrstvy. Tento vzťah znázorňuje obrázok 1. Vrstva metamodelu definuje napr., čo je to Trieda (Class). Hovorí o tom, že trieda môže mať atribúty a operácie, a že medzi triedami môžu existovať asociácie. Trieda v metamodeli je metatrieda, koncept, ktorý popisuje čo to trieda je a ako sa používa. Inštanciou metatriedy Trieda je konkrétna trieda, ktorú môžeme vidieť v UML diagrame. Rôzne inštancie metatriedy Trieda popisujú rôzne typy objektov. Metamodel takisto definuje aj Asociáciu, ktorá je tiež metatriedou, podobne ako Trieda. V tomto dvojvrstvovom príklade, vrstva metamodelu definuje symboly ako triedy a asociácie, ktoré môžu byť použité v modeli. Vrstva modelu popisuje informácie ako napr. osoby, autá a ich vzťahy, viď obrázok, použitím symbolov definovaných v metamodeli. UML diagramy vytvorené vývojármi existujú na vrstve modelu. Vrstva metamodelu definuje pravidlá, podľa ktorých je možné vytvárať diagramy a definovať elementy diagramu.



Obrázok 1 – Vzťah medzi modelom a metamodelom

3.1.6 Štvorvrstvová architektúra

OMG definuje štvorvrstvovú architektúru, ktorú používa pre svoje štandardy. Jednotlivé vrstvy sa nazývajú M0 (inštancie), M1 (model systému), M2 (metamodel), M3 (meta-metamodel).

3.1.6.1 Vrstva M0: inštancie

Na tejto vrstve sa nachádza bežiaci systém (run-time) s aktuálnymi inštanciami, ktoré v ňom existujú. Tieto inštancie sú napríklad zákazník "Janko Mrkvička" bývajúci na "Všeobecnej ulici 25" v "Bratislave". Vrstva M0 teda obsahuje dáta aplikácie, napríklad inštancie objektovo-orientovaného systému alebo riadky tabuliek v relačnej databáze. Táto vrstva je inštanciou modelu, teda vrstvy M1, ktorá sa nachádza o úroveň vyššie.

Všimnime si, že keď modelujeme biznis a nie softvér, inštancie na vrstve M0 sú konkrétne entity daného biznisu, napr. zamestnanci, faktúry a produkty. Keď modelujeme softvér, inštancie sú softvérovými reprezentáciami skutočných entít, napríklad počítačové verzie faktúr, objednávok a pod.

3.1.6.2 Vrstva M1: Model systému

Táto vrstva obsahuje modely, napríklad UML model softvérového systému. Vrstva M1 je model, ktorý popisuje artefakty a pravidlá danej domény. Model je inštanciou metamodelu. V M1 modeli je napríklad definovaná trieda Zákazník s atribútmi meno, ulica a mesto. Na tejto úrovni sa nachádzajú aj diagramy (diagramy tried, sekvenčné diagramy a iné..).

Medzi vrstvami M0 a M1 je určitý vzťah. Koncepty na vrstve M1 sú všetky klasifikáciami inštancií na vrstve M0. Podobne, každý element vrstvy M0 je vždy

inštanciou elementu vrstvy M1. Zákazník “Janko Mrkvička” je inštanciou M1 elementu triedy Zákazník.

3.1.6.3 Vrstva M2: Metamodel

Hlavnou úlohou vrstvy metamodelu je definovať jazyk na špecifikovanie modelov. Elementy vrstvy M2 sú teda modelovacie jazyky. Vrstva M2 definuje koncepty, ktoré môžu byť použité pri modelovaní elementu vrstvy M1 a pravidlá pre model na vrstve M1. V prípade UML, vrstva M2 definuje metatriedy “Trieda (Class)”, “Asociácia” a pod. Metamodel je inštancia meta-metamodelu, čo znamená, že každý element metamodelu je inštanciou niektorého elementu v meta-metamodeli. Známe príklady metamodelov sú UML a OMG Common Warehouse Metamodel (CWM).

Rovnaký vzťah ako je medzi elementmi vrstiev M0 a M1 existuje aj medzi elementmi vrstiev M1 a M2. Každý element vrstvy M1 je inštanciou niektorého M2 elementu, a každý element vrstvy M2 klasifikuje M1 elementy.

3.1.6.4 Vrstva M3: Meta-metamodel

Vrstva meta-metamodelu tvorí základ metamodelovacej architektúry. Vrstva M3 definuje koncepty, ktoré môžu byť použité pri definovaní modelovacích jazykov. Táto úroveň abstrakcie podporuje vytváranie mnohých rôznych modelov pochádzajúcich z rovnakej množiny základných konceptov. Takže koncept UML Triedy, ktorý patri do M2, môže byť považovaný za inštanciu prislúchajúceho elementu z M3, ktorý presne definuje, čo je Trieda a jej vzťahy s ostatnými UML konceptmi.

Zase platí, že rovnaký vzťah, ako je medzi elementmi vrstiev M0 a M1, a elementmi vrstiev M1 a M2, existuje aj medzi elementmi z vrstiev M2 a M3.

V rámci OMG, MOF je štandardný M3 jazyk. Všetky modelovacie jazyky (ako napr. UML, CWM a pod.) sú inštancie MOF.

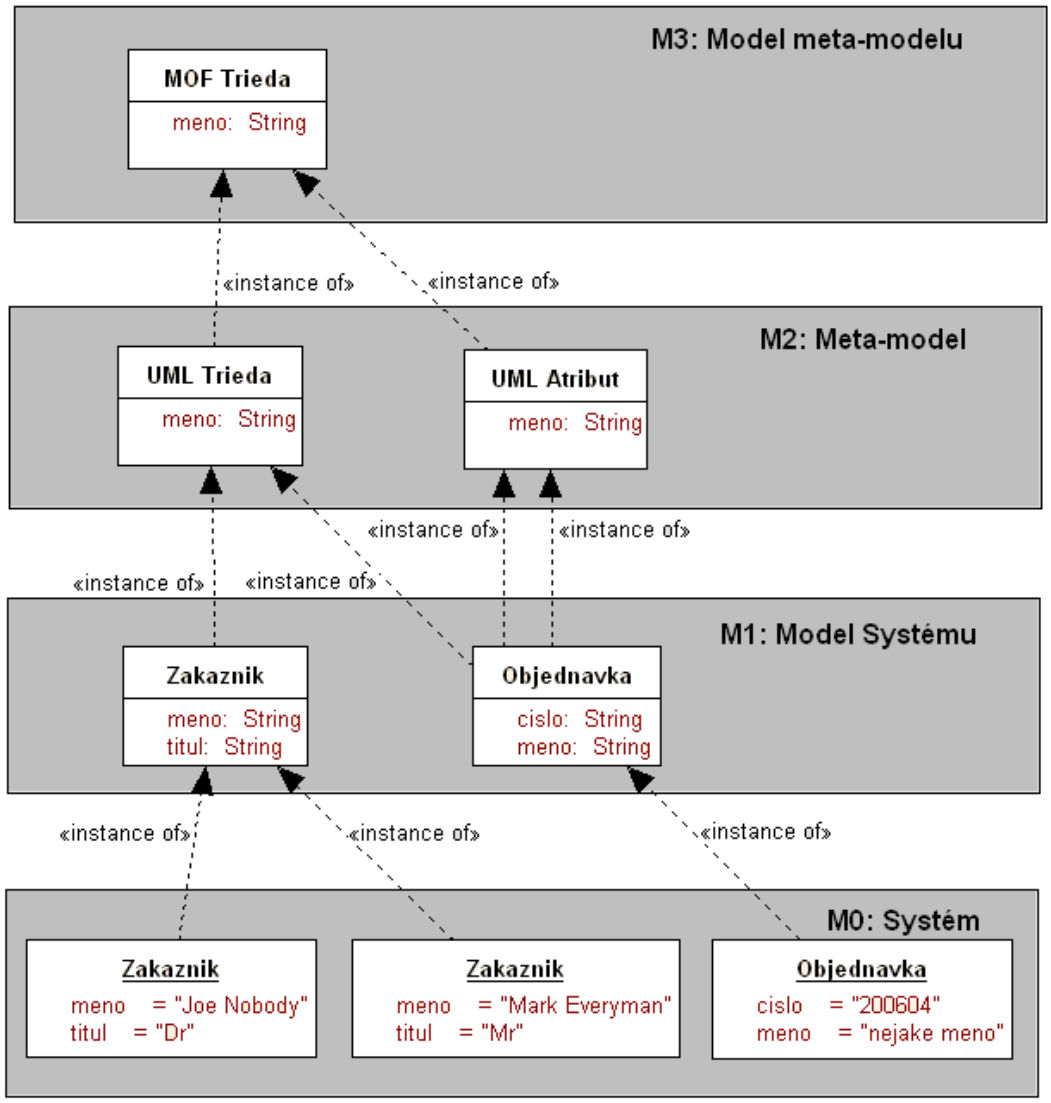
3.1.6.5 Zhrnutie

Keď si to zrekapitulujeme, vrstva M0 reprezentuje aktuálne artefakty danej domény, pozostávajúce z elementov vytvorených počas behu. Vrstva M1 je modelom pre M0. Vrstva M2 je modelom vrstvy M1, teda metamodel a na tejto vrstve sa nachádzajú modelovacie jazyky. No a nakoniec vrstva M3 je modelom vrstvy M2, tzv. meta-metamodel, kde sa nachádza MOF. Znázorňuje to obrázok 2.

Pokiaľ sa zaoberáme viacerými meta-vrstvami, obvykle tie nad vrstvou M2 sa stávajú postupne menšími a kompaktnejšími, čím vyššie sa nachádzajú v hierarchii. Špecifickou vlastnosťou metamodelovania je schopnosť vytvárať jazyky, ktoré môžu byť použité na definovanie samých seba. Knižnica Infraštruktúra (Infrastructure Library) jazyka UML je takýmto príkladom, keďže obsahuje všetky metatriedy

potrebné na zadefinovanie samej seba. MOF má tiež túto vlastnosť, keďže je založený na Infraštruktúre, a preto nie je potrebné mať ďalšie metavrstvy nad MOF (vrstva M3).

Hoci toto je konečná podoba UML architektúry, viacvrstvová architektúra môže mať v skutočnosti nekonečný počet vrstiev. Nasledujúce vyššie vrstvy vzniknú procesom abstrakcie, prirodzeného procesu spresňovania a zjemňovania pravidiel. Je to tak trochu ako s matematikou, v tom zmysle, že ako postupne robíme pokroky v našich vedomostiach, objavujeme najprv základné koncepty ako napr. sčítanie a odčítanie. Potom zistíme, že existujú pravidlá, ktoré hovoria o tom, prečo a ako sčítanie a odčítanie fungujú. Pri našom ďalšom štúdiu narážame na stále všeobecnejšie a abstraktnejšie princípy, ktoré môžu byť použité v rôznych oblastiach matematiky.

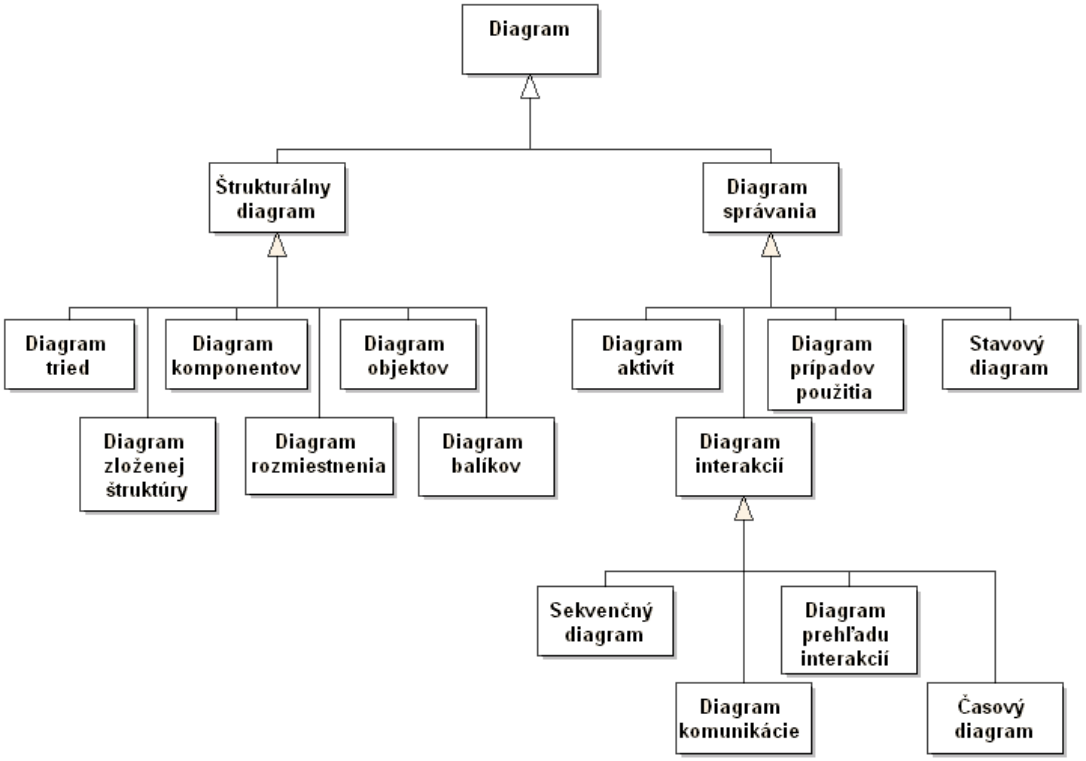


Obrázok 2 – Príklad štvorvrstvovej architektúry jazyka UML

3.1.7 Typy diagramov

UML model obvykle pozostáva z jedného alebo viac diagramov. Diagram graficky znázorňuje entity a vzťahy medzi týmito entitami. Tieto entity môžu byť skutočné objekty reálneho sveta, softvérové entity a konštrukcie a pod. Je bežné, že jedna entita sa vyskytuje aj na viacerých diagramoch. Každý diagram totiž znázorňuje konkrétny pohľad na danú entitu, ktorá je modelovaná.

UML 2.0 popisuje 13 oficiálnych typov diagramov. Delí ich na dve kategórie: štrukturálne diagramy (structural) a diagramy správania (behavioral). Celkové delenie diagramov je znázornené na obrázku 3.



Obrázok 3 – Delenie typov diagramov v jazyku UML

3.1.7.1 Štrukturálne diagramy

Štrukturálne diagramy znázorňujú statické vlastnosti modelu. Používajú sa na zachytenie fyzickej organizácie entít v systéme. UML 2.0 definuje 6 štrukturálnych diagramov:

- **Diagramy tried (Class diagrams) –** používajú triedy a rozhrania (interfaces) na znázornenie detailov o entitách vytvárajúcich systém a na zobrazenie statických vzťahov medzi nimi. Diagramy tried patria medzi najpoužívanejšie UML diagramy.

- **Diagramy komponentov (Component diagrams) –** znázorňujú časti softvéru v implementačnom prostredí. Tam, kde diagramy tried a balíkov modelujú

logický návrh softvéru, diagramy komponentov modelujú implementačný pohľad.

- **Diagramy zloženej štruktúry (Composite structure diagrams) –** koncepčne spájajú diagramy tried a komponentov. Ukazujú, ako elementy v systéme spolupracujú na uskutočnení komplexných cieľov, odhaľujú návrh komplikovaných komponentov a takisto rozhranie do komponentov oddelene od ich štruktúry.

- **Diagramy rozmiestnenia (Deployment diagrams) –** znázorňujú, ako je systém realizovaný a priradený k rozličnému typu hardvéru. Obvykle sa používajú na zobrazenie toho, ako sú komponenty nastavené počas behu systému.

- **Diagramy balíkov (Package diagrams) –** sú v podstate špeciálnym prípadom diagramov tried. Používajú tú istú notáciu, ale ich zameriavajú sa skôr na to, ako sú triedy a rozhrania spolu zoskupené.

- **Diagramy objektov (Object diagrams) –** používajú tú istú syntax ako diagramy tried a zobrazujú, ako spolu súvisia konkrétne inštancie tried v špecifickom časovom okamihu. Môžu sa použiť na zachytenie vzťahov v systéme počas behu.

3.1.7.2 Diagramy správania

Diagramy správania sa zameriavajú na správanie elementov v systéme. UML 2.0 definuje 7 diagramov správania:

- **Diagramy aktivít (Activity diagrams) -** zachycujú tok z jedného správania alebo aktivity do druhého.

- **Diagramy prípadov použitia (Use case diagrams) –** zachytávajú funkčné požiadavky na systém. Poskytujú pohľad nezávislý od implementácie na to, čo by systém mal robiť a umožňujú tvorcom systému zamerať sa viac na potreby užívateľov, ako na detaily týkajúce sa samotnej realizácie.

- **Stavové diagramy (State machine diagrams) –** zachytávajú vnútorné zmeny stavu nejakého elementu. Element môže byť malý ako jedna trieda, alebo veľký ako celý systém.

- **Diagramy interakcií (Interaction diagrams) -** V predchádzajúcej verzii jazyka UML sa výraz *diagramy interakcií* týkal sekvenčných diagramov (sequence diagrams) a diagramov spolupráce (collaboration diagrams). Tieto dva typy diagramov popisovali komunikáciu medzi objektmi za účelom splnenia istej úlohy, napr. zadanie objednávky. V UML 2.0 výraz *diagramy interakcií* zahŕňa sekvenčné diagramy, diagramy komunikácie, diagramy prehľadu interakcií a časové diagramy. Diagramy spolupráce boli nahradené diagramami komunikácie, ktoré sú ich trošku jednoduchšou verziou. Každý

z týchto štyroch typov diagramov reprezentuje určitý aspekt komunikácie medzi objektmi a poskytuje osobitý pohľad na túto komunikáciu.

- **Sekvenčné diagramy (Sequence diagrams) -** zdôrazňujú typ a poradie správ posielané medzi objektmi počas realizácie správania. Sú najbežnejším typom diagramov interakcií a sú veľmi intuitívne pre nových používateľov UML.

- **Diagramy komunikácie (Communication diagrams) –** zameriavajú sa na objekty zúčastňujúce sa určitého správania a ich štruktúru, a takisto si všímajú správy, ktoré si medzi sebou posielajú.

- **Diagramy prehľadu interakcií (Interaction overview diagrams) –** sú určitou verziou diagramov aktivít, avšak namiesto zdôrazňovania aktivity v každom kroku sa zameriavajú na to, ktoré elementy participujú na vykonávaní danej aktivity.

- **Časové diagramy (Timing diagrams) –** zameriavajú sa na podrobné časové špecifikácie pre správy. Majú špecifickú notáciu na zaznamenanie toho, ako dlho má systém spracovávať a odpovedať na správy. Často sa používajú v real-time systémoch.

3.2 Procesy vývoja v analýze

Práca analytika obvykle zahŕňa zber požiadaviek na systém, analýzu týchto požiadaviek, popis funkcionality systému a prípravu modelov a podkladov pre návrhárov. Je veľmi dôležité pochopiť, čo zákazník naozaj potrebuje. Častokrát zákazník nedokáže veľmi dobre vyjadriť svoje požiadavky, prípadne dochádza k nedorozumeniam a následne vytvorený informačný systém potom nerieši reálne potreby zákazníka, alebo ich nerieši dostatočne dobre.

V podstate existujú tri hlavné prístupy k procesu práce analytika:

- Analytik vytvorí najprv **Model prípadov použitia (Use Case Model).** Tento model pozostáva z diagramov prípadov použitia (Use Case Diagrams). Takto analytik zachytí a popíše funkčné požiadavky na systém.

- Kým pri prvom prístupe začal analytik hneď prostredníctvom prípadov použitia popisovať, čo by systém mal robiť, druhý prístup sa snaží najprv pochopiť zákazníkov biznis prostredníctvom biznis modelovania. Takže prvé, čo analytik spraví, je **Model biznis procesov (Business Process Model)** a až potom bude vytvárať model prípadov použitia. Týmto spôsobom dokáže lepšie pochopiť prostredie, pre ktoré je informačný systém vyvíjaný, a funkčné požiadavky budú lepšie zodpovedať reálnym potrebám zákazníka.

- Tretí prístup sa snaží zákazníkovi porozumieť ešte lepšie ako predchádzajúci prístup. Ide o vytvorenie tzv. **Modelu biznis stratégie (Business Strategy Model).** V tomto modeli sa analytik snaží zachytiť zákazníkove ciele a celkovú

firemnú stratégiu. Až potom bude nasledovať modelovanie biznis procesov a následné modelovanie prípadov použitia a funkčných požiadaviek. Tento model stratégie je obvykle vyjadrený vo forme vízie firmy, kde je všetko popísané slovne. V takom prípade však v modeli neexistuje traceability (sledovateľnosť, vystopovateľnosť), či dané ciele a predstavy zákazníka sú vytvoreným informačným systémom aj naozaj naplnené.

My sa v tejto práci budeme venovať práve Modelu biznis stratégie a pokúsime sa umožniť sledovateľnosť zákazníkových cieľov v modeli.

4 Problémy spojené s modelovaním systémov

V tejto sekcii si uvedieme existujúce problémy, ktoré sa v tejto práci budeme snažiť riešiť.

4.1 Viacznačnosť jazyka UML

Samotné UML diagramy môžu byť častokrát pochopené dvoma analytikmi trochu odlišne. Vyplýva to zo slabo definovanej sémantiky UML, ktorá je definovaná len semiformálne, naturálnym jazykom (slovne). UML teda nemá formálne definovanú sémantiku, a práve preto sa môže stať, že dvaja ľudia pochopia určité koncepty každý trochu inak.

UML diagram, napríklad diagram tried, obvykle nie je dostatočne presný. Často je potrebné doplniť dodatočné podmienky týkajúce sa objektov v modeli. Tieto podmienky sú obvykle popísané v bežnom hovorovom jazyku. V praxi sa ukázalo, že takýto prístup vedie zväčša k nejasnostiam.

Tento nedostatok sa snaží OMG riešiť pomocou jazyka OCL, ktorým sa dá spresniť syntax aj sémantika jazyka UML. O jazyku OCL si povieme viac v sekcii 5.2.

4.2 Nedostatočná špecializácia na konkrétnu doménu

UML je všeobecne použiteľný modelovací jazyk. V jazyku UML sú častokrát rovnaké modelovacie elementy použité na popísanie rôznych objektov sveta.

Skutočnosť, že UML je dosť všeobecný jazyk, môže byť nevýhodou pri modelovaní niektorej konkrétnej domény, kde by viac špecializované jazyky mohli byť vhodnejšie. Jazyk UML preto poskytuje určité mechanizmy umožňujúce rozšírenie a úpravu jeho syntaxe a sémantiky na prispôsobenie sa danej doméne. Tieto mechanizmy sú zoskupené v UML Profiloch.

UML profily si bližšie priblížime v sekcii 5.3.

4.3 Problém s unifikáciou

Ako sme si už spomenuli, jazyk UML nepopisuje softvérový proces a ani to nie je jeho snahou. Je na analytikovi, ako bude postupovať pri svojej práci a tvorbe modelov. Ako je teda možné zabezpečiť, aby analytici pri svojej práci postupovali rovnako?

Riešenie tohto problému je možné dosiahnuť kombináciou UML profilov a jazyka OCL, ktorý tento UML profil spresní.

4.4 Neúplnosť traceability na stratégiu

Pri modelovaní biznis stratégie sa ciele a vízia zákazníka obvykle popisujú len slovne. Ak sa však tieto ciele nezachytia nejakým rozumným spôsobom v modeli a neprepoja sa s definíciami funkčných požiadaviek, nebude možné dosiahnuť sledovateľnosť toho, či vytvorenie informačného systému podľa funkčných požiadaviek aj reálne naplní strategické ciele zákazníka.

Priamo v jazyku UML momentálne neexistuje explicitná možnosť vyššie spomenutého modelovania. V tejto práci sa tento problém pokúsime vyriešiť.

5 Možnosti riešenia

V predchádzajúcej kapitole sme si uviedli existujúce problémy súvisiace s modelovaním systémov. Povedali sme si aj akými prostriedkami je ich možné riešiť. V tejto kapitole si teda popíšeme spomínané prostriedky a mechanizmy na ich riešenie. Najprv si v sekcii 5.1 popíšeme UML profily a následne v sekcii 5.2 jazyk OCL.

5.1 UML profily

UML je rozsiahla a detailná špecifikácia. Ale nie je dostatočne rozsiahla alebo detailná na to, aby zahŕňala všetky modelovacie aspekty pre každú platformu či doménu. Aby sa vyhovelo týmto potrebám, jazyk UML môže byť rozšírený.

UML je rozšíriteľný jazyk v tom zmysle, že poskytuje mechanizmy na zavedenie nových elementov pre špecifické domény, ako sú napríklad webové aplikácie, databázové aplikácie, biznis modelovanie, procesy vývoja softvéru, dátové sklady atď. UML teda umožňuje určité rozšírenia bez toho, aby bolo potrebné modifikovať samotný UML metamodel.

Metamodel jazyka UML môže byť rozšírený tromi spôsobmi: stereotypmi (stereotypes), metaatribútmi (tagged values) a pomocou obmedzujúcich podmienok (constraints). UML profil je vlastne zoskupením množiny týchto rozšírení.

5.1.1 Popis profilu

Balík Profily z knižnice Infraštruktúra obsahuje mechanizmy, ktoré umožňujú rozšíriť metatriedy z existujúcich metamodelov a prispôsobiť ich na rôzne účely.

Profil je vlastne balík, ktorý popisuje rozšírenia pre podmnožinu základného metamodelu a definuje stereotypy a obmedzujúce podmienky, ktoré môžu byť aplikované na vybranú podmnožinu metamodelu. Deklarácia stereotypu môže definovať aj metaatribúty (tags) existujúcich metamodelových elementov. V modeli vytvorenom používateľom môže byť potom stereotyp aplikovaný na element daného metatypu a obmedzujúce podmienky môžu byť aplikované na konkrétnu metatriedu.

Obmedzujúce podmienky môžu byť pridané v profile, ale existujúce obmedzujúce podmienky týkajúce sa existujúceho metamodelu nemôžu byť odstránené ani zoslabené. Model s profilom zostane stále UML modelom, a musí preto spĺňať všetky UML obmedzujúce podmienky.

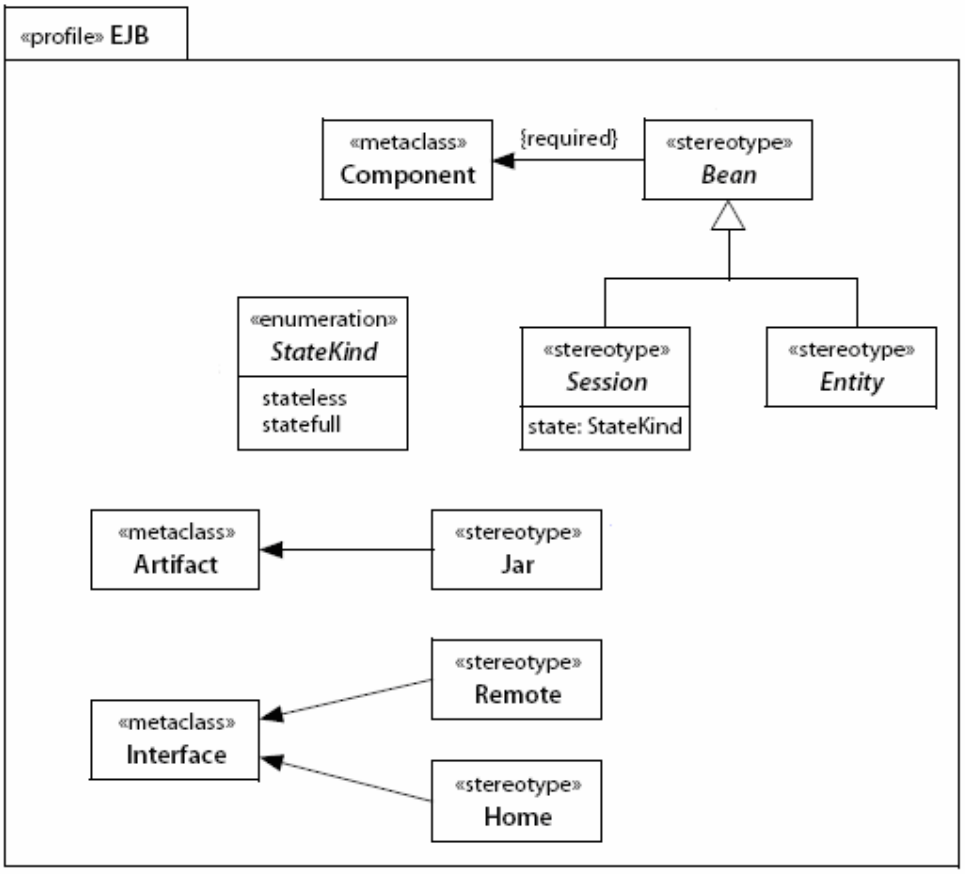
Notácia profilov

Profil je zobrazený ako symbol balíka (obdĺžnik s informačným štítkom vľavo hore) s kľúčovým slovom «profile» nad menom balíka (profilu). Obsahuje metamodelové elementy vybrané do tohto profilu, zadeklarované stereotypy so šípkami indikujúcimi

rozšírenie smerujúce k príslušným metamodelovým elementom, a obmedzujúce podmienky, ktoré sa aplikujú na metamodelové elementy. Profil môže importovať deklarácie typov, ktoré môžu byť použité ako typy pre metaatribúty stereotypov.

UML profily umožňujú prispôsobenie a rozšírenie ľubovoľného metamodelu definovaného použitím MOF, nielen UML metamodelu. Podobne UML profil môže tiež špecializovať iný UML profil.

Na obr. 4 môžeme vidieť definíciu jednoduchého profilu pre prostredie EJB (Enterprise JavaBeans).



Obrázok 4 - Definícia EJB profilu

Profil sa stane dostupný pre daný balík v modeli prostredníctvom aplikovania profilu (profile application) na tento balík. Stereotypy definované v profile môžu byť použité na modelové elementy v danom balíku a obmedzujúce podmienky v profile sa aplikujú na elementy daného balíka. Mnoho rôznych profilov môže byť definovaných, eventuálne s protikladnými definíciami. Na daný balík môže byť aplikovaný jeden alebo viac profilov. Viac profilov však môže byť aplikovaných na jeden balík len za predpokladu, že ich obmedzujúce podmienky nie sú v rozpore. Ak aplikovaný profil závisí na inom profile, obidva profily musia byť na aplikované na daný balík.

Aplikovanie profilu je znázornené čiarkovanou šípkou označujúcu závislosť, šípka smeruje od balíka, na ktorý sa profil aplikuje, smerom k aplikovanému profilu. Kľúčové slovo «apply» je umiestnené na šípku.

5.1.1.1 Stereotypy (Stereotypes)

Stereotyp je nový druh modelového elementu definovaný v rámci profilu a založený na existujúcom modelovom elemente. Je to v podstate nová metatrieda. Stereotypy môžu rozširovať sémantiku, ale nie štruktúru existujúcich metatried. Obyčajne stereotyp reprezentuje odlišný význam a použitie, ako pôvodný modelový element, ktorý je týmto stereotypom rozšírený.

Stereotypy môžu byť aplikované na všetky typy UML elementov, ako sú triedy, prípady použitia, komponenty, asociácie, a pod.

Každý stereotyp je odvodený z niektorej základnej metatriedy. Všetky elementy označené daným stereotypom majú vlastnosti tejto metatriedy. Stereotyp môže vzniknúť aj špecializovaním iného stereotypu. V tomto prípade dediaci stereotyp má tiež vlastnosti rodičovského stereotypu. Napokon teda platí, že každý stereotyp je založený na niektorej metatriede.

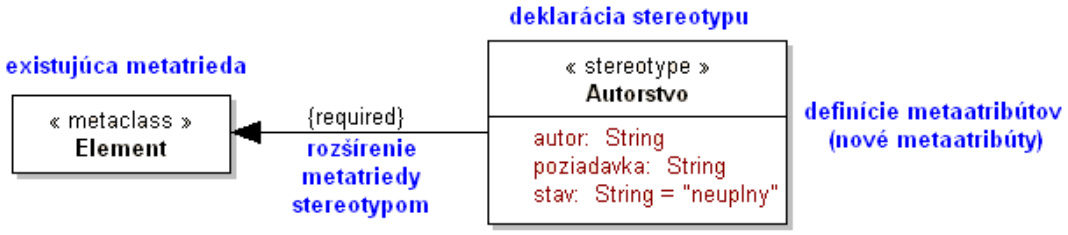
Element so stereotypom môže mať dodatočné obmedzujúce podmienky mimo tých, ktoré sa týkajú samotného modelového elementu. Takisto môže mať odlišný symbol a dodatočné vlastnosti (metaatribúty) definované prostredníctvom definícií metaatribútov (tag definitions). Predpokladá sa, že napr. generátory kódu alebo iné nástroje budú pristupovať ku elementom so stereotypmi trochu odlišne, napr. generovaním iného kódu. Zámer je ten, že všeobecný modelovací nástroj by mal pristupovať ku elementu so stereotypom vo väčšine prípadov ako ku bežnému elementu s určitými dodatočnými textovými informáciami o tomto elemente. Ale mal by rozlišovať medzi týmito elementmi pri určitých sémantických operáciách, ako je napr. kontrolovanie, či je model vytvorený podľa zodpovedajúcich pravidiel (well-formedness rules).

Modelové elementy môžu mať žiadny, jeden, alebo viac stereotypov. Určité stereotypy sú už v UML vopred zadefinované, ostatné môžu užívatelia zadefinovať sami.

Notácia stereotypov

Stereotypy sú definované v rámci profilu. V tomto diagrame (profil je vlastne špeciálny druh balíka a teda ide o diagram balíka) je každý stereotyp zobrazený ako symbol pre triedu (obdĺžnik) s kľúčovým slovom «stereotype» nad menom tohto stereotypu. Tento obdĺžnik obsahuje aj priestor pre atribúty, kde sú zobrazené dodefinované metaatribúty (tags) a platí pre nich rovnaká syntax ako pre normálne atribúty (v diagramoch tried). Stereotyp sa aplikuje na metatriedu zo základného metamodelu. Táto metatrieda je zobrazená takisto ako symbol pre triedu, s kľúčovým slovom «metaclass».

Vzťah medzi stereotypom a metatriedou, na ktorú je tento stereotyp aplikovaný, je zobrazený šípkou smerujúcou od stereotypu k danej metatriede. Táto šipka má na konci (pri danej metatriede) hrot v tvare vyplneného trojuholníka.



Obrázok 5 – Príklad definície stereotypu

Niekedy môžeme požadovať, aby v diagramoch, ktoré aplikujú náš profil, mali všetky elementy konkrétnej metatriedy daný steretotyp, ktorý ju rozširuje. Tak je to napr. aj v spomínanom profile pre Enterprise JavaBeans, kde všetky komponenty v diagrame musia mať stereotyp «Bean». Presnejšie povedané, keďže «Bean» je abstraktný stereotyp, tak všetky komponenty v modeli (inštancie metatriedy Component) musia mať stereotyp «Entity» alebo «Session». Takže v tomto prípade sa táto naša požiadavka znázorní umiestnením podmienky {required} na šípku medzi stereotypom a rozširovanou metatriedou.

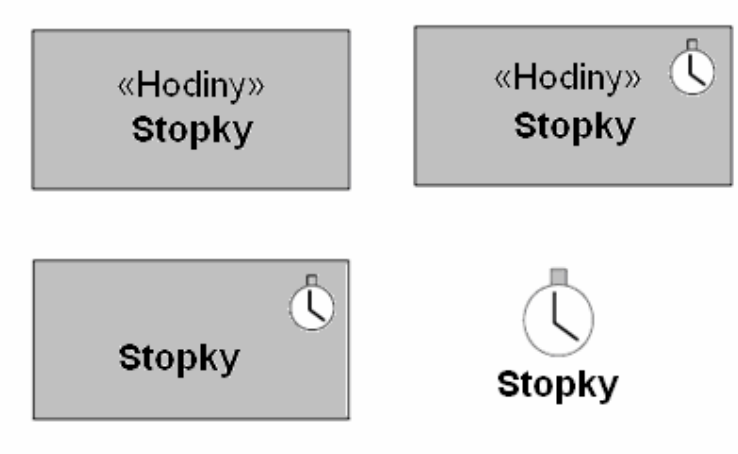
Jednu metatriedu môžeme rozšíriť aj viac ako jedným stereotypom a jeden konkrétny stereotyp môže rozširovať viac ako jednu metatriedu.

V modeli sú stereotypy znázornené tak, že ich názov je zobrazený nad menom daného elementu. Napr. komponent, ktorý má stereotyp Entity, bude mať nad svojim menom «Entity». Ak má nejaký element priradených viac stereotypov, tak mená týchto stereotypov sú oddelené čiarkou, napr. «Stereotyp1, Stereotyp2».

Hodnoty prislúchajúcich metaatribútov (tagged values) môžu byť zobrazené v symbole poznámky pripojenej k danému modelovému elementu. V rámci tejto poznámky je meno stereotypu napísané nad zoznamom metaatribútov a ich hodnôt. Ak daný element má viac stereotypov, môže byť v jednej poznámke uvedených aj viac takýchto zoznamov, vždy sa uvedie najprv názov stereotypu, a pod ním prislúchajúce metaatribúty s ich hodnotami.

Jazyk UML umožňuje aj určité obmedzené grafické rozšírenie svojej notácie. Grafická ikona môže byť asociovaná s určitým stereotypom. Táto ikona môže byť použitá v princípe dvoma spôsobmi. Jeden spôsob je, že môže byť zobrazená namiesto alebo súčasne s názvom stereotypu v rámci symbolu pre element, na ktorý bol stereotyp aplikovaný. Napríklad v symbole pre triedu je ikona zobrazená v pravom hornom rohu. Pokiaľ je ikona použitá týmto spôsobom, obsah elementu je možné vidieť v symbole pre daný element (napr. atribúty triedy). Ďalší spôsob je, že namiesto pôvodného symbolu pre element, na ktorý bol stereotyp aplikovaný, sa použije ikona asociovaná s daným stereotypom. Táto ikona obsahuje názov tohto elementu, prípadne je názov umiestnený nad alebo pod ikonou. V tomto prípade ostatné informácie prislúchajúce k tomuto modelovému elementu zobrazené nie sú

(ako napr. už spomínané atribúty triedy). Obrázok 6 znázorňuje jednotlivé možnosti zobrazenia stereotypov.



Obrázok 6 – Možnosti zobrazenia stereotypu v modeli

Známym príkladom použitia ikony pre stereotyp je symbol Aktéra (Actor) v diagrame prípadov použitia. Aktér je totiž trieda s priradeným stereotypom «actor». Pri modelovaní aktérov si môžeme vybrať, či použijeme klasickú zaužívanú ikonu predstavujúcu panáčika, alebo symbol pre triedu, kde bude nad názvom triedy zobrazené kľúčové slovo «actor».

5.1.1.2 Metaatribúty (Tagged values)

Užívateľom dodefinované metaatribúty sú dodatočnou vlastnosťou, ktorá môže byť pridaná UML elementu, aby bolo možné špecifikovať dodatočné informácie, ktoré sa predtým špecifikovať nedali.

Príkladom použitých metaatribútov môže byť biznis analytik, ktorý vytvorí prípad použitia, alebo autor triedy, prípadne dátum a čas vytvorenia alebo modifikovania danej triedy.

Pri popise metaatribútov budeme rozlišovať medzi definíciou metaatribútov (tag definition), ktorá sa nachádza na úrovni metamodelu, a použitím metaatribútu aj s jeho prislúchajúcou hodnotou (tagged value), ktorý sa nachádza už na úrovni modelu.

Definícia metaatribútu prislúcha konkrétnemu stereotypu, ktorý tento metaatribút definuje. Je to vlastne pridaný metaatribút pre metatriedu, ktorú daný stereotyp rozširuje. Definícia metaatribútu má meno a typ. Definuje vlastnosti elementov na úrovni modelu.

Definícia metaatribútu je zobrazená ako definícia klasického atribútu v obdĺžnikovom symbole zobrazujúcom deklaráciu stereotypu.

Použitie metaatribútu (tagged value) je rozširovací mechanizmus, ktorý umožňuje pridanie dodatočných informácií do modelov. Je to vlastne dvojica meno-hodnota, ktorá môže byť pripojená k modelovému elementu, na ktorý je aplikovaný stereotyp obsahujúci príslušnú definíciu metaatribútu.

Keďže stereotyp má reprezentovať po sémantickej stránke čiastočne odlišný element, ako pôvodný element, na ktorý je stereotyp aplikovaný, tak metaatribúty zadefinované v stereotype obvykle vyjadrujú vlastnosti entity, ktorú daný stereotyp reprezentuje. Hodnota metaatribútu sa takisto často používa aj na uchovávanie informácií týkajúcich sa manažmentu projektu, ako napr. autor elementu, verzia elementu, stav testovania elementu a pod.

Aby sa predišlo nejasnostiam, zadefinované metaatribúty by mali byť odlišné od už existujúcich metaatribútov modelových elementov, na ktoré sú aplikované. Modelovací nástroj by mal umožniť takúto kontrolu.

V momente, keď sa stereotyp aplikuje na modelový element, tento modelový element získava metaatribúty definované daným stereotypom. Pre každý metaatribút môže byť špecifikovaná jeho hodnota.

Notácia použitia metaatribútu

Ako sme si už spomenuli v sekcii o stereotypoch, metaatribúty a ich hodnoty sú na úrovni modelu zobrazené v poznámke pripojenej ku konkrétnemu modelovému elementu (pripomíname, že tento modelový element musí mať stereotyp, ktorý definuje príslušné metaatribúty).

Metaatribúty sú zobrazené ako textové reťazce s názvom metaatribútu, znakom '=', a hodnotou prislúchajúcou danému metaatribútu, teda majú nasledovnú formu: názov = hodnota.

5.1.1.3 Obmedzujúce podmienky (Constraints)

Ako sme si už spomenuli, stereotyp môže obsahovať aj určité obmedzujúce podmienky, ktoré sa týkajú možností jeho použitia. Tieto podmienky obvykle vychádzajú z domény, ktorú daný stereotyp modeluje. Príklad takejto obmedzujúcej podmienky môže byť, že trieda so stereotypom «biznis organizácia» môže byť spojená len s ďalšou biznis organizáciou a nie s ľubovoľnou triedou. Alebo môže obmedzujúca podmienka hovoriť o tom, že v modeli nemôžeme vytvoriť asociáciu medzi dvomi triadami so stereotypom «entity».

Obmedzujúce podmienky podmienky nám teda umožňujú špecifikovať pravidlá a obmedzenia týkajúce sa modelových elementov jazyka UML. Týmto spôsobom môžeme v modeli zachytiť pravidlá, ktoré existujú aj v doméne, ktorú modelujeme, a teda zabezpečíme, že naše modely budú konzistentné s touto doménou.

Obmedzujúce podmienky môžu byť vyjadrené dvoma spôsobmi. Prvou možnosťou je použiť prirodzený jazyk, a teda špecifikovať tieto obmedzujúce podmienky

neformálne. V tomto prípade nebude možné zabezpečiť validáciu modelu vzhľadom na tieto pravidlá (well-formedness rules).

Druhou možnosťou je definovať tieto obmedzujúce podmienky formálne prostredníctvom jazyka Object Constraint Language (OCL). OCL je štandardný jazyk na špecifikovanie obmedzujúcich podmienok a vlastností modelových elementov. V prípade, že obmedzujúce podmienky vyjadríme v jazyku OCL, získame možnosť validácie modelov vzhľadom na tieto podmienky (v prípade, že daný modelovací nástroj podporuje validáciu modelov vzhľadom na obmedzujúce podmienky definované v profile).

5.1.1.4 Zoznam hodnôt (Enumeration)

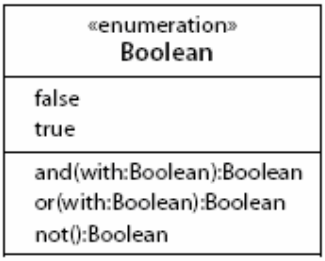
V rámci profilu máme možnosť zadefinovať si vlastný typ, pre ktorý vymenujeme hodnoty, ktoré môže nadobúdať. Tento zoznam hodnôt (enumeration) má názov a zoradený zoznam prípustných hodnôt (literály) pre tento užívateľom zadefinovaný typ.

Napríklad môžeme chcieť pridať do modelu farby. Vytvoríme zoznam hodnôt nazvaný RGBFarba, ktorý bude mať prípustné hodnoty “červená”, ”zelená”, ”modrá”.

Dátový typ Boolean je vopred definovaný zoznam hodnôt s možnými hodnotami false a true (pravda a nepravda).

Hodnoty zo zoznamu hodnôt môžu byť porovnávané na rovnosť a na relatívnu pozíciu v zozname hodnôt.

Zoznam hodnôt môže definovať operácie, ktoré berú ako vstupné argumenty literály definované v tomto zozname a vracajú literály ako výsledok operácie. Napríklad už spomínaný typ Boolean má zadefinované operácie nad jeho hodnotami false a true.



Obrázok 7 – Príklad zoznamu hodnôt (enumeration) typu Boolean

Notácia

Zoznam hodnôt je zobrazený ako obdĺžnikový symbol používaný pre triedu, s kľúčovým slovom «enumeration» nad menom tohto zoznamu hodnôt. Hodnoty sú zobrazené v priestore, v ktorom má trieda zobrazené atribúty, a prípadné operácie sú zobrazené podobne ako je to v prípade operácií pre triedu.

5.1.2 Dôvody na použitie UML profilov

UML 2.0 popisuje niekoľko dôvodov, prečo by mohlo byť výhodné a zmysluplné rozšíriť metamodel a prispôsobiť ho konkrétnym potrebám:

- Umožniť v modeloch využívanie terminológie, ktorá je zaužívaná v rámci určitej platformy alebo domény (napr. zachytenie terminológie EJB)
- Počítať so syntaxou aj pre konštrukcie, ktoré nemajú notáciu
- Umožnenie rôznej notácie pre už existujúce symboly, notácie, ktorá bude vhodnejšia pre danú aplikačnú doménu (ako napr. použiť obrázok počítača na znázornenie počítača v sieti namiesto bežného UML symbolu)
- Pridanie sémantiky, ktorá zostala nešpecifikovaná v metamodeli (napr. ako riešiť priority pri prijímaní signálov v stavových diagramoch)
- Pridanie sémantiky, ktorá neexistuje v metamodeli (napr. zadefinovanie času a hodín)
- Pridanie obmedzujúcich podmienok, ktoré určitým spôsobom obmedzujú možnosti použitia metamodelu a jeho konštrukcií (napr. vynútiť existenciu určitých asociácií medzi modelovými elementmi)
- Pridanie informácií, ktoré môžu byť použité pri transformácii modelu do iného modelu alebo do kódu (napr. definovanie mapovacích pravidiel medzi modelom a kódom v Jave)

5.1.3 Návod na definovanie profilov

Špecifikácia jazyka UML 2.0 iba definuje koncepty UML profilu a elementov, ktoré spolu vytvárajú profil. Nehovorí nič o tom, ako by bolo vhodné postupovať pri vytváraní vlastného profilu. Nasledujúce kroky poskytujú určitý návod na to, ako postupovať v prípade, že chceme zadefinovať nový profil.

1) Najprv potrebujeme určiť, aké elementy tvoria našu platformu či doménu, a aké vzťahy sú medzi týmito elementmi. Potom to vyjadríme vo forme metamodelu. Do metamodelu teda zahrnieme všetky entity našej domény, vzťahy medzi nimi, a obmedzujúce podmienky týkajúce sa štruktúry a správania týchto entít. Takto zadefinovaný metamodel nám umožní lepšie pochopiť doménu, ktorú sa chystáme modelovať a pomôže nám lepšie zadefinovať profil vrátane obmedzujúcich podmienok.

2) Keď už máme metamodel pre našu doménu hotový, sme pripravení zadefinovať samotný profil. Pre každý relevantný element z nášho metamodelu (potenciálne pre všetky elementy), ktorý chceme zahrnúť do profilu, vytvoríme príslušný stereotyp. Takto každý stereotyp aplikujeme na príslušnú metatriedu, ktorú sme v našom metamodeli použili na definíciu

daného konceptu alebo vzťahu. Aby sme vyjasnili vzťah medzi metamodelom a profilom, každý stereotyp bude pomenovaný rovnako ako korešpondujúci element z metamodelu.

3) Pre všetky atribúty objavujúce sa v metamodeli zadefinujeme metaatribúty v rámci príslušných stereotypov.

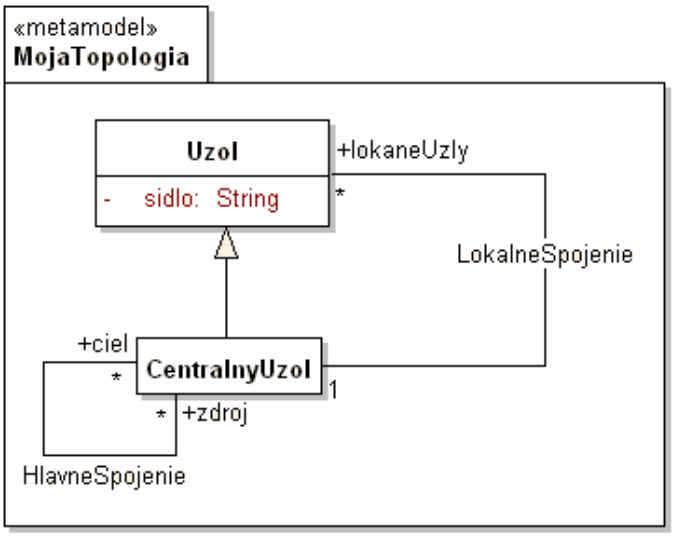
4) Z pravidiel, ktoré existujú v modelovanej doméne vytvoríme obmedzujúce podmienky pre náš profil.

5.1.4 Príklad

Teraz si ukážeme zjednodušený príklad vytvárania jednoduchého profilu. Budeme sa riadiť návodom, ktorý sme si uviedli v predchádzajúcej sekcii. Keďže jazyk OCL si vysvetlíme až neskôr, v tomto príklade sa nebudeme zaoberať obmedzujúcimi podmienkami.

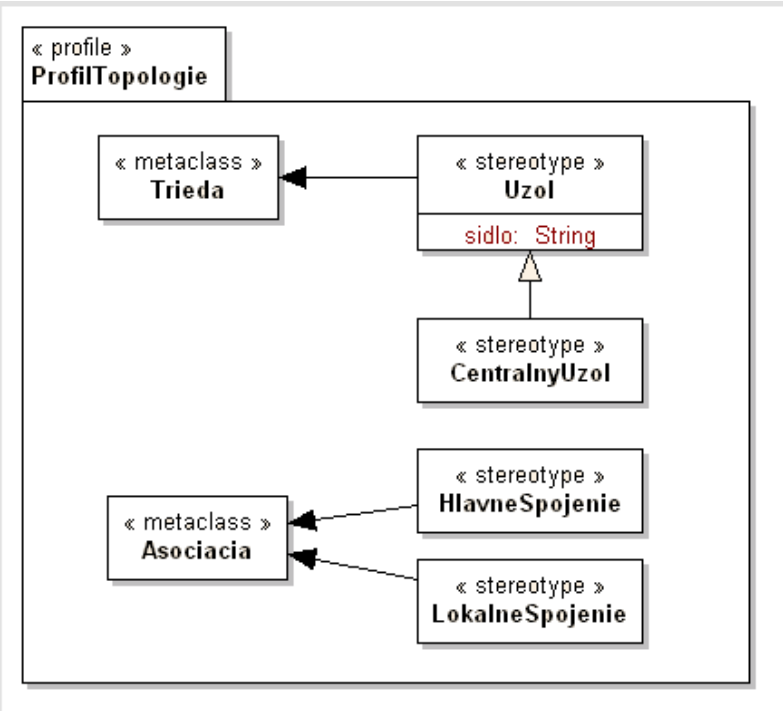
Predstavme si, že potrebujeme modelovať spojenia medzi elementmi určitého informačného systému, ktorého sieťová infraštruktúra zodpovedá hviezdicovej topológii. V tejto topológii platí, že uzly v sieti sú vždy napojené na centrálne uzly, a tieto centrálne uzly sú prepojené medzi sebou.

Najprv vytvoríme metamodel, ktorý popisuje túto doménu. Takže v metamodeli zadefinujeme uzly, ktoré budú reprezentované triedou Uzol a centrálne uzly, tieto reprezentujeme triedou Centrálny uzol. Keďže v rámci jednej hviezdice v sieťovej topológii môžu byť prepojené uzly s centrálnym uzlom, zadefinujeme spojenie medzi nimi, reprezentované asociáciou Lokálne spojenie. Centrálne uzly z rôznych hviezdic môžu byť spojené medzi sebou. Toto spojenie reprezentujeme asociáciou Hlavné spojenie. Každý uzol má aj svoje sídlo, to vyjadríme metaatribútom Sídlo. Tento metamodel je znázornený na obr. 8.



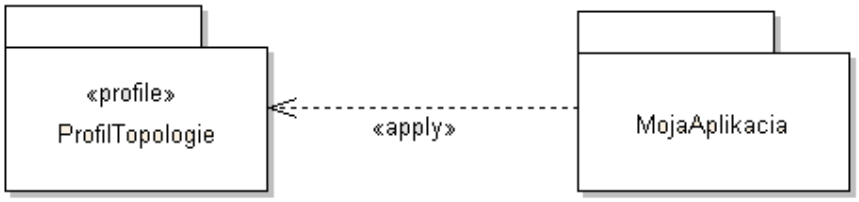
Obrázok 8 – Príklad zadefinovaného metamodelu

Teraz na základe tohto metamodelu vytvoríme profil Hviezdicová topológia. Profil definuje štyri rôzne stereotypy. Stereotypy Uzol a Centrálny uzol rozširujú metatriedu Trieda, a stereotypy Lokálne spojenie a Hlavné spojenie rozširujú metatriedu Asociácia. Stereotyp Uzol definuje aj metaatribút Sídlo typu String. Tento profil je znázornený na obr. 9.



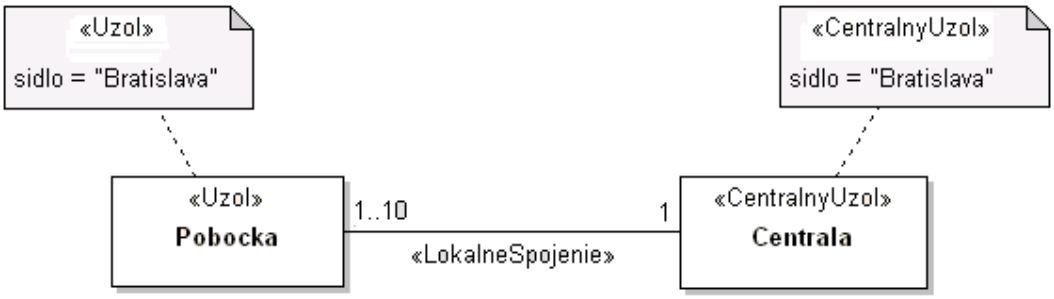
Obrázok 9 – Príklad profilu pre topológiu

Keď sme už zadefinovali náš UML profil, môžeme použiť vzťah závislosti so stereotypom «apply» na znázornenie toho, že tento profil použijeme v našej aplikácii.



Obrázok 10 – Príklad aplikovania profilu

Vďaka použitiu tohto profilu môžeme vytvárať diagramy, ako napr. na obr. 11, ktorý zobrazuje dve triedy spojené asociáciou. Jedna trieda má stereotyp Uzol, druhá Centrálny uzol. Asociácia medzi nimi má stereotyp Lokálne spojenie.



Obrázok 11 – Príklad použitia stereotypov v modeli

5.1.5 Existujúce profily

Dá sa povedať, že profilov môže existovať ľubovoľne veľa. Vždy keď bude mať niekto potrebu prispôsobiť jazyk UML doméne, ktorú modeluje, využije na to štandardný spôsob a zadefinuje si vlastný profil. Avšak cieľom jazyka UML (a teda aj konzorcia OMG) je poskytnúť štandardný jazyk na komunikáciu medzi softvérovými tímami.

Ak by napríklad niekoľko rôznych organizácií vytvorilo svoj vlastný profil na modelovanie distribuovaných systémov, nebolo by jednoduché porovnať ich modely, vytvorené použitím rôznych modelov. Preto sa OMG snaží vytvoriť množinu štandardných profilov pre rôzne, často modelované domény.

Nasledovné profily boli prijaté konzorciom OMG a sú dostupné na jeho stránke (www.omg.org).

- UML profil pre integráciu aplikácií - UML Profile and Interchange Models for Enterprise Application Integration (EAI)
- UML profil pre distribuované objektové programovanie – UML Profile for Enterprise Distributed Object Computing (EDOC)
- UML profil pre plánovanie, výkonnosť a čas - UML Profile for Schedulability, Performance, and Time
- UML profil pre testovanie - UML Testing Profile
- UML profil pre CORBA - UML Profile for CORBA

OMG takisto poskytlo ukážkové profily vo svojich špecifikáciách jazyka UML:

- UML profil pre biznis modelovanie, v špecifikáciách UML 1.0 – 1.4 (UML Profile for Business Modeling)
- UML profil pre proces vývoja softvéru, obsiahnutý v špecifikáciách UML 1.0 – 1.4 (UML Profile for Software Development Processes)

- Príklady profilov pre komponenty, pre J2EE/EJB, COM, .NET, and CCM. Sú poskytnuté v prílohe ku špecifikácii UML 2.0 na ukážku toho, ako môže byť UML prispôsobený pre komponentové architektúry. Tieto príklady nie sú mienené ako hotové a kompletné profily.

Zároveň sa pracuje aj na ďalších profiloch, ktoré momentálne ešte nie sú dostupné.

Okrem týchto OMG profilov sa v softvérovom priemysle vyvinuli aj iné profily. Medzi najznámejšie patria napr.:

- UML profil pre modelovanie dát (Data Modeling Profile for UML)
- Profil pre webové aplikácie (Web Application Extension (WEA) profile)

5.2 OCL 2.0

Object Constraint Language (OCL) je súčasťou špecifikácie jazyka UML 2.0. Je to formálny jazyk, ktorý poskytuje možnosť spresnenia modelov prostredníctvom obmedzujúcich podmienok a výrazov. Tieto výrazy obvykle špecifikujú invarianty, t.j. podmienky, ktoré musia stále platiť, alebo dotazy (queries) nad objektmi popísanými v modeli.

OCL je jazyk, ktorý dokáže vyjadriť dodatočné potrebné informácie a podmienky týkajúce sa modelov a mal by byť použitý v spojení s UML diagramami. Použitím kombinácie UML a OCL je možné zachytiť podstatne viac informácií, ako len pomocou samotného UML.

Napríklad môžeme jazykom OCL vyjadriť, že vek osoby musí byť vždy väčší ako 0, alebo že v spoločnosti musí byť vždy jedna sekretárka pripadajúca na každých desiatich zamestnancov.

Slovo Constraint (obmedzujúca podmienka) v názve jazyka OCL pochádza z jeho staršej verzie, ktorá umožňovala špecifikovať len takéto jednoduché podmienky. OCL 2.0 sa vyvinul na expresívnejší a výkonnejší jazyk, ktorého expresivita je podobná jazyku SQL. Môže byť použitý na vyjadrenie ľubovoľného výrazu týkajúceho sa elementov v diagrame.

Jazyk OCL má matematické základy. Je založený na teórii množín a predikátovej logike. Avšak jeho notácia neobsahuje matematické symboly, pretože matematická notácia by nebola vhodná pre štandardný jazyk.

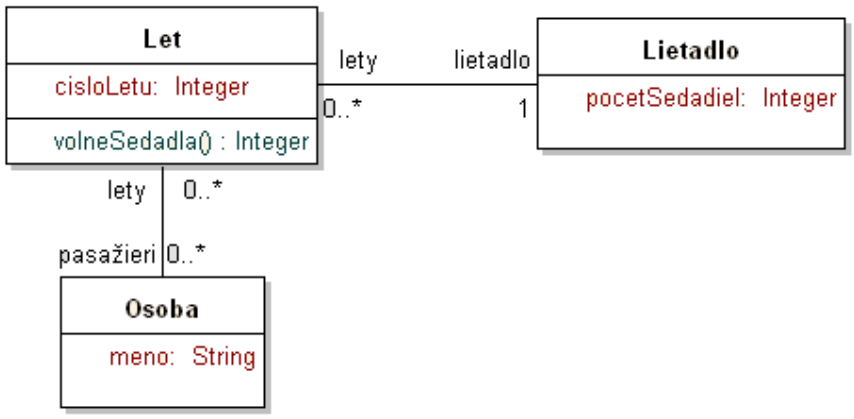
Každý OCL výraz vyjadruje hodnotu alebo objekt v rámci systému. Napríklad výraz 1+3 je platný OCL výraz typu Integer (celé číslo) a reprezentuje celočíselnú hodnotu 4.

OCL je typový jazyk. Je potrebné umožniť kontrolu OCL výrazov skôr, ako bude existovať spustiteľná verzia modelovaného systému. Tým, že OCL je typový jazyk, sa umožní kontrola OCL výrazov už počas modelovania, a teda chyby v modeli sa môžu odhaliť v skorých štádiách projektu.

OCL je takisto aj deklaratívny jazyk. V procedurálnych programovacích jazykoch sú výrazy popisom akcií, ktoré musia byť uskutočnené. V deklaratívnom jazyku výraz len vyjadruje, čo by malo byť spravené, nie ako by to malo byť spravené. OCL výrazy teda nemajú žiadne vedľajšie účinky (side-effects), čiže vyhodnotenie OCL výrazu nezmení stav systému a nemôže ani zmeniť nič v modeli. Keď vyhodnotíme nejaký OCL výraz, jednoducho nám vráti hodnotu. To znamená, že stav systému sa nemôže nikdy zmeniť vyhodnotením OCL výrazu, aj napriek tomu, že OCL môže byť použitý na špecifikovanie zmeny stavu. Okrem toho, vyhodnotenie OCL výrazu je okamžité, takže stavy objektov v modeli sa nemôžu zmeniť počas vyhodnocovania výrazu.

Prečo kombinovať UML a OCL

Modelovanie softvérových systémov bolo často synonymom pre vytváranie diagramov. Väčšina modelov pozostáva z rôznych geometrických útvarov a šípok, a tieto sú doplnené popisujúcim textom. Informácie vyjadrené takýmto spôsobom sú náchylné byť nekompletné, neformálne, nepresné a niekedy aj nekonzistentné. Mnohé takéto nedostatky v modeli sú spôsobené obmedzenými možnosťami použitých diagramov. Diagram jednoducho nemôže zachytiť všetky podmienky týkajúce sa systému.



Obrázok 12 – Príklad nedostatočnej presnosti UML diagramov

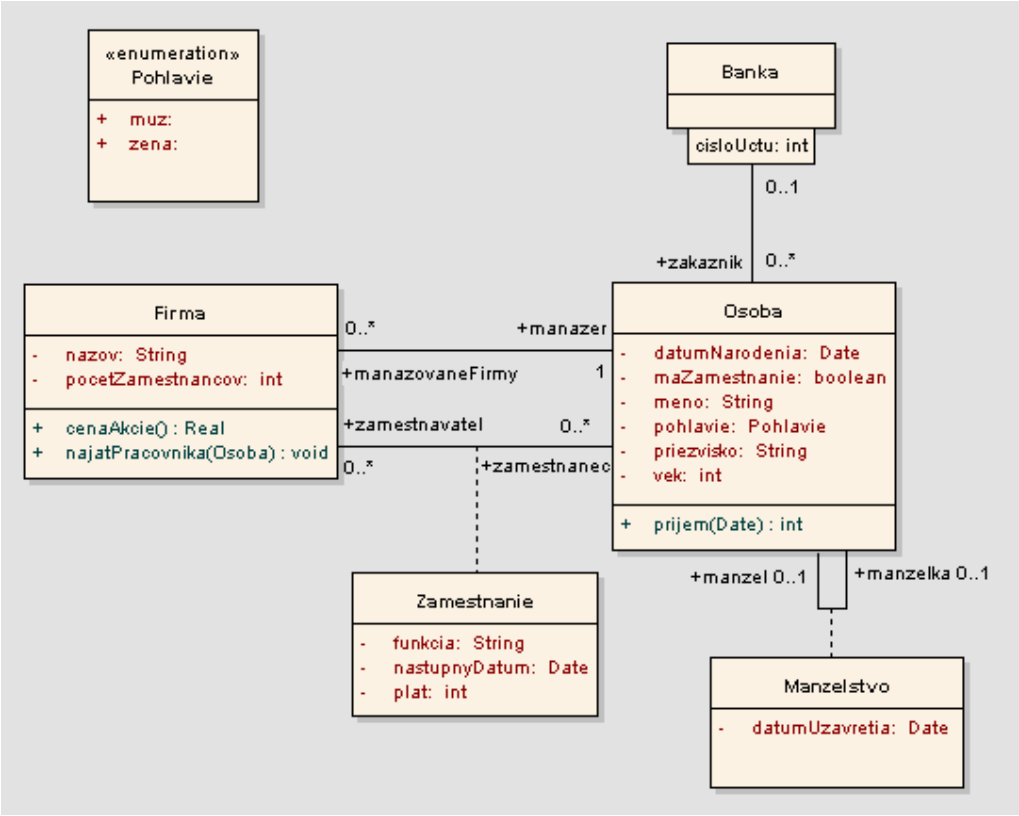
Zoberme si nasledujúci príklad. Diagram tried na obr. 12 zobrazuje tri triedy: Let, Lietadlo, Osoba, a vzťahy medzi nimi. Asociácia medzi triedou Let a triedou Osoba znázorňuje, že určitá skupina osôb je skupinou cestujúcich daného letu. Táto asociácia zobrazuje násobnosť mnoho (0..*) pri triede Osoba. To znamená, že počet cestujúcich je neobmedzený. V skutočnosti však bude počet cestujúcich daného letu obmedzený počtom sedadiel v príslušnom lietadle. Túto podmienku nie je možné znázorniť v diagrame. Pomocou OCL by táto podmienka vyzerala nasledovne:

```
context Let
inv: pasazieri->size() <= lietadlo.pocetSedadiel
```

Výrazy napísane v presnom, matematicky založenom jazyku ako je OCL, poskytujú množstvo výhod. Takéto výrazy nemôžu byť pochopené rôzne dvoma rôznymi ľuďmi, napr. analytikom a návrhárom. Tieto výrazy môžu byť tiež kontrolované počítačovými nástrojmi, a tým je možné zabezpečiť, že model je konzistentný a zodpovedá daným OCL podmienkam. Takisto sa zlepší a spresní generovanie kódu z modelu.

Kombinácia UML a OCL sa teda javí ako najlepšie riešenie na špecifikovanie presných, jednoznačných a konzistentných modelov. Samotné UML diagramy totiž nie sú dostatočne presné a jednoznačné. Naopak použitie samotného OCL by tiež nebolo vhodné, lebo OCL výrazy by sa odkazovali na neexistujúce modelové elementy, keďže OCL neposkytuje možnosť deklarácie tried a asociácií. Nehovoriac o tom, že diagramy sú prehľadnejšie ako samotný text.

V nasledujúcich častiach si popíšeme samotný jazyk OCL. V príkladoch sa budeme odkazovať na model, znázornený na obr. 13. Nie je však naším cieľom podať úplný a vyčerpávajúci prehľad možností tohto jazyka.



Obrázok 13 – Ukážkový príklad použitý pri výklade OCL

5.2.1 Základné prvky jazyka

5.2.1.1 Definícia kontextu

Akýkoľvek model, ktorého je OCL súčasťou, pozostáva aj z nejakých UML diagramov. OCL sa vo svojich výrazoch odkazuje na typy definované v UML modeli. Daný UML element, s ktorým je OCL výraz spojený, definuje ďalšie entity modelu, ktoré budú môcť byť použité vo výraze. Napríklad výraz spojený s triedou Osoba môže používať jej atribúty, alebo sa odkazovať na triedy, s ktorými je trieda Osoba spojená.

Spojenie elementu z UML diagramu s nejakým OCL výrazom sa nazýva definícia kontextu daného OCL výrazu. Označuje sa kľúčovým slovom **`context`**, za ktorým nasleduje meno typu (UML elementu).

Príklad definície kontextu pre triedu Osoba:

```
context Osoba
```

OCL výrazy môžu využívať rôzne vlastnosti alebo funkcie v závislosti od kontextu výrazu. Ak je napríklad kontext nejaký atribút, OCL výraz môže definovať jeho počiatočnú hodnotu. Ale ak je kontext trieda, v tomto prípade OCL výraz nemôže definovať počiatočnú hodnotu.

5.2.1.2 Kľúčové slovo self

Každý OCL výraz je napísaný v kontexte inštancie špecifického typu. V OCL výraze je kľúčové slovo **`self`** použité na odvolanie sa na túto inštanciu. Napríklad v kontexte triedy Osoba, sa slovo self odkazuje na inštanciu triedy Osoba. V prípade, že odkaz na inštanciu v danom kontexte je zjavný, môže sa slovo self vynechať. Na self sa môžeme pozerať ako na objekt, v ktorom začíname vyhodnocovať OCL výraz. Príklad si ukážeme v ďalšej sekcii.

5.2.1.3 Invariant

Invariant je podmienka, ktorá musí byť vždy splnená. V prípade, že OCL výraz je invariant nejakého typu (definovaného kontextom), tak musí byť vždy pravdivý pre všetky inštancie daného typu. OCL výrazy, ktoré vyjadrujú invarianty, sú typu Boolean (pravda alebo nepravda) a sú vyhodnotené ako true (pravdivé), ak je daný invariant splnený. Invariant sa v OCL označuje kľúčovým slovom **`inv`**.

Nasledujúci výraz určuje invariant, ktorý hovorí, že počet zamestnancov vo firme musí byť vždy väčší ako 10:

```
context Firma inv:
self.pocetZamestnancov > 10
```

Self v tomto príklade je inštancia triedy Firma. Tento invariant teda platí pre všetky inštancie Firmy. V tomto príklade vidíme aj použitie slova inv. Nasleduje za kontextom s príslušným typom (v tomto prípade Firma), a za slovom inv nasleduje dvojbodka. Za ňou je už samotný OCL výraz.

Vo väčšine prípadov (aj v tomto) môžeme slovo self vynechať, lebo kontext je zrejmý.Ako alternatívu pre self môžeme použiť aj iný názov, ktorý nahradí slovo self. Nasledujúci invariant je ekvivalentný s predchádzajúcim:

```
context f:Firma inv:
f.pocetZamestnancov > 10
```

OCL výraz môžeme aj pomenovať, názov bude nasledovať za kľúčovým slovom inv. Nasledujúci príklad ukazuje stále tú istú podmienku, ktorú sme nazvali dostatocnyPocetZamestnancov:

```
context f:Firma inv dostatocnyPocetZamestnancov:
f.pocetZamestnancov > 10
```

K jednému kontextu je možné priradiť aj viac podmienok:

```
context Osoba
inv: self.meno = 'Ján'
inv: self.vek = 24
```

5.2.1.4 Kontext balíkov

OCL výrazy môžu byť podobne ako všetky UML elementy zahrnuté v balíku. OCL výraz prislúcha vždy elementu, ktorý je uvedený ako kontext, a teda patrí do rovnakého balíka ako daný element. Na explicitné špecifikovanie balíka, do ktorého OCL výraz patrí, máme dve možnosti.

Prvou možnosťou je použitie kľúčových slov **package** a **endpackage**, ako to ukazuje nasledovný príklad:

```
package Balik::VnorenyBalik

context A
inv: ... nejaký invariant ...

context A::nazovOperacie(...)
pre: ... nejaká prekondícia ...

endpackage
```

Druhou možnosťou je názov balíka zahrnúť priamo do definície kontextu:

```
context Balik::VnorenyBalik::A
inv: ... nejaký invariant ...

context Balik::VnorenyBalik::A::nazovOperacie(...)
pre: ... nejaká prekondícia ...
```

Príklady ukazujú aj definíciu operácie a podmienky typu prekondícia. Tieto si vysvetlíme v ďalších sekciách.

5.2.1.5 Komentáre

OCL výrazy môžu obsahovať komentáre. Krátky komentár začína dvoma pomlčkami (--) a všetok text za pomlčkami až do konca riadku je považované za komentár. Komentáre dlhšie ako jeden riadok môžu byť uzavreté medzi symbolmi /* a */. Nasledovný príklad ukazuje platné OCL výrazy pozostávajúce len z komentárov.

```
-- toto je príklad krátkeho komentáru

/* toto je komentár presahujúci
   do druhého riadku  */
```

5.2.2 Navigácia

V OCL výraze môžeme využiť navigáciu v diagrame tried. Keď začneme v konkrétnom objekte, môžeme sa navigovať cez asociáciu príslušnej triedy a odkazovať sa na iné objekty a ich vlastnosti. Navigovať sa cez asociáciu môžeme prostredníctvom opačného konca asociácie:

```
objekt.nazovKoncaAsociacie
```

Tento výraz vyjadruje množinu objektov na druhom konci asociácie. Ak je násobnosť konca asociácie najviac jedna (0..1 alebo 1), vtedy ide o jeden objekt (môžeme sa však na neho pozerať aj ako na množinu obsahujúcu jeden prvok).

Keď začneme v kontexte Firmy, môžeme napísať nasledovné podmienky:

```
context Firma
inv: self.manazer.maZamestnanie = true
inv: self.zamestnanec->notEmpty()
```

V prvom invariante je self.manazer jeden objekt triedy Osoba, lebo násobnosť asociácie je 1. V druhom invariante je self.zamestnanec množina (Set) objektov triedy Osoba. V OCL existuje typ Kolekcia (Collection), ktorý zahŕňa Množinu (Set), UsporiadanúMnožinu (OrderedSet), Hromadu (Bag) a Sekvenciu (Sequence). Tieto typy obsahujú špecifické operácie. K vlastnostiam alebo operáciám kolekcií sa

pristupuje použitím šípky '->'. Viac o týchto typoch a príslušných operáciách si povieme v neskorších sekciách.

V prípade, že koniec asociácie nie je pomenovaný, použijeme na navigáciu typ pri danom konci asociácie, začínajúci s malým písmenom. Takže ak by v predchádzajúcom príklade nebol koniec asociácie nazvaný manažér, prepísali by sme invariant takto:

```
context Firma
inv: self.osoba.maZamestnanie = true
```

Ak by takéto riešenie viedlo k nejednoznačnostiam, vtedy je použitie názvu konca asociácie povinné. Toto sa môže napríklad stať v prípade nepomenovaných reflexívnych asociácií.

Ako sme si už spomenuli, v prípade, že OCL výraz vracia jeden objekt, môžeme s ním pracovať aj ako s Množinou obsahujúcou jeden prvok. To nám umožní využívať operácie zadefinované na Množinách. Nasledujúci príklad ukazuje platný invariant:

```
context Firma
inv: self.manazer->size() = 1
```

5.2.2.1 Navigácia do asociačných tried

Na určenie navigácie do asociačnej triedy použijeme podobný prístup, ako v prípade chýbajúceho mena konca asociácie, teda názov triedy s malým písmenom.

```
context Osoba
inv: self.zamestnanie.plat > 0
```

V prípade rekurzívnej asociácie (asociácia začínajúca aj končiaca v tej istej triede) meno asociačnej triedy nestačí. Potrebujeme totiž rozlíšiť smer, v ktorom je asociácia navigovaná. V našom príklade je takouto triedou Manželstvo. Keď sa chceme navigovať do tejto triedy, máme dve možnosti v závislosti od smeru. Aby sme tento smer vedeli rozlíšiť, pridáme k názvu asociačnej triedy aj názov konca asociácie v tom smere, v ktorom uskutočňujeme navigáciu.

```
context Osoba
inv: self.manzelstvo[manzel].datumUzavretia >
     self.datumNarodenia
```

V tomto výraze sa podvýraz self.manzelstvo[manzel] odkazuje na množinu objektov triedy Manželstvo prislúchajúcich kolekcii manželov.

5.2.2.2 Navigácia z asociačnej triedy

Navigácia z asociačnej triedy prebieha podobne ako klasická navigácia.

```
context Zamestnanie
inv: self.zamestnanec.vek > 18
```

Navigácia z asociačnej triedy smerom k jednému z objektov asociácie sa odkazuje vždy na práve jeden objekt. Vyplýva to z definície asociačnej triedy.

5.2.2.3 Navigácia cez podmienenú asociáciu (qualified association)

Podmienené asociácie používajú jeden alebo viac atribútov triedy na prístup k objektom na opačnom konci asociácie. Na navigáciu cez takúto asociáciu pridáme hodnoty atribútov (qualifiers) do hranatých zátvoriek. V prípade, že nezadáme hodnotu atribútov, výsledkom navigácie bude množina objektov danej triedy.

V nasledujúcom príklade sa pýtame na konkrétny jeden objekt triedy Osoba, s číslom účtu 123456.

```
context Banka
inv: self.zakaznik[123456].vek > 15
```

5.2.3 Atribúty a operácie

V tejto časti si ukážeme tie črty jazyka OCL, ktoré sa týkajú atribútov a operácií tried (prípadne premenných).

5.2.3.1 Lokálne premenné

Na zadefinovanie lokálnej premennej, resp. podvýrazu, ktorý sa v celkovom výraze opakuje viackrát, môžeme použiť kombináciu kľúčových slov **let** a **in**. Premenná musí mať meno, typ a výraz špecifikujúci jej hodnotu. Táto premenná môže byť použitá iba v tomto výraze za kľúčovým slovom **in**.

```
context Osoba
inv: let prijem: Integer = self.zamestnanie.plat->sum()
       in
     if not maZamestnanie then
         prijem < 100
           else
         prijem >= 100
     endif
```

Lokálna premenná (výraz) zadefinovaná pomocou **let** platí len vo výraze, v ktorom bola zadefinovaná. Výrazy zadefinované pomocou **let** môžu byť aj vnorené, ale pohodlnejšie je zadefinovať viac lokálnych premenných a zahrnúť ich do jedného **let** výrazu, v ktorom budú oddelené čiarkou.

5.2.3.2 Dodatočné atribúty a operácie

Atribúty alebo operácie môžu byť definované pomocou OCL výrazu. Definovanie atribútov alebo operácií týmto spôsobom znamená, že každá inštancia typu uvedeného v kontexte, má k dispozícii daný atribút, resp. operáciu. Na takúto definíciu sa použije kľúčové slovo **def**.

Nasledujúci príklad obsahuje definíciu atribútu iniciála, ktorý obsahuje prvé písmeno mena Osoby a definíciu operácie máFunkciu, ktorá zistí, či osoba má funkciu zadanú ako parameter operácie.

```
context Osoba
def: iniciala : String = self.meno.substring(1,1)
def: maFunkciu(f:String):Boolean = self.zamestnanie->
     exists(funkcia = f)
```

5.2.3.3 Počiatočné hodnoty a odvodzovacie pravidlá

V tejto sekcii si popíšeme, aké funkcie môžu mať OCL výrazy, keď kontextom je atribút alebo názov konca asociácie.

Počiatočná hodnota atribútu alebo typu na konci asociácie je hodnota, ktorú bude mať inštancia daného kontextu v momente vytvorenia. Na uvedenie počiatočnej hodnoty sa používa kľúčové slovo **init**.

Syntax inicializácie je nasledovná:

```
context MenoTypu::menoAtributu: Typ
init: -- nejaký výraz reprezentujúci počiatočnú hodnotu
context MenoTypu::nazovKoncaAsociacie: Typ
init: -- nejaký výraz reprezentujúci počiatočnú hodnotu
```

Výraz musí zodpovedať typu atribútu v kontexte. V prípade, že kontextom je koniec asociácie a násobnosť je najviac jedna, výraz musí zodpovedať typu na konci asociácie, ak je násobnosť mnoho, musí výraz zodpovedať Množine alebo UsporiadanejMnožine typu na konci asociácie.

V nasledujúcom príklade je znázornené, že inštancia triedy Osoba v momente jej vytvorenia ešte nemá zamestnanie (nastavili sme počiatočnú hodnotu atribútu maZamestnanie).

```
context Osoba::maZamestnanie: Boolean
```

```
init: false
```

Odvodzovacie pravidlo špecifikuje, že hodnota elementu uvedeného v kontexte (atribút alebo koniec asociácie) musí byť rovnaká, ako hodnota, ktorú získame vyhodnotením odvodzovacieho pravidla. Na použitie odvodzovacieho pravidla sa používa kľúčové slovo **derive**. Syntax je rovnaká ako pri inicializácii, len slovo **init** nahradíme slovom **derive**.

5.2.3.4 Telo dotazovacích operácií

Dotazovacie operácie (query operations) môžu byť zadefinované OCL výrazom špecifikovaním výsledku operácie. Vykonanie dotazovacej operácie vráti hodnotu alebo množinu hodnôt, nezmení stav systému.

Syntax je nasledovná:

```
context MenoTypu::menoOperacie(param1: Typ1,...): NavratovyTyp
body: -- nejaký výraz
```

Príklad zadefinovania operácie prijem triedy Osoba:

```
context Osoba::prijem(datum: Date): int
body: self.zamestnanie.plat->sum()
```

5.2.3.5 Prekondície a postkondície

OCL výraz môže byť súčasťou prekondície (podmienky, ktorá musí platiť pred vykonaním danej operácie) a postkondície (podmienky, ktorá musí platiť po vykonaní danej operácie).

Kľúčové slovo **self** môže byť vo výraze použité ako odkaz na objekt, ktorého operácia bola zavolaná. Kľúčové slovo **result** vyjadruje výsledok operácie, ak nejaký je. Názvy parametrov operácie môžu byť takisto použité vo výraze.

Syntax prekondícií a postkondícií je rovnaká, ako pri definovaní tela dotazovacích operácií, len namiesto slova **body** použijeme kľúčové slovo **pre**, resp. **post**. Je možné uviesť aj názov prekondície, resp. postkondície. Uvádza sa medzi kľúčovým slovom **pre**, resp. **post** a dvojbodkou.

Príklad:

```
context Firma::cenaAkcie(): Real
pre: pocetZamestnancov > 10
post: result > 1000
```

5.2.3.6 Predchádzajúce hodnoty v postkondíciách

Výraz v postkondícii sa môže odkazovať na hodnoty vlastností daného objektu v dvoch časových okamihoch:

- hodnota na začiatku operácie
- hodnota po skončení operácie

Hodnota vlastnosti objektu v postkondícii je hodnota po skončení operácie. Na to, aby sme v postkondícii mohli použiť hodnotu, ktorá bola pred začatím operácie, použijeme slovo **@pre**.

V nasledujúcom príklade je ukázané, že po vykonaní operácie najatPracovnika budú inštancie triedy Firma obsahovať odkaz aj na práve najatú osobu, ktorá bola uvedená ako atribút operácie. Zároveň sa aj zvýši cena akcie. Všimnime si, že pri operáciách musí byť slovo **@pre** uvedené ešte pred zátvorkami prislúchajúcimi k operácii (ako je to v prípade operácie cenaAkcie).

```
context Firma::najatPracovnika(p: Osoba)
post: zamestnanec = zamestnanec@pre->including(p) and
      cenaAkcie() = cenaAkcie@pre() + 100
```

5.2.4 Typy v OCL a príslušné operácie

Typy v OCL sú rozdelené do týchto skupín:

- OCL typy zadefinované v štandardnej knižnici:
  - Základné typy
  - Typy kolekcií
- Užívateľom zadefinované typy

5.2.4.1 Základné typy a ich operácie

Základné typu sú Integer (celé číslo), Real (reálne číslo), String (reťazec) a Boolean.

Boolean

Typ Boolean môže mať len dve hodnoty: `true` (pravda) a `false` (nepravda). V tabuľke 1 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Boolean. Štandardnou operáciou pre typ boolean, ktorá však v mnohých programovacích jazykoch chýba a teda je možno menej známa, je operácia `implies`. Pre túto operáciu platí: ak je hodnota prvého operandu true, potom musí byť aj hodnota druhého operandu true a v tom prípade má celý výraz hodnotu true.

V prípade, že prvý operand má hodnotu false, celý výraz má hodnotu true, nezávisle na hodnote druhého operandu.

Operácia	Popis
or (b: Boolean): Boolean	Vracia true, ak self alebo b je true
xor (b: Boolean): Boolean	Vracia true, ak self alebo b je true, ale nie obe naraz.
and (b: Boolean): Boolean	Vracia true, ak self a zároveň b sú true
not: Boolean	Vracia true, ak self má hodnotu false
implies (b: Boolean): Boolean	Vracia true, ak self je false, alebo ak self je true a zároveň b je true

Tabuľka 1 - Popis operácií definovaných pre typ Boolean

V OCL je možné použiť aj vetvenie podmienok, pomocou konštrukcie **`if-then-else`**. Obidva výrazy v časti then a else musia byť rovnakého typu.

Syntax:

```
if <OCL výraz typu Boolean>
then <OCL výraz>
else <OCL výraz>
endif
```

Integer

Typ Integer predstavuje celé čísla. V tabuľke 2 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Integer.

Operácia	Popis
−: Integer	Záporná hodnota zo self
+ (i: Integer): Integer	Hodnota súčtu self a i
- (i: Integer): Integer	Hodnota rovná odčítaniu i od self
*** (i: Integer): Integer**	Hodnota súčinu self a i
/ (i: Integer): Real	Hodnota self po delení hodnotou i
abs(): Integer	Absolútna hodnota zo self
div(i: Integer): Integer	Počet, koľkokrát sa i nachádza celé v self, resp. celočíselné delenie
mod(i: Integer): Integer	Zvyšok po celočíselnom delení self hodnotou i
max (i: Integer): Integer	Maximum zo self a i
min(i: Integer): Integer	Minimum zo self a i

Tabuľka 2 - Popis operácií definovaných pre typ Integer

Nasledujúce príklady sú typu Boolean a všetky sú pravdivé (vyhodnotené ako true).

```
22.max(33) = 33
15.min(12) = 12
15.mod(2) = 1
15.div(2) = 6
-24.abs() = 24
```

Real

Typ Real predstavuje reálne čísla. V tabuľke 3 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Real.

Operácia	Popis
−: Real	Záporná hodnota zo self
+ (r: Real): Real	Hodnota súčtu self a r
- (r: Real): Real	Hodnota rovná odčítaniu r od self
*** (r: Real): Real**	Hodnota súčinu self a r
/ (r: Real): Real	Hodnota self po delení hodnotou r
abs(): Real	Absolútna hodnota zo self
max(r: Real): Real	Maximum zo self a r
min(r: Real): Real	Minimum zo self a r
floor(): Integer	Najväčšie celé číslo menšie alebo rovné self, resp. dolná celá časť
round(): Integer	Najbližšie celé číslo k self. Ak sú také čísla dve, potom väčšie z nich
< (r: Real): Boolean	Vracia true, ak self je menšie ako r
> (r: Real): Boolean	Vracia true, ak self je väčšie ako r
<= (r: Real): Boolean	Vracia true, ak self je menšie alebo rovné r
>= (r: Real): Boolean	Vracia true, ak self je väčšie alebo rovné r

Tabuľka 3 - Popis operácií definovaných pre typ Real

Nasledujúce príklady sú typu Boolean a všetky sú pravdivé (vyhodnotené ako true).

```
10 * 4.3 + 1.1 = 44.1
(5.2).floor() = 5
(-8.9).abs() = 8.9
12 > 22.7 = false
```

String

Typ String predstavuje textové reťazce. Tieto sú uzavreté medzi apostrofmi, napr. ‘toto je textový reťazec’. V tabuľke 4 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ String.

Operácia	Popis
size(): Integer	Počet písmen v self
concat(s: String): String	Spojenie reťazcov self a s do jedného reťazca
substring(lower: Integer, upper: Integer): String	Podreťazec self začínajúci písmenom na pozícii lower a končiaci písmenom na pozícii upper (vrátane). Číslovanie písmen je od 1 po self.size()
toInteger(): Integer	Skonvertuje self na typ Integer (ak to je možné)
toReal(): Real	Skonvertuje self na typ Real (ak to je možné)

Tabuľka 4 - Popis operácií definovaných pre typ String

Nasledujúce príklady sú typu Boolean a všetky sú pravdivé (vyhodnotené ako true).

```
'OCL'.size() = 3
'meno '.concat('a priezvisko') = 'meno a priezvisko'
'metamodelovanie'.substring(5, 9) = 'model'
```

5.2.4.2 Typy kolekcií a ich operácie

V objektovo-orientovaných systémoch je veľmi bežná práca s kolekciami objektov. V OCL existuje 5 typov kolekcií. Štyri z nich sú konkrétne typy a môžu byť použité vo výrazoch. Sú to Set (Množina), OrderedSet (UsporiadanáMnožina), Bag (Hromada) a Sequence (Sekvencia). Piaty typ je Collection (Kolekcia). Je to abstraktný typ, rodičovský typ pre ostatné štyri a je použitý na zadefinovanie operácií, ktoré sú spoločné pre zvyšné štyri typy kolekcií.

Definícia konkrétnych typov:

- **Set** – neobsahuje duplicitné elementy (každá inštancia je tam najviac raz). Elementy v tejto kolekcii nie sú usporiadané.
- **OrderedSet** – je to Set, elementy ktorej sú usporiadané.
- **Bag** – môže obsahovať duplicitné elementy, teda tá istá inštancia sa tu môže vyskytovať viackrát. Tento typ je obvykle výsledkom navigovania sa cez viac asociácií. Elementy v tejto kolekcii nie sú usporiadané.
- **Sequence** – je to Bag, elementy ktorej sú usporiadané.

Teraz si popíšeme operácie pre jednotlivé typy kolekcií.

Collection

V tabuľke 5 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Collection.

Operácia	Popis
size(): Integer	Počet elementov v kolekcii self
includes(object: T): Boolean	Vracia true, ak object patrí do kolekcie self, inak vracia false
excludes(object: T): Boolean	Vracia true, ak object nepatrí do kolekcie self, inak vracia false
count(object: T): Integer	Počet výskytov object v kolekcii self
includesAll(c: Collection(T)): Boolean	Vracia true, ak kolekcia self obsahuje všetky elementu kolekcie c
excludesAll(c: Collection(T)): Boolean	Vracia true, ak kolekcia self neobsahuje ani jeden element kolekcie c
isEmpty(): Boolean	Vracia true, ak self je prázdna kolekcia
notEmpty(): Boolean	Vracia true, ak self je neprázdna kolekcia
sum():T	Súčet elementov v kolekcii self. Elementy musia byť typu, ktorý podporuje operáciu +. Intefer a Real sú typickým typom, ktorý môže použiť túto operáciu.
product(c2: Collection(T2)) : Set(Tuple(first: T, second: T2))	Kartézsky súčin kolekcie self a c

Tabuľka 5 – Popis operácií definovaných pre typ Collection

Set

V tabuľke 6 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Set.

Operácia	Popis
union(s: Set(T)): Set(T)	Zjednotenie množín self a s
union(bag: Bag(T)): Bag(T)	Zjednotenie self a bag
= (s: Set(T)): Boolean	Vracia true, ak self a s obsahujú rovnaké elementy
intersection(s:Set(T)):Set(T)	Prienik self a s, teda množina elementov, ktoré sa nachádzajú súčasne v self aj v s
intersection(bag: Bag(T)):Set(T)	Prienik self a bag
- (s: Set(T)): Set(T)	Tie elementy self, ktoré nie sú v s

including(object: T): Set(T)	Množina obsahujúca všetky elementy self plus object
excluding(object: T): Set(T)	Množina obsahujúca všetky elementy self okrem object
symmetricDifference(s:Set(T)):Set(T)	Množina obsahujúca všetky elementy nachádzajúce sa v self alebo s, ale nie v oboch naraz.(zjednotenie mínus prienik)
flatten():Set(T2)	Zmení kolekciu kolekcií na jednoduchú kolekciu, ktorej elementy sú už priamo objekty (nie kolekcie objektov)
asSet(): Set(T)	Množina identická so self
asSequence(): Sequence(T)	Sekvencia (Sequence), ktorá obsahuje všetky elementy self
asBag(): Bag(T)	Hromada (Bag), ktorá obsahuje všetky elementy self
asOrderedSet() : OrderedSet(T)	Usporiadaná Množina, ktorá obsahuje všetky elementy self

Tabuľka 6 - Popis operácií definovaných pre typ Set

Príklad operácie flatten:

```
Set { Set { 1, 2 }, Set { 2, 3 }, Set { 4, 5, 6 } }->flatten()
=
Set { 1, 2, 3, 4, 5, 6 }
```

OrderedSet

V tabuľke 7 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ OrderedSet.

Operácia	Popis
append (object: T) : OrderedSet(T)	Množina elementov self, za ktorými nasleduje object
prepend(object : T) : OrderedSet(T)	Usporiadaná Množina, pozostávajúca z object, za ktorým nasledujú elementy self
insertAt(index : Integer, object : T) : OrderedSet(T)	Množina pozostávajúca z elementov self, v ktorej je object vložený na pozíciu index
subOrderedSet(lower : Integer, upper : Integer) : OrderedSet(T)	Podmnožina self začínajúca na čísle lower a končiaca na čísle upper vrátane
at(i : Integer) : T	Element zo self nachádzajúci sa na pozícii i
indexOf(obj : T) : Integer	Index objektu object v množine self
first() : T	Prvý element zo self
last() : T	Posledný element zo self

Tabuľka 7 - Popis operácií definovaných pre typ OrderedSet

Bag

V tabuľke 8 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Bag.

Operácia	Popis
union(set: Set(T)): Bag(T)	Zjednotenie množín self a set
union(bag: Bag(T)): Bag(T)	Zjednotenie self a bag
= (bag : Bag(T)) : Boolean	Vracia true, ak self a bag obsahujú rovnaké elementy, rovnaký počet krát.
intersection(set:Set(T)):Set(T)	Prienik self a set
intersection(bag: Bag(T)): Bag(T)	Prienik self a bag
including(object: T): Bag(T)	Hromada (Bag) obsahujúca všetky elementy self plus object
excluding(object: T): Bag(T)	Hromada (Bag) obsahujúca všetky elementy self okrem výskytov object
flatten():Bag(T2)	Zmení kolekciu kolekcií na jednoduchú kolekciu, ktorej elementy sú už priamo objekty (nie kolekcie objektov)
asSet(): Set(T)	Množina obsahujúca všetky elementy zo self, duplikáty sú odstránené
asSequence(): Sequence(T)	Sekvencia (Sequence), ktorá obsahuje všetky elementy self
asBag(): Bag(T)	Hromada (Bag) identická so self
asOrderedSet() : OrderedSet(T)	Usporiadaná Množina, ktorá obsahuje všetky elementy self, duplikáty sú odstránené

Tabuľka 8 - Popis operácií definovaných pre typ Bag

Príklad operácie flatten na Bag:

```
Bag { Set { 1, 2 }, Set { 1, 2 }, Set { 4, 5, 6 } }->flatten()
=
Bag { 1, 1, 2, 2, 4, 5, 6 }
```

Sequence

V tabuľke 9 sú uvedené všetky operácie definované v OCL štandardnej knižnici pre typ Sequence.

Operácia	Popis
append (object: T) : Sequence(T)	Sekvencia elementov self, za ktorými nasleduje object

prepend(object : T) : Sequence(T)	Sekvencia, pozostávajúca z object, za ktorým nasledujú elementy self
insertAt(index : Integer, object : T) : Sequence(T)	Sekvencia pozostávajúca z elementov self, v ktorej je object vložený na pozíciu index
subSequence(lower : Integer, upper : Integer) : Sequence(T)	Podsekvencia self začínajúca na čísle lower a končiaca na čísle upper vrátane
at(i : Integer) : T	Element zo self nachádzajúci sa na pozícii i
indexOf(obj : T) : Integer	Index objektu object v sekvencii self
first() : T	Prvý element zo self
last() : T	Posledný element zo self
= (s : Sequence(T)) : Boolean	Vracia true, ak self obsahuje také isté elementy ako s v tom istom poradí
union (s : Sequence(T)) : Sequence(T)	Sekvencia (Sequence) pozostávajúca zo všetých elementov self, za ktorými nasledujú všetky elementy s
flatten() : Sequence(T2)	Zmení kolekciu kolekcií na jednoduchú kolekciu, ktorej elementy sú už priamo objekty (nie kolekcie objektov)
including(object : T) : Sequence(T)	Sekvencia pozostávajúca zo všetkých elementov self plus object je pridaný ako posledný element
excluding(object : T) : Sequence(T)	Sekvencia pozostávajúca zo všetkých elementov self okrem všetkých výskytov object
asBag() : Bag(T)	Hromada obsahujúca všetky elementy zo self, zahŕňa aj duplikáty
asSequence() : Sequence(T)	Sekvencia identická so self
asSet() : Set(T)	Množina obsahujúca všetky elementy zo self, duplikáty sú odstránené
asOrderedSet() : OrderedSet(T)	Usporiadaná Množina obsahujúca všetky elementy zo self v tom istom poradí, duplikáty sú odstránené

Tabuľka 9 - Popis operácií definovaných pre typ Sequence

5.2.4.3 Iterovacie operácie

OCL definuje niekoľko operácií, ktoré umožňujú iteráciu nad elementmi v kolekcii. Tieto operácie vezmú postupne každý element v kolekcii a vyhodnotia na ňom príslušný výraz. Každá takáto operácia dostane OCL výraz ako parameter, tento parameter sa nazýva body. Zároveň tieto operácie môžu mať nepovinný parameter iterator. Premenná iterator je potom použitá vo výraze body na označenie elementu z kolekcie, pre ktorý je výraz body vyhodnocovaný. Typ tejto premennej je vždy typ elementov v kolekcii, preto môže, ale nemusí byť uvedený. Uvedenie typu môže sprehľadniť daný OCL výraz.

Collection

V tabuľke 10 sú uvedené všetky iterovacie operácie definované v OCL štandardnej knižnici pre typ Collection.

Operácia	Notácia	Popis
exists	**source->exists(iterators | body)**	Vracia true, ak sa výraz body vyhodnotí ako true aspoň pre jeden element kolekcie source
forAll	**source->forAll(iterators | body)**	Vracia true, ak sa výraz body vyhodnotí ako true pre všetky elementy kolekcie source
isUnique	**source->isUnique(iterators | body)**	Vracia true, ak výraz body vracia rôzne hodnoty pre každý element kolekcie source
sortedBy	**source->sortedBy(iterator | body)**	Sekvencia obsahujúca všetky elementy kolekcie source. Element kolekcie source, pre ktorý ma výraz body najmenšiu hodnotu, bude vo výsledku prvý atď. Typ výrazu body musí mať definovanú operáciu <. Operácia < musí vracať Boolean a musí byť tranzitívna, t.j. platí, že ak a<b a b<c, tak a<c. Operácia sortedBy môže mať najviac jednu premennú iterator.
any	**source->any(iterator | body)**	Vráti element z kolekcie source, pre ktorý výraz body vráti true. Ak je takýchto elementov viac, operácia any vráti len jeden z nich. Musí byť však aspoň jeden takýto element, inak je výsledkom nedefinovaný typ OclUndefined. Operácia any môže mať najviac jednu premennú iterator.
one	**source->one(iterator | body)**	Vráti hodnotu true, ak existuje práve jeden element v kolekcii source, pre ktorý výraz body vráti true. Operácia one môže mať najviac jednu premennú iterator.
collect	**source->collect(iterators | body)**	Kolekcia elementov, ktorá vznikne aplikovaním výrazu body na každý element kolekcie source. Na výsledok je ešte použitá operácia flatten().

Tabuľka 10 - Popis iterovacích operácií definovaných pre typ Collection

Set

V tabuľke 11 sú uvedené všetky iterovacie operácie definované v OCL štandardnej knižnici pre typ Set.

Operácia	Notácia	Popis
select	**source->select(iterator | body)**	Výsledkom sú elementy množiny source, pre ktoré sa výraz body vyhodnotí na true. Operácia select môže mať najviac jednu premennú iterator.
reject	**source->reject(iterator | body)**	Výsledkom sú elementy množiny source, pre ktoré sa výraz body vyhodnotí na false. Operácia reject môže mať najviac jednu premennú iterator.
collectNested	**source->collect(iterators | body)**	Výsledkom je Bag, pozostávajúca z elementov, ktoré vzniknú aplikovaním výrazu body na každý element množiny source.

Tabuľka 11 - Popis iterovacích operácií definovaných pre typ Set

Pre typy **Bag** a **Sequence** sú zadefinované presne tie isté operácie ako pre Set, rozdiel je len v návratových typoch. Typ **OrderedSet** v špecifikácii spomenutý síce nie je, ale predpokladáme, že pre tento typ existujú tie isté operácie, ako pre zvyšné typy. Špecifikácia je totiž rozsiahly dokument, ktorý sa postupne vyvíja a je celkom možné, že autori na typ OrderedSet pri popisovaní iterovacích operácií zabudli.

V nasledujúcom príklade je ukážka použitia iterovacej operácie forAll. Daný OCL výraz vyjadruje, že všetci zamestnanci firmy musia mať aspoň 18 rokov. Všetky tri invarianty sú totožné, rozdiel je len v notácii, ktorá môže mať práve tieto tri podoby. Prvý invariant má kompletnú notáciu, je uvedená iterovacia premenná z a jej typ, oddeľujúca čiara '|' a výraz (body) aplikovaný na všetky inštancie typu osoba. Druhý invariant neuvádza typ, to však nevadí, lebo je implicitný. Vyplýva to z toho, že operácia forAll je aplikovaná na kolekciu zamestnancov, a tí sú typu Osoba. Tretí invariant používa najkratšiu notáciu, kde je uvedený už len výraz, ktorý sa vyhodnocuje pre všetky príslušné inštancie. Tieto tri druhy notácie platia pre všetky iterovacie operácie, nielen pre forAll.

```
context Firma
inv: self.zamestnanec->forAll(z: Osoba | z.vek >= 18)
inv: self.zamestnanec->forAll(z | z.vek >= 18)
inv: self.zamestnanec->forAll( vek >= 18 )
```

5.2.4.4 Užívateľom zadefinované typy

Užívateľom zadefinované typy vyplývajú vždy z UML modelov. Všetky elementy UML modelu sú typy pre OCL výrazy aplikované na tento model.

5.2.4.5 Nedefinované hodnoty

Niektoré OCL výrazy budú mať po vyhodnotení nedefinovanú hodnotu. To sa môže stať napr. pri snahe získať prvý element (`->first()`) prázdnej kolekcie. Vo všeobecnosti platí, že výraz, v ktorom jeho nejaká časť je nedefinovaná, bude mať nedefinovanú hodnotu. Avšak existujú aj výnimky tohto pravidla.

Prvá výnimka sa týka logických operátorov (na type Boolean):

- true OR hocičo bude mať hodnotu true
- false AND hocičo bude mať hodnotu false
- false IMPLIES hocičo bude mať hodnotu true
- hocičo IMPLIES true bude mať hodnotu true

Pravidlá pre OR a AND sú platné nezávisle od poradia argumentov.

Ďalšou výnimkou je výraz IF. Bude platný (definovaný) v prípade, že vybraná vetva (then alebo else, podľa výrazu za IF) je platná (definovaná).

Existuje aj operácia na testovanie, či hodnota výrazu je alebo nie je definovaná. Je to operácia **`oclIsUndefined()`**, ktorá vráti true, ak jej argumenty sú nedefinované, inak vráti false.

5.2.4.6 Zoznam hodnôt (Enumeration)

Tento typ sme si už popísali v sekcii 5.1.1.4. Definuje prípustné hodnoty pre tento typ. V našom ukážkovom príklade sa nachádza aj enumeration Pohlavie. Definuje dve možné hodnoty, muž a žena.

Nasledujúci príklad ukazuje použitie tohto typu. Invariant tvrdí, že osoby musia byť muži:

```
context Osoba
inv: pohlavie = Pohlavie::muz
```

5.2.4.7 Zisťovanie typov

Na zisťovanie typov existujú v OCL dve operácie: **`oclIsTypeOf(type: OclType)`** a **`oclIsKindOf(type: OclType)`**. Ich popis je v tabuľke 12.
.

Operácia	Popis
`oclIsTypeOf(type: OclType)`	Vracia true, ak sú typy self a type rovnaké.
`oclIsKindOf(type: OclType)`	Vracia true, ak sú typy self a type rovnaké, alebo ak typ self je potomkom typu type.

Tabuľka 12 - Popis operácií na zisťovanie typov

Príklad:

```
context Osoba
inv: self.oclIsTypeOf( Osoba ) – vráti true
inv: self.oclIsTypeOf( Firma ) -- vráti false
```

5.2.5 Kľúčové slová

Kľúčové slová v OCL sú rezervované slová. To znamená, že sa nemôžu vyskytovať v OCL výrazoch ako meno balíka, typu alebo vlastnosti. Nasledovný zoznam obsahuje kľúčové slová jazyka OCL:

- **and**
- **attr**
- **context**
- **def**
- **else**
- **endif**
- **endpackage**
- **if**
- **implies**
- **in**
- **inv**
- **let**
- **not**
- **oper**
- **or**
- **package**
- **post**
- **pre**
- **then**
- **xor**

5.2.6 Precedencia operátorov

Poradie precedencií operátorov, začínajúc od najvyššej precedencie, je nasledovné:

- **@pre**
- **dot a arrow operátory: ‘.’ a ‘->’**
- **unárne ‘not’ a unárne mínus ‘-’**
- **‘*’ a ‘/’**
- **‘+’ a binárne ‘-’**
- **‘if-then-else-endif’**
- **‘<’, ‘>’, ‘<=’, ‘>=’**
- **‘=’, ‘<>’**
- **‘and’, ‘or’ a ‘xor’**
- **‘implies’**

Zátvorky ‘(’ a ‘)’ môžu byť použité na zmenu precedencie.

6 Vlastný návrh riešenia

V tejto kapitole sa zameriame na vyriešenie spomínaných problémov a na dosiahnutie stanovených cieľov. Najprv vytvoríme dva UML profily. V časti 6.1 popíšeme biznis profil na biznis modelovanie a modelovanie biznis stratégie. V časti 6.2 uvedieme profil na modelovanie požiadaviek. Tieto dva profily spolu súvisia, a spoločne poskytujú spôsob, akým môžu analytici vytvárať svoje modely unifikovaným spôsobom, za použitia entít, ktoré popisujú modelovanú doménu lepšie, ako štandardné UML elementy. Týmto vyriešime problém s unifikáciou modelov a splníme stanovený cieľ unifikácie analytických modelov v organizácii.

Ďalej sa zameriame viac na modelovanie biznis stratégie. V časti 6.3 si popíšeme jednotlivé modely, ktoré sú súčasťou tohto modelovania. Pre lepšie pochopenie uvedieme v časti 6.4 ukážkový príklad z praxe, na ktorom si lepšie vysvetlíme celé modelovanie biznis stratégie.

V časti 6.5 uvedieme OCL podmienky, ktoré spresňujú entity zúčastňujúce sa modelovania biznis stratégie. V časti 6.6 si nakoniec povieme niečo o možnostiach validácie týchto modelov.

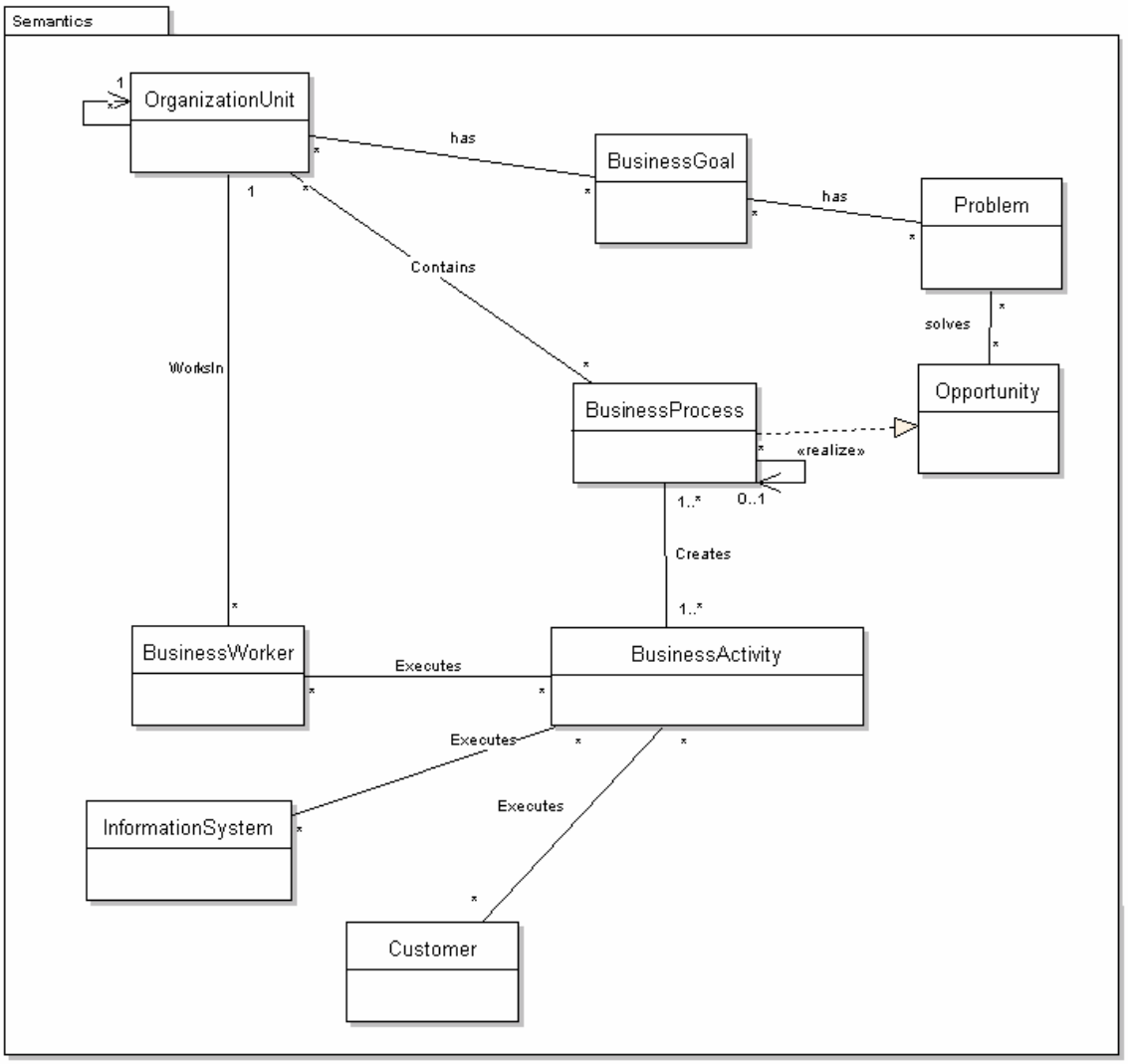
Touto kapitolou teda riešime všetky spomínané problémy a splníme všetky stanovené ciele.

6.1 Biznis profil

Biznis profil obsahuje entity, ktoré sa vyskytujú pri biznis modelovaní a modelovaní biznis stratégie. Použitím tohto profilu môže analytik využívať vo svojich modeloch entity zadefinované v tomto profile. Výstupy analytikov budú teda lepšie popisovať oblasť modelovania ako štandardné UML entity a zároveň sa zavedie určitý štandard, ako postupovať pri biznis modelovaní a modelovaní biznis stratégie.

Najprv si popíšeme sémantiku profilu, teda metamodel, ktorý je znázornený na obr. 14. Sémantika znázorňuje, aké sú medzi jednotlivými entitami vzťahy.

6.1.1 Sémantika (metamodel)



Obrázok 14 – Sémantika (metamodel) biznis profilu

Teraz si popíšeme jednotlivé elementy nachádzajúce sa v metamodeli biznis profilu:

OrganizationUnit (Organizačná jednotka)

Entita zastupujúca organizačnú jednotku, ktorá definuje rozsah biznis procesov prebiehajúcich na organizačnej jednotke organizácie. Môže sa skladať z ďalších organizačných jednotiek. Organizačná jednotka má zadefinované určité ciele (Business Goal), ktoré sleduje a chce splniť.

BusinessProcess (Biznis proces)

Biznis procesy sa vykonávajú na úrovni organizácie, alebo na úrovni organizačných jednotiek, pričom samotný biznis proces sa môže skladať z ďalších biznis procesov (nižšia úroveň pohľadu). Biznis procesy pozostávajú z biznis aktivít (Business Activity). Biznis proces môže realizovať Príležitosť (Opportunity), teda špecifikovať riešenie.

BusinessActivity (Biznis aktivita)

Biznis procesy na najnižšej úrovni abstrakcie sú tvorené biznis aktivitami, ktoré vykonávajú konkrétni pracovníci v organizácii v jednotlivých organizačných jednotkách. Každá biznis aktivita, ktorá je závislá od práce s informačným systémom, je prípad použitia.

BusinessWorker (Biznis pracovník)

Biznis pracovník (zamestnanec) je entita používajúca sa pri špecifikácii pracovníkov organizácie, pričom podmienka tejto špecifikácie je, že tento pracovník priamo pracuje s informačným systémom.

InformationSystem (Informačný systém)

Informačný systém, ktorý zabezpečuje časť biznis procesu. Vykonáva teda biznis aktivitu a poskytuje údaje alebo odoberá údaje pre iné biznis aktivity, resp. iniciuje iné biznis aktivity.

Customer (Zákazník)

Zákazník je entita používajúca sa pri špecifikácii zákazníkov, pričom podmienka tejto špecifikácie je, že zákazník pracuje s informačným systémom spoločnosti.

BusinessGoal (Biznis cieľ)

Biznis cieľ je entita prislúchajúca organizačnej jednotke. Definuje ciele organizácie, resp. organizačnej jednotky.

Problem (Problém)

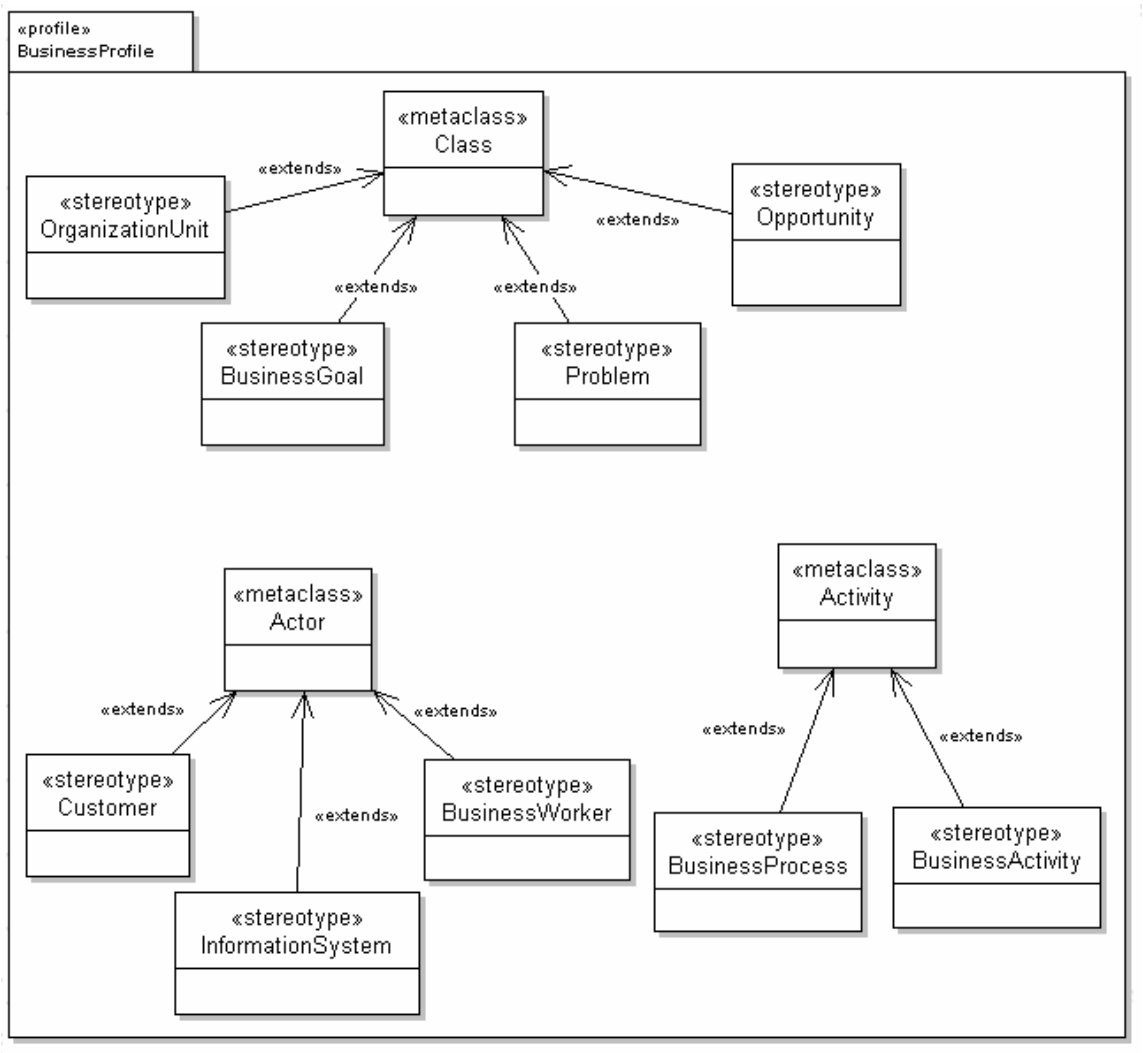
Problém je entita prislúchajúca biznis cieľu. Popisuje, čo bráni splneniu daného biznis cieľa.

Opportunity (Príležitosť)

Príležitosť prislúcha problému. Popisuje možnosť vyriešenia daného problému, a tým pádom dosiahnutie stanoveného biznis cieľa. Opportunita môže byť (v rámci tohto profilu) realizovaná biznis procesom. V profile na modelovanie požiadaviek si ukážeme, že môže byť realizovaná aj prípadom použitia.

6.1.2 Syntax

Keď už máme vytvorený metamodel, môžeme následne vytvoriť samotný UML biznis profil rozšírením príslušných metatried jazyka UML.



Obrázok 15 – Syntax biznis profilu

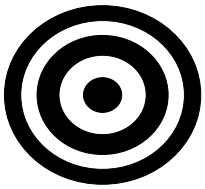
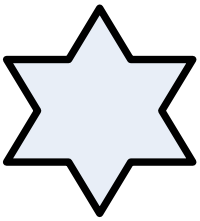
Biznis profil teda definuje 9 nových stereotypov. Sú to presne tie, ktoré sa nachádzajú v metamodeli popísanom v predchádzajúcej sekcii. V tabuľke 13 je zoznam týchto stereotypov, a príslušná metatrieda, ktorú stereotyp rozširuje.

Stereotyp	Metatrieda
OrganizationUnit	Class
BusinessGoal	Class
Problem	Class
Opportunity	Class
Customer	Actor

InformationSystem	Actor
BusinessWorker	Actor
BusinessProcess	Activity
BusinessActivity	Activity

Tabuľka 13 – Zoznam stereotypov definovaných v biznis profile

Pre nasledovné stereotypy sme sa rozhodli použiť osobitnú ikonku, zoznam týchto stereotypov a príslušných ikoniek je uvedený v tab....

Stereotyp	Ikonka
BusinessGoal	
Problem	
Opportunity	

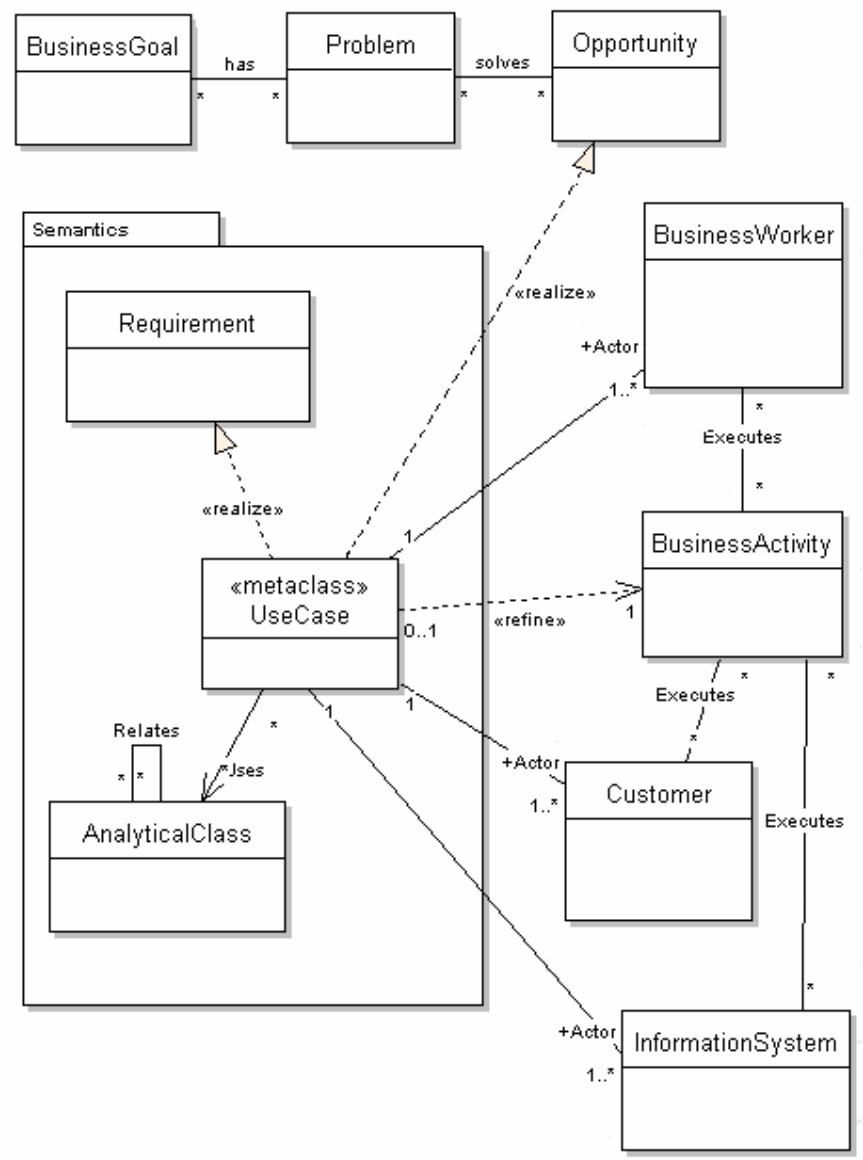
Tabuľka 14 – Zoznam stereotypov so špeciálnou grafickou ikonkou

6.2 Profil na modelovanie požiadaviek

Tento profil obsahuje entity, ktoré sa vyskytujú pri zbere požiadaviek, ich modelovaní a modelovaní funkcionality systému. Použitím tohto profilu môže analytik využívať vo svojich modeloch entity zadefinované v tomto profile

Správne modelovanie požiadaviek je základom úspešnej aplikácie. Pri konzultáciách so zákazníkom je potrebné dávať dôraz na funkcionalitu, teda na to, čo má systém umožniť, nie na to, ako to má systém poskytnúť.

6.2.1 Sémantika



Obrázok 16 – Sémantika profilu na modelovanie požiadaviek

Teraz si popíšeme jednotlivé elementy nachádzajúce sa v metamodeli profilu na modelovanie požiadaviek:

Requirement (Požiadavka)

Entita Požiadavka zastupuje požiadavku všeobecnej funkcionality. Táto požiadavka bude v systéme realizovaná prípadom použitia.

AnalyticalClass (Analytická trieda)

Entita Analytická trieda zastupuje logický objekt, teda statickú formu prípadu použitia. Ide v podstate o špeciálny prípad klasickej triedy (Class).

Tým sme si popísali nové entity, ktoré tento profil pridal k už existujúcim entitám. V sémantike profilu sú však znázornené aj ďalšie entity, konkrétne metatrieda UseCase (prípad použitia) a entity z biznis profilu.

Keďže entity BusinessWorker, Customer a InformationSystem sú vlastne aktéri (rozširujú metatriedu Actor), sú automaticky vo vzťahu s metatriedou UseCase, vyplýva to z UML metamodelu.

UseCase (Prípad použitia)

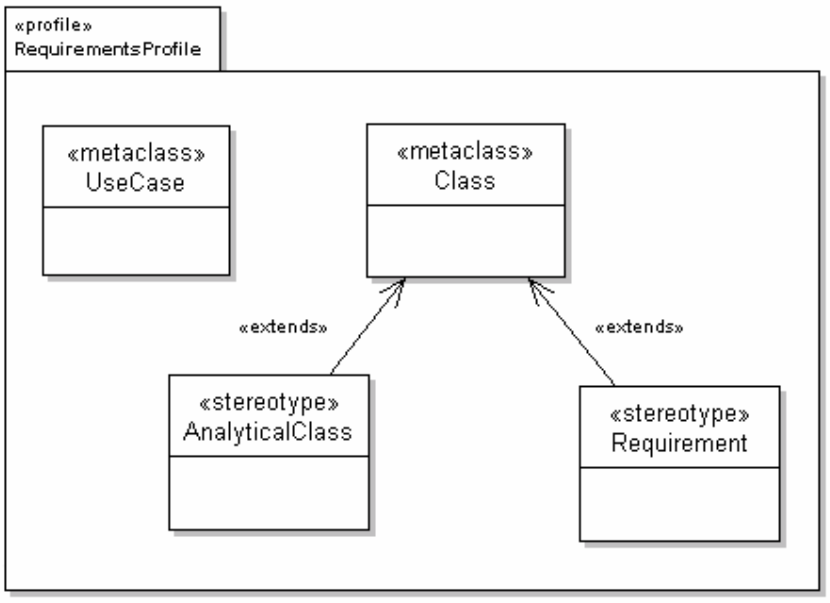
Vysvetlíme si teraz sémantiku prípadu použitia v kontexte tohto profilu. V prvom rade prípad použitia reprezentuje funkcionalitu systému. Preto realizuje Požiadavku (Requirement) a Príležitosť (Opportunity).

Ďalej si môžeme všimnúť, že spresňuje biznis aktivitu. To znamená, že kedykoľvek nejaká aktivita v rámci biznis procesu súvisí s práve vyvíjaným softvérovým systémom, môžeme túto aktivitu spresniť, resp. nahradiť prípadom použitia. Takto získame ďalšiu potrebnú funkcionalitu systému, ktorá sa nám prirodzene ukáže pri modelovaní biznis procesov organizácie, resp. organizačnej jednotky. Zároveň tak napomôžeme tomu, aby informačný systém lepšie vyhovoval skutočným potrebám organizácie.

Prípad použitia takisto využíva Analytickú triedu, ktorá je určitým krokom medzi popisom potrebných funkcionalít v podobe prípadov použitia, a medzi konkrétnou realizáciou danej požiadavky v podobe návrhových tried.

6.2.2 Syntax

Z metamodelu opäť vytvoríme samotný UML profil na modelovanie požiadaviek.



Obrázok 17 – Syntax profilu na modelovanie požiadaviek

Profil na modelovanie požiadaviek definuje 2 stereotypy. Tieto stereotypy, spolu s metatriedou, ktorú rozširujú, sú uvedené v tabuľke....

Stereotyp	Metatrieda
Requirement	Class
AnalyticalClass	Class

Tabuľka 15 – Zoznam stereotypov zadefinovaných v profile na modelovanie požiadaviek

6.3 Modelovanie biznis stratégie

V ďalšom texte sa už budeme zameriavať len na modelovanie biznis stratégie. Za týmto účelom si potrebujeme všímať nasledovné entity:

- z **biznis profilu**: BusinessGoal, Problem, Opportunity, BussinessProcess
- z **profilu na modelovanie požiadaviek**: UseCase (aj keď samozrejme UseCase je metatrieda z UML metamodelu, myslíme tu však na kontext použitia)

Na modelovanie biznis stratégie sa používajú nasledovné modely:

- Model cieľov
- Celkový model stratégie
- Model problémov
- Model príležitostí

6.3.1 Model cieľov

Prvé, čo pri modelovaní biznis stratégie zákazníka musíme spraviť, je určiť jeho strategické ciele. Následne vytvoríme model, v ktorom budú tieto ciele znázornené spolu so závislosťami medzi týmito cieľmi. To nám umožní pochopiť, ako so sebou jednotlivé ciele súvisia a pomôže nám to určiť, ktoré ciele potrebujeme riešiť najskôr.

6.3.2 Celkový model stratégie

Tento model je zo všetkých najkomplexnejší. Po tom, čo sme vytvorili model cieľov, môžeme z neho plynule vytvoriť celkový model stratégie. K jednotlivým cieľom pridáme problémy, ktoré bránia naplneniu daného cieľa. S problémami spojíme príležitosti, ktoré reprezentujú možnosti, ako vyriešiť daný problém a dosiahnuť tak sledovaný cieľ. No a nakoniec nasleduje namodelovanie samotného riešenia, v podobe realizácie príležitostí biznis procesmi alebo prípadmi použitia.

6.3.3 Model problémov

Na základe celkového modelu vytvoríme osobitný model problémov, v ktorom znázorníme závislosti medzi týmito problémami.

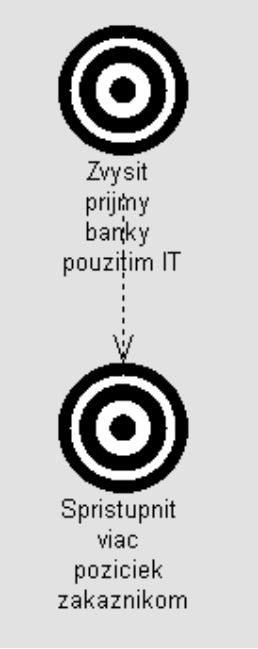
6.3.4 Model príležitostí

Podobne ako v modeli problémov, aj tu na základe celkového modelu vytvoríme model príležitostí, v ktorom znázorníme závislosti medzi existujúcimi príležitosťami.

6.4 Príklad modelovania biznis stratégie

Na nasledujúcom zjednodušenom príklade si ukážeme, ako vyzerá také modelovanie biznis stratégie. Klient, v tomto prípade požiada softvérovú firmu o pomoc. Cieľ, ktorý by chceli dosiahnuť, je zvýšenie tržieb prostredníctvom informačného systému. Po konzultáciách s klientom sa objaví ešte jeden, konkrétnejší cieľ. Banka by chcela sprístupniť viac pôžičiek svojim klientom. Analytik na základe tohto popisu môže hneď vytvoriť model cieľov (možno už priamo u klienta počas konzultácií). Tento model je znázornený na obr. 18.

Keď už je model cieľov hotový, môže začať analytik rozvíjať a vytvoriť tak celkový model stratégie. Najprv potrebuje určiť existujúce problémy, ktoré bránia splneniu cieľov. Po príslušnej analýze sa zistilo, že neexistuje jednoznačná koncepcia poskytovania úverov a takisto, že schvaľovací proces je pomalý. Pri analýze schvaľovacieho procesu sa zistili ďalšie problémy, konkrétne neefektívna skóringová formula na schvaľovanie pôžičiek a s tým súvisel aj problém neefektívnej detekcie podvodníckych žiadateľov o pôžičku (takí žiadatelia, ktorí nemajú v pláne splácať, ktorí špekulujú).

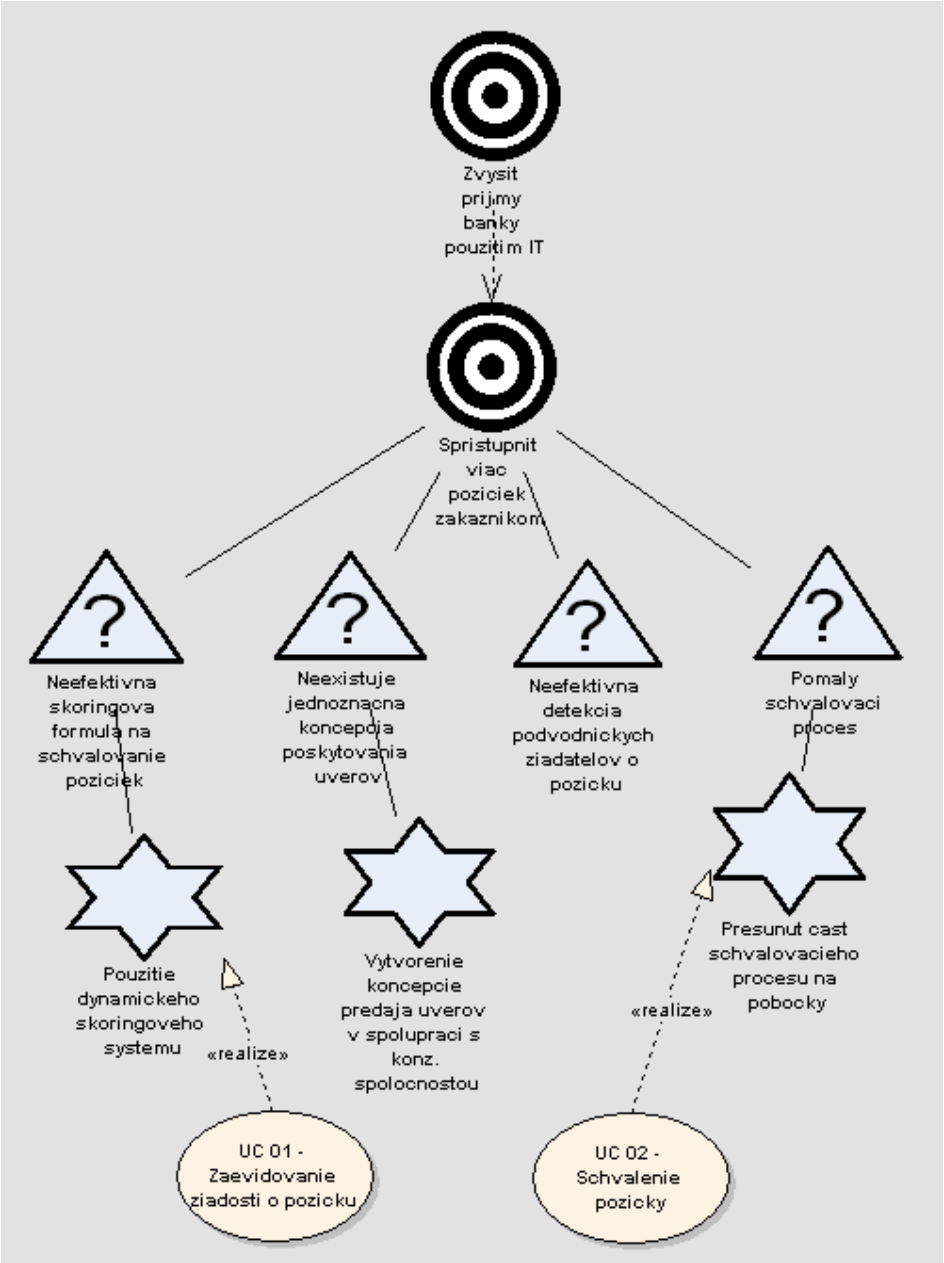


Obrázok 18 – Príklad na model cieľov

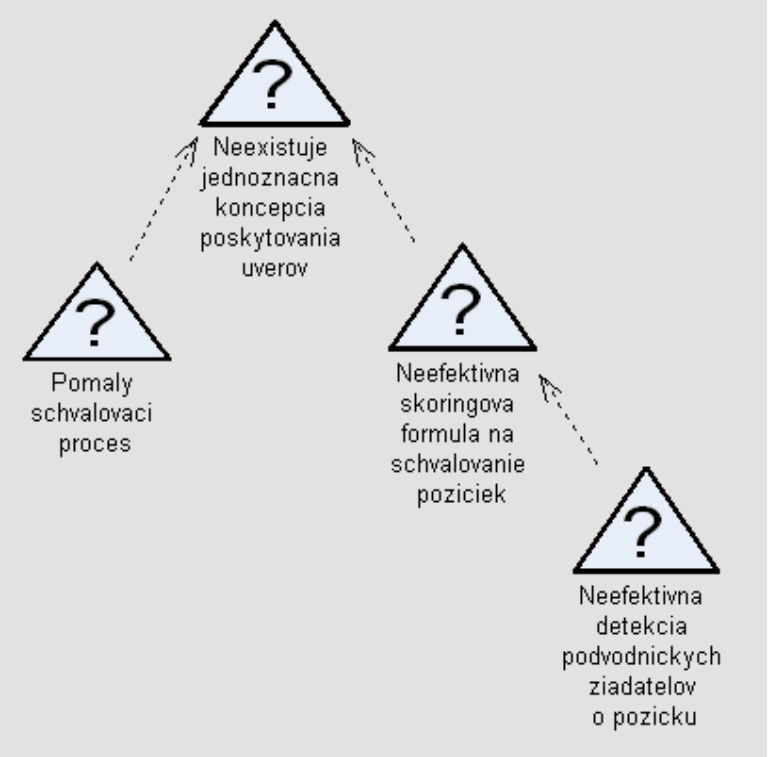
Po určení problémov je možné pokračovať v analýze nájdením príležitostí. Ako prvá príležitosť sa načrtlo vytvorenie poriadnej koncepcie poskytovania úverov za pomoci konzultantskej spoločnosti. Ďalej sa navrhlo urýchliť schvaľovací proces tým, že sa presunie istá časť schvaľovania (do určitej sumy) na pobočky. Takisto sa banka rozhodla pre použitie dynamického skóringového systému, ktorý by umožnil lepšie vyhodnocovanie možných kandidátov na pôžičky.

Na záver už je možné začať modelovať funkcionalitu informačného systému. Všetky príležitosti, ktoré bude riešiť informačný systém, môžeme vyjadriť v podobe prípadov použitia, prípadne v podobe biznis procesov (ktoré pozostávajú z biznis aktivít, a aktivity, ktoré súvisia s používaním informačného systému, sa stanú prípadmi použitia). Znázorníme to prostredníctvom realizácie, čiže príslušný prípad použitia, resp. biznis proces, bude realizovať danú príležitosť. Celkový model stratégie je znázornený na obr. 19.

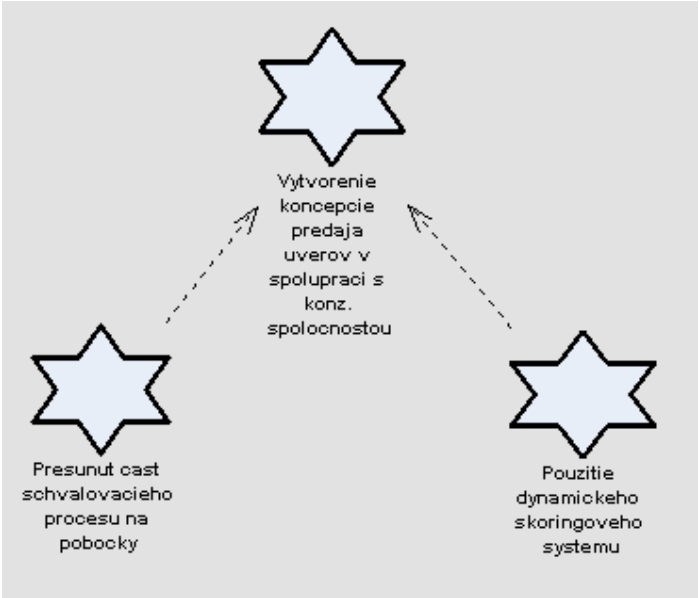
Zostáva nám ešte vytvoriť modely problémov a príležitostí. V týchto modeloch sa budú vyskytovať tie problémy, resp. príležitosti, ktoré sa nachádzajú aj v celkovom modeli stratégie. Na týchto modeloch sa znázorní závislosť medzi problémami, resp. príležitosťami. To umožní stanoviť si priority a určiť celkový proces riešenia a napĺňania cieľov. Model problémov je znázornený na obr. 20, a model príležitostí na obr. 21.



Obrázok 19 – Príklad na celkový model biznis stratégie



Obrázok 20 – Príklad na model problémov



Obrázok 21 – Príklad na model príležitostí

6.5 OCL podmienky pre modelovanie stratégie

Teraz si uvedieme OCL podmienky, ktoré musia v spomínaných modeloch platiť pre entity BusinessGoal, Problem a Opportunity.

6.5.1 OCL podmienky pre BusinessGoal

Prvá podmienka je, že BusinessGoal môže byť spojený závislosťou (dependency) len s iným BusinessGoal-om. Táto podmienka vyzerá v OCL nasledovne:

```
context BusinessGoal inv:
self.supplierDependency->union(self.clientDependency)->forAll(
        client->forAll( stereotype.name = 'BusinessGoal' )
    and supplier->forAll(stereotype.name = 'BusinessGoal' ))
```

Druhá podmienka je, že BusinessGoal môže byť asociáciou spojený len s Problémom. Táto podmienka vyzerá v OCL nasledovne:

```
context BusinessGoal inv:
self.ownedAttribute.opposite.class->forAll(
        class.stereotype.name = 'Problem' )
```

V tejto druhej podmienke sme využili odvodenú asociáciu, zadefinovanú v UML metamodeli pre metatriedu Property nasledovne:

```
opposite =
if owningAssociation->notEmpty() and
   association.memberEnd->size() = 2 then
   let otherEnd = (association.memberEnd - self)->any() in
       if otherEnd.owningAssociation->notEmpty()
         then otherEnd
         else Set{}
       endif
else Set {}
endif
```

6.5.2 OCL podmienky pre Problem

Prvá podmienka je, že Problem môže byť asociáciou spojený len s BusinessGoal-om alebo s Opportunity. Táto podmienka vyzerá v OCL nasledovne:

```
context Problem inv:
self.ownedAttribute.opposite.class->forAll(
        class.stereotype.name = 'BusinessGoal' or
         class.stereotype.name = 'Opportunity')
```

Druhá podmienka je, že Problem môže byť spojený závislosťou len s iným Problémom. Táto podmienka vyzerá v OCL nasledovne:

```
context Problem inv:
self.supplierDependency->union(self.clientDependency)->forAll(
        client->forAll( stereotype.name = 'Problem' )
    and supplier->forAll(stereotype.name = 'Problem' ))
```

6.5.3 OCL podmienky pre Opportunity

Prvá podmienka je, že Opportunity môže byť asociáciou spojená len s Problémom. Táto podmienka vyzerá v OCL nasledovne:

```
context Opportunity inv:
self.ownedAttribute.opposite.class->forAll(
        class.stereotype.name = 'Problem' )
```

Druhá podmienka je, že Opportunity môže byť spojená závislosťou len s inou Opportunity. Táto podmienka vyzerá v OCL nasledovne:

```
context Opportunity inv:
self.supplierDependency->union(self.clientDependency)->forAll(
        client->forAll( stereotype.name = 'Opportunity' )
    and supplier->forAll(stereotype.name = 'Opportunity' ))
```

Tretia podmienka je, že Opportunity môže byť realizovaná (spojená realizáciou) len prostredníctvom Prípadu použitia alebo Biznis procesu. Táto podmienka vyzerá v OCL nasledovne:

```
context Opportunity inv:
self.supplierDependency->forAll( oclIsTypeOf( Realization ) implies
       client->forAll( oclIsTypeOf(UseCase) or
                       stereotype.name = 'BusinessProcess') )
```

6.5.4 Poznámka k UML metamodelu

V UML metamodeli 1.5 existovala možnosť pýtať sa na stereotyp elementu v modeli (pomocou stereotype.name). V UML 2.0 špecifikácii však nie je zmienka o tom, ako zistiť stereotyp elementu v modeli. Napriek tomu špecifikácia OCL 2.0 používa uvedený spôsob. Nakoľko existuje nekonzistencia medzi špecifikáciami a zároveň neexistuje v UML 2.0 možnosť zisťovania stereotypu modelového elementu, rozhodli sme sa použiť prístup z UML 1.5 a OCL 2.0. Predpokladáme, že do špecifikácie UML 2.0 sa to časom doplní.

6.6 Validácia modelov

OCL podmienky uvedené v predchádzajúcej časti formálne spresňujú entity používané na modelovanie biznis stratégie. Hlavný význam majú však v tom, že na základe týchto podmienok bude v modelovacích nástrojoch možné validovať modely vytvorené analytikmi, a tým zabezpečiť, že tieto modely budú zodpovedať uvedeným podmienkam.

V závislosti od implementácie validácie OCL v modelovacích nástrojoch, buď tieto nástroje ani neumožnia porušenie týchto podmienok (napr. spojiť BusinessGoal závislosťou s Opportunity), alebo to síce umožnia, ale po spustení validácie sa zobrazia chyby indikujúce porušenie OCL podmienok.

My sme mali možnosť pracovať s modelovacím nástrojom Enterprise Architect od firmy Sparx Systems (http://www.sparxsystems.com.au). Mali sme možnosť pracovať s jeho najnovšou dostupnou verziou v tej dobe (6.1.790). Táto verzia však spomínanú validáciu ešte neumožňovala v dostatočnej miere.

Firmu Sparx Systems sme mailom kontaktovali a chceli sme zistiť možnosti riešenia. Dozvedeli sme sa, že je možné naprogramovať si svojpomocne tzv. add-in do Enterprise Architecta, ktorý bude mať na starosti samotné validačné podmienky. Poslali nám aj ukážkové templates v jazyku C#. Naprogramovanie takejto validácie však už podstatne prekračuje rámec a rozsah tejto diplomovej práce.

7 Záver

V tejto práci sme sa zaoberali možnosťami využitia UML profilov a jazyka OCL za účelom dosiahnutia unifikácie analytických modelov v organizácii a následného zefektívnenia vývojového procesu softvéru. Popísali sme súčasné trendy v tvorbe analytických modelov, kde sme sa venovali štandardnému modelovaciemu jazyku UML a spomenuli sme si procesy vývoja v samotnej analýze. Ďalej sme poukázali na existujúce problémy súvisiace s modelovaním systémov. Popísali sme existujúce možnosti riešenia týchto problémov, kde sme sa venovali UML profilom a jazyku OCL. Na záver sme vytvorili vlastné riešenie spomínaných problémov použitím dostupných riešení. Vytvorili sme dva UML profily, biznis profil a profil na modelovanie biznis stratégie. Tieto profily poskytujú spôsob, akým môžu analytici vytvárať svoje modely unifikovaným spôsobom, za použitia entít, ktoré popisujú modelovanú doménu lepšie, než štandardné UML elementy. Ďalej sme sa zamerali na modelovanie biznis stratégie. Entity zúčastňujúce sa na tomto modelovaní, sme spresnili pomocou OCL podmienok. Vysvetlili sme si, že tieto OCL podmienky umožnia validovanie týkajúcich sa modelov. Modelovací nástroj, ktorý sme používali (Enterprise Architect), momentálne neumožňuje takúto validáciu, ale je možnosť naprogramovať add-in, ktorý túto validáciu umožní. Tu vidíme aj možnosť ďalšieho rozpracovania tejto práce. Touto prácou sme teda riešili uvedené problémy a splnili sme všetky stanovené ciele.

Zoznam literatúry

[1] Object Management Group. *UML 2.0 Infrastructure Specification*, OMG document ptc/04-10-14, 2004

[2] Object Management Group. *UML 2.0 Superstructure Specification*, OMG document formal/05-07-04, 2005

[3] Object Management Group. *Object Constraint Language 2.0 Specification*, OMG document ptc/2005-06-06, 2005

[4] Tom Pender, *UML Bible*, John Wiley & Sons, 2003

[5] Jos Warmer, Anneke Kleppe, *Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition*, Addison Wesley, 2003

[6] Dan Pilone, Neil Pitman, *UML 2.0 in a Nutshell*, O'Reilly, 2005

[7] James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual, Second Edition*, Addison Wesley, 2005

[8] Lidia Fuentes-Fernández and Antonio Vallecillo-Moreno, *An Introduction to UML Profiles*, UPGRADE Vol. V, No. 2, April 2004, http://www.upgrade-cepis.org/issues/2004/2/up5-2Fuentes.pdf