UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Využití jazyka OCL v modelech Enterprise Architect

Petr Bludský

Diplomová práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Petr Bludský**
Osobní číslo:	**I11365**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Využití jazyka OCL v modelech Enterprise Architect**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

1. V úvodní části práce je nutné provést úvod do problematiky OCL. O co se jedná, historie, důvody vzniku, jaké jsou možnosti využití.
2. Sestavit přehled, jak je OCL implementován v různých verzích Enterprise Architect (EA).
3. Ověřit kontrolu integrity na vlastním příkladu objektově orientovaného modelu v EA.
4. Navrhnout doporučení nebo zásady pro využití OCL v modelech v EA s přihlédnutím na uplatnění v semestrálních nebo diplomových pracích.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. http://www.omg.org/cgi-bin/doc?formal/03-03-13
2. Arlow J., Neustad I., UML2 a unifikovaný proces vývoje aplikací, Computer Press 2007, 567s, ISBN 978-80-251-1503-9
3. Dokumentace k Enterprise Architect

Vedoucí diplomové práce: **Ing. Karel Šimerda**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2012**
Termín odevzdání diplomové práce: **17. května 2013**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2012

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 17. 5. 2013 Petr Bludský

Poděkování

Děkuji panu Ing. Karlovi Šimerdovi za odborné vedení diplomové práce. Rovněž děkuji svým rodičům za podporu při studiu. Speciální poděkování patří mému studentskému kolegovi Lukášovi Kuchtovi za poskytnutí prostoru na svém serveru pro online repozitář OCL generátoru.

Anotace

Diplomová práce se věnuje využití jazyka Object Constraint Language (OCL) v objektově orientovaných modelech. Jako prostředí pro práci s OCL byl původně vybrán modelovací nástroj Enterprise Architect (EA). Vzhledem ke zjištění nedostatečné podpory jazyka OCL v EA se práce věnuje také nalezení vhodného OCL prostředí. Dále se zabývá vývojem OCL pluginu pro platformu Eclipse umožňujícího generovat validační kód pro kontrolu diagramů tříd vyvářených v rámci kurzů objektově orientovaného modelování.

Klíčová slova

Ecore, Eclipse, EA, EMF, Eclipse Modeling Framework, Enterprise Architect, modelování, OCL, Object Constraint Language, UML, Unified Modeling Language

Title

Use of the OCL language in Enterprise Architect models

Annotation

The main purpose of this master thesis is to describe how Object Constraint Language (OCL) can be used in object-oriented models. The modeling tool Enterprise Architect (EA) had been originally selected as an OCL environment but was evaluated insufficient. The thesis is therefore focused on finding a suitable OCL environment as well. The thesis is also devoted to the development of an OCL plugin for the Eclipse platform. The developed plugin functions as a validation code generator for checking class diagrams created in object-oriented modeling courses.

Keywords

Ecore, Eclipse, EA, Enterprise Architect, EMF, Eclipse Modeling Framework, modeling, OCL, Object Constraint Language, UML, Unified Modeling Language

Obsah

Seznam zkratek

API	Application Programming Interface
CASE	Computer Aided Software Engineering
CIM	Computation Independent Model
CMOF	Complete Meta Object Facility
CORBA	Common Object Request Broker Architecture
DTD	Document Type Definition
EA	Enterprise Architect
EAI	Enterprise Application Integration
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
ECL	Epsilon Comparison Language
EGL	Epsilon Generation Language
EML	Epsilon Merging Language
EOL	Epsilon Object Language
ETL	Epsilon Generation Language
EVL	Epsilon Validation Language
HTML	HyperText Markup Language
IDE	Integrated Development Environment
MDA	Model Driven Architecture
MDT	Model Development Tools
MOF	Meta Object Facility
MVC	Model-view-controller
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
RCP	Rich Client Platform
SOIL	Simple OCL-based Imperative Language
SQL	Structured Query Language
SysML	System Modeling Language
SWT	Standard Widget Toolkit
UI	User Interface
UML	Unified Modeling Language
xtUML	Executable Unified Modeling Language
XPath	XML Path Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Seznam bloků kódu

Všechny uvedené bloky kódu jsou dílem autora práce. Zdroj v hlavním textu práce proto není uváděn.

Slovníček pojmů

Computer Aided Software Engineering (CASE) – nástroj umožňující automatizovat některé činnosti v různých fází tvorby softwarového produktu.

Deklarativní jazyk – popisuje cíl, kterého má být dosaženo. Nepopisuje již, jakým způsobem jej dosáhnout. Výrazy deklarativního jazyka nedokáží měnit stav systému.

Diagram – grafická reprezentace modelu. Představuje pohled na vybranou část modelu.

Diagram aktivit – používá se k modelování procesu jako aktivity, která se skládá z kolekcí uzlů spojených hranami. Lze jej využít k modelování procesů, procedurální logiky a zachycení workflow.

Diagram interakce – popisuje spolupráci souboru objektů pro dosažení nějakého chování či cíle.

Diagram tříd – vizualizace struktury modelovaného systému. Zachycuje třídy a vztahy mezi nimi.

(Dynamická) instance modelu – množina instancí klasifikátorů vytvořených z modelu. Používá se pro testování modelu (pojem „dynamická“ pochází z prostředí Eclipse).

Ecore – meta-metamodel charakteristický především vysokou mírou flexibility (lze jej využít i jako metamodel pro definici modelů). V textu práce je na něj odkazováno převážně jako na metamodel. Byl vyvinut pro platformu Eclipse.

Eclipse – open source vývojová a modelovací platforma charakteristická velkou možnou mírou přizpůsobení požadovanému účelu.

Eclipse Modeling Framework (EMF) – modelovací framework pro platformu Eclipse. Kromě modelování umožňuje např. generování kódu, validování modelu apod.

Enterprise Architect (EA) – prostředí pro vizuální modelování a návrh společnosti Sparx Systems. Zaměřené je především na UML, ovšem nabízí podporu i dalších modelovacích jazyků.

Imperativní jazyk – popisuje, jakým způsobem dosáhnout vytyčeného cíle (zapisuje algoritmy). Definuje posloupnost příkazů s potenciálem měnit stav systému.

Integritní omezení – definuje podmínky, které musejí být v každém případě splněny ze strany dat systému.

Invariant – Obecně platné tvrzení, které musí být platné po celou dobu existence instance klasifikátoru s výjimkou časového okamžiku, ve kterém instance provádí operaci.

Klasifikátor – předpis pro prvky společných vlastností. Každý prvek, ze kterého lze vytvořit instanci, je zároveň klasifikátorem.

Kontext (OCL) – definuje oblast, pro kterou je daný výraz či výrazy platný. Může se jednat např. o oblast balíčku, klasifikátoru, operace apod.

Kontextová instance (OCL) – instance, na kterou se vztahuje výraz či výrazy definované v kontextu. Vztahují se tedy již na konkrétní balíček, instanci klasifikátoru, definovanou operaci apod. Např. výraz definující omezení v kontextu třídy Osoba by se vyhodnocoval pro jednotlivé konkrétní osoby (např. Jan Novák).

Meta-metamodel – definuje výrazové prostředky pro tvorbu metamodelů.

Metamodel – definuje prostředky pro tvorbu modelu. Obsahuje sadu pravidel, které musí model splňovat, aby byl pro daný metamodel validní. Zároveň na něj lze pohlížet jako na instanci meta-metamodelu.

Meta Object Facility (MOF) – standard konsorcia OMG. Představuje framework pro tvorbu metamodelů.

Model – abstrakce systému reálného světa. Je definován jako popis systému v nějakém modelovacím jazyce. Zároveň na něj lze pohlížet jako na instanci metamodelu.

Modelování – proces vytváření abstrakce zkoumaného systému reálného světa. Abstrahuje se od vlastností systému, které nejsou z hlediska zkoumání podstatné nebo není v technických možnostech je vystihnout.

Object Constraint Language (OCL) – deklarativní jazyk používaný nejen pro definici integritních omezení modelu. Lze jej také použít např. jako dotazovací jazyk.

Pre-condition – podmínka, která musí být splněna před započetím operace.

Profil metamodelu – rozšíření metamodelu. Definuje rozšiřující metatřídy (v UML známé jako stereotypy, které rozšiřují sémantiku prvku), metaatributy (v UML definují označené hodnoty), metaasociace a jejich integritní omezení. Profil metamodel žádným způsobem nemění, pouze jej rozšiřuje.

Post-condition – podmínka, která musí být splněna v okamžiku provedení operace.

Sekvenční diagram – vizualizace spolupráce vybraných objektů, které si mezi sebou zasílají zprávy.

Unified Modeling Language (UML) – univerzální jazyk pro vizuální modelování systémů.

Validace modelu – proces ověření, zda model vyhovuje pravidlům definovaných jeho metamodelem.

Typografické konvence

Pro lepší orientaci čtenáře v textu jsou použity následující typografické konvence:

- **derive** Tučně jsou psány termíny a pasáže, které chce autor zdůraznit.
- *forAll* Kurzívou jsou psány klíčová slova a identifikátory, tj. jména proměnných, funkcí, typů apod. Není rozlišováno, zda se jedná o standardní součást jazyka nebo o jména definovaná autorem.
- *pre-condition* Kurzívou jsou dále psány důležité pojmy.
- OCL Zkratky jsou psány kapitálkami.
- <CTRL+A> Klávesové zkratky jsou psány kapitálkami v ostrých závorkách.
- `pre: castka > 0` Neproporcionální písmo je použito pro krátké výřezy zdrojového kódu v textu.
- Deklarativní kód, imperativní kód nebo pseudokód je ohraničen světle šedou barvou se zvýrazněním klíčových slov a programových konstrukcí. Pod každým blokem kódu je uvedeno, o jaký jazyk se jedná společně s popiskem a pořadovým číslem výskytu, na který je v textu odkazováno.

```
-- Komentář je psán zelenou barvou
inv uniqueAssociationEnds:
let aEnds : Bag = eAllReferences->asBag() in aEnds->isUnique(self)
```

Blok 0 – OCL. Popis bloku kódu

1 Úvod

Cílem diplomové práce je seznámit čtenáře s jazykem Object Constraint Language (OCL) a jeho využitím pro objektově orientované modelování. Jazyk OCL je v současné době poněkud přehlíženým řešením, pro které existuje zatím jen málo uceleného materiálu. Vzhledem ke své vysoké odlišnosti od klasických programovacích jazyků je často zavrhován programátory, zatímco softwarovým analytikům se zase může jevit zbytečně složitým. Práce se snaží najít vhodné možnosti aplikace jazyka OCL. Představeny ovšem budou i poměrně nekonvenční možnosti jeho užití.

Původním cílem práce bylo také definovat integritní omezení popsané v jazyce OCL pro vlastní objektově orientovaný model v modelovacím CASE nástroji Enterprise Architect (EA) a ověřit je. Dalším krokem mělo být formulování sady doporučení pro využití jazyka OCL v tomto nástroji s přihlédnutím k uplatnění ve školních pracích. Vzhledem ke zjištění nemožnosti ověření integritních omezení v EA byly cíle práce na základě diskuze s vedoucím práce přeformulovány. Novým cílem je najít vhodné prostředí pro práci s jazykem OCL a v návaznosti na něj demonstrovat použití OCL na ukázkovém modelu. Dalším krokem je vytvořit rozšíření tohoto prostředí. Funkcionalita rozšíření by měla směřovat k využití jazyka OCL pro kontrolu školních prací v předmětech zabývajících se objektově orientovaným modelovaním.

Text vlastní práce je strukturován do kapitol. Kapitola 2 se zabývá aspekty jazyka OCL. Čtenář je uveden do problematiky jazyka, je seznámen s jeho historií, výrazovými prostředky a možnostmi užití. Kapitola 3 mapuje podporu jazyka OCL v modelovacím CASE nástroji Enterprise Architect. Kapitola 4 je rešeršního charakteru a prezentuje možné platformy a prostředí pro práci s jazykem OCL. V kapitole 5 je představen ukázkový model polikliniky, na kterém je demonstrováno použití jazyka OCL. Kapitola 6 se zabývá popisem vlastního řešení OCL pluginu pro platformu Eclipse.

Od čtenáře se předpokládá základní přehled v oblasti softwarového modelování.

2 Jazyk OCL

Cílem kapitoly je poskytnout solidní úvod do problematiky jazyka OCL. Čtenář je postupně seznámen s aspekty jazyka, jeho historií, možnostmi užití a mnohým dalším.

2.1 Charakteristika

Následující text vychází z publikací [1], [2], [3] a [4].

Object Constraint Language (OCL) je deklarativní jazyk (viz Slovníček pojmů) pro formalizaci popisu skutečností a faktů o modelovaném systému. Převedením popisu struktury a chování systému z přirozeného jazyka do formalizované podoby odpadá nejednoznačnost, která je nedílnou součástí přirozeného jazyka. Druhou výhodou je možnost strojového zpracování s potenciálem přispět ke zkvalitnění a akceleraci vývoje softwarového produktu jako částečně i jeho testování.

OCL umožňuje definovat výrazy nad modelem (viz Slovníček pojmů). Ty mohou být použity pro dodatečný popis operací nebo akcí, jež mají potenciál ovlivnit stav modelovaného systému. Pouhé vyhodnocení těchto výrazů na stav systému však žádný efekt nemá. Mnohem častěji se vyskytují ve formě *invariantů*, tedy obecně platných tvrzení, kterým musí softwarový systém v libovolném časovém okamžiku jeho existence vyhovovat.

OCL není jazykem, kterým lze modelovat, a je tedy plně závislé na zvoleném modelovacím jazyku (např. UML). Bez modelovacího jazyku je OCL samo o sobě nepoužitelné, neboť výrazy v něm psané musejí být definovány pro konkrétní model.

OCL je silně typovým jazykem, který primárně pracuje s prvky definovanými v modelu. Kromě toho také disponuje svou vlastní typovou sadou obsahující jak primitivní datové typy známé z vyšších programovacích jazyků, tak vlastní vestavěné typy.

Původně byl tento jazyk pouze integrovanou součástí UML. Dnes již toto tvrzení neplatí a OCL je možné nasadit v modelech vytvořených pod metamodely (viz Slovníček pojmů) různých domén i v definici metamodelů samotných, jejichž sémantiku pomáhá formalizovat.

Jakým způsobem jsou OCL výrazy k danému modelu přidruženy, je v režii jednotlivých modelovacích nástrojů. Běžně jsou definovány v samostatných textových dokumentech (výhodné z hlediska přenositelnosti a přehlednosti) nebo vkládány do XMI[1] souborů.

2.2 Historie

Počátky OCL se datují do roku 1995 [5], kdy společnost IBM vytvořila první verzi jazyka označovanou jako „Business Engineering Language“, přičemž vycházela z balíčku

[1] XMI definuje formát pro přenos modelových metadat mezi modelovacími nástroji. V době psaní práce se nachází ve verzi 2.0, přičemž ve vývoji je momentálně verze 2.1. [43]

objektově orientovaných analytických a designových nástrojů Syntropy vytvořeného na začátku 90. let. I tehdy byly grafické notace stejně jako dnes upřednostňovány před formálním popisem z důvodů snadné čitelnosti a menších nároků kladených na jejich pochopení. Daní za grafické ztvárnění byla nutnost zjednodušování a vypouštění některých informací, které ze své podstaty nemohly být v grafických diagramech zachyceny. Původ OCL lze tedy hledat v jazyku, který by jednoduchým způsobem (přístupném pro lidi z byznys sféry) umožnil formálně zachytit skutečnosti jinak popsatelné často složitým matematickým zápisem.

Syntropy definovalo 3 náhledy na vytváření objektových modelů [6]:

- **Základní model** – sloužící k popisu elementů a jejich chování podle skutečné předlohy.
- **Specifikační model** – ve smyslu abstraktního pohledu na softwarový systém fungující na bázi akce a reakce, ve kterém platí předpoklad nekonečné výpočetní síly a nelimitovaných zdrojů.
- **Implementační model** – jako detailní pohled na softwarový systém, který již bere v úvahu výpočetní omezení a dostupnost zdrojů.

OCL bylo zařazeno konsorciem OMG do specifikace UML 2.0 jako formální specifikační jazyk v roce 2003, avšak do dnešního dne se bohužel potýká s nedostatečnou podporou ze strany vývojářů modelovacích CASE nástrojů. Ta byť se postupem času zlepšuje, stále ještě není na takové úrovní, aby mohl být potenciál OCL plně využit. Vážným nedostatkem je zejména častá absence interpretu, který by umožňoval vyhodnocování OCL výrazů nad modelem.

2.3 Syntaxe a sémantika

Syntaxe jazyka OCL je poměrně rozsáhlá a místy lehce komplikovaná. V této podkapitole jsou uvedeny její aspekty společně s popisy typů a operací nad nimi ze standardní knihovny oficiální specifikace OCL konsorcia OMG [2].

Jazyk OCL se neřadí do rodiny klasických programovacích jazyků (nejedná se totiž v pravém slova smyslu o programovací jazyk). Jsou mu tedy odepřeny standardní konstrukce určené pro psaní programové logiky nebo pokročilého řízení toku programu.

OCL není jazykem imperativním (viz Slovníček pojmů), jako např. Java či C#, ale jazykem deklarativním. Jeho prostřednictvím je tedy popsán až konečný výstup a nikoli způsob, jakým jej lze dosáhnout. Oproti imperativním jazykům je kód zapsaný v OCL obecně mnohem kratší.

Obecná syntaxe OCL má strukturu dle bloku 1.

```
package <cestaKBalíčku>
kontext výrazu { context <názevKontextovéInstance>:<názevPrvkuModelu>
                   <kategorieVýrazu> <názevVýrazu>:
                   <těloVýrazu>
```

```
                    <kategorieVýrazu> <názevVýrazu>:
                    <tělоVýrazu>
                    ... }
endpackage
```

Blok 1 – Pseudokód. Obecná struktura syntaxe OCL

Obecnou formu OCL výrazu lze rozdělit do tří částí:

- kontext balíčku (nepovinný),
- kontext výrazu (povinný),
- 1..N výrazů (vždy alespoň jeden výraz, prázdný kontext není povolen).

Účelem kontextu balíčku je specifikovat jmenný prostor pro výrazy OCL s tím, že chování je podobné jako u jmenných prostorů známých z klasických programovacích jazyků. Vzhledem k tomu, že prvky v balíčku mají vždy svůj jedinečný název, lze specifikaci kontextu balíčku vynechat[2]. [1], [3]

Kontext výrazu slouží k definici kontextové instance ve tvaru *název:typ*, kde první položka je nepovinná. Kontextovou instancí se rozumí instance typu, na který je kontext vztažen. Například u třídy *Lekar* by se tedy jednalo o kterýkoli objekt vytvořený z této třídy. Název kontextové instance není v praxi příliš používán z jediného důvodu: OCL obsahuje klíčové slovo *self*, pomocí kterého lze na instanci uvnitř výrazu zacílit. Tělo výrazu pak tvoří množina výrazů (dalo by se říci také podvýrazů), které se skládají z konstrukcí jazyka OCL. [1], [3]

Kontext pro třídu *Lekar* nacházející se v balíčku *Nemocnice* je znázorněn v bloku 2.

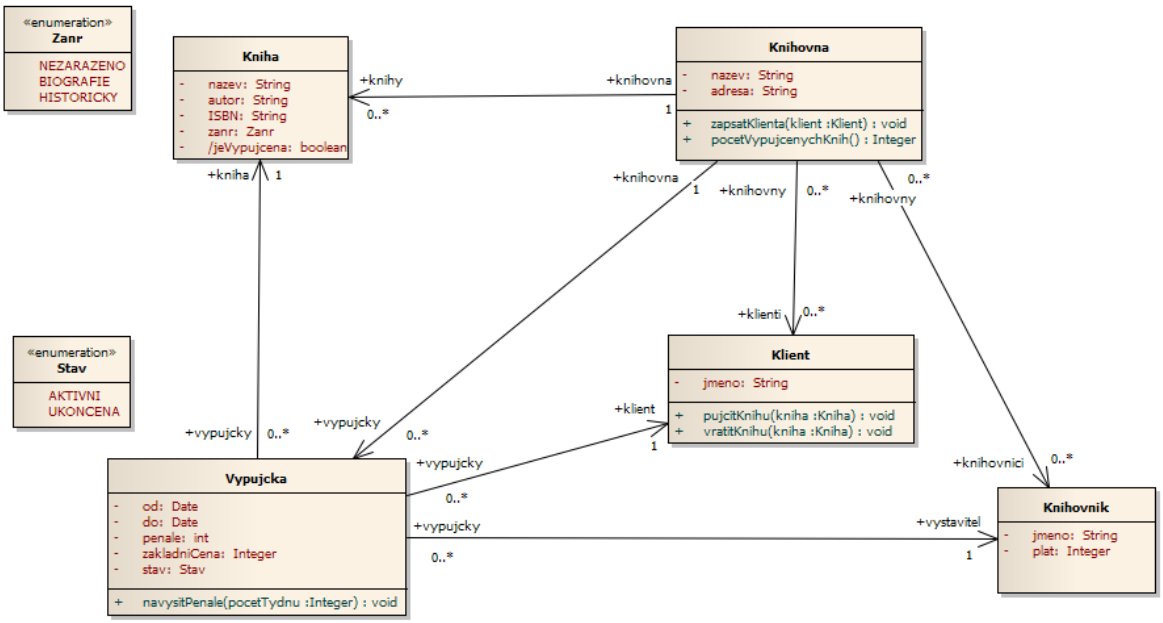
```
-- Začátek balíčku, pro který jsou OCL výrazy definovány
package Nemocnice
-- Kontext třídy Lekar
context Lekar
...
-- Kontext třídy Lekar včetně cesty k balíčku Nemocnice
context Nemocnice::Lekar
...
-- Kontext třídy Lekar s názvem kontextové instance LekarovaInstance
context LekarovaInstance:Lekar
...
-- Konec balíčku
endpackage
```

Blok 2 – OCL. Ukázka zápisu kontextu

2.3.1 Kategorie výrazů

[2] Jiná situace nastává, když je v aktuálním balíčku nutné odkázat na prvky modelu, které se nacházejí v jiném balíčku. Pak je kontext balíčku nutné specifikovat.

V této části jsou popsány kategorie výrazů jazyka OCL včetně ukázkových příkladů. Příklady jsou určeny pro model knihovny znázorněný diagramem tříd na obrázku 1. Doposud nepředstavené operace OCL, které jsou součástí uvedených výrazů, budou vysvětleny dále v textu. Pro pochopení sémantiky jednotlivých kategorií výrazů nejsou prozatím podstatné.



Obrázek 1 – Diagram tříd knihovny. Zdroj: autor

Výrazy jazyka OCL lze rozdělit do dvou kategorií. Na booleovské výrazy, které specifikují omezení [3]:

- **inv** – je *invariant* klasifikátoru (výraz musí platit pro všechny kontextové instance),
- **pre** – značí *pre-condition* u operace vlastněné klasifikátorem,
- **post** – je *post-condition* u operace vlastněné klasifikátorem.

Dále pak na výrazy specifikující atributy, těla operací, funkce a proměnné [3]:

- **body** – definuje tělo dotazovací operace (provedení operace nemění stav objektu),
- **def** – definuje pomocné proměnné či funkce pro zvolený kontext,
- **derive** – definuje derivační pravidlo pro odvozené atributy nebo konce asociací,
- **init** – inicializuje atribut nebo konec asociace na zadanou hodnotu,
- **let –** vytváří lokální proměnné uvnitř aktuálního OCL výrazu.

První kategorie výrazů pochází z metodiky vyvinuté Bertrandem Meyerem zvané Design by Contract [6], která popisuje vzájemnou spolupráci prvků softwarového systému na bázi povinností a benefitů. Aby mohla protistrana využít benefitu (zavolat operaci), musí nejprve dostát svých povinností (zde myšleno poskytnout korektní argumenty). Tato pravidla jsou zapsána ve formě booleovských výrazů.

Výraz inv – definice invariantu

Tento výraz se vyhodnocuje na úrovni instance klasifikátoru a musí být pravdivý po celou dobu její existence kromě okamžiku provádění operace, kdy je připuštěna jeho neplatnost (může nabývat hodnoty *false*). Bezprostředně po ukončení provádění operace musí být *invariant* opět platný. [6], [7]

Mějme integritní omezení, které říká, že knihu může mít v daném čase vypůjčenou pouze jeden klient. Toto omezení lze vystihnout *invariantem* znázorněným v bloku 3.

```
context Kniha
inv singleActiveLoanCheck: vypujcky->select(stav = Stav::AKTIVNI)->size() <= 1
```

Blok 3 – OCL. Invariant definující integritní omezení pro model knihovny

Výraz pre – definice pre-condition

Pre-condition ověřuje, zda se systém před provedením dané operace nachází v požadovaném stavu. Definuje také požadavky kladené na vstupní argumenty operace. Splnění těchto podmínek je nutným předpokladem pro provedení dané operace. Zodpovědnost, že tomu tak skutečně je, spočívá na straně volající operace. [6], [7]

Operace *zapsatKlienta* třídy *Knihovna* by měla do evidence knihovny zapsat pouze ty klienty, kteří ještě nejsou registrovaní. Tuto skutečnost pro zmíněnou operaci zajistí právě *pre-condition* uvozená výrazem *pre* (viz blok 4).

```
context Knihovna::zapsatKlienta(klient : Klient) : OclVoid
pre: klienti->excludes(klient)
```

Blok 4 – OCL. Pre-condition operace zapsatKlienta třídy Knihovna

Výraz post – definice post-condition

Tato podmínka je, stejně jako *pre-condition*, aplikovatelná pouze na operace a popisuje stav, ve kterém se daný objekt musí nacházet v okamžiku ihned po jejím vykonání. Straně, která volá danou operaci a zároveň splnila požadavek na *pre-condition* a *invariant* pro danou instanci je platný, je garantováno, že se systém bude nacházet ve stavu vyhovujícím definované *post-condition*. OCL umožňuje u tohoto výrazu použít klíčové slovo **@pre** odkazující na hodnotu atributu před provedením operace. [6], [7]

Operace *navysitPenale* třídy *Vypujcka* navýší penále za výpujčku dle počtu týdnů navíc. Přičítá se 20 Kč za každý týden. Kromě již známého výrazu *pre* zde přibývá výraz *post* vyjadřující *post-condition*. Za výrazem *post* je definováno, jak se změní stav instance třídy *Vypujcka* přesně poté, co proběhne operace *navysitPenale*. Aktuální hodnota penále za výpujčku se zvýší o požadovaný počet. Aby bylo možné zjistit výši penále před vykonáním těla operace, je použito klíčové slovo *@pre*. Korespondující výrazy jsou znázorněny v bloku 5.

```
context Vypujcka::navysitPenale(pocetTydnu : Integer) : OclVoid
pre: pocetTydnu > 0 and stav = Stav::AKTIVNI
post: penale = penale@pre + pocetTydnu * 20
```

Blok 5 – OCL. Popis operace navysitPenale třídy Vypujcka užitím pre-condition a post-condition

Samotný způsob provedení operace (její tělo), která mění stav systému, nelze v OCL zapsat. Jedná se již o úlohu imperativních jazyků.

Je důležité zmínit, že v případě použití dědičnosti, kdy může docházet k překrývání operací, platí pro redefinovanou operaci následující pravidla [3]:

- *pre-condition* může být pouze méně restriktivní,
- *post-condition* může být pouze restriktivnější.

Tato pravidla je nutné dodržet pro zachování Liskov Substitution Principle (LSP)[3].

Specifikace chování třídních operací pomocí *pre* a *post-condition* se používá v raných fázích vývoje softwarového díla. Pomáhá k lepšímu pochopení zodpovědností tříd a může usnadnit rozhodnutí činěná ve fázi návrhu, případně analýzy[4].

Výraz body – definice těla dotazovací operace

Dotazovací operace je taková operace, která nemění stav objektu. Pouze tyto operace je OCL schopno vyhodnocovat a žádné jiné (OCL interpret je může spouštět a na základě jejich provedení vracet příslušné hodnoty). Definice těla dotazovací operace je uvozena výrazem *body*. [1]

Třída *Knihovna* disponuje dotazovací operací *pocetVypujcenychKnih*, pomocí které je možné zjistit aktuální počet knih ve výpujčce (viz blok 6).

```
context Knihovna::pocetVypujcenychKnih() : Integer
body: knihy->select(k : Kniha | k.vypujcky.stav->includes(Stav::AKTIVNI))
->size()
```

Blok 6 – OCL. Definice těla dotazovací operace pocetVypujcenychKnih třídy Knihovna

Výrazy def – definice pomocné proměnné či funkce

Výrazem *def* lze definovat pomocnou proměnnou či funkci pro zvolený kontext, kterou lze pak využívat v ostatních výrazech kontextu. Jedná se tedy svým způsobem o definici globální proměnné či funkce v rozsahu kontextu. [1]

Mějme integritní omezení, které určuje, že každá knihovna může zaměstnávat nejvýše deset knihovníků. Maximální počet knihovníků lze uložit do pomocné proměnné s názvem *maxLibrarians*. V *invariantu* vystihující integritní omezení je možné se na něj pak odvolat. Uvedené skutečnosti znázorňuje blok 7.

```
context Knihovna
def: maxLibrarians : Integer = 10
```

[3] LSP říká, že konkrétnější prvek lze použít všude tam, kde by bylo možné použít prvek obecnější, aniž by byl ohrožen chod systému [45].

[4] Ovšem pouze za předpokladu, že analytické třídy jsou vytvářeny detailnějším způsobem, tj. je nutné stanovovat datové typy třídních atributů, plné signatury operací a pojmenovávat konce asociací stejně jako určovat jejich násobnosti.

```
context Knihovna
inv maxLibrariansCheck: knihovnici->size() <= maxLibrarians
```

Blok 7 – OCL. Užití výrazu def v kombinaci s invariantem pro kontext Knihovna

Výraz derive – derivační pravidlo

Derivační pravidlo umožní přiřadit funkční složku atributu nebo konci asociace. Hodnota je pak určena výrazem za *derive*. [1]

Třída *Kniha* disponuje odvozeným atributem *jeVypujcena*. Odvozený atribut se v diagramu tříd standardně uvozuje znakem '/'. Tento booleovský atribut pak bude mít hodnotu *true* nebo *false* v závislosti na tom, zda pro konkrétní instanci knihy existuje nějaká aktivní výpujčka (viz blok 8).

```
context Kniha::jeVypujcena : Boolean
derive: not vypujcky->select(stav = Stav::AKTIVNI)->isEmpty()
```

Blok 8 – OCL. Derivační pravidlo pro atribut jeVypujcena třídy Kniha

Výraz init – inicializace atributu či konce asociace

Výrazem *init* lze nastavit počáteční hodnotu atributu či konce asociace. V okamžiku, kdy dojde ke vzniku konkrétního objektu, proběhne příslušná inicializace. [1]

Nově vzniklá kniha bude žánrově nezařazená. Atributu *zanr* se tedy nastaví výčtový literál *NEZARAZENO* (viz blok 9).

```
context Kniha::zanr : Zanr
init: Zanr::NEZARAZENO
```

Blok 9 – OCL. Inicializace atributu zanr třídy Kniha

Nově vzniklá kniha taktéž nebude mít ještě žádnou výpujčku. Tuto informace lze promítnout nastavením asociačního konce *vypujcky*. Takovéto využití nemá v praxi valného významu, neboť kolekce výpujček bude pro novou knihu implicitně prázdná. Jako ukázka však postačí (viz blok 10).

```
context Kniha::vypujcky : Set(Vypujcka)
init: Set{}
```

Blok 10 – OCL. Inicializace konce asociace vypujcky třídy Kniha

Výraz let – tvorba lokální proměnné

Lokální proměnná se vytváří výrazem *let*. Tento výraz je vhodné použít zejména v případě, že se v daném OCL výrazu nachází několik duplicitních podvýrazů. Dále je možné *let* použít pro řetězení výrazů, zpřehlednění výrazu a taktéž jako argument OCL operací. Po *let* následuje povinné klíčové slovo *in*, které dovolí navázat na další výraz a ukončí tím definici lokální proměnné. V jednom výrazu *let* je možné definovat více lokálních proměnných (oddělují se čárkou). [1]

Jako ukázka poslouží OCL dotaz, který vrátí názvy všech knižních titulů z knihoven *Knihovna Pardubice* a *Knihovna Chrudim* a taktéž jména všech klientů těchto knihoven (viz blok 11). Bez použití *let* by bylo nutné celý podvýraz `Knihovna.allInstances()->select(Set{'Knihovna Pardubice', 'Knihovna Chrudim'}->includes(nazev))` napsat dvakrát. Takto se jeho výsledek dosadí místo lokální proměnné *libs*. Kontextem je v tomto případě *Model*, neboť právě nad modelem dotaz probíhá (v OCL dotazovacích nástrojích se většinou neuvádí).

```
context Model
let libs : Set(Knihovna) = Knihovna.allInstances()->select(Set{'Knihovna
Pardubice', 'Knihovna Chrudim'}->includes(nazev)) in
Tuple{nazvyKnih = Set{libs.knihy}, jmenaKlientu = Set{libs.klienti}}
```

Blok 11 – OCL. Dotaz s užitím výrazu let

2.3.2 Klíčová slova

Klíčová slova jsou v jazyce OCL rezervována a tedy nemohou být použita jako identifikátor.

`and, body, context, def, derive, else, endif, endpackage, false, if, implies, in, init, inv, invalid, let, not, null, or, package, post, pre, self, then, true, xor`

2.3.3 Priority operátorů a výrazů

Priorita je seřazena v sestupném pořadí. Prioritu je možné měnit pomocí závorek.

1. ‘ (’, ‘)’, ‘if-then-else-endif’
2. ‘let-in’
3. ‘@pre’
4. ‘.’, ‘->’, ‘^’, ‘^^’
5. ‘not’, ‘-’
6. ‘*’, ‘/’
7. ‘+’, ‘-’
8. ‘>’, ‘<’, ‘<=’, ‘>=’
9. ‘=’,‘<>’
10. ‘and’, ‘or’, ‘xor’
11. ‘implies’
12. ‘in’

2.3.4 Komentáře

Komentáře (viz blok 12) je možné vložit kamkoli do textu výrazu. Pro jednořádkové komentáře se používá řetězec ‘--’. Víceřádkové komentáře začínají řetězcem ‘/*’ a končí ‘*/’. Během parsování kódu jsou komentáře ignorovány.

```
-- Jednořádkový komentář

/*
 * Víceřádkový komentář
*/
```

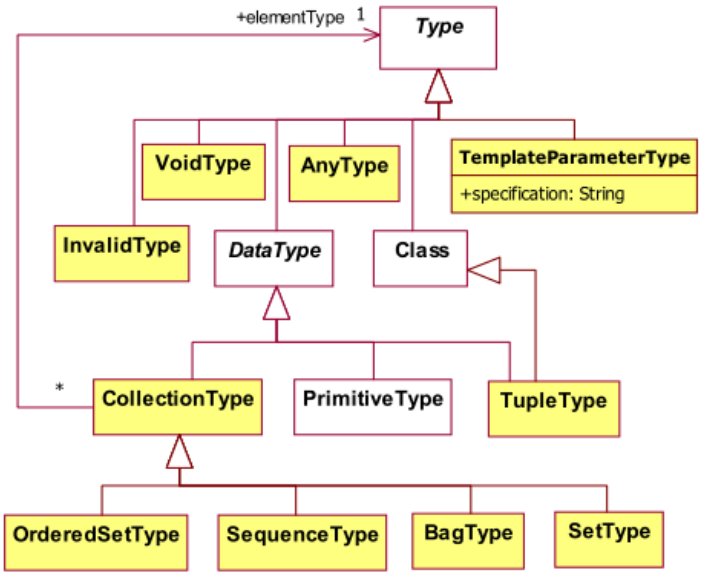
Blok 12 – OCL. Způsob zápisu komentáře

2.3.5 Typy

Typy jazyka OCL lze rozdělit do dvou kategorií:

- primitivní a strukturované,
- vlastní.

Typy jazyka OCL definuje metamodel typů znázorněný diagramem na obrázku 2.



Obrázek 2 – Metamodel typů OCL [2]

Primitivní a strukturované

Jak již bylo zmíněno, jazyk OCL si stejně jako každý klasický programovací jazyk s sebou nese vlastní sadu primitivních typů. Jedná o typy *Boolean*, *Integer*, *Real*, *String* a *UnlimitedNatural* (reprezentuje nekonečno). Vzhledem k povaze jazyka OCL tyto typy nemají žádné omezení rozsahu. Příslušný interpret jazyka OCL samozřejmě musí provést konverzi na rozsahově omezený implementační typ, se kterým je procesor schopen pracovat. [1],[3]

V následujícím textu je použité klíčové slovo *self*, které zde představuje první operand nebo instanci, na kterou se daný operátor nebo operace uplatňuje. Za znakem „rovná se ” je uveden návratový typ.

Operace a operátory typu Boolean

- **self : Boolean or b : Boolean = Boolean**
  Vrací *true*, pokud je *self* nebo *b true*, jinak *false*. Jedná se o logickou disjunkci.
- **self : Boolean xor b : Boolean = Boolean**
  Vrací *true*, pokud je buď *self*, nebo *b true*, jinak *false*. Jedná se o exkluzivní logický součet.

- **self : Boolean and b : Boolean = Boolean**
  Vrací *true*, pokud jsou *self* a *b* zároveň *true*, jinak *false*. Jedná se o logickou konjunkci.
- **not self : Boolean = Boolean**
  Negace *self*. Vrací *true*, pokud je *self false*, jinak *false*. Jedná se o logickou negaci.
- **self : Boolean implies b : Boolean = Boolean**
  Vrací *true*, pokud je *self false* nebo *self* je *true* a zároveň *b* je *true*, jinak *false*. Jedná se o logickou implikaci.
- **self : Boolean = b : Boolean = Boolean**
  Vrací *true*, pokud je *self* a *b* zároveň *false* nebo *true*, jinak *false*. Jedná se o logickou ekvivalenci.
- **self.toString() : String**
  Převede *self* na řetězec.

Společné operace a operátory pro číselné typy

Typ *Number* v následujícím výčtu substituuje kterýkoli z typů *Integer, Real* nebo *UnlimitedNatural*. Všechny číselné typy podporují základní matematické a porovnávací operátory (sčítání, odečítání, větší než, …).

- **self : Number div n : Number = Number**
  Vrací dělení *self* číslem *n*. Výsledkem je *invalid*, pokud je *n* nula nebo *unlimited* nebo pokud je *self unlimited.*
- **self : Number mod n : Number = Number**
  Vrací modulo *self* číslem *n*. Výsledkem je *invalid*, pokud je *n unlimited* nebo pokud je *self unlimited.*
- **self.max(n : Number) = Number**
  Vrací maximum *self* a *n*.
- **self.min(n : Number) = Number**
  Vrací minimum *self* a *n*.
- **self.toString() = String**
  Převede *self* na řetězec.

Operace typu Integer

- **self.abs() = Integer**
  Vrací absolutní hodnotu *self*.

Operace typu Real

- **self.abs() = Real**
  Vrací absolutní hodnotu *self*.
- **self.floor() = Integer**
  Provede zaokrouhlení *self* směrem dolu.
- **self.round() = Integer**
  Provede zaokrouhlení *self* směrem nahoru.

Operace typu String

Typ *String* podporuje porovnávací operátory (větší než, menší než, …), kterými lze porovnávat dva řetězce na základě definice vlastnosti *oclLocale*[5] konkrétního běhového prostředí.

- **self.size() = Integer**
  Vrací počet znaků v *self*.
- **self.concat(s : String) = String**
  Vrací zřetězení *self* a *s*. Tato operace je zaměnitelná s operátorem '+' pro typ *String*.
  `'a'.concat('b') = 'a' + 'b'`
- **self.substring(lower : Integer, upper : Integer) = String**
  Vrací podřetězec *self* začínající na indexu *lower* a končící na indexu *upper*. Indexuje se od 1.
- **self.toInteger() = Integer**
  Převede *self* na celé číslo.
- **self.toReal() = Real**
  Převede *self* na reálné číslo.
- **self.toUpperCase() = String**
  Převede všechny znaky *self* na velká písmena dle definice *oclLocale* konkrétního běhového prostředí.
- **self.toLowerCase() = String**
  Převede všechny znaky *self* na malá písmena dle definice *oclLocale* konkrétního běhového prostředí.
- **self.indexOf(s : String) = Integer**
  Vrací pozici podřetězce *s* v *self*. Prázdný řetězec je brán jako podřetězec každého řetězce kromě prázdného. Vrací 0 v případě nenalezení shody.
- **self.equalsIgnoreCase(s : String) = Boolean**
  Aplikuje operaci *toUpperCase* na self a *s* a provede jejich vzájemné porovnání. Vrací *true*, pokud se oba řetězce shodují, jinak *false*.
- **self.at(i : Integer) = String**
  Vrací znak *self* na zadaném indexu *i*.
- **self.characters() = Sequence(String)**
  Vrací jednotlivé znaky *self* v kolekci typu *Sequence*.
- **self.toBoolean() = Boolean**
  Převede *self* na hodnotu typu *Boolean*.

Typ Tuple

Kromě primitivních typů je k dispozici strukturovaný typ *Tuple* schopný pojmout vícero objektů různého typu. *Tuple* Reprezentuje klasickou n-tici hodnot. Do OCL specifikace byl

[5] Vlastnost *oclLocale* byla představena v OCL verzi 2.3 a je ji možné využít pro definování jazykové mutace, čímž je následně ovlivněno chování některých operací u typu *String* (např. převod na velká či malá písmena) [2].

přidán ve chvíli, kdy vyvstal požadavek na navrácení více jak jedné hodnoty při použití OCL jako dotazovacího jazyka. Absence typu *Tuple* do té doby snižovala možnost využití jazyka OCL pro dotazování, neboť bylo možné vrátit vždy pouze jednu hodnotu. Jeho využití je samozřejmě možné také mimo oblast dotazování. Syntaxe typu *Tuple* je uvedena v bloku 13.

```
Tuple{část1.typ = hodnota, část2.typ = hodnota, ...}
```

Blok 13 – Pseudokód. Syntaxe typu Tuple

Příklad pojmenovaného *Tuple* pro lékaře by mohl vypadat např. takto: `LekarTuple : Tuple{jmeno: String = 'Karel Páv', vek: Integer = 50}`

Přístup k jednotlivým částem *Tuple* probíhá přes tečkovou notaci. Pro vypsání hodnoty věku z předchozího příkladu lze tedy jednoduše napsat `LekarTuple.vek`.

Vlastní

Vlastním typem se kromě níže uvedených typů OCL stává i typ definovaný v příslušném modelu, nad kterým je jazyk OCL nasazen. Je-li např. vytvořena třída *A*, typ *A* se automaticky stává vlastním typem jazyka OCL a je na něj možné aplikovat všechny operace pro vlastní typy definované.

Základními vlastními typy jazyka OCL jsou [2]:

- *OclAny* – instance metatypu *AnyType*, která figuruje jako nadtyp pro všechny ostatní typy. Klasifikátory definované v modelu dědí všechny operace *OclAny* začínající prefixem *ocl*, aby se pokud možno předešlo konfliktům v pojmenování.
- *OclType* – zastupuje jakýkoli vestavěný OCL typ.
- *OclMessage* – zastupuje zprávy, které si mezi sebou objekty zasílají, včetně jejich parametrů. Lze použít pouze v *post-condition*.
- *OclState* – je velmi podobný výčtovému typu. Nejsou na něm definovány žádné operace. Používán je pouze v rámci operace *OclInState*, kde odkazuje na konkrétní stav ve stavovém automatu (použitelné pro stavový diagram).
- *OclInvalid* – instance metatypu *InvalidType*. V modelu jej reprezentuje jeho instance *invalid* (ve starších verzích se jmenovala *OclUndefined*), která se používá v případě nastání výjimečných situací (např. dělení nulou).
- *OclVoid* – instance metatypu *VoidType*, který stejně jako *OclInvalid* disponuje jedinou instancí *null*. Ta je používána pro reprezentaci absence hodnoty (původně úloha instance *invalid*, než došlo ze strany OMG k rozšíření specifikace).

Operace vlastních typů

Typ *Classifier* v následujícím výčtu představuje typ klasifikátoru konkrétního metamodelu nebo modelu podle toho, na jaké úrovni jsou příslušné OCL výrazy vyhodnocovány.

- **self.oclIsNew() = Boolean**
  Může být použito pouze v definici *post-condition*. Vrací *true*, pokud je *self* vytvořen v průběhu operace, kterou *post-condition* popisuje.
- **self.oclIsUndefined() = Boolean**
  Vrací *true*, pokud je *self* roven *invalid* nebo *null*.
- **self.oclIsInvalid() = Boolean**
  Vrací *true*, pokud je *self* roven *invalid*.
- **self.oclAsType(type : Classifier) = T**
  Provede přetypování *self* na typ *type*. Vrácen je typ *T* z množiny klasifikátorů metamodelu, jehož je daný model instancí. Není-li přetypování možné, je vráceno *invalid*.
- **self.oclIsTypeOf(type : Classifier) = Boolean**
  Vrací *true*, pokud je *self* typu *type*, ale nikoli již jeho podtypem.
- **self.oclIsKindOf(type : Classifier) = Boolean**
  Vrací *true*, pokud je *self* typu *type* nebo jeho podtypem.
- **self.oclInState(statespec : OclState) = Boolean**
  Vrací *true*, pokud je *self* ve stavu definovaném parametrem *statespec* (týká se pouze stavových diagramů).
- **self.oclType() = Classifier**
  Vrací typ *self*.
- **self.allInstances() = Set(T)**
  Vrací množinu všech aktuálně existujících instancí typu *self*.

2.3.6 Kolekce

Kolekce hrají v OCL významnou roli. Specifikace jazyka uvádí poměrně širokou paletu operací s nimi. V současné době panuje jistá míra disparity mezi oficiální specifikací a implementací těchto operací v podání modelovacích nástrojů podporujících OCL.

OCL nabízí čtyři typy šablon pro kolekce:

- *Bag* – nesetříděná, povoluje duplicity,
- *Set* – nesetříděná, nepovoluje duplicity,
- *Sequence* – setříděná, povoluje duplicity,
- *OrderedSet* – setříděná, nepovoluje duplicity.

Kolekcí se automaticky stává jakýkoli atribut či parametr operace, který má násobnost vyšší než jedna. V takovém případě je pro něj možné nastavit dvojici booleovských atributů *IsOrdered* a *IsUnique* určujících, zda je kolekce setříděná a zda je v ní povolen výskyt duplicit. Na základě těchto atributů je pak odvozen výsledný typ kolekce (viz tabulka 1). [1]

Tabulka 1 – Odvození OCL kolekcí dle atributů IsOrdered a IsUnique [1]

IsOrdered	IsUnique	Kolekce
false	*false*	*Bag*

false	*true*	*Set*
true	*false*	*Sequence*
true	*true*	*OrderedSet*

Operace s kolekcemi lze rozdělit na pět typů:

- **přístupové** – přímý přístup k prvkům je podporován pouze setříděnými kolekcemi typu *Sequence* a *OrderedSet*,
- **zjiťovací** – extrahují požadované informace z cílové kolekce,
- **porovnávací** – porovnává se cílová kolekce s jinou kolekcí, nikoli objekty v rámci kolekce vůči sobě navzájem,
- **konverzní** – umožňují přetypování cílové kolekce,
- **výběrové** – vracejí nadmnožinu nebo podmnožinu cílové kolekce,
- **iterační** – postupně procházejí prvky v kolekci a aplikují na ně definovaný OCL výraz.

V případě, že je volána operace měnící obsah kolekce, je vrácena nová kolekce a původní zůstává nezměněná.

Následují popisy operací podle výše uvedené taxonomie doplněné o vysvětlující příklady. Není-li uvedeno jinak, je operace použitelná pro všechny typy kolekcí. Jako *T* je označován typ prvku, pro který je kolekce určena.

Přístupové operace

Všechy přístupové operace jsou definovány pouze pro kolekce typu *Sequence* a *OrderedSet*. Indexuje se od 1. Pokud je index mimo rozsah kolekce, je vrácena instance typu *OclUndefined*.

- **self->at(index : Integer) = T**
  Vrací prvek kolekce *self* na pozici *index*.
  `Sequence{'a', 'b', 'c'}->at(1) = 'a'`
- **self->indexOf(object : T) = Integer**
  Vrací pořadí prvku *object* kolekce *self*. . Neobsahuje-li *self* požadovaný prvek, je vráceno *null*.
  `Sequence{'a', 'b', 'c'}->indexOf('c') = 3`
- **self->first() = T**
  Vrací první prvek kolekce *self*. Je-li *self* prázdná, je vráceno *null*.
  `Sequence{'a', 'b', 'c'}->first() = 'a'`
- **self->last() = T**
  Vrací poslední prvek kolekce *self*. Je-li *self* prázdná, je vráceno *null*.
  `Sequence{'a', 'b', 'c'}->last() = 'c'`

Zjišťovací operace

- **self->size() = Integer**
  Vrací počet prvků v *self*.
  `Bag{1, 2, 3}->size() = 3`
- **self->includes(object : T) = Boolean**
  Vrací *true*, pokud je *object* součástí kolekce *self*, jinak *false*.
  `Bag{1, 2, 3}->includes(1) = true`
- **self->excludes(object : T) = Boolean**
  Vrací *true*, pokud *object* není součástí kolekce *self*, jinak *false*.
  `Bag{1, 2, 3}->excludes(1) = false`
- **self->count(object : T) = Integer**
  Vrací počet výskytů *object* v koleci *self*.
  `Bag{1, 2, 2}->count(2) = 2`
- **self->includesAll(c2 : Collection(T)) = Boolean**
  Vrací *true*, pokud kolekce *self* obsahuje všechny prvky kolekce *c2*, jinak *false*.
  `Bag{1, 2, 2}->includesAll(Bag{2, 2, 1}) = true`
- **self->excludesAll(c2 : Collection(T)) = Boolean**
  Vrací *true*, pokud kolekce *self* neobsahuje ani jeden z prvků kolekce *c2*, jinak *false*.
  `Bag{1, 2, 2}->excludesAll(Bag{2, 2, 1}) = false`
- **self->isEmpty() = Boolean**
  Vrací *true*, pokud je kolekce *self* prázdná, jinak *false*.
  `Bag{}->isEmpty() = true`
- **self->notEmpty() = Boolean**
  Vrací *true*, pokud kolekce *self* není prázdná, jinak *false*.
  `Bag{}->notEmpty() = false`
- **self->max() = T**
  Vrací maximální prvek kolekce *self*. Kolekce musí být definována pro prvky typu *T* podporující operaci *max*.
  `Bag{1, 2, 3}->max() = 3`
- **self->min() = T**
  Vrací minimální prvek kolekce *self*. Kolekce musí být definována pro prvky typu *T* podporující operaci *min*.
  `Bag{1, 2, 3}->min() = 1`
- **self->sum() = T**
  Vrací součet prvků kolekce *self*. Typ prvků kolekce musí podporovat operátor součtu '+'.
  `Bag{1, 2, 3}->sum() = 6`

Porovnávací operátory

- **self: Collection(T) = c2 : Collection(T) = Boolean**
  Vrací *true*, pokud kolekce *self* obsahuje stejné prvky jako kolekce *c2*, jinak *false*.
  Je-li jedna z kolekcí setříděná, je navíc kontrolováno vzájemné pořadí prvků z obou kolekcí.
  Je třeba dát pozor na různé vlastnosti jednotlivých typů kolekcí. Následující porovnání jsou platnými tvrzeními.

```
Set{1, 2} = Set{1, 1, 2}
OrderedSet{1, 2, 2} = OrderedSet{1, 2, 1}
```

- **self: Collection(T) <> c2 : Collection(T) = Boolean**
  Vrací *true*, pokud kolekce *self* obsahuje nestejné prvky jako kolekce *c2*, jinak *false*. Je-li jedna z kolekcí setříděná, je navíc kontrolováno vzájemné pořadí prvků z obou kolekcí.
  Následující porovnání jsou platnými tvrzeními.
  ```
  Bag{1, 2} <> Set{1, 2}
  Sequence{1, 2} <> OrderedSet{1, 2}
  OrderedSet{1, 2, 3} <> OrderedSet{1, 3, 2}
  ```

Konverzní operace

- **self->asSet() = Set(T)**
  Přetypuje kolekci *self* na typ *Set* (duplicity jsou odstraněny).
  ```
  Bag{1, 1, 2}->asSet() = Set{1, 2}
  ```
- **self->asOrderedSet() = OrderedSet(T)**
  Přetypuje kolekci *self* na typ *OrderedSet* (duplicity jsou odstraněny).
  ```
  Bag{1, 1, 2}->asOrderedSet() = OrderedSet{1, 2}
  ```
- **self->asSequence() = Sequence(T)**
  Přetypuje kolekci *self* na typ *Sequence*.
  ```
  Bag{1, 1, 2}->asSequence() = Sequence{1, 1, 2}
  ```
- **self->asBag() = Bag(T)**
  Přetypuje kolekci *self* na typ *Bag*.
  ```
  Set{1, 2, 3}->asBag() = Bag{1, 2, 3}
  ```
- **self->flatten() = Collection(T)**
  Rekurzivně zploští strukturovanou kolekci *self* na jednoúrovňovou. Pokud *self* není strukturovaná kolekce, je vrácena v původní podobě.
  ```
  Bag{Set{Bag{'string', 2, 3}}, Sequence{OrderedSet{2, 3, 1}}}->flatten() =
  Bag{1, 2, 3, 2, 3, 'string'}
  ```

Výběrové operace

- **self->union(c2 : Collection(T)) = Collection(T)**
  Provede sjednocení kolekce *self* s kolekcí *c2*. *c2* může být pouze typu *Bag* nebo *Set* s výjimkou *self* jako typu *Sequence*, kde *c2* musí být také typu *Sequence* . Pokud je *self* typu *Sequence* nebo *OrderedSet*, proběhne zároveň setřídění položek výstupní kolekce.
  ```
  Bag{1, 2, 3}->union(Set{1, 2, 3}) = Bag{1, 1, 2, 2, 3, 3}
  ```
- **self->intersection(c2 : Collection (T)) = Collection(T)**
  Provede průnik kolekce *self* s kolecí *c2*. *c2* může být pouze typu *Bag* nebo *Set*. Tato operace není definována pro typ kolekce *Sequence*. Pokud je *self* typu *OrderedSet*, proběhne zároveň setřídění položek výstupní kolekce.
  ```
  Bag{1, 2, 3}->intersection(Set{1, 2}) = Set{1, 2}
  ```
- **self->including(object : T) = Collection(T)**
  Vrací novou kolekci *self* rozšířenou o prvek *object*.
  ```
  Bag{1, 2, 3}->including(4) = Bag{1, 2, 3, 4}
  ```

- **self->excluding(object : T) = Set(T)**
  Vrací novou kolekci *self*, ze které je vyřazen prvek *object*.
  `Bag{1, 2, 3}->excluding(3) = Bag{1, 2}`
- **self->symmetricDifference(set : Set(T)) = Set(T)**
  Vrací novou kolekci *self* jako symetrický rozdíl s kolekcí *set* (z nové kolekce jsou vyloučeny prvky společné pro kolekce *self* a *c2*). Tato operace je definována pouze pro kolekce typu *Set*.
  `Set{1, 2, 3}->symmetricDifference(Set{3, 4, 5}) = Set{1, 2, 4, 5}`
- **self->product(c2: Collection(T2)) = Set(Tuple(first: T, second: T2))**
  Vrací kartézský součin prvků kolekce *self* a kolekce *c2* jako kolekci typu *Set*.
  `Bag{1, 2}->product(Bag{1, 2}) = Set{Tuple{first = 1, second = 1}, Tuple{first = 1, second = 2}, Tuple{first = 2, second = 1}, Tuple{first = 2, second = 2}}`
- **self->append(object: T) = Collection(T)**
  Vrací novou kolekci *self*, na jejíž konec je přidán prvek *object*. Tato operace je definována pouze pro kolekce typu *Sequence* a *OrderedSet*.
  `Sequence{1, 2, 3}->append(4) = Sequence{1, 2, 3, 4}`
- **self->prepend(object: T) = Collection(T)**
  Vrací novou kolekci *self*, na jejíž začátek je přidán prvek *object*. Tato operace je definována pouze pro kolekce typu *Sequence* a *OrderedSet*.
  `Sequence{1, 2, 3}->prepend(4) = Sequence{4, 1, 2, 3}`
- **self->insertAt(index : Integer, object : T) = Collection(T)**
  Vrací novou kolekci *self*, na jejíž *index* je přidán prvek *object*. Tato operace je definována pouze pro kolekce typu *Sequence* a *OrderedSet*. Indexuje se od 1.
  `Sequence{'a', 'b', 'c'}->insertAt(2, 'd') = Sequence{'a', 'd', 'b', 'c'}`
- **self->subOrderedSet(lower : Integer, upper : Integer) = Collection(T)**
  Vrací podmnožinu kolekce *self* vymezenou indexy *lower* a *upper* představující dolní a horní mez. Tato operace je definována pouze pro kolekce typu *Sequence* a *OrderedSet*.
  `OrderedSet{'a', 'b', 'c'}->subOrderedSet(2, 3) = OrderedSet{'b', 'c'}`

Iterační operace

Procházení kolekcí zajišťují iterační operace, jejíž syntaxe je zachycena v bloku 14 [3].

```
<kolekce>-><operace>(<varIterátor>:<typ> |
                        <výrazIterátoru>
                    )
```

Blok 14 – Pseudokód. Syntaxe iterační operace

`<varIterátor>` a `<typ>` jsou nepovinné. Kromě již definovaných iteračních operací, jako např. *exists*, *isUnique* nebo *forAll*, lze vytvořit vlastní iterační rutinu s pomocí OCL operace *iterate* (viz blok 15) [3].

```
<kolekce>->iterate(<varIterátor>:<typ>;
             <výsledek>:<typVýsledku> = <inicializačníVýraz> |
<výrazIterátoru>)
```

Blok 15 – Pseudokód. Syntaxe operace iterate

`<varIterátor>` a `<typ>` jsou tentokrát již povinné údaje. Proměnná `<výsledek>` je inicializována přes `<inicializačníVýraz>`. Při procházení je pak každý prvek kolekce načten do proměnné `<varIterátor>`, na kterou se použije `<výrazIterátoru>` a výstup uloží do `<výsledek>`. Operace *iterate* pak vrátí konečnou hodnotu proměnné `<výsledek>`.

Příkladem může být spojení jednotlivých písmen do slova.

```
Sequence{'a', 'h', 'o', 'j'}->iterate(letter : String; word : String = '' |
word + letter) = 'ahoj'
```

- **self->select(expr : OclExpression) = Collection(T)**
  Vrací novou kolekci prvků z původní kolekce *self*, nad kterou provede výběr na základě booleovského výrazu *expr*.
  ```
  Bag{1, 2, 3}->select(i : Integer | i >= 2) = Bag{2, 3}
  ```
- **self->reject(expr : OclExpression) = Collection(T)**
  Vrací novou kolekci prvků z původní kolekce *self*, nad kterou provede inverzní výběr na základě booleovského výrazu *expr*.
  ```
  Bag{1, 2, 3}->reject(i : Integer | i >= 2) = Bag{1}
  ```
- **self->any(expr : OclExpression) = T**
  Vrací právě jeden prvek kolekce *self*, který odpovídá zadanému booleovskému výrazu *expr*. Je-li takových prvků víc, je vrácen v případě setříděné kolekce první z nich. V případě nesetříděné kolekce je vrácen prvek nedefinovaným způsobem.
  ```
  Sequence{1, 2, 3}->any(i : Integer | i >= 2) = 2
  ```
- **self->one(expr : OclExpression) = Boolean**
  Vrací *true*, pokud kolekce *self* obsahuje právě jeden prvek vyhovující booleovskému výrazu *expr*, jinak *false*.
  ```
  Sequence{1, 2, 3}->one(i : Integer | i = 2) = true
  ```
- **self->exists(expr : OclExpression) = Boolean**
  Vrací *true*, pokud kolekce *self* obsahuje alespoň jeden prvek vyhovující booleovskému výrazu *expr*, jinak *false*.
  ```
  Sequence{1, 2, 3}->exists(i : Integer | i = 2) = true
  ```
- **self->forAll(expr : OclExpression) = Boolean**
  Vrací *true*, pokud je booleovský výraz *expr* platný pro všechny prvky kolekce *self*, jinak *false*. Pro prázdnou kolekci je vždy vráceno *true*.
  ```
  Bag{1, 2, 3}->forAll(i : Integer | i > 0) = true
  ```

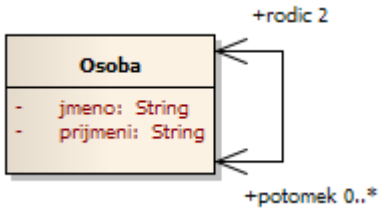
  Lze použít více iteračních proměnných, které vytvoří kartézský součin prvků kolekce *self* se sebou sama, na základě čehož pak proběhne vyhodnocování výrazu *expr*. Ukázkový příklad prověří, zda mají všechny nestejné dvojice prvků typu *String* shodné délky.
  ```
  Bag{'a', 'b', 'c'}->forAll(a : String, b : String | a <> b implies
  a->size() = b->size()) = true
  ```

- **self->isUnique(expr : OclExpression) = Boolean**
  Vrací *true*, pokud je booleovský výraz *expr* platný pro nejvýše jeden prvek kolekce *self*, jinak *false*.
  ```
  Bag{}->isUnique(i | i) = true
  Bag{1, 2, 3}->isUnique(i : Integer | i) = true
  Bag{Tuple{first = 1, second = 2}, Tuple{first = 1, second =
  2}}->isUnique(first) = false
  ```
- **self->collect(expr : OclExpression) = Collection(T)**
  Vrací kolekci prvků, které vzniknou aplikováním výrazu *expr* na jednotlivé prvky kolekce *self*. Na výslednou kolekci je před návratem aplikována operace *flatten*.
  ```
  Sequence{Tuple{first = 'a', second = 'd'}, Tuple{first = 'b', second =
  'c'}}->collect(second) = Sequence{'d', 'c'}
  ```
- **self->collectNested(expr : OclExpression) = Collection(T)**
  Pro každý prvek kolekce *self* vrací kolekci prvků, které vyhovují výrazu *expr*. Pro nesetříděnou kolekci *self* vrací kolekci typu *Bag*, pro setříděnou pak *Sequence*. Je respektováno zanoření prvků. Jednotlivé kolekce jsou uloženy do výstupní kolekce (opět buď *Bag*, nebo *Sequence*).
  Pro účely příkladu se předpokládá, že existuje třída *Pes* s referencí *potomek*.
  Operace je provedena na kolekci *psi* typu *Bag* obsahující prvky *pes1*, *pes2*, přičemž platí `pes1.potomek = Bag{'Max', 'Koudy'}, pes2.potomek = Bag{'Mr. White'}`.
  Operace *collectNested* na kolekci *psi* proběhne následovně:
  ```
  psi->collectNested(potomek) = Bag{Bag{'Max', 'Koudy'}, Bag{'Mr. White'}}
  ```
- **self->closure(expr : OclExpression) = OrderedSet(T)**
  Jedná se o operaci umožňující rekurzivně procházet objekt *self* přes výraz *expr*. Odpovídající prvky jsou kumulativně ukládány do výstupní kolekce typu *OrderedSet*.
  Vyjdeme z předchozího příkladu u operace *collectNested*, ale tentokrát kolekce *psi* obsahuje trojici psů *Max*, *Koudy* a *Mr. White* jako *pes1*, *pes2*, *pes3*, kteří jsou v určeném pořadí ve vztahu prarodič, rodič, syn.
  ```
  pes1->closure(potomek) = OrderedSet{pes2, pes3}
  ```
- **self->sortedBy(expr : OclExpression) = Collection(T)**
  Vrací novou kolekci *self* setříděnou podle kritéria výrazu *expr*. Tato operace je definována pouze pro kolekce typu *Sequence* a *OrderedSet*.
  ```
  Sequence{Tuple{first = 'a', second = 'd'}, Tuple{first = 'b', second =
  'c'}}->sortedBy(second) = Sequence{Tuple{first = 'b', second = 'c'},
  Tuple{first = 'a', second = 'd'}}
  ```
- **self->reverse() = Collection(T)**
  Otočí pořadí prvků kolekce *self*. Tato operace je definována pouze pro kolekce typu *Sequence* a *OrderedSet*.
  ```
  Sequence{1, 2, 3}->reverse() = Sequence{3, 2, 1}
  ```

2.3.7 Shrnutí

Uvedený materiál by měl pomoci čtenáři orientovat se v kódu OCL a usnadnit mu jeho pochopení ve zbylé části textu diplomové práce.

Důsledkem rozmanitosti jazyka OCL je skutečnost, že konkrétní omezení lze velmi často zapsat více způsoby. Mějme třídu *Osoba* s reflexivní asociací zachycující vztah rodič–potomek (viz obrázek 3).



Obrázek 3 – Třída Osoba. Zdroj: autor

Omezení, že konkrétní osoba nemůže být sama sobě potomkem, je pak možné zapsat minimálně následujícími způsoby:

a) `not self.potomek->includes(self)`
b) `self.potomek->excludes(self)`
c) `(self.potomek->select(osoba : Osoba | osoba = self))->size() = 0`
d) `(self.potomek->select(osoba : Osoba | osoba = self))->isEmpty()`
e) `Set{self}->intersection(self.potomek)->isEmpty()`
f) `(self.potomek->reject(osoba : Osoba | osoba <> self))->isEmpty()`
g) `self.potomek->forAll(osoba : Osoba | osoba <> self)`
h) `not self.potomek->exists(osoba : Osoba | osoba = self)`

Na místě by mohla být polemika, zda je uvedená rozmanitost jazyka OCL pro strojové zpracování výhodná, či nikoli.

Je třeba také připomenout fakt, že sémantika OCL není zcela striktní, následkem čehož se lze u jednotlivých interpretů setkat s mírně odlišným chováním některých operací. Při práci ve zvoleném OCL prostředí je tedy vždy nutné zjistit případné odchylky od oficiální specifikace [2] a brát je v úvahu.

Nevýhody

OCL bohužel nenabízí žádné typy ani operace pro práci s datem a časem, což lze považovat za poměrně významný nedostatek. OCL taktéž nenabízí ani podporu generických typů. Jako negativní vlastnost je také nutné vnímat fakt, že výraz v *post-condition* lze cílit pouze na popis stavu instance daného klasifikátoru po provedení příslušné operace. Důsledkem je odstínění od okolních instancí, což zásadně oslabuje aparát popisu chování komplexnějších operací. Tento jev je znám pod termínem *frame problem* [8]. OCL se s touto skutečností snaží částečně vypořádat zavedením operace *allInstances*, která je schopna vrátit všechny instance zvoleného klasifikátoru. Tato operace je mírně problematická, neboť není standardní součástí většiny dnešních programovacích jazyků, ve kterých může být OCL interpret realizován. Konsorcium OMG ji tedy implementačně ponechává jako volitelnou [2].

2.4 Možnosti využití jazyka OCL

Role OCL se postupem času více uplatňovala. V současné době je již jeho záběr poměrně široký, byť pro některé kategorie úloh již existují propracovanější řešení v podobě alternativních deklarativních, dotazovacích či jiných jazyků (viz kapitola 2.8).

Integritní omezení

Integritní omezení definují podmínky, které musejí být v každém případě splněny ze strany dat systému. Softwarový systém je zodpovědný za to, že data jsou konzistentní v souladu s těmito pravidly. V případě porušení integritních omezení by data nemusela dostatečně přesně reflektovat zobrazovanou realitu, následkem čehož by softwarový systém nemusel fungovat podle předpokladů či mohl přestat fungovat zcela.

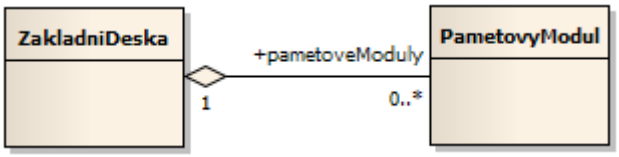
Integritní omezení lze v kontextu metamodelování naopak chápat jako soubor strukturálních omezení daný metamodelem nebo jeho profilem (viz Slovníček pojmů), které musí splňovat konkrétní model z něj vytvořený, aby byl validní.

Proti OCL integritním omezením lze testovat konkrétní instanci modelu (viz Slovníček pojmů), jak je ukázáno v kapitole 5.

Zvýšení informační hodnoty diagramů

Diagram v kombinaci s OCL výrazy zvyšuje svou informační hodnotu jakožto zprostředkovatel pohledu na model.

Mějme diagram tříd, který zachycuje základní desku počítače osazenou paměťovými moduly (viz obrázek 4).



Obrázek 4 – Diagram tříd základní desky. Zdroj: autor

Do základní desky je možné nainstalovat nejvýše tolik paměťových modulů, kolik má slotů. Tuto informaci není možné v diagramu tříd promítnout standardní grafickou notací. Analytik by sice mohl k diagramu připojit vhodnou poznámku, která by o této skutečnosti informovala[6], avšak tato by nemohla být dále strojově zpracována. Navíc by vlivem neformálnosti přirozeného jazyka mohlo dojít k nepřesnému vyložení informace jiným analytikem. V tomto případě je tedy vhodné zadefinovat OCL omezení popsané blokem 16.

```
context ZakladniDeska
inv slotCapacity: pocetPametovychSlotu >= pametoveModuly->size()
```

Blok 16 – OCL. Invariant definující integritní omezení pro základní desku

[6] OCL výraz lze vložit i do poznámky navázané k prvku modelu (pokud se jedná o omezení, vkládá se do složených závorek). V takovém případě již není nutné uvádět kontext, protože je zřejmý z příslušnosti poznámky k prvku modelu. Tento přístup, dle očekávání, zamezí možnosti strojového zpracování OCL.

OCL může pomoci zpřesnit model, ovšem za cenu zvýšení člověkohodin na jeho tvorbu vyhrazených. Je tedy vždy na místě posoudit charakter a rozsah modelovaného systému a zda by nasazení jazyka OCL v daném případě přineslo dostatečnou užitnou hodnotu. Lze říci, že hlavní podstata OCL spočívá v přesnějším modelování (viz Slovníček pojmů).

Validace modelu

Je-li definice metamodelu či jeho profilu doplněna o OCL, lze výsledný model pod ním vytvořený zvalidovat za pomocí k tomu uzpůsobených CASE nástrojů. Lze se tak dozvědět, které elementy modelu nejsou v konzistentním souladu s definovanými OCL omezeními, a provést nápravu. Na tomto využití OCL staví aplikace OCL generátor, která byla vytvořena v rámci kapitoly 6.

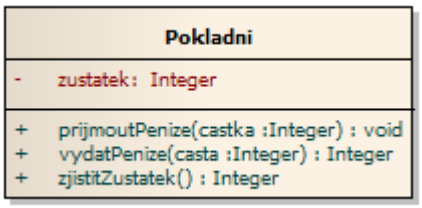
Navigace v modelu a dotazování se nad ním

Pomocí jazyka OCL je možné se uvnitř modelu navigovat a formulovat dotazy s cílem zjistit požadované skutečnosti o něm. Lze např. určit, kolik se v modelu nachází abstraktních tříd, nebo získat přehled klasifikátorů, které realizují nějaké konkrétní rozhraní. Stejný přístup lze uplatnit i na instanci modelu v případě, že daný modelovací nástroj podporuje její programové vytvoření. V takovém případě dotazování probíhá nad konkrétními objekty.

Pomocí navigačních výrazů se lze v OCL odkazovat na [3]:

- klasifikátory,
- atributy,
- konce asociací,
- vyhledávací operace.

Mějme ukázkovou třídu reprezentující pokladní (viz obrázek 5).



Obrázek 5 – Třída Pokladní. Zdroj: autor

Navigace se provádí přes tečkovou notaci. V kontextové instanci *Pokladni* se lze na atribut *zustatek* odkázat buď pomocí výrazu `self.zustatek` anebo pouze `zustatek`. Stejný princip platí pro navigování se na klasifikátory, konce asociací[7] a vyhledávací operace.

[7] Specifikem jazyka OCL je, že dokáže procházet i neprůchodné konce asociací [3]. Tato vlastnost platí v teoretické rovině, ovšem při strojovém zpracování OCL mezi jednotlivými instancemi musí existovat spojení (instance asociace) v požadovaném směru.

Dotazování pomocí OCL je ukázáno v kapitole 5. Zde je uveden jeden z možných způsobů využití. OCL výraz v bloku 17 vrátí všechny veřejné operace třídy *Pokladni* v UML modelu.

```
context Pokladni
ownedOperation->select(operation : Operation | operation.visibility =
VisibilityKind::public)
```

Blok 17 – OCL. Ukázka dotazu nad UML modelem

Generování kódu

Další devizou OCL je rozšíření schopnosti modelovacího nástroje generovat kód v požadovaném programovacím jazyce, kdy výstup reflektuje definovaná integritní omezení a popisy operací. OCL omezeními lze obohatit i relační datový model a při generování SQL z něj získat korespondující pohledy a triggery.

Operaci *vydatPenize* předchozí ukázkové třídy (viz obrázek 5) lze popsat výrazem v bloku 18.

```
context Pokladni::vydatPenize(castka : Integer) : Integer
pre: castka > 0 and zustatek >= castka
post: zustatek = zustatek@pre - castka
```

Blok 18 – OCL. Popis operace vydatPenize třídy Pokladni

Aby mohl pokladní peníze vydat, musí disponovat příslušnou peněžní sumou. Výraz *post* popisuje stav instance třídy po úspěšném provedení operace, kdy je již výsledná částka odečtena.

Blok 19 popisuje, jak by mohl vypadat zdrojový kód vygenerovaný pro operaci *vydatPenize* v jazyce Java 1.5 a vyšší s ohledem na definovanou *pre-condition* (konkrétní podoba závisí na použitém generátoru).

```
public Integer vydatPenize(Integer castka){
      if (castka <= 0 && zustatek < castka){
            return null;
      }
      ...
return castka;
}
```

Blok 19 – Java. Ukázka možné podoby vygenerované metody vydatPenize třídy Pokladni

Modelové transformace

OCL může doplnit daný metamodel o integritní omezení (a učinit jej tak přesným), což lze využít u modelových transformací. Modelovou transformací se rozumí konverze jednoho typu modelu na druhý. Aby se tato mohla uskutečnit, je nutné nejprve provést mapování prvků mezi zdrojovým a cílovým metamodelem. Výstupem transformace je pak podoba vstupního modelu v cílovém metamodelu. Na tomto přístupu staví metodika MDA (více v kapitole 2.6), kdy transformace obvykle probíhají mezi heterogenními modely (modely

jsou instancemi nestejných metamodelů). Transformace může být provedena také mezi homogenními modely. Rovněž lze několik modelů sloučit do jednoho nebo naopak z jednoho získat více výstupů. [9]

OCL v Executable UML (xtUML)

Executable UML, známé pod zkratkou xtUML, je profil jazyka UML. Základní vlastností xtUML modelů je možnost kompilace do zdrojového kódu, čímž se zajistí jejich spustitelnost. Z xtUML těží hlavně stavové diagramy a diagramy aktivit na úrovni objektů. [10]

Nasazením OCL v xtUML se zabývá například čínská vývojová korporace Hitachi, která navrhla rozšíření OCL o možnost měnit stav systému. Specifikace je popsána v dokumentu [11].

OCL se intenzivně zabývají také německé univerzity. V roce 2012 bylo univerzitou v Brémách představeno rozšíření v podobě jazyka Simple OCL-based Imperative Language (zkráceně SOIL), jehož použití se však omezuje jen na převážně výukový nástroj USE a navíc k němu ještě není vytvořena uživatelská dokumentace. [12]

Nasazení OCL u spustitelných modelů je poměrně mladá disciplína, ve které se stále ještě formují specifikace a vyvíjejí potřebné nástroje. Žádná z xtUML OCL specifikací zatím nebyla oficiálně schválena konsorciem OMG. Jazyk OCL pro tyto účely původně nebyl určen, a tak je dle potřeby rozšiřován. Měněna je zejména jeho prvotní podstata z jazyka deklarativního na imperativní, případně je využívána kombinace deklarativního a imperativního přístupu.

2.5 OCL v UML diagramech

OCL je možné nasadit u všech třech rodin UML diagramů s různou mírou užitečnosti. Jsou uvedeny pouze ty, u kterých se aplikace OCL jeví podle [3] jako vhodný doplněk:

a) **strukturální diagramy** (diagram tříd),
b) **diagramy interakce** (sekvenční diagram),
c) **diagramy chování** (diagram aktivit, stavový diagram).

U diagramů interakce a chování se OCL výrazy většinou vkládají přímo do grafické podoby (viz obrázky 8, 10 a 14). Nadměrné užívání OCL, případně velmi složité výrazy, může významným způsobem snížit čitelnost jednotlivých diagramů. Užití OCL je tedy v tomto případě vždy na zvážení analytika a na cílovém publiku, pro které jsou diagramy určeny.

2.5.1 Diagram tříd

Diagram tříd (viz Slovníček pojmů) nachází pro jazyk OCL bezesporu největší uplatnění:

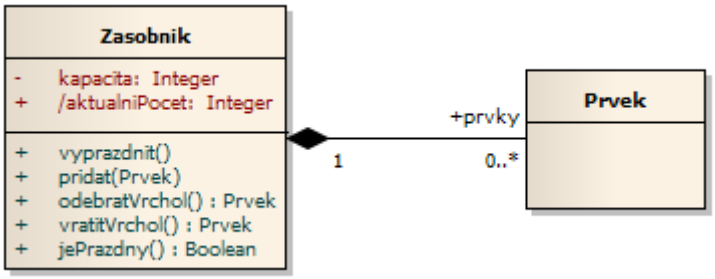
- Výrazem *inv* lze popsat integritní omezení.

- Pomocí *derive* lze vytvářet odvozené třídní atributy, které budou disponovat zvolenou funkční složkou.
- Prostřednictvím *init* lze přiřadit výchozí hodnoty třídním atributům či koncům asociací.
- Lze popisovat operace ve smyslu metodiky Design by Contract za pomocí výrazů *pre* a *post*.
- Tělo dotazovací operace (v UML je vlastnost operace *isQuery* nastavena na hodnotu *true*) lze popsat výrazem *body*. Pouze tento typ operace může být OCL interpretem spouštěn.
- V rámci tříd lze navigovat.

Platí, že diagram musí být již dostatečně přesný, aby v něm OCL mohlo být použito. Musí tedy být určeny typy atributů, signatury operací a názvy konců asociací. Z toho vyplývá, že např. diagramy analytických tříd, u kterých je běžnou praxí výše zmíněné neuvádět, z OCL mohou těžit jen málo.

Příklad zásobníku

Následuje demonstrace výše zmíněného na příkladu implementace datové struktury zásobník (viz obrázek 6). Operace třídy *Zasobnik* jsou popsány jazykem OCL, který formálně vystihuje požadavky kladené na jejich spuštění. Popisuje taktéž, jak se pro každou úspěšně vykonanou operaci změní stav konkrétní instance zásobníku. Dále jsou popsány atributy zásobníku. Formální popis celé datové struktury je uveden v bloku 20.



Obrázek 6 – Diagram tříd zásobníku. Zdroj: autor

```
-- Prvky v zásobníku jsou řazeny v sekvenci
context Zasobnik
inv stackElementsInSequence:
prvky.oclIsTypeOf(Sequence)

-- Odvozený atribut aktualniPocet vrací aktuální počet prvků v zásobníku
context Zasobnik::aktualniPocet : Integer
derive: prvky->size()

-- Kapacita zásobníku je inicializována na 10
context Zasobnik::kapacita : Integer
init: 10

-- Operace vyprazdnit
context Zasobnik::vyprazdnit() : OclVoid
post: prvky->isEmpty()
```

```
-- Operace pridat
context Zasobnik::pridat(prvek : Prvek) : OclVoid
pre: kapacita > aktualniPocet
post: prvky->size() = prvky@pre->size() + 1
      and
      prvky->first() = prvek

-- Operace odebratVrchol
context Zasobnik::odebratVrchol() : Prvek
pre: aktualniPocet > 0
post: prvky->size() = prvky@pre->size() - 1
        and
        prvky->excludes(prvky@pre->first())

-- Dotazovací operace vratitVrchol
context Zasobnik::vratitVrchol() : Prvek
pre: aktualniPocet > 0
body: prvky->first()

-- Dotazovací operace jePrazdny
context Zasobnik::jePrazdny() : Boolean
body: prvky->isEmpty()
```

Blok 20 – OCL. Formální popis datové struktury zásobník

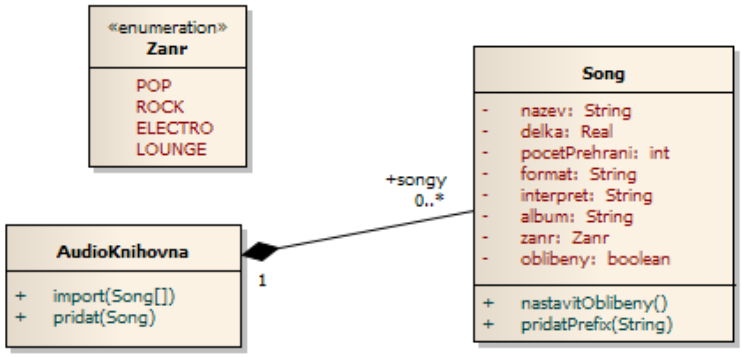
2.5.2 Diagram interakce

V diagramu interakce (viz Slovníček pojmů) je možné OCL využít pro [3]:

- kontrolní podmínku přechodu,
- určení selektoru[8] pro čáru života (lifeline),
- určení parametru zprávy.

Příklad audio knihovny

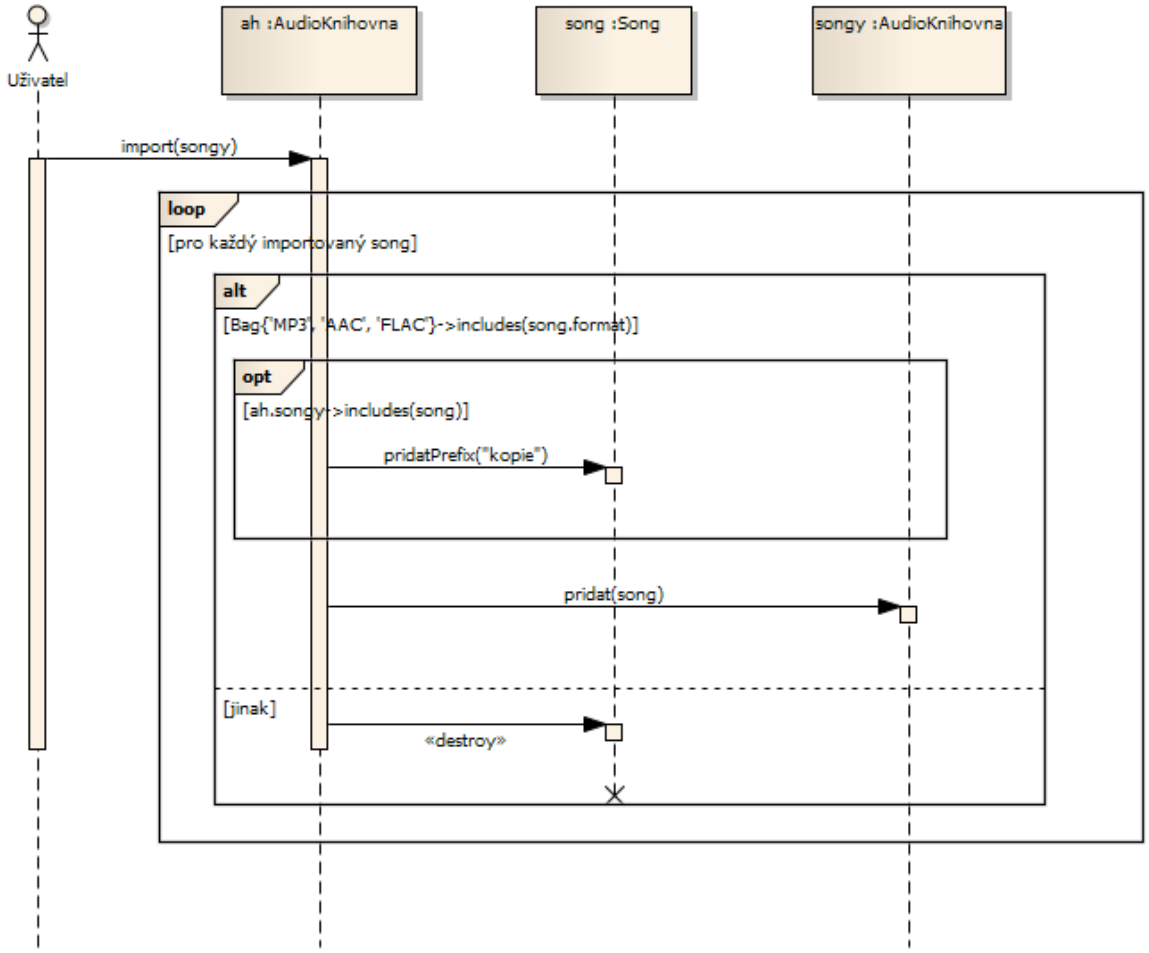
Mějme příklad audio knihovny, jejíž struktura je zachycena diagramem tříd na obrázku 7.



Obrázek 7 – Diagram tříd audio knihovny. Zdroj: autor

[8] Selektorem se rozumí výraz, který blíže definuje účastníka interakce za předpokladu, že čára života neznázorňuje jednu, ale více entit. Pokud selektorem není specifikován konkrétní účastník, předpokládá se účast náhodné entity [46]. Má-li např. třída *Banka* kolekci *klienti* a v diagramu interakce chceme znázornit klienta Petra Černého, lze selektor u lifeline pomocí OCL definovat takto: `klienti[self->select(jmeno = 'Petr' and prijmeni = 'Černý')]:Banka`.

Import kolekce songů do audio knihovny, při kterém se vyřadí položky s nepodporovaným formátem a pro duplicitní položky přidá odpovídající prefix, lze znázornit sekvenčním diagramem (viz Slovníček pojmů). OCL výrazy jsou vloženy do kombinovaných fragmentů *alt* a *opt* (zápsány v hranatých závorkách, viz obrázek 8). Splněný booleovský výraz `Bag{'MP3', 'AAC', 'FLAC'}->includes(song.format)` u fragmentu *alt* znamená, že právě zpracovávaný song je v podporovaném formátu a tedy bude přidán do audio knihovny. Detekci, zda je importovaný song již v knihovně, zajistí výraz `ah.songy->includes(song)`. V takovém případě je k názvu songu navíc přidán prefix indikující, že jde o kopii.



Obrázek 8 – Sekvenční diagram audio knihovny. Zdroj: autor

2.5.3 Diagram aktivit

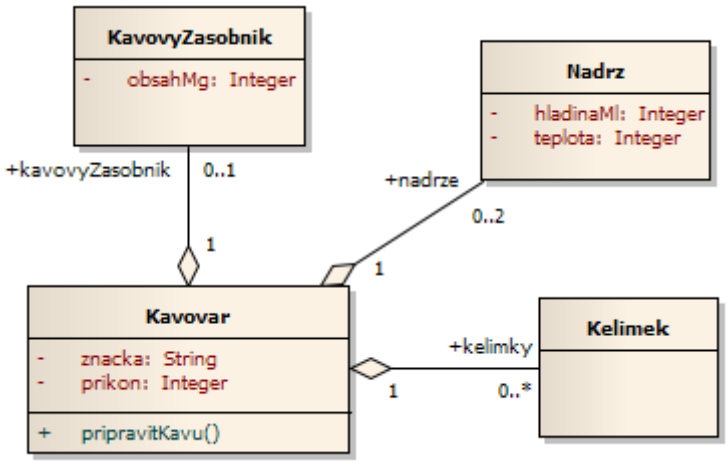
V diagramu aktivit (viz Slovníček pojmů) lze OCL využít pro určení [3]:

- akčních uzlů,
- kontrolních podmínek přechodů,
- objektových uzlů,
- operací a argumentů,
- definici stavů objektů.

Arlow ve své knize [3] zpochybňuje přínos OCL pro diagramy aktivit s odůvodněním, že diagram aktivit je ze své podstaty nepřesný. Tento typ diagramu je primárně cílen na popis pracovních postupů (workflow), byznys procesů a procedurální logiky, proto se může jevit vhodnější popsat např. podmínky přechodů mezi jednotlivými aktivitami přirozeným jazykem.

Příklad kávovaru

Mějme příklad kávovaru disponující dvěma vyměnitelnými nádržemi na vodu. Struktura kávovaru je zachycena diagramem tříd na obrázku 9.



Obrázek 9 – Diagram tříd kávovaru. Zdroj: autor

Operaci *pripravitKavu* třídy *Kavovar* lze znázornit diagramem aktivit s využitím jazyka OCL (viz obrázek 10). Kávovar kontroluje přítomnost potřebných komponent, což je zachyceno ve strukturované aktivitě *Kontrola komponent*. Výrazy `nadrze->isEmpty()`, `kelimky->isEmpty()` a `(kavovyZasobnik.oclIsUndefined()) or kavovyZasobnik.obsahMg < 40` rozhodují o případné absenci dané komponenty (či nedostatečném objemu kávy pro přípravu). Platnost kteréhokoli z uvedených výrazů operaci *pripravitKavu* ukončí.

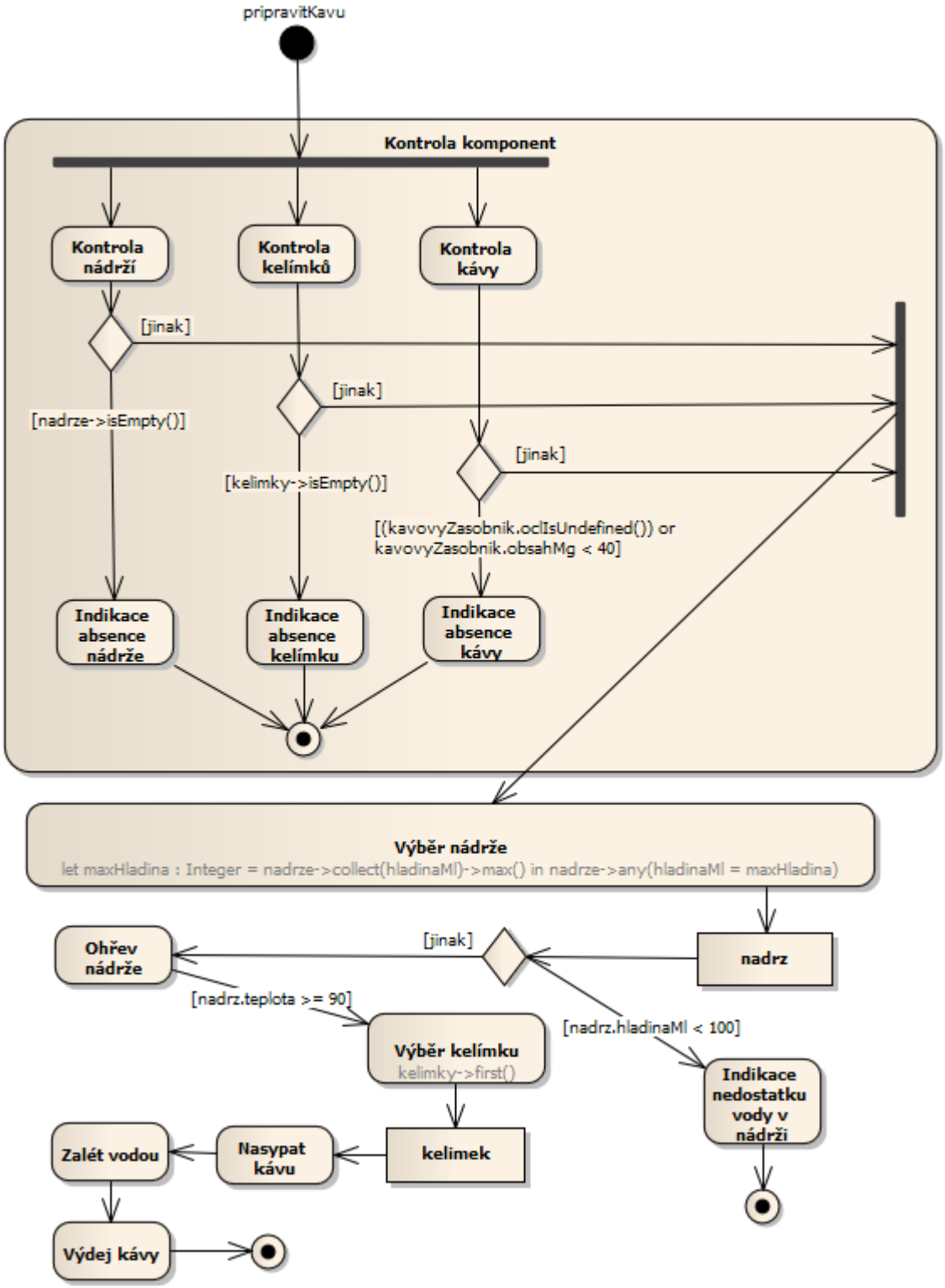
Po uspěšné kontrole komponent je třeba vybrat vhodnou vodní nádrž. To zajistí výraz `let maxHladina : Integer = nadrze->collect(hladinaMl)->max() in nadrze ->any(hladinaMl = maxHladina)`. Vrácena je nádrž s vyšší hladinou vody (v případě, že jsou instalovány obě nádrže a jejich hladina je stejná, je proveden náhodný výběr). Pokud však vybraná nádrž neobsahuje alespoň 100 ml vody (detekováno výrazem `nadrz.hladinaMl < 100`), operace skončí.

Dále je voda v nádrži zahřívána na teplotu 90 °C. Aktivita *Ohřev nádrže* skončí ve chvíli, kdy dojde k platnosti výrazu `nadrz.teplota >= 90`.

Posléze je vybrán první kelímek výrazem `kelimky->first()`. Aby tento výraz mohl být vyhodnocen, musí být kolekce *kelimky* setříděná. Musí tedy zároveň platit buď výraz `kelimky.oclIsTypeOf(Sequence)`, nebo `kelimky.oclIsTypeOf(OrderedSet)`. Tato

skutečnost se již v diagramu aktivit neuvádí, neboť náleží struktuře modelu, kterou zde zachycuje diagram tříd.

Po vykonání poslední trojice aktivit je operace *pripravitKavu* úspěšně ukončena.



Obrázek 10 – Diagram aktivit kávovaru, operace pripravitKavu. Zdroj: autor

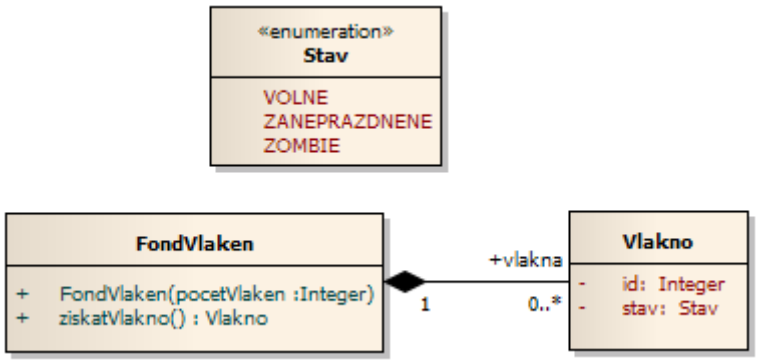
2.5.4 Stavový diagram

OCL je ve stavových diagramech (viz Slovníček pojmů) možné využít pro určení [3]:

- kontrolních podmínek,
- podmínek stavů,
- cílů akcí,
- operací a argumentů.

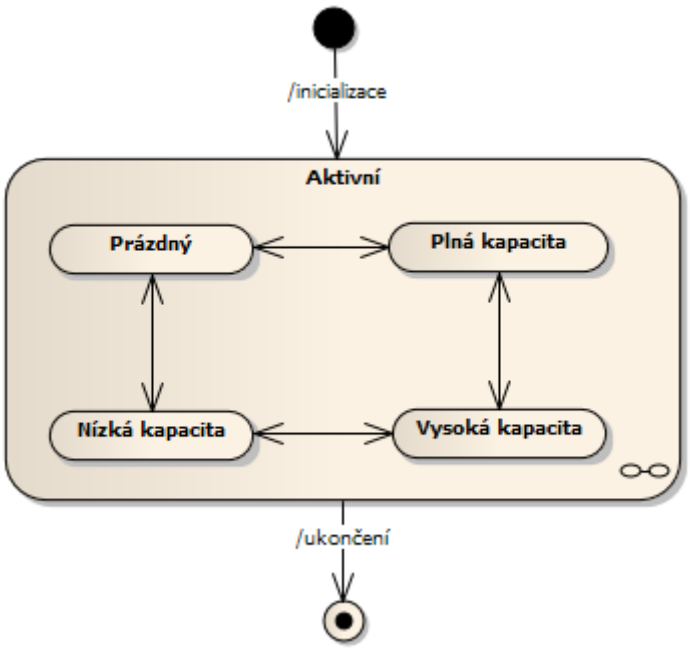
Příklad fondu vláken

Mějme fond vláken (thread pool), který obsahuje vlákna v určitých stavech. Struktura fondu vláken je znázorněna diagramem tříd na obrázku 11.



Obrázek 11 – Diagram tříd fondu vláken. Zdroj: autor

Stavy fondu z hlediska dostupnosti volných vláken lze zachytit prostřednictvím stavového diagramu na obrázku 12.



Obrázek 12 – Stavový diagram fondu vláken. Zdroj: autor

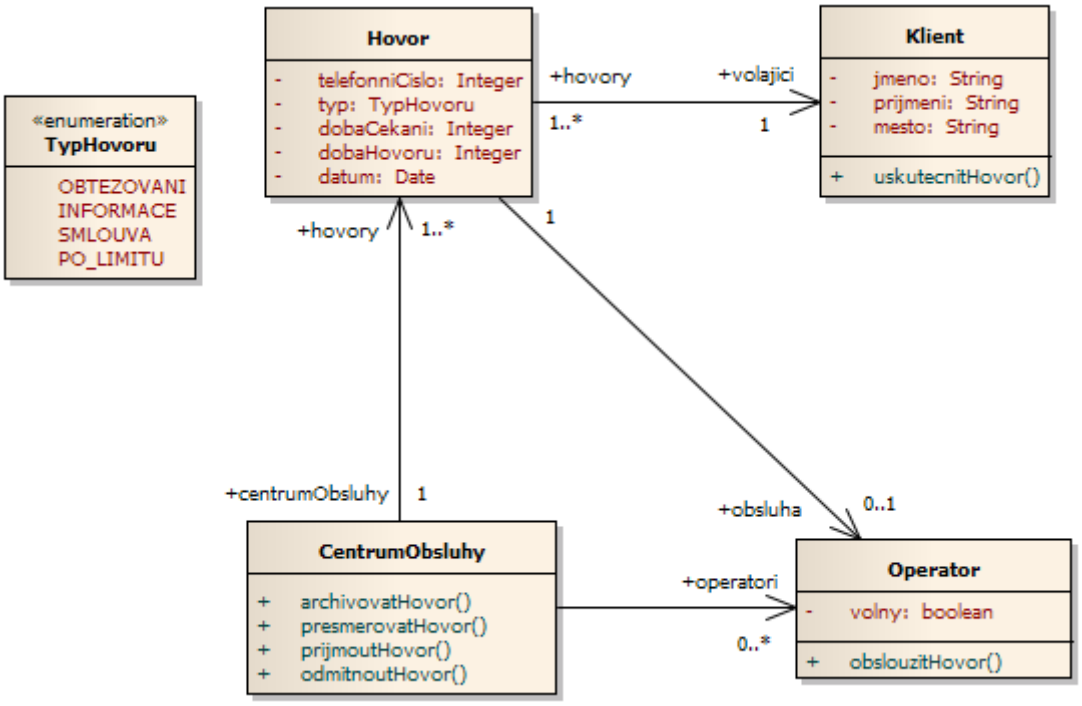
OCL lze v tomto případě využít pro přesnou definici jednotlivých stavů (viz blok 21).

```
context FondVlaken

-- Fond neobsahuje žádná volná vlákna
inv emptyState:
oclInState(Prázdný) implies vlakna->isEmpty()

-- Pomocná funkce, která zjistí počet vláken ve fondu dle zadaných stavů
def: getThreadCountByStates(states : Set(Stav)) : Integer =
vlakna->select(v : Vlakno | states->includes(v.stav))->size()

-- Ve fondu převažují zaneprázdněná a zombie vlákna
inv lowCapacityState:
let unavailableThreadCount : Integer =
getThreadCountByStates(Set{Stav::ZANEPRAZDNENE, Stav::ZOMBIE}),
freeThreadCount : Integer = getThreadCountByStates(Set{Stav::VOLNE}) in
oclInState(Nízká kapacita) implies unavailableThreadCount > freeThreadCount

-- Ve fondu převažují volná vlákna
inv highCapacityState:
let unavailableThreadCount : Integer =
getThreadCountByStates(Set{Stav::ZANEPRAZDNENE, Stav::ZOMBIE}),
freeThreadCount : Integer = getThreadCountByStates(Set{Stav::VOLNE}) in
oclInState(Vysoká kapacita) implies (freeThreadCount >= unavailableThreadCount
and unavailableThreadCount > 0)

-- Všechna vlákna ve fondu jsou k dispozici
inv fullCapacityState:
oclInState(Plná kapacita) implies (not vlakna->isEmpty() and vlakna->forAll
(v : Vlakno | v.stav = Stav::VOLNE))
```

Blok 21 – OCL. Formální definice stavů fondu vláken

Příklad call centra

Dále mějme příklad call centra, které provozuje pasivní telemarketing. Struktura call centra je znázorněna diagramem tříd na obrázku 13.



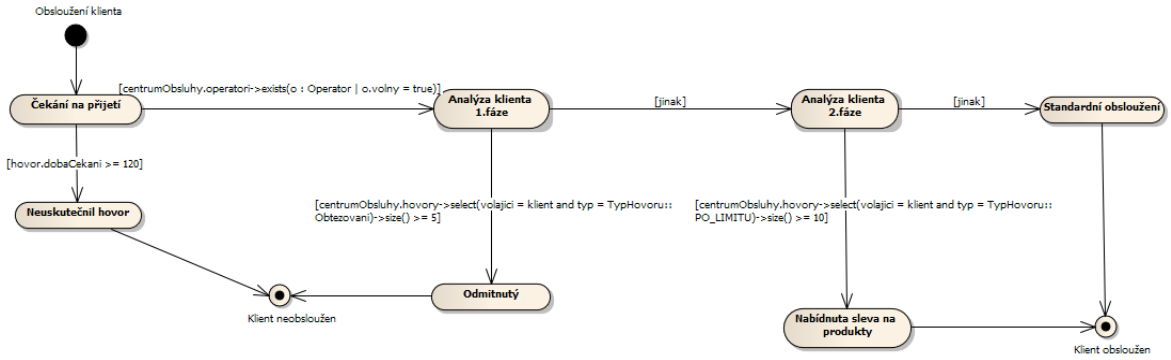
Obrázek 13 – Diagram tříd call centra. Zdroj: autor

Využití OCL v podobě kontrolních podmínek přechodů mezi stavy je demonstrováno na stavovém diagramu klienta, který uskutečňuje hovor do call centra (viz obrázek 14). Klient, který na lince čeká 120 a déle sekund, je nespokojený a telefon položí. Odpovídající přechod hlídá výraz `hovor.dobaCekani >= 120`.

Klient je obsloužen v případě, že je volný některý z operátorů. Tento přechod je hlídán výrazem `centrumObsluhy.operatori->exists(o : Operator | o.volny = true)`.

V průběhu první fáze analýzy klienta je zjišťováno, zda je klient veden jako notorický obtěžovač (klientovo číslo je spárováno s archivovanými záznamy, přičemž záznamy starší jednoho roku se automaticky mažou). Pokud tomu tak je, systém automaticky hovor ukončí. Odpovídající přechod je hlídán výrazem `centrumObsluhy.hovory->select (volajici = klient and typ = TypHovoru::Obtezovani)->size() >= 5`.

V druhé fázi analýzy klienta se ověřuje, zda tento neměl v minulosti větší počet neobsloužených hovorů. Uvedená skutečnost je zjištěna výrazem `centrumObsluhy.hovory ->select(volajici = klient and typ = TypHovoru::PO_LIMITU)->size() >= 10`. Pokud je výraz platný, obsloužení klienta se přesune do stavu, kdy je mu nabídnuta sleva na vybrané produkty společnosti.



Obrázek 14 – Stavový diagram klienta call centra. Zdroj: autor

2.6 OCL a Model Driven Architecture (MDA)

Při popisu jazyka OCL je vhodné zmínit softwarovou metodiku vývoje MDA [13] , pro kterou jsou deklarativní jazyky důležitým prvkem. MDA je neustále se rozvíjející standard spravovaný pod hlavičkou konsorcia OMG, který definuje sadu vývojových metod založených na dobře zavedených standardech UML, XMI, CORBA a dalších.

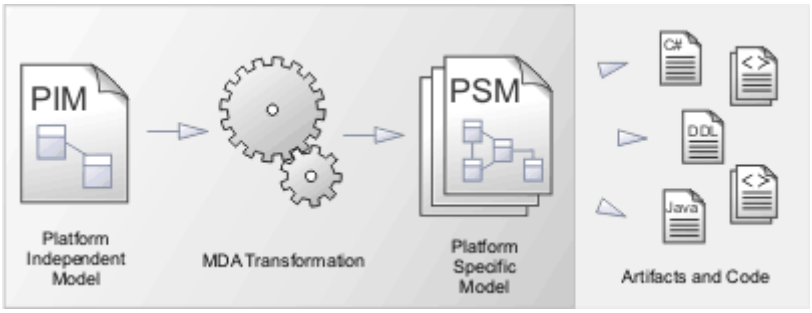
2.6.1 MDA

Základem metodiky MDA je model a jeho transformace, kdy v každé fázi vývoje softwarového produktu probíhá strojová přeměna ze zdrojového modelu na cílový. Touto cestou se projektový tým dostává z počátečního doménového modelu až ke zdrojovému kódu. Výhodami metodiky MDA je fakt, že zdrojový kód aplikace je vždy synchronizován s jejím modelem, a znovupoužitelnost modelů při generování kódu pro různé platformy.

MDA rozeznává 3 typy modelů [13]:

- Computer Independent Model (CIM),
- Platform Independent Model (PIM),
- Platform Specific Model **(**PSM).

Jsou-li k dispozici patřičné nástroje, je možné aplikovat reverzní inženýrství a ze zdrojového kódu transformacemi postupně získat modely typu PSM a PIM. Princip metodiky MDA je znázorněn na obrázku 15.



Obrázek 15 – Princip MDA [9]

Computer Independent Model

CIM, jinak známý také jako doménový model, pohlíží na modelovaný objekt ve smyslu obecných požadavků na systém, přičemž se nezaobírá detaily ani způsobem implementace. Vymezuje důležité oblasti systému, které budou programově realizovány. Tento model není povinný, přesto je vhodné ho mít pro získání ucelené představy o řešeném problému. Typicky se jedná o procesní model. [13]

Platform Independent Model

PIM, jinak známý také jako konceptuální nebo analytický model, vychází z doménového modelu a je taktéž platformově nezávislý. Popisuje logické části systému na úrovni algoritmů, principů a omezení. Transformaci z CIM do PIM jako jedinou nelze zajistit strojově. Vždy záleží na analytikovi, které prvky doménového modelu považuje za důležité (ty budou po stránce algoritmů a byznys procesů popsány). Výhodou tohoto modelu je zejména jeho znovupoužitelnost, neboť jej lze transformovat do modelu závislého na zvoleném implementačním jazyku dle aktuální potřeby. Tím lze snížit náklady pro vývoj aplikací určených pro odlišné implementační platformy. Proti uvedenému přístupu však hovoří skutečnost, že rozdíly mezi jednotlivými implementačními jazyky mohou být tak velké, že nebude možná jeho smysluplná realizace. Typicky se jedná o objektový model. [13]

Platform Specific Model

Jedná se o finální model MDA odrážející pohled na systém v návrhové rovině. PSM by měl obsahovat dostatek takových informací, aby mohl být transformován na zdrojový kód. Lze tedy říci, že PSM je jakousi formou vizualizace zdrojového kódu výsledné aplikace. Příkladem PSM může být návrhový model pro nějaký programovací jazyk nebo relační datový model pro konkrétní databázovou platformu. [13]

2.6.2 Role OCL v MDA

Modelování lze rozdělit podle vyspělosti, s jakou přistupuje k zachycení modelovaného systému, do šesti úrovní [14]:

a) bez specifikace,
b) použití textové specifikace,
c) text s diagramy (občasné využití grafického modelovacího jazyka),
d) model s textovým popisem (převažuje využití grafického modelovacího jazyka),
e) přesný model (využití grafického modelovacího jazyka společně s deklarativním jazykem),
f) pouze model.

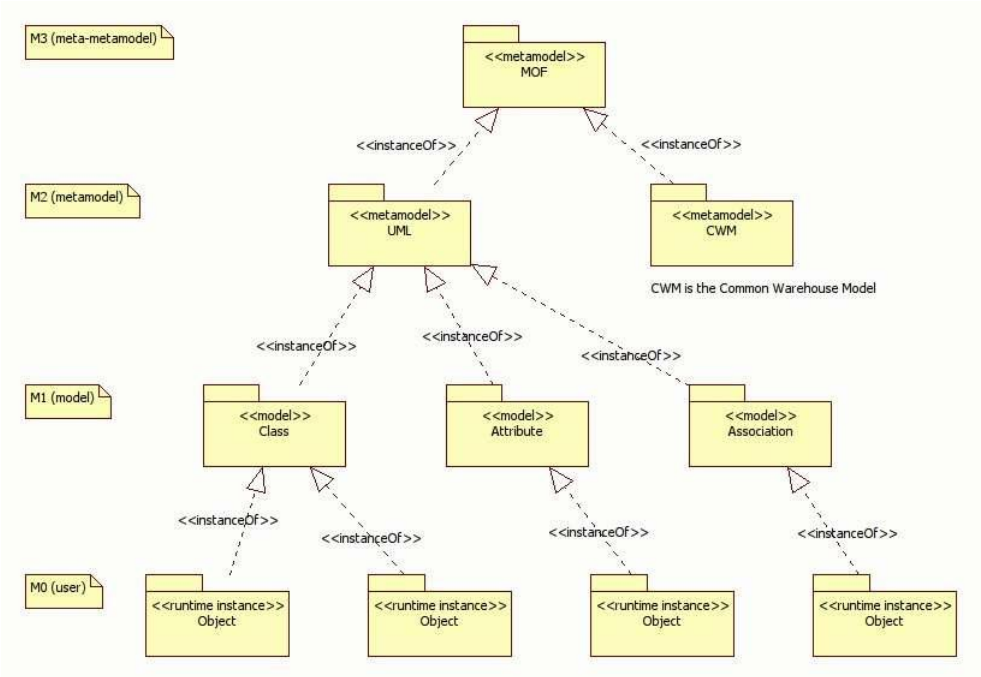
Poslední případ (viz f) staví modelovací jazyky na úroveň vyšších programovacích jazyků a její dosažení je prozatím utopií (model je již tak přesný a detailní, že umožňuje kompletní generování kódu ve zvoleném programovacím jazyce bez zásahu programátora). Než se však dosažení této úrovně stane realitou, zůstává cílem vytvoření modelu přesného do takové míry, aby umožňoval přímé spojení se zdrojovým kódem výsledné aplikace. Na tom se obvykle podílí některý z formálních deklarativních jazyků (např. OCL).

OCL může metodice MDA přispět těmito způsoby [14]:

- Doplňuje **modelovací jazyky** (s jeho pomocí lze vytvořit sadu omezení a pravidel platnou pro daný metamodel).
- Schopností definovat **přesnější modely**, což vede k rozsáhlejším možnostem v oblasti generování kódu z PSM.
- Jako základ **modelových transformací**.

2.7 Meta Object Facility

Meta Object Facility (MOF) je standard konsorcia OMG, který popisuje framework pro spravování modelových metadat. Jedná se o uzavřenou architekturu definující 4-úrovňovou hierarchii (jednotlivé úrovně jsou označeny shora M3, M2, M1, M0), kde elementy *i-té* úrovně tvoří instance elementů úrovně $i - 1$ s výjimkou M3, jejíž elementy popisují samy sebe. Původně vznikl jako reakce na potřebu prostředku, který by definoval metamodel UML (v tomto ohledu se někdy používá pojem *meta-metamodel*). [15], [16]



Obrázek 16 – Úrovně architektury MOF [17]

2.7.1 Čtyři úrovně MOF

Definice jednotlivých úrovní MOF podle obrázku 16 je následující:

- **M3** – meta-metamodel, který slouží pro specifikaci abstraktní syntaxe modelovacích jazyků.
- **M2** – metamodely pro specifické modelovací jazyky, které definují jejich strukturu. Např. UML popisuje typy diagramů, typy elementů, které se v nich mohou vyskytovat, a také vazby, které mezi nimi mohou vznikat.

- **M1** – jedná se o model vyvíjeného systému. V případě použití UML se na této úrovni nacházejí instance prvků metamodelu UML znázorněné konkrétními diagramy (diagram tříd, aktivit, …).
- **M0** – zde se již hovoří o vytvořeném systému. Do této úrovně tedy patří existující instance klasifikátorů (např. v UML by byly znázorněny diagramem objektů).

2.7.2 Základní model MOF

Konsorcium OMG definuje základní MOF model (označovaný jako CMOF) obsahující množinu konstrukcí pro objektově orientované modelování [18]:

- **Balíček** – kontejner pro modularizaci metamodelů podle jejich logických oblastí. Balíčky lze zanořovat do jiných balíčků do libovolné hloubky.
- **Třída** – konstrukt, jehož instance mají vlastní stav (definován pomocí atributů a konstant), identitu (představuje jedinečnou existenci objektu v čase a prostoru, která jej odlišuje od ostatních objektů) a rozhraní (definováno pomocí operací).
- **Asociace** – konstrukt, který popisuje binární vztahy mezi dvěma třídami, případně unární vztah u třídy samotné.
- **Datový typ** – typ, který nelze instantizovat. Může se jednat o primitivní datové typy jako *Integer, Boolean* nebo strukturované typy, kolekce, výčtové typy apod.
- **Omezení** – pravidla, která určují, za jakých podmínek je model validní.

MOF obsahuje celou řadu specifikací, jako např. MOF Core, MOF XMI Mapping, MOF Query/View/Transformations a další. Jeho modulární architektura předpokládá využití těchto specifikací nezávisle na sobě. [15], [16]

MOF definoval některé základní principy MDA (např. model typu PIM) a byl také využit ke specifikaci několika standardních OMG metamodelů, jako např. UML, EAI a dalších. V současné době existují varianty Complete MOF (CMOF) a Essential MOF (EMOF). Hlavní rozdíl mezi nimi je takový, že CMOF definuje asociace jako samostatné entity, zatímco v EMOF se naviguje podle referencí, které vlastní třída. Praktický důsledek těchto odlišností je ukázán v popisu implementace nástroje OCL generátor, který byl vypracován v rámci diplomové práce, v kapitole 6. [15], [16]

2.7.3 OCL ve vztahu k MOF

Z pohledu 4-úrovňové architektury MOF lze využití jazyka OCL shrnout podle tabulky 2.

Tabulka 2 – OCL na jednotlivých úrovních MOF. Zdroj: autor

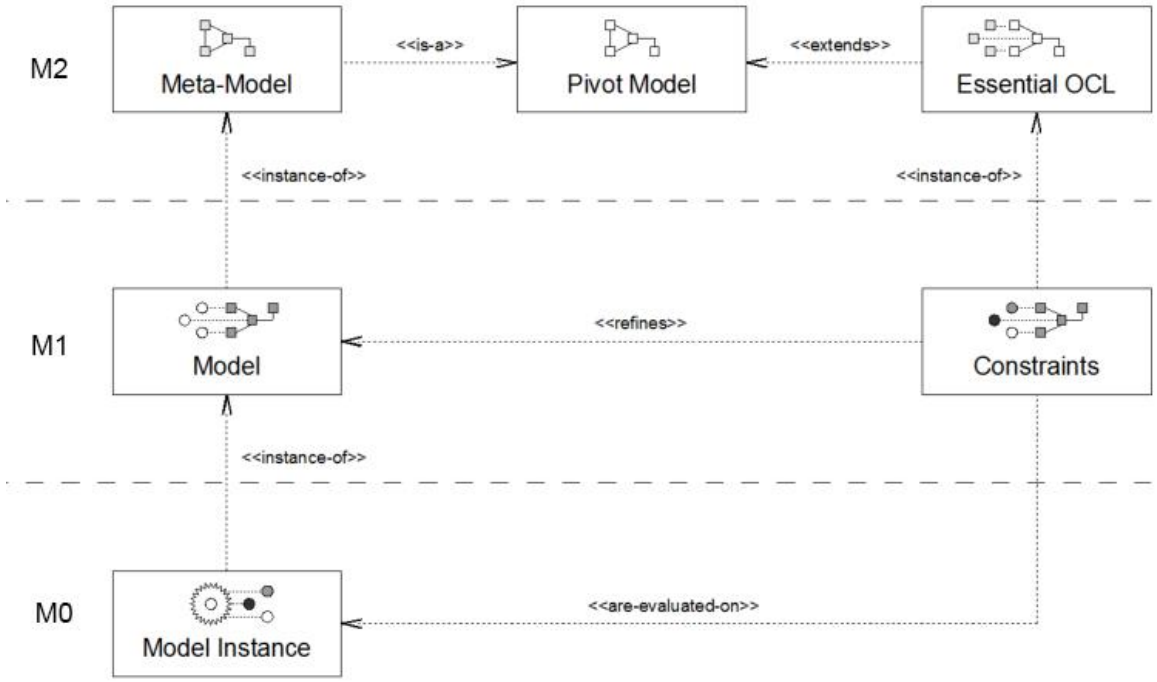
Úroveň MOF	OCL
M3	-
M2	Specifikace integritních omezení metamodelu či jeho profilu, specifikace modelovacích směrnic (vynucení určitých pravidel, které musí softwarový analytik při tvorbě modelu následovat).
M1	Validace modelu podle integritních omezení metamodelu či profilu metamodelu, specifikace

	integritních omezení modelu, dotazování se nad modelem.
M0	Validace instance modelu dle integritních omezení modelu, dotazování se nad instancí modelu.

Při tvorbě metamodelů se vychází z MOF při definici konceptů a vztahů cílové domény. OCL je zase používáno pro formalizaci jazyka metamodelu (např. skutečnost, že konec asociace v UML nesmí mít zápornou hodnotu násobnosti, je vyjádřena pomocí OCL). Jak dokládá studie Juana Cadavida [19], metamodely vybudované pomocí MOF jen zřídkakdy využívají OCL, a proto se stávají nepřesnými.

2.7.4 Vyhodnocování OCL omezení nad MOF modely

Z tabulky 2 je zřejmé, že OCL omezení lze definovat na úrovních M2 (metamodelu) a M1 (modelu). Aby bylo možné definovaná omezení zvolené úrovně ověřit, musí být daný CASE nástroj schopen pracovat buď s interpretativním nebo generativním přístupem vyhodnocování OCL výrazů [20]. Principy vycházejí z popisu funkcionality nástroje Dresden OCL (viz kapitola 4.3) pro platformu Eclipse. Jedná se však v zásadě o víceméně obecné přístupy k problému vyhodnocování OCL omezení. Obrázek 17 je podkladem pro vysvětlení obou přístupů.



Obrázek 17 – Podkladové schéma pro vyhodnocování OCL omezení nad MOF modely [20]

Interpretativní přístup

Interpretativní přístup ověřuje OCL omezení tak, že je interpretuje na modelu a jeho objektech. Lze jej shrnout do těcho kroků [20]:

1. Je vytvořen model (M1) popsaný prostřednictvím MOF metamodelu (M2). Zvolený metamodel je adaptován na tzv. *pivot model*. *Pivot model* je intermediární metamodel, jehož úkolem je přizpůsobit zvolený metamodel pro dané prostředí.
2. Model (M1) je doplněn o OCL omezení, která jsou definována na typech a operacích modelu. Předpokládána je dostupnost parseru, který ověří správnost syntaxe vytvořených OCL omezení.
3. Je vytvořena instance modelu (M0).
4. Interpret ověří definovaná OCL omezení modelu (M1) na instanci modelu (M0).
5. Výsledkem proběhlé interpretace je množina booleovských hodnot (*true* nebo *false*). Každé omezení je tedy vyhodnoceno jako booleovský výraz.

Generativní přístup

Generativní přístup spočívá v generování spustitelného zdrojového kódu z modelu, na kterém jsou OCL omezení ověřena. Skládá se z těchto kroků [20]:

1. Je vytvořen model (M1) popsaný prostřednictvím MOF metamodelu (M2), který je adaptován na *pivot model*.
2. Model (M1) je doplněn o OCL omezení, která jsou definována na typech a operacích modelu. Předpokládána je dostupnost parseru, který ověří správnost syntaxe vytvořených OCL omezení.
3. Je vygenerován nový model prostřednictvím šablon či transformačních pravidel, která jsou definována na základě jednotlivých elementů metamodelu (M2) a jejich omezení.
4. Výstupem je zdrojový kód, který lze spustit za účelem ověření definovaných OCL omezení.

Na těchto přístupech staví zejména nástroje modelovacího frameworku Eclipse EMF (viz kapitola 4.3).

2.8 Alternativy k OCL

OCL není jediným deklarativním jazykem, který lze pro popis modelu využít. Níže jsou představena dvě perspektivní alternativní řešení v podobě Epsilon project a Alloy.

2.8.1 Epsilon project

Informace v této podkapitole vycházejí z [21].

Epsilon je sada jazyků a nástrojů pokrývající širokou škálu požadavků pokročilých přístupů k objektově orientovaným modelům. Jádrem projektu je Epsilon Object Language (EOL). Epsilon využívá modelovací kapacity Eclipse Modeling Framework Project (viz kapitola 4.3).

Epsilon Object Language (EOL)

Jedná se o modelově orientovaný jazyk kombinující procedurální způsob programování podobný tomu u webového skriptovacího jazyka Javascript a dotazovací schopnosti

OCL, především co se práce s kolekcemi týče. EOL je zároveň imperativním jazykem umožňujícím, na rozdíl od OCL, psát výrazy, které mohou měnit stav systému.

Sekce kódu v bloku 22 vybere z Ecore modelu (viz kapitola 4.3) všechny abstraktní třídy a vypíše pro každou z nich její název a počet strukturálních členů, které ji tvoří (atributy, operace). Ze zápisu lze vidět, že konstrukce těla iterátoru v EOL a OCL je prakticky totožná.

```
for (c in EClass.all.select(c | c.abstract)) {
    var className = " " + c.name;
    className = className + "->" +
        c.eStructuralFeatures.size();
    className.println();
}
```

Blok 22 – EOL. Ukázka výpisu abstraktních tříd s počtem strukturálních členů

Epsilon Validation Language (EVL)

EVL je validační jazyk s velmi podobnou syntaxí, jakou má OCL. Oproti OCL umožňuje navíc vytvářet závislosti mezi jednotlivými omezeními, což umožňuje neprovádět vyhodnocení těch omezení, jejichž rodičovský prvek byl vyhodnocen jako neplatný.

Epsilon Transformation Language (ETL)

Jazyk pro definici transformací mezi modely. Umožňuje násobné vstupy i výstupy (např. transformovat jeden vstupní model na více výstupních modelů). Pomocí ETL lze také strojově převést objektově orientovaný model na relační datový model.

Epsilon Generation Language (EGL)

Jedná se o jazyk pro definici šablon, pomocí kterého lze vytvářet textové artefakty ze vstupního modelu. Výstupem může být např. HTML dokumentace či kostra zdrojového kódu modelovaného systému.

Epsilon Comparison Language (ECL)

ECL je jazyk zaměřený na hledání podobností mezi dvojicí vzájemně homogenních či heterogenních modelů. U homogenních modelů je úlohou najít překrývající se elementy tak, aby se v případě sjednocování ve výstupním modelu neprojevily jako duplicity. U heterogenních modelů je vytvoření mapování mezi elementy z jednotlivých modelů nutným předpokladem pro provedení transformací nebo sjednocování modelů.

Epsilon Merging Language (EML)

EML je jazyk vycházející z transformačního jazyka ETL určený ke sjednocování jak homogenních tak heterogenních modelů. Využívá se především v kombinaci s ECL.

2.8.2 Alloy

Informace v této podkapitole vycházejí z [22], [23].

Alloy je modelovací jazyk zaměřený na automatickou analýzu, jehož vývoj byl silně ovlivněn deklarativním jazykem Z, který vznikl v druhé polovině 80. let. Alloy je založený

na predikátové logice. První prototyp byl vyvinut v roce 1997 pod vedením prof. Daniela Jacksona na univerzitě Massachusetts Institute of Technology a po funkční stránce byl značně limitován. Postupem času se Alloy vyvinul do podoby jazyka schopného velmi dobře vyjádřit strukturální a behaviorální složku modelovaného systému. Oproti OCL má Alloy více konvenční syntaxi a jednodušší sémantiku.

Model úplného grafu

Možnosti jazyka Alloy lze demonstrovat na jednoduchém modelu úplného grafu (každý vrchol je spojen hranou s ostatními vrcholy grafu).

Nejprve je třeba definovat **signatury**, které představují strukturální složku modelu. V tomto případě je vytvořena signatura *Vertex* reprezentující vrchol grafu společně s jejím atributem *adjacentSet* jako množinou sousedních vrcholů (viz blok 23).

```
sig Vertex {
  adjacentSet : set Vertex
}
```

Blok 23 – Alloy. Definice signatur pro model úplného grafu

Dalším krokem je definice **faktů**. Fakta určují globálně platná omezení. Ve fázi simulování modelu bude Alloy Analyzer [9] vyřazovat ty instance, které porušují alespoň jeden stanovený fakt. Faktem v ukázkovém modelu je informace o existenci všech možných dvojic vrcholů grafu vyjma identických (viz blok 24). Dvojice stejných vrcholů by značila přítomnost smyčky.

```
fact {
  adjacentSet = Vertex -> Vertex - iden
}
```

Blok 24 – Alloy. Definice faktů pro model úplného grafu

Poté je definován **predikát**, který omezuje generování instancí. V tomto případě takové omezení není třeba, proto je predikát uveden prázdný (viz blok 25).

```
pred exec {}
```

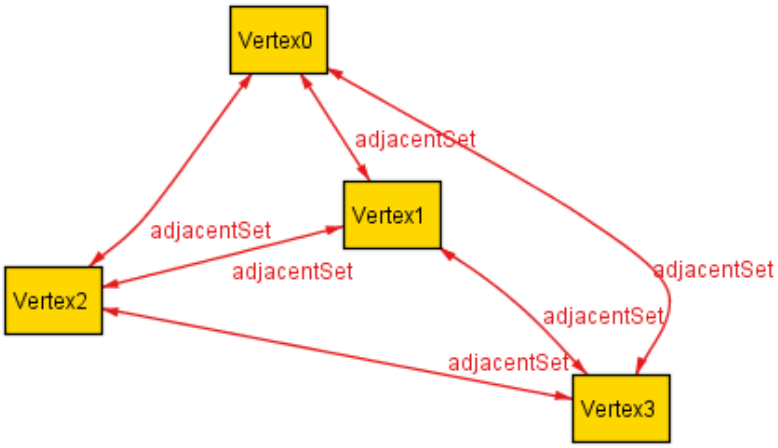
Blok 25 – Alloy. Definice predikátu pro model úplného grafu

Posledním krokem je samotné spuštění simulace modelu. Je třeba zvolit, pro kolik instancí tato proběhne (viz blok 26). Následně program Alloy Analyzer zobrazí grafický výstup, který je uveden na obrázku 18.

```
run exec for exactly 4 Vertex
```

[9] Vývojové grafické prostředí vybavené kompilátorem a analytickou komponentou pro jazyk Alloy.

Blok 26 – Alloy. Spuštění simulace modelu úplného grafu v programu Alloy Analyzer



Obrázek 18 – Simulace úplného grafu v Alloy Analyzer pro 4 instance. Zdroj: autor

3 Podpora jazyka OCL v Enterprise Architect

Cílem této kapitoly je zmapování podpory jazyka OCL v modelovacím nástroji Enterprise Architect (dále jen EA). Pozornost je věnována zejména verzi 9.3, která je poslední stabilní verzí [24] v době psaní práce. Uveden je také stav podpory OCL pro starší verze.

EA je nabízen v šesti různých edicích [25] od nejnižší Desktop až po Ultimate, která v sobě obsahuje veškerou funkcionalitu nižších edicí. Podpora OCL je dostupná ve stejném rozsahu již od edice Desktop ve všech nabízených edicích.

3.1 Přehled podpory OCL podle verzí

Údaje v této sekci vycházejí z oficiálního popisu geneze podpory OCL modelovacího nástroje Enterprise Architect společnosti Sparx Systems [24].

Verze 5.0

EA začalo s podporou OCL od verze 5.0 (konkrétně v buildu 764), kdy došlo k rozšíření stávajících typů omezení *invariant*, *pre-condition*, *post-condition* a *process* o *OCL constraints*. Ty bylo možné přidávat k UML klasifikátorům, atributům a operacím. Součástí byla také kontrola syntaxe.

Verze 6.0

Tato verze (build 778) by měla být z hlediska podpory OCL klíčová, protože podle Sparx Systems zavádí hned několik podstatných funkcí:

- podporu pro psaní OCL skriptů,
- možnost vyhodnocování OCL výrazů a reportování chyb,
- možnost validace UML modelu oproti OCL omezením.

Kromě toho má zavádět podporu OCL i pro hlavní UML profily[10], které EA nabízí.

Verze 6.5

Bugfix, který řeší chybu validace při iteraci OCL kolekce. Dále adresuje chybu v diagramu objektů, kde se OCL validace nemusela korektně vyhodnotit.

Verze 7.5

Kontrola OCL omezení rozšířena o podporu tvrzení typu *isStereotyped*[11].

Verze 8.0

Ukládání OCL výrazů jako HMTL entit vinou RTF editoru má být opraveno. Při testu exportu modelu do XMI se však HTML entity v OCL výrazech stále vyskytovaly.

Současná verze 9.3

[10] Profily rozšiřují možnosti modelování pro specifické domény (např. UML profil zaměřený na desktopové aplikace nabízí jinou kolekci rozšíření než profil pro webové aplikace).

[11] Není definováno v OMG specifikaci OCL [2]. Patrně ověřuje, zda je na element aplikován stereotyp.

Společnost Sparx Systems žádné další změny týkající se podpory OCL od verze 8 a výš neuvádí. Další část mapuje podporu OCL pro verzi 9.3.

3.2 Přidávání omezení do EA modelu

EA umožňuje přidat libovolný počet OCL omezení k následujícím prvkům:

- klasifikátor,
- asociace,
- atribut a operace.

OCL omezení lze přidávat k výše uvedeným prvkům přes formulářové rozhraní EA, nikoli však prostřednictvím nějakého vlastního OCL editoru do samostatného souboru, který by byl přidružen k modelu. Praktický důsledek je takový, že z popisu omezení je nutné vynechat klíčové slovo *context*, neboť kontext výrazu je pro EA zřejmý již ze skutečnosti, který UML element byl zvolen. Zadefinování kontextu naopak vede k selhání validace syntaxe.

Podpora přidávání OCL omezení k atributům je poněkud matoucí, protože podle specifikace nemůže být atribut vlastníkem omezení. Může mu však být přiřazena funkční složka pomocí *derive* nebo může být inicializován pomocí *init*.

U operací EA nabízí možnost uvést *pre-condition* a *post-condition*.

Způsob, jakým je v EA řešeno přidávání OCL omezení do modelu, není příliš dobrý. Důvodem jsou zvýšená nepřehlednost a vyšší časové nároky na správu. Při přejmenování daného prvku modelu nedochází k aktualizaci příslušných OCL omezení a je nutná manuální korekce. OCL se pro některé elementy píše v klasickém textovém editoru, který do výrazů při exportu modelu do XMI přidává dokonce znakové entity. Jinde je zase nutné se spokojit pouze s jednořádkovým polem, což velmi znesnadňuje psaní delších výrazů a znemožňuje jejich formátování.

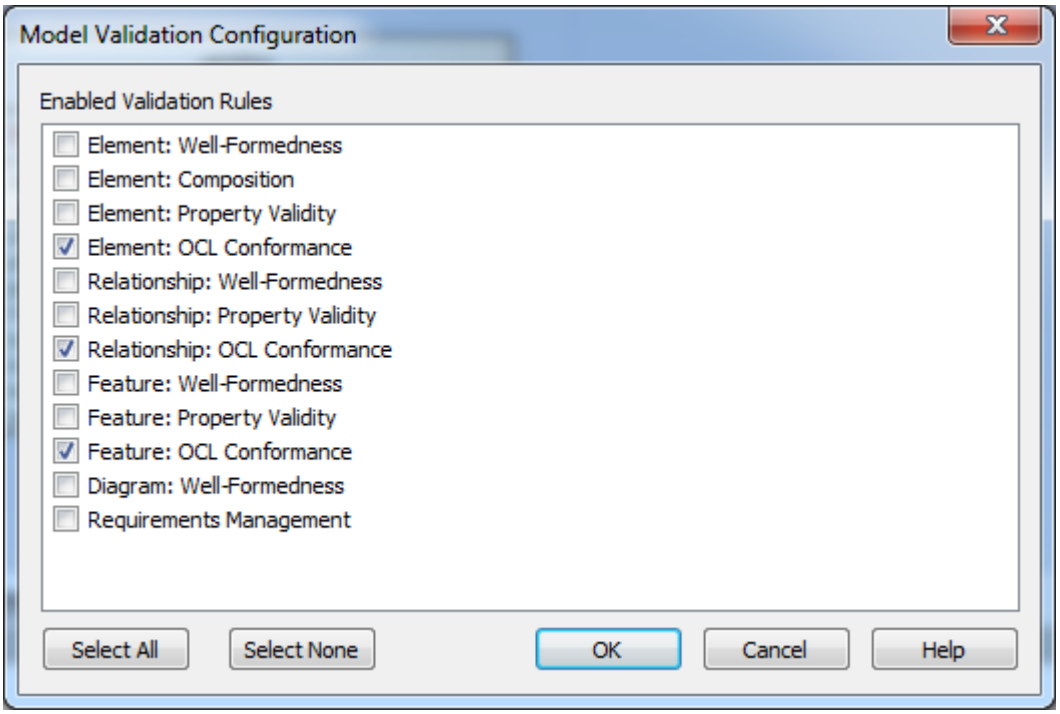
Validace modelu

EA nabízí validaci modelu oproti OCL ve čtyřech rozsazích:

- pro zvolený element (kontrole podléhá element samotný, jeho děti, atributy, operace a asociace k němu přidružené),
- pro zvolený diagram (zkontrolovány budou všechny elementy, které jsou v něm zobrazeny),
- pro zvolený balíček (zahrnuta rekurzivní kontrola všech podbalíčků v něm obsažených),
- pro celý model.

Validaci je možné spustit navigováním v hlavním menu přes *Project, Model Validation, Validate Selected* anebo pomocí klávesové zkratky <CTRL+ALT+V> nad vybraným objektem. Její výstup se poté zobrazí v dolní liště pod záložkou *Model Validation*

s kódovým označením *MVRX0001*, kde *X* je pořadí validačního skriptu v konfiguračním dialogu v hexadecimální podobě. Nutné je ověřit, zda je validace OCL zaškrtnuta v dialogovém okně *Model Validation Configuration* (viz obrázek 19).



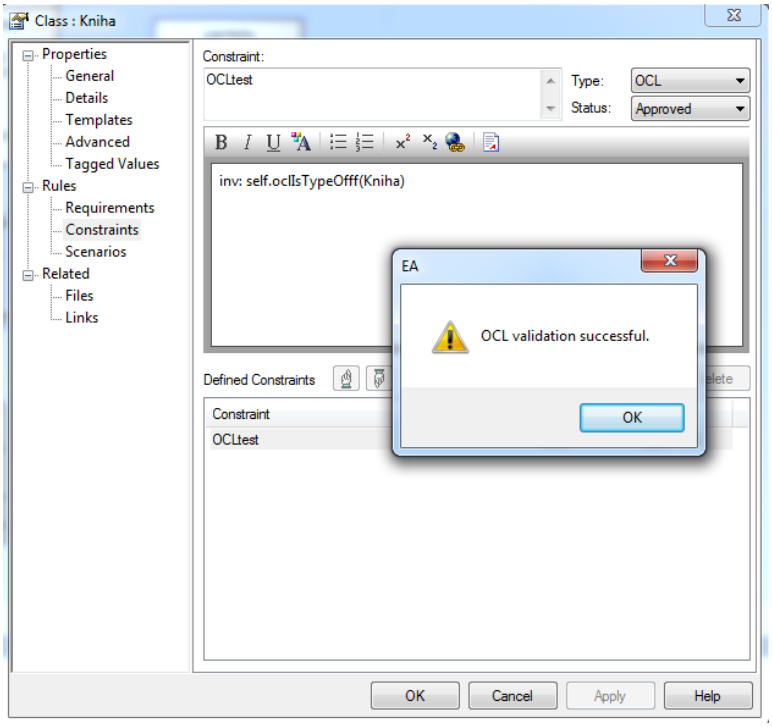
Obrázek 19 – Konfigurační okno validace modelu v EA 9.3. Zdroj: autor

3.3 Ověření podpory OCL v EA

V první řadě je třeba uvést, že Sparx Systems podporu OCL ve svém nástroji příliš nezdokumentovalo. K dispozici jsou pouze dvě velice stručné a obecné ukázky v nápovědě a výčet změn pro jednotlivé verze. Také oficiální internetové fórum EA [26] neskýtá mnoho informací týkajících se podpory OCL. Těch několik málo dotazů na ni je často ponecháno bez odpovědi vývojářů spolu s rozčarováním některých uživatelů, kteří měli o této funkcionalitě jiné představy.

Parsování syntaxe

EA poskytuje jednoduchý parser, který umožňuje zvalidovat syntaxi vkládaných OCL výrazů ihned po jejich uložení. Validací však projdou často nesmyslné výrazy (viz obrázek 20).



Obrázek 20 – EA OCL parser je v jádru velmi triviální. Zdroj: autor

Další testování ukázalo, že EA parser kontroluje pouze strukturu výrazu a nikoli jednotlivá klíčová slova, natož zda použité konce asociací dávají v kontextu klasifikátoru smysl. Přínos takového parseru pro psaní netriviálních výrazů není velký, a proto byla provedena rešerše existujících EA pluginů, případně účelových externích nástrojů, které by nedostatky původního OCL parseru dokázaly kompenzovat.

Oclarity

Oclarity je podpůrný nástroj zaměřený na přidávání OCL pravidel do modelu, který umožňuje naimportovat metadata modelu obsažená v XMI exportu daného modelovacího nástroje [27]. Předpokladem je dodržení specifikace XMI CASE nástrojem, ze kterého je export prováděn. Při testu importu modelu z EA do Oclarity ve formátu XMI byla metadata modelu přenesena korektně.

Oclarity kromě procházení naimportovaného modelu plně podporuje OCL 2.0 specifikaci a poskytuje zvýrazňování syntaxe, plnohodnotné parsování OCL pravidel a tím i statickou kontrolu modelu vůči přidaným OCL omezením. Výhodou tohoto nástroje je kromě jeho malé velikosti i fakt, že je společností EmPowerTec poskytován zdarma, byť s uzavřeným zdrojovým kódem.

Enterprise Analyst

Jedná se o komerční add-in pro EA vyvíjený společností CRAFTWARE Consultores Ltda. původem z Chile [28]. Nabízí statickou analýzu modelu, možnost kompilace diagramu tříd a stavových diagramů s cílem interpretace a otestování modelu. Pro psaní OCL výrazů

nabízí jednoduchý OCL expression builder. OCL lze v tomto add-inu [12] využít pro definování *pre* a *post-conditions,* pro definici dotazovacích operací u tříd a také pro dotazování nad vytvořenými objekty. Nenabízí však plnou podporu OCL podle aktuální specifikace OMG.

Vývoj Enterprise Analyst byl k datu 20.8. 2011 ukončen z důvodu přílišné restriktivity mateřské platformy EA, čímž vzniká riziko možné nekompatibility s budoucími verzemi EA. K dispozici je trial verze omezující maximální počet tříd v diagramu na deset. Plná verze je nabízena v ceně 199$ za licenci (lze nainstalovat pouze na jeden počítač) se slevou od pěti a více zakoupených licencí. [29]

Vyhodnocování OCL výrazů a omezení

EA podle dokumentace umožňuje vyhodnocování OCL omezení na úrovni metamodelu (M2). Tato funkcionalita byla otestována a shledána nefunkční. Pravděpodobně se jedná o ověření konzistence modelu oproti metamodelu UML, avšak tato je již vynucena samotným uživatelským rozhraním. Vhodné by to bylo v případě použití UML profilů doplněné o OCL omezení.

EA umožňuje vytvoření objektového diagramu. Pro vytvořené objekty však nelze vyhodnocovat integritní omezení v jazyce OCL na úrovni M1. Stejně tak nelze vyhodnocovat dotazovací operace popsané jazykem OCL pro jednotlivé objekty.

OCL dotazování

EA pro účely dotazování nenabízí žádné nástroje. Vhodná by byla například přítomnost OCL konzole.

Generování kódu

Definování *pre* a *post-condition* u operací nemá na generování kódu žádný vliv. Díky aplikaci vytvořeného studentem ČVUT FEL [30] lze však generovat SQL kód z UML modelu obohaceného o OCL na bázi exportovaného XMI souboru z EA.

MDA Transformace

OCL omezení jsou MDA transformacemi v EA ignorována.

Shrnutí

Současný stav podpory OCL v EA byl vyhodnocen jako neuspokojivý. Ani připravovaná verze 10 patrně žádné změny nepřinese (Sparx Systems se ve výčtu změn o OCL nezmiňuje) [31]. Z důvodů uvedených výše tedy nelze tento CASE nástroj v jeho současné podobě doporučit pro práci s jazykem OCL, neboť interakce s ním se omezuje na pouhé přidávání a editaci textu k prvkům modelu a diagramům.

Přiloženy jsou odkazy na několik příspěvků převážně z oficiálního fóra Sparx Systems, které podporují závěrečné vyhodnocení:

[12] Označení, které EA používá pro pluginy.

- http://www.sparxsystems.com/cgi-bin/yabb/YaBB.cgi?num=1340995368,
- http://www.sparxsystems.com/cgi-bin/yabb/YaBB.cgi?num=1347305472,
- http://www.sparxsystems.com/cgi-bin/yabb/YaBB.cgi?num=1352637505,
- http://www.sparxsystems.com/cgi-bin/yabb/YaBB.cgi?num=1357290561,
- http://www.empowertec.de/blog/2007/01/15/ocl-support-in-enterprise-architect/.

Na základě těchto zjištění bylo rozhodnuto dále opustit myšlenku využití EA, která byla formulována v cílech zadání diplomové práce.

4 Rešerše nekomerčních nástrojů a platforem podporujících OCL

Vzhledem k nevyhovujícímu stavu podpory OCL v Enterprise Architect, který bránil ve splnění cílů diplomové práce, byla provedena rešerše nekomerčních nástrojů pro práci s jazykem OCL. Z dostupných řešení byla k podrobnějšímu testování vybrána trojice řešení, které se zdály být nejvíce perspektivní. Vypracování této kapitoly odhalilo současně největší slabinu OCL, kterou je velmi nízká až žádná podpora tohoto jazyka ze strany většiny komplexnějších modelovacích CASE nástrojů. Vyběr tedy probíhal převážně z univerzitních projektů, které mnohdy nedisponovaly dostatečnou dokumentací, tutoriály, uživatelskou základnou a v neposlední řadě byly delší dobu neaktualizované.

4.1 Object Constraint Language Environment

Object Constraint Language Environment (OCLE) je nástroj rumunské univerzity Babes-Bolyai, Cluj-Napoca pro vyhodnocování OCL výrazů nad UML modelem [32]. Poslední verze OCLE 2.0.4 vyšla 1.6. 2005. Jedná se tedy o poměrně letitý software, jehož vývoj pravděpodobně již skončil. Na webových stránkách nástroje je udávána podpora OCL 2.x a UML do verze 1.5. Při startu poslední verze tohoto nástroje prostřednictvím dávkového souboru je však zobrazena informace o podpoře pouze UML verze 1.3.

Mezi hlavní funkcionality OCLE patří:

- Editor diagramu tříd,
- OCL parser a editor se zvýrazňováním syntaxe,
- vyhodnocování OCL na úrovni M2 umožňující úplnou nebo částečnou kontrolu modelu (uživatel zvolí, které elementy modelu budou validovány),
- vyhodnocování OCL na úrovni M1 (instance modelu vytvářena prostřednictvím diagramu objektů),
- import objektů z XML,
- podpora XMI ve verzích 1.0 a 1.1,
- generování kódu z UML modelu na základě definovaných OCL výrazů v jazyce Java,
- nástroje pro debugging,
- validace XML dokumentů oproti definovaným DTD.

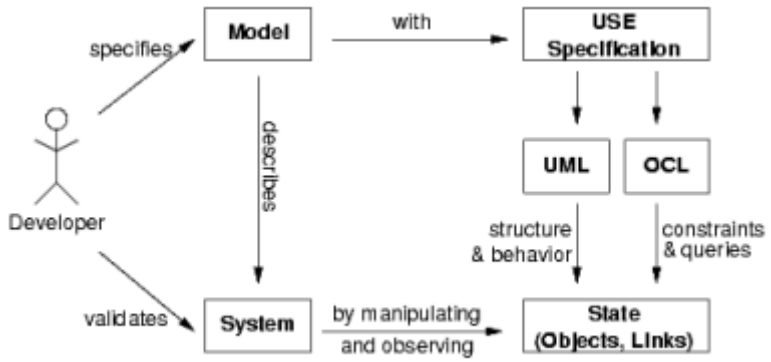
K dispozici je kromě manuálu také sbírka ukázkových modelů a OCL dokumentů. OCLE je poměrně propracovaný nástroj, jehož kvalitu však sráží neaktuálnost a absence implementace posledních specifikací konsorcia OMG.

4.2 USE - UML Based Specification Environment

USE je open source nástroj vyvíjený německou Universität Bremen šiřitelný pod podmínkami GNU General Public License [33]. Aktuální verze 3.0.6 pochází z března 2013. Implementovány jsou poslední specifikace UML a OCL. Nástroj využívá

proprietární formát pro specifikaci modelu a jeho omezení, což je zároveň i jeho největší slabinou. USE není kompatibilní s výstupy většiny modelovacích CASE nástrojů a navíc bez podpory XMI. Vstupní model musí být v textové podobě vyhovující formátu USE.

Ilustrační schéma fungování nástroje USE je znázorněno na obrázku 21.



Obrázek 21 – Ilustrační schéma práce s USE [33]

Mezi hlavní funkcionality USE patří:

- možnost tvorby UML modelu přes proprietární formát USE v textovém režimu,
- OCL parser a editor,
- vyhodnocování OCL na úrovni M2, jež je součástí definic jednotlivých verzí metamodelu UML,
- vyhodnocování OCL na úrovni M1 prostřednictvím diagramu objektů,
- rozšiřující imperativní jazyk SOIL umožňující měnit stav instance modelu (jazyk je bohužel zatím bez uživatelské dokumentace),
- konzole pro vyhodnocování OCL výrazů s podporou debuggingu,
- podpora příkazů USE v konzoli (umožňuje např. konzolové vyváření objektů z nahraného modelu, vytváření spojení mezi nimi apod.).

Stejně jako v případě OCLE je i součástí tohoto nástroje sbírka ukázkových příkladů. USE je používán převážně pro výuku a testovací účely. Podobně jako u OCLE se nejedná o plnohodnotný modelovací CASE nástroj.

4.3 Eclipse Modeling Framework Project

Eclipse Modeling Framework (EMF) je modelovací framework určený pro platformu Eclipse (viz Slovníček pojmů) [34]. Rozsah jeho záměru je široký, od modelování, generování kódu z modelu, definici metamodelů až po možnost využití ve vývoji pluginů pro Eclipse IDE. EMF využívá přednostně Ecore. Podpora metamodelu UML je přítomna také, nikoli však v takovém rozsahu jako pro Ecore. Implementace UML je v EMF realizována v podobě UML2-SDK extenderu [35].

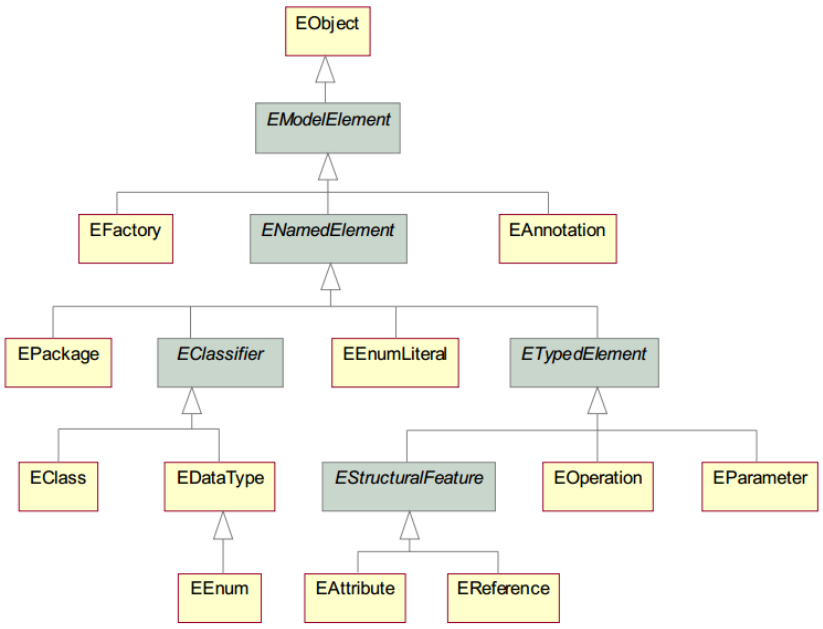
Ecore

Ecore je meta-metamodel (M3), který je ve větší míře podobný specifikaci EMOF, se zaměřením na tvorbu modelovacích jazyků. Oproti CMOF je menší a méně robustní, zato velmi flexibilní. Díky jeho flexibilitě jej lze použít i jako metamodel (M2) pro definici modelů [36], což je ukázáno v kapitole 5. Různé zdroje na něj často poukazují pouze jako na metamodel a toto označení bude uplatňováno i pro zbytek textu diplomové práce, neboť je v ní takto používán. Důležitá je z hlediska OCL možnost instantizace[13] Ecore modelu v EMF, která umožňuje vyhodnocení výrazů definovaných na úrovni M1. V porovnání s UML obsahuje pouze ty prvky, které jsou důležité z hlediska vytváření diagramu tříd. Vzhledem k tomu, že se lze na Ecore dívat jako na podmnožinu elementů užívaných v diagramu tříd UML, lze zajistit strojovou konverzi modelu z Ecore do UML, nikoli však obráceně.

Základními stavebními prvky metamodelu Ecore jsou:

- **EClass** – představuje třídu,
- **EAttribute** – atribut, který je definován svým názvem a typem,
- **EOperation** – operace třídy,
- **EParameter** – parametr operace,
- **EReference** – představuje referenci cílící na klasifikátor,
- **EDataType** – datový typ,
- **EEnum** – výčtový typ.

Hierarchie elementů metamodelu Ecore je znázorněna na obrázku 22.



Obrázek 22 – Hierarchie elementů Ecore [37]

[13] Vytvoření dynamické instance modelu, nad kterou lze vyhodnocovat OCL výrazy.

Ecore Tools

Ecore Tools je komponenta nabízející prostředí pro komplexní správu a udržování Ecore modelů. Její hlavní složkou je grafický editor pro tvorbu diagramů tříd. Obsahuje také možnost validace modelu podle struktury metamodelu Ecore (využívá EMF Validation Framework).

Model Development Tools (MDT)

MDT je soubor nástrojů pro EMF, které implementují standardizované metamodely a poskytují prostředky pro tvorbu z nich vycházejících objektově orientovaných modelů. Tento soubor kromě realizace podpory OCL obsahuje např. již zmiňovaný UML2-SDK extender.

MDT OCL

Jedná se o implementaci OCL pro modely založené na EMF. Kromě podpory OCL pro Ecore a UML zahrnuje zejména [38]:

- API pro parsování a vyhodnocování OCL výrazů nad EMF modelem.
- Možnost vložení zdrojového dokumentu s OCL výrazy do aplikací třetích stran v rámci platformy Eclipse.
- Generátor Java kódu pro Ecore modely s embedded OCL výrazy (výrazy se nenacházejí v odděleném souboru, ale jsou součástí zdrojové reprezentace Ecore modelu).

Papyrus

Papyrus je grafický editor pro tvorbu UML a SysML[14] diagramů v EMF vyhovující specifikaci verze 2.2. Jeho předchůdcem pro podporu UML modelování v EMF byla sada nástrojů UML2 Tools společnosti Borland, jejíž vývoj ustal v roce 2009. [39]

Model vytvořený v tomto nástroji lze validovat oproti OCL omezením definovaným na úrovni M2. Součástí je podpora grafické indikace prvků modelu, které omezení porušují.

Dresden OCL

Informace zde uvedené byly čerpány z oficiálního manuálu Dresden OCL [40].

Dresden OCL je sada nástrojů pro práci s OCL v prostředí EMF vydávanou pod podmínkami GNU Lesser General Public License. Ve vývoji je již od roku 1999 skupinou Technické univerzity v Drážďanech. Současná verze 3.2.0 obecně vyhovuje specifikacím OCL 2.3 podle OMG s několika výjimkami, které jsou v manuálu řádně zdokumentovány.

Hlavními komponenty Dresden OCL jsou:

[14] SysML je metamodel určený pro systémové inženýrství, který využívá specifické části UML [47].

- OCL parser, editor a interpret jazyka,
- nástroj pro generování Java/AspectJ[15] kódu z modelu při napojení na zvolená OCL omezení,
- generátor SQL pro daný model a OCL omezení.

Dresden OCL umožňuje pracovat jak s EMF Ecore tak s UML modely. Podporuje import dynamické instance modelu v podobě Ecore XMI[16] či *ProviderClass* v Javě a práci s ní. *ProviderClass* je v tomto případě jednoduchá třída, která přes metodu *getModelObjects* vrací seznam všech objektů společně tvořících dynamickou instanci modelu. Nevýhodou je nutnost generování Java kódu z modelu a programové vytvoření jednotlivých objektů pro případ instantizace UML modelu.

Dresden OCL obsahuje také komponentu Dresden OCL Metrics tool, která poskytuje jednoduché statistiky pro zvolená OCL omezení, jako např. počet a jednotlivé typy výrazů použitých v daném omezení.

Generování čistého Java kódu z OCL zatím není možné. Přítomna je pouze nezdokumentovaná metoda *generateJavaCode* v jádru Dresden OCL, která vrací často nekorektní podobu zdrojového kódu v jazyku Java.

4.4 Souhrnné srovnání OCLE, USE a Eclipse EMF

Uvedené nástroje byly podrobně otestovány. Byl vyhodnocen jejich přínos a možnosti pro práci s jazykem OCL. Závěrečné shrnutí je obsaženo v tabulce 3.

Tabulka 3 – Srovnání OCLE, USE a Eclipse EMF + MDT. Zdroj: autor

Funkcionalita/Nástroj	OCLE 2.0.3	USE 3.0.6	Eclipse EMF + MDT
UML 2.4.1	NE (pouze 1.3 a diagram tříd)	ANO (diagram tříd, rozšíření pro stavové diagramy)	ANO (slabší podpora, implementačně postavené na Ecore)
Další metamodely	NE	NE	ANO (Ecore, SysML, …)
OCL 2.3.x	NE (pouze 2.0)	ANO	ANO
OCL parser a editor	ANO	ANO	ANO
Vyhodnocování OCL výrazů úrovně M2	ANO	ANO	ANO
Vyhodnocování OCL výrazů úrovně M1	ANO	ANO	ANO (pouze u Ecore modelů lze vytvořit instanci modelu, DresdenOCL vyžaduje zdrojový kód v Javě)
Podpora OCL pro	NE	NE	NE

[15] Jazyk aspektově orientovaného programování (AOP). Jedná se o programovací paradigma, které je ve své podstatě komplementem objektově orientovaného programování a jehož cílem je nahradit v kódu často se opakující činnosti. [44]

[16] Při testování v prostředí Eclipse Juno odmítal Dresden OCL dynamickou instanci nahrát a zobrazoval výjimky na úrovni zdrojového kódu.

diagramy aktivit			
Podpora OCL pro stavové diagramy	NE	ANO (rozšiřující jazyk SOIL)	NE
OCL konzole a dotazování	NE	ANO (M0)	ANO (M1, M0)
Rozšíření OCL operací o možnost měnit stav systému.	NE	ANO (rozšiřující jazyk SOIL, bohužel bez zpracované dokumentace)	ANO (Epsilon Object Language, která je však OCL pouze inspirována, nejedná se o OCL)
Generování Java kódu z OCL	Omezené	NE	Omezené (Genmodel + OCLinEcore)
Generování SQL kódu z OCL	NE	NE	Omezené (Dresden OCL)
Podpora XMI	ANO (pouze do 1.1)	NE	ANO
Podpora MDA	NE	NE	ANO (Acceleo, Obeo, Epsilon project)
Rozšiřitelnost	NE	Nutnost adaptace proprietárního formátu USE	Framework a prostředí pro tvorbu pluginů
Podpora výrobce/dostupné aktualizace	Od roku 2005 neaktualizovaný	ANO	ANO

Na základě uvedeného srovnání byla pro splnění cílů diplomové práce zvolena platforma Eclipse s modelovacím frameworkem EMF a souborem modelovacích nástrojů MDT. Sekundárním kritériem výběru byla možnost realizace rozšíření pro posuzovanou platformu či nástroj.

5 EMF Ecore model polikliniky

Cílem této kapitoly je představit ukázkový model soukromé polikliniky typu PIM v EMF Ecore a provést následující úkony:

a) Definovat OCL integritní omezení na úrovni metamodelu Ecore (M2).
b) Definovat OCL integritní omezení na úrovni modelu (M1).
c) Ověřit OCL integritní omezení nad dynamickou instancí modelu.
d) Formulovat a provést OCL dotazy nad dynamickou instancí modelu.
e) Formulovat a provést OCL dotazy nad modelem.

Protože se jedná charakterem o výukový model, je u popisu vybraných konstrukcí stručně zopakována základní funkcionalita některých OCL operací. OCL je uloženo v externích souborech podle úrovní MOF, na které operuje. Tyto soubory je nutné nejprve přidružit k modelu či instanci modelu a poté lze provést validaci. Pro vyhodnocení OCL omezení je využíváno interpretativního přístupu. Model je součástí datového média diplomové práce (včetně OCL omezení a dotazů v této kapitole neuvedených).

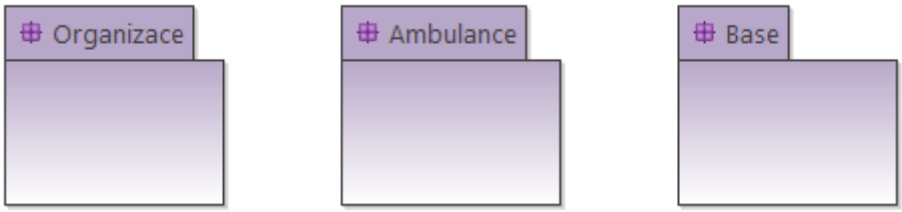
5.1 Fakta o poliklinice

Soukromá poliklinika poskytuje pacientům své služby v rámci jednotlivých ambulančních místností, které si pronajímají různé firmy. O poliklinice jsou známa následující fakta:

- Firmy mají smlouvy s pojišťovnami, které jim proplácejí lékařské výkony na jejich pojištěncích.
- Firmy zaměstnávají lékaře a zdravotní sestry, které jim v ambulanci asistují.
- Základní mzda lékaře se odvíjí podle počtu jeho atestací.
- Každému zaměstnanci jsou navíc k jeho základní mzdě vypláceny finanční prémie.
- Pojišťovna vydává kartičku pojišťovny, kterou se pacient na vyšetření prokazuje.
- Místnosti ambulace jsou značeny číslem unikátním v rámci poschodí, na kterém se místnost nachází, nikoli však unikátním v rámci celé budovy.
- Každá ambulance má vedoucího lékaře, který za její provoz zodpovídá.
- Místnost ambulance může být vybavena různým lékařským zařízením, které z důvodu možných omezujících podmínek (požadavky na místo, teplotu, elektroinstalaci) instaluje, spravuje a udržuje společnost provozující polikliniku.
- Jednotlivá lékařská zařízení musejí být pravidelně revidována. Na zařízení nezpůsobilém k provozu nesmí být proveden lékařský výkon.
- Pacientovi může být na základě vyšetření stanovena diagnóza, podle které může ambulantní lékař zjistit případné kontraindikace.
- Je veden přehled o aktuálně užívaných lécích pacienta a o lécích, na které trpí alergií.
- U vybraných entit jsou evidovány různé typy adres (trvalá, přechodná, fakturační, …).

5.2 Balíčky

Pomocí nástroje Ecore Tools byly vytvořeny balíčky **Organizace, Ambulance** a **Base** definovány svým názvem, jmenným prostorem a namespace URI. Kooperaci jednotlivých balíčků EMF Ecore bohužel nelze v diagramu balíčků graficky zachytit ani jinak explicitně vyjádřit.

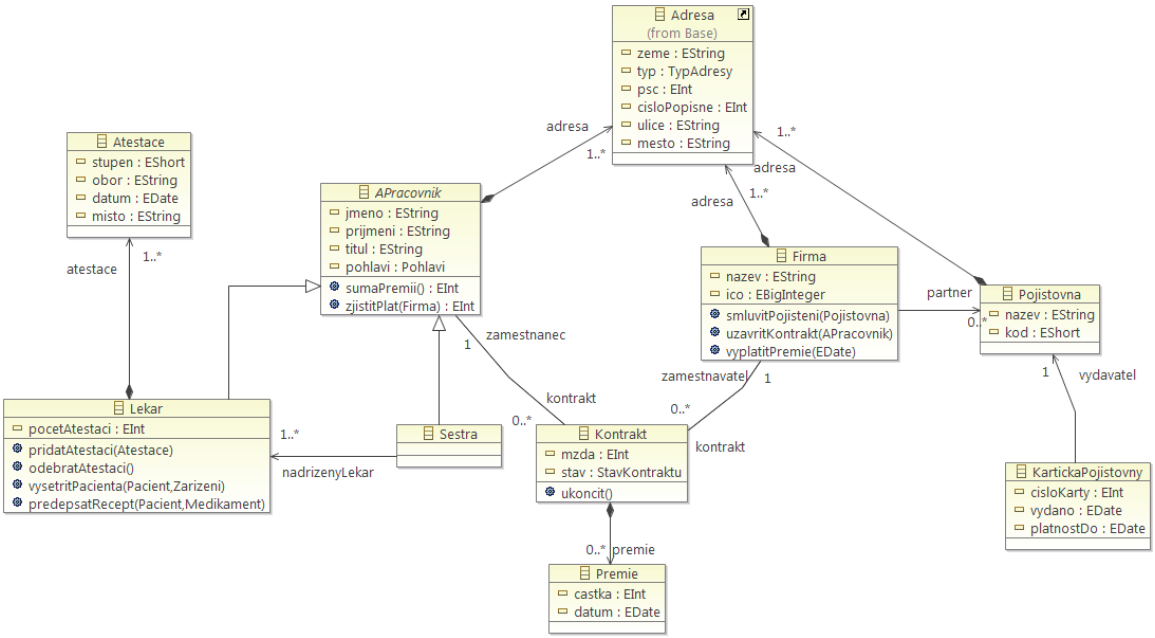


Obrázek 23 – Balíčky Ecore modelu polikliniky. Zdroj: autor

5.3 Diagramy tříd

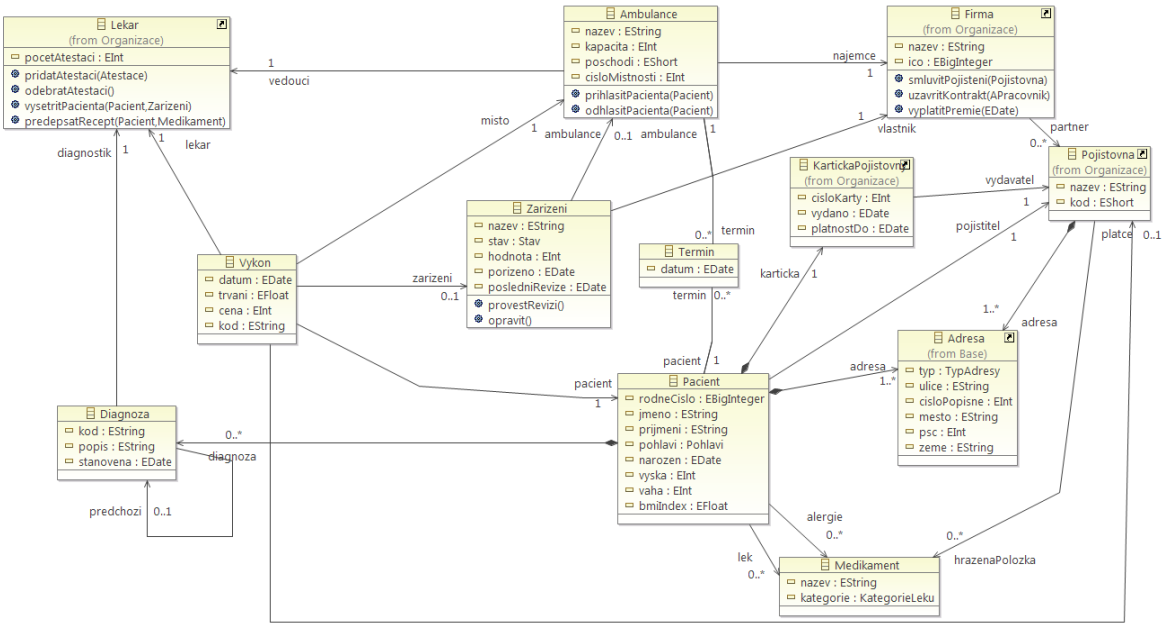
Ecore zná pouze diagram tříd, který se od svého UML protějšku v některých aspektech liší (absence stereotypů, nemožnost stanovit viditelnost atributů a operací, chudší výběr ornamentů, nezná asociační třídy, rozhraní není samostatným klasifikátorem, …). Ecore se zaměřuje na strukturu modelovaného systému a nedisponuje takovým záběrem jako UML.

Diagram tříd balíčku **Organizace** zachycuje organizační strukturu subjektů působících na či majících vztah k poliklinice (viz obrázek 24).



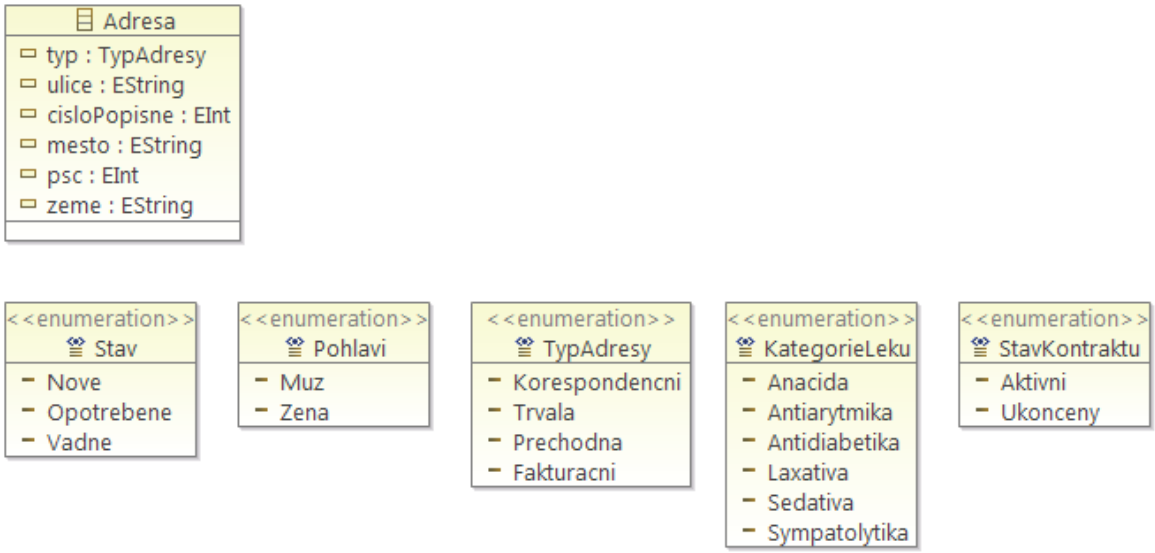
Obrázek 24 – Ecore diagram tříd balíčku Organizace. Zdroj: autor

Diagram tříd balíčku **Ambulance** zachycuje vnitřní strukturu polikliniky (viz obrázek 25).



Obrázek 25 – Ecore diagram tříd balíčku Ambulance. Zdroj: autor

Poslední balíček **Base** obsahuje definice výčtových typů a třídu *Adresa* (viz obrázek 26).



Obrázek 26 – Ecore diagram tříd balíčku Base. Zdroj: autor

5.4 OCL na úrovni M2

Při psání OCL výrazů pro modelovací elementy Ecore je nutné vycházet ze stromové hierarchie těchto prvků podle obrázku 22. Na této úrovni lze definovat převážně ty druhy omezení, které nějakým způsobem mohou korigovat práci týmu softwarových analytiků, případně customizovat daný metamodel pro práci na konkrétním projektu. Poměrně důležité je volit vhodné a výstižné názvy *invariantů*, aby v případě neúspěšné validace bylo možné dohledat, kterými nedostatky se model vůči OCL omezením provinil.

Vynucení jmenné konvence

Při tvorbě modelu je vhodné dodržovat jednotný styl pojmenování *CamelCase*[17]: variantu *upper* pro názvy tříd, rozhraní a výčtových typů a variantu *lower* pro názvy atributů, operací a parametrů operací.

S využitím klíčového slova *def* lze definovat pomocnou funkci, která přes regulární výrazy ověří, zda vstupní řetězec vyhovuje zvolené formě *CamelCase* (viz blok 27). Definice musí proběhnut v kontextu prvku, který zastřešuje (je rodičem) ty prvky, pro něž bude funkce používána. V tomto případě se jedná o kontext prvku *ENamedElement*, nicméně v úvahu by připadal i jeho rodič *EModelElement*. Vhodné je volit minimální možnou oblast platnosti, aby kontrole nebyly podrobovány prvky, pro které *invariant* není určen.

Dostupnost podpory pro regulární výrazy záleží na OCL enginu, který dané vývojové prostředí implementuje. V případě EMF jsou k dispozici standardní ASCII regulární výrazy. Použitá operace *matches* pro porovnávání řetězců je specifická pro implementaci OCL v Eclipse EMF.

```
context ENamedElement
/*
 * Pomocná funkce pro parsování řetězců kontrolující upper nebo lower CamelCase
* styl
*/
def: parseCamelCase(elementName : String, upper : Boolean) : Boolean =
if upper then
elementName.matches('[A-Z][A-Za-z0-9]*')
else
elementName.matches('[a-z][A-Za-z0-9]*')
endif
```

Blok 27 – OCL. Definice funkce parseCamelCase

Pro definování *invariantu* hlídajících, zda je název v souladu s *upper CamelCase* (viz blok 28), je využito kontextu prvku *EClassifier*, který je v Ecore hierarchii rodičem tříd, rozhraní (reprezentováno stejným prvkem jako třída, tedy *EClass*) a výčtových typů. Stejně tak pod něj spadá i *EDataType*, který umožňuje v modelu definovat vlastní datové typy vázané na datové typy jazyku Java, se kterým je implementace metamodelu Ecore v EMF úzce spjata.

```
context EClassifier
-- Parser pro korektní názvy klasifikátorů (vynutí upper CamelCase)
inv upperCamelCaseParserCheck: parseCamelCase(name, true)
```

Blok 28 – OCL. Invariant vynucující upper CamelCase

Dále je vynucen lower CamelCase v kontextu prvku *ETypedElement* (viz blok 29).

```
/*
```

[17] Jedná se o způsob psaní víceslovných názvů, kdy jednotlivá slova nejsou oddělena mezerami. Každé ze slov začíná velkým písmenem. Počáteční písmeno je vždy velké pro variantu *upper* a vždy malé pro variantu *lower*.

```
 * Parser pro korektní názvy atributů, operací, parametrů operací a konců
asociací (vynutí lower CamelCase)
*/
context ETypedElement
inv lowerCamelCaseParserCheck: parseCamelCase(name, false)
```

Blok 29 – OCL. Invariant vynucující lower CamelCase

Názvům abstraktních tříd a rozhraní by dle uznávané jmenné konvence měly předcházet prefixy „A“, resp „I“, což lze zajistit OCL operacemi s řetězci (viz blok 30). Operace *at* vrátí znak na požadovaném indexu. Definována je pomocná funkce, která v závislosti na atributech kontrolované instance typu *EClass* zkontroluje prefix. Abstraktní třída v Ecore musí mít nastavený atribut *abstract* na *true*, rozhraní pak navíc ještě atribut *interface* na *true*. Ecore jako rozhraní definuje *EClass*, která má oba atributy nastavené na *true*, nicméně tuto skutečnost již automaticky hlídá vestavěný validátor.

```
context EClass
/*
 * Pomocná funkce pro ověření požadovaného prefixu
*/
def: namePrefixCheck(eClass : EClass) : Boolean =
-- Názvům abstraktních tříd musí předcházet znak 'A'
((eClass.abstract and not eClass.interface) implies name.at(1) = 'A')
and
-- Názvům rozhraní musí předcházet znak 'I'
(eClass.interface implies name.at(1) = 'I')

inv eClassPrefixCheck:
namePrefixCheck(self)
```

Blok 30 – OCL. Ověření prefixů u klasifikátorů

Unikátnost členů a názvů

V rámci klasifikátorů je třeba dodržet unikátnost názvů atributů, signatur operací a názvů konců asociací (jinak by docházelo k nejednoznačnosti při navigaci v modelu). Stejně tak ve výčtových typech je požadováno zachovat jedinečné názvy literálů. Je tedy nutné vrátit se ke kontextu *EClassifier* a definovat pro něj příslušný *invariant*. V tomto případě je navíc rozlišeno, jakého typu kontrolovaná instance je. K tomuto účelu v jazyce OCL slouží operace *oclIsType*. Po určení typu je nutné provést přetypování instance přes operaci *oclAsType*, čímž je zajištěn přístup ke kolekcím *eAttributes* a *eOperations* u třídního klasifikátoru a *eLiterals* u výčtového typu**.** Nad těmito je provedena iterativní operace *forAll*. Jsou porovnány nestejné dvojice prvků a pomocí logické implikace vytvořen booleovský výraz, který detekuje duplicity (*forAll* vrací *true*, pokud všechny prvky kolekce vyhovují zadanému výrazu, jinak *false*). Vše uvedené zachycuje blok 31.

```
context EClassifier
/*
 * Pomocná funkce pro ověření unikátnosti členů klasifikátorů
*/
def: uniqueMembers(eClassifier : EClassifier) : Boolean =
eClassifier.oclIsTypeOf(EClass) implies(
-- Kontrola unikátnosti názvů atributů
```

```
      let eClass : EClass = eClassifier.oclAsType(EClass) in
      eClass.eAttributes->isUnique(name)
      and
-- Kontrola unikátnosti signatur operací
      eClass.eOperations->forAll(a, b | a <> b implies let match : Boolean =
a.name.matches(b.name)
  in not match or (match implies a.eParameters->collect(eType)->asOrderedSet()
<> b.eParameters->collect(eType)->asOrderedSet()))
      and
-- Kontrola unikátnosti názvů konců asociací
      eClass.eReferences->isUnique(name))
and
eClassifier.oclIsTypeOf(EEnum) implies(
-- Kontrola unikátnosti literálů u vyčtového typu
      let eEnum : EEnum = eClassifier.oclAsType(EEnum) in
      -- MDT OCL bug: funguje v konzoli, selže během validace
      eEnum.eLiterals->forAll(a, b | a <> b implies (a.name <> b.name and
a.literal <> b.literal)))

inv uniqueMembersCheck:
uniqueMembers(self)
```

Blok 31 – OCL. Vynucení unikátnosti atributů, signatur operací a konců asociací

V tomto případě bylo detekováno chybné chování MDT OCL během validace při zpracování části `eEnum.eLiterals->forAll(a, b | a <> b implies (a.name <> b.name and a.literal <> b.literal))`, který kontroluje jedinečnost literálů. Validace pokaždé proběhne úspěšně v testovací OCL konzoli, avšak ve většině případů selže (vrátí *invalid*) při validaci z OCL dokumentu. Tuto část je tedy prozatím vhodné zakomentovat, než bude zjednána náprava ze strany vývojářů.

Omezení dědičnosti

OCL umožňuje daný metamodel přizpůsobit potřebám konkrétního programovacího jazyka. Pokud daný vyšší programovací jazyk např. nepřipouští vícenásobnou dědičnost, lze tento fakt do metamodelu promítnout. V tomto případě však takové omezení není třeba. Vhodné se z hlediska dědičnosti jeví zamezení nesmyslné konstrukce, kdy třída či rozhraní dědí samy od sebe (viz blok 32).

```
context EClass
-- Rozhraní/třída nemohou dědit samy od sebe
inv noCyclicInheritanceCheck:
not eSuperTypes->exists(superType : EClass | superType = self)
```

Blok 32 – OCL. Omezení dědičnosti

Kolekce *eSuperTypes* obsahuje typy klasifikátorů, které kontrolovaná instance dědí. Iterační operace *exists* s uvedeným výrazem zjistí, zda je členem této kolekce i příslušná instance typu *EClass*. V případě, že ano, je *invariant* porušen.

Omezení pro řešenou doménu

Doposud byla uvedena obecná omezení, která by měla každá instance metamodelu Ecore dodržovat. Omezení v bloku 33 přízpůsobuje metamodel Ecore konkrétnímu zadání soukromé polikliniky.

```
context EClass
-- Třídy Lekar a Sestra musejí dědit od abstraktní třídy APracovnik
inv docAndNurseInheritanceCheck:
let classNames : Set = Set{'Lekar', 'Sestra'} in
classNames->exists(n | n = name) implies
      eSuperTypes->exists(st | st.name = 'APracovnik')
```

Blok 33 – OCL. Vynucení dědičnosti pro třídy Lekar a Sestra

Uvedené pravidlo zajistí, že vyskytne-li se v modelu třída *Lekar* nebo *Sestra*, musí tato dědit od abstraktní třídy *APracovnik*. Výrazem *let* je definována dvouprvková množina s názvy tříd, pro které kontrola proběhne.

Pravidla pro adresy

Vybrané třídy (*APracovnik, Pojistovna, Firma, …*) obsahují referenci na třídu *Adresa*. Je vynuceno, aby pokud daná třída referenci na adresu obsahuje, činila tak vždy v násobnosti 1..N a navíc aby se tato reference vyskytovala nejvýše jednou (viz blok 34).

```
context EClass
/*
 * Třídy, ktere obsahují adresu, tak musejí činit v násobnosti 1..N
 * Rovněž je povolena pouze jedna vazba na třídu Adresa
*/
inv addressCheck:
let refs = eReferences->select(r | r.eReferenceType.name = 'Adresa' )
->asSequence() in
(refs->size() = 1 implies
      (refs->first().lowerBound = 1 and refs->first().upperBound = -1))
and
(refs->size() > 1 implies false)
```

Blok 34 – OCL. Pravidla pro adresy

Nad kolekcí *eReferences* je provedena operace *select*, která vrátí všechny prvky kolekce vyhovující kritériu v zadaném výrazu. V tomto případě je získána kolekce referencí třídy s názvem *Adresa* typu sekvence. Dále dojde k ověření, zda vrácená kolekce obsahuje skutečně jen jednu referenci na adresu, přes operaci *size*. Operace *first*, kterou disponují pouze kolekce se seřazenými prvky (*Sequence*, *OrderedSet*), pak vrátí vedoucí (zde jediný) prvek kolekce. U tohoto prvku je ověřena dolní a horní hodnota násobnosti, kterou v metamodelu Ecore uchovávají atributy *lowerBound*, resp. *upperBound*. Násobnost N je reprezentována číselnou hodnotou -1.

Omezení pro atestace

Následující jednoduchý *invariant* zajistí, že referenci na třídu *Atestace* bude moci mít pouze třída *Lekar* (viz blok 35).

```
context EClass
```

```
-- Atestaci může mít pouze třída Lekar
inv medicalAttestationCheck:
(eReferences->exists(r | r.eReferenceType.name = 'Atestace')) =
(name = 'Lekar')
```

Blok 35 – OCL. Omezení pro atestace

Shrnutí

V této podkapitole byly vysvětleny OCL výrazy vztahující se na metamodel Ecore. Obecná omezení informační hodnotu metamodelu obohacují, avšak definování OCL pravidel závislých na konkrétní instanci metamodelu se může jevit jako poněkud netradiční. Tato možnost tu však je, a jak bylo zmíněno dříve, lze ji využít např. pro korigování týmu softwarových analytiků při práci na větším projektu, kde je část textových specifikací přenesena do formy OCL výrazů.

Jednotlivé *invarianty* se píší společně pod svůj kontext, zde však bylo pro názornost užito odděleného zápisu s duplicitním kontextem.

5.5 OCL na úrovni M1

M1 OCL omezení jsou ověřována na konkrétní instanci modelu, kterou je nejprve nutné vytvořit. V EMF Ecore lze pro tento účel využít **Sample Reflective Ecore Model** editor a v něm dynamicky vytvářet jednotlivé instance klasifikátorů a spojení mezi nimi. Dynamické instance jsou uspořádány stromově, proto je pro realizaci komplexní instance modelu nutné vytvořit zapouzdřovací třídu, která se stane kořenem tohoto stromu (viz kapitola 5.4.3). Nevýhodou tohoto přístupu je skutečnost, že lze nejvýše pracovat pouze s jedním stromem. Důsledkem je nemožnost kombinace dynamických instancí různých modelů.

Struktura uvedených OCL výrazů pro jednotlivé balíčky je výjádřena v bloku 36.

```
package <NázevBalíčku>
výraz1
výraz2
…
výrazN
endpackage
```

Blok 36 – OCL. Struktura balíčků

5.5.1 Balíček Organizace

Mzdová a prémiová omezení a výrazy

Minimální mzda lékaře je odvozena podle stupně jeho odbornosti (viz blok 37). Je stanoveno, že základní mzda pro lékaře s dvěmi a více atestacemi nesmí klesnout pod určenou částku. Navíc se počítá s mzdovým přírůstkem pro každou další atestaci.

```
/*
 * Minimální mzda lékaře je od dvou a více atestací odstupňována po 5 000 Kč
*počínaje mzdou 30 000 Kč
```

```
*/
context Lekar
inv minimumWageCheck:
let min : Integer = 30000, increment : Integer = 5000 in
atestace->size() >= 2 implies
      kontrakt->select(stav = Base::StavKontraktu::Aktivni)->forAll(contract |
contract.mzda >= min + ((atestace->size() - 2) * increment))
```

Blok 37 – OCL. Minimální mzda lékaře

Pro kontext abstraktní třídy *APracovnik* je definován *invariant* hlídající maximální sumu prémií aktivního kontraktu pro jednotlivé profese (viz blok 38).

```
/*
 * Celková suma prémií nemůže pro aktivní kontrakt překročit částku 200 000 Kč
* za lékaře a 100 000 Kč za sestru
*/
context APracovnik
inv maxBonusCheck:
let docBonus : Integer = 200000, nurseBonus : Integer = 100000, contracts :
Bag(Kontrakt) = kontrakt->select(stav = Base::StavKontraktu::Aktivni) in
if self.oclType() = Lekar then
contracts->forAll(
c | c.premie->collect(castka)->asBag()->sum() <= docBonus)
else
contracts->forAll(
c | c.premie->collect(castka)->asBag()->sum() <= nurseBonus)
endif
```

Blok 38 – OCL. Omezení prémií

Nejprve je nutné pro vybraného pracovníka vybrat všechny jeho aktivní kontrakty za pomocí operace *select* (atribut *stav* je porovnáván s literálem *Aktivni* výčtového typu *StavKontraktu*). Hodnoty částek jednotlivých prémií přístupné přes konec asociace *premie* daného kontraktu jsou sesbírány pomocí operace *collect* do kolekce typu *Bag* a následně sečteny přes *sum*. Důležité je rozlišit, jakého typu je instance třídy *APracovnik*, k čemuž je využita operace *oclType*.

Celkovou sumu prémií pracovníka lze zjistit přes metodu *sumaPremii* využívající stejného principu z předchozího omezení (viz blok 39).

```
/*
 * Vrací sumu všech prémií pro lékařského pracovníka
*/
context APracovnik::sumaPremii() : Integer
body:
let costs : Bag(Integer) = kontrakt.premie->collect(castka)->asBag() in
costs->sum()
```

Blok 39 – OCL. Dotazovací operace sumaPremii třídy APracovnik

Lékařské atestace

Atributu *pocetAtestaci* třídy *Lekar* lze pomocí klíčového slova *derive* přiřadit funkční složku tak, aby výraz `<instanceTřídyLekar>.pocetAtestaci` vracel aktuální počet atestací zvoleného lékaře (viz blok 40).

```
/*
 * Atribut pocetAtestaci je odvozen od výrazu atestace->size(), který vrátí
*aktuální počet atestací lékaře
*/
context Lekar::pocetAtestaci : Integer
derive: atestace->size()
```

Blok 40 – OCL. Odvozený atribut pocetAtestaci třídy Lekar

Je definována operace, která lékaři přidělí atestaci. Jedná se o operaci, která mění stav systému, proto zde má smysl definovat pouze *post-condition*. Ta popisuje, že počet atestací se po provedení operace zvedne o jedna a nová atestace bude součástí kolekce atestací lékaře (viz blok 41). Voláním `@pre` je zajištěn přístup k atributům ve stavu před provedením operace.

```
/*
 * Přidělí lékaři atestaci
*/
context Lekar::pridatAtestaci(novaAtestace : Atestace) : OclVoid
post: pocetAtestaci->size() = atestace@pre->size() + 1
      and
      atestace->includes(novaAtestace)
```

Blok 41 – OCL. Operace pridatAtestaci třídy Lekar

Další operace naopak odebere poslední udělenou atestaci (viz blok 42).

```
/*
 * Odebere lékaři poslední udělenou atestaci
*/
context Lekar::odebratAtestaci() : OclVoid
pre: atestace->size() > 0
post: pocetAtestaci->size() = atestace@pre->size() - 1
      and
      atestace->excludes(atestace@pre->last())
```

Blok 42 – OCL. Operace odebratAtestaci třídy Lekar

Vyšetření pacienta

Jsou definovány dvě *pre-conditions* pro operaci *vysetritPacienta* (viz blok 43).

```
/*
 * Vyšetření pacienta
*/
context Lekar::vysetritPacienta(pacient : Ambulance::Pacient, zarizeni :
Ambulance::Zarizeni) : OclVoid
-- Pacient musí být evidován u ambulance, kterou lékař vede
pre: not Ambulance::Ambulance.allInstances()->select(vedouci = self)
       ->intersection(pacient.ambulance)->isEmpty()
```

```
-- Je-li k vyšetření potřeba lékařské zařízení, musí být toto způsobilé k
-- provozu
pre: not zarizeni.oclIsUndefined() implies zarizeni.stav <> Base::Stav::Vadne
```

Blok 43 – OCL. Operace vysetritPacienta třídy Lekar

Předpis léků

Medikament lze pacientovi předepsat pouze v případě, že jej hradí jeho pojišťovna a zároveň na něj není alergický (viz blok 44).

```
/*
 * Medikament lze pacientovi předepsat pouze v případě, že jej hradí jeho
*  pojišťovna a zároveň
 * na něj není alergický
*/
context Lekar::predepsatRecept(pacient : Ambulance::Pacient, medikament :
Ambulance::Medikament) : OclVoid
pre:
pacient.pojistitel.hrazenaPolozka->includes(medikament)
and
pacient.alergie->excludes(medikament)
post: pacient.lek->size() = pacient.lek@pre->size() + 1
      and
      pacient.lek->includes(medikament)
```

Blok 44 – OCL. Operace predepsatRecept třídy Lekar

Pokud jsou splněny oba výrazy v *pre-condition* a není-li porušen žádný *invariant*, je zaručena skutečnost popsaná v *post-condition* (v tomto případě přídání medikamentu mezi pacientem užívané léky).

Omezení pro unikátní hodnoty

Instance vybraných tříd musejí mít jisté hodnoty atributů unikátní (viz blok 45). Operace *allInstances* vrací kolekci aktuálně existujících objektů v instanci modelu, na kterých se provede test unikátnosti požadovaného atributu.

```
/*
 * Kód pojišťovny musí být unikátní
*/
context Pojistovna
inv uniqueInsuranceCompanyCodeCheck: self.oclType().allInstances()->isUnique(p
| p.kod)
/*
 * Číslo kartičky pojišťovny musí být unikátní
*/
context KartickaPojistovny
inv uniqueCardNumberCheck: self.oclType().allInstances()->isUnique(p |
p.cisloKarty)
```

Blok 45 – OCL. Omezení pro unikátní hodnoty

Datum platnosti a vydání kartičky pojišťovny

Invariant v bloku 46 zajistí, že kartička pojišťovny bude mít korektní hodnoty data platnosti a vydání.

```
context KartickaPojistovny
inv correctDateCheck: platnostDo > vydano
```

Blok 46 – OCL. Omezení pro data kartičky pojišťovny

Firemní omezení pojistky

Aby mohla mít firma uzavřenou smlouvu o proplácení lékařských výkonů s pojišťovnou, musí zaměstnávat alespoň 3 lékaře a libovolný počet zdravotních sester (viz blok 47).

```
/*
 * Aby mohla mít firma smlouvu s pojišťovnou, musí zaměstnávat alespoň 3 lékaře
*/
context Firma
inv threeDoctorsPerCompanyMinimumCheck:
partner->size() >= 1 implies
      kontrakt->select(employee | employee.zamestnanec.oclIsTypeOf(Lekar) and
stav = Base::StavKontraktu::Aktivni)->size() >= 3
```

Blok 47 – OCL. Omezení pojištění

Nejprve je zjištěno, zda firma vůbec nějakou smlouvu uzavřenou má, a následně kombinací operací *select* a *size* počet zaměstnanců typu *Lekar*, který je porovnán s požadovaným množstvím.

Výplata prémií

Firmy odměňují své lékaře na bázi odvedených výkonů vždy od určitého data. Blok 48 popisuje nasazení OCL u operace *vyplatitPremie* třídy *Firma*. Je definována *pre-condition*, která říká, že pro udělení odměn musí mít firma alespoň jeden aktivní lékařský kontrakt. Je zde využito pomocné funkce *aktivniKontrakty*, která vrátí množinu aktivních lékařských kontraktů pro konkrétní firmu. *Post-condition* popisuje stav po úspěšně vykonané operaci, kdy je odměněným lékařům firmy připsána prémie. Tento příklad využívá operaci *selectByType*[18], která z kolekce extrahuje prvky požadovaného typu.

```
/*
 * Firma vyplácí svým lékařům prémie na bázi počtu odevedených výkonů od
* určitého data
 * Je vyplácena jednotná částka 500 Kč za každý provedený výkon
*/
context Firma
def: aktivniKontrakty(firma : Firma) : Set(Kontrakt) =
kontrakt->reject(stav = Base::StavKontraktu::Ukonceny)->asSet()

context Firma::vyplatitPremie(od : ecore::EDate) : OclVoid
pre: not aktivniKontrakty(self).zamestnanec->selectByType(Lekar)->isEmpty()
```

[18] Tato operace je specifická pro implementaci OCL v Eclipse EMF a není definována konsorciem OMG. Jedná se v podstatě o zkrácený zápis `self->select(OclIsTypeOf(<typ>))`, kde *self* je kolekcí. Další takovou operací je *selectByKind*, jež je, dle očekávání, zkráceným zápisem `self->select(OclIsKindOf(<nadtyp>))`.

```
post: aktivniKontrakty(self).zamestnanec->selectByType(Lekar)->forAll(
l : Lekar |
let bonus : Integer = Ambulance::Vykon.allInstances()->select(datum >= od
and lekar = l and misto.najemce = self)->size() * 500 in
let contract : Kontrakt = l.kontrakt->select(zamestnavatel = self and stav =
Base::StavKontraktu::Aktivni) in
contract.premie->last().castka = bonus and contract.premie->size() =
contract.premie@pre->size() + 1)
```

Blok 48 – OCL. Výplata prémií

5.5.2 Balíček Ambulance

Pacient

Uvedena je sada *invariantů* pro pacienta (viz blok 49). Kromě standardního požadavku na unikátní rodné číslo je důležité zamezit tomu, aby pacient užíval ty léky, které u něj vyvolávají alergické reakce. Stejně tak je v zájmu objektivity vhodné u řetězení diagnóz vynucovat nestejného diagnostika v případě, že řetězení dosáhne úrovně vyšší jak dvě.

```
context Pacient
-- Rodné číslo pacienta musí být unikátní
inv uniqueBirthNumberCheck:
self.oclType().allInstances()->isUnique(p | p.rodneCislo)
-- Pacient nemůže užívat lék, na který má zároveň alergii
inv medicamentAlergyCheck:
alergie->intersection(lek)->size() = 0
-- Pacient může mít pouze tu kartičku pojišťovny, u které je evidován
inv insuranceCardConflictCheck:
karticka->one(vydavatel = pojistitel)
-- Má-li pacient více jak 2 na sebe navazující diagnózy, nesmějí být určeny
-- stejným diagnostikem
inv diagnosisDocConflictCheck:
let diagSq : Bag(Diagnoza) = diagnoza->select(d : Diagnoza |
d->closure(predchozi)->size() >= 2)->asBag() in
not diagSq->isEmpty()
implies
diagSq->forAll(d : Diagnoza | let dCheck : Bag(Diagnoza) =
d->closure(predchozi)->union(d->asBag()) in
      not dCheck.diagnostik->forAll(doc1, doc2 | doc1 = doc2))
```

Blok 49 – OCL. Sada invariantů pro kontext Pacient

V lékařském prostředí je často nutné znát hodnotu indexu BMI pacienta. Atributu *bmiIndex* je tedy přiřazena funkční složka pomocí výrazu *derive* (viz blok 50).

```
/*
 * BMI index je odvozen od biometrických hodnot pacienta
*/
context Pacient::bmiIndex : Real
derive: vaha / ((vyska * vyska) / 10000)
```

Blok 50 – OCL. Odvození BMI indexu

Omezení pro čísla místností ambulancí

Jak bylo uvedeno v popisu polikliniky, čísla ambulantních místností jsou unikátní pouze v rámci jednotlivých poschodí (viz blok 51).

```
/*
 * Čísla místností musejí být v rámci jednotlivých poschodí unikátní
*/
context Ambulance
inv uniqueRoomIdCheck:
self.oclType().allInstances()->forAll(a, b |
a <> b implies (
 a.poschodi = b.poschodi implies a.cisloMistnosti <> b.cisloMistnosti
))
```

Blok 51 – OCL. Omezení pro čísla místností ambulancí

Omezení pro užívání léků

Antiarytmika nesmějí být užívána společně se sedativy nebo sympatolitiky (viz blok 52).

```
-- Antiarytmika nesmějí být užívána společně se sedativy nebo sympatolitiky
context Pacient
inv medicamentConflictCheck:
self.lek->select(kategorie = Base::KategorieLeku::Antiarytmika)->size() > 0
implies(
let conflictingMedicaments : Set(Base::KategorieLeku) =
Set{Base::KategorieLeku::Sedativa, Base::KategorieLeku::Sympatolytika} in
self.lek->collect(lek.kategorie)->intersection(conflictingMedicaments)->size()
= 0)
```

Blok 52 – OCL. Omezení pro užívání léků

V tomto případě je využit výčtový typ *KategorieLeku*, na jehož literály je nutné se odvolat prostřednictvím čtyřtečkové notace. Nejprve je zjištěno, zda pacient užívá léky kategorie antiarytmika. V případě, že ano, je definována množina s konfliktními kategoriemi léků a přes velikost průniku s kategoriemi léků pacientem aktuálně užívanými je zjištěna přítomnost potenciálního konfliktu.

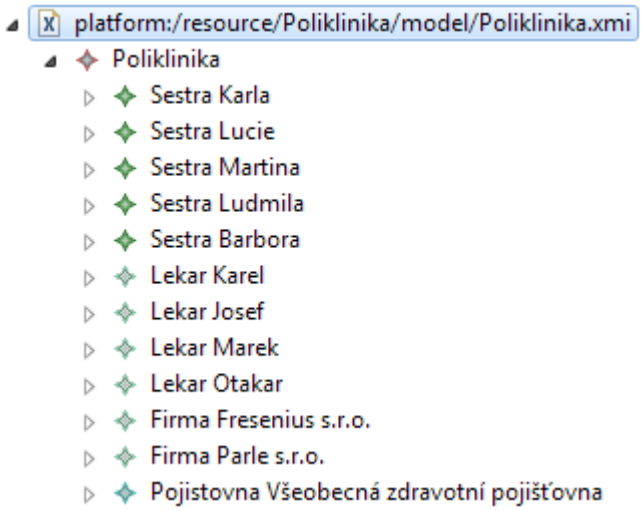
5.6 Dynamická instance modelu

Byla vytvořena vrstva M0 modelu soukromé polikliniky za pomocí Sample Reflective Ecore Model editoru (viz obrázek 27), na které byla otestována správnost definovaných M1 integritních omezení. Dynamická instance modelu je soubor formátu XMI, který je přímo nalinkován k danému Ecore modelu.

Nepříjemným omezením vytváření dynamické instance v uživatelském prostředí Eclipse EMF je fakt, že nelze tuto vytvořit pro celý model či alespoň balíček. Vždy je nutné zvolit daný element z M1 vrstvy a pouze pro něj a elementy, které referencuje, bude instance vytvořena. Tento nedostatek lze obejít prostřednictvím pomocného kontejneru následujícím způsobem:

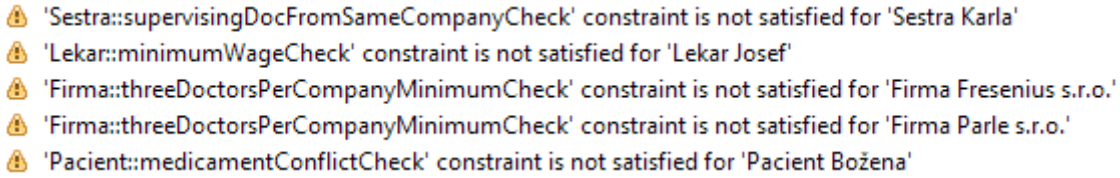
1. Vytvořit libovolnou neabstraktní *EClass* jako pomocný kontejner v požadovaném Ecore diagramu.

2. Navázát ke kontejneru ty elementy, které chceme podrobit zkoumání.
3. U všech takto nově vytvořených referencí nastavit atribut *isContainment* na *true* a násobnost na 0..*.
4. Vytvořit dynamickou instanci z pomocného kontejneru příkazem *Create Dynamic Instance* z kontextového menu prvku v Sample Reflective Ecore Model editoru.



Obrázek 27 – Výřez z dynamické instance modelu polikliniky. Zdroj: autor

Tato konkrétní instance záměrně porušuje některé *invarianty*. Validace neplatné *invarianty* úspěšně odhalila, což dokazuje obrázek 28. Vytvořenou dynamickou instanci modelu polikliniky lze různými způsoby alterovat, aby např. neodpovídala nějakému konkrétnímu požadovanému *invariantu* či naopak byla plně validní.



Obrázek 28 – Výsledek validace dynamické instance modelu polikliniky. Zdroj: autor

Při testování OCL výrazů úrovně M1 nad dynamickou instancí modelu bylo zároveň zjištěno, že interpret jazyka v Eclipse EMF nedokáže vyhodnotit pomocné funkce definované výrazy *def* jako i odvozené atributy *derive*. Stejně tak nedokáže vyhodnotit uživatelem definované dotazovací operace. Byl učiněn pokus přidat OCL výrazy přímo do zdrojového kódu modelu Ecore prostřednictvím editoru OCLinEcore, který však při uložení modelu ztrácel formátování, mazal komentáře a závorky určující priority výrazů. Taktéž měl problémy s diakritikou i přes zvolené UTF-8 kódování.

5.6.1 Dotazování se nad dynamickou instancí modelu

K dotazování se nad dynamickou instancí modelu je možné využít **MDT OCL konzoli**. Následující dotazy je nutné spouštět v kontextu kořenu stromu dynamické instance polikliniky. Pro získání všech instancí konkrétního typu není pak nutné používat operaci

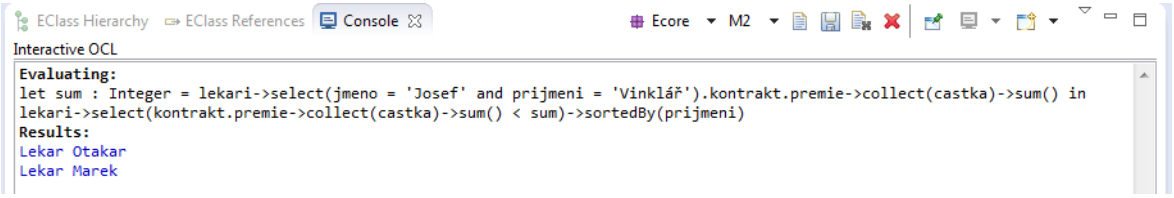
allInstances, ale stačí název reference (např. všechny lékaře polikliniky lze místo volání `Lekar.allInstances()` získat jednoduše přes `lekari`).

OCL dotaz 1

Najdi lékaře, kteří mají celkový objem získaných finančních prémií menší než jejich kolega Josef Vinklář, a seřad je podle jejich příjmení (viz blok 53).

```
let sum : Integer = lekari->select(jmeno = 'Josef' and prijmeni =
'Vinklář').kontrakt.premie->collect(castka)->sum() in
lekari->select(kontrakt.premie->collect(castka)->sum() < sum)
->sortedBy(prijmeni)
```

Blok 53 – OCL. Dotaz 1



Obrázek 29 – Znázornění výsledku OCL dotazu v interaktivní konzoli. Zdroj: autor

OCL dotaz 2

Najdi všechny zaměstnance firmy Parle s.r.o., kteří jsou profesí lékař a slouží pod nimi alespoň jedna zdravotní sestra (viz blok 54).

```
sestry->reject(nadrizenyLekar->isEmpty()).nadrizenyLekar->asSet()->intersection
(
firmy->select(nazev = 'Parle s.r.o.').kontrakt.zamestnanec
->selectByType(Lekar).oclAsSet()
)
```

Blok 54 – OCL. Dotaz 2

Nejprve jsou pomocí operace *reject* odstraněni ti lékaři, pro které je porušena podmínka počtu sester. Dále je proveden množinový průnik s druhou částí dotazu. V té jsou nejprve zjištěni zaměstnanci firmy Parle s.r.o. pomocí operace *select*, avšak výsledkem je v tomto případě množina objektů typu *APracovnik* (mohou to být tedy lékaři nebo sestry). Operací *selectByType* jsou z této množiny vybrány objekty požadovaného typu. Vrácen je však abstraktní typ *Collection* (obecná kolekce), který s *intersection* není možné použít, proto je ještě nutné provést přetypování tohoto na kolekci typu *Set* pomocí operace *oclAsSet*.

OCL dotaz 3

Seřad pracovníky společností Fresenius s.r.o. a konkurenční Parle s.r.o. podle výše jejich mzdy sestupně a zjisti, kteří zaměstnanci Fresenius s.r.o. mají mzdu vyšší nebo stejnou než konkurenční pracovník na identické pozici v druhém seznamu (viz blok 55).

```
let rev1 : Sequence(Kontrakt) = firmy->select(nazev =
'Fresenius s.r.o.').kontrakt->select(stav = Base::StavKontraktu::Aktivni)
->sortedBy(mzda)->asSequence() in
let size1: Integer = rev1->size() in
```

```
let company1 : Sequence(Kontrakt) = Sequence{0..size1-1}->collect(i :
Integer | rev1->at(size1 - i)) in
let rev2 : Sequence(Kontrakt) = firmy->select
(nazev = 'Parle s.r.o.').kontrakt->select(stav =
Base::StavKontraktu::Aktivni)->sortedBy(mzda)->asSequence() in
let size2: Integer = rev2->size() in
let company2 : Sequence(Kontrakt) = Sequence{0..size2-1}->collect(i :
Integer | rev2->at(size2 - i)) in
let count : Integer = size1.min(size2) in
let final : Sequence(Integer) = Sequence{1..count}->select(i: Integer
| let con1 : Kontrakt = company1->at(i), con2 : Kontrakt = company2->at(i)
in con1.mzda
>= con2.mzda) in
let employees : Sequence(APracovnik) = final->collect(i : Integer |
company1->at(i).zamestnanec) in
employees
```

Blok 55 – OCL. Dotaz 3

Zde je již situace o něco složitější, neboť v MDT OCL není implementována operace *reverse*. Za pomocí řetězení výrazu *let* je úloha rozdělena na jednodušší části. Nejprve je třeba seřadit aktivní kontrakty obou společností podle mzdy a otočit jejich pořadí procházením sekvence indexů, kdy jsou odečítáním od zjištěné velikosti daných kolekcích získávány objekty v opačném pořadí a ty následně ukládány do nových kolekcí prostřednictvím operace *collect*.

Dále je zjištěno minimum počtu pracovníků obou firem, aby nedošlo k porovnávání s neexistujícími prvky v případě, že obě firmy nedisponují stejným počtem zaměstnanců.

Na závěr se do kolekce *final* uloží indexy těch objektů, které vyhovují uvedené mzdové podmínce. Na základě těchto indexů je opět nutné získat objekty, což lze provést použitím již známé operace *collect*.

V dotazu je u definic kolekcí použit výraz udávající rozpětí (značí se dvěma tečkami). Např. zápis `Sequence{1..4}` vrátí kolekci `Sequence{1, 2, 3, 4}`.

OCL dotaz 4

Pro všechny pacienty ambulancí MUDr. Karla Páva vypiš chronologické pořadí vývoje jejich diagnóz a příjmení diagnostika (viz blok 57). Výstupem pro každého pacienta bude struktura popsaná blokem 56.

```
{ {<kódDiagnózy>,
<příjmeníDiagnostika>,
<předchozíDiagnózaKód1>, <předchozíDiagnózaKód2> … <předchozíDiagnózaKódN>},
… }
```

Blok 56 – Pseudokód. Struktura výstupu dotazu 4

Protože je kontejner obsahující dynamickou instanci modelu umístěn v balíčku **Organizace**, je třeba u některých klasifikátorů explicitně stanovit kontext pomocí operátoru :: (čtyřtečky) v případě, že jsou definovány v jiných balíčcích.

```
let patients : Set(Ambulance::Pacient) = pacienti->select(ambulance =
ambulance->select(vedouci.jmeno = 'Karel' and vedouci.prijmeni = 'Páv')) in
patients->collect(item | let patient : Ambulance::Pacient = item, diagPairs :
Set(Tuple(diagCode: String , docSurname: String, previousDiags: Set(String))) =
item.diagnoza->collect(d | Tuple{ diagCode = d.kod, docSurname =
d.diagnostik.prijmeni, previousDiags =
d->closure(predchozi)->collect(kod)->asSet()->sortedBy(d.stanovena)})->asSet()
in
Tuple{pacient = patient, diagSet = diagPairs})
```

Blok 57 – OCL. Dotaz 4

Prvním krokem je získání kolekce pacientů registrovaných v ambulancích určeného lékaře. Požadovaná výstupní struktura je definována za pomocí objektu typu *Tuple* představujícího uspořádanou n-tici záznamů. V dotazu je použita rekurzivní operace *closure* kumulativně ukládající do cílové kolekce ty objekty, které referencuje výraz jejího argumentu. V tomto případě jsou diagnózy uspořádány na způsob jednosměrného lineárního spojového seznamu přes atribut *predchozi*.

5.7 Dotazování se nad modelem

Stejně jako v předchozím případě se i zde využívá MDT OCL konzole. Pro účely dotazování nad modelem je vhodnější tento otevřít v **Sample Ecore Model** editoru místo předchozího Sample Reflective Ecore Model editoru. Je vytvořena čtveřice dotazů, které demonstrují schopnost jazyka OCL z modelu extrahovat požadované informace. Uveden je dotaz, který pro zvolený balíček vypíše název třídy s nejvíce atributy (včetně zděděných) společně s počtem (viz blok 58).

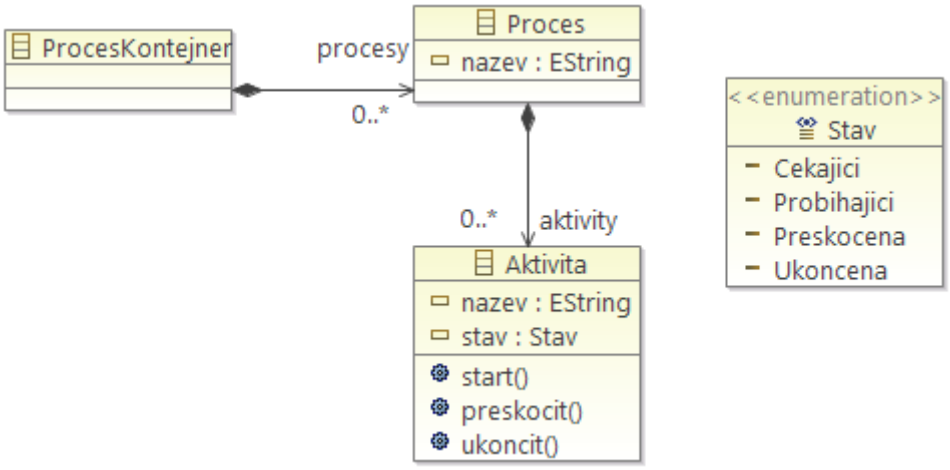
```
let eclasses : Collection(EClass) = eClassifiers->selectByType(EClass) in
let max : Tuple(name : String, attributeCount : Integer) =
eclasses->iterate(currentEClass : EClass; res : Tuple(name : String,
attributeCount : Integer) =
Tuple{name = 'No class found', attributeCount = 0} |
let currAttributeCount : Integer = currentEClass.eAllAttributes->size() in
if (currAttributeCount > res.attributeCount) then
Tuple{name = currentEClass.name, attributeCount = currAttributeCount}
else
res
endif) in
max
```

Blok 58 – OCL. Dotaz nad modelem

5.8 Deklarativní přístup k modelování workflow v poliklinice

Na základě návrhu univerzity v Rostocku [41] je představen deklarativní přístup k modelování workflow na ukázkovém Ecore modelu polikliniky. Přístup spočívá

ve využití diagramu tříd, který znázorňuje pomocnou strukturu pro zachycení procesu a jeho aktivit (viz obrázek 30). Původní návrh je modifikován pro potřeby MDT OCL v Eclipse EMF.



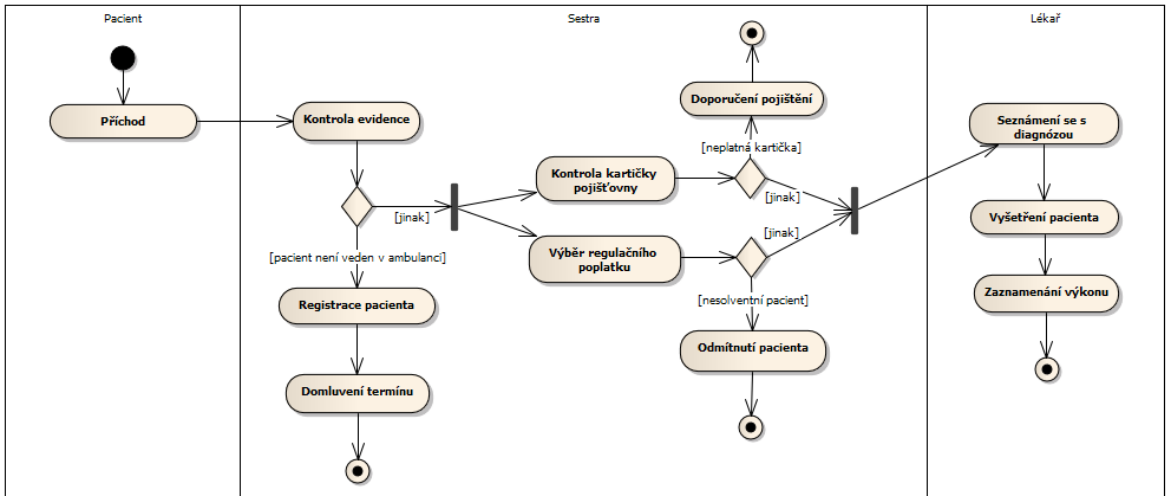
Obrázek 30 – Deklarativní přístup k zachycení workflow, diagram tříd. Zdroj: autor

Proces se skládá z aktivit. Každá aktivita se může nacházet v jednom ze čtyř stavů:

a) čekající,
b) probíhající,
c) přeskočená,
d) ukončená.

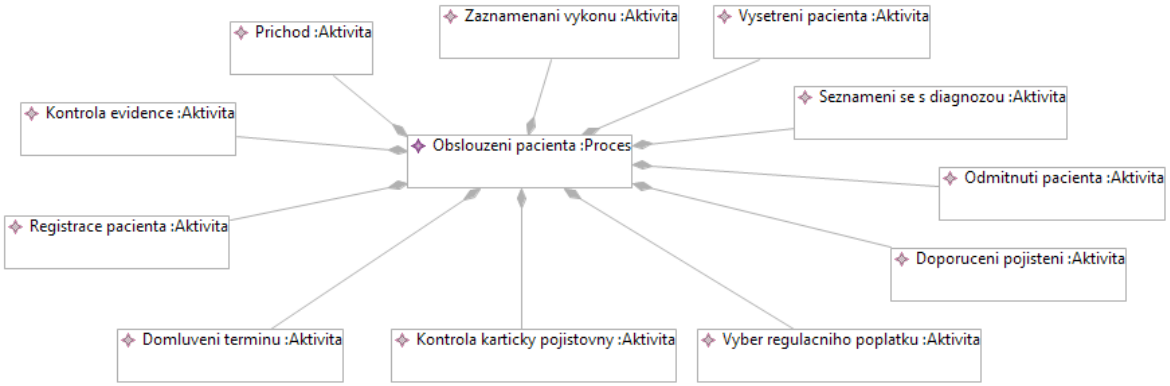
Výchozí stav všech aktivit je čekající. Aktivita má dále definované operace *start, preskocit* a *ukoncit*, které mění její stav. Deklarativní přístup modelování workflow předpokládá, že ve výchozím stavu může být spuštěna libovolná z aktivit. Protože se zde nepracuje s imperativním jazykem, který by toto umožnil, jsou operace pouze popsány prostřednictvím OCL. Stavy jednotlivých aktivit je nutné přepínat v editoru dynamické instance konkrétního procesu. Uvedená struktura pro zachycení workflow sama o sobě nestačí. To je zachyceno formou OCL *invariantů* určujících podmínky, za kterých může být daná aktivita spuštěna.

Proces obsloužení pacienta po příchodu do ambulance lze standardně zachytit diagramem aktivit (viz obrázek 31).



Obrázek 31 – Diagram aktivit obsloužení pacienta v ambulanci. Zdroj: autor

Pro tento případ je vytvořena dynamická instance celého procesu (viz obrázek 32).



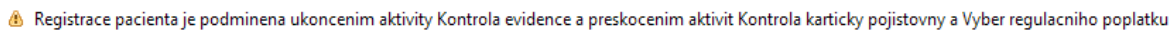
Obrázek 32 – Diagram dynamické instance procesu obsloužení pacienta v ambulanci. Zdroj: autor

Dále jsou doplněny *invarianty* definující příslušnou workflow. Například spuštění aktivity *Registrace pacienta* je podmíněno ukončením aktivity *Kontrola evidence* a přeskočením aktivit *Kontrola kartičky pojišťovny* a *Výběr regulačního poplatku*, které mohou probíhat paralelně. Tuto skutečnost zachycuje *invariant* v bloku 59.

```
inv regPacientaAlt:
let registracePacienta : Aktivita = aktivity->any(nazev = 'Registrace
pacienta'),
kontrolaEvidence : Aktivita = aktivity->any(nazev = 'Kontrola evidence'),
kontrolaKartickyPojistovny : Aktivita = aktivity->any(nazev = 'Kontrola
karticky pojistovny'),
vyberRegulacnihoPoplatku : Aktivita = aktivity->any(nazev = 'Vyber regulacniho
poplatku') in
registracePacienta.stav = Stav::Probihajici implies
(kontrolaEvidence.stav = Stav::Ukoncena
and kontrolaKartickyPojistovny.stav = Stav::Preskocena
and vyberRegulacnihoPoplatku.stav = Stav::Preskocena)
```

Blok 59 – OCL. Ukázka invariantu pro deklarativní modelování workflow

Na konkrétním stavu procesu lze ověřit, zda daná aktivita může být v tu chvíli spuštěna, či nikoli (viz obrázek 33).



Obrázek 33 – Výstup validace dynamické instace pro jeden ze stavů procesu obsloužení pacienta v ambulanci. Zdroj: autor

Tímto způsobem jsou modelovány všechny zbývající aktivity procesu.

Deklarativní přístup modelování workflow je oproti klasickému UML diagramu aktivit méně názorný. Přínosem je zejména možnost testovat proces v jednotlivých stavech s ověřením, zda konkrétní aktivita může být za daných podmínek spuštěna. Další výhodou je přesnější zachycení celého procesu a možnost modelování skutečností, které v diagramu aktivit modelovat nelze. Například znázornění, že dvě aktivity nemohou běžet ve stejný okamžik, v diagramu aktivit nelze docílit. Ideální se jeví kombinace obou přístupů.

5.9 Shrnutí

Na příkladu objektově orientovaného modelu polikliniky v Ecore byla prezentována schopnost jazyka OCL definovat integritní omezení ve formě *invariantů* s cílem zvýšit vypovídající hodnotu modelu. Definovaná omezení byla následně ověřena prostřednictvím dynamické instance modelu. Díky interaktivní OCL konzoli bylo možné ověřit i správnost zápisu *pre-conditions* a *post-conditions*. Dále byly demonstrovány dotazovací schopnosti jazyka OCL na instanci modelu i modelu samotném. Na závěr byl představen deklarativní přístup k modelování workflow prostřednictvím diagramu tříd.

Prostředí OCL v Eclipse EMF bohužel trpí několika nedostatky, které mohou nezkušeného uživatele obtěžovat či zmást (tvorba probíhala ve verzi platformy Juno).

a) OCL editor nebere v úvahu případné změny v modelu provedené od okamžiku otevření souboru s OCL výrazy. Parser pak hlásí falešné chyby. Po provedených změnách je tedy vždy nutné soubor v editoru otevřít znovu.
b) Při vložení většího bloku OCL kódu do editoru se může stát, že parser přestane fungovat korektně. Řešením je opět znovuotevření souboru.
c) Nelze vytvořit soudržnou dynamickou instanci celého modelu, ale vždy pouze instance vybraných klasifikátorů (lze obejít přidáním pomocného kontejneru).
d) Nelze spouštět funkce definované výrazem *def* na úrovni M1. Neplatí pro úroveň M2.
e) Absentuje implementace OCL operace *symmetricDifference*.
f) Pokud je soubor s OCL výrazy přidružen k modelu či dynamické instanci modelu v Sample (Reflective) Ecore Model editoru a parser v něm najde chybu, pohled na model či jeho instanci se zneplatní a je nutné znovuotevření editoru.
g) Vkládání OCL výrazů přímo do kódu modelu Ecore prostřednictvím nástroje OCLinEcore je možné pouze u triviálních výrazů. Tento problém byl již diskutován

na internetových diskuzních fórech EMF, avšak nebylo dosud nalezeno funkční řešení.

h) Při vytváření dynamické instance modelu jako diagramu prostřednictvím pluginu dochází ke zdvojování instancí klasifikátorů. Jedná se patrně pouze o grafický bug, protože při změně atributů jedné z nich se změna promítne u obou instancí.

6 OCL generátor pro podporu výuky tvorby objektově orientovaných modelů

Cílem této kapitoly je popis analýzy, návrhu a implementace generátoru validačního kódu jazyka OCL pro vývojovou a modelovací platformu Eclipse. Předpokládáno je nasazení tohoto nástroje jako výukové a zkušební pomůcky pro kurzy modelování objektově orientovaných softwarových systémů se zaměřením na diagramy tříd. Validační kód bude sloužit jako vstup příslušných validačních nástrojů platformy Eclipse.

Analytická část je provedena v jazyce UML za pomocí modelovacího CASE nástroje Enterprise Architect. Model je součástí datového média diplomové práce.

OCL generátor je realizován formou pluginu do Eclipse. Jsou popsány některé implementační sekce generátoru se zaměřením na jazyk OCL. Instalace generátoru a práce s ním je popsána samostatně v instalační a uživatelské příručce, která je součástí přílohy B.

Motivace k vytvoření OCL generátoru

Hlavním motivačním faktorem je realizovat nástroj, který by napomáhal studentům kurzů objektového modelování interaktivní formou lépe uchopit jeho podstatu a prohloubit jejich znalosti a zkušenosti s tvorbou diagramů tříd jak v analytické, tak návrhové formě.

Dalším faktorem je možnost použití výstupu tohoto nástroje pro strojovou kontrolu domácích úkolů nebo pro vyhodnocení praktické části zkoušky.

Protože je modelování kreativní činností, je třeba této skutečnosti přizpůsobit i zadání úloh. Typovými úlohami může být např. sestavení diagramu na bázi sémantiky názvů jednotlivých klasifikátorů, které student předem obdrží, nebo doplnění rozpracovaného diagramu o požadované informace. Pro velmi obecné či rozsáhlé úlohy je vhodné strojové kontrole podrobit pouze její vybrané segmenty, případně pouze vynucovat obecně platné modelovací konvence.

6.1 Analýza OCL generátoru

V rámci analýzy byly nejprve zachyceny funkční a nefunkční požadavky a dále případy užití. Integrita mezi funkčními požadavky a případy užití byla ověřena prostřednictvím matice pokrytí. Dále byly nalezeny analytické třídy a rozděleny do odpovídajících balíčků. Analytické třídy byly hledány metodou podstatných jmen a sloves z popisu požadavků, textu scénářů případů užití a částečně také na bázi znalosti metamodelů Ecore a UML.

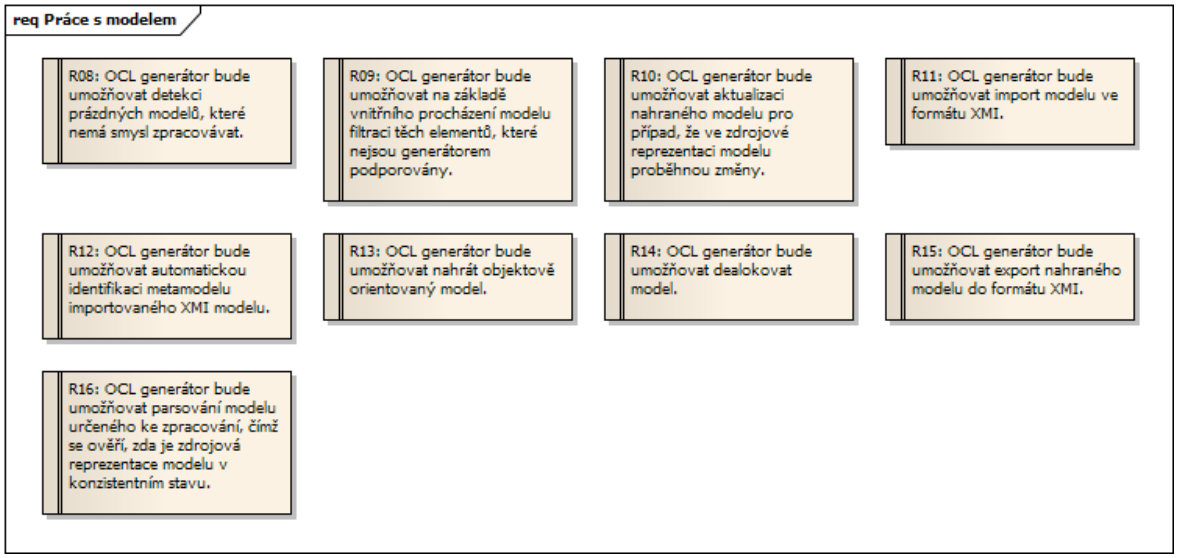
6.1.1 Požadavky

Oba typy požadavků jsou opatřeny svým jedinečným identifikátorem. Funkční požadavky jsou zapsány ve formátu `<id>: <systém>` bude `<funkce>` s výjimkou kategorie Validační kód, kde je místo zápisu „Systém bude umožňovat generování validačního kódu, který ověří …" pro přehlednost užito formy „Validační kód bude umožňovat ověřit …". Je použito `<id>` ve formátu `R[0-9]{2}` pro funkční požadavky a `NR[0-9]{2}` pro nefunkční požadavky. Příklady formulovaných požadavků jsou znázorněny na obrázcích 34, 35 a 36.

Funkční požadavky

Funkční požadavky jsou podle svého charakteru rozděleny do čtyř kategorií:

- **Generování validačního kódu** – požadavky zaměřené na funkcionalitu a konfiguraci generování validačního kódu.
- **Práce s modelem** – požadavky kladené na schopnost aplikace pracovat se vstupním modelem.
- **Validační kód** – fundamentální kategorie popisující požadované funkční vlastnosti validačního kódu.
- **Vizualizace modelu** – požadavky kladené na zobrazení struktury modelu.



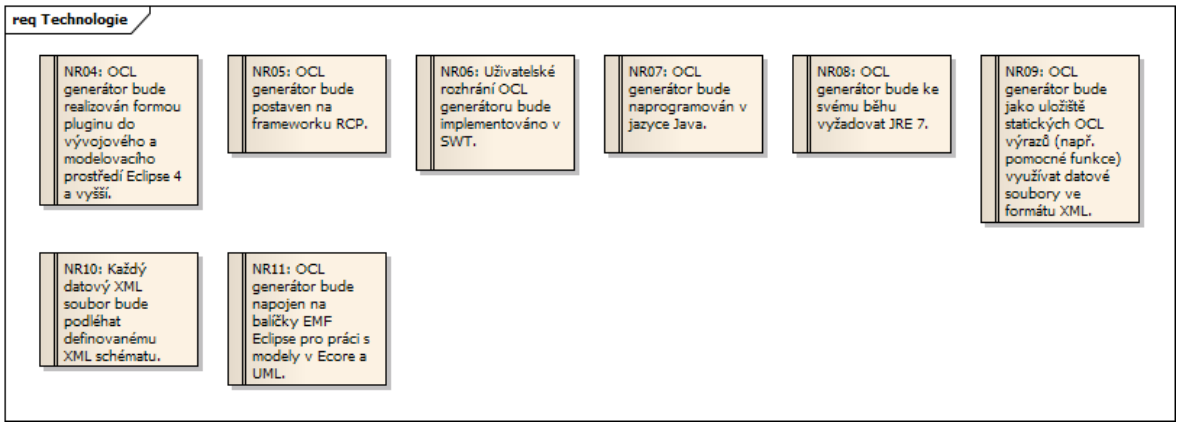
Obrázek 34 – Funkční požadavky: kategorie Práce s modelem. Zdroj: autor

Náhled ostatních kategorií funkčních požadavků je součástí přílohy A.

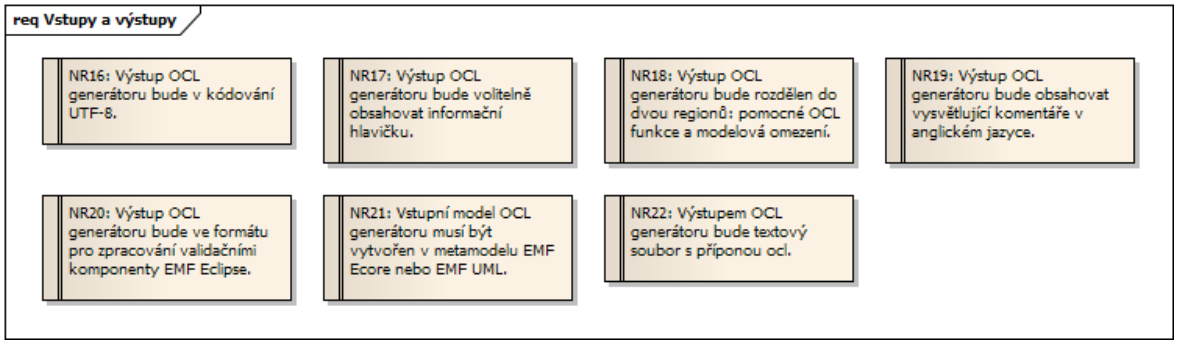
Nefunkční požadavky

Nefunkční požadavky jsou podle svého charakteru rozděleny do čtyř kategorií:

- **Distribuce** – požadavky zaměřené na zpřístupnění aplikace uživatelům včetně dodatečné podpory.
- **Technologie** – soubor implementačních technologických omezení.
- **Uživatelské rozhraní** – požadavky kladené na uživatelské rozhraní.
- **Vstupy a výstupy** – upřesňující požadavky kladené na vstupy a výstupy aplikace.



Obrázek 35 – Nefunkční požadavky: kategorie Technologie. Zdroj: autor



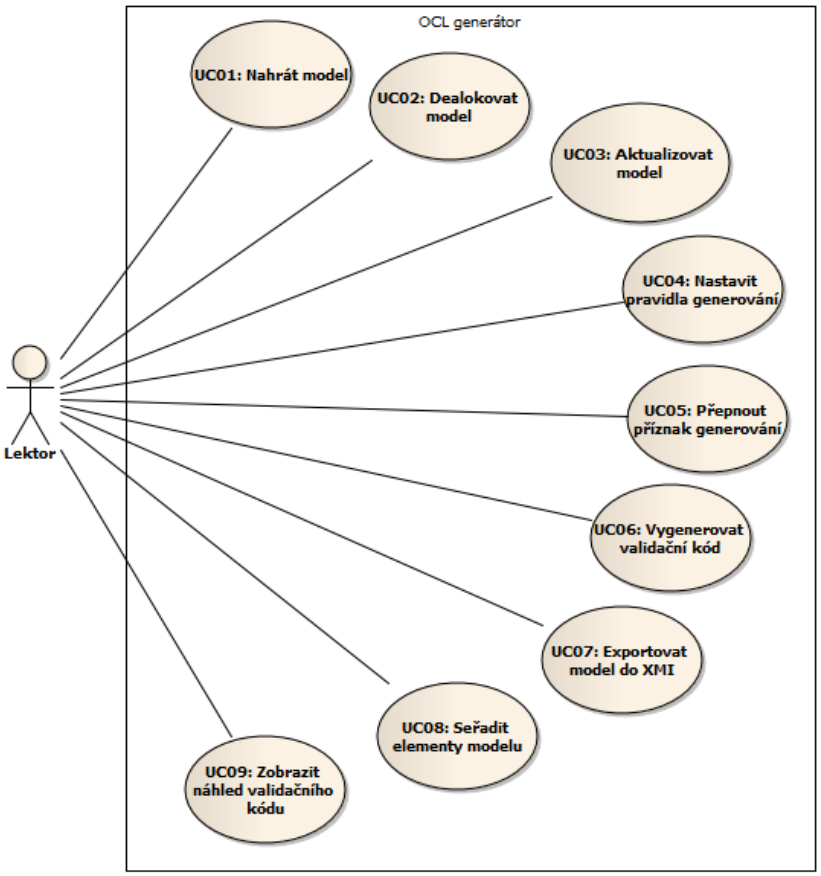
Obrázek 36 – Nefunkční požadavky: kategorie Vstupy a výstupy. Zdroj: autor

Náhled ostatních kategorií nefunkčních požadavků je součástí přílohy A.

6.1.2 Aktéři a případy užití

Jedinou identifikovanou rolí OCL generátoru je **Lektor**, který aplikaci využívá dle diagramu případů užití na obrázku 37. Jedná se o osobu vedoucí kurz objektového modelování.

Názvy případů užití se řídí formátem `UC[0-9]{2}: <názevPřípaduUžití>`. Matice pokrytí funkčních požadavků případy užití je součástí přílohy A.



Obrázek 37 – Diagram případů užití OCL generátoru. Zdroj: autor

Každý případ užití je opatřen hlavním a alternativními scénáři dostupnými ke zhlédnutí v příloze A.

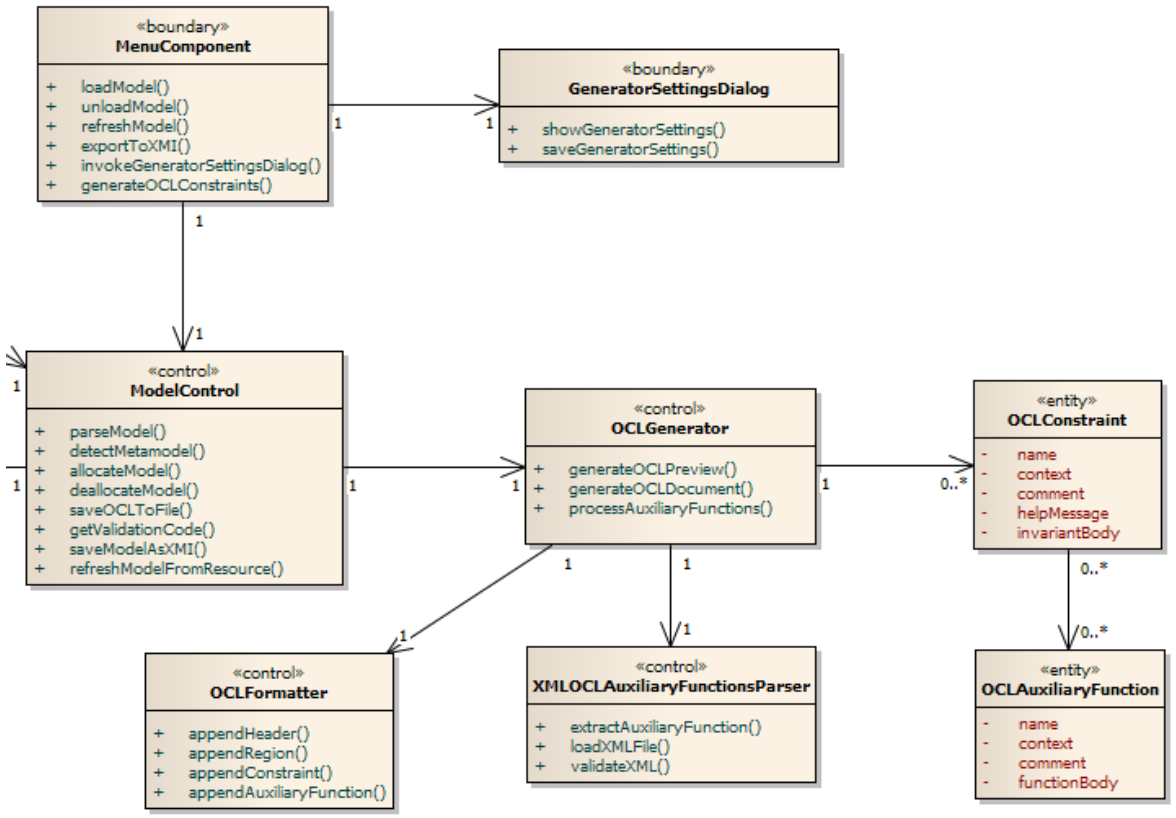
Tabulka 4 – Hlavní scénář případu užití UC01. Zdroj: autor

Případ užití: UC01: Nahrát model
Stručný popis: Model se nahraje ze zdrojového souboru a následně zpracuje.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: Žádné.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem „Load Model...“ z menu. 2 Zobrazí se dialogové okno pro výběr zdrojového souboru modelu. 3 Lektor vybere zdrojový soubor modelu. 4 Zpracuje se zdrojový soubor modelu. 4.1 Provede se parsování zdrojového souboru modelu. 4.2 Ověří se, zda zdrojový soubor obsahuje model vytvořený v jednom z podporovaných metamodelů.

4.3 Ověří se, zda zdrojový soubor obsahuje neprázdný model. 4.4 Zjistí se informace o jednotlivých elementech modelu. 5 Model se nahraje do aplikace. 6 Tabulková reprezentace modelu se zobrazí v zobrazovací komponentě. 7 V menu se zpřístupní nové příkazy pro operace s modelem.
Výstupní podmínky: 1 Model byl nahrán. 2 Tabulková reprezentace modelu byla zobrazena v zobrazovací komponentě. 3 V menu byly zpřístupněny nové příkazy pro operace s modelem.
Alternativní scénáře: Poškozený model Nepodporovaný metamodel Prázdný model

6.1.3 Analytické třídy

Nalezené třídy byly rozděleny dle své logiky do příslušných balíčků a opatřeny stereotypy *boundary*, *control*, *entity*. Názvy analytických tříd a jejich strukturální složky jsou v anglickém jazyce. Diagramy analytických tříd se nacházejí v příloze A. Ukázka výřezu jednoho z diagramů je na obrázku 38.



Obrázek 38 – Výřez z diagramu analytických tříd balíčku generating. Zdroj: autor

6.2 Návrh a implementace OCL generátoru

Protože je validační kód OCL generátoru využíván validačními komponenty v Eclipse EMF a vstupem mu jsou EMF modely, jeví se jako nanejvýš vhodné generátor integrovat

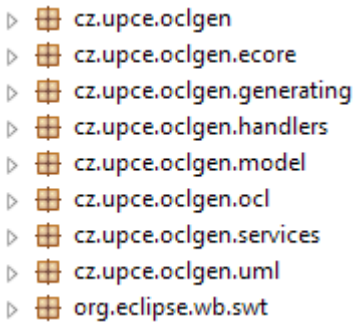
přímo do prostředí Eclipse ve formě pluginu. Rozšiřuje se tak portfolio OCL nástrojů, které platforma nabízí. Zároveň je tím rozhodnuto o implementačním jazyku, jímž je v případě tvorby Eclipse pluginů mandatorně Java.

Plugin využívá frameworku Rich Client Platform (RCP), který usnadňuje integraci pluginu do mateřské platformy Eclipse. Součástí RCP je mimo jiné UI framework JFace a meziplatformní UI knihovna Standard Widget Toolkit (SWT) obsahující layout manažery, tlačítka a další. Kód i komentáře jsou psány v anglickém jazyce, protože se předpokládá uvolnění nástroje jako open source.

Návrhové třídy byly získány metodou reverzního inženýrství ze zdrojového kódu vyvinutého pluginu. Důvodem tohoto přístupu se stala potřeba prvotního experimentování s platformou Eclipse a frameworkem EMF, kdy chyběly zkušenosti vývoje aplikace podobného typu. Ukázalo se přesto, že analytické třídy byly navrženy ve větší míře správně a lze je považovat za dostatečně vypovídající. Tento postup však rozhodně není vhodný, protože upírá možnost generování kostry aplikace ve zvoleném implementačním jazyce.

6.2.1 Struktura balíčků

Návrhové třídy aplikace jsou rozděleny do devíti balíčků podle funkčních oblastí (viz obrázek 39).



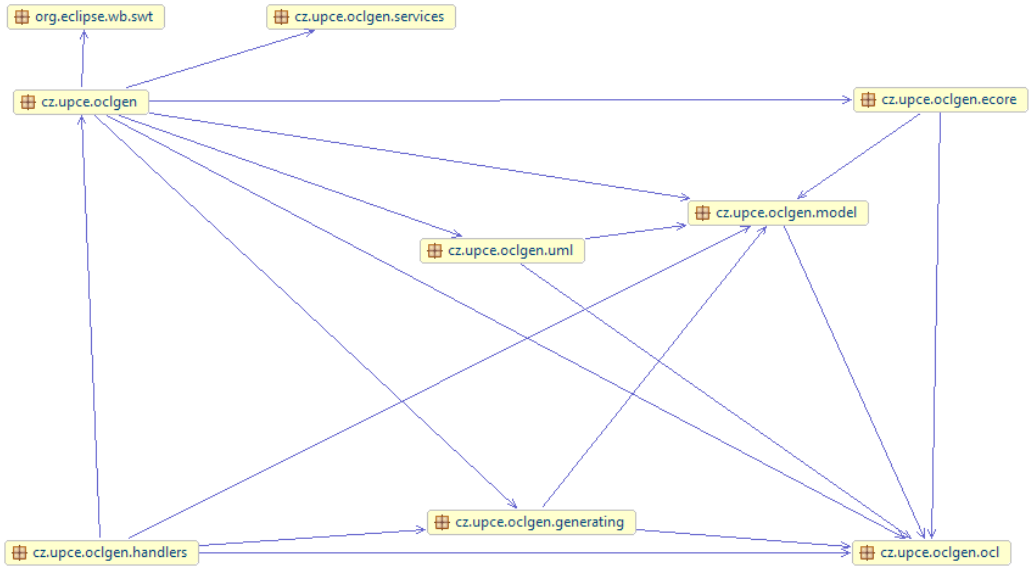
Obrázek 39 – Výčet balíčků OCL generátoru. Zdroj: autor

- **cz.upce.oclgen** – hlavní balíček obsahující aktivační třídu pluginu (*Activator.java*) a zobrazovací a jiné komponenty,
- **cz.upce.oclgen.ecore** – obsahuje implementační třídy metamodelu Ecore a třídy pro práci s Ecore modelem,
- **cz.upce.oclgen.generating** – obsahuje třídy pro generování OCL validačního souboru z modelu.
- **cz.upce.oclgen.handlers** – obsahuje handlery pro RCP command framework,
- **cz.upce.oclgen.model** – obsahuje rozhraní pro elementy modelu a pro třídy pracující s modelem společně s definicí vlastních modelových výjimek,
- **cz.upce.oclgen.ocl** – obsahuje definice OCL kontextu, omezení, funkcí apod.,
- **cz.upce.oclgen.services** – obsahuje služební třídy pro RCP command framework,

- **cz.upce.oclgen.uml** – obsahuje implementační třídy metamodelu UML a třídy pro práci s UML modelem,
- **org.eclipse.wb.swt** – obsahuje manažer zdrojů SWT.

Jsou využity návrhové vzory Singleton (realizovaný výčtovým typem v Javě) a Parametrized Factory.

Mezi balíčky se nevyskytují cyklické závislosti, jak ukazuje diagram na obrázku 40. Přítomnost takových závislostí by indikovala chybu v návrhu.



Obrázek 40 – Diagram závilostí balíčků OCL generátoru. Zdroj: autor

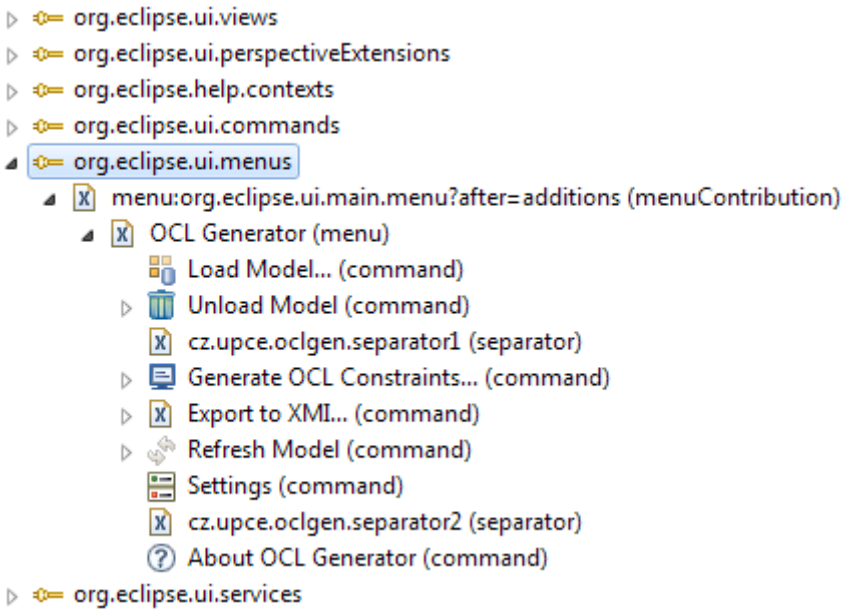
6.2.2 Konfigurace pluginu

OCL generátor je konfigurován prostřednictvím grafického rozhraní pro vývoj pluginů v Eclipse (jeho ukázka je na obrázku 41). Na základě prováděného nastavení je přepisován soubor *plugin.xml*. Konfiguraci lze v případě potřeby provádět i manuálním způsobem přímo ve zmiňovaném souboru. Konfigurační položky jsou následující:

- základní informace o pluginu,
- cesta k aktivační třídě,
- zda bude plugin zaváděn jako singleton,
- specifikace požadovaného běhového prostředí,
- definice závislostí na balíčcích a ostatních pluginech Eclipse,
- konfigurace rozšíření,
    - *commands* – definuje příkazy a k nim určené handlery, které se starají o jejich vykonání,
    - *menus* – definuje jednotlivá menu, jejich umístění, položky, zobrazovací logiku a navázání položek na příkazy definované v *commands*,
    - *views* – definuje zobrazovací komponenty a kategorie,

- *perspectiveExtensions* – definuje rozšíření perspektivy (Eclipse umožňuje ukládat profily nastavení UI a libovolně mezi nimi přepínat),
- *contexts* – definuje kontextovou nápovědu,
- *services* – obsahuje definici služeb,

- konfigurace buildu.



Obrázek 41 – Výřez z konfigurační části pluginu. Zdroj: autor

6.2.3 Nahrání modelu do pluginu

Nahrání modelu obstarává metoda *loadModel* třídy *EcoreModelManipulator* nebo *UMLModelManipulator* v závislosti na zvoleném metamodelu. Tato metoda je realizací abstraktní metody třídy *AModelManipulator*.

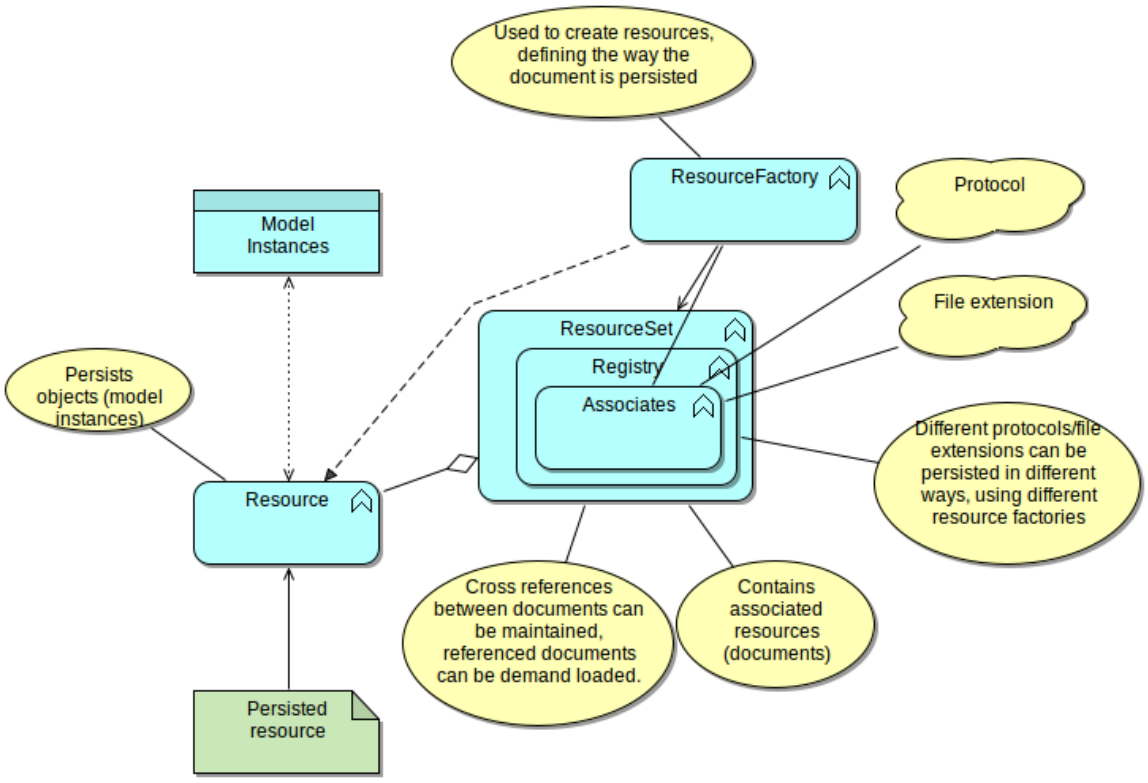
Vstupní model je ze svého zdrojového souboru načten jako zdroj (resource) EMF. Pokud je vstupem EMF UML model, je třeba zajistit dodatečné namapování EMF UML infrastruktury. S prvky modelu je pak možné pracovat na úrovni Java objektů. Přiložen je zdrojový kód pro nahrání EMF UML modelu (viz blok 60).

```
public void loadModel(String path) throws ModelManipulationException,
            FileNotFoundException {

      if (!new File(path).isFile())
            throw new FileNotFoundException("Model file not found");

      // Resource set init
      ResourceSetImpl resourceSet = new ResourceSetImpl();

      // Package registry settings for UML
      resourceSet.getPackageRegistry().put(UMLPackage.eNS_URI,
                  UMLPackage.eINSTANCE);
      // Provide UML extension mapping
      resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
                  .put("*", UMLResource.Factory.INSTANCE);
```

```
        URI uri = URI.createFileURI(path);

        // Load the model as a resource
        Resource resource = resourceSet.getResource(uri, true);

        // Erase any previous content
        unloadModel();

        model = resource;

        // Crawl the freshly loaded model
        crawlModel();

        // Empty or invalid format
        if (modelElements.size() == 0)
              throw new ModelManipulationException(
                    "Empty model or invalid EMF UML model format");
}
```

Blok 60 – Java. Metoda loadModel třídy UMLModelManipulator

Schéma správy zdrojů v EMF je zachyceno prostřednictvím rich picture diagramu na obrázku 42.



Obrázek 42 – Schéma správy zdrojů v EMF [42]

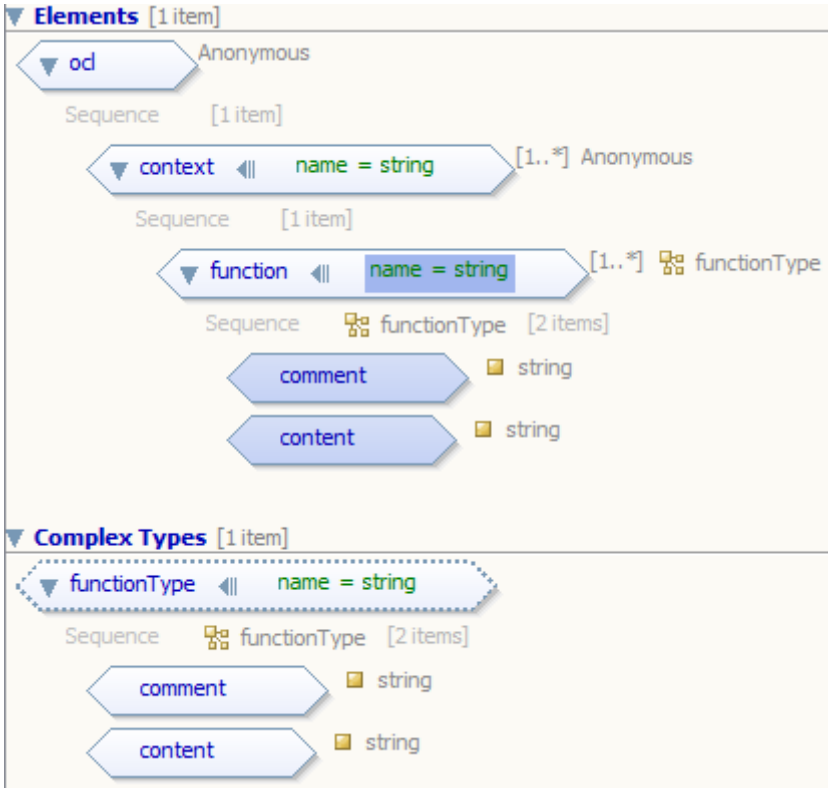
6.2.4 Podpůrné OCL funkce

Každý vygenerovaný *invariant* je závislý na jedné či více podpůrných funkcích, které jsou extrahovány z datového XML souboru příslušného metamodelu, pod nímž je model

vytvořen. Tento postup, kdy jsou výstupní *invarianty* závislé na externě uložených funkcích, s sebou nese kromě zřejmého zabránění opakování validačního kódu následující výhody:

- výstupní *invarianty* mají jednoduchou strukturu,
- podpůrné OCL funkce lze modifikovat či přidávat za běhu aplikace bez nutnosti její rekompilace,
- přenositelnost,
- datové XML soubory lze při nahrání validovat oproti definovanému XML schématu a tím ověřit integritu jejich struktury.

Vizualizace jednoduchého XML schématu, který OCL generátor využívá, je na obrázku 43. Schéma bylo sestaveno ve vývojovém prostředí NetBeans.



Obrázek 43 – Vizualizace XML schématu pro datový XML soubor s podpůrnými OCL funkcemi. Zdroj: autor

Každá podpůrná OCL funkce operuje na úrovni požadovaného kontextu a je extrahována prostřednictvím jazyka XPath (XML Path Language), který umožňuje v XML souboru navigovat. Přiložen je zdrojový kód metody *extractAuxiliaryFunction* třídy *XMLOCLAuxiliaryFunctionsParser*, která má extrakci podpůrných OCL funkcí na starost (viz blok 61). Tato třída využívá implementaci XML SAX parseru[19] v Javě.

```
@Override
```

[19] Na rozdíl od DOM parseru SAX funguje na bázi sekvenčního zpracování XML založeného na událostech.

```
public IOCLAuxiliaryFunction extractAuxiliaryFunction(
      OCLAuxiliaryFunctionType function) {
IOCLAuxiliaryFunction auxFunction = null;
try {
      // Get the node containing the required function
      XPathExpression exp = xpath.compile(String.format(
      "//oclgen:function[@name='%s']", function.toString()));
      Node node = (Node) exp.evaluate(doc, XPathConstants.NODE);

      // Node not found
      if (node == null)
            return null;

      // Extract the node content
      String comment = extractTextFromNode(node.getChildNodes().item(0));
      String functionContent = extractTextFromNode(node.getChildNodes()
            .item(1));
      String contextName = node.getParentNode().getAttributes()
            .getNamedItem("name").getNodeValue();

      OCLContext context = OCLContext.valueOf(contextName.toUpperCase());

      auxFunction = new OCLAuxiliaryFunction(context, functionContent,
            comment);
} catch (XPathExpressionException e) {
      // Malformed XPath expression, XML already validated against its schema
      return null;
}
      return auxFunction;
}
```

Blok 61 – Java. Metoda extractAuxiliaryFunction třídy XMLOCLAuxiliaryFunctionsParser

Následuje popis vybraných podpůrných OCL funkcí a rozbor jejich implementace pro metamodely Ecore a UML. Pro lepší orientaci v textu začínají úseky validačního kódu vždy komentářem s názvem metamodelu, pro který jsou určeny.

Sestavení kvalifikovaných názvů

Aby bylo možné jednoznačně identifikovat element modelu podle jeho názvu, je nutné sestavit tzv. kvalifikovaný název. Součástí toho názvu jsou hierarchicky uspořádané názvy balíčků od nejméně po nejhlouběji zanořený oddělené čtyřtečkou. Implementace UML metamodelu v Eclipse EMF pro tento případ zavádí odvozený atribut *qualifiedName*, který kvalifikovaný název sestaví automaticky. V implementaci Ecore metamodelu však podobná vlastnost absentuje, proto je třeba vytvořit podpůrnou OCL funkci, jež tento nedostatek nahradí (viz blok 62).

```
-- Ecore
context ENamedElement
/* Get the qualified name for the model package
(eg. MainPackage::SubPackage::SubSubPackage) */
def: getQualifiedName(ePackage : EPackage) : String =
let ePackages : Sequence(EPackage) = ePackage->closure(eSuperPackage)
->asSequence() in
let prefix : String = ePackages->iterate(currentEPackage; result : String = ''
| result.concat(currentEPackage.name+'::')) in
```

```
prefix.concat(ePackage.name)
```

Blok 62 – OCL. Sestavení kvalifikovaného názvu v Ecore

Funkce sestaví kvalifikovaný název balíčku (v Ecore je to *EPackage*) libovolného zanoření. Využívá se zde atributu *eSuperPackage*, který odkazuje na nadřazený balíček. S jeho pomocí je přes rekurzivní operaci *closure* vybudována sekvence názvů nadřazených balíčků sestupně podle zanoření. Sekvence je následně procházena operací *iterate* a jednotlivé názvy přidruženy k výstupnímu řetězci.

Existence elementů v modelu

Rozdíly v implementaci obou metamodelů mírně odlišují i způsob ověření existence elementů v modelu. Součástí implementace UML je kontext *Model*, pod kterým lze ověření v kombinaci s atributem *qualifiedName* uskutečnit. V Ecore je třeba operovat na úrovni jednotlivých balíčků kontextu *EPackage*. Příslušné podpůrné funkce ukazuje blok 63.

```
-- Ecore
context EPackage
-- Extract classes from designated package
def: classExtract(ePackage : EPackage) : Set(EClass) =
ePackage.eClassifiers->select(oclIsTypeOf(EClass)).oclAsType(EClass)->asSet()

-- Class exists within package check
def: classExists(elementName : String, ePackage : EPackage) : Boolean =
not classExtract(ePackage)->select(name.matches(elementName) and not abstract
and not interface)->any(true).oclIsUndefined()

-- Abstract class exists within package check
def: abstractClassExists(elementName : String, ePackage : EPackage) : Boolean =
not classExtract(ePackage)->select(name.matches(elementName) and abstract and
not interface)->any(true).oclIsUndefined()

-- Interface exists within package check
def: interfaceExists(elementName : String, ePackage : EPackage) : Boolean =
not classExtract(ePackage)->select(name.matches(elementName) and interface)
->any(true).oclIsUndefined()

-- Enumeration exists within package check
def: enumerationExists(elementName : String, ePackage : EPackage) : Boolean =
not ePackage.eClassifiers->select(oclIsTypeOf(EEnum)).oclAsType(EEnum)
->asSet()->select(name.matches(elementName))->any(true).oclIsUndefined()
```

Blok 63 – OCL. Sada podpůrných funkcí pro ověření existence elementů v Ecore

Podpůrná funkce *classExtract* extrahuje kolekci klasifikátorů daného balíčku, ve které se pak hledá konkrétní element. Ověření existence balíčku samotného pak probíhá na základě podpůrné funkce *packageExists*. Podpůrnou funkci zachycuje blok 64.

```
-- Ecore
-- Package exists within the model check
context EPackage
def: packageExists(elementName : String) : Boolean =
```

```
EPackage.allInstances()
->exists(element | getQualifiedName(element).matches(elementName))
```

Blok 64 – OCL. Ověření existence balíčku v Ecore

Vygenerovaný *invariant*, který ověří existenci konkrétního balíčku, pak např. může vypadat jako dle bloku 65.

```
-- Ecore
context EPackage
-- Package Decorator exists within the model
inv PackageDecoratorExistsWithinModel('Package Decorator does not exist within
the model'):
eSuperPackage.oclIsUndefined() implies packageExists('Decorator')
```

Blok 65 – OCL. Ukázka invariantu pro ověření existence balíčku v Ecore

Kontrolu je nutné provádět na úrovni hlavního balíčku modelu, což zajistí výraz `eSuperPackage.oclIsUndefined()` užitý v implikaci (hlavní balíček modelu zřejmě nemá žádný nadřazený).

Konečně *invariant* pro existenci konkrétního elementu v EMF Ecore modelu má podobu dle bloku 66. V tomto případě je ověřena existence třídy *Steak* v balíčku *Decorator*.

```
-- Ecore
-- Class Steak exists within package Decorator
context EPackage
inv ClassSteakExistsWithinPackageDecorator('Class Steak does not exist within
package Decorator'):
getQualifiedName(self).matches('Decorator') implies classExists('Steak', self)
```

Blok 66 – OCL. Ukázka invariantu pro ověření existence třídy v Ecore

V EMF UML je situace jednodušší díky přítomnosti zmiňovaného kontextu *Model* a atributu *qualifiedName*. Podpůrné funkce jsou zachyceny v bloku 67.

```
-- UML
context Model
-- Class exists check
def: classExists(elementName : String) : Boolean =
not Class.allInstances()->select(qualifiedName.matches(elementName))
->any(true).oclIsUndefined()

-- Interface exists check
def: interfaceExists(elementName : String) : Boolean =
not Interface.allInstances()->select(qualifiedName.matches(elementName))
->any(true).oclIsUndefined()

-- Enumeration exists check
def: enumerationExists(elementName : String) : Boolean =
not Enumeration.allInstances()->select(qualifiedName.matches(elementName))
->any(true).oclIsUndefined()
```

Blok 67 – OCL. Sada podpůrných funkcí pro ověření existence elementů v UML

Pomocí operace *allInstances* v kontextu *Model* lze přistupovat ke všem instancím jednotlivých typů klasifikátorů v modelu. Pak už jen stačí nalézt shodu v kvalifikovaném názvu.

Abstraktnost tříd se v tomto případě ověřuje odděleně, jak ukazuje blok 68.

```
-- UML
context Class
-- Class Lektor is abstract
inv ClassLektorIsAbstract('Class Lektor is not abstract'):
qualifiedName.matches('Model::Lektor') implies isAbstract
```

Blok 68 – OCL. Ukázka invariantu pro ověření abstraktnosti třídy v UML

Tento *invariant* ověří, zda je třída *Lektor*, definovaná v hlavním balíčku modelu, abstraktní.

Generalizace a realizace

Implementace Ecore nerozlišuje tyto dva vztahy. Každá *EClass* si udržuje kolekci *eSuperTypes*, ve které se nacházejí klasifikátory, jež daná třídá generalizuje nebo realizuje. O jaký vztah se pak jedná, lze rozhodnout na základě atributů *abstract* a *interface* dané instance *EClass* (viz blok 69).

```
-- Ecore
context EClass
-- Ecore specific generalization/realization check
def: classExtendsClass(eClass : EClass, superTypeName : String, ePackageName :
String) : Boolean =
eClass.eSuperTypes->exists(name.matches(superTypeName) and
getQualifiedName(ePackage).matches(ePackageName))
```

Blok 69 – OCL. Generalizace v Ecore

Konkrétní případ *invariantu* popisuje blok 70. *Invariant* ověří, zda třída *Hamburger* definovaná v balíčku *Decorator* dědí abstraktní třídu *Pokrm*.

```
-- Ecore
context EClass
-- Class Hamburger extends AbstractClass Pokrm
inv ClassHamburgerExtendAbstractClassPokrm('Class Hamburger does not extend
AbstractClass Pokrm'):
name.matches('Hamburger') and getQualifiedName(ePackage).matches('Decorator')
implies classExtendsClass(self, 'Pokrm', 'Decorator')
```

Blok 70 – OCL. Ukázka ověření generalizace v Ecore

V implementaci UML je situace mírně odlišná. Nejprve je uvedena podpůrná funkce pro ověření generalizace (viz blok 71). Atribut *general* poskytuje kolekci klasifikátorů, ze kterých třída dědí.

```
-- UML
context Class
-- Class extends another class
```

```
def: classExtendsClass(class : Class, ancestorName : String) : Boolean =
class.general->exists(qualifiedName.matches(ancestorName) and
oclIsTypeOf(Class))
```

Blok 71 – OCL. Generalizace pro třídy v UML

Dále jsou uvedeny podpůrné funkce pro kontext *NamedElement*, pod který spadá jakýkoli element modelu disponující pojmenováním.

Následující funkce je analogická k předchozí, jen je určena pro rozhraní (viz blok 72).

```
-- UML
context NamedElement
-- Interface extends another interface
def: interfaceExtendsInterface(interface : Interface, ancestorName : String) :
Boolean =
interface.general->exists(qualifiedName.matches(ancestorName) and
oclIsTypeOf(Interface))
```

Blok 72 – OCL. Generalizace pro rozhraní v UML

Podpůrná funkce *namedElementRealizesInterface* ověří, zda daný klasifikátor realizuje požadované rozhraní (viz blok 73). Informaci je třeba extrahovat z kolekce *clientDependency*, která uchovává závislosti mezi elementy modelu. Nutné je také provést vhodnou typovou filtraci. Pro implementaci UML metamodelu v Eclipse EMF je charakteristická vysoká míra flexibility, kdy např. výčtový typ může realizovat rozhraní, což je skutečně i vlastnost programovacího jazyka Java verze 5 a vyšší.

```
-- UML
context NamedElement
-- NamedElement realizes interface
def: namedElementRealizesInterface(elementName : String, interfaceName :
String) : Boolean =
let src : Set(NamedElement) = clientDependency.source->reject(not
oclIsKindOf(NamedElement)).oclAsType(NamedElement),
trgt : Set(Interface) = clientDependency.target->reject(not
oclIsTypeOf(Interface)).oclAsType(Interface)
in
clientDependency->exists(
not src->any(qualifiedName.matches(elementName)).oclIsUndefined()
and
not trgt->any(qualifiedName.matches(interfaceName)).oclIsUndefined())
```

Blok 73 – OCL. Realizace v UML

Asociace a jejich vlastnosti

Z pohledu asociace je nutné se zaměřit na typ, násobnost a průchodnost obou jejích konců. Asociace jsou v každém z metamodelů implementovány v Eclipse EMF rozdílně. Ecore pojem asociace nezná, protože vychází z EMOF, zatímco UML z CMOF. Místo toho pracuje s referencemi (typ *EReference*), které jsou součástí jednotlivých klasifikátorů. Z toho vyplývá, že v případě Ecore nemá smysl testovat průchodnost (reference je vždy průchozí), a také skutečnost, že reference má pouze jeden konec. Naproti tomu EMF UML

implementace přistupuje k asociacím jako k samostatným elementům, které nejsou k ničemu přidruženy a mají klasicky dva konce.

V zájmu jednotnosti a menšího zmatku je k Ecore referencím v kódu a uživatelském rozhraní přistupováno názvoslovně jako k asociacím, byť je to do jisté míry zavádějící. Uvedená podpůrná funkce *associationFeatureCheck* kontroluje v závislosti na uživatelském nastavení OCL generátoru typ a násobnost reference (viz blok 74). Jelikož se nelze odvolávat na kvalifikované názvy referencí (vynucování přesných názvů by degradovalo použití aplikace pro výukové účely), je třeba ověřovat i počet referencí shodných vlastností, které cílí na stejný klasifikátor.

```
-- Ecore
context EClass
-- Association feature check based on user's preferences (type/multiplicity
checks). Navigability is auto-checked (Ecore specific).
def: associationFeatureCheck(
checks : Tuple(typeCheck : Boolean, multiplicityCheck : Boolean),
eClass : EClass,
end : Tuple(typeName : String, ePackageName : String, isContainment : Boolean,
lower : Integer, upper : Integer),
count : Integer) : Boolean =
-- Associations extraction
let associationSet : Set(EReference) = associationExtract(eClass,
Tuple{typeName = end.typeName, ePackageName = end.ePackageName}) in
not associationSet->isEmpty() implies(
associationSet->select(ref : EReference |
-- Is containment check
(checks.typeCheck
implies
associationTypeCheck(ref, end.isContainment)
)
and
-- Multiplicity check
(checks.multiplicityCheck
implies
associationMultiplicityCheck(ref, end.lower, end.upper)
)
)->size() >= count)
```

Blok 74 – OCL. Ověření vlastností reference v Ecore

Extrakce referencí z klasifikátoru, ověření existence reference, ověření typu a násobnosti jsou delegovány na následující podpůrné funkce.

Funkce *associationExtract* vrací kolekci všech Ecore referencí cílících na požadovaný klasifikátor (viz blok 75).

```
-- Ecore
context EClass
-- Association extract
def: associationExtract(eClass : EClass, end : Tuple(typeName : String,
ePackageName : String)) : Set(EReference) =
eClass.eReferences->select(eType.name.matches(end.typeName) and
getQualifiedName(eType.ePackage).matches(end.ePackageName))
```

Blok 75 – OCL. Extrakce reference v Ecore

Funkce *associationExists* ověří existenci požadovaného počtu referencí cílících na daný klasifikátor (viz blok 76).

```
-- Ecore
context EClass
-- Association exists check (navigability autocheck)
def: associationExists(eClass : EClass, end : Tuple(typeName : String,
ePackageName : String), count : Integer) : Boolean =
associationExtract(eClass, end)->size() >= count
```

Blok 76 – OCL. Ověření existence reference v Ecore

Funkce *associationTypeCheck* ověří typ reference (viz blok 77). Ecore zná pouze typ *containment*, který je v jistém smyslu analogický ke kompozici v UML.

```
-- Ecore
context EClass
-- Association type check
def: associationTypeCheck(eReference : EReference, isContainment : Boolean) :
Boolean =
eReference.containment = isContainment
```

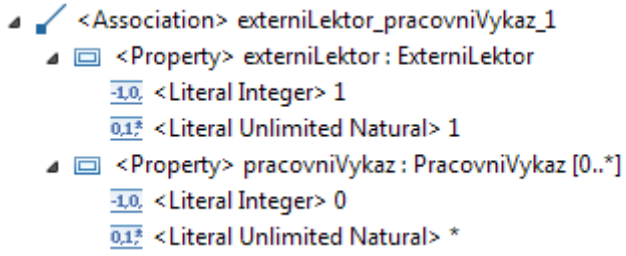
Blok 77 – OCL. Ověření typu reference v Ecore

Funkce *associationMultiplicityCheck* ověří dolní a horní hranici intervalu násobnosti (viz blok 78).

```
-- Ecore
context EClass
-- Association multiplicity check
def: associationMultiplicityCheck(eReference : EReference, lower : Integer,
upper : Integer) : Boolean =
eReference.lowerBound = lower and eReference.upperBound = upper
```

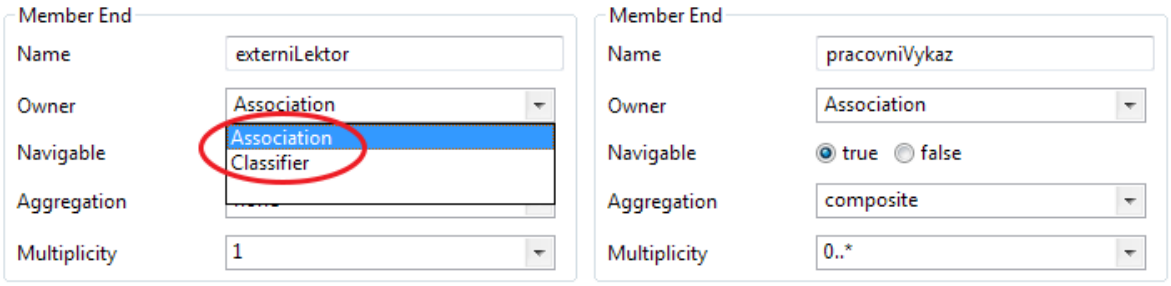
Blok 78 – OCL. Ověření násobnosti reference v Ecore

Jak bylo uvedeno, V EMF UML má asociace dva konce a navíc lze testovat jejich průchodnost. Každý konec je identifikován podle kvalifikovaného názvu klasifikátoru, ke kterému je přidružen. Problém však vzniká u reflexivních asociací, kdy od sebe nelze rozlišit jednotlivé konce, protože oba kvalifikované názvy jsou přirozeně shodné. Implementace asociace v EMF UML navíc **nerozlišuje** jednotně jejich pořadí. Konce jsou vždy uspořádány podle toho, z jakého směru uživatel při tvorbě diagramu asociaci táhne.



Obrázek 44 – Ukázka asociace v EMF UML. Zdroj: autor

Situace se dále zhoršuje vlivem skutečnosti, že v EMF UML lze dále nastavit vlastníka konce asociace. V případě, že je jako vlastník příslušného konce nastaven klasifikátor, dojde k odštěpení tohoto konce od vlastního elementu asociace, což má za následek nemožnost ověření jeho vlastností. Při tvorbě UML modelu v prostředí Eclipse EMF je tedy vždy nutné dbát na to, aby oba konce byly vlastněny jejich asociací. Pokud je vlastníkem příslušného konce klasifikátor, objeví se u něj v příslušném diagramu tříd indikátor v podobě černé tečky, čímž lze tento případ snadno vizuálně identifikovat.



Obrázek 45 – Nastavení konců asociace v modelovacím nástroji Papyrus. Zdroj: autor

Následuje podpůrná funkce *associationFeatureCheck* pro EMF UML (viz blok 79). U reflexivních asociací neprobíhá ověřování vlastností jejich konců.

```
-- UML
context Model
-- Association feature check based on user's preferences
(navigability/type/multiplicity checks)
def: associationFeatureCheck(
checks : Tuple(navigabilityCheck : Boolean, typeCheck : Boolean,
multiplicityCheck : Boolean),
end1 : Tuple(name : String, navigable: Boolean, kind : AggregationKind, lower :
Integer, upper : Integer),
end2 : Tuple(name : String, navigable: Boolean, kind : AggregationKind, lower :
Integer, upper : Integer),
count : Integer) : Boolean =
-- Associations extraction
let associationSet : Set(Association) = associationExtract(end1.name,
end2.name) in
-- Reflexive associations not supported due to the EMF UML2 limitations
not associationSet->isEmpty() and not end1.name.matches(end2.name) implies(
associationSet->select(a : Association |
-- Navigability check
(checks.navigabilityCheck
implies
associationNavigabilityCheck(a, Tuple{typeName = end1.name, navigable =
end1.navigable},
      Tuple{typeName = end2.name, navigable = end2.navigable}
))
and
-- Aggregation type/kind check
(checks.typeCheck
implies
associationTypeCheck(a, Tuple{typeName = end1.name, kind = end1.kind},
      Tuple{typeName = end2.name, kind = end2.kind}
))
```

```
and
-- Multiplicity check
(checks.multiplicityCheck
implies
associationMultiplicityCheck(a, Tuple{typeName = end1.name, lower = end1.lower,
upper = end1.upper},
      Tuple{typeName = end2.name, lower = end2.lower, upper = end2.upper}
))
)->size() >= count)
```

Blok 79 – OCL. Ověření vlastností asociace v UML

Extrakce se provádí pouze pro ty asociace, kterým patří oba jejich konce (viz blok 80).

```
-- UML
context Model
-- Association extraction
def: associationExtract(classifier1 : String, classifier2 : String) :
Set(Association) =
Association.allInstances()->select(member.oclAsType(Property)->forAll(not
type.oclIsUndefined()))->select(member
.oclAsType(Property)->forAll(type.qualifiedName.matches(classifier1) or
type.qualifiedName.matches(classifier2)))->reject(member->size() < 2)->asSet()
```

Blok 80 – OCL. Extrakce asociace v UML

Existence je opět ověřována pro požadovaný počet asociací (viz blok 81).

```
-- UML
context Model
-- Association exists check
def: associationExists(classifier1 : String, classifier2 : String, count :
Integer) : Boolean =
associationExtract(classifier1, classifier2)->size() >= count
```

Blok 81 – OCL. Ověření existence asociace v UML

Dále je kontrolována průchodnost jednotlivých konců asociace na základě informací z kolekce, kterou poskytuje atribut *navigableOwnedEnd* (viz blok 82).

```
-- UML
context Model
-- Association navigability check
def: associationNavigabilityCheck(association : Association,
      end1 : Tuple(typeName : String, navigable : Boolean),
      end2 : Tuple(typeName : String, navigable : Boolean)
) : Boolean =
association.navigableOwnedEnd.oclAsType(Property)
->exists(type.qualifiedName.matches(end1.typeName)) = end1.navigable
and
association.navigableOwnedEnd.oclAsType(Property)
->exists(type.qualifiedName.matches(end2.typeName)) = end2.navigable
```

Blok 82 – OCL. Ověření průchodnosti konců asociace v UML

Jako typ konce asociace je v EMF UML možné zvolit *none*, *shared* nebo *composite*. V určeném pořadí se jedná o standardní asociaci (třída *A* ví o třídě *B*), agregaci a kompozici. Tyto hodnoty jsou obsaženy ve výčtovém typu *AggregationKind* (viz blok 83).

```
-- UML
context Model
-- Association type check
def: associationTypeCheck(association : Association,
      end1 : Tuple(typeName : String, kind : AggregationKind),
      end2 : Tuple(typeName : String, kind : AggregationKind)
) : Boolean =
let properties : OrderedSet(Property) = association.member->asOrderedSet() in
properties->size() = 2 implies
(let first : Property = properties
->select(type.qualifiedName.matches(end1.typeName))->first(),
second : Property = properties
->select(type.qualifiedName.matches(end2.typeName))->first() in
first.aggregation = end1.kind and second.aggregation = end2.kind)
```

Blok 83 – OCL. Ověření typu konců asociace v UML

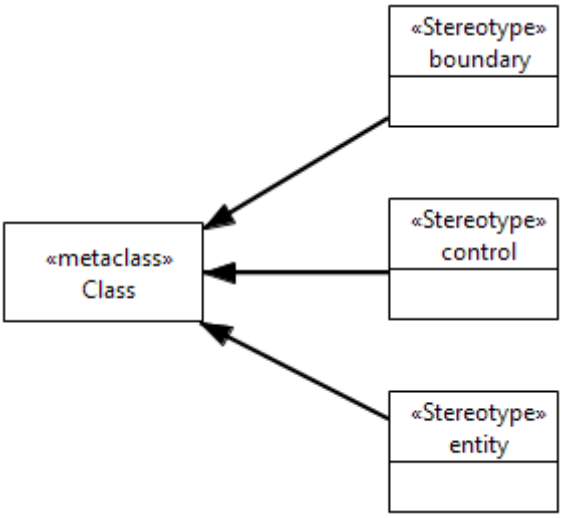
Poslední podpůrná funkce ověřuje oba intervaly násobnosti (viz blok 84).

```
-- UML
context Model
-- Association multiplicity check
def: associationMultiplicityCheck(
      association : Association,
      end1 : Tuple(typeName : String, lower : Integer, upper : Integer),
      end2 : Tuple(typeName : String, lower : Integer, upper : Integer)
      ) : Boolean =
let properties : OrderedSet(Property) = association.member->asOrderedSet() in
properties->size() = 2 implies
(let first : Property = properties
->select(type.qualifiedName.matches(end1.typeName))->first(),
second : Property = properties
->select(type.qualifiedName.matches(end2.typeName))->first() in
first.lower = end1.lower and first.upper = end1.upper and second.lower =
end2.lower and second.upper = end2.upper)
```

Blok 84 – OCL. Ověření násobnosti konců asociace v UML

MVC stereotypy

Byla učiněna snaha rozšířit funkcionalitu OCL generátoru o kontrolu MVC (Model-view-controller) stereotypů pro UML třídy. Nejprve byl vytvořen UML MVC profil, který rozšiřuje metatřídu o stereotypy *boundary*, *control* a *entity*. Tento profil by pak bylo nutné aplikovat na požadovaný model, přičemž kontrola by probíhala pomocí operace *getAppliedStereotype*, která je součástí implementace metamodelu UML v Eclipse EMF. Zatímco však tato operace v OCL konzoli vykazuje požadovanou funkčnost, v Complete OCL (OCL je uloženo v samostatném validačním souboru s příponou ocl, což je výstup generátoru) podpora UML operací bohužel zatím ještě není integrována.



Obrázek 46 – UML MVC profil. Zdroj: autor

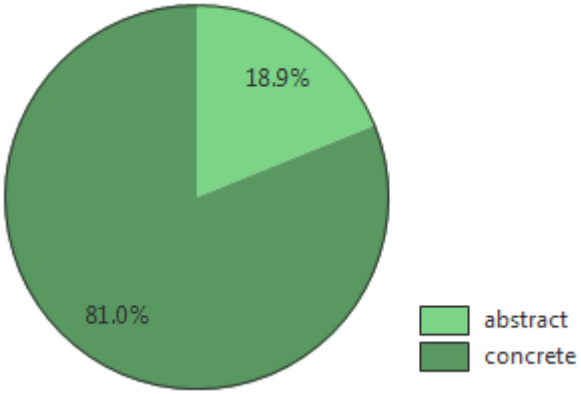
Vývoj OCL nástrojů a rozšíření pro Eclipse EMF však postupuje stále dopředu. Ve chvíli, kdy bude tato překážka odstraněna, je možné vydat příslušnou aktualizaci pro OCL generátor.

6.2.5 Generování validačního kódu

Hlavní část činnosti zajišťuje metoda *generateOCLDocument* třídy *OCLGenerator*. Metoda jako svůj parametr přijímá bitové pole (realizované strukturou *EnumSet*), které obsahuje pravidla generování. Postupně jsou procházeny jednotlivé elementy modelu, které uživatel nevyjmul z generování. Zpracovány jsou také neviditelné elementy, jako např. asociace a balíčky. Pro každý z nich je dále zjištěno, které podpůrné OCL funkce jsou k vygenerovanému *invariantu* třeba. Ty jsou poté extrahovány z příslušného XML datového souboru. Jsou sestaveny dva regiony validačního kódu, podpůrné OCL funkce a samotné *invarianty*. Ukázka zdrojového kódu vybraných metod třídy *OCLGenerator* je součástí přílohy D.

6.3 Analýza zdrojového kódu

Zdrojový kód aplikace byl analyzován nástrojem CodePro společnosti Google. Nástroj poskytuje programátorovi relevantní metriky zdrojového kódu a umožňuje jeho auditování (upozorňuje na redundance, zbytečná přetypování apod.). Na základě provedené analýzy byla zredukována velikost kódu a ošetřena slabá místa. Ukázka jednoho z výstupů CodePro je na obrázku 47.



Obrázek 47 – Metrika využití abstraktních typů ve zdrojovém kódu OCL generátoru. Zdroj: autor

6.4 Ověření funkčnosti aplikace

Aplikace byla podrobena experimentování, kdy se zkoumalo její chování jako celku a také reakce na různé vstupy. Výstupní validační OCL kód byl v součinnosti s validačními komponenty Eclipse EMF podrobně zkoušen pro různé případy modelů a to jak obecně tak i selektivně pro jednotlivá pravidla generování. Zapojena byla i nezainteresovaná osoba, která nedisponovala detailními znalostmi o řešené problematice. Nalezené chyby byly opraveny s výjimkou kontroly vlastností reflexivních asociací u EMF UML modelů, která nemohla být vyřešena z důvodu implementačních omezení UML metamodelu v Eclipse EMF. Testování probíhalo ve verzích Eclipse označených kódovými názvy Juno a Kepler.

Funkčnost aplikace je doložena dvojicí ukázkových úloh, které jsou součástí přílohy C.

7 Závěr

V úvodní části diplomové práce byly vysvětleny principy deklarativního jazyka Object Constraint Language (OCL) a možnosti jeho využití pro objektově orientované modely.

Proběhlo zmapování podpory OCL v modelovacím CASE nástroji Enterprise Architect napříč jednotlivými verzemi. Nástroj byl s ohledem na cíle diplomové práce (ověření OCL integritních omezení na vlastním objektově orientovaném modelu) shledán nevyhovujícím. V závislosti na tomto zjištění byla provedena rešerše nekomerčních nástrojů podporujících jazyk OCL, jejich otestování a srovnání. Pro další postup byl vybrán Eclipse Modeling Framework (EMF), který je součástí vývojového a modelovacího prostředí Eclipse.

Byl vytvořen ukázkový model polikliniky a definovaná integritní omezení v jazyce OCL ověřena na jeho dynamické instanci. Na modelu byly taktéž představeny dotazovací schopnosti jazyka OCL a deklarativní přístup k modelování workflow.

Dále byl vytvořen OCL plugin pro Eclipse EMF. Plugin umožňuje generování validačního kódu v jazyce OCL z referenčního modelu na základě zvolené konfigurace generačních pravidel. Výstup je pak možné použít v součinnosti s validačními komponenty Eclipse EMF pro strojovou kontrolu školních úloh zaměřených na tvorbu diagramů tříd. Plugin podporuje metamodely Ecore a UML a je navržen tak, aby mohl být snadno funkčně rozšířen v oblasti kontroly modelů. Pro plugin byl zřízen online repozitář umožňující jeho pohodlnou instalaci a stažení případných aktualizací v prostředí Eclipse. Předpokládá se experimentální nasazení tohoto nástroje v kurzech objektově orientovaného modelování.

Jako největší slabina jazyka OCL se ukázala slabá podpora ze strany modelovacích CASE nástrojů. V průběhu vypracovávání diplomové práce bylo nutné několikrát improvizovat a překonávat různé nedodělky a nástrahy zvoleného OCL prostředí. Vše se však podařilo zdárně vyřešit.

Práce ukázala, že jazyk OCL může být vítaným doplňkem objektově orientovaných modelů a má smysl se jím zabývat.

Použité zdroje

[1] **Warmer, Jos a Kleppe, Anneke.** *The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition).* Addison-Wesley, 2003. ISBN: 978-0321179364.

[2] **OMG.** OMG Object Constraint Language (OCL) 2.3.1. *Object Management Group.* [Online] 1. 1. 2012. [Citace: 5. 11. 2012.] http://www.omg.org/spec/OCL/2.3.1/PDF/.

[3] **Arlow, Jim a Neustadt, Ila.** *UML2 a unifikovaný proces vývoje aplikací.* Computer Press a.s., 2008. ISBN 978-80-251-1503-9.

[4] **Richta, Karel.** Jazyk OCL a modelem řízený vývoj. *Katedra softwarového inženýrství, MFF UK.* [Online] 2010. [Citace: 10. 10. 2012.] https://www.ksi.mff.cuni.cz/~richta/publications/Richta-MD-2010.pdf.

[5] **Hußmann, Prof.** Formal Specification of Software Systems. *Technische Universitat Dresden.* [Online] [Citace: 4. 10. 2012.] http://www-st.inf.tu-dresden.de/fs/slides/fss5a-sl.pdf.

[6] **Cook, Steve a Daniels, John.** *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, 1994. ISBN 0-13-203860-9.

[7] **Etutorials.org.** Constraint Rules. *Etutorials.org.* [Online] [Citace: 10. 11. 2012.] http://etutorials.org/Programming/UML/Chapter+4.+Class+Diagrams+The+Essentials/Constraint+Rules/.

[8] **Cabot, Jordi.** Ambiguity issues in OCL postconditions. [Online] [Citace: 25. 4. 2013.] http://jordicabot.com/papers/WOCL06.pdf.

[9] **Sparx Systems.** Model Transformations - MDA. *Sparx Systems.* [Online] 19. 2. 2013. http://www.sparxsystems.com/enterprise_architect_user_guide/software_development/mdastyletransforms.html.

[10] **Sulistyo, Selo a Najib, Warsun.** EXECUTABLE UML. [Online] http://www.oocities.org/warsunnajib/Warsun2file/Executable_UML_Report.pdf.

[11] **Jiang, Ke, Zhang, Lei a Miyake, Shigeru.** Using OCL in Executable UML. [Online] 2007. http://www-st.inf.tu-dresden.de/Ocl4All2007/papers/jiang.pdf.

[12] **Hamann, Lars, Hofrichter, Oliver a Gogolla, Martin.** On Integrating Structure and Behavior Modeling with OCL. *University of Bremen, Computer Science Department.* [Online] 2012. http://www.db.informatik.uni-bremen.de/publications/Hamann_2012_MODELS.pdf.

[13] **OMG.** Model Driven Architecture. *Object Management Group.* [Online] 30. 10 2012. [Citace: 13. 11. 2012.] http://www.omg.org/mda/.

[14] **Warmer, Jos.** The role of OCL in the Model Driven Architecture. *International Symposia on Formal Methods for Components and Objects.* [Online] 2010. [Citace: 15. 11. 2012.] fmco.liacs.nl/2002/warmer.ppt.

[15] **OMG.** Meta Object Facility. *Object Management Group.* [Online] 8. 11. 2007. [Citace: 11. 12 2012.] http://www.omg.org/spec/MOF/2.4.1/PDF/.

[16] **Goldschmidt, Thomas.** *View-based textual modelling.* KIT Scientific Publishing, Karlsruhe, 2011. ISBN 978-3-86644-642-7.

[17] **UMLBase.com.** The UML Metamodel. *UMLBase.* [Online] 2012. [Citace: 11. 12. 2012.] http://umlbase.com/learn/tag/meta-object-facility/.

[18] **Pícka, Marek.** Metamodelování a vývoj informačních systémů. [Online] 2004. [Citace: 20. 4. 2013.] http://www.agriculturejournals.cz/publicFiles/58690.pdf.

[19] **Cadavid, Juan.** MOF/OCL Analysis. [Online] [Citace: 11. 4. 2013.] http://people.rennes.inria.fr/Juan.Cadavid/?page_id=48.

[20] **Demuth, Birgit a Wilke, Claas.** Model and Object Verification by Using Dresden OCL. [Online] 1 2009. [Citace: 23. 4. 2013.] http://www.claaswilke.de/publications/workshops/demuth_ufa09.pdf.

[21] **Kolovos, Dimitris, a další, a další.** The Epsilon Book. *Epsilon.* [Online] 22. 1. 2013. [Citace: 28. 1. 2013.] http://www.eclipse.org/epsilon/doc/book/.

[22] **JMC Suprema Group One 2008.** An Introduction To Alloy. *A Guide to Alloy.* [Online] [Citace: 5. 2. 2013.] http://www.doc.ic.ac.uk/project/examples/2007/271j/suprema_on_alloy/Web/intro.php.

[23] **Jackson, Daniel.** Alloy FAQ. *Alloy.* [Online] 2012. [Citace: 5. 2. 2013.] http://alloy.mit.edu/alloy/faq.html.

[24] **Sparx Systems.** Enterprise Architect History. *Enterprise Architect - UML Design Tools and UML CASE tools for software development.* [Online] [Citace: 10. 11. 2012.] Dostupné z http://www.sparxsystems.com/products/ea/history.html.

[25] **Sparx Systems.**Enterprise Architect Editions. *Enterprise Architect - UML Design Tools and UML CASE tools for software development.* [Online] [Citace: 10. 11. 2012.] http://www.sparxsystems.com/products/ea/index.html#editions.

[26] **Sparx Systems.** *Sparx Systems Forum.* [Online] [Citace: 11. 11. 2012.] http://www.sparxsystems.com/cgi-bin/yabb/YaBB.cgi.

[27] **EmPowerTec.** *EmPowerTec.* [Online] [Citace: 13. 11. 2012.] http://www.empowertec.de/.

[28] **CRAFTWARE Consultores Ltda.** *Enterprise Analyst.* [Online] [Citace: 16. 11. 2012.] http://www.enterpriseanalyst.net/index_us.php.

[29] **CRAFTWARE Consultores Ltda.** News. *Enterprise Analyst.* [Online] [Citace: 16. 11. 2012.] http://www.enterpriseanalyst.net/htm/en/noticias.htm.

[30] **Donátek, Jan.** Generování SQL ze specifikace UML a OCL. *Bakalářská práce.* ČVUT FEL Praha, 2011.

[31] **Sparx Systems.** Enterprise Architect 10. *Enterprise Architect - UML Design Tools and UML CASE tools for software development.* [Online] [Citace: 15. 11. 2012.] http://www.sparxsystems.com.au/products/ea/10/index.html.

[32] **Univerzita Babes Bolyai, Cluj-Napoca, Rumunsko.** ocle Object Constraint Language Environment. [Online] 1. 6 2005. [Citace: 15. 11. 2012.] http://lci.cs.ubbcluj.ro/ocle/index.htm.

[33] **Universität Bremen.** USE: UML-based Specification Environment. [Online] 9. 3. 2013. [Citace: 16. 3. 2013.] http://sourceforge.net/projects/useocl/.

[34] **The Eclipse Foundation.** Eclipse Modeling Framework Project (EMF). *Eclipse.org.* [Online] [Citace: 16. 11. 2012.] http://www.eclipse.org/modeling/emf/.

[35] **The Eclipse Foundation.** Downloads. *Eclipse.org.* [Online] [Citace: 16. 11. 2012.] http://www.eclipse.org/modeling/mdt/downloads/?project=uml2.

[36] **Vogel, Lars.** Eclipse Modeling Framework (EMF) - Tutorial. *Vogella.com.* [Online] 29. 12. 2012. [Citace: 20. 4. 2013.] http://www.vogella.com/articles/EclipseEMF/article.html.

[37] **Bacvanski, Vladimir a Graff, Petter.** Mastering Eclipse Modeling Framework. *Eclipsecon.org.* [Online] 28. 2. 2005. [Citace: 16. 11. 2012.] http://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf.

[38] **The Eclipse Foundation.** Project Plan For Model Development Tools (MDT). *Eclipse.org.* [Online] 2012. [Citace: 16. 11. 2012.] http://www.eclipse.org/projects/project-plan.php?planurl=http://www.eclipse.org/modeling/mdt/ocl/project-info/plan_juno.xml&component=Eclipse#introduction.

[39] **The Eclipse Foundation.** UML2 Tools. *Eclipse.org.* [Online] 2012. [Citace: 16. 11. 2012.] http://www.eclipse.org/modeling/mdt/downloads/?project=uml2tools.

[40] **Wilke, Claas, a další, a další.** Documentation. *Dresden OCL.* [Online] 26. 9. 2012. [Citace: 28. 1. 2013.] http://141.76.65.213/dresdenocl_updatesite/manual.pdf.

[41] **Brüning, Jens.** Declarative Workflow Modeling with UML class diagrams. [Online] 2009. [Citace: 8. 4. 2013.] http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/jens/papers/YRW-MBP'09.pdf.

[42] **Magosányi, Árpád.** *Eclipse Community Forums.* [Online] 11. 12. 2011. [Citace: 7. 4. 2013.] http://www.eclipse.org/forums/index.php/m/764497/.

[43] **OMG.** MDA Specifications. *Object Management Group.* [Online] [Citace: 13. 11. 2012.] http://www.omg.org/mda/specs.htm#XMI.

[44] **Zikmund, Milos.** Diplomová práce Aspektově orientované programování a jeho podpora. *Informační systém Masarykovy univerzity*. [Online] 30. 6. 2010. [Citace: 28. 1. 2013.] http://is.muni.cz/th/173275/fi_m/dp.pdf.

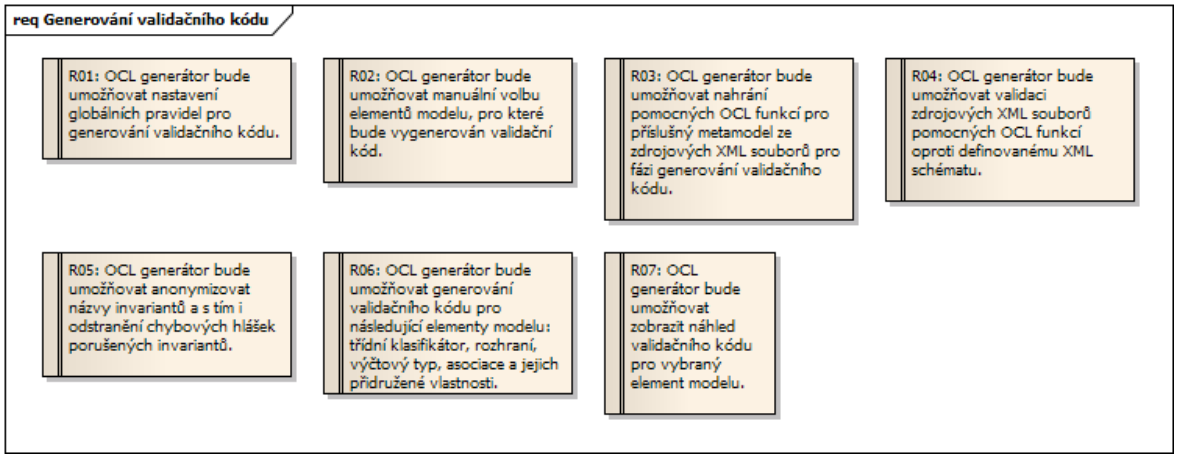
[45] **Object Mentor Inc.** The Liskov Substitution Principle. *Object Mentor.* [Online] [Citace: 16. 2. 2013.] http://www.objectmentor.com/resources/articles/lsp.pdf.

[46] **OMG.** OMG Unified Modeling Language, Superstructure. *Object Management Group.* [Online] 6. 8. 2011. [Citace: 1. 4 2013.] http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

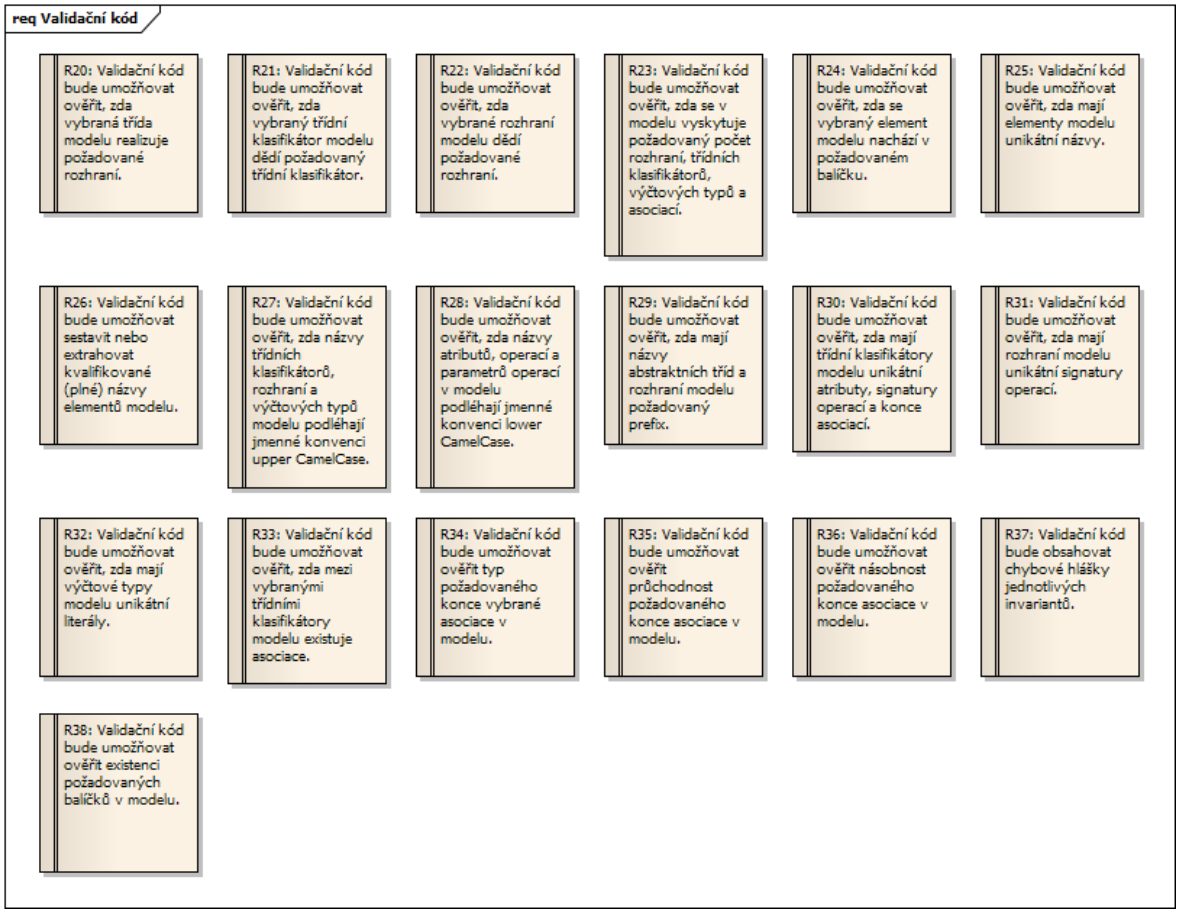
[47] **OMG.** Systems Modeling Language (SysML). *Object Management Group.* [Online] 6. 1. 2012. [Citace: 7. 4. 2013.] http://www.omg.org/spec/SysML/1.3/.

Příloha A – Požadavky, scénáře užití, matice pokrytí a analytické třídy OCL generátoru

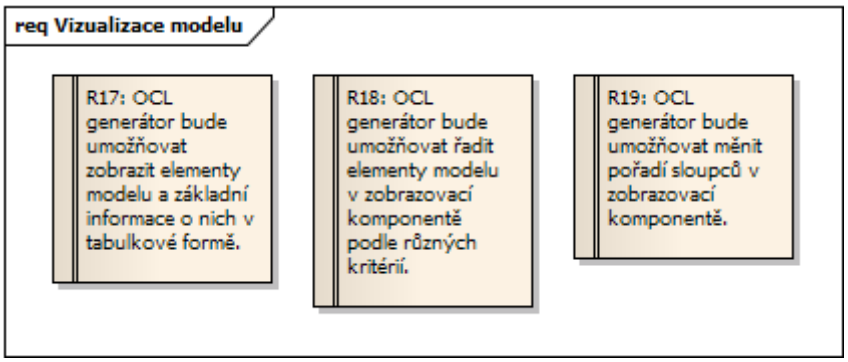
Přítomny jsou diagramy a scénáře neobsažené v hlavním textu práce. Ostatní diagramy (aktivit, sekvenční, návrhové třídy) lze zhlédnout v modelu EA na datovém médiu diplomové práce.



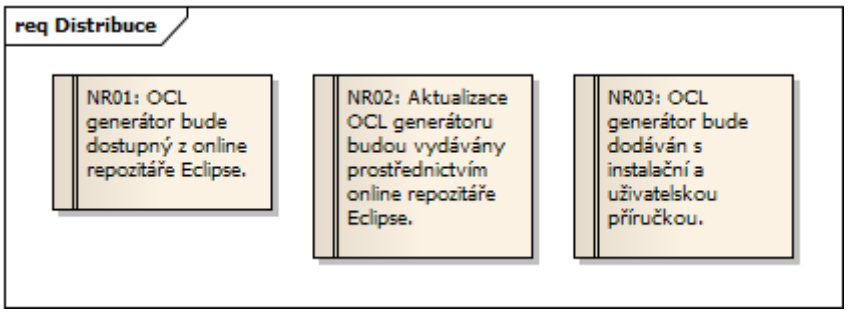
Obrázek 48 – Funkční požadavky: kategorie Generování validačního kódu. Zdroj: autor



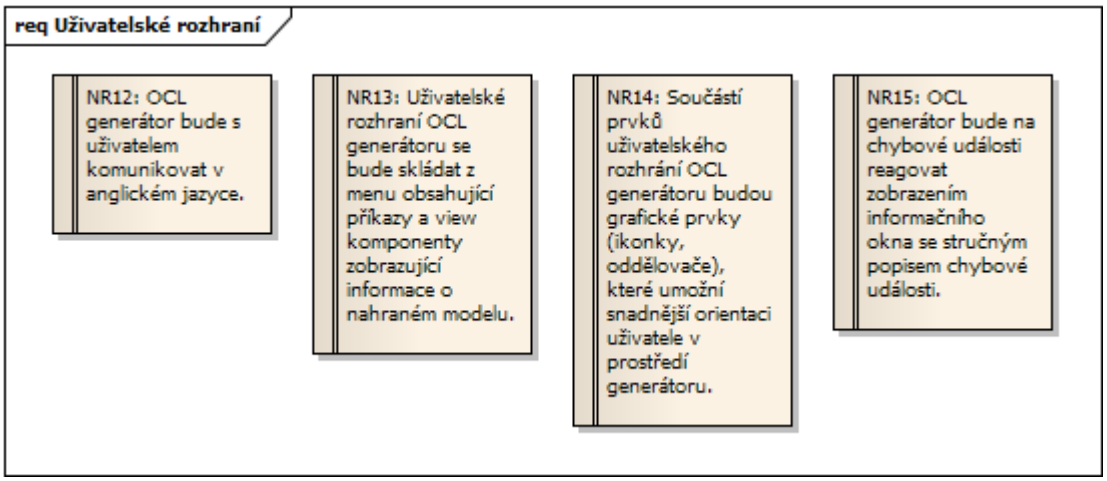
Obrázek 49 – Funkční požadavky: kategorie Validační kód. Zdroj: autor



Obrázek 50 – Funkční požadavky: kategorie Vizualizace modelu. Zdroj: autor



Obrázek 51 – Nefunkční požadavky: kategorie Distribuce. Zdroj: autor



Obrázek 52 – Nefunkční požadavky: kategorie Uživatelské rozhraní. Zdroj: autor

Tabulka 5 – Alternativní scénář Poškozený model případu užití UC01. Zdroj: autor

Případ užití: UC01: Nahrát model
Stručný popis: Lektor je informován, že vstupní model je poškozený.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.

Vstupní podmínky: 1 Lektor nahrál poškozený model.
Alternativní scénář: 1 Alternativní scénář začíná krokem 4.1 hlavního scénáře. 2 Lektor je informován, že zadaný zdrojový soubor obsahuje poškozený model.
Výstupní podmínky: Žádné.

Tabulka 6 – Alternativní scénář Nepodporovaný metamodel případu užití UC01. Zdroj: autor

Případ užití: UC01: Nahrát model
Stručný popis: Lektor je informován, že vstupní model je vytvořený v nepodporovaném metamodelu.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Lektor nahrál model vytvořený v nepodporovaném metamodelu.
Alternativní scénář: 1 Alternativní scénář začíná krokem 4.2 hlavního scénáře. 2 Lektor je informován, že metamodel vstupního modelu není podporován.
Výstupní podmínky: Žádné.

Tabulka 7 – Alternativní scénář Prázdný model případu užití UC01. Zdroj: autor

Případ užití: UC01: Nahrát model
Stručný popis: Lektor je informován, že vstupní model je prázdný.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Lektor nahrál prázdný model.
Alternativní scénář: 1 Alternativní scénář začíná krokem 4.3 hlavního scénáře. 2 Lektor je informován, že zadal prázdný model.
Výstupní podmínky: Žádné.

Tabulka 8 – Hlavní scénář případu užití UC02. Zdroj: autor

Případ užití:

UC02: Dealokovat model
Stručný popis: Nahraný model se dealokuje.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem "Unload Model" z menu. 2 Dealokuje se aktuálně nahraný model. 3 V menu se schovají příkazy pro operace s modelem. 4 Zobrazovací komponenta modelu se aktualizuje. 5 Lektor je informován o úspěšné dealokaci modelu.
Výstupní podmínky: 1 Nahraný model byl dealokován. 2 Zobrazovací komponenta byla aktualizována. 3 V menu byly schovány příkazy pro operace s modelem.

Tabulka 9 – Hlavní scénář případu užití UC03. Zdroj: autor

Případ užití: UC03: Aktualizovat model
Stručný popis: Model se nahraje ze zdrojového souboru a následně zpracuje.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem "Refresh Model" z menu. 2 Zpracuje se zdrojový soubor modelu. 2.1 Model se nahraje ze souborové cesty aktuálního modelu. 2.2 Provede se parsování zdrojového souboru modelu. 2.3 Ověří se, zda zdrojový soubor obsahuje model vytvořený v jednom z podporovaných metamodelů. 2.4 Ověří se, zda zdrojový soubor obsahuje neprázdný model. 2.5 Zjistí se informace o jednotlivých elementech modelu. 3 Model se aktualizuje. 4 Tabulková reprezentace modelu v zobrazovací komponentě se aktualizuje.
Výstupní podmínky: 1 Model byl aktualizován. 2 Tabulková reprezentace modelu v zobrazovací komponentě byla aktualizována.
Alternativní scénáře: Chybná cesta

Poškozený model Nepodporovaný metamodel Prázdný model

Tabulka 10 – Alternativní scénář Chybná cesta případu užití UC03. Zdroj: autor

Případ užití: UC03: Aktualizovat model
Stručný popis: Lektor je informován, že se zdrojový model nenachází v požadovaném umístění.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model. 2 Zdrojový model nebyl nalezen.
Alternativní scénář: 1 Alternativní scénář začíná krokem 2.1 hlavního scénáře. 2 Lektor je informován, že nelze provést aktualizaci z důvodu nenalezení zdrojového modelu.
Výstupní podmínky: Žádné.

Tabulka 11 – Alternativní scénář Poškozený model případu užití UC03. Zdroj: autor

Případ užití: UC03: Aktualizovat model
Stručný popis: Lektor je informován, že zdrojový model je poškozený.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model. 2 Zdrojový model byl poškozený.
Alternativní scénář: 1 Alternativní scénář začíná krokem 2.2 hlavního scénáře. 2 Lektor je informován, že nelze provést aktualizaci z důvodu poškozeného zdrojového modelu.
Výstupní podmínky: Žádné.

Tabulka 12 – Alternativní scénář Nepodporovaný metamodel případu užití UC03. Zdroj: autor

Případ užití:

UC03: Aktualizovat model
Stručný popis: Lektor je informován, že zdrojový model je vytvořený v nepodporovaném metamodelu.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model. 2 Zdrojový model nebyl vytvořený v podporovaném metamodelu.
Alternativní scénář: 1 Alternativní scénář začíná krokem 2.3 hlavního scénáře. 2 Lektor je informován, že nelze provést aktualizaci z důvodu nepodporovaného metamodelu.
Výstupní podmínky: Žádné.

Tabulka 13 – Alternativní scénář Prázdný model případu užití UC03. Zdroj: autor

Případ užití: UC03: Aktualizovat model
Stručný popis: Lektor je informován, že zdrojový model je prázdný.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model. 2 Zdrojový model byl prázdný.
Alternativní scénář: 1 Alternativní scénář začíná krokem 2.4 hlavního scénáře. 2 Lektor je informován, že nelze provést aktualizaci z důvodu prázdného zdrojového modelu.
Výstupní podmínky: Žádné.

Tabulka 14 – Hlavní scénář případu užití UC04. Zdroj: autor

Případ užití: UC04: Nastavit pravidla generování
Stručný popis: Pravidla pro generování se nastaví.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky:

1 Byl již nahrán model.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem "Settings" z menu. 2 Zobrazí se dialogové okno s pravidly pro generování. 3 Lektor vybere kombinaci požadovaných pravidel. 4 Lektor potvrdí výběr pravidel. 5 Pravidla generování se uloží.
Výstupní podmínky: 1 Pravidla generování byla uložena.

Tabulka 15 – Hlavní scénář případu užití UC05. Zdroj: autor

Případ užití: UC05: Přepnout příznak generování
Stručný popis: Příznak generování vybraného elementu modelu se přepne.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model. 2 Model obsahoval zobrazitelné elementy.
Hlavní scénář: 1 Případ je Lektorem spuštěn při dvojkliku na element modelu v zobrazovací komponentě nebo příkazem "Toggle Include" z kontextového menu elementu modelu. 2 POKUD vybraný element modelu není označen pro generování. 2.1 Příznak vybraného elementu je nastaven na "included". 3 JINAK 3.1 Příznak vybraného elementu je nastaven na "not included".
Výstupní podmínky: 1 Příznak vybraného elementu byl přepnut.

Tabulka 16 – Hlavní scénář případu užití UC06. Zdroj: autor

Případ užití: UC06: Vygenerovat validační kód
Stručný popis: Vygeneruje se validační kód OCL.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem "Generate OCL Constraints..." z menu. 2 Zobrazí se dialogové okno pro výběr cílového umístění souboru, do kterého bude

validační kód uložen. 3 Lektor vybere cílové umístění. 4. Načtou se pravidla generování. 5 Vygeneruje se validační kód. 5.1 PRO KAŽDÝ element modelu označený pro generování: 5.1.1 Zjistí se vztahy s ostatními elementy modelu. 5.1.1.1 PRO KAŽDÉ nastavené pravidlo generování: 5.1.1.1.1 Vygeneruje se část validačního kódu. 6 Validační kód se uloží do souboru.
Výstupní podmínky: 1 Validační kód byl vygenerován. 2 Soubor s validačním kódem byl vytvořen.

Tabulka 17 – Hlavní scénář případu užití UC07. Zdroj: autor

Případ užití: UC07: Exportovat model do XMI
Stručný popis: Provede se export nahraného modelu do formátu XMI.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem "Export to XMI" z menu. 2 Zobrazí se dialogové okno pro výběr cílového umístění modelu. 3 Lektor vybere cílové umístění. 4 Provede se konverze nahraného modelu do XMI. 5 Model se uloží na cílové umístění ve formátu XMI.
Výstupní podmínky: 1 Byl vytvořen XMI soubor obsahující model.

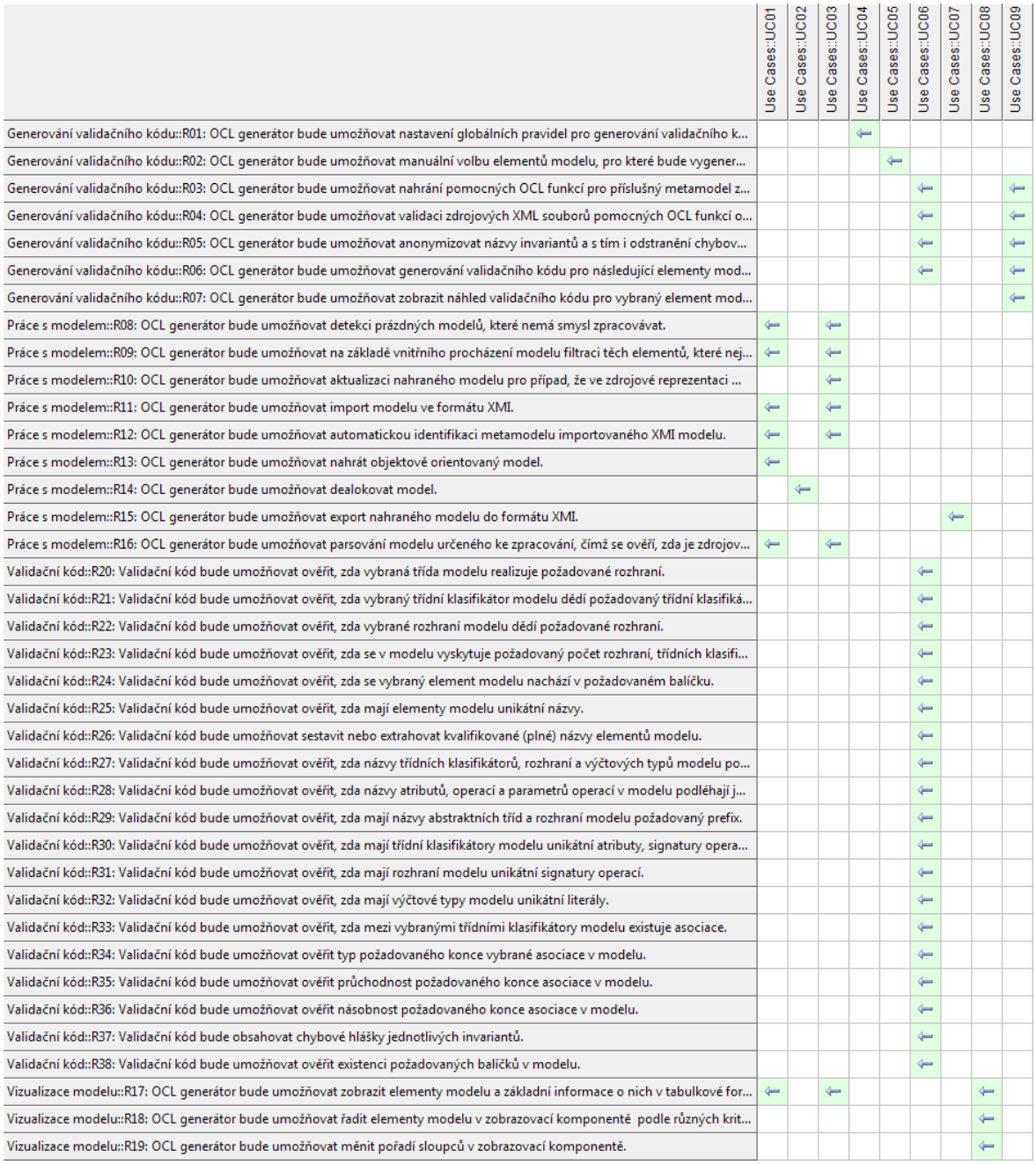
Tabulka 18 – Hlavní scénář případu užití UC08. Zdroj: autor

Případ užití: UC08: Seřadit elementy modelu
Stručný popis: Elementy modelu se seřadí podle zvoleného kritéria.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model. 2 Model obsahoval zobrazitelné elementy.
Hlavní scénář:

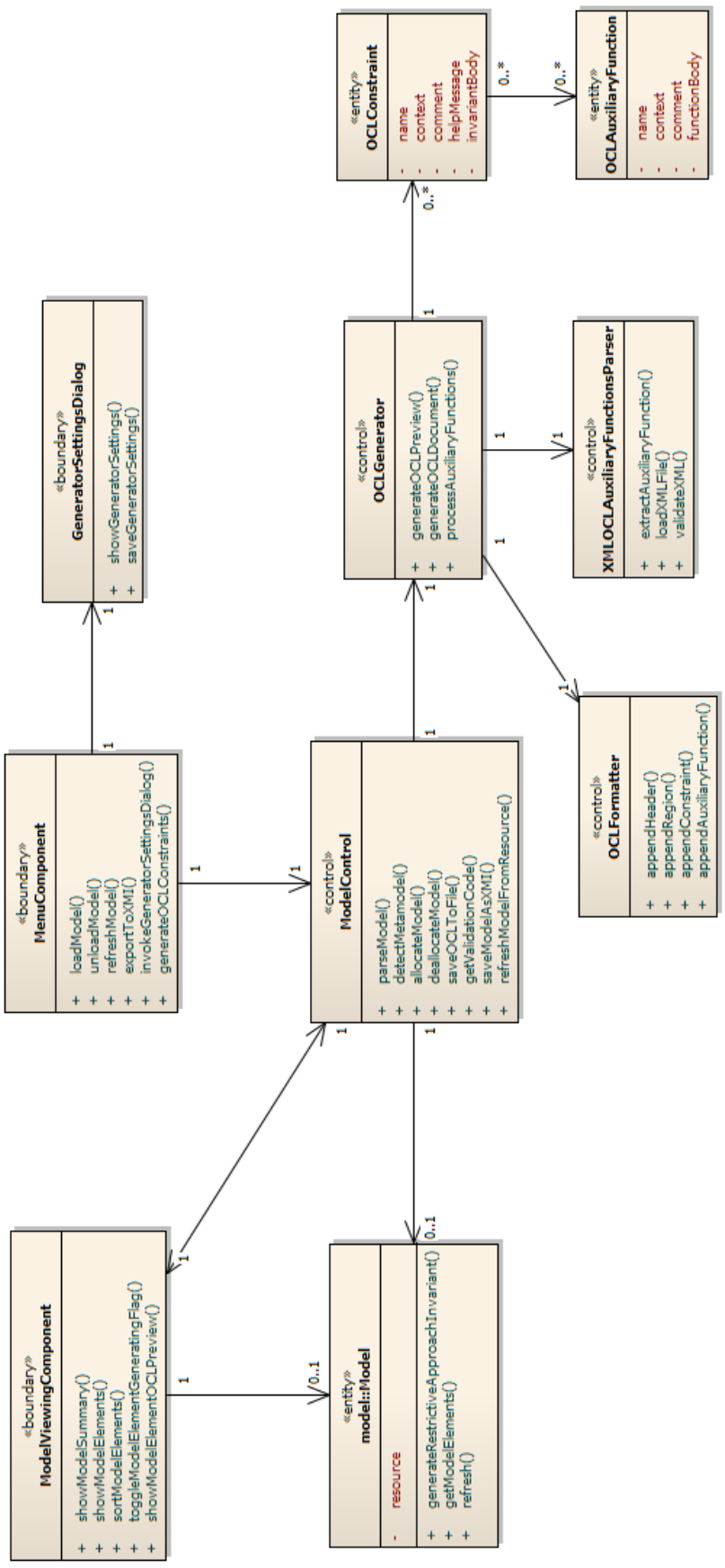
1 Případ je Lektorem spuštěn kliknutím na sloupec v zobrazovací komponentě. 2 Elementy modelu se seřadí podle kritéria sloupce vzestupně či sestupně v závislosti na výchozím stavu.
Výstupní podmínky: 1 Elementy modelu byly seřazeny.

Tabulka 19 – Hlavní scénář případu užití UC09. Zdroj: autor

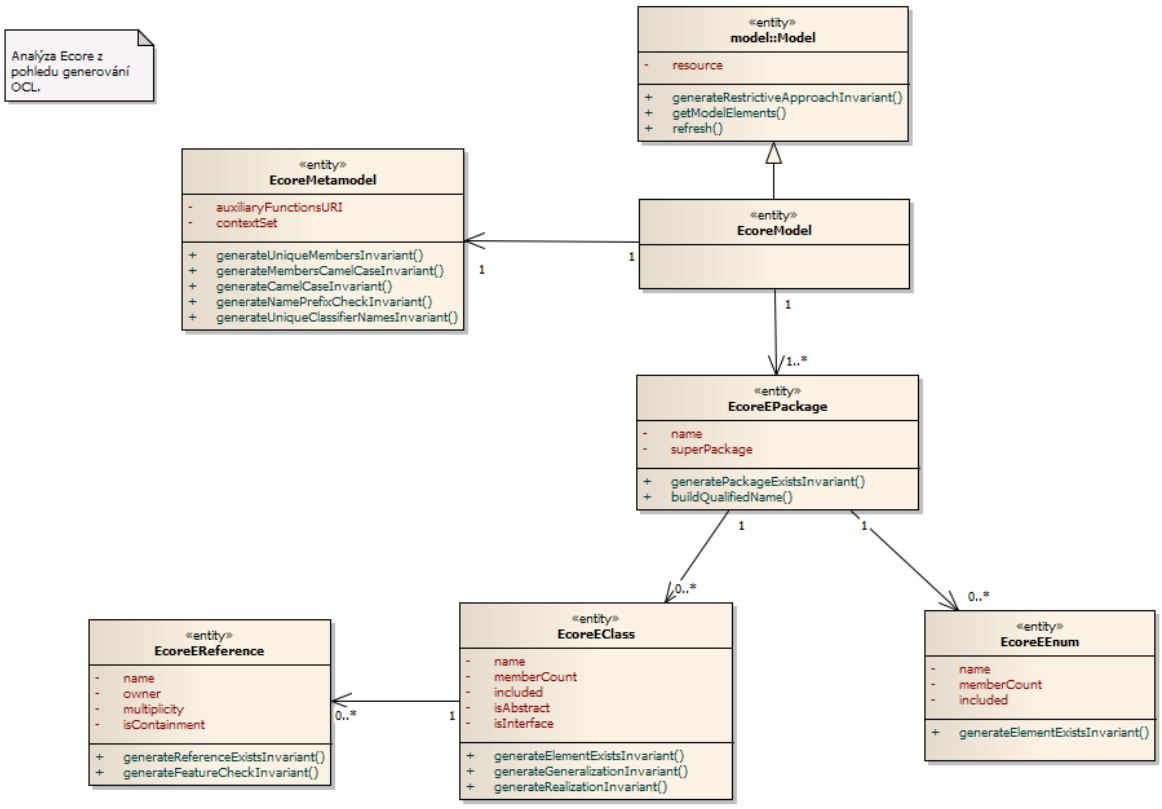
Případ užití: UC09: Zobrazit náhled validačního kódu
Stručný popis: Zobrazí se náhled validačního kódu pro vybraný element modelu.
Hlavní aktéři: Lektor.
Vedlejší aktéři: Žádní.
Vstupní podmínky: 1 Byl již nahrán model.
Hlavní scénář: 1 Případ je Lektorem spuštěn příkazem "OCL Preview" z kontextového menu elementu modelu. 2. Načtou se pravidla generování. 3 Vygeneruje se validační kód pro zvolený element modelu. 3.1 Zjistí se vztahy zvoleného elementu s ostatními elementy modelu. 3.2 PRO KAŽDÉ nastavené pravidlo generování: 3.2.1 Vygeneruje se část validačního kódu. 4 Zobrazí se okno s náhledem validačního kódu vygenerovaného pro zvolený element.
Výstupní podmínky: 1 Okno s náhledem validačního kódu bylo zobrazeno.



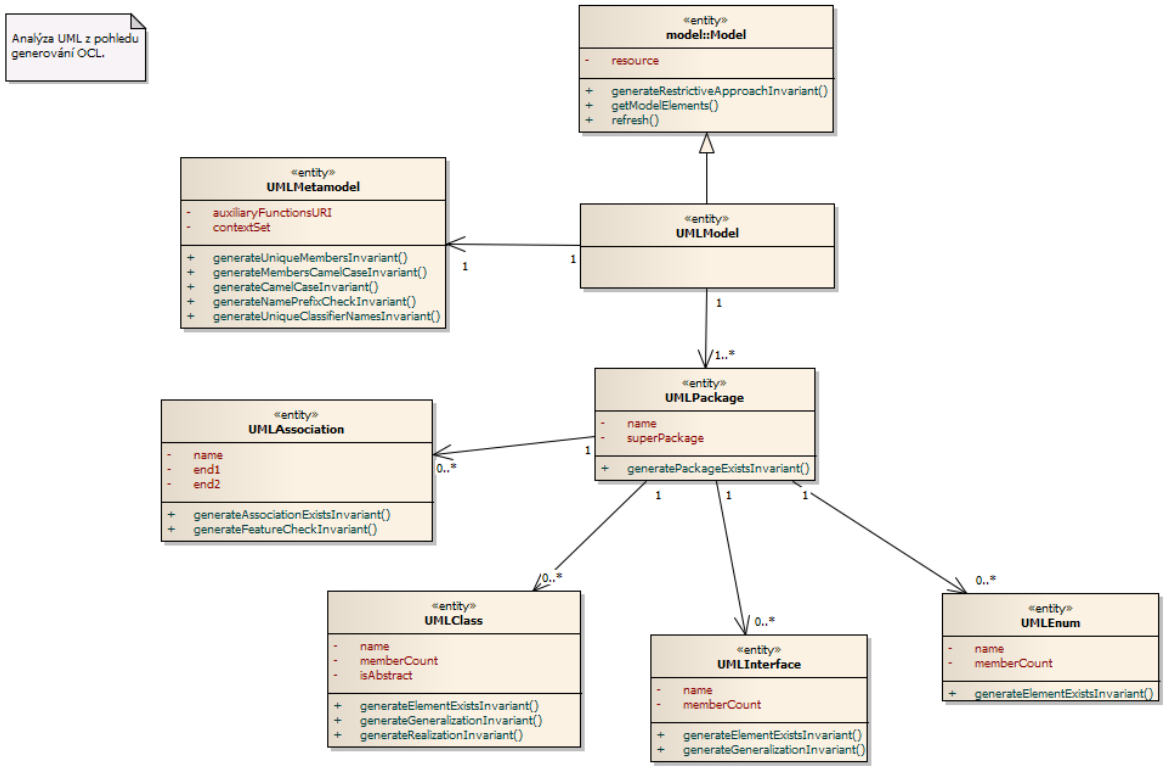
Obrázek 53 Matice pokrytí funkčních požadavků případy užití. Zdroj: autor



Obrázek 54 – Analytické třídy balíčku generating. Zdroj: autor



Obrázek 55 – Analytické třídy balíčku ecore. Zdroj: autor



Obrázek 56 – Analytické třídy balíčku uml. Zdroj: autor

Příloha B – Instalační a uživatelská příručka OCL generátoru

Obsah

1 Návod k instalaci

Tato sekce poskytuje návod pro instalaci pluginu OCL generátor do vývojové a modelovací platformy Eclipse. Součástí je také seznam Eclipse nástrojů, které je doporučeno pro práci s pluginem využívat.

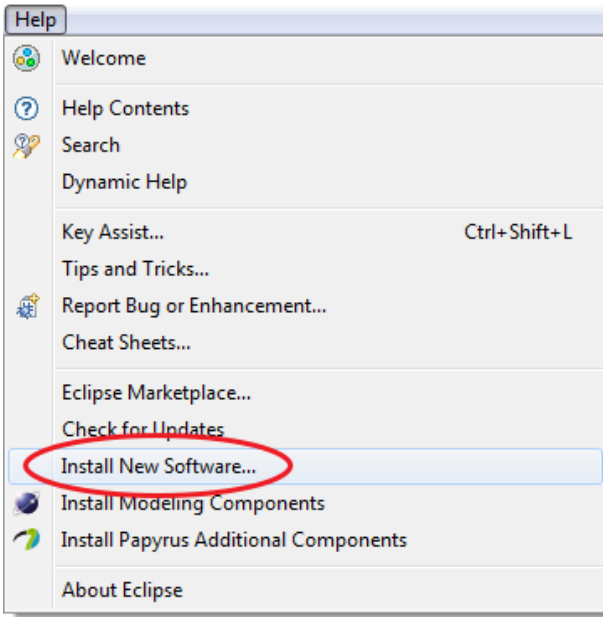
Autor diplomové práce nabízí předpřipravený balíček Eclipse Juno 4.2 pro Windows 64b, ve kterém jsou již OCL generátor a všechny níže uvedené nástroje pro práci s ním nainstalovány. Balíček se nachází na datovém médiu diplomové práce.

1.1 Instalace pluginů do Eclipse

Dle dostupnosti vybraného pluginu lze využít následující typy instalací.

P2 Update Manager

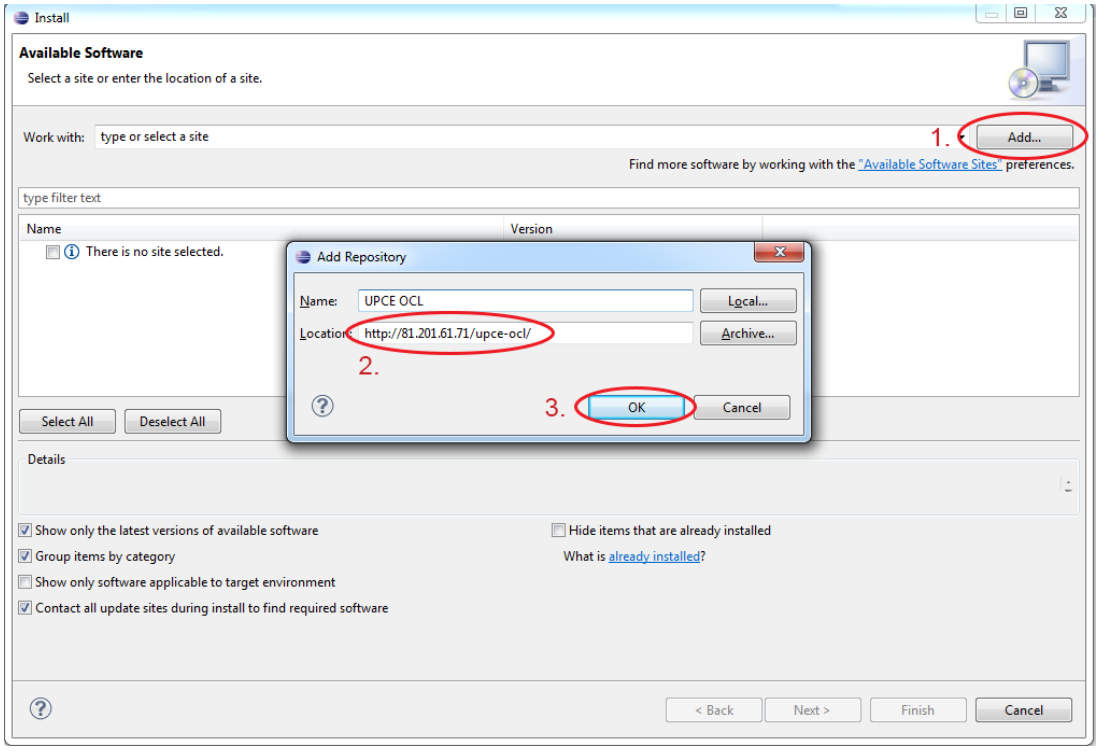
1) V hlavním menu Eclipse (viz obrázek 57) zvolíme záložku *Help* a poté příkaz *Install New Software...*.



Obrázek 57 – Instalace pluginů do Eclipse přes P2 Update Manager, krok 1. Zdroj: autor

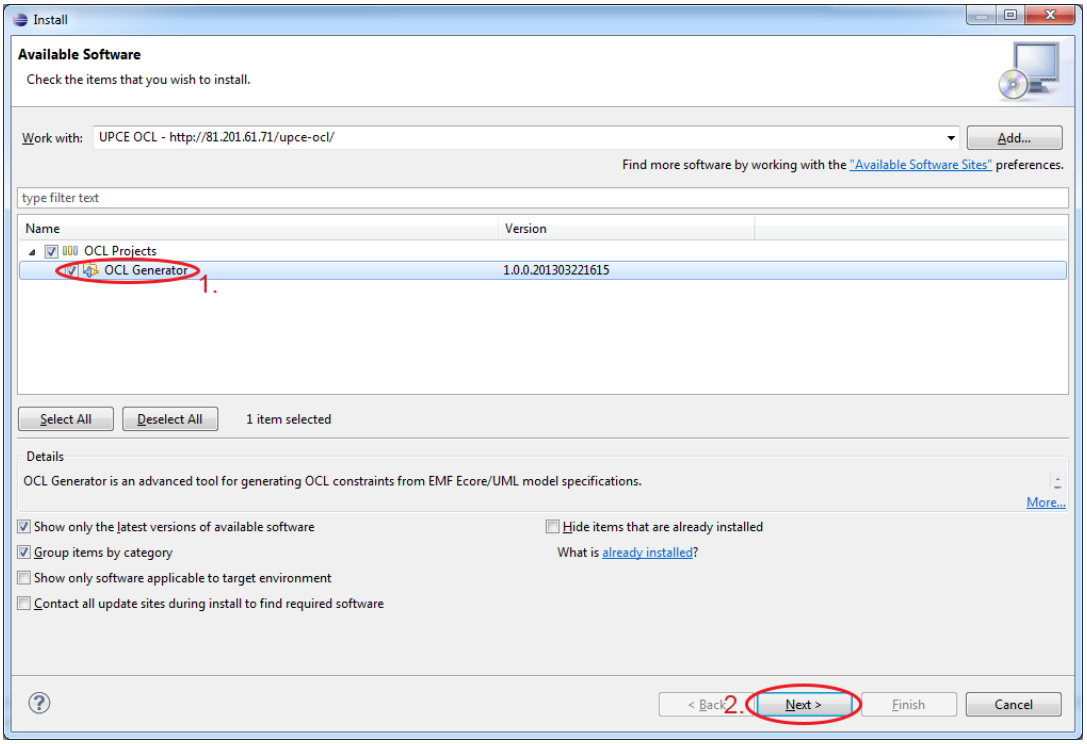
2) V zobrazeném okně (viz obrázek 58) je neprvé nutné přidat repozitář se seznamem nabízených pluginů. Pokud již repozitář přidaný máte, vyhledejte a potvrďte jej v roletkovém menu *Work with*. V opačném případě nejprve klikněte na tlačítko *Add...* (1.), v dialogovém okně vyplňte URL repozitáře (2.) a potvrďte tlačítkem *OK* (3.).

Pokud je repozitář přístupný na lokálním disku, lze jej vybrat kliknutím na tlačítko *Local...* a zvolením přístupové cesty. Pokud máte k dispozici stažený archiv ve formátu jar či zip, lze jej nahrát přes tlačítko *Archive...*.



Obrázek 58 – Instalace pluginů do Eclipse přes P2 Update Manager, krok 2. Zdroj: autor

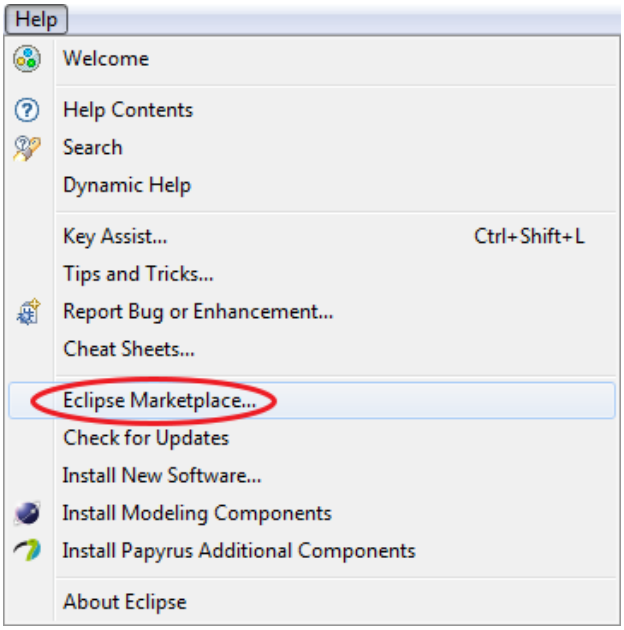
3) Ze seznamu pluginů (viz obrázek 59) vyberte požadované položky (1.) a klikněte na tlačítko *Next* (2.). Proběhne vyhodnocení prerekvizit a závislostí. Dále se řiďte jednoduchými pokyny instalačních průvodců pro jednotlivé pluginy. Chybějící součásti jsou před samotnou instalací vyhledány a staženy dle dostupných repozitářů.



Obrázek 59 – Instalace pluginů do Eclipse přes P2 Update Manager, krok 3. Zdroj: autor

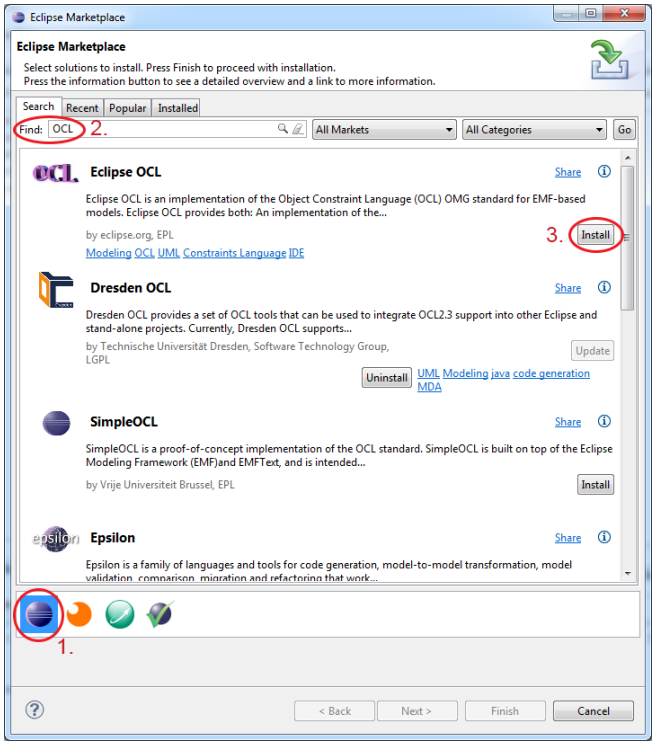
Eclipse Marketplace

1) V hlavním menu Eclipse (viz obrázek 60) zvolíme záložku *Help* a poté příkaz *Eclipse Marketplace...*.



Obrázek 60 – Instalace pluginů do Eclipse přes Eclipse Marketplace, krok 1. Zdroj: autor

2) V zobrazeném okně (viz obrázek 61) nejprve zvolte požadovaný marketplace (1.). Zadejte název pluginu a nechte vyhledat (2.). Spusťe instalaci zvoleného pluginu kliknutím na tlačítko *Install*. Dále se řiďte jednoduchými pokyny instalačního průvodce.



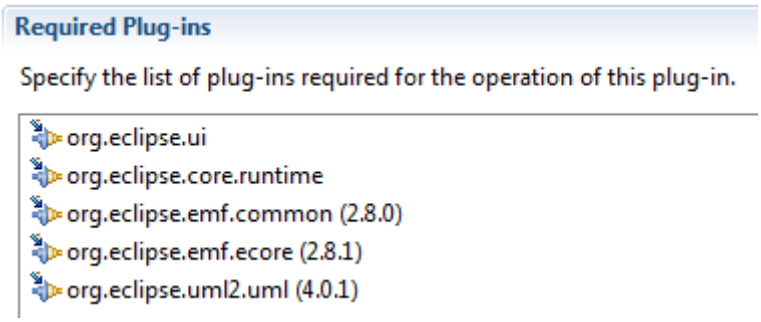
Obrázek 61 – Instalace pluginů do Eclipse přes Eclipse Marketplace. Zdroj: autor

Manuální instalace

Nechcete-li využít žádnou z předcházejících možností, je zde ještě varianta manuální instalace. Binární soubory pluginu společně se všemi závislostmi vložte do adresáře *plugins*, který se nachází v kořenovém adresáři prostředí Eclipse. Poté je Eclipse nutné restartovat.

1.2 Prerekvizity

Hlavními požadavky pro OCL generátor jsou platforma **Eclipse 4** a vyšší s rozšířením **Eclipse Modeling Framework Project (EMF)** a běhové prostředí **JRE 7**. Doporučuje se využít připraveného balíčku Eclipse Modeling Tools, který lze stáhnout z http://www.eclipse.org/downloads/.



Obrázek 62 – Závislosti pluginu OCL generátor. Zdroj: autor

Dále je vyžadována **EMF implementace metamodelu UML**, která je součástí Model Development Tools (MDT). Dostupná je na oficiálních webových stránkách projektu http://www.eclipse.org/modeling/mdt/downloads/?project=uml2.

1.3 Instalace OCL generátoru

OCL generátor se instaluje přes P2 Update Manager z repozitáře na adrese http://81.201.61.71/upce-ocl/. Veškeré chybějící prerekvizity jsou staženy automaticky společně s ním. Není nutné nic manuálně stahovat ani instalovat. Ze stejného repozitáře také Eclipse stahuje vydané aktualizace pro tento plugin.

1.4 Komplementární nástroje

Vhodným doplňkem OCL generátoru jsou tyto nástroje a rozšíření:

- MDT OCL – implementace OCL pro Eclipse.
  http://www.eclipse.org/modeling/mdt/downloads/?project=ocl
- Ecore Tools – grafický editor modelů v Ecore.
  http://www.eclipse.org/modeling/emft/downloads/?project=ecoretools

- Papyrus[20] – grafický editor modelů v UML. http://www.eclipse.org/papyrus/downloads/index.php
- Dresden OCL – nástroj pro práci s OCL v UML modelech. Dostupné z Eclipse Marketplace. Návod na http://marketplace.eclipse.org/marketplace-client-intro.

[20] Nezaměňovat se stejnojmenným projektem na http://www.papyrusuml.org, který je delší dobu neaktualizovaný. Jedná se o původní projekt, ze kterého nový Papyrus vychází.

2 Uživatelský průvodce

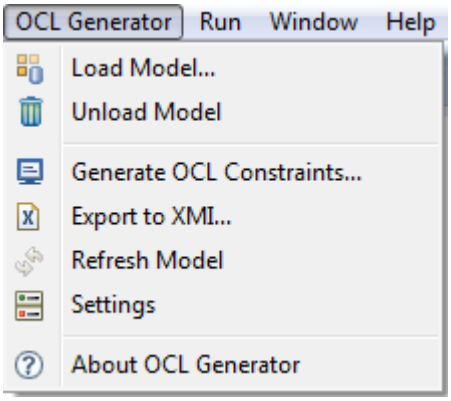
Tato sekce popisuje uživatelské rozhraní nástroje OCL generátor. Vysvětluje také, jak s nástrojem pracovat v součinnosti s validačními komponenty Eclipse EMF.

2.1 Uživatelské rozhraní

Uživatelské rozhraní OCL generátoru tvoří dvě části: hlavní menu a zobrazovací komponenta modelu.

Hlavní menu

Po nainstalování nástroje se do hlavního menu Eclipse přidá záložka *OCL Generator* (viz obrázek 63). Pro zpřístupnění všech příkazů nástroje je nejprve třeba nahrát EMF Ecore/UML model.



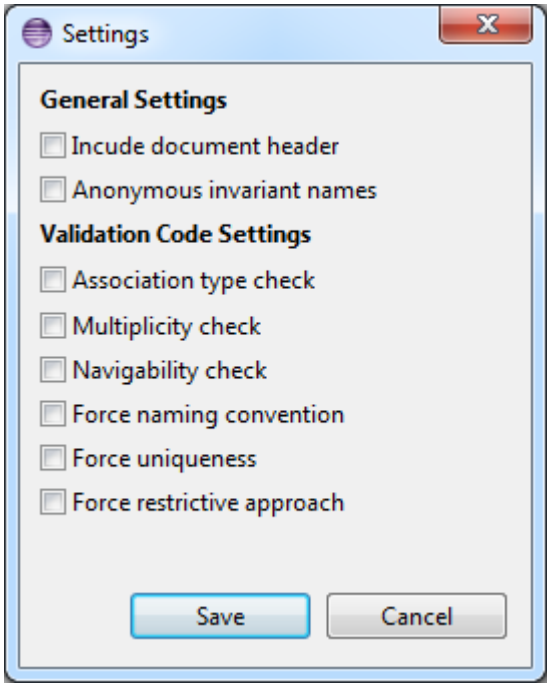
Obrázek 63 – Hlavní menu OCL generátoru po nahrání modelu. Zdroj: autor

- *Load Model...* – otevře dialogové okno pro výběr zdrojového souboru modelu, který má být nahrán. Při výběru souboru s koncovkou xmi je provedena automatická detekce typu modelu.
- *Unload Model* – dealokuje nahraný model.
- *Generate OCL Constraints...* – otevře dialogové okno pro výběr cílového validačního souboru. Poté dojde k vygenerování a uložení validačního kódu do souboru.
- *Export to XMI...* – otevře dialogové okno pro výběr cílového souboru, do kterého má být exportovaný model uložen. Následně je proveden XMI export nahraného modelu a jeho uložení do souboru. Tato funkcionalita je přípravou na možné rozšíření pluginu, které bude umožňovat editaci modelu.
- *Refresh Model...* – aktualizuje nahraný model z jeho původního zdroje.
- *Settings* – zpřístupní nastavení pro generování validačního kódu.
- *About OCL Generator* – zobrazí stručné informace o aplikaci.

Nastavení generování validačního kódu

Pravidla pro generování validačního kódu lze určit přes dialog nastavení (viz obrázek 64), který se vyvolá příkazem *Settings* v hlavním menu. Pravidla zde neuvedená (kontrola

existence elementů, generalizace, realizace, …) jsou vynucována automaticky a nelze je vypnout.



Obrázek 64 – Nastavení pravidel pro generování validačního kódu. Zdroj: autor

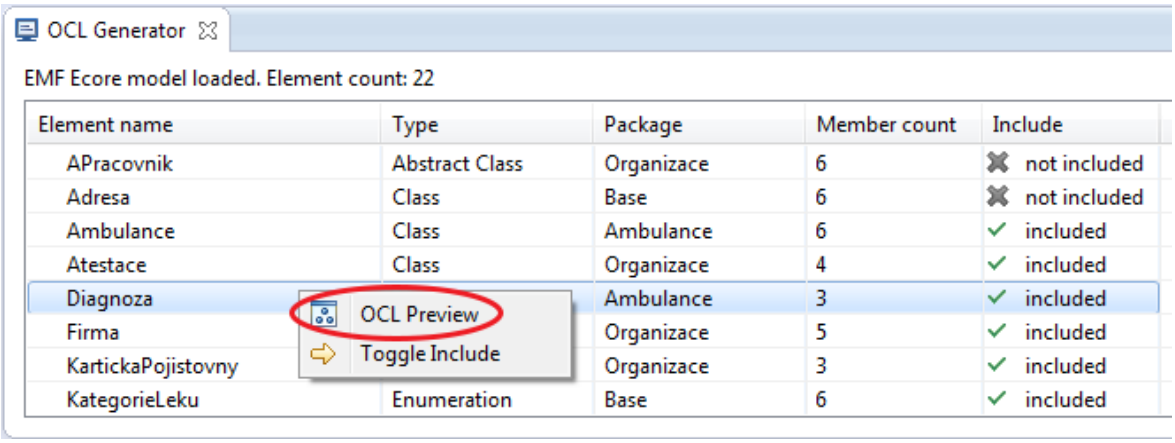
- *General Settings*
  - *Include document header* – součástí validačního souboru bude hlavička s informacemi o průběhu generování.
  - *Anonymous invariant names* – vygenerovaná omezení budou mít anoymní názvy a hlášky ve formátu `Invariant no. <pořadovéČíslo> violated!`
- *Validation Code Settings*
  - *Association type check* – validační kód bude ověřovat typy konců asociací.
  - *Multiplicity check* – validační kód bude ověřovat násobnost konců asociací.
  - *Navigability check* – validační kód bude ověřovat průchodnost konců asociací.
  - *Force naming convention* – validační kód bude vynucovat jmennou konvenci *CamelCase* a prefixy názvů klasifikátorů podle jejich typu. Tato volba není vhodná pro úlohy, ve kterých mají studenti za úkol vytvářet předem definované klasifikátory.
  - *Force uniqueness* – validační kód bude vynucovat unikátnost členů a názvů.
  - *Force restrictive approach* – validační kód bude vynucovat přesné počty jednotlivých typů elementů v modelu.

Zobrazovací komponenta modelu

Zobrazovací komponenta modelu (viz obrázek 65) se po nainstalování OCL generátoru obvykle objeví v dolní části prostředí Eclipse v závislosti na používané perspektivě. Instalace rozšíří kategorie pohledů o položku *OCL Generator*. Zavřete-li tuto komponentu

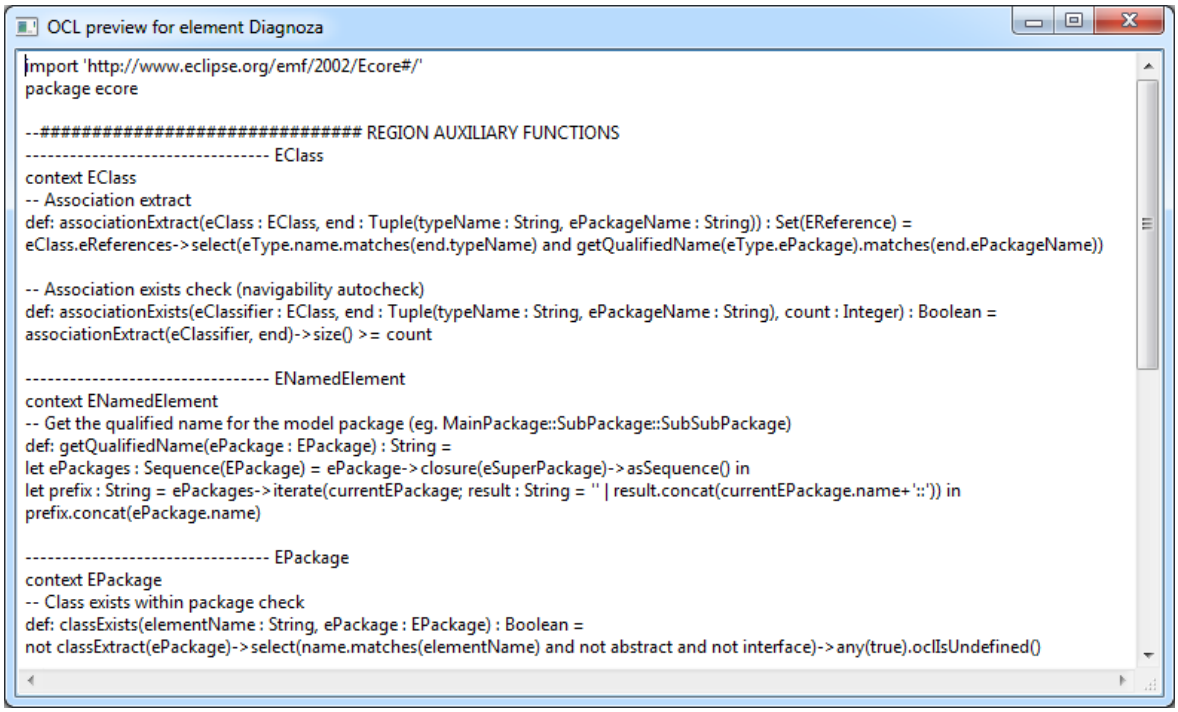
a chcete-li ji znovu zobrazit, lze tak učinit navigováním z hlavního menu Eclipse přes *Window*, *Show View*, *Other* a v zobrazeném okně vyhledáním kategorie *OCL Generator*, ve které je zobrazovací komponenta modelu umístěna.

Zobrazovací komponenta poskytuje informace o elementech modelu. Elementy lze řadit dle příslušných sloupců. Umožňuje také označit ty elementy modelu, pro které se validační kód nebude generovat (pozor, neplést s odstraněním elementu z modelu).



Obrázek 65 – Zobrazovací komponenta modelu. Zdroj: autor

Možné je zobrazit náhled validačního kódu pro vybraný element příkazem *OCL Preview* z kontextového menu a ověřit si tak dopad nastavení pravidel na jeho generování (viz obrázek 66).



Obrázek 66 – Náhled validačního kódu. Zdroj: autor

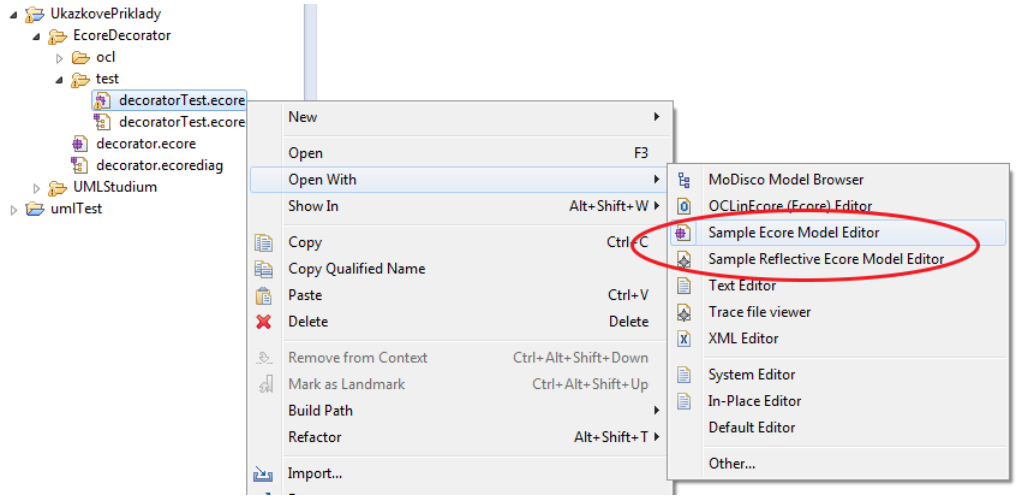
2.2 Validace modelu

Vygenerovaný validační soubor lze používat pro validaci vypracovaných úloh studentů (studenti jej také mohou obdržet jako pomůcku pro tvorbu) či v případě obecných omezení pro libovolný jiný model.

EMF Ecore model

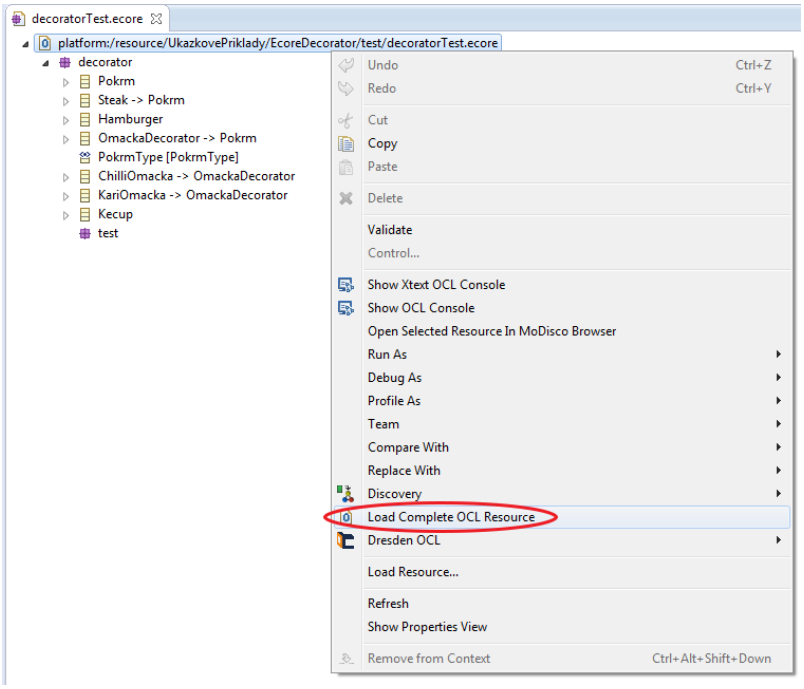
Validaci EMF Ecore modelu se doporučuje provádět prostřednictvím stromového editoru Sample (Reflective) Ecore Model Editor. Využít lze i jiné nástroje, které umožňují přidružení validačního souboru k modelu, jako např. Dresden OCL.

1) Otevřete požadovaný EMF Ecore model ve výše zmiňovaném editoru. Pokud jej nemáte nastavený jako výchozí, orientujte se podle následujícího obrázku (lze použít libovolný z vyznačené dvojice na obrázku 67).



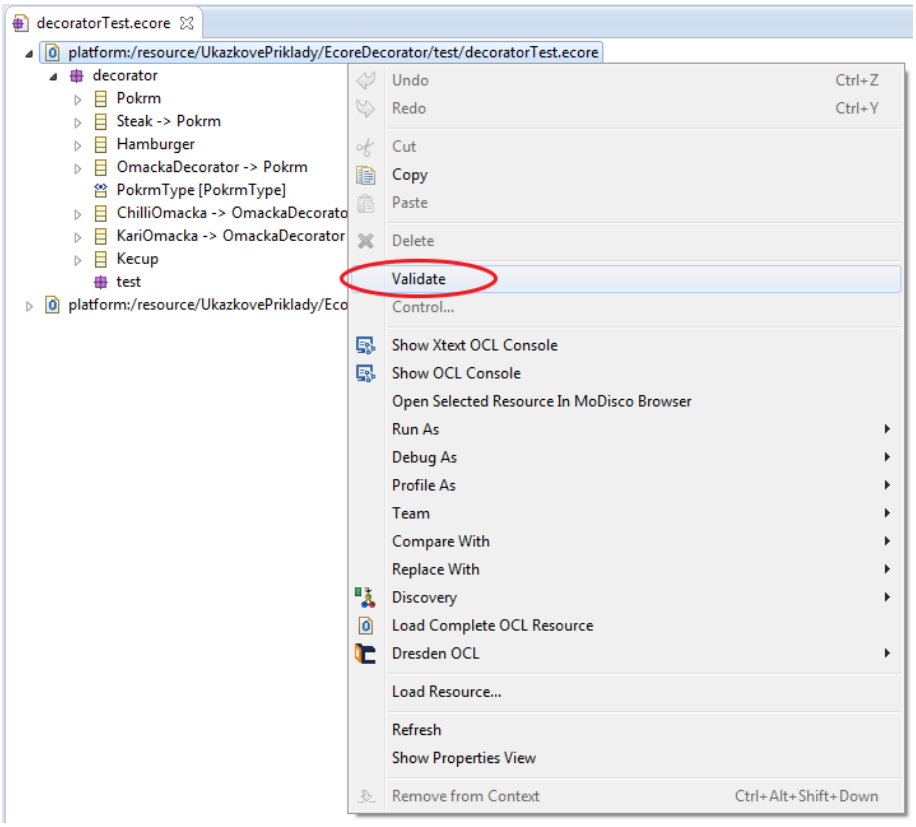
Obrázek 67 – Validace EMF Ecore modelu, krok 1. Zdroj: autor

2) Vyvolejte kontextové menu odkudkoli z prostoru editoru a klikněte na příkaz *Load Complete OCL Resource* (viz obrázek 68). Objeví se dialogové okno pro výběr validačního souboru. Validační soubor lze přetáhnout přímo z projektové části Eclipse.



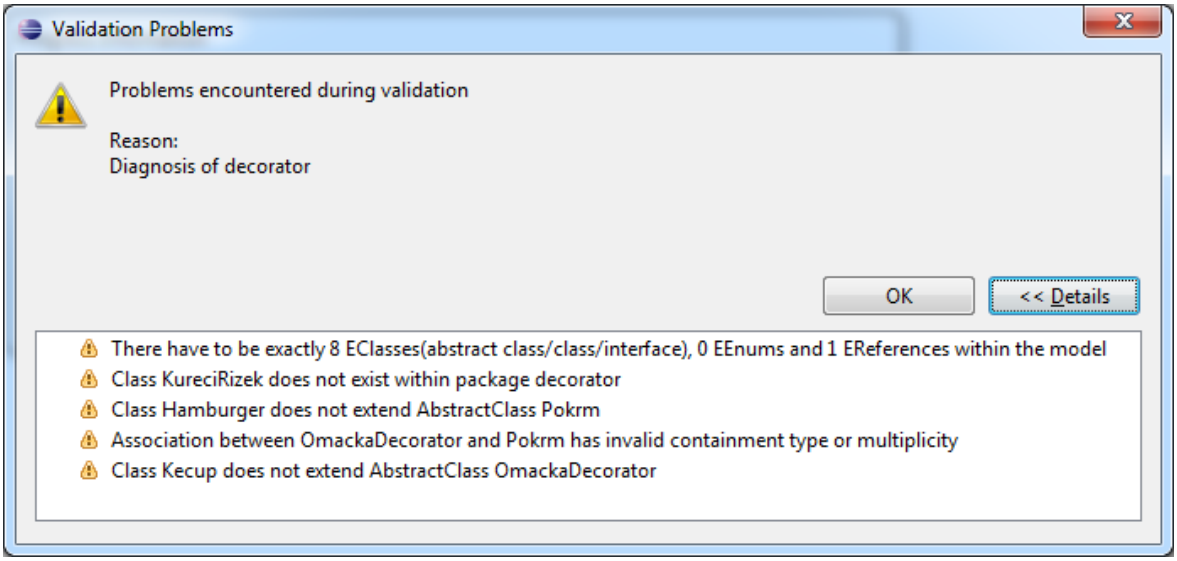
Obrázek 68 – Validace EMF Ecore modelu, krok 2. Zdroj: autor

3) Po přidružení validační souboru vyvolejte kontextové menu ze zdrojové položky modelu (začíná na *platform:/resource...*) nebo hlavního balíčku modelu. Klikněte na příkaz *Validate* (viz obrázek 69).



Obrázek 69 – Validace EMF Ecore modelu, krok 3. Zdroj: autor

4) Zobrazí se okno s výsledkem validace (viz obrázek 70). Kliknutím na tlačítko *Details* zobrazíte chybové hlášky v případě, že model neprošel validací.



Obrázek 70 – Validace EMF Ecore modelu, krok 4. Zdroj: autor

Pro průběžnou validaci se doporučuje mít model, na kterém se zrovna pracuje v grafickém modelovacím editoru, zároveň otevřený i v uvedeném stromovém editoru a validovat podle potřeby (validační soubor není nutné pokaždé nahrávat znovu). V budoucnu se má tato forma validace integrovat přímo do vybraných grafických modelovacích editorů, což učiní celý proces komfortnějším.

EMF UML model

Postup validace EMF UML modelu je analogický k předchozímu. Stromovým editorem je v tomto případě UML Model Editor. Některé grafické modelovací editory, jako např. Papyrus, navíc zobrazí v diagramu výstražné trojúhelníky u těch elementů, které způsobily selhání validace.

U validace UML modelů je nutné upozornit na bug v Eclipse EMF, který nepozorovaně znefunkční validaci v případě, že je validační procedura uskutečněna před otevřením modelu v příslušném grafickém editoru (model se uživateli vyhodnotí vždy jako validní). Jediným známým řešením je zatím restart Eclipse. Je tedy vždy nutné nejprve otevřít model v grafickém editoru (je-li to vyžadováno) a teprve poté nahrát validační soubor přes stromový editor.

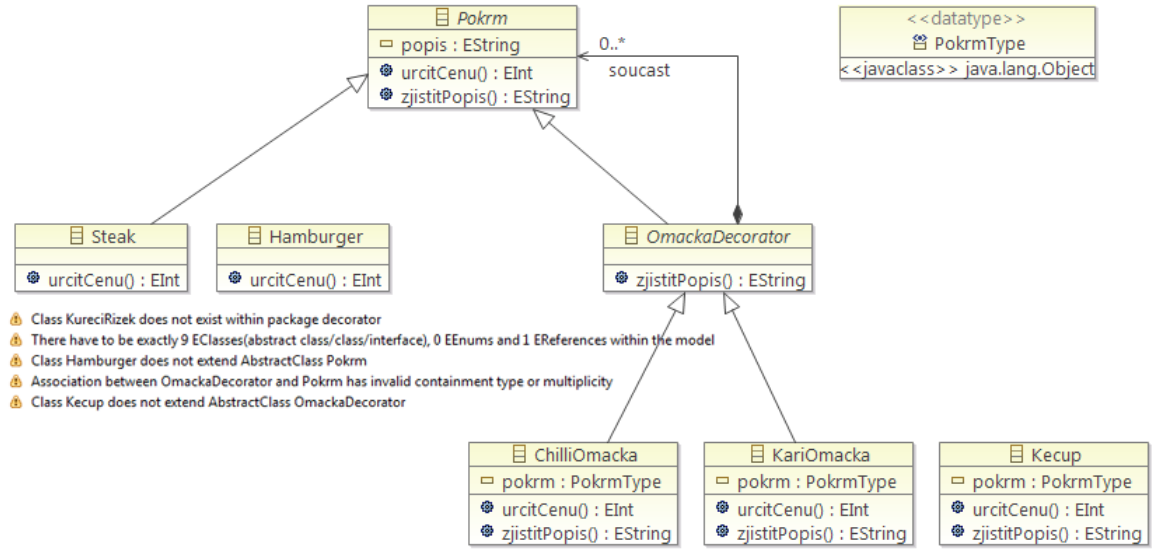
Příloha C – Ukázkové úlohy v EMF Ecore/UML

Tato příloha obsahuje dvojici ukázkových úloh pro metamodely Ecore a UML, které slouží jako námět užití OCL generátoru pro účely výuky objektově orientovaného modelování. Veškeré podklady (modely, diagramy, validační soubory) se nacházejí na datovém médiu diplomové práce.

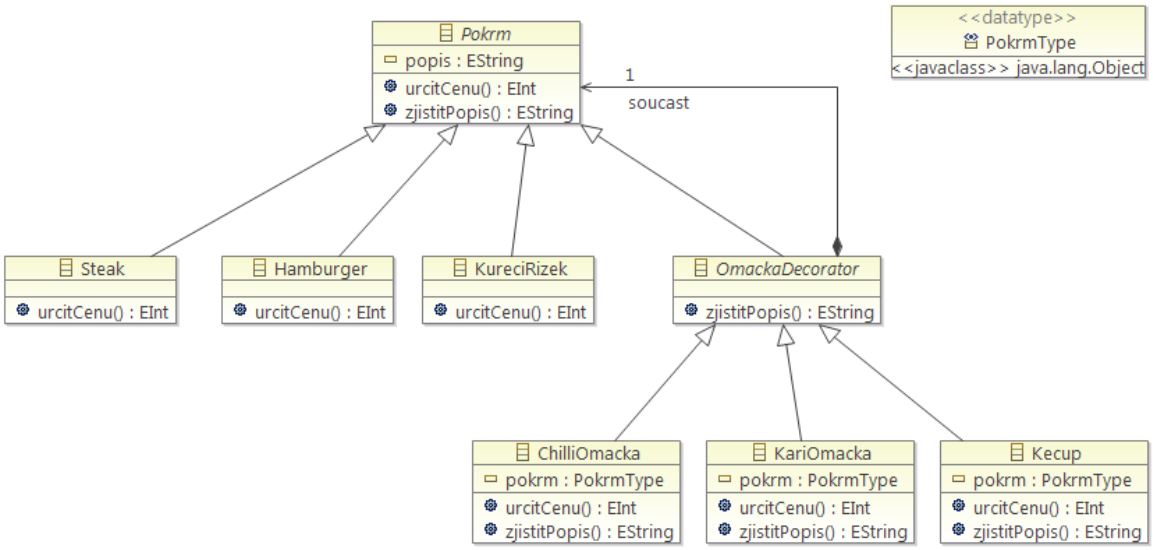
Návrhový vzor dekorátor v Ecore

Zadání: vytvořte diagram tříd návrhového vzoru dekorátor pro pokrmy a omáčky v metamodelu Ecore. Použijte následující klasifikátory: *Hamburger*, *ChilliOmacka*, *KariOmacka*, *Kecup*, *KureciRizek*, *OmackaDecorator*, *Pokrm*, *Steak*. Veškeré elementy vytvářejte v balíčku *decorator*. Správnost výsledného modelu ověřte obdrženým validačním souborem prostřednictvím validátoru v Eclipse EMF.

Pro tvorbu diagramu se doporučuje využít grafického modelovacího nástroje Ecore Tools.



Obrázek 71 – Rozpracovaná úloha na návrhový vzor dekorátor po pokusu o validaci. Zdroj: autor



Obrázek 72 – Validní podoba úlohy na návrhový vzor dekorátor. Zdroj: autor

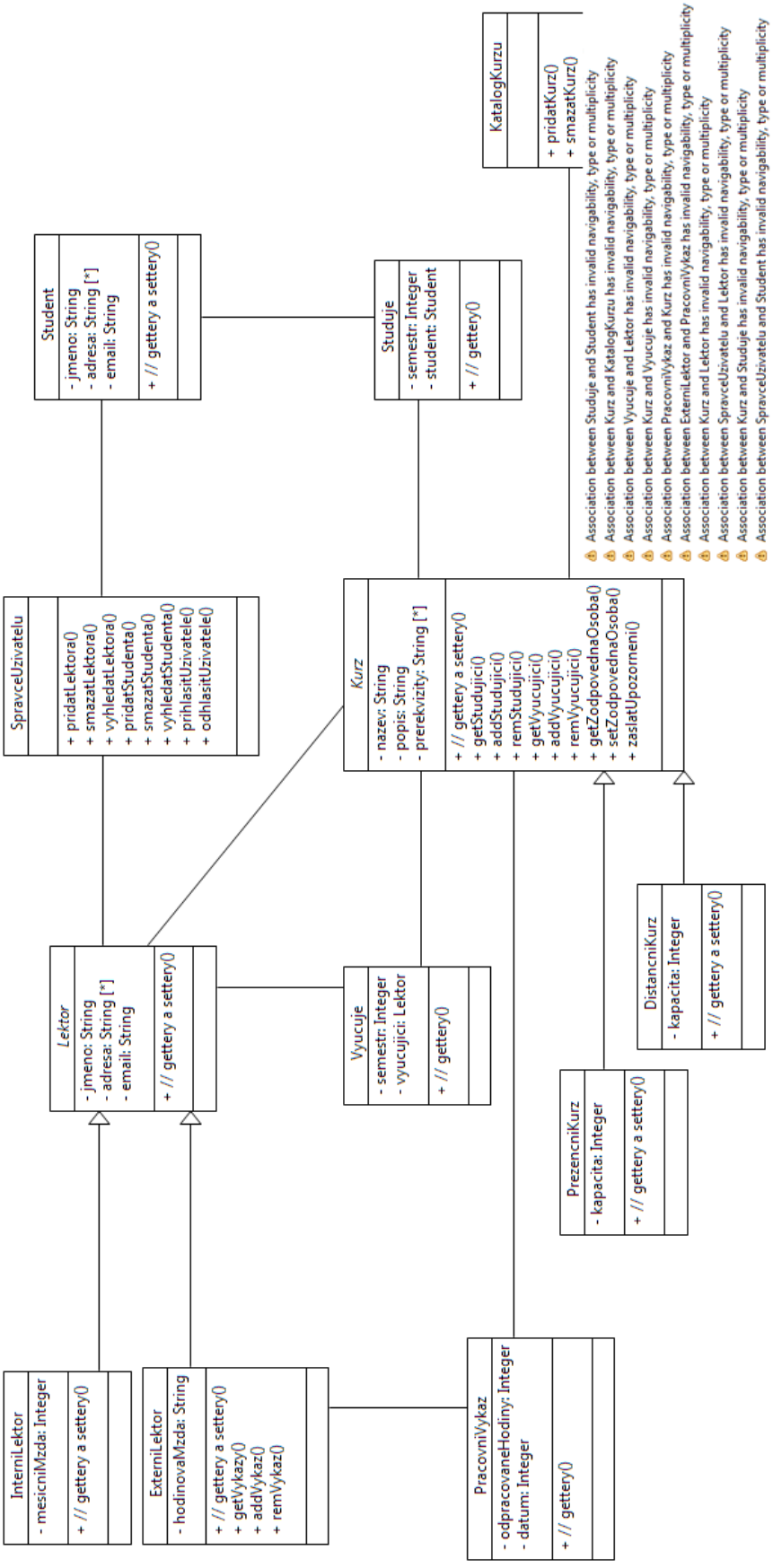
Informační systém Studium v UML

Zadání: určete vhodné typy, násobnosti a průchodnosti konců asociací v diagramu tříd informačního systému Studium vytvořeného v metamodelu UML. Správnost výsledného modelu ověřte obdrženým validačním souborem prostřednictvím validátoru v Eclipse EMF.

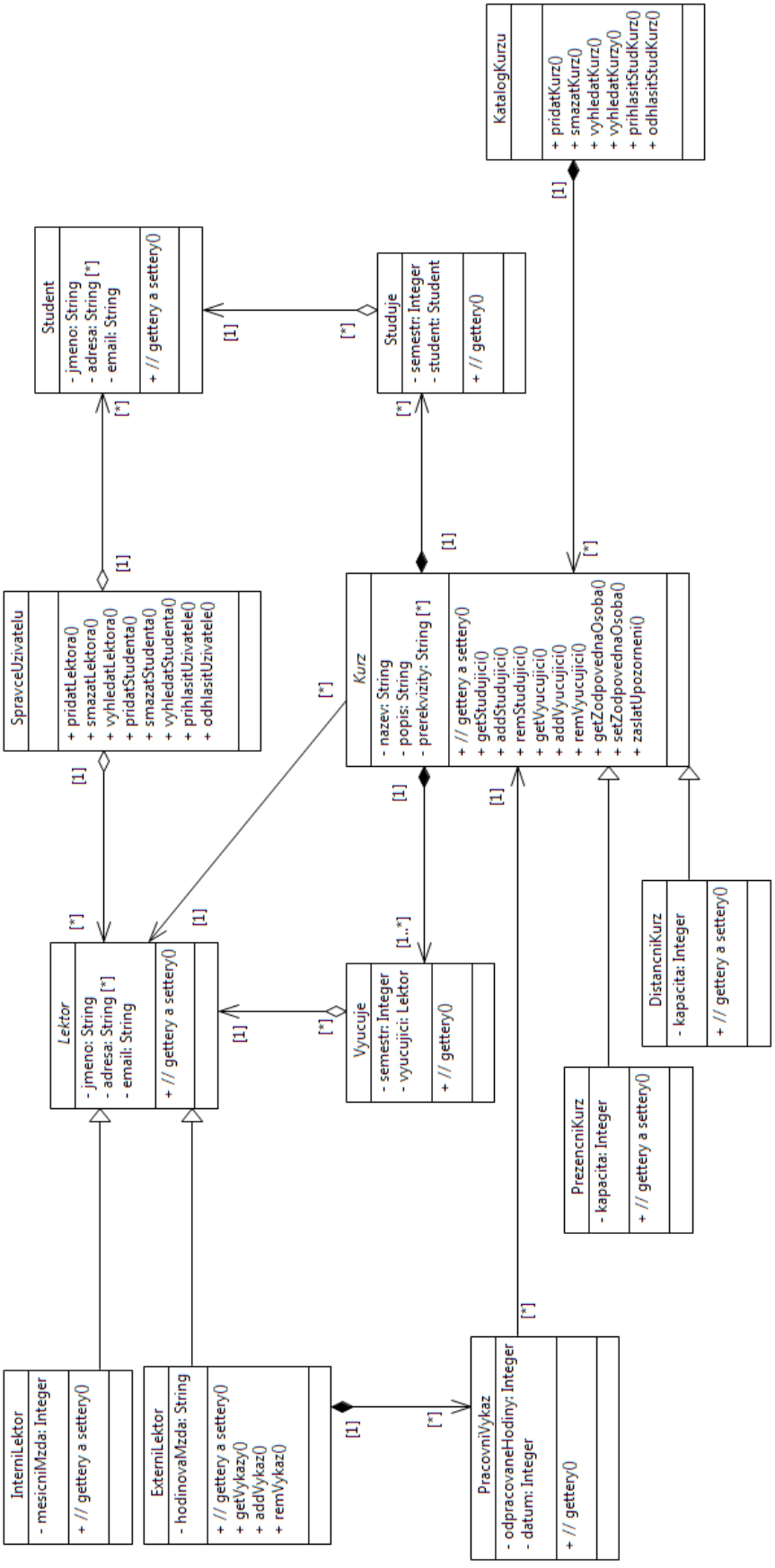
Výchozí model informačního systému se doporučuje editovat prostřednictvím grafického modelovacího nástroje Papyrus.

Popis IS Studium: Informační systém Studium bude sloužit pro podporu správy kurzů včetně elektronického přihlašování a odhlašování. Správa studijích kurzů umožní lektorům přidávání nových kurzů, mazání kurzů a úpravu stávajících parametrů, kterými jsou název, popis, prerekvizity, kapacita, nastavení rozvrhu a rozlišení, zda jde o prezenční nebo distanční kurz. Studenti dostanou možnost využít systém k elektronickému zápisu do kurzů a odhlašování z kurzů. Zápis do kurzu bude umožněn jen v případě, že dosud není naplněna kapacita kurzu. Pokud dojde k vyčerpání kapacity kurzu, systém pošle upozornění na email lektora zodpovědného za kurz. Jednou z doplňkových funkcí systému bude podpora pro vkládání pracovních výkazů o odpracovaných hodinách pro externí lektory, kteří na rozdíl od interních lektorů nepobírají měsíční mzdu a jsou placeni na základě odpracovaných hodin.

Úloha byla vytvořena na základě ilustračního příkladu z webové stránky Ing. RNDr. Barbory Bühnové, Ph.D http://www.fi.muni.cz/~buhnova/PV167/priklad.html.



Obrázek 73 – Výchozí stav úlohy IS Studium po pokusu o validaci. Zdroj: autor



Obrázek 74 – Validní podoba úlohy IS Studium. Zdroj: autor

Příloha D – Ukázka zdrojového kódu OCL generátoru

```
@Override
public String generateOCLDocument(EnumSet<OCLFlags> flags)
		throws OCLGeneratorException {
	// Nothing to generate from
	if (model.getModelElements().size() == 0)
		return null;

	StringBuilder functionBuilder = new StringBuilder();
	StringBuilder constraintBuilder = new StringBuilder();

	// XML auxiliary functions parser init
	String URI = model.getAuxiliaryFunctionsURI();
	IXMLOCLAuxiliaryFunctionsParser parser;
	try {
		parser = new XMLOCLAuxiliaryFunctionsParser(URI);
	} catch (ParserConfigurationException | SAXException | IOException e) {
		throw new OCLGeneratorException(e.getMessage());
	}

	// Anonymous invariants?
	boolean anonymous = flags.contains(OCLFlags.ANONYMOUS_INVARIANTS);

	// Generate the constraint sets for various context levels
	List<IOCLConstraint> modelConstraints = processElements(
			model.getContextSet(), flags);

	// Add metamodel constraints
	modelConstraints.addAll(model.getMetamodelConstraints(flags));

	// Restrictive approach active?
	if (flags.contains(OCLFlags.RESTRICTIVE_APPROACH))
		modelConstraints.add(model.generateRestrictiveApproachInvariant());

	// Required auxiliary functions "bit field"
	EnumSet<OCLAuxiliaryFunctionType> requiredAuxFunctions =
processConstraints(
			constraintBuilder, modelConstraints, anonymous);

	// Auxiliary functions
	processAuxiliaryFunctions(functionBuilder, requiredAuxFunctions, parser);

	// Clear the preview model element
	previewModelElement = null;

	// Include header?
	String header = "";
	if (flags.contains(OCLFlags.INCLUDE_HEADER)) {
		header = OCLFormatter.appendHeader(processedElements,
			functionCount, constraintCount, flags);
	}

	// Form the ouput
	String output = header + model.getImportToken() + "\n"
			+ OCLFormatter.getPackageStart(model.getPackageName()) +
"\n\n"
			+ functionBuilder.toString() + constraintBuilder.toString()
```

```
                  + OCLFormatter.getPackageEnd();

      return output;
}

private List<IOCLConstraint> processElements(
            EnumSet<OCLContext> contextSet, EnumSet<OCLFlags> flags) {

      List<IOCLConstraint> modelConstraints = new ArrayList<IOCLConstraint>();
      for (IModelElement modelElement : model.getModelElements()) {
            // Preview?
            if (previewModelElement != null &&
!modelElement.equals(previewModelElement))
                  continue;

            // Ignore non included elements
            if (!modelElement.isIncluded() &&
!modelElement.equals(previewModelElement))
                  continue;

            List<IOCLConstraint> constraints = modelElement
                        .generateOCLConstraints(flags);

            // No constraints generated
            if (constraints == null)
                  continue;

            for (IOCLConstraint constraint : constraints) {
                  if (constraint != null
                              &&
contextSet.contains(constraint.getContext()))
                        modelConstraints.add(constraint);
            }
            processedElements++;
      }

      return modelConstraints;
}

private EnumSet<OCLAuxiliaryFunctionType> processConstraints(
            StringBuilder constraintBuilder,
            List<IOCLConstraint> modelConstraints, boolean anonymous) {
      // Sort the constraints by their context
      Collections.sort(modelConstraints, new OCLContextComparator());

      EnumSet<OCLAuxiliaryFunctionType> requiredAuxFunctions = EnumSet
                  .noneOf(OCLAuxiliaryFunctionType.class);

      // Current context
      OCLContext currentContext = null;

      OCLFormatter.appendRegion(constraintBuilder, "Constraints");
      int counter = 0;
      for (IOCLConstraint constraint : modelConstraints) {
            if (anonymous)
                  counter++;

            // Context header
```

```
            if (currentContext == null
                        || (currentContext != null && !currentContext
                                    .equals(constraint.getContext()))) {

                  currentContext = constraint.getContext();
                  OCLFormatter.appendContextHeader(constraintBuilder,
                              currentContext);
            }
            requiredAuxFunctions.addAll(constraint
                        .getRequiredAuxiliaryFunctions());
            OCLFormatter.appendConstraint(constraintBuilder, constraint,
                        counter);
            constraintCount++;
      }

      return requiredAuxFunctions;
}
```

Příloha E – Adresářová struktura přiloženého datového média

- **PraktickaCast** – praktická část diplomové práce.
  - **EclipsePackage** – předpřipravený balíček Eclipse s nainstalovaným pluginem OCL generátor a komplementárními nástroji.
  - **oclgen**
    - **cz.upce.oclgen** – Eclipse projekt OCL generátoru.
    - **cz.upce.oclgen.feature** – komponenta online repozitáře OCL generátoru.
    - **cz.upce.oclgen.updatesite** – online repozitář OCL generátoru.
    - **Dokumentace**
      - **EAModelExport** – vyexportovaná dokumentace OCL generátoru z EA ve formátu HTML (nefunguje v Chrome).
      - **JavadocExport** – vyexportovaná Javadoc dokumentace ze zdrojového kódu OCL generátoru ve formátu HTML.
    - **Model** – EA model OCL generátoru.
  - **Poliklinika** – EMF Ecore model z kapitoly 5.
  - **UkazkoveUlohy** – ukázkové úlohy z přílohy C.
- **TextovaCast** – textová část diplomové práce ve formátu PDF.