Vysoká škola báňská – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Softwarová podpora pro konfigurace podnikových procesů

Software Support for Business Processes Reconfiguration

2015 **Ondřej Trlifaj**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Ondřej Trlifaj**

Studijní program: B2647 Informačni a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Softwarová podpora pro konfigurace podnikových procesů
Software Support for Business Processes Reconfiguration

Zásady pro vypracování:

Cílem práce je analyzovat, navrhnout a implementovat nástroj, který nad existujícím, či nově vzniklým rámcem umožní modelovat a konfigurovat podnikové procesy.

1. Student nastuduje podnikové procesy.
2. Student nastuduje existující aplikační rámce pro podporu modelování a rekonfigurace procesů.
3. Student nad vybraným frameworkem vytvoří grafickou modelovací nadstavbu a implementuje nástroj pro konfiguraci existujícího procesu na základě grafických modelů.
4. Student na základě návrhu podnikového procesu namodeluje ukázkové procesy a demonstruje možnosti konfigurace.

Seznam doporučené odborné literatury:

[1] AALST, Will van der. The Application of Petri Nets to Workflow Management. [online]. s. 53 [cit. 2012-07-26]. Dostupné z: http://wwwis.win.tue.nl/~wvdaalst/publications/p53.pdf
[2] AALST, Will van der. Workflow Patterns. [online]. s. 68 [cit. 2012-07-26]. Dostupné z: http://www.workflowpatterns.com/documentation/documents/wfs-pat-2002.pdf
[3] AALST, Will van der. Verification of Workflow Nets. [online]. [cit. 2012-07-26]. Dostupné z: http://wwwis.win.tue.nl/~wvdaalst/publications/p44
[4] JENSEN, Kurt a Lars Michael KRISTENSEN. Coloured Petri Nets: modelling and validation of concurrent systems. Dordrecht: Springer, c2009, xi, 384 s. ISBN 978-3-642-00283-0.
[5] VONDRÁK, Ivo. METODY BYZNYS MODELOVÁNÍ: pro kombinované a distanční studium. [online]. [cit. 2012-07-26]. Dostupné z: http://vondrak.cs.vsb.cz/download/Metody_byznys_modelovani.pdf
[6] VONDRÁK, Ivo. Úvod do softwarového inženýrství. [online]. [cit.2012-07-26]. Dostupné z: http://vondrak.cs.vsb.cz/download /Uvod_do_softwaroveho_inzenyrstvi.pdf
[7] VONDRÁK, Ivo. Neuronové sítě. [online]. [cit. 2012-07-26]. Dostupné z: http://vondrak.cs.vsb.cz/download/Neuronove_site.pdf

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Michael Alexander Košinár**

Datum zadání: 01.09.2014

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: *17. července 2015*

……..……
podpis studenta

Poděkování

Velmi rád bych poděkoval Ing. Michaelu Alexanderu Košinárovi za odbornou pomoc, vstřícný a motivující přístup a konzultace při vytváření této bakalářské práce.

Abstrakt

Cílem práce je analyzovat, navrhnout a implementovat nástroj, který na platformě .NET umožní modelovat a konfigurovat existující softwarové procesy, konkrétně behaviorální perspektivu softwarového procesu pomocí aktivitního diagramu UML. Samotný softwarový proces je zachycen v metamodelu, který vychází ze stávajícího výzkumu na Fakultě elektrotechniky a informatiky při VŠB-TUO. V teoretické části jsou zmíněny podstatné oblasti, ze kterých má práce vychází - softwarový proces, XML, OWL, metamodely SP a aktivitní diagram UML.

V praktické části je práce zaměřena na implementaci nástroje, který umožňuje graficky modelovat behaviorální perspektivu existujícího softwarového procesu pomocí notace aktivitního diagramu UML v prostředí .NET WPF. Vstupem je OWL/XML znalostní báze obsahující konkrétní model procesu vytvořený v programu Protége 5.0. Je popsáno řešení pomocí knihovny pro vykreslování diagramů v .NET WPF GoXam od společnosti Northwoods Software a také popsána práce s OWL API jakožto rozhraním pro získávání informací z OWL/XML formátu.

Klíčová slova

GoXam, OWL API, XML, softwarový proces, UML, diagram aktivit

Abstract

The goal of this work is an analysis, design and implementation of a software product for .NET platform for modeling and reconfiguration of existing software processes, namely behavioral perspective of the software process using UML activity diagram. Software process itself is represented in a metamodel based on existing research of Faculty of Electrical Engineering and Computer Science, VŠB-TUO. Theoretical part covers essential theoretical concepts of software process metamodels, UML activity diagrams, and XML. The practical part is focused on implementation of a software product that allows users to graphically model the behavioral perspective of a particular software process using UML activity diagram notation. Implementation platform is .NET, namely .NET WPF. Input for this software product is an OWL/XML knowledge base containing concrete software process model created with Protége 5.0. A brief description of Northwoods Software library for diagram drawing in .NET WPF ‘GoXam’ and an OWL/XML data mining interface ‘OWL API’ by the University of Manchester is also included.

Key words

GoXam, OWL API, XML, Software Process, UML, Activity Diagram

Seznam použitých symbolů a zkratek

Zkratka	Význam
API	Application Programming Interface
ARIS	Architecture of Information Systems
AD	Diagram aktivit
BPMN	Business Process Modeling Notation
CMM	Capability Maturity Model
EMF	Eclipse Modeling Framework
EPC	Event-driven Process Chain
GMF	Graphical Modeling Framework
IDEF	Integration Definition
IRI	Internationalized Resource Identifier
LINQ	Language Integrated Query
OWL	Ontology Web Language
SEI	Software Engineering Institute
RUP	Rational Unified Process
SP	Softwarový proces
SPEM	Software Process Engineering Meta-Model
UML	Unified Modeling Language
UPMM	Unified Process Meta-Model
W3C	World Wide Web Consortium
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language
XML	Extensible Markup Language

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Každá společnost zabývající se vývojem softwaru se snaží vyvíjet co nejkvalitnější software a to pokud možno při předvídatelné ceně a v naplánovaném čase. Tohoto cíle dosahuje tak, že se veškerá práce organizuje do podoby softwarového procesu. Každá společnost zabývající se vývojem softwarových produktů nějaký softwarový proces implementuje a tento proces je buď zachycen a zmapován, nebo není.

Softwarový proces lze zachytit do modelu pomocí formální nebo neformální specifikace. Neformální specifikace je pro člověka přirozená a ne příliš složitá např. pomocí přirozené řeči nebo obrázků, zatímco formální specifikace využívá metod s jasnou syntaxí a sémantikou a tato specifikace je tedy korektní pro strojové zpracování, ovšem klade vysoké nároky na člověka, který tento softwarový proces modeluje. Na pomezí formálních a neformálních metod nalezneme semi-formální metody, které mají striktně definovanou syntaxi, ale sémantiku lze chápat různě. Mezi tyto metody můžeme zahrnout grafický jazyk UML, který je kompromisem mezi čitelností pro člověka a zpracovatelností pro stroj – diagramy zachycené v UML se dají převést do programového kódu.

Na softwarový proces se můžeme dívat z několika perspektiv – v této práci se zaměřím na vizuální modelování behaviorální perspektivy softwarového procesu, protože zachycuje jádro chování firmy. Jako modelovací jazyk jsem zvolil UML, jehož aktivitní diagram je ideální pro zachycení této perspektivy.

1.1 Cíl práce

Cílem této práce je analyzovat, navrhnout a implementovat nástroj, který na platformě .NET umožní konfigurovat existující podnikové (konkrétně softwarové) procesy, primárně behaviorální perspektivu softwarového procesu pomocí aktivitního diagramu UML. Samotný model softwarového procesu je zachycen pomocí meta-modelu, který vychází ze stávajícího výzkumu na Fakultě elektrotechniky a informatiky při VŠB-TUO.

Praktická implementace spočívá v oddělení vrstvy modelu softwarového procesu od konkrétní modelovací notace. Hlavním důvodem pro oddělení těchto vrstev je snaha o vytvoření nástroje, který pracuje s procesy 3. úrovně (Definovaný proces) na stupnici CMM. Jako vstup pro nástroj slouží soubor obsahující OWL/XML znalostní bázi se znovupoužitelnými prvky konkrétního procesu a tento nástroj si pomocí striktně definovaných mapovacích pravidel tuto bázi zpracuje a přiřadí jednotlivá individua z modelu ke konkrétním prvkům aktivitního diagramu. Dále také kontroluje vztahy mezi těmito individui a definuje pravidla pro mapování těchto vztahů na elementy notace diagramu aktivit. Na základě tohoto mapování a OWL/XML báze softwarového procesu lze graficky modelovat procesy.

1.2 Obsah kapitol

Ve druhé kapitole se zabývám seznámením čtenáře se základními teoretickými oblastmi, na kterých tato práce staví, nejprve softwarovým procesem, následuje modelování softwarového procesu a metamodel softwarového procesu, dále je popsán diagram aktivit jazyka UML, následovaný XML a OWL jakožto jazyka postaveného na XML, který je užit pro specifikaci základní formální vrstvy softwarového procesu

Třetí kapitola je věnována praktické implementaci nástroje pro platformu .NET 4.5, konkrétně s využitím technologie .NET WPF, pomocí knihoven GoXam a OWL API a jejich použití na této platformě. Nejprve jsou popsány důležité části a principy těchto knihoven a posléze popsáno konkrétní řešení.

2 Teoretická část

Tato kapitola pokrývá úvod do problematiky softwarového procesu, metamodelů softwarového procesu, jazyka UML a jeho aktivitního diagramu, XML a OWL.

2.1 Softwarový proces

Definice: Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla. (1)

Podle stupnice CMM rozlišujeme pět úrovní vyspělosti softwarového procesu:

- *Počáteční* - softwarový proces není definován a každý projekt se řeší ad-hoc.
- *Opakovatelný* - v jednotlivých projektech byly nalezeny opakovatelné postupy a firma je schopna je použít v dalších projektech.
- *Definovaný* - proces je zdokumentován a definován na základě dříve identifikovatelných opakovatelných kroků.
- *Řízený* – přináší oproti definovanému možnost řízení a monitorování výkonu a kvality.
- *Optimalizovaný* – přidává k řízené úrovni možnost optimalizace na základě dlouhodobého průběžného monitorovacího procesu.

V celém textu vycházím z předpokladu, že softwarový proces je speciálním případem byznys procesu. (1) Existuje mnoho typů softwarových procesů a použití jednotlivých typů procesů se liší podle konkrétních požadavků na softwarový produkt. V následujících podkapitolách ve zkratce popíši několik z nich.

2.1.1 Vodopádový model

Vodopádový model rozděluje proces vývoje softwarového produktu na čtyři fáze: *analýza a specifikace požadavků*, *návrh softwarového systému, implementace* a *testování a údržba*. Princip je takový, že každá konkrétní fáze může začít až tehdy, když byla kompletně dokončena fáze předchozí. Výsledek závisí na korektní specifikaci požadavků na začátku - změny požadavků v pozdějších fázích jsou velmi pracné a nákladné. Výsledný produkt je možno vidět až po proběhnutí všech fází. Tento model je prakticky nepoužitelný pro projekty velkého rozsahu.

2.1.2 Iterativní model

Iterativní model odstraňuje nedostatky vodopádového modelu – proces vývoje se skládá z několika iterací a během každé iterace jsou zahrnuty všechny čtyři fáze z vodopádového modelu. V každé iteraci je zpracována vybraná část požadavků a na konci iterace je výsledkem spustitelný kód. Výsledek jedné iterace je vstupem iterace další.

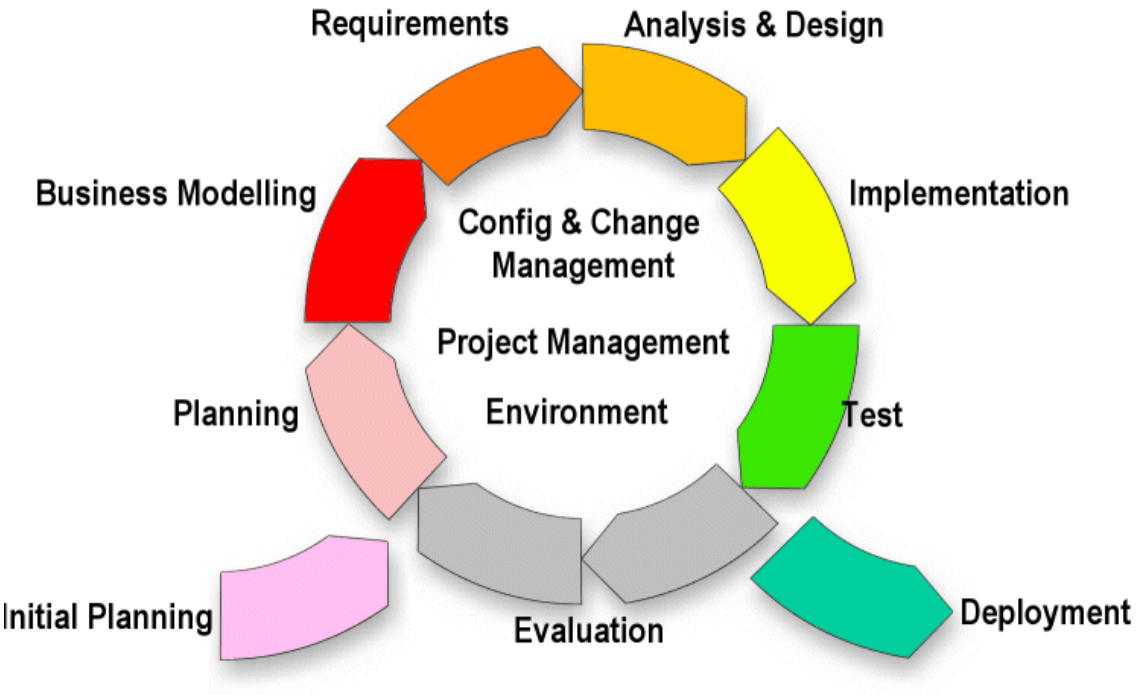
2.1.3 Rational Unified Process

RUP definuje disciplinovaný přístup k přiřazování úkolů a zodpovědností v rámci vývojové organizace. Jeho cílem je zajistit vytvoření produktu vysoké kvality požadované zákazníkem v rámci predikovatelného rozpočtu a časového rozvrhu. Schéma RUP znázorňuje Obrázek 1 : Rational Unified Process. (1)

V praxi to znamená, že RUP klade velký důraz na procesy a podpůrné nástroje, obsah dokumentace a provázanost s UML.

2.1.4 Agilní metodiky

Jedná se o metodiky, které jsou založené na iterativním a inkrementálním vývoji. Kladou velký důraz na rychlý vývoj softwaru a snahu o pružnou reakci na změnu požadavků v průběhu vývojového cyklu. Velmi důležitá je komunikace mezi vývojáři a komunikace se zákazníkem, od kterého se očekávají připomínky k aktuálnímu stavu produktu, který se mu průběžně předkládá. Do pozadí u těchto metodik ustupuje dokumentace. Mezi zástupce těchto metodik patří SCRUM nebo Extrémní programování. (2)



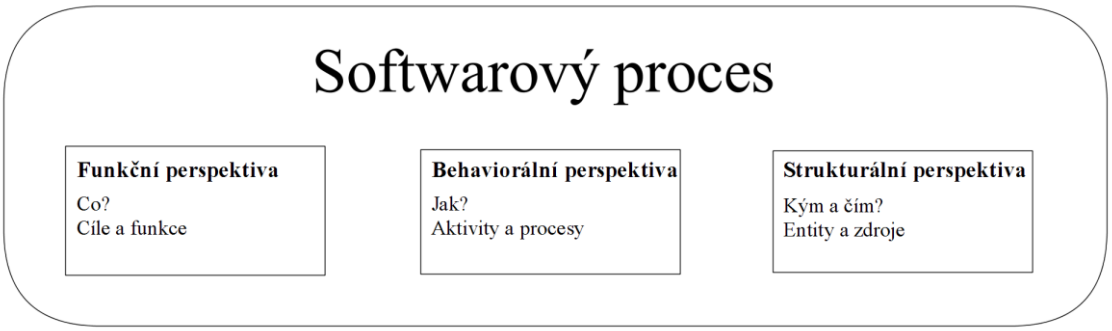
Obrázek 1 : Rational Unified Process

2.2 Modelování softwarového procesu

Definice: Model byznys procesu je abstraktní reprezentace byznys procesu obvykle umožňující jeho další zpracování automatizovaným způsobem. (3)

Při modelování můžeme na softwarový proces nahlížet ze tří perspektiv:

- *Funkční perspektiva* – je zaměřená na funkce se vstupy a výstupy, pravidla pro transformaci vstupů na výstupy a dosažení cílů.
- *Behaviorální perspektiva* – je zaměřená na řídící aspekt vykonávání procesu pomocí stanovení událostí a podmínek, za kterých mohou být jednotlivé funkce prováděny.
- *Strukturální perspektiva* – je zaměřená na statický aspekt procesu, snaží se zachytit veškeré entity a zdroje vystupující v procesu, jejich atributy a vzájemné vztahy.



Obrázek 2 : Perspektivy softwarového procesu

Žádná z těchto perspektiv nestojí osamoceně, pro kompletní zachycení softwarového procesu je potřeba zachytit proces ze všech tří perspektiv. Tyto perspektivy můžeme zachytit třemi způsoby:

- *Neformálními metodami*
- *Formálními metodami*
- *Semi-formálními metodami*

2.2.1 Neformální metody

Pro člověka jsou tyto metody nejpřirozenější a také od něj nevyžadují žádné znalosti z oblasti modelování a zachycování znalostí. Příkladem této specifikace může být přirozená řeč, poznámky, obrázky, náčrtky atd. Z předchozích vět je patrné, že neformální metody nemají jasně danou syntaxi ani sémantiku a nelze je tedy použít pro vytvoření takové specifikace, kterou by bylo možno využít k automatizovanému zpracování.

2.2.2 Formální metody

Tyto metody jsou vhodné pro automatizované zpracování (např. analýza specifikace a její validace) a mají jasně definovanou syntaxi a sémantiku. Pro člověka bez zkušeností z oblasti reprezentace znalostí jsou ovšem velmi těžko pochopitelné. Mezi zástupce těchto metod můžeme zařadit OWL, Petriho sítě, konečné automaty nebo WF-sítě. (4)

2.2.3 Semi-formální metody

Na pomezí formálních a neformálních metod nalezneme semi-formální metody, které mají striktně definovanou syntaxi, ale sémantiku lze chápat různě. Mezi tyto metody můžeme zahrnout grafický jazyk UML, který je kompromisem mezi čitelností pro člověka a zpracovatelností pro stroj – diagramy zachycené v UML se dají převést do programového kódu. Z důvodu nepřesně definované sémantiky však nelze provádět např. validaci nebo analýzu specifikace. Mezi další zástupce těchto metod řadíme IDEF, ARIS, BPMN nebo EPC. (5) Pro modelování podnikových procesů se v současnosti používá primárně BPMN, avšak já se v této práci zaměřuji na modelování pomocí UML. Mám k tomu hned několik důvodů: UML je pro uživatele jednodušší na pochopení a je univerzálnější co do možností použití, kdežto hlavním účelem BPMN je podpora procesního řízení a používá se primárně při komunikaci mezi analytiky a vývojáři, což není cílem této práce.

2.3 Meta-modely

Definice: Meta-model je model, který definuje jazyk určený pro vytvoření modelu. (3)

Jedním z cílů této práce je reprezentovat softwarový proces odděleně od konkrétního vizuálního jazyka, musíme si tedy nadefinovat zvlášť meta-model byznys procesu a notaci vizuálního jazyka UML. V této podkapitole se zaměřím na meta-model byznys procesu.

Existuje celá řada procesních meta-modelů, např. AMENITIES meta-model, NATURE kontextově orientovaný meta-model atd. Hlavním účelem existence procesních meta-modelů je na vyšší úrovni abstrakce popsat strukturu byznys procesu. Každý byznys proces obsahuje (ať už používá tuto, nebo jinou terminologii) procesní kroky, kterým je buď atomická aktivita, nebo vnořený proces. Procesní kroky jsou nějakým způsobem koordinovány (větvení, souběžnost, posloupnost atd.) a každý proces se snaží realizovat nějaké cíle. Proces používá objekty, které mohou být aktivní (zdroje) nebo pasivní (entity). Entity mohou být informace nebo hmatatelný materiál. Zdroje můžou být lidské nebo strojové. Každý zdroj poskytuje kompetence, které jsou vyžadovány rolemi. Zdroje tedy vystupují v rolích, které jsou nutné k vykonání aktivit. Tyto aktivity zpracovávají entity v průběhu vykonávání. (3) (5)

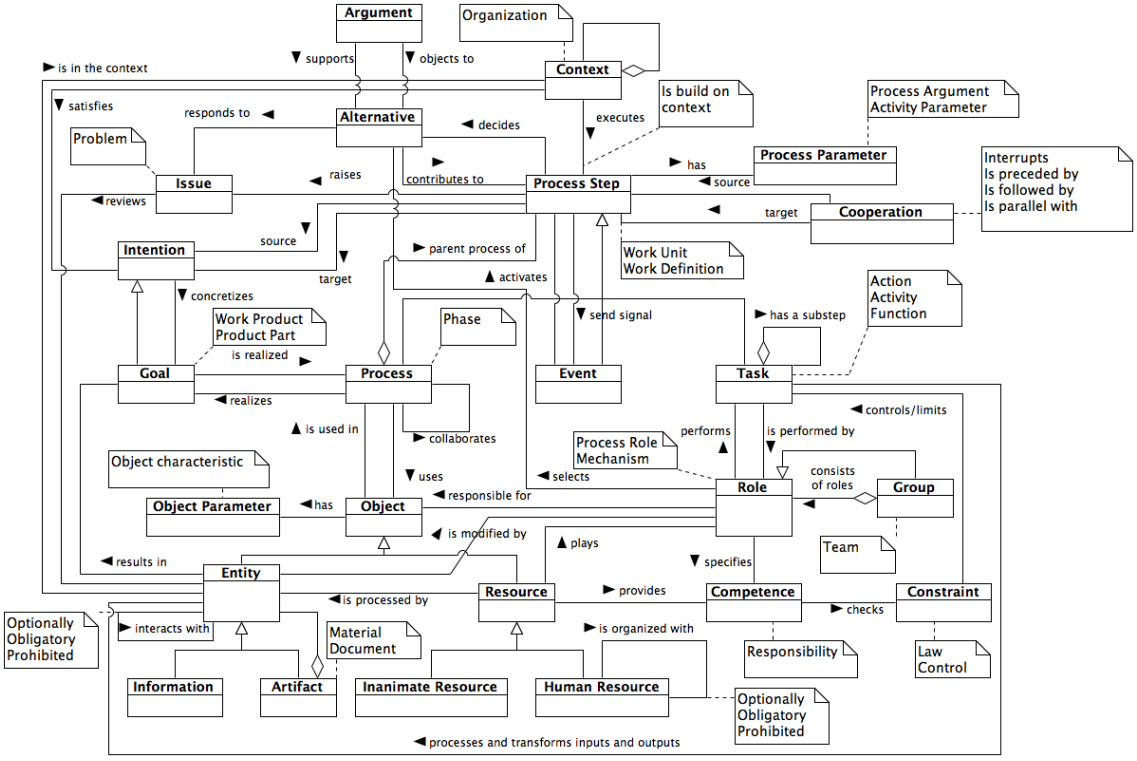
V této práci je jako referenční meta-model softwarového procesu použit meta-model z aktuálně probíhající výzkumné práce na Fakultě informatiky a elektrotechniky na VŠB-TUO. (5) Jako jazyk definice tohoto metamodelu je použit jazyk OWL, který definuje pouze koncepty (třídy) a vztahy mezi nimi. Z této definice lze vytvořit doménový model softwarového procesu.

2.3.1 Unified Process Meta-Model

Většina meta-modelů je zaměřená na jeden aspekt softwarového procesu a zůstávají proto relativně jednoduché, zatímco UPMM se snaží zachytit všechny podstatné koncepty, kterých je definováno přes 20 – viz Obrázek 3 : Třídní diagram Unified Process Meta-Modelu. Tento meta-model také zachovává kompatibilitu s ostatními metamodely (minimálně jednosměrnou), je kompatibilní s Meta-Object Facility a je v souladu se SPEM. (5)

Koncepty v UPMM můžeme rozčlenit na 6 základních částí:

- *Process Dynamics* – jádro celého UPMM zachycuje behaviorální perspektivu pomocí konceptů *Process Step*, *Task*, *Event*, *Process* a *Cooperation*
- *Process Control* – zaměřeno na řízení procesu a kompetence s tím spojené, hlavní koncepty jsou *Competence*, *Role*, *Group* a *Constraint*
- *Process Resources* – popisuje terminologii spojenou s objekty, které jsou procesem využívány – *Object*, *Entity*, *Information*, *Artifact*, *Resource*, *Inanimate Resource*, *Human Resource*
- *Process Scope* – zachycuje kontext vykonávání procesu („firemní kulturu“) pomocí *Context*, *Intention*, *Goal*
- *Process Exceptions* – část orientovaná na zachycení výjimečných stavů, které se mohou během vykonávání procesu vyskytnout – *Issue*, *Alternative*, *Argument*
- *Process Properties* – přidává možnost přizpůsobení procesu v podobě přidání vlastních parametrů, konkrétně pomocí *Process Parameter* a *Object Parameter*



Obrázek 3 : Třídní diagram Unified Process Meta-Modelu

Z obrázku 3 je patrné, že UPMM je velmi komplexní a robustní meta-model, který otevírá širokou škálu možností, jak s procesem pracovat, počínaje vizuálním modelováním konkrétních perspektiv pomocí různých semi-formálních metod až po simulaci procesu. Tato práce je zaměřena na modelování behaviorální perspektivy a proto zde popíši část UPMM zaměřenou na dynamiku procesu (*Process Dynamics*) a řízení procesu (*Process Control*). Na obrázku 7 můžeme vidět UPMM nadefinovaný pomocí formální metody OWL a vizualizovaný v programu Protége. Takto reprezentovaný proces vyžaduje patřičné znalosti formálního modelování a i přes tyto znalosti je taková reprezentace pro člověka velmi nesrozumitelná.

2.3.1.1 ***Dynamika procesu***

Krok procesu (*Process Step*) je jádrem meta-modelu a zastřešuje všechny hlavní elementy procesu jako abstraktní definice práce vykonávané ve společnosti. Krok procesu je abstraktní proto, že jeho hlavním účelem je specifikace společných atributů a vztahů pro jeho specializace.

Úkol *(Task*) je základní specializací procesního kroku a reprezentuje někým nebo něčím vykonávané aktivity v rámci organizace. Úkol se může skládat z dalších (atomických) úkolů, které už jsou dále nedělitelné, atomičnost je reprezentována pomocí boolean vlastnosti *isAtomicStep*.

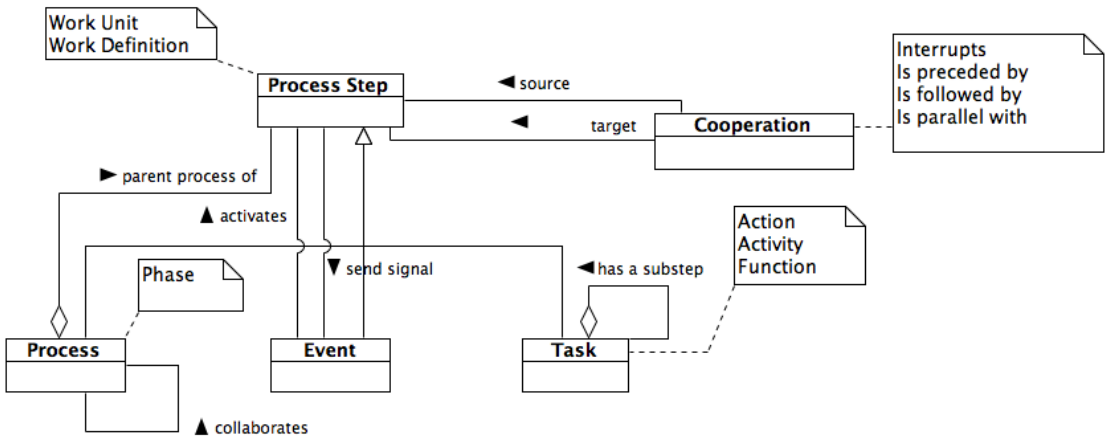
Událost (*Event*) je další specializací procesního kroku, zaměřuje se na systémové aktivity a události v reálném světě – tzn. procesní kroky, které nejsou někým vykonávány. Událost může být spuštěna úkolem (vlastnost *isFiredBy*) a také může spouštět úkol (vlastnost *triggers*).

Proces (*Process*) je také specializací procesního kroku, ale na rozdíl od úkolu dovede zachytit složitější procesy a hierarchicky je rozložit na další procesy, události a úkoly. Hlavním účelem je zachytit vztahy mezi procesy a podprocesy.

Kooperace (*Cooperation*) spojuje dva procesní kroky a identifikuje typ spolupráce těchto kroků. Vždy je určen zdrojový a cílový procesní krok.

Pomocí příslušných vlastností lze u kooperace zachytit:

- Sekvenční vykonávání procesů (*precedes*, *isFollowedBy*)
- Souběžné vykonávání procesů (*isParallelWith*)
- Jeden proces přerušuje vykonávání druhého (*interrupts*)



Obrázek 4 : Dynamika procesu

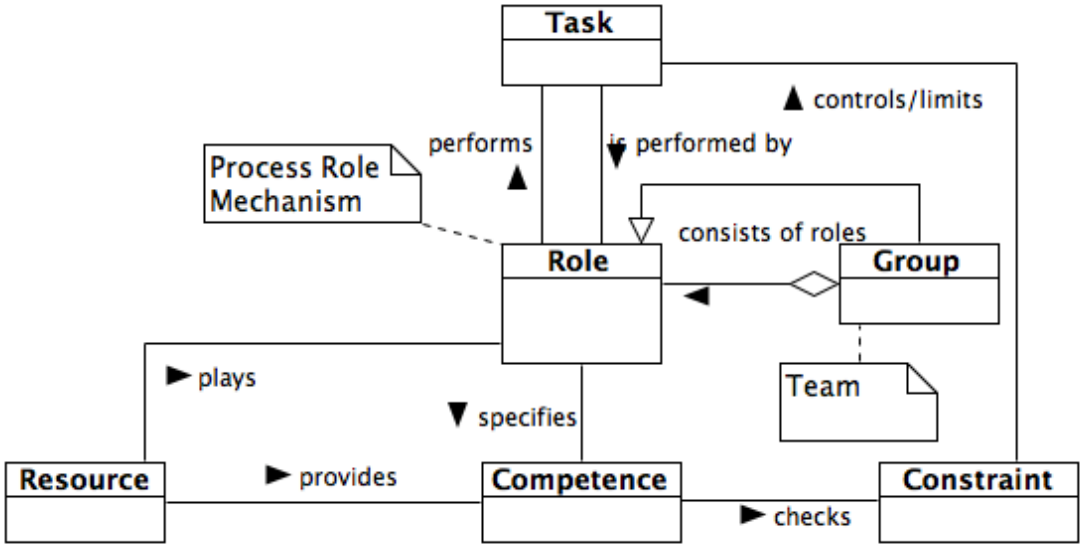
2.3.1.2 ***Řízení procesu***

Kompetence (*Competence*) přiřazují rolím dovednosti, které jsou potřebné pro vykonávání úkolů. Bez příslušných kompetencí nemůže role vykonávat konkrétní úkol.

Role definují pozice nebo zodpovědnosti primárně lidských zdrojů v organizaci a mají své specifické kompetence. Každý úkol může být vykonáván pouze specifikovanými rolemi, popř. může role vybírat alternativu.

Skupina (*Group*) je specializací role a může být zkomponovaná z více rolí. Smyslem existence skupiny na úrovni rolí je zachycení obecné struktury týmu potřebného pro vykonání specifického úkolu.

Předpis (*Constraint*) je v kombinaci s kompetencemi jedním z kontrolních mechanismů řízení procesu. Může například určovat požadovanou úroveň kompetence pro vykonání úkolu.



Obrázek 5 : Řízení procesu

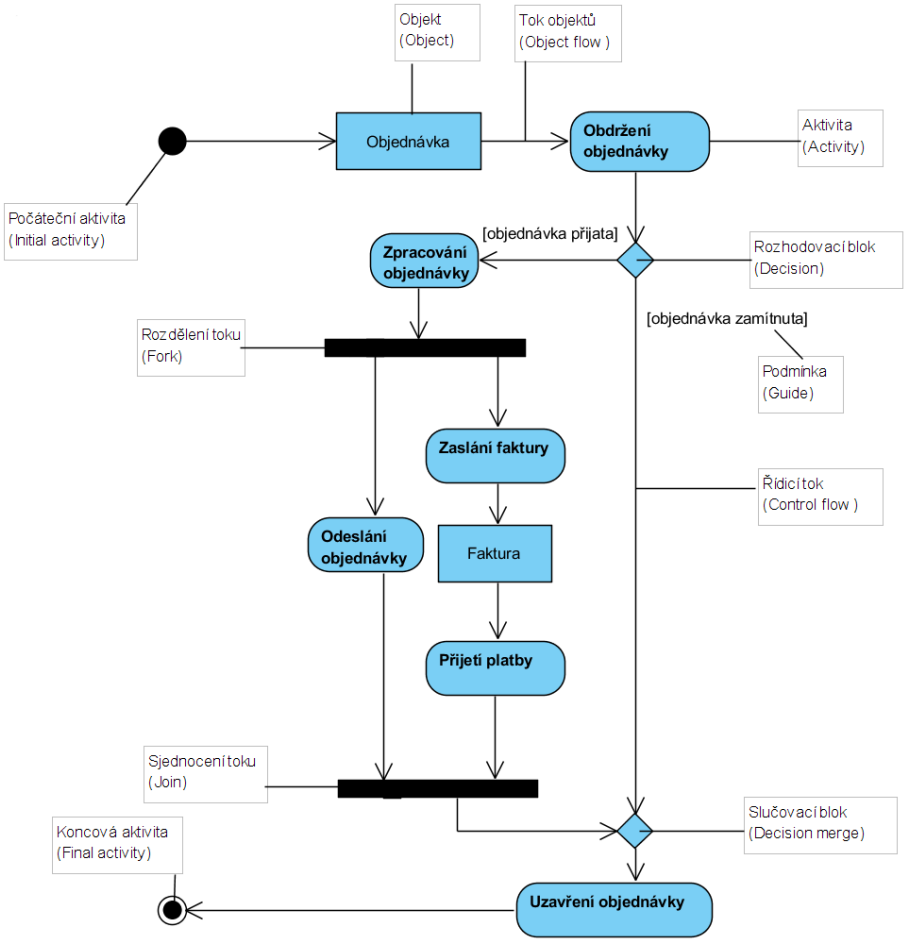
2.4 UML

UML je grafický modelovací jazyk, který slouží hlavně k vizualizaci, návrhu, specifikaci a dokumentaci softwarových systémů, podnikových procesů, atd. UML je standardizovaný organizací OMG a jeho aktuální standardizovaná verze je 2.4.1. Nejvíce využívanou částí standardu jazyka UML jsou diagramy, které se dělí do několika skupin podle perspektivy, ze které se dívají na systém/proces. (6)

- Funkční perspektiva
- Logická perspektiva
- Behaviorální perspektiva (zachycení dynamického chování)
- Implementační perspektiva

2.4.1 Diagram aktivit

Hlavním důvodem k vytvoření diagramu aktivit je zachycení toku činností v procesu. Popisuje jednotlivé procesní kroky a entity pomocí aktivit a objektů a přechody mezi nimi pomocí hran.



Obrázek 6 : Ukázka aktivitního diagramu

2.5 XML

XML (*Extensible Markup Language*) je značkovací jazyk standardizovaný organizací W3C. Extensible znamená, že značky, které se v dokumentu používají, si můžeme vytvářet sami, podle našich potřeb. XML dokument má stromovou strukturu a skládá se z elementů, které mohou mít atributy. Každý dokument je reprezentován kořenovým elementem, do kterého se další elementy vnořují. Obecně slouží XML pro výměnu informací, není spjaté s žádnou konkrétní platformou a je zdarma. (7)

2.5.1 OWL

OWL (Web Ontology Language) je značkovací jazyk pro vytváření ontologií. Pokud se v tomto textu vyskytuje pojem OWL, vždy mám na mysli novější verzi OWL2. Existuje několik možných syntaktických zápisů OWL dokumentů, v této práci je použita syntaxe OWL/XML, která specifikuje XML serializaci struktury OWL ontologie. V širším slova smyslu hovoříme o tom, že OWL je jazyk pro reprezentaci znalostí. Znalostní báze popsané v OWL jsou navrženy primárně pro interpretaci počítačem, ale lze je vizualizovat např. pomocí nástroje Protége. V této práci jsou pomocí OWL zachyceny jednotlivé elementy konkrétního softwarového procesu (jinými slovy znalostní báze softwarového procesu) a budeme tedy tuto znalostní bázi brát jako vstup našeho nástroje, který si tuto bázi zapamatuje a umožní nám pomocí mapovacích pravidel vizuálně modelovat v konkrétní notaci a mít při tom jistotu, že používáme pouze validní elementy procesu, který byl do této báze zachycen. Kromě znalostní báze je v OWL popsána struktura meta-modelu SP, který je potřeba naimplementovat jako doménový model. (4)

Ilustrační příklad OWL2 XML syntaxe:

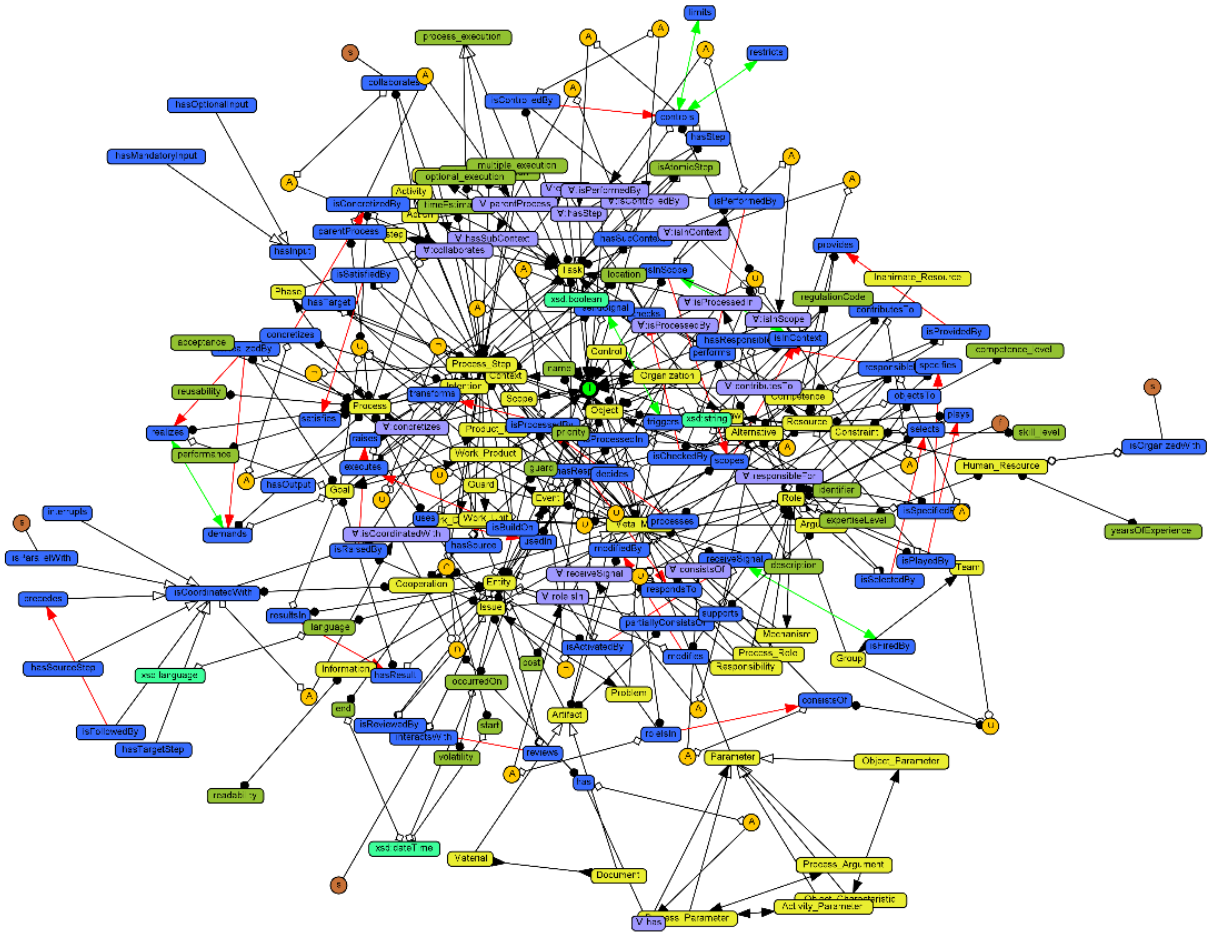
```
<Ontology ontologyIRI="http://example.com/myontology.owl">
   <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
   <Declaration>
     <Class IRI="Tea"/>
   </Declaration>
</Ontology>
```

Malá část struktury kódu OWL dokumentu použitého v této práci je pro ilustraci v příloze A. Tento kód je poskládaný z výňatků z několika částí jednoho dokumentu, je tedy neúplný – samotný dokument se znalostní bází má tisíce řádků.

Ontologie v OWL je množina axiomů, které poskytují explicitní logická tvrzení o třech typech věcí: třída (*class*), individuum (*individual*) a vlastnost (*property*). Každá třída, individuum i vlastnost má IRI, což je unikátní identifikátor dané věci. V OWL existují dva typy vlastností: datové vlastnosti (*data property*), což jsou relace spojující individuum a data určeného typu nebo objektové vlastnosti (*object property*) jakožto binární relace mezi dvěma individui. Pro odvozování nových faktů, které jsou v ontologii implicitně obsaženy, se používá tzv. reasoner, což je část software. Veškeré axiomy meta-modelu SP zobrazuje Obrázek 7 : Meta-model SP v OWL vizualizovaný v pluginu PG ETI SOVA v Protége 5.0.

Seznam nejdůležitějších axiomů v OWL:

- *Class declaration* – definuje třídu. Třída může obsahovat individua.
- *Individual declaration* – definuje pojmenované individuum.
- *Class assertion* – určuje příslušnost individua ke třídě.
- *Subclass assertion* – říká, že jedna třída je specializací druhé třídy.
- *Property declaration* – definuje buď *Object property* nebo *Data property*
- *Property assertion* – určuje, že individuum má buď nějakou *Object property* včetně odkazu na individuum v této relaci, nebo *Data property* včetně hodnoty



Obrázek 7 : Meta-model SP v OWL vizualizovaný v pluginu PG ETI SOVA v Protége 5.0

3 Praktická část

Tato kapitola je věnována praktické implementaci nástroje pro platformu Microsoft .NET, konkrétně .NET WPF pomocí knihoven GoXam a OWL API. Nejprve jsou popsány důležité části a principy těchto knihoven a posléze vysvětlena implementace metamodelu softwarového procesu, implementace zpracování znalostní báze SP a nakonec implementace notace aktivitního diagramu a mapovací pravidla mezi meta-modelem SP a notací diagramu aktivit.

3.1 Platforma Microsoft .NET

Microsoft .NET Framework je platforma pro vývoj aplikací pro systémy Windows, Windows Phone, Windows Server a Microsoft Azure. Skládá se z modulu CLR (Common Language Runtime) a knihovny tříd, která obsahuje třídy a rozhraní pro velké množství technologií obsažených v tomto frameworku. V rámci .NET Frameworku je možné vyvíjet ve více programovacích jazycích, jako např. C# nebo Visual Basic .NET, kód se vždy přeloží do mezijazyka Common Intermediate Language. Modul CLR je základem rozhraní .NET Framework. Runtime modul si lze představit jako agenta, který spravuje kód v době provádění, poskytuje základní služby, jako je například správa paměti, správa vláken, vzdálená komunikace a současně také zajišťuje přísnou bezpečnost typů a další formy přesnosti kódu, které podporují zabezpečení a robustnost. Ve skutečnosti je koncept správy kódu základním principem modulu runtime. Kód, který se zaměřuje na modul runtime, je znám jako spravovaný kód. Zatímco kód, který se nezaměřuje na modul runtime, je znám jako nespravovaný kód. Knihovna tříd je všeobecná, objektově orientovaná kolekce opakovaně použitelných typů, které lze použít pro vývoj aplikací – od tradičních aplikací pro příkazový řádek nebo WPF aplikací s grafickým uživatelským rozhraním (GUI) až po aplikace založené na nejnovějších metodách poskytovaných technologií ASP.NET. (8)

Jak lze očekávat od objektově orientované knihovny tříd, tak v rozhraní .NET Frameworku umožňuje provádět řadu běžných programovacích úkolů, včetně úkolů jako je například správa řetězců, shromažďování dat, možnosti připojení k databázi nebo přístup k souborům. Rozhraní .NET Frameworku můžeme použít k vytvoření aplikace Windows Presentation Foundation (WPF) a touto cestou se budu ubírat i v této práci. (9)

3.1.1 Windows Presentation Foundation

Windows Presentation Foundation je součástí .NET Frameworku od verze 3.0 a jeho účelem je tvorba aplikací s GUI. Pro návrh vzhledu se používá značkovací jazyk XAML, díky čemuž dochází k oddělení aplikačního kódu (*code behind*) a vzhledu aplikace. WPF nabízí možnost škálování velikosti aplikace nezávisle na rozlišení a typu zařízení, tvorbu 2D a 3D grafiky, vektorovou a rastrovou grafiku, animace, vykreslování textu, podporu multimediálních funkcí, provázání dat (*data binding*) a také možnost hostovat aplikaci v okně prohlížeče Internet Explorer 7 a vyšším. Ve WPF lze také nadefinovat vlastní ovládací prvky včetně jejich chování

pomocí rozšíření třídy System.Windows.Controls.**Control**, na čemž je postavena rozsáhlá knihovna pro tvorbu diagramů a grafů ve WPF - GoXam od společnosti Northwoods Software.

3.1.2 GoXam

Knihovna GoXam od firmy Northwoods Software, konkrétně tedy GoWPF, což je implementace knihovny GoXam pro použití ve WPF aplikacích, poskytuje ovládací prvky pro implementaci diagramů. (10)

Třída **Diagram** je specializací výše zmíněné třídy **Control** z WPF a podporuje charakteristické vlastnosti WPF aplikací – styly, data binding, animace, šablony, příkazy a tisk. Diagram se skládá z uzlů, které mohou být propojeny hranami a seskupené dohromady do skupin. Všechny tyto části jsou shromážděny ve vrstvách (**Layers**) a většina částí je propojena s daty z aplikace.

Každý diagram má **Model**, který interpretuje aplikační data a určuje hranové propojení uzlů a vztahy část-celek v závislosti na formátu aplikačních dat.

Každý diagram má také **PartManager**, který je zodpovědný za vytvoření **Node** (uzlu) pro každý datový prvek v kolekci **NodeSource** v modelu diagramu a za vytváření a rušení **Link** (hran).

Každý uzel nebo hrana jsou definovány pomocí **DataTemplate**, který definuje jejich vzhled a chování. Uzly mohou být pozicovány manuálně (interaktivně popř. programaticky) nebo může být jejich pozice organizována automaticky v **Layoutu** (dispozici) diagramu.

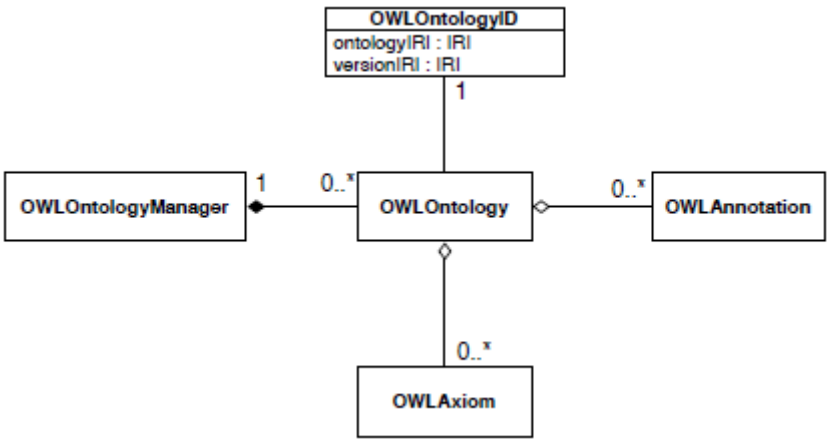
Tools (nástroje) zachytávají události vyvolávané myší. Každý diagram má několik nástrojů, jež vykonávají interaktivní úlohy jako vybrání části diagramu, přesouvání nebo vytváření nových hran mezi dvěma uzly. **ToolManager** rozhoduje, který nástroj by měl reagovat v závislosti na událostech myši a dalších okolnostech.

Každý diagram má také **CommandHandler**, který implementuje řadu příkazů (např. mazání) a umí reagovat na události vyvolané klávesnicí.

DiagramPanel poskytuje možnost posouvat nebo přibližovat/oddalovat části diagramu. Ovládací prvek **Overview** (přehled) umožňuje uživateli vidět celý diagram a volit, která část se bude zobrazovat. Ovládací prvek **Palette** (paleta) v sobě drží uzly, které může uživatel pomocí drag-and-drop přesunout do diagramu.

3.1.3 OWL API

OWL API je rozhraní pro programování v jazyce Java s open source licencí LGPL pro vytváření, úpravu a serializaci OWL ontologií. OWL API je vyvíjeno na University of Manchester a v aktuální verzi úzce spjato s OWL 2 a podporuje analýzu a interpretaci OWL ontologií v syntaxích definovaných ve W3C specifikaci (funkcionální, RDF/XML, OWL/XML a Manchester OWL), manipulaci s ontologickými strukturami a použití tzv. reasonerů (kapitola OWL3.1.3.2). OWL API je v současnosti používáno v mnoha nástrojích a aplikacích, například v nástroji Protége. (11) (12) (13)



Obrázek 8 : Základní třídní diagram OWL API pro správu ontologií

Rozhraní **OWLOntology** poskytuje přístup k metodám pro efektivní přístup k axiomům obsaženým v ontologii. Metody pro ukládání axiomů jsou poskytovány pomocí různých implementací tohoto rozhraní. V OWL API je kladen důraz na striktní oddělení mezi komponentami, které poskytují konkrétní funkcionalitu, jako jsou reprezentace, manipulace, analýza a interpretace ontologie. Každá ontologie a verze ontologie má také své unikátní **IRI**, což není nic jiného než standardizovaný řetězec, pomocí kterého můžeme jednoznačně identifikovat ontologii a v návaznosti také veškeré třídy, individua a vlastnosti v ontologii.

Třída **OWLOntologyManager** poskytuje jakýsi středobod pro vytváření, načítání, úpravu a ukládání ontologií, které jsou instancemi OWLOntology a drží si v paměti veškeré načtené ontologie, takže k ontologiím se přistupuje výhradně skrz tohoto správce. Každá ontologie je vytvořena nebo načtena pomocí tohoto správce a každá instance ontologie je unikátní pro instanci správce. Také veškeré změny v ontologiích se dějí skrze tohoto správce. OWLOntologyManager umožnil zakrýt velmi komplexní mechanismy spojené s výběrem správných komponent pro analýzu (*parsing*) a interpretaci (*rendering*) při načítání a ukládání ontologií.

Rozhraní **OWLReasoner** poskytuje přístup k funkcionalitě pro odvozování implicitních znalostí, podporuje kontrolu hierarchií tříd a vlastností, kontrolu konzistence atd. Díky tomuto

rozhraní můžeme v klientské aplikaci vyměnit implementaci reasonerů bez změn v již napsaném kódu, avšak ne všechny reasonery se chovají stejně, i když implementují toto rozhraní.

3.1.3.1 ***OWL API for .NET***

Jak bylo napsáno na začátku této podkapitoly, OWL API je vytvořeno pro použití v jazyce Java, avšak implementační platforma této práce je .NET. Pro .NET ovšem neexistuje žádné odladěné API pro práci s OWL ontologiemi a syntaktickým formátem OWL2 XML a implementovat kompletní funkcionalitu pro práci s tímto formátem by časově i tematicky přesáhlo úroveň této práce. Řešením je projekt OWL API for .NET. (14)

OWL API for .NET je open source projekt, který má na starosti společnost Cognitum a jeho cílem bylo přenést OWL API na platformu .NET. Tento projekt využívá projektu IKVM.NET který umí staticky překompilovat knihovny z jazyka Java na .NET knihovny a implementuje JVM (java virtual machine) pro běh na platformě .NET. (15)

Tento přístup nám tedy umožní používat kód napsaný v jazyce Java v prostředí .NET, avšak s několika úskalími. V průběhu vývoje jsem zjistil, že generika v Javě a .NETu jsou implementována odlišně, takže je potřeba uzpůsobit tomu C# kód – například metody, které v Javě vrací Set<Class> vrací při použití v C# pouze Set a je tedy nutné zřetězit několik dalších metod pro převedení na pole a použití dynamického typování „var“ a následné přetypování jednotlivých objektů na typ, který je v původním Set<*CLASS*> označen jako *CLASS* (příklad níže)

```
var annotations = individual.getAnnotations(Manager.getOntology(ProfileIRI),
                        DataFactory.getRDFSIsDefinedBy()).toArray().ToList();
foreach (OWLAnnotation item in annotations)
{ ... }
```

Dalším úskalím je, že ne všechen kód z Javy se musí nutně správně zkompilovat do výsledného souboru assembly .dll a je tedy potřebné tyto chyby odstranit přidáním na první pohled nesmyslných řádků kódu před použitím špatně překompilovaného kódu (příklad pod tímto odstavcem). Toto všechno určitě také nějakým způsobem degraduje výkon aplikace, ale je to přípustná daň za to, že není nutno implementovat celé OWL API pro tuto platformu ručně.

```
com.sun.org.apache.xerces.@internal.jaxp.SAXParserFactoryImpl s = new
com.sun.org.apache.xerces.@internal.jaxp.SAXParserFactoryImpl();
```

3.1.3.2 ***Reasonery***

Reasoner je klíčová komponenta pro práci s OWL ontologiemi. Ve skutečnosti by měly být všechny dotazy na OWL ontologii provedeny pomocí reasoneru. To proto, že znalosti v ontologii nemusí být určeny explicitně a reasoner je potřeba použít k odvození implicitních znalostí, aby dotazy vracely korektní výsledky. OWL API obsahuje rozhraní OWLReasoner pro přístup k reasonerům, ale je to pouze rozhraní, takže je potřeba přiložit konkrétní implementaci takového reasoneru. Krátký seznam několika reasonerů, které implementují rozhraní OWLReasoner: Chainsaw, FaCT++, JFact, HermiT, Pellet. V této práci používám **StructuralReasoner**, který je přímo součástí OWL API a **Pellet** reasoner v místech, kde předchozí jmenovaný svojí funkčností neodpovídá požadavkům.

3.2 Architektura řešení

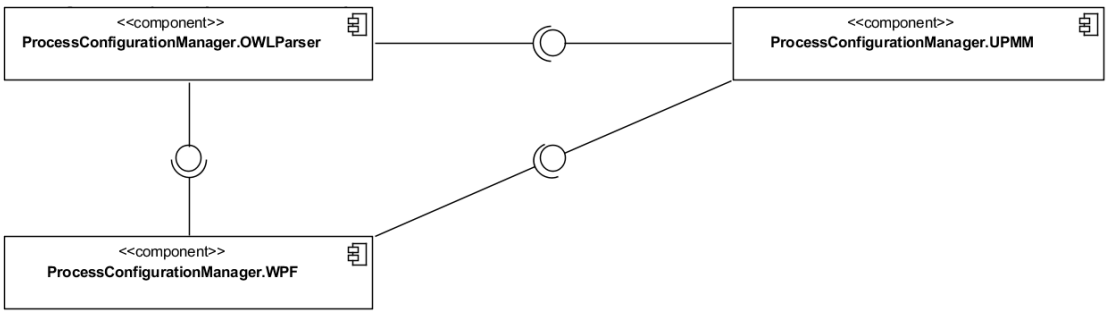
Řešení jsem se rozhodl rozdělit do několika komponent tak, aby od sebe byly logicky odděleny části kódu podle řešeného problému. Řešení nese název ProcessConfigurationManager a skládá se z 3 assembly:

- ProcessConfigurationManager.**UPMM**
- ProcessConfigurationManager.**OWLParser**
- ProcessConfigurationManager.**WPF**

ProcessConfigurationManager.**UPMM** je assembly, ve které se nachází kód doménového modelu UPMM a nepoužívá žádné nestandardní knihovny. Doménový model nám umožňuje oprostit se od reprezentace dat v ontologii a zjednodušit tak kód používaný ve WPF aplikaci.

ProcessConfigurationManager.**OWLParser** shromažďuje veškerý kód pro práci s OWL API a transformaci individuí z ontologií na List (v češtině *seznam*, ale nadále budu používat pojem List, protože je to přímo název třídy reprezentující seznam) objektů doménového modelu UPMM. Tato assembly potřebuje ke svému běhu některé knihovny projektu IKVM.NET kvůli správnému běhu OWL API for .NET a kromě těchto knihoven ještě samotnou knihovnu OWL API a knihovnu Pellet reasoneru, obě překompilované pomocí IKVM.NET pro použití na této platformě. Navíc potřebujeme také referenci na assembly s doménovým modelem. V Resources (zdrojích) této knihovny je uložen soubor UPMM.owl obsahující ontologii definující třídy a vlastnosti (meta-model) UPMM, protože tato ontologie je potřebná pro dotazování reasonerem, jelikož je vyžadována skrz include v souboru OWL s profilem SP, který obsahuje individua těchto tříd.

ProcessConfigurationManager.**WPF** je assembly WPF aplikace. Tato assembly spravuje kód GUI a kód potřebný pro správnou funkčnost diagramatické části programu, jsou zde vytvořeny třídy vyžadované jako model diagramu – uzel a hrana, jejichž implementace je spjata s modelovými třídami z knihovny Northwoods.GoWPF. Jsou vyžadovány reference na knihovnu Northwoods.GoWPF a reference na obě zbývající assembly z tohoto projektu.



Obrázek 9 : Diagram vytvořených komponent řešení ProcessConfigurationManager

3.3 Implementace doménového modelu UPMM

Jako mezikrok mezi modelem softwarového procesu zachyceným v OWL ontologii a modelem diagramu jsem zvolil vytvoření doménového modelu podle UPMM. Tento způsob řešení spočívá v tom, že pro každou třídu definovanou v UPMM ontologii je vytvořena C# třída, pro každou datovou vlastnost (*data property*) v UPMM je vytvořena property (*vlastnost*) u ekvivalentní C# třídy a pro každou relaci mezi třídami (*object property*) v UPMM je vytvořen generický **List**<*CLASS*> u ekvivalentní třídy pro každou takovouto relaci, kde *CLASS* je C# třída, která odpovídá třídě individuí v UPMM na druhém konci relace.

Toto řešení má několik argumentů pro a několik argumentů proti. Argumenty pro toto řešení jsou následující.

1. Odstíníme část programu, která má na starosti mapování individuí mezi elementy UPMM a elementy notace aktivitního diagramu UML, od neustálého dotazování reasoneru, což by mohlo mít velký vliv na výkon celé aplikace.
2. Možnost zachycení celého profilu SP (ontologie s konkrétními individui) do Listu, který bude obsahovat prvky tohoto doménového modelu a tento list dotazovat mnohem jednoduššími dotazy pomocí LINQ to Objects, než v případě dotazů pomocí reasoneru a následné transformaci dat po každém dotazu.
3. Použití těchto doménových objektů pro validaci modelování pouze povolených vztahů v dané notaci na základě mapovacích pravidel.

Argumenty proti tomuto řešení:

1. Tímto řešením si znemožníme, nebo lépe řečeno velmi znesnadníme možnost v budoucnu ukládat změny zpět do ontologie. (Toto chování ale není chtěné!)
2. Je potřeba navíc vytvořit komponentu, která se bude starat o kompletní a korektní převod individuí z profilu SP na objekty doménového modelu. Tato komponenta bude relativně rozsáhlá kvůli nutnosti navázat každou vlastnost na ekvivalentní List<CLASS> u dané třídy.
3. Jsme fixováni na konkrétní verzi ontologie meta-modelu UPMM, takže při aktualizacích této ontologie bude potřeba pozměnit a popřípadě přidat ekvivalentní kód v doménovém modelu a v o několik řádků výše zmíněné transformační komponentě.

Po zvážení všech těchto argumentů jsem se rozhodl vytvořit doménový model UPMM i za cenu nutnosti setrvání u konkrétní verze ontologie UPMM a absenci ukládání změn zpět do ontologie, což není chtěným chováním konfiguračního nástroje. Cílem této práce je umožnit modelovat na základě daných pravidel vztahy povolené v UPMM z již existujících elementů SP.

Základní třídou celého doménového modelu UPMM je třída **SoftwareProcessElement**, která je předkem všech ostatních tříd a obsahuje anotační atributy, které má každý element v ontologii. Tato třída má 3 řetězcové property – *IRI* (jednoznačný identifikátor individua), *Name* (název individua) a *Description* (popis individua). Kompletní hierarchie dědičnosti tříd je v Příloha B: .

```
public abstract class SoftwareProcessElement
    {
        public string IRI { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public SoftwareProcessElement(string identifier=null,
                    string name=null, string description=null)
        {
            IRI = identifier;
            Name = name;
            Description = description;
        }
    }
```

Při implementaci doménového modelu jsem se musel rozhodnout, které třídy implementovat, jelikož v ontologii existuje object property, která určuje, že některé třídy si mohou být ekvivalentní, což by při tvorbě doménového modelu mohlo být velmi matoucí, takže jsem implementoval vždy pouze jednu z několika možných ekvivalentních tříd. V ontologii UPMM (verze 1.0) byly např. ekvivalentní třídy ***Law*** a ***Constraint*** a já si vybral pro implementaci třídu **Law**, jelikož v ontologii jsou tyto třídy významově identické, takže při použití reasoneru je jedno, jestli se zeptáme na všechna individua třídy Law nebo třídy Constraint, výsledek bude totožný.

Některé *object property* jsou v ontologii UPMM definovány jako navzájem inverzní, což jsem naimplementoval tak, že u obou tříd, které jsou spolu v relaci, je vytvořen **List**<*CLASS*>, kde *CLASS* je typ protější třídy z této relace. Tohle není nejlepší řešení, protože při přidávání a odebírání těchto spojení z Listů je vždy potřeba myslet na to, že spojení se musí přidat/odebrat z Listu daného spojení i z Listu spojení inverzního. Tohle je velmi pracné řešení, ale v současné době, kdy je při transformaci OWL profilu SP na doménový model UPMM požadována pouze jednosměrná transformace z OWL profilu na doménový model, toto řešení postačuje.

Všechny parametry v parametrických konstruktorech, které jsou u tříd vytvořeny, mají nastavenou defaultní hodnotu na null a to kvůli případu, kdy bychom neměli všechny parametry inicializovány – stačí nám jeden konstruktor a není potřeba větší množství konstruktorů pro všechny možné případy.

Meta-model je převeden do kódu celý, i když pro splnění cílů práce je potřebná implementace jen dvou z jeho šesti částí – to z důvodu možné budoucí práce s ostatními částmi meta-modelu v dalších vizuálních modelovacích notacích. Některé třídy, jmenovitě **SoftwareProcessElement**, **ProcessStep**, **Object**, **Entity** a **Resource**, jsou definovány jako abstraktní – vycházel jsem z textu, který popisoval UPMM a ve kterém bylo výslovně řečeno, že tyto třídy slouží pouze pro zachycení společných vlastností a relací pro některé další třídy. (5)

Jediná odchylka od ontologie UPMM, která nastala při implementaci, byla u třídy **Cooperation**, kde je místo dědičné hierarchie vlastností zavedena vlastnost *Relation*, která nabývá hodnot výčtového typu **CooperationType** – *IsCoordinatedWith*, *IsParallelWith*, *IsFollowedBy*, *IsPrecededBy*, *Interrupts*. Kromě tohoto výčtového typu je v této assembly ještě výčtový typ **UPMMTypes**, který má pro každou třídu doménového modelu jednu hodnotu – tento výčtový typ slouží pro assembly ProcessConfigurationManager.OWLParser jako vstup pro

slovník (Dictionary) mapující ekvivalentní IRI tříd z UPMM ontologie s třídami doménového modelu.

3.4 Implementace převodu OWL modelu SP na doménový model

V tuto chvíli máme vytvořený doménový model UPMM, ale nemáme komponentu, která by převedla veškerá data, která jsou zachycena v OWL souboru s profilem SP do tohoto modelu. Vytvořil jsem tedy novou assembly ProcessConfigurationManager.OWLParser, která obsahuje fakticky jen jednu statickou třídu s názvem **OWLAPI** rozdělenou do dvou souborů (OWLAPI.cs a OWLToCodeTransformer.cs), tedy partial class. Důvod rozdělení je čistě praktický – přehlednost během programování. Kód třídy má totiž přes 2500 řádků. Kromě této třídy je ve zdrojích této assembly z výše popsaných důvodů také soubor UPMM.owl s ontologií UPMM.

Třída **OWLAPI** je statická z toho důvodu, že ji používáme jen pro zavolání dvou veřejných metod a nenese žádná data a není tedy žádný důvod vytvářet její instanci. V souboru OWLAPI.cs jsou tedy vytvořeny dvě veřejné statické metody (***Initialize*** a ***GetSoftwareProcess)*** a kromě těchto také několik privátních vlastností, které jsou vypsány v kódu pod tímto odstavcem.

```
public static partial class OWLAPI
{
    private static OWLOntologyManager Manager { get; set; }
    private static OWLReasoner Reasoner { get; set; }
    private static OWLReasoner PelletReasoner { get; set; }
    private static OWLDataFactory DataFactory { get; set; }
    private static IRI MetamodelIRI { get; set; }
    private static IRI ProfileIRI { get; set; }
    private static Dictionary<IRI, UPMM.UPMMTypes> IRIClassMap = new
                   Dictionary<IRI, UPMM.UPMMTypes> { ... }
    ...
}
```

Funkce pro ***Manager***, ***Reasoner*** a ***PelletReasoner*** je popsána v 3.1.3. ***DataFactory*** je továrna pro vytváření instancí různých tříd spjatých s reprezentací ontologie v knihovně OWL API a pro tyto účely obsahuje spoustu metod. ***MetamodelIRI*** a ***ProfileIRI*** potřebujeme, abychom mohli v ***Manageru*** přistupovat k jednotlivým ontologiím, jsou to tedy takové „klíče“ pro výběr správné ontologie. Poslední definovanou vlastností je ***IRIClassMap***, což je generický slovník mapující ekvivalentní IRI tříd z UPMM ontologie na třídy doménového modelu. Klíčem je **IRI** a hodnotou je některá z hodnot výčtového typu **UPMMTypes** z doménového modelu. Pod tímto odstavcem nalezneme část inicializace tohoto slovníku, která je v předchozím útržku nahrazena třemi tečkami.

```
private static Dictionary<IRI, UPMM.UPMMTypes> IRIClassMap = new Dictionary<IRI, UPMM.UPMMTypes>
{
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Action"),
UPMM.UPMMTypes.Task},
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Activity"),
UPMM.UPMMTypes.Task},
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Activity_Parameter"),
UPMM.UPMMTypes.Parameter},
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Alternative"),
UPMM.UPMMTypes.Alternative},
```

```
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Argument"),
UPMM.UPMMTypes.Argument},
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Artifact"),
UPMM.UPMMTypes.Artifact},
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Competence"),
UPMM.UPMMTypes.Competence},
    {IRI.create("http://www.kosinar.me/ontologies/UnifiedProcessMetaModel#Constraint"),
...
}
```

Metoda ***Initialize*** požaduje jako vstup řetězec s cestou k souboru s OWL profilem SP a nemá návratovou hodnotu. Jejím účelem je, jak už název napovídá, inicializace – vytvoření paměťové reprezentace ontologií, vytvoření Manageru, DataFactory a obou reasonerů a inicializace MetamodelIRI a ProfileIRI. Volání této metody v kódu nastává pouze v té době, když vkládáme soubor s profilem SP a tato metoda musí být zavolána před ***GetSoftwareProcess***.

```
public static void Initialize(string ontologyProfilePath)
{
    try
    {
        Manager = OWLManager.createOWLOntologyManager();
        //tento řádek je potřebný kvůli nedokonale převedené knihovně
        com.sun.org.apache.xerces.@internal.jaxp.SAXParserFactoryImpl s = new
                com.sun.org.apache.xerces.@internal.jaxp.SAXParserFactoryImpl();

        // načtení owl souboru, který obsahuje metamodel
        byte[] owlResourceUPMM = Properties.Resources.UPMM;
        java.io.ByteArrayInputStream upmmInputStream = new
                java.io.ByteArrayInputStream(owlResourceUPMM);
        OWLOntology ontologyUPMM =
                Manager.loadOntologyFromOntologyDocument(upmmInputStream);
        MetamodelIRI = ontologyUPMM.getOntologyID().getOntologyIRI();
        Trace.WriteLine("Loaded Metamodel Ontology : " + MetamodelIRI);

        // načtení owl báze konkrétního profilu
        java.io.File processModel = new java.io.File(ontologyProfilePath);
        OWLOntology knowledgeProfile =
                Manager.loadOntologyFromOntologyDocument(processModel);
        ProfileIRI = knowledgeProfile.getOntologyID().getOntologyIRI();
        Trace.WriteLine("Loaded Profile Ontology : " + ProfileIRI);

        // vytvoření reasoneru nad profilem
        Reasoner = new StructuralReasonerFactory().
                createReasoner(knowledgeProfile, new SimpleConfiguration());
        PelletReasoner = new PelletReasonerFactory()
               .createReasoner(knowledgeProfile, new SimpleConfiguration());
        Trace.WriteLine("Reasoner is running!");

        DataFactory = Manager.getOWLDataFactory();
    }
    catch (Exception ex)
    {
        throw new ApplicationException("Ontology loading failed!", ex);
    }
}
```

V kódu je možno vidět, že při práci s knihovnou OWL API na této platformě musíme psát kód, ve kterém používáme třídy jazyka Java, což se jeví jako malá komplikace, ale naštěstí je pole bajtů podporované na obou platformách stejným způsobem. Další věcí, která stojí za povšimnutí, je odchytávání výjimek. V C# kódu nemůžeme jednoduše odchytávat specifické výjimky, které jsou vyvolávány knihovnou OWL API, jelikož tyto výjimky nemají nic společného s výjimkami, které jsou na této plaformě, ale specializují výjimky z knihoven Javy, které jsou v našem projektu nareferencované v knihovnách IKVM.NET a nedědí z System.Excpetion. IKVM.NET má mechanismus, který nějakým způsobem výjimky vyhazuje, ale musíme zachycovat obecně System.Exception.

Metoda ***GetSoftwareProcess*** je bezparametrická a vrací **List**<*SoftwareProcessElement*>. Tuto metodu voláme po inicializační metodě a jejím smyslem je zavolat privátní logiku této třídy která je definovaná ve zdrojovém souboru OWLToCodeTransformer.cs. Výstupem této metody je převedený model softwarového procesu v doménových objektech z načteného souboru OWL.

```
public static List<UPMM.SoftwareProcessElement> GetSoftwareProcess()
{
    try
    {
        Trace.WriteLine("Begining UPMM transformation...");
        List<UPMM.SoftwareProcessElement> softwareProcess = new
                 List<UPMM.SoftwareProcessElement>();
        Transform(GetClasses(), softwareProcess);
        Trace.WriteLine("Process transformation done...");

        return softwareProcess;
    }
    catch (Exception ex)
    {
        throw new ApplicationException("Process transformation failed!", ex);
    }
}
```

Soubor OWLToCodeTransformer.cs obsahuje privátní logiku třídy OWLAPI, jako jednoduché pomocné metody pro získání Listu tříd a Listu individuí daných tříd obsažených v naší ontologii (***GetClasses*** a ***GetIndividuals***), metody pro získání anotačních vlastností pro individuum (***GetDescription***, ***GetLabel*** a ***GetBasicProperties***). Dále metodu pro získání hodnoty zadané datové vlastnosti konkrétního individua ***GetConcreteDataPropertyValue*** a metodu ***GetConcreteObjectPropertyValues*** pro získání Listu řetězců, reprezentujících IRI individuí, které jsou v zadané relaci (object property) s konkrétním individuem.

Dále jsou zde pro každou třídu z doménového modelu dvě metody. Jedna metoda pro vytvoření instancí všech individuí dané třídy a inicializaci hodnot jejich datových vlastností a druhá metoda pro inicializaci všech definovaných relací mezi individui této třídy a individui ostatních tříd. Jedinou výjimkou je třída Cooperation, která má pouze jednu metodu z důvodů, které objasním později. Část kódu, která ukazuje základní princip fungování obou těchto metod u jedné ze tříd, uvádím pro její relativně větší rozsah v Příloha C: .

Mezi poslední tři metody této třídy patří metoda ***Transform***, která nedělá nic jiného, než že po sobě zavolá v tomto pořadí metody ***CreateAllIndividuals*** a ***CreateAllRelationships***. Obě tyto metody berou jako vstupní parametry referenci na List všech tříd naší ontologie a referenci na List<SoftwareProcessElement>, do kterého se promítají změny a slouží pro finální výstup celého běhu této logiky (instance je vytvořena v metodě GetSoftwareProcess).

Metoda ***CreateAllIndividuals*** pro každou třídu z Listu tříd ontologie najde hodnotu UPMMTypes ve slovníku IRIClassMap na základě IRI této třídy a následně se kód rozvětví a podle toho, jaká hodnota UPMMTypes je zavolána konkrétní metoda určená pro danou třídu pro vytvoření instancí všech individuí dané třídy a inicializaci hodnot jejich datových vlastností. Kromě abstraktních tříd je jedna třída, která tuto metodu nemá, respektive se její instance vytváří až při běhu metody ***CreateAllRelationships*** – třída **Cooperation**. Důvod tohoto počínání je kvůli tomu, že při vytváření instance Cooperation je potřeba v konstruktoru předat odkaz na zdrojový a cílový objekt třídy ProcessStep z našeho List<SoftwareProcessElement> a tyto ještě nemusí být v tomto okamžiku vytvořeny. Proto je vhodnější vytvořit pouze jednu metodu pro tuto třídu a volat ji až ve chvíli, kdy máme jistotu, že jsou všechna individua již vytvořena.

Metoda ***CreateAllRelationships*** funguje na stejném principu, jako metoda **CreateAllIndividuals** s tím rozdílem, že se pro danou třídu zavolá metoda pro inicializaci všech definovaných relací mezi individui této třídy a individui ostatních tříd.

3.5 Implementace AD notace

Popsat veškeré nabízené možnosti knihovny Northwoods.GoWPF by zabralo několik desítek stran, takže tímto odstavcem odkazuji na dokument, který podrobně popisuje všechny možnosti a alternativy implementace této knihovny. Dále se zaměřím na konkrétní implementaci diagramu a funkčnosti s ním spjaté. (10)

3.5.1 Datový model diagramu aktivit UML

Jako první věc bylo potřeba naimplementovat 2 třídy (**ActivityDiagramNodeData** a **ActivityDiagramLinkData**), které slouží jako modelová data pro náš diagram. Třída **ActivityDiagramNodeData** specializuje **GraphLinksModelNodeData**<String>, kde String je typ unikátního klíče uzlu, a tato třída slouží jako datový model pro všechny uzly v diagramu. Třída **ActivityDiagramLinkData** zase specializuje **GraphLinksModelLinkData**<String, String> a slouží jako datový model pro všechny hrany v diagramu.

Veškeré definované vlastnosti na těchto třídách při změně hodnoty vyvolávají událost ***RaisePropetyChanged***, protože kolekce uzlů a kolekce hran uložené v modelu diagramu jsou implementovány jako **ObservableCollection**<T>, aby byl diagram schopen se automaticky překreslit při úpravě některé z těchto datových kolekcí. T je buď ActivityDiagramNodeData, nebo ActivityDiagramLinkData – v závislosti na tom, o kterou kolekci modelu diagramu se jedná. Obě tyto třídy mají metody pro serializaci a deserializaci (***MakeXElement*** a ***LoadFromElement***) a tyto metody jsou využity při volání metody Load nebo Save na modelu diagramu.

V třídě **ActivityDiagramNodeData** jsou z nadřazené třídy používány vlastnosti ***Key*** a ***Category***, kde Key je unikátní klíč uzlu a Category je typ uzlu. Key se používá při jakékoliv práci s uzlem, jako vytváření hran atd. Category má svůj význam při párování modelu diagramu s vizuálními vzory uzlů (*node template*) – toto nám zaručí, že uzly každé kategorie se budou zobrazovat tak, jak si nadefinujeme v node template v XAMLu stránky, která obsahuje komponenty diagramu. ***Category*** je řetězec nabývající hodnot: *Activity*, *Object*, *Initial Activity*, *Final Activity*, *Decision*, *Decision Merge*, *Fork*, *Join*, *Role*, *Send Signal Action*, *Accept Event Action* a *Note*. ***Key*** je řetězec, který vznikl spojením názvu elementu SP a kategorie, na kterou se mapuje. Dále jsou nově definovány vlastnosti ***IRI***, ***Name***, ***Description***, ***Stereotype***, ***Color*** a ***BorderColor***, které jsou spjaté s elementem doménového modelu, který byl na danou kategorii uzlu namapován. Logicky, IRI = IRI elementu SP, Name = název elementu SP, Description = popis elementu a Stereotype obsahuje název třídy mapovaného elementu z doménového modelu uzavřenou mezi „<<“ a „>>“. ***Color*** je vlastnost, která vrací řetězec s názvem barvy (pro každý ***Stereotype*** je vrácena jiná barva), jež je použit při vytváření palety pro její větší přehlednost – tato vlastnost se neukládá při serializaci. ***BorderColor*** slouží jako barva okraje uzlu a může být buď červená, nebo černá – mění se v závislosti na tom, zda je uzel validní v aktuální Swimlane.

Třída **ActivityDiagramLinkData** definuje řetězcové vlastnosti ***Guide*** a ***Category***. Guide slouží jako volitelný popisek u hrany a Category specifikuje typ hrany – *Control Flow*, *Object Flow* nebo *Anchor*. Použití Category je podobné jako u předchozí třídy – párování s vizuálním vzorem pro hranu z XAMLu.

Pro reprezentaci diagramu slouží třída **Northwoods.GoXam.Diagram**. Při vytváření instance stránky, která obsahuje diagram, musíme vytvořit model, kterému inicializujeme kolekce pro reprezentaci datových tříd uzlů a hran, a také nastavit některé řetězcové vlastnosti pro rozpoznávání vztahů mezi uzly a hranami v našem modelu – viz kód níže.

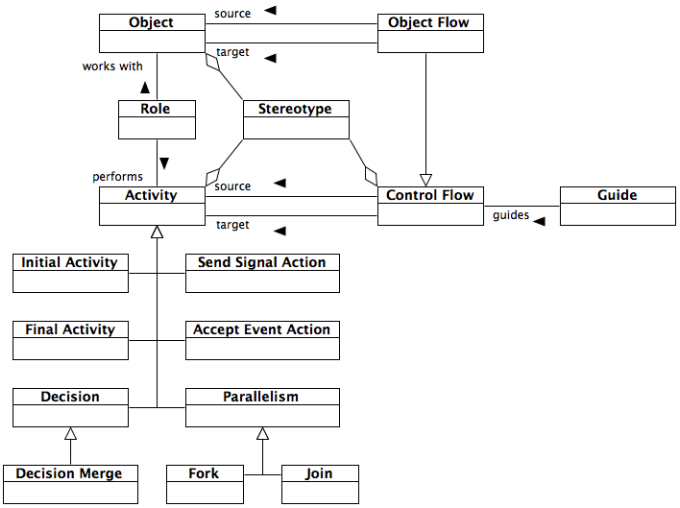
Samotný diagram pro mou implementaci nestačí – vytvořil jsem si dále dvě palety (**Northwoods.Goxam.Palette**). Tato třída je specializací předchozí třídy, takže její instanci také musíme přiřadit model, který obsahuje tentokrát pouze kolekci uzlů. Obě tyto palety slouží pro drag-and-drop uzlů do diagramu. Jedna paleta drží namapované elementy našeho procesu (konkrétní mapovací pravidla dále v textu) a druhá uzly pro podporu toku – větvení, rozhodovací bloky, počáteční a koncová aktivita.

```
var model = new GraphLinksModel<ActivityDiagramNodeData, String, String,
                                                ActivityDiagramLinkData>();
model.NodesSource = new ObservableCollection<ActivityDiagramNodeData>();
model.LinksSource = new ObservableCollection<ActivityDiagramLinkData>();
model.Modifiable = true;
model.NodeKeyPath = "Key";
model.LinkFromPath = "From";
model.LinkToPath = "To";
model.NodeCategoryPath = "Category";
model.NodeIsGroupPath = "IsSubGraph";
model.GroupNodePath = "SubGraphKey";
diagram.Model = model;
diagram.AllowDrop = true;
palette.Model = new GraphLinksModel<ActivityDiagramNodeData, String, String,
```

```
                                                    ActivityDiagramLinkData>();
flowPalette.Model = new GraphLinksModel<ActivityDiagramNodeData, String,
                                        String, ActivityDiagramLinkData>();
flowPalette.Model.NodesSource = new List<ActivityDiagramNodeData>()
       { /* uzly pro podporu toku */ }
```

3.5.2 Definice tvarů jednotlivých typů uzlů a hran

Northwoods.GoWPF hojně využívá data bindingu, takže není nic lepšího, než si všechny tvary uzlů vytvořit jako vzory (<***DataTemplate***>), kde x:Key nabývá hodnot vlastnosti Category zmíněných výše. Tedy nadefinovat si je jako statické zdroje (v elementu <***Page.Resources***> <***go:DataTemplateDictionary***>) a pomocí postupů popsaných v dokumentaci vytvořit jednotlivé tvary a například <***TextBlock***> pomocí data bindingu navázat na některou z vlastností, které jsme nadefinovali v modelové třídě v předchozí podkapitole. Meta-model popisující vztahy mezi uzly a hranami je na Obrázek 10. V mém případě je *Swimlane* ekvivalentem Role a kromě tohoto rozdílu jsem ještě přidal typ uzlu pro poznámku (*Note*) a typ hrany pro ukotvení poznámky (*Anchor*). Ukázka definice vzoru pro uzel *Activity* je pod obrázkem 11.



Obrázek 10 : Meta-model UML diagramu aktivit

```
<DataTemplate x:Key="Activity">
    <go:NodePanel Sizing="Auto" MaxWidth="200"
                  go:Part.SelectionElementName="Shape">
        <Rectangle x:Name="Shape" RadiusX="15" RadiusY="15"
                   Stroke="Black" StrokeThickness="1"
                   Fill="{Binding Path=Data.Color, Converter={StaticResource
                                                    theStringBrushConverter}}" />
        <StackPanel HorizontalAlignment="Center"
                    VerticalAlignment="Top" Margin="5,8,5,12">
            <TextBlock Text="{Binding Path=Data.Stereotype}"
                       Style="{StaticResource TextBlockStyle}"/>
            <TextBlock TextWrapping="WrapWithOverflow"
                       Text="{Binding Path=Data.Name}"
                       Style="{StaticResource TextBlockStyle}"/>
        </StackPanel>
    </go:NodePanel>
</DataTemplate>
```

3.5.3 Úprava nástrojů pro účely diagramu aktivit

Dalšími třídami, jejichž implementaci jsem musel lehce upravit, jsou třídy nástrojů (Tools), které reagují na uživatelský vstup. Všechny tyto třídy jsem předal instanci diagramu v XAML kódu tohoto diagramu – jsou vytvořeny automaticky v rámci metody ***InitializeComponent*** v kódu stránky diagramu aktivit.

SwimlaneDraggingTool specializuje **Northwoods.GoXam.DraggingTool**. Důvodem úpravy tohoto nástroje je, že uzel Swimlane je datově reprezentován jako uzel skupiny (Group), do které můžeme přidávat uzly. Bylo tedy nutné zajistit, aby do Swimlane nebylo možno přidávat další Swimlane. Dále bylo potřeba vytvořit funkčnost, která by komunikovala s modelem procesu a z tohoto modelu zjistila, zda přidávaný uzel může skutečně (podle mapovacích pravidel) být přidán do této Swimlane. Tohoto jsem docílil překrytím metody ***IsValidMember***.

Pro validaci nově vytvořených hran jsem musel upravit implementaci u dalších dvou nástrojů. **ActivityDiagramLinkingTool** (specializace **Northwoods.GoXam.LinkingTool**) a **ActivityDiagramRelinkingTool** (specializace **Northwoods.GoXam.RelinkingTool**). Bylo potřeba vytvořit validační nástroj, který by zakázal vytváření reflexivních hran. Tohoto jsem docílil překrytím metody ***IsValidLink*** u obou těchto nástrojů. Implementace v prvním i druhém jmenovaném nástroji je totožná. Nástroje jsem musel vytvořit dva kvůli tomu, že jeden se stará o validaci nových hran, zatímco druhý má na starosti validaci hran, které „přepojujeme“ z jednoho uzlu ke druhému. **SimpleLabelDraggingTool** je nástroj pro změnu pozice popisku u hrany, který specializuje **Northwoods.GoXam.DiagramTool.**

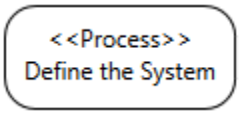
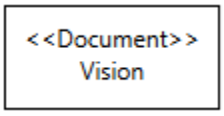
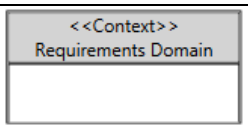
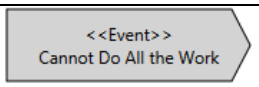
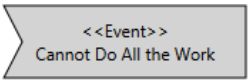
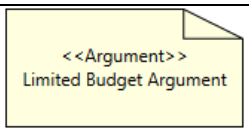
Kromě těchto nástrojů bylo potřeba vytvořit několik event handlerů pro diagram a paletu (metod pro reagování na události) a zaregistrovat je k patřičným událostem. Handler ***diagram_ExternalObjectsDropped*** slouží pro vkládání uzlu – pokud je zakázáno vytváření duplikátních uzlů, tak se stejný uzel nepřidá podruhé. Dále handlery ***palette_SelectionChanged*** a ***diagram_SelectionChanged*** pro zobrazování popisu vybraného uzlu, handler ***ADElementsListbox_SelectionChanged*** pro filtraci palety podle vybraného typu uzlu a nakonec velmi podstatný handler ***diagram_LinkDrawn***, který volá metodu, která se stará o vybrání správné kategorie hrany v závislosti na tom, mezi jakými uzly hrana vznikla (toto se projeví na vizuální reprezentaci hrany), dále volá metodu instance třídy **UML4UPMM** ***CheckADRelationship***, která podle mapovacích pravidel zjistí, zda je možno mezi dvěma uzly, které tato hrana spojuje, toto spojení povoleno. Pokud je spojení povoleno a je vypnutá validace s modelem procesu, tak se hrana vytvoří s popisem, který obsahuje typ vztahu z UPMM, díky kterému je tato hrana povolena. Pokud je tento vztah datově zachycen v modelu procesu, tak má hrana barvu černou, pokud tento vztah není zachycen, tak červenou. V případě, že je spojení povoleno a validace zapnutá, tak nástroj povolí propojení pouze těch uzlů, které mají odpovídající data v modelu procesu.

Nakonec se zde nachází ještě několik metod - pro ukládání a načítání diagramu z .kotr XML souboru, který je možné otevřít pouze v tomto nástroji, a metoda pro uložení PNG obrázku diagramu. Při načítání diagramu je kontrolováno, jestli diagram a OWL profil procesu odpovídají – jestliže některý prvek diagramu má IRI, které se nenachází v profilu, tak nelze diagram otevřít.

3.5.4 Mapování UML4UPMM

Srdcem veškerého mapování je třída **UML4UPMM**. Tato třída obsahuje metody, které porovnávají diagram podle definovaných pravidel s modelem softwarového procesu zachyceným pomocí **List<UPMM.SoftwareProcessElement>** a rozhodují, zda provedená úprava diagramu (vytvoření hrany mezi uzly s určitými daty – ne už jen obecně mezi uzly dané kategorie) je na základě několika postupů popsaných níže povolená úprava. V konstruktoru je naplněn slovník zmíněný níže a předán instanci model softwarového procesu.

V této třídě je vytvořený slovník s mapovacími pravidly pro vytváření palety uzlů, který je naplněn v konstruktoru. Slovník je typu **Dictionary<*UPMM.UPMMTypes*, *List<String>*>**, kde UPMM.UPMMTypes je výčtový typ elementu UPMM a List<String> obsahuje kategorie uzlů (Object, Activity, Swimlane…), které jsou s tímto typem elementu spojeny a je tedy možné vložit element do palety jako uzly těchto kategorií. S tímto slovníkem pracuje metoda ***MapUPMMToActivityDiagramNodeData***, jejímž výstupem je list s namapovanými uzly pro paletu diagramu. Pravidla pro mapování uzlů ukazuje Tabulka 1. (5)

Typ uzlu	**Typ UPMM**	**Tvar uzlu**
Activity	Task, Process, Alternative	
Object	Role, Group, Competence, Law, Object, Entity, Information, Artifact, Material, Document, Resource, Human Resource, Inanimate Resource	
Swimlane	Context	
Send Signal Action	Event, Issue	
Accept Event Action	Event, Issue	
Note	Goal, Intention, Argument	

Tabulka 1 : Mapovací pravidla pro vytváření uzlů (UML4UPMM)

Velmi důležitá metoda je `public string CheckADRelationShip(string sourceIRI, string targetIRI, bool validation, out string color).` Návratovou hodnotou této metody je řetězec s názvem vztahu, který je v UPMM mezi typy těchto elementů definován, popř. null, pokud definován vztah není nebo je zaplá validace podle dat modelu procesu a tato hrana nemá v modelu ekvivalentní data. Při návratu hodnoty null se hrana nevytvoří. Tato metoda má 4 parametry:

- sourceIRI – řetězec s IRI elementu, ze kterého má vést hrana
- targetIRI – řetězec s IRI elementu, do kterého má vést hrana
- validation – bool parametr, je pravdivý, když je zaplá validace podle modelu procesu
- color – výstupní řetězec, který nese barvu hrany – buď „Red“ (červená) nebo „Black“ (černá), podle toho, jestli je hrana datově obsažena v modelu procesu

Povolené vztahy jsou uvedeny v Tabulka 2 a vztahují se i na všechny elementy, které jsou specializací těchto tříd. Z důvodu přílišné komplexnosti implementace paralelních a přerušujících se procesních kroků jsem se rozhodl kontrolu těchto dvou vztahů neimplementovat a povolovat ji automaticky. (5)

Třída zdrojového UPMM elementu	**Vztah a třída cílového UPMM elementu**
Alternative	**Contributes to** – Process Step
Argument	**Supports** – Alternative **Objects to** – Alternative
Competence	**Checks** – Law
Entity	**Mandatory input** – Task **Optional input** – Task **Input** – Task **Results in** – Goal
Event	**Activates** – Event
Intention	**Concretizes** – Goal
Issue	**Has response** – Alternative
Law	**Controls** - Task
Object	**Is used in** - Process
Process Step	**Decides** – Alternative **Raises** – Issue **Precedes** – Process Step **Is followed by** – Process Step

Process	**Sends signal** – Event **Realizes** – Goal
Resource	**Provides** – Competence **Plays** – Role **Processes** – Entity
Role	**Performs** – Task **Selects** – Alternative **Specifies** – Competence **Is responsible for** - Object
Task	**Sends signal** – Event **Output** - Entity

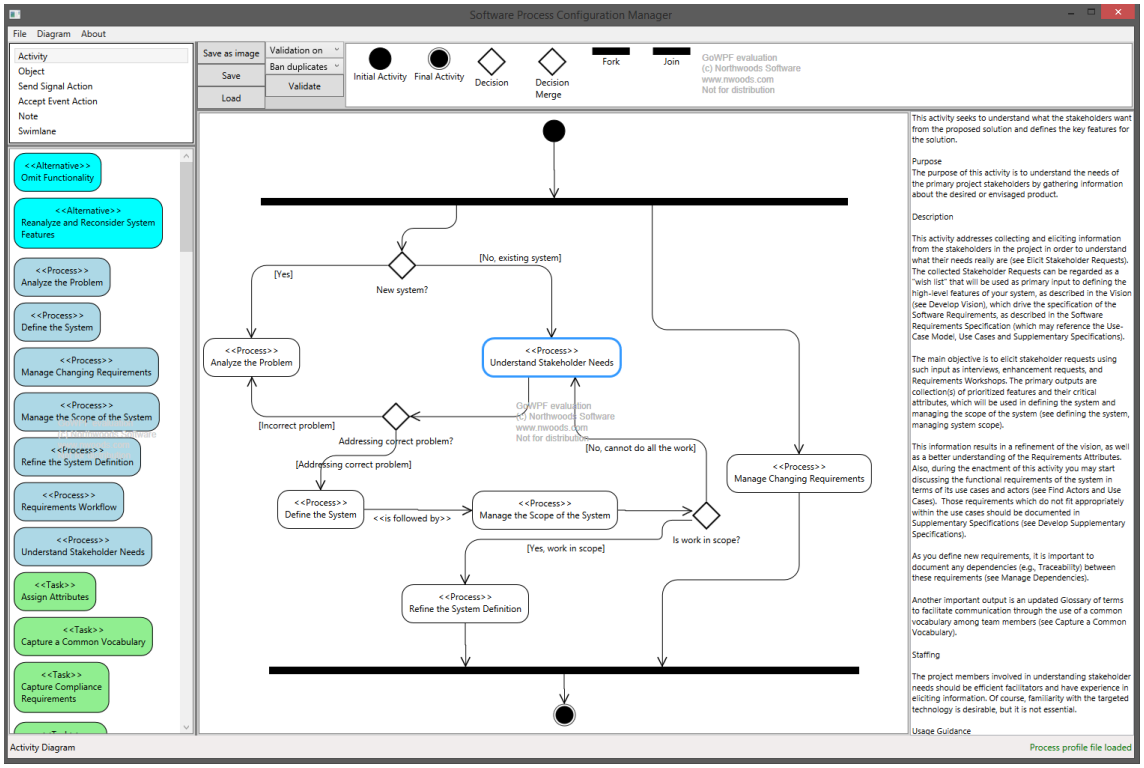
Tabulka 2 : Povolené vztahy mezi elementy UPMM v diagramu aktivit (UML4UPMM)

3.6 Software Process Configuration Manager

Výsledný nástroj implementuje zpracování OWL báze obsahující model softwarového procesu, transformaci elementů do palety diagramu, drag-and-drop modelování, modelování pouze povolených vztahů z UPMM, validaci vytvořeného diagramu k modelu procesu, uložení diagramu do obrázku ve formátu PNG a také ukládání a načítání diagramu z formátu .kotr, což je XML obsahující serializovaný model diagramu (uzly a hrany) včetně pozice a velikosti uzlů a barvy hran. Je možno uložit i diagram, který obsahuje i spojení, ke kterým nejsou v modelu procesu žádná data (stále jen u povolených vztahů). Při vypnuté validaci s modelem procesu se hrany, ke kterým nejsou odpovídající data v modelu procesu, zbarví červeně. Totéž platí pro přidávání uzlů do Swimlane, kde je kromě červeného okraje uzlu ještě přidána hláška „UNSUPPORTED“. Lze načíst jen diagram, jehož všechny mapovatelné prvky jsou také obsaženy v aktuálně načteném modelu procesu z OWL. Pokud tomu tak není, je uživateli oznámeno, že diagram neodpovídá modelu z OWL – každý diagram je tedy závislý na načteném modelu z OWL a nelze jej upravovat při použití jiného OWL modelu. Dosažení tohoto chování nástroje bylo nezbytné – diagram slouží jako konfigurace konkrétního procesu. Ukázková konfigurace existujícího procesu z tohoto nástroje je v Příloha D:

Menu nástroje obsahuje několik podmenu – File, Diagram, About. Ve File menu je možnost (Start page) zobrazit úvodní stránku se stručným návodem, možnosti pro připojení a odpojení OWL modelu procesu (Load profile, Unload profile), v menu Diagram -> UML je možnost Activity Diagram, která zobrazí konfiguraci profilu pomocí UML diagramu aktivit.

Ve status baru se nachází informace o tom, na které stránce se uživatel právě nachází a zda je načtený profil OWL s modelem procesu.



Obrázek 11 : Software Process Configuration Manager

Obrázek 11 : Software Process Configuration Manager ukazuje uživatelské rozhraní mnou vytvořeného nástroje. Vlevo nahoře je komponenta pro přepínání zobrazených elementů v paletě vlevo dole. Tato paleta obsahuje elementy modelu načteného procesu v podobě uzlů, barevně odlišené podle zdrojové třídy elementu v modelu. Nad diagramem jsou tlačítka pro ukládání a načítání diagramu a možnosti pro validaci diagramu s modelem a možnost pro vytváření duplikátních uzlů z palety vlevo dole. Vedle těchto tlačítek je paleta obsahující uzly pro diagram aktivity, které se nemapují za elementy modelu procesu. Vpravo od diagramu můžeme vidět definici/dokumentaci k prvku, který je vybrán v paletě vlevo nebo v diagramu. V tomto případě můžeme vybraný prvek vidět v diagramu zvýrazněný modře. Pokud dokumentace chybí, tak se dokumentace schová. Posledním prvkem je samotný diagram – uzly se vytváří pomocí drag-and-drop z palet. Hranu lze vytvořit najetím myši na okraj uzlu (kurzor myši se změní na ruku) a přetažením k jinému uzlu. Diagram reaguje na klasické klávesové zkratky – Ctrl-A pro výběr všech prvků, Ctrl+klik pro výběr více prvků, Delete smaže vybrané prvky. Popisky u hran lze přesouvat myší. Popisky u hran a popisek u rozhodovacího bloku jde editovat po dvojkliku na text.

4 Závěr

Cílem této práce bylo analyzovat, navrhnout a implementovat nástroj, který na platformě .NET umožní konfigurovat existující softwarové procesy, primárně behaviorální perspektivu softwarového procesu pomocí aktivitního diagramu UML.

Při řešení jsem se nejprve zaměřil na platformu Eclipse GMF, protože nabízí velký potenciál pro tvorbu nástroje založeného na metamodelu a zaměřeného na vizuální modelování a odstínila mě od problému tvorby grafického jádra a ručního psaní kódu metamodelů. Prvním pilířem v implementaci bylo nadefinovat metamodel softwarového procesu a metamodel aktivitního diagramu na platformě Eclipse EMF. Pro vytvoření těchto metamodelů jsem musel pochopit architekturu řešení na této platformě a naučit se definovat modely v metamodelu EMF, tzv. ecore. Na tomto ecore modelu zakládá platforma GMF, ve které jsem se snažil navrhnout a implementovat nástroj pro vizuální modelování, avšak při implementaci funkčnosti zaměřené na import znalostní báze a namapování a importování prvků procesu do palety diagramu jsem narazil na značné problémy způsobené celkovou náročností vývoje pluginu pro Eclipse pomocí GMF, jelikož se tento vývoj děje na základě vytvoření několika modelů pomocí grafického průvodce a následném vygenerování částí kódu a toto se několikrát opakuje. Takto vygenerovaný kód se ručně upravuje, takže je vývoj náchylný na drobné chyby, což u jednoduchých editorů diagramů nevadí, ale při tvorbě komplexního nástroje stojí každá menší chyba mnoho hodin procházení generovaného kódu. Co se týká dokumentace, ta není příliš obsáhlá a je velmi nepřehledná a v mnoha případech obsahuje nefunkční odkazy na oficiálním webu, takže jsem se po konzultaci s vedoucím práce rozhodl pro změnu platformy na platformu. NET.

Implementace na platformě .NET s sebou přinesla určitá rizika, jako přenesení knihoven z Javy na .NET a vyhledání vhodné knihovny pro vykreslování diagramů, která by byla pro akademické účely zdarma. Obě rizika se mi povedlo vyřešit a mohl jsem se tak zdokonalit na poli vývoje WPF aplikací, práci s OWL ontologiemi, použitím kódu napsaného pro jinou platformu, použití knihovny GoXam a verzování kódu v systému TFS. Všechny tyto technologie mně byly před touto prací neznámé. Implementací výše popsané funkčnosti byly všechny dříve definované cíle splněny. Tato bakalářská práce byla vytvořena jako podpůrná práce k disertační práci Ing. Michaela Alexandera Košinára s tématem Knowledge Support for Software Processes. (5)

Architektura řešení je navržena obecnou metodou, meta-model softwarového procesu byl naimplementován kompletní, i když vizuální modelování behaviorální perspektivy nevyužívá všechny jeho prvky. Budoucí práce může být zaměřena na implementaci strukturální perspektivy pomocí třídního diagramu, funkční perspektivy pomocí use-case diagramu, popř. rozšíření aktuálního řešení o paralelní běh procesních kroků. Za zmínku také stojí možnost vytvoření mapovacích pravidel pro jiné notace vizuálního modelování, jako BPMN nebo EPC. Alternativní cestou vývoje by mohlo být vynechání doménového modelu a tím pádem umožnit práci s různými verzemi ontologie a dynamické mapování podle verze.

Použitá literatura

1. **Vondrák, Ivo.** Úvod do softwarového inženýrství. [Online] 2002. [Citace: 10. 7 2015.] http://vondrak.cs.vsb.cz/download/Uvod_do_softwaroveho_inzenyrstvi.pdf.

2. **Schwaber, Ken a Sutherland, Jeff.** ™Průvodce SCRUMEM. [Online] Srpen 2013. [Citace: 10. 7 2015.] http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-CS.pdf.

3. **Vondrák, Ivo.** METODY BYZNYS MODELOVÁNÍ pro kombinované a distanční studium. [Online] 2004. [Citace: 10. 7 2015.] http://vondrak.cs.vsb.cz/download/Metody_byznys_modelovani.pdf.

4. **W3C.** OWL 2 Web Ontology Language. [Online] 2012. [Citace: 10. 7 2015.] http://www.w3.org/standards/techs/owl#w3c_all.

5. **Košinár, Michael, Alexander.** *Knowledge Support for Software Processes.* Ostrava : Ostrava: VŠB-TUO, Fakulta elektrotechniky a informatiky, Katedra informatiky. Disertační práce, 2015.

6. **OMG.** Unified Modeling Language ™ (UML®). [Online] 2011. [Citace: 10. 7 2015.] http://www.omg.org/spec/UML/.

7. **W3C.** Extensible Markup Language (XML). [Online] 2009. [Citace: 10. 7 2015.] http://www.w3.org/standards/techs/xml#w3c_all.

8. **Microsoft.** .NET Framework 4.5. *msdn.microsoft.com.* [Online] [Citace: 10. 7 2015.] https://msdn.microsoft.com/cs-cz/library/w0x726c2(v=vs.110).aspx.

9. —. Úvod do WPF. *msdn.microsoft.com.* [Online] [Citace: 10. 7 2015.] https://msdn.microsoft.com/cs-cz/library/aa970268(v=vs.110).aspx.

10. **Northwoods Software.** Introduction to GoXam. *www.goxam.com.* [Online] [Citace: 10. 7 2015.] http://www.goxam.com/2.1/GoXamIntro.pdf.

11. **Horridge, Mattew a Bechhofer, Sean.** The OWL API: A Java API for OWL Ontologies. *Semantic Web Journal.* Special Issue on Semantic Web Tools and Systems, 2011, stránky 11-21.

12. **Palmisano, Ignazio.** The Rough Guide to the OWL API: a tutorial. *owlapi.sourceforge.net.* [Online] 5. 6 2011. [Citace: 10. 7 2015.] http://owlapi.sourceforge.net/owled2011_tutorial.pdf.

13. **Kuba, Martin.** OWL 2 and SWRL Tutorial. *ics.muni.cz.* [Online] [Citace: 10. 7 2015.] http://dior.ics.muni.cz/~makub/owl/.

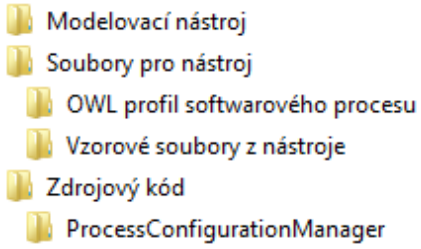
14. **Cognitum.** OWL API for .NET. *owlapinet.codeplex.com.* [Online] [Citace: 10. 7 2015.] https://owlapinet.codeplex.com/.

15. **Frijters, Jeroen.** IKVM.NET Home Page. *ikvm.net.* [Online] [Citace: 10. 7 2015.] http://www.ikvm.net.

Seznam příloh

Součástí BP je CD.

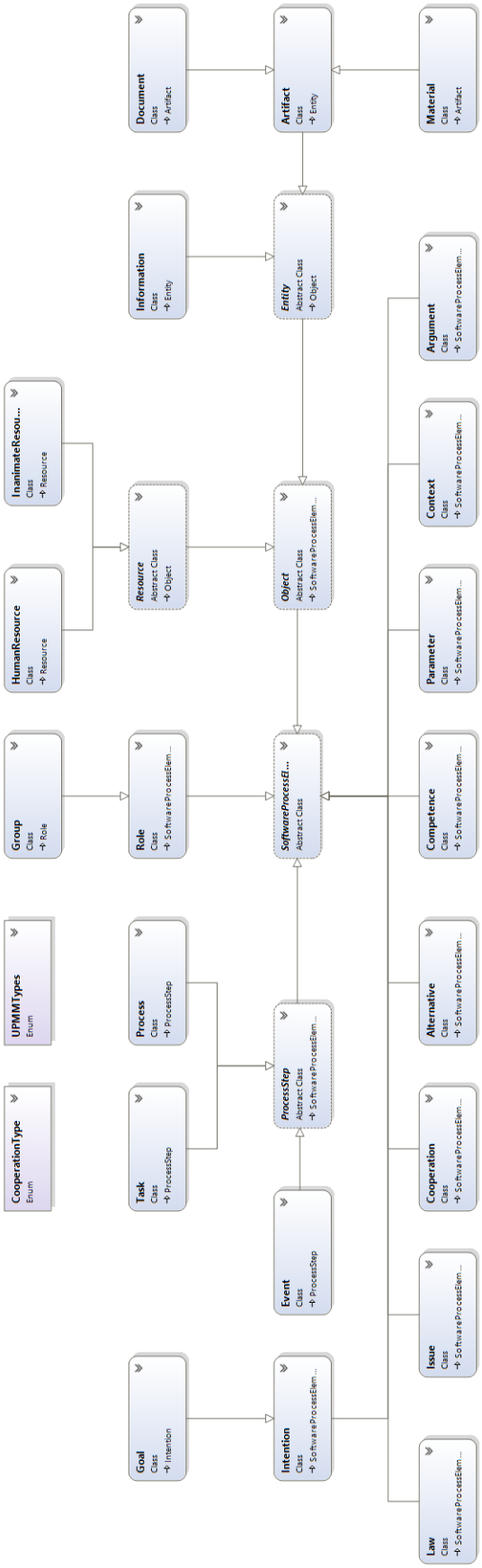
Adresářová struktura přiloženého CD:



Příloha A: *Struktura OWL dokumentu*

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://www.kosinar.me/ontologies/RequirementsDisciplineProfile"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 ontologyIRI="http://www.kosinar.me/ontologies/RequirementsDisciplineProfile">
    <Prefix name="" IRI="http://www.w3.org/2002/07/owl#"/>
    <Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
    <Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
    <Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
    <Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
    <Import>http://www.kosinar.me/ontologies/UnifiedProcessMetaModel</Import>
    <Declaration>
        <NamedIndividual IRI="#Prioritize_Use_Cases"/>
    </Declaration>
    <ClassAssertion>
        <Class IRI="http://www.kosinar.me/ontologies/
                       UnifiedProcessMetaModel#Task"/>
        <NamedIndividual IRI="#Prioritize_Use_Cases"/>
    </ClassAssertion>
    <DataPropertyAssertion>
       <DataProperty IRI="http://www.kosinar.me/ontologies/
                     UnifiedProcessMetaModel#isAtomicStep"/>
       <NamedIndividual IRI="#Prioritize_Use_Cases"/>
       <Literal datatypeIRI="&xsd;boolean">true</Literal>
    </DataPropertyAssertion>
    <AnnotationAssertion>
        <AnnotationProperty abbreviatedIRI="rdfs:label"/>
        <IRI>#Present_the_Use_Case_Model_in_Diagrams</IRI>
        <Literal xml:lang="en" datatypeIRI="&rdf;PlainLiteral">
            Present the Use-Case Model in Diagrams
        </Literal>
    </AnnotationAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="http://www.kosinar.me/ontologies/
                                UnifiedProcessMetaModel#hasStep"/>
        <NamedIndividual IRI="#Prioritize_Use_Cases"/>
        <NamedIndividual IRI="#Evaluate_Your_Results"/>
    </ObjectPropertyAssertion>
    ...
</Ontology>
```

Příloha B: *Kompletní hierarchie dědičnosti tříd doménového modelu UPMM*



Příloha C: *Metody pro převod individuí z OWL ontologie na instance třídy domény*

```
private static void CreateLaws(OWLClass owlClass, List<SoftwareProcessElement>
softwareProcess)
{
    List<OWLNamedIndividual> individuals = GetIndividuals(owlClass);
    foreach (OWLNamedIndividual individual in individuals)
    {
        string IRI;
        string name;
        string description;
        GetBasicPropeties(individual, out IRI, out name, out description);

        int competenceLevel = 0;
        int.TryParse(GetConcreteDataPropertyValue(individual,
                    "competence_level"), out competenceLevel);
        string regulationCode = GetConcreteDataPropertyValue(individual,
                                                             "regulationCode");

        softwareProcess.Add(new Law(IRI, name, description, competenceLevel,
                                                             regulationCode));
    }
}

private static void CreateLawRelationships(OWLClass owlClass,
List<SoftwareProcessElement> softwareProcess)
{
    List<OWLNamedIndividual> individuals = GetIndividuals(owlClass);

    foreach (OWLNamedIndividual individual in individuals)
    {
        string individualIRI = individual.getIRI().toString();
        UPMM.Law law = softwareProcess.Where(x => x.IRI == individualIRI &&
              x.GetType() == typeof(UPMM.Law)) .Select(x => x as UPMM.Law)
              .FirstOrDefault();

        if (law == null)
            continue;

        foreach (string iri in GetConcreteObjectPropertyValues(individual,
                                                             "controls"))
        {
            UPMM.Task task = softwareProcess.Where(x => x.IRI == iri)
                                    .Select(x => x as UPMM.Task).FirstOrDefault();

            if (law.Controls.Contains(task))
                continue;
            else
            {
                law.Controls.Add(task);
                task.IsControlledBy.Add(law);
            }
        }
```

```
foreach (string iri in GetConcreteObjectPropertyValues(individual,
                                                       "isCheckedBy"))
        {
            UPMM.Competence competence = softwareProcess.Where(x => x.IRI == iri)
                                .Select(x => x as UPMM.Competence).FirstOrDefault();

            if (law.IsCheckedBy.Contains(competence))
                continue;
            else
            {
                law.IsCheckedBy.Add(competence);
                competence.Checks.Add(law);
            }
        }
    }
}
```

Příloha D: *Ukázková konfigurace procesu*