VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Analýza, návrh a realizace simulačního nástroje pro podporu softwarových procesů

Analysis, Design and Realization of Software Process Simulation Tool

2014 Jan Czopik

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Czopik**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Analýza, návrh a realizace simulačního nástroje pro podporu softwarových procesů
Analysis, Design and Realization of Software Process Simulation Tool

Zásady pro vypracování:

Cílem práce je analyzovat a zvolit vhodnou metodiku pro simulaci softwarových procesů včetně návrhu a realizace nástroje pro tyto simulační účely. Výstupem práce bude implementovaný simulačí software.

Specifické cíle budou:
1. Student nastuduje oblast formalizačních nástrojů vhodných pro business procesy, převážně pak pro softwarové procesy.
2. Student nastuduje oblast metod vhodných pro simulaci podnikových procesů v softwarových společnostech.
3. Na základě vybraných metod z ad 1 a 2 student analyzuje a navrhne softwarový nástroj pro simulaci softwarových procesů (s možností využití již existujících softwarových knihoven či konfiguračních nástrojů jako např. BP Studio).
4. Student aplikuje a otestuje výslednou implementaci na základě ukázkových dat a vyhodnotí výsledky simulací.

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. Work ow 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 diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

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

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 5. května 2014 ..... Jan Čapek

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

V Ostravě 5. května 2014 ..... Jan Čapek

Rád bych na tomto místě poděkoval za pomoc a cenné rady všem, bez kterých by tato práce nevznikla.

Zejména chci poděkovat vedoucímu diplomové práce Ing. Michaelu Alexandrovi Košinárovi za veškerou pomoc a konstruktivní kritiku, lektorům Ing. Jakubovi Štolfovi a Ing. Svatopluku Štolfovi, Ph. D. za odborné konzultace a prof. Ing. Ivo Vondrákovi, CSc. za úvodní konzultaci, poskytnutí studijních materiálů a zdrojových kódů BP studia.

Abstrakt

Moderní doba žádá moderní přístup k vývoji aplikací. Pomoci nám mohou softwarové prostředky, které umožňují podpořit vývoj aplikací přesnou definicí a následnou správou softwarového procesu. Díky těmto prostředkům jsme schopni proces jednoduše vymodelovat, odsimulovat daný model a použít výsledky simulace k analýze modelu procesu (jeho efektivity, správnosti).

Tato diplomová práce se zabývá kompletní realizací právě takového softwarového nástroje. Cílem diplomové práce je představení jednotlivých modelovacích a simulačních metod a jejich následné zhodnocení.

Vybrané metody budou poté implementovány v rámci výše zmíněného softwarového nástroje.

V závěru budou k dispozici výsledky simulací předem vymodelovaných ukázkových procesů.

Klíčová slova: softwarový proces, metody modelování, metody simulace, Petriho sítě, UML, Qt, c++

Abstract

Modern times asks modern approaches towards application development. We can benefit from using software tools which can support application development by modeling and managing our software processes. Thanks to these tools we are able to seamlessly model and simulate model of a process and then use the simulation results to analyze the model (its efficiency, correctness).

This diploma thesis deals with complete realization of such software tool. The goal of the thesis is to introduce individual modeling a simulation methods and evaluate them.

Chosen methods will be implemented as part of the above mentioned software tool.

There will be results of simulations of beforehand modeled sample processes available in the conclusion.

Keywords: software process, modeling methods, simulation methods, petri nets, UML, Qt, c++

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
BFS	–	Breadth First Search
BPEL4WS	–	Business Process Execution Language For Web Services
BPM	–	Business Process Modeling
BPMI	–	Business Process Management Initiative
BPMN	–	Business Process Modeling Notation
c++	–	programovací jazyk
CPN	–	Coloured Petri Nets
CPN ML	–	Skriptovací jazyk CPN
DAML	–	DARPA Agent Markup language
DFS	–	Depth First Search
DYNAMO	–	DYNAmic MOdels
EPC	–	Event-driven process chain
GE	–	General Eletric
GUI	–	Graphical User Interface
OWL	–	Ontology Inference Layer
OWL	–	Web Ontology Language
Qt	–	Qt knihovna
SADT	–	Structured Analysis and Design Technique
Scrum	–	Agilní metoda vývoje
SEI	–	Software Engineering Institute
SIMPLE	–	Simulation of Industrial Management Problems with Lots of Equations
SPM	–	Software Process Modeler
UI	–	User Interface
UML	–	Unified Modeling Language
W3C	–	World Wide Web Consortium
XMI	–	XML Metadata Interchange

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Tato diplomová práce se zabývá kompletní realizací (od analýzy až po implementaci) modelovacího a simulačního nástroje, určeného výhradně pro softwarový proces, který jeho uživatelům umožní vymodelování dynamického aspektu softwarového procesu. Následně pak na základě vytvořených modelů bude možné simulovat průběh daného procesu pro zjištění jeho efektivnosti a správnosti. Diplomová práce se také zabývá porovnáním a zhodnocením nejpoužívanějších modelovacích a simulačních metod.

Text diplomové práce je rozdělen na tři části. Část teoretickou, která obsahuje stručné představení vybraných modelovacích a simulačních metod, dále část prakticky zaměřenou, věnující se především popisu implementovaného softwarového nástroje spolu s metodou převodu UML diagramu aktivit na CPN a pak závěr, který představí výsledky práce.

Začneme kapitolou, která popíše softwarový proces, jeho definici a účel.

Další kapitola vás obeznámí s metodami modelování softwarového (respektive byznys) procesu.

Následně si popíšeme vybrané druhy simulačních metod, opět je porovnáme a zvolíme nejvhodnější z nich.

Následující kapitola bude shrnutím předešlých kapitol popisujících metody modelování a simulace. V rámci této kapitoly vybereme konkrétní metody pro následné praktické použití a předložíme hypotézu.

Po těchto teoretických kapitolách přejdeme k popisu našeho modelovacího a simulačního nástroje. Budeme se zabývat analýzou požadavků, návrhem a implementací systému. Těmto kapitolám bude předcházet kapitola věnující se formalizaci vybrané semi-formální modelovací metody.

V kapitole zabývající se analýzou požadavků vám krátce představím způsob, jakým jsem při tomto procesu postupoval a výsledek, jehož jsem dosáhl.

Část popisující návrh systému obsahuje architektonický návrh softwaru, jeho komponent spolu s jejich popisem. Poněkud detailněji bude popsáno simulační jádro, jež bylo vyvinuto speciálně pro potřeby tohoto nástroje. Na konci této části také naleznete koncepty, na nichž je postaveno uživatelské rozhraní.

Následující kapitola popisuje implementační detaily jednotlivých komponent, použité algoritmy a vybrané snímky uživatelského rozhraní. Samozřejmostí je krátký popis technologie, která byla použita při implementaci tohoto nástroje. Naleznete v ní také popis limitů simulačního jádra a vytvořeného softwaru obecně.

Na závěr krátce shrneme poznatky, které jsme získali při představení vybraných modelovacích a simulačních metod a podrobněji se podíváme na výsledky simulací ukázkových procesů. V závěru bych také rád rozvedl náměty na zlepšení tohoto softwarového nástroje a také další možnosti vývoje.

2 Softwarový proces

Definice softwarového procesu (dle [Von02]) zní následovně:

Definice 2.1 *Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla.*

Z definice nemusí být na první pohled zřejmé, že:

- *krokem* je míněna aktivita nebo další proces (podproces)
- aktivity či podprocesy můžou probíhat v čase současně

V závislosti na využití a míře implementace softwarového procesu lze konkrétní společnosti hodnotit dle stupnice, která odráží vyspělost dané společnosti. Tato stupnice je rozdělena na tyto úrovně: (převzato z [Von02])

1. *Počáteční* - společnost nemá definován softwarový proces a každý projekt je řešen zvlášť a od začátku.

   Příklad: Společnost má vyvinout softwarový nástroj. Vytvoří se tým vývojářů a manažerů, kteří se domluví na tom, jak vývoj bude probíhat, jak budou probíhat jednotlivé činnosti vývoje, testování atp. Vývoj je ukončen a poté přijde jiná zakázka, znovu se sestaví tým (ne nutně stejný) a celý proces se opakuje. Úkoly jsou během projektu většinou přidělovány za běhu, což činí „proces“ netransparentní a nelze tak identifikovat konkrétní prvky.

2. *Opakovatelná* - v jednotlivých projektech byly nalezeny opakovatelné postupy a tyto postupy jsou poté použitelné v nových zakázkách.

   Příklad: Společnost získala zakázku na softwarový nástroj, je vytvořen tým, který na podobném projektu již v minulosti pracoval a není tak nutné identifikovat a koordinovat celé procesy znovu a od začátku.

3. *Definovaná* - softwarový proces je definován (a dokumentován) a jsou použity dříve identifikované procesy.

   Příklad: Společnost byla schopná identifikovat opakující se procesy při vývoji, tyto procesy byly zdokumentovány a nyní není nutné při začátku nového projektu tuto záležitost znovu řešit. Vývojáři se tak mohou soustředit například na výběr vhodných technologií pro realizaci zakázky.

4. *Řízená* - společnost je na základě metriky schopna daný softwarový proces řídit

   Příklad: Společnost zavedla metriky a zjistila, že projekty vyvíjené v létě se potýkají s problémy s fluktuací zaměstnanců (dovolené), konkrétně s nedostatkem vývojářů pro tzv. „peer review“.

5. *Optimalizovaná* - zpětná vazba získaná monitorováním procesu je následně použita pro jeho vylepšení

   Příklad: Proces je optimalizován na základě zjištění nedostatku vývojářů pro peer review v letním období. Proces se upraví tím způsobem, že pakliže není žádný vývojář schopný poskytnout peer review v daný čas, kód vyžadující toto review je vložen do fronty a jakmile se nějaký vývojář uvolní, může review poskytnout.

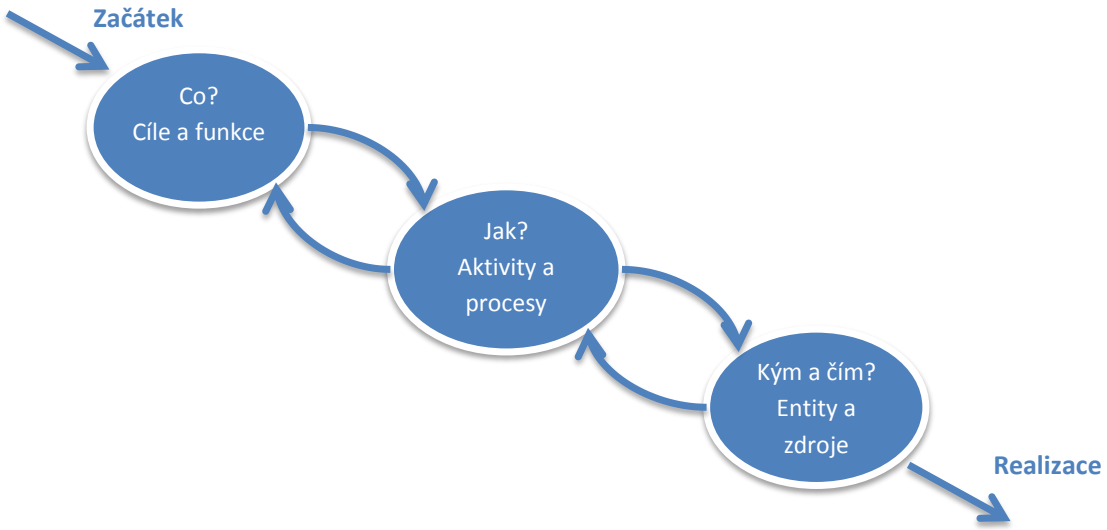
Cílem každé společnosti zabývající se vývojem softwaru by měla být brzká identifikace již používaných (byť nezdokumentovaných) softwarových procesů, jejich vylepšení a následná integrace v rámci vývoje produktu. Výsledkem začlenění dokumentovaných procesů bude zvýšená kvalita dodávaného produktu spolu se snížením provozních nákladů (čerpáno z [Kos10]). V této diplomové práci se zaměříme na softwarové procesy na třetí úrovni (tedy definované).

3 Metody modelování softwarového procesu

Pro převod softwarového procesu do digitální podoby, jež nám umožňuje s procesem lépe manipulovat, používáme počítačové modely. Modelem máme na mysli abstraktní reprezentaci procesu, která se dá použít k jeho validaci a verifikaci automatizovaným způsobem.

Korektně vymodelovaný proces nám poskytuje informace, sloužící k pochopení účelu jednotlivých aktivit, souvislostí mezi těmito aktivitami a prvků (zdroje, výstupy) použitých v daných procesech.

K tomuto účelu byla vyvinuta celá řada metod, které si v následujících kapitolách popíšeme. Než se zaměříme na detaily jednotlivých technologií, bylo by vhodné zmínit, že všechny tyto metody používají společný základ (Obrázek 1, převzat z [Von04]).



Obrázek 1: Postup návrhu byznys (software) procesu

Z diagramu jsou patrné tři základní přístupy, využívané v modelování procesů (dle [Von04]):

1. *Funkční přístup* zaměřený na funkce, jejich vstupy a výstupy (cíle).
2. *Přístup specifikací chování* stanovující provádění jednotlivých aktivit/procesů na základě vyhodnocování událostí a podmínek uvnitř modelu.
3. *Strukturální přístup* je zaměřen čistě na statický aspekt procesu modelující entity, jejich atributy a vazby mezi nimi.

3.1 Neformální a semi-formální metody

Tyto metody těží ze snadné srozumitelnosti takto vymodelovaných procesů, ať už za použití přirozeného jazyka pro vytvoření modelu (neformální) anebo modelu grafické povahy (semi-formální).

Dalo by se říci, že prvky modelu, které určují, zda model patří do skupiny modelů neformálních, semi-formálních či formálních, jsou dva. Konkrétně syntaxe a sémantika. U neformálních modelů není přesně vymezen ani jeden z těchto prvků, takže každý může chápat proces vymodelovaný tímto přístupem subjektivně, tedy odlišně. U semi-formálních přístupů je syntaxe vyjádřena dostatečně známou grafickou notací, stále však není přesně definována sémantika [Von04].

Následující podkapitoly také obsahují ukázky softwarového procesu *Implementace v metodologii scrum* definovaného ve [Štr13]. Tento proces bude následně použit i v praktické ukázce v kapitole 7.4.

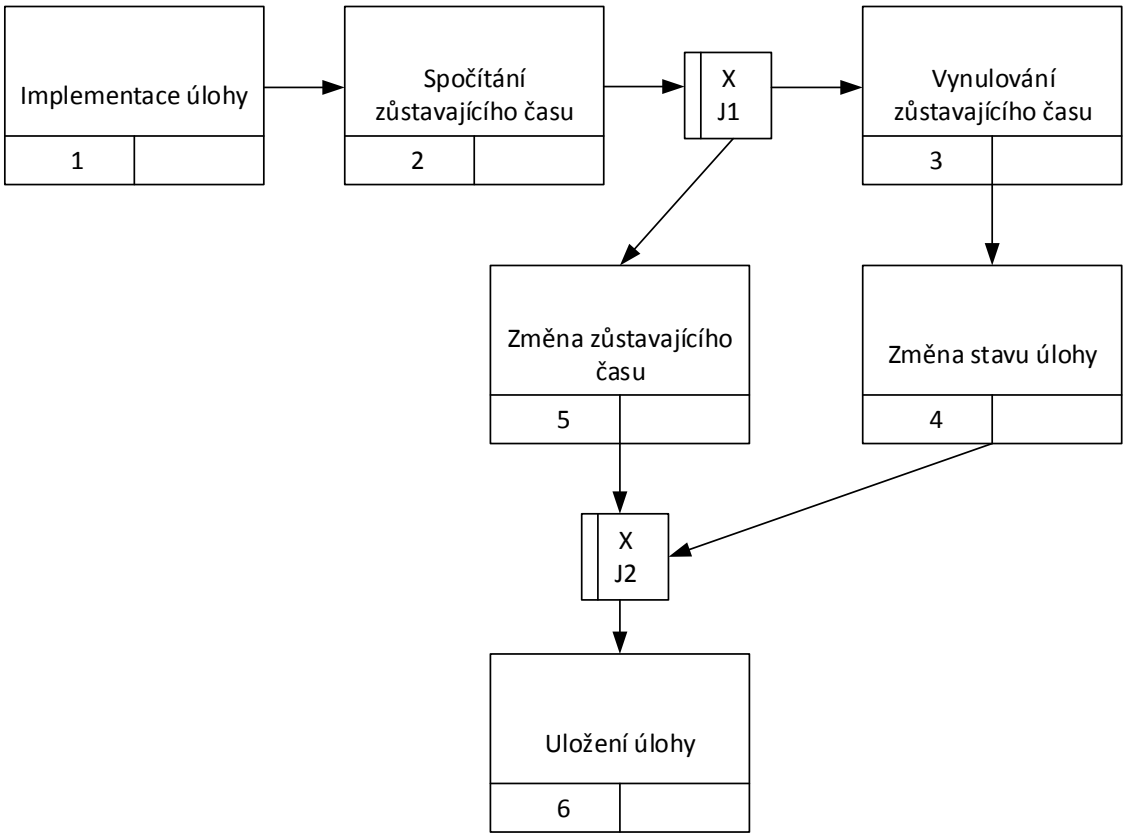
3.1.1 IDEF

Rodina modelovacích metod IDEF nabízí ucelený přístup a široké možnosti pro modelování byznys, respektive softwarových procesů. Jako základ slouží detailně vypracovaná syntaxe vycházející z grafického jazyka SADT. Pokud bychom měli vybrat konkrétní metody z početné rodiny skupiny IDEF použitelné pro modelování softwarových procesů, byly by to tyto (čerpáno z [IDE10]):

1. *IDEF0* je určená pro funkcionální modelování a je tedy vhodná pro aplikaci na funkční přístup.
2. *IDEF1* zaměřená na informační modelování, respektive modelování Entit a jejich vazeb. Tato metoda je použitelná pro strukturální přístup.
3. *IDEF3* slouží jako doplňková metoda k IDEF0 k modelování dynamiky systému (jeho chování). Pomocí této metody lze modelovat specifikaci chování.

Výhody této technologie tkví v *robustnosti* a v *uzrálosti*, IDEF se stal za dlouhá léta své existence de facto *podnikovým standardem*.

Z těchto faktů se však dají vyvodit také nevýhody této metody. Díky robustnosti a komplexitě se velké IDEF modely stávají *nepřehlednými* a *špatně udržovatelnými*. Stáří IDEF technologie ji také činí problematickou ve vztahu k integraci s nově vyvinutými softwarovými nástroji pro modelování a simulaci byznys (software) procesů [IDE10].



Obrázek 2: Příklad IDEF3 diagramu

3.1.2 BPMN

Tato technologie vznikla na popud neziskové organizace BPMI, která má za úkol podporovat rozšíření často používaných byznys procesů. Jedná se o grafickou notaci určenou pro modelování specifikace chování procesu, tedy jeho dynamiku. BMPI si dalo za úkol vytvořit moderní, jednoduchou a přehlednou modelovací metodu, která bude velice snadno chápana uživateli z byznys sféry (čerpáno z [Whi04]).

Tvůrci se rozhodli rozdělit symboly, používané při vytváření modelů do čtyř kategorií, aby je bylo možné později rozšířit bez výrazné změny základního vzhledu diagramu (dle [Whi04]):

1. *Plovoucí objekty* základní množina nejpoužívanějších symbolů obsahující *událost*, *aktivitu* a *bránu*.

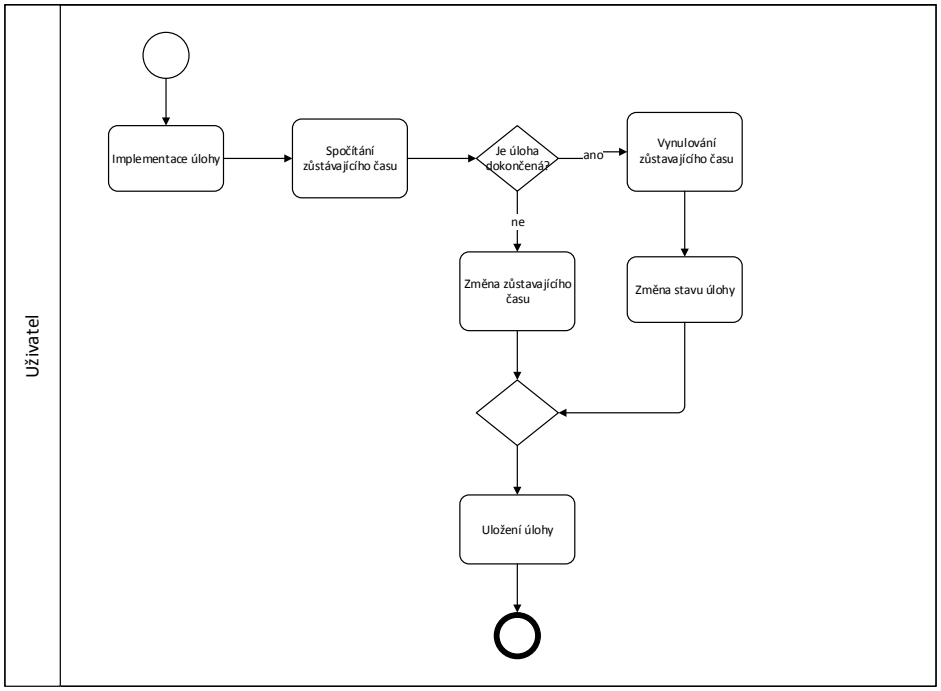
2. *Spojovací objekty* určené pro vytváření vazeb mezi symboly umožňující modelování sekvenčního přechodu, poslání zprávy mezi účastníky procesu a asociace objektů s artefakty.

3. *Plavecké dráhy* slouží pro lepší organizaci modelu a udržování pořádku a čitelnosti.

4. *Artefakty* rozšiřují základní notaci o širší možnosti modelování a k poskytnutí kontextu jednotlivých aktivit. BPMN také nabízí možnost deklarovat své vlastní artefakty.

Hlavní výhodou BPMN je, že obsahuje interní model, ze kterého je možno *automatizovaně vygenerovat* spustitelný *BPEL4WS model*, jenž je možné využít v implementaci procesu. Historicky se modely byznys procesů převáděly na exekuční modely ručně, což dávalo prostor k chybám [Whi04]. Rád bych také vyzdvihl *široké možnosti* této metody při samotném vytváření modelů. Samozřejmostí je *přehledná* a *jednoduchá syntaxe* grafické notace.

Nevýhodou této technologie je, že *nenabízí ucelený rámec metod* pro vymodelování všech aspektů byznys (software) procesu.



Obrázek 3: Příklad BPMN diagramu

3.1.3 EPC

V této kapitole si představíme další metodu, s jejíž pomocí můžeme modelovat dynamiku procesu. Podstata této metody spočívá v řetězení událostí a aktivit do posloupnosti, která vede k požadovanému cíli. Tento princip umožňuje efektivně a srozumitelně popsat modelovaný proces. Syntaxe metody EPC je ušita na míru komunity, která se zabývá problematikou modelování byznys procesů (čerpáno z [Von04]).

Pro modelování procesu pomocí EPC využíváme následujících prvků (dle [Von04]):

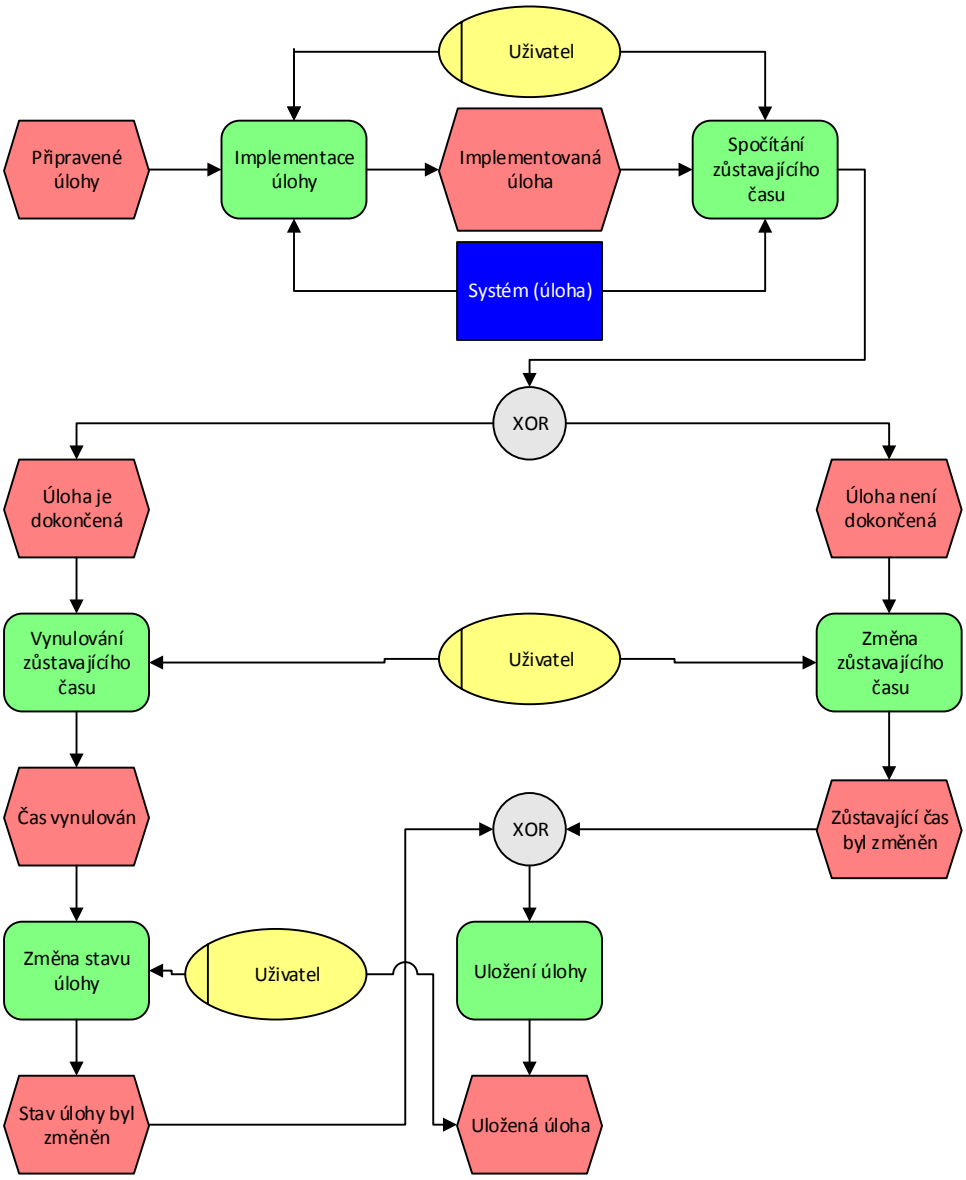
1. *Aktivity* můžeme označit za základní stavební blok, který určuje, co má být v rámci procesu vykonáno.

2. *Události* slouží pro popis situací, které nastávají před a/nebo po vykonání určité aktivity, jedná se tedy o vstupní/výstupní podmínky aktivit.

3. *Logické spojky* se používají k řízení toku (cesty kterou se vydáme) procesu. Existují tři typy spojek inspirované logikou a to AND, OR a XOR. Obecně se spojky používají buď na rozdělení toku činností nebo na jeho spojení.

Díky této malé množině syntaktických prvků poskytuje EPC technologie velice *jednoduchý princip* spojení událostí a aktivit pro modelování i velice komplexních procesů. Nespornou výhodou této metody je také fakt, že je *integrována* v *nejpoužívanějších nástrojích pro byznys modelování*.

Přes všechny své plusy trpí EPC hlavně *nedůsledně specifikovanou sémantikou*, což může vést k *nejednoznačnostem při specifikaci procesů*. Tento fakt může vést k problémům při softwarové implementaci těchto procesů v podobě standardního workflow.



Obrázek 4: Příklad EPC diagramu

3.1.4 UML

Před nástupem první verze UML standardu bylo vizuální modelování procesů roztříštěno vlivem nekompatibility jednotlivých notací (metod) a ideí jejich tvůrců [Wat08]. Autoři UML technologie si tedy dali za úkol sjednotit různé přístupy při vytváření specifikací nutných při vývoji software. Podařilo se jim vytvořit množinu diagramů, s jejíž pomocí jsme schopni popsat modelovaný systém ze všech možných perspektiv a stupňů abstrakcí [Von04].

Z této množiny se dají aplikovat na modelování byznys (software) procesů především tyto typy diagramů (převzato z [Von04]):

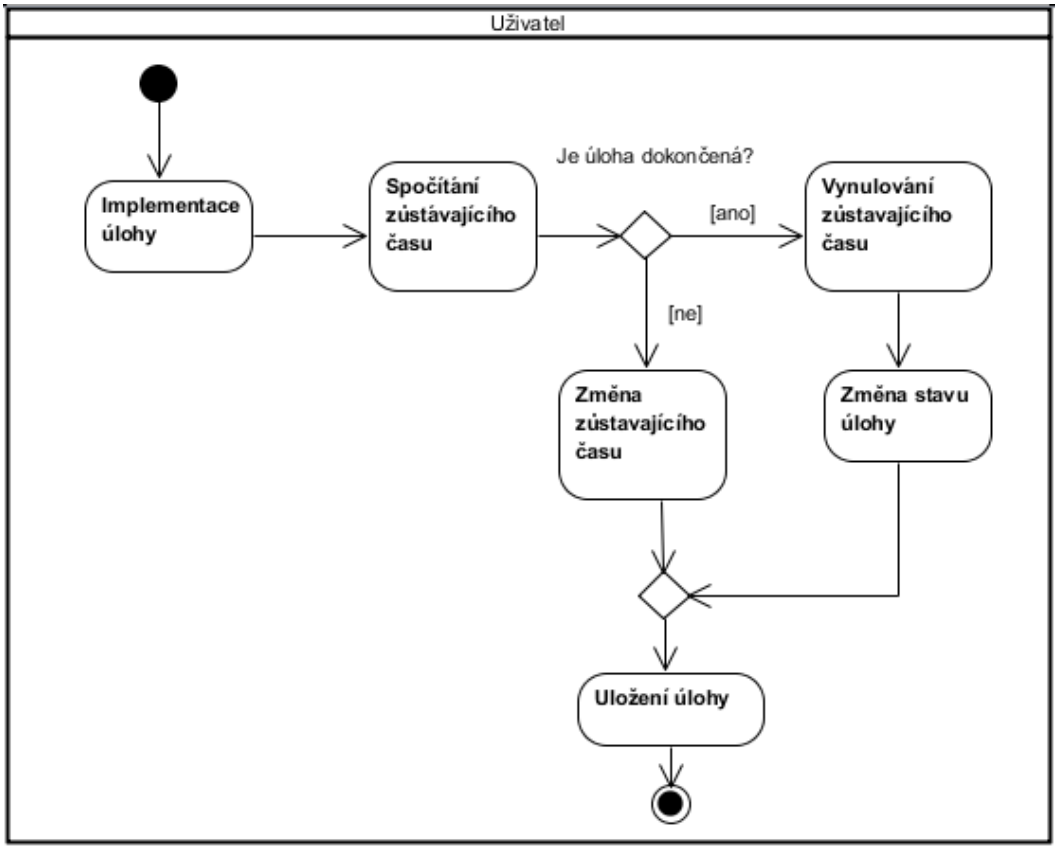
1. *Diagram případů užití* slouží k popisu funkcí, tedy k popisu funkčního přístupu.
2. *Diagram aktivit* vyjadřuje dynamiku systému a je vhodný pro specifikaci chování.
3. *Diagram tříd* použijeme pro modelování struktury systému, tzn. entit a jejich vazeb.

Jelikož se v této diplomové práci zabýváme především dynamickým aspektem softwarového procesu, tak se podíváme blíže na UML diagram aktivit. Tento diagram je reprezentován sekvencí kroků (aktivit). Pro určení pořadí vykonávání aktivit slouží řídící tok. Diagram aktivit může obsahovat tyto prvky:

- Akce/Aktivita - vyjadřuje vykonávanou činnost.
- Řídící tok - slouží ke spojení jednotlivých elementů. Řídící tok může obsahovat omezující podmínku. Pokud je tato podmínka splněna, lze přistoupit ke spuštění následujícího kroku.
- Inicializace aktivity - slouží jako vstupní bod aktivity (diagramu).
- Ukončení aktivity/toku - je použito jako ukončující bod diagramu, popřípadě pouze větve diagramu (v případě ukončení toku).
- Rozhodnutí - slouží k podmíněnému rozdělení toku diagramu.
- Synchronizované rozdělení a spojení toku - tyto elementy se používají v případě, že chceme tok diagramu paralelizovat, popřípadě synchronizovaně spojit do jednoho toku.
- Sloučení toku - element sloužící ke sloučení toku z více větví, na rozdíl od *spojení toku* vyžaduje splnění pouze jedné příchozí větve toku.
- Údálosti - dvojice elementů (příchozí událost a odchozí událost) určená k modelování údálostí. U odchozí údálosti lze také použít tzv. časovanou událost.
- Data - diagram aktivit umožňuje i modelování předávání dat, k tomu slouží elementy objektové toky.

Všechny typy UML diagramů mají základ de facto ve *standardizované grafické notaci*, kvůli které se tato technologie protlačila do celé řady softwarových systémů určených pro modelování procesů. Díky široké podpoře se UML jazyk stal *standardem* což umožňuje expertům z byznys sféry bez větších obtíží komunikovat s techničtěji zaměřenými odborníky z oboru vývoje softwaru. Autoři UML také usilovně pracují na *zajištění jednoznačnosti* této technologie z hlediska *syntaxe a sémantiky*.

Bohužel i přes zjevné snahy o formalizaci UML zatím *nelze* tento jazyk *považovat za formálně definovaný* [Von04].



Obrázek 5: Příklad UML diagramu aktivit

3.2 Formální metody

V předchozích kapitolách jsme si představili nejznámější zástupce převážně semi-formálních metod. Všechny tyto metody sdílely propracovanou syntaktickou stránku, některé dokonce i poměrně dobře navrženou sémantickou specifikaci. Všem však chyběla precizní sémantika podpořená matematickým základem. Z této skutečnosti vyplývá neschopnost verifikovat vymodelovaný systém a neexistuje možnost hlubší analýzy modelů.

Důvod, proč formální metody nepoužíváme při modelování přímo, je jejich použitelnost pro specifikování rozsáhlých systémů. Čas, který vynaložíme na vymodelování systémů je příliš dlouhý. Proto se formální metody modelování nikdy nestaly masovou záležitostí. Oblast jejich použití je omezena na kritické systémy vyžadující bezchybný běh (čerpáno z [Von04]).

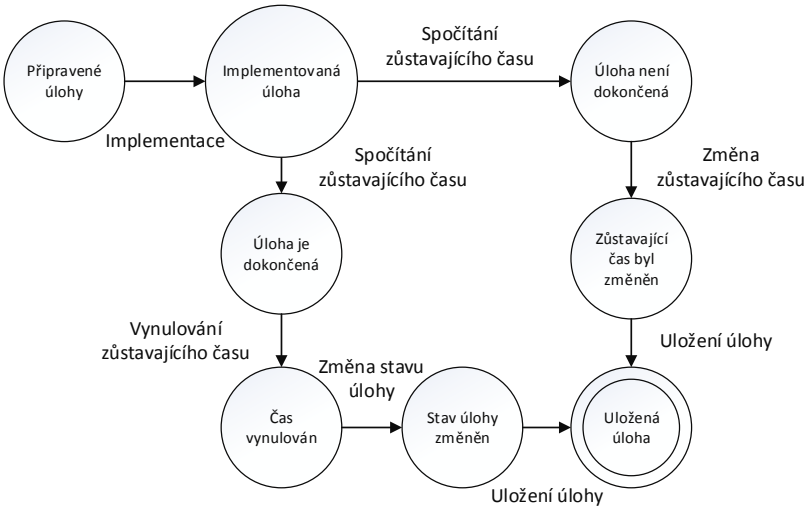
3.2.1 Konečné automaty

Konečné automaty jsou jednou z nejstarších metod používaných ke specifikaci chování systému. Strukturu automatu tvoří konečná množina stavů a přechodů mezi stavy. Automaty slouží k modelování vnitřních stavů systému, popřípadě procesu a umožňují nám náhlednout na konkrétní stav v daném časovém momentu.

Definice 3.1 *Konečný automat je uspořádaná pětice* $KA = (Q, I, \delta, q_0, F)$*, kde (dle [Von04]):*

1. Q *je konečná množina stavů.*
2. I *je konečná neprázdná množina vstupů.*
3. $\delta : Q \times I \mapsto Q$ *je přechodová funkce.*
4. $q_0 : \in Q$ *je počáteční stav.*
5. F $\subseteq Q$ *je množina koncových (přijímacích) stavů.*

Princip výpočtu konečného automatu probíhá tak, že se čtou elementy vstupu a automat na základě přechodové funkce δ rozhodne, do jakého stavu se přejde. Toto se opakuje dokud není přečten celý vstup. Pakliže se automat po skončení čtení vstupu nachází v *koncovém* stavu, vstup je přijat. V opačném případě je vstup odmítnut (čerpáno z [Von04]).



Obrázek 6: Příklad konečného automatu

3.2.2 Petriho sítě

Hned v úvodu bych rád upozornil, že tato kapitola se bude věnovat pouze Barevným Petriho sítím známým pod svou anglickou zkratkou CPN, přičemž popis CPN je rozdělen na tři části. V této části je obsažen pouze popis základní struktury CPN pro srovnání s ostatními metodami modelování, následuje představení z hlediska dynamiky v kapitole 4.3 a popis je završen formální definicí v kapitole 6. Hlavním důvodem pro rozhodnutí zabývat se pouze CPN je obrovská univerzálnost a široké možnosti modelování v těchto sítích oproti jejich poněkud primitivnějším „černobílým“ sourozencům. Pokud byste se přece jen rádi dověděli více o obyčejných Petriho sítích, můžete se obrátit na [Pet81].

Barevné Petriho sítě jsou grafický jazyk se silným matematickým základem, určený pro vytváření konkuretních systémů a jejich následnou analýzu. Tato technologie v sobě spojuje propracovaný základ v podobě normálních Petriho sítí spolu s možnostmi vysokoúrovňového skriptovacího jazyka. Z normálních Petriho sítí je převzata grafická notace a prvky sloužící k modelování paralelismu, komunikace a synchronizace. Nad tímto základem stojí skriptovací jazyk CPN ML, který vychází z jazyka Standard ML a poskytuje definici datových typů a výrazů jazyka. Primárně jsou CPN vhodné a používané pro vysoce konkuretní systémy, např. komunikační protokoly, datové sítě a distribuované algoritmy. Díky své univerzálnosti jsou více než dobře použitelné právě při modelování byznys (software) procesů, kde slouží spolu s konečnými automaty k modelování specifikace chování.

Model sítě popisuje stavy, ve kterých se systém může nacházet a události vedoucí ke změně stavu daného systému[Jen09].

Síť je tvořena posloupností těchto syntaktických prvků:

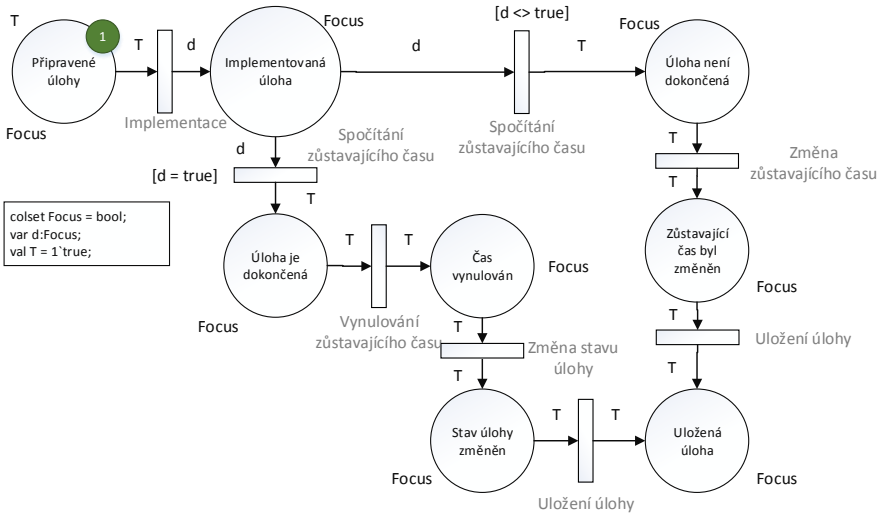
1. *Místo* reprezentuje nádobu na tokeny a slouží jako vstupní podmínka *přechodu*. Souhrn počtu tokenů na všech místech sítě nám udává stav, v jakém se síť v daném okamžiku nachází.

2. *Přechod* je prvek, jehož spuštěním dochází ke změně stavu sítě. Přechod by se dal označit za událost v systému.

3. *Hrana* propojuje předešlé prvky a definuje pomocí hranového výrazu počet tokenů, který bude odebrán, respektive přidán z přidruženého *místa*. Hrana vždy propojuje jedno místo s jedním přechodem.

Barevné Petriho sítě jsou výborným prostředkem pro modelování procesů. Jsou dokonce schopny *eliminovat nepříjemnou vadu* formálních modelovacích metod v podobě *nepoužitelnosti* těchto metod při zpracovávání velkých systémů, a to díky podpoře hierarchie. Díky hierarchickým sítím jsme schopni rozdělit větší systémy do menších celků a ty modelovat samostatně.

Za jedinou nevýhodu lze považovat *nutnost hlubších znalostí* této metody jako takové, oproti semi-formálním systémům, kde nevyvstává nutnost větší expertízy.



Obrázek 7: Příklad Barevné Petriho sítě

Z příkladu je patrné, že Petriho sítě mají ke konečným automatům velmi blízko (viz. příklad 6).

3.2.3 OWL

V předchozích kapitolách byly představeny formální metody určené pro specifikaci chování procesu (pro modelování dynamiky procesu). Z kapitoly 2 je však patrné, že softwarový proces má také svou statickou stránku. Pro modelování tohoto aspektu lze použít formální jazyk OWL.

Jde o jazyk vyvinutý konsorciem W3C pro vytváření ontologií a podporu sémantického webu. Jde o nástupce DAML+OIL s rozšířenými možnostmi modelování a podporou hierarchie (čerpáno z [OWL12]).

S pomocí OWL můžeme podpořit formální statické modelování software (byznys) procesu. Obě metody (tedy formální dynamické a statické modelování) lze vzájemně kombinovat (dle z [Kos10]).

4 Simulační metody

Než si představíme jednotlivé metody simulace, pojďme si o simulaci jako vědní discplíně něco říci.

Co je základní myšlenkou simulace? Simulace má za úkol napodobit chod poměrně složitého reálného systému pomocí počítačového modelu a poté při experimentování s modelem pozorovat chování vymodelovaného systému [Dlo07].

Jaký byl důvod pro zavedení simulace? S roustoucí komplexitou podnikových procesů přestaly stačit analyticky přesná řešení používající matematiku jako základ. Pro co nejreálnější modelování složitých systémů, které obsahují řadu navzájem provázaných prvků s pravděpodobnostními a dynamickými charakteristikami, je tedy vhodnější počítačová simulace (čerpáno z [Dlo07]).

Simulace nám nabízí možnost měnit vnější a vnitřní podmínky systému a dopad těchto změn na systém otestovat (odsimulovat), aniž bychom museli reálně zasahovat do procesu (výroby, vývoje softwarového produktu) [Dlo07].

Pro dosažení optimálních výsledků simulace nastiňuje [Dlo07] 8 fází projektu:

1. *Rozpoznání problému a stanovení cílů* - správná formulace problému je zásadním krokem, jedná se o fázi vymezení problému a stanovení dosažitelných cílů
2. *Vytvoření konceptuálního modelu* - tato fáze nám pomůže vytvořit si náhled o modelovaném systému
3. *Sběr dat* - simulace je datově náročnou metodou a v této fázi bychom měli zajistit správnost dat
4. *Tvorba simulačního modelu* - převod konceptuálního modelu na počítačový model, v této fázi můžeme doladit nedostatky konceptu
5. *Verifikace a validace modelu* - ověříme shodu počítačového modelu s konceptuálním modelem (verifikace) a také s realným systémem (validace)
6. *Provedení experimentu a analýza výsledků* - spuštění simulace a následná interpretace výsledků
7. *Dokumentace modelu* - důležitá fáze z hlediska znovupoužitelnosti
8. *Implementace* - zavedení navržených změn do praxe

4.1 Systémová dynamika

Historie industriální (posléze systémové) dynamiky sahá do 50. let minulého století, kdy byla představena Jay W. Forresterem. Tento původně elektrotechnický inženýr stanovil základní principy systémové dynamiky v reakci na problémy svých kolegů z GE v jednom z výrobních závodů. Na základě pozorování byly vytvořeny první simulace a vznikl první kompilátor diferenciálních rovnic pojmenovaný SIMPLE. Diferenciální rovnice sloužily k popisu simulačních modelů. V 60. letech se Systémová dynamika postupně rozšířila a pole působnosti se zvětšilo o modelování jiných než podnikových systémů a přibyly také nové simulační softwarové nástroje jako DYNAMO, Vensim, STELLA a Powersim (čerpáno z [Mil03]).

Vytváření simulačního modelu a modelování vnitřních vztahů systému může být složité, Systémová dynamika nám poskytuje pohled a potřebné nástroje, díky kterým se můžeme s komplexitou systému lépe vypořádat [Mil13].

V současné době se používají dva přístupy při vytváření simulačních modelů (dle [Mil13]):

1. *Přičinné smyčkové diagramy*
2. *Diagramy stavů a toků*

Systémová dynamika těží ze své *bohaté minulosti* a schopnosti simulovat systém *jakékoliv velikosti* (od jednoduchého byznys procesu, až po model celého města). Bohužel tato metoda klade *vysoké nároky* na *zkušenosti* a *odbornost* při vytváření modelu a nemusí tak být použitelná pro každého byznys uživatele.

4.2 Multiagentní modelování

Základem metody multiagentního modelování jsou entity zvané *agenti*. Definice agenta dle [Wol09] zní v překladu takto:

Definice 4.1 *Agent je počítačový systém, který je schopen* nezávislé *(autonomní) akce jménem svého uživatele či vlastníka (aneb autonomní vyvození činností, kterých je třeba provést k dosažení cíle bez neustálého dozoru).*

Jelikož se multiagentní modelování používá pro řadu aplikací, návrh a činnost agentů se liší dle konkrétního použití. Agenti mohou vykonávat jednoduché činnosti v rámci systému, ale mohou to také být umělé inteligence samy o sobě. Woolridge identifikoval ve své publikaci ([Wol98]) klíčové vlastnosti agentů:

1. *Autonomie* - agenti operují sami o sobě bez vnějšího zásahu a jsou také schopni kontrolovat své akce a interní stavy.
2. *Schopnost socializace* - agenti jsou schopni interakce s lidmi či dalšími agenty, pakliže je to potřeba pro splnění jejich cílů.
3. *Reaktivita* - agenti vnímají své okolí, jsou schopni reagovat na změny v tomto okolí.

4. *Pro-aktivita* - agenti se neřídí pouze podněty ze svého okolí, jsou schopni převzít iniciativitu, aby splnili své cíle.

Metoda multiagentního modelování je vhodná v případě, že je modelovaný systém velmi *dynamický*, *nestálý* či *komplexní*. Agenti jsou schopni díky své schopnosti *adaptace* se s těmito problémy vypořádat. Další výhodou je fakt, že simulaci pomocí multiagentního modelování lze použít pro *distribuované systémy* (čerpáno z [Wol98]). Obecným problémem multiagentního modelování je jeho *roztříštěnost* z hlediska sémantiky mezi jednotlivými doménami použití [Nia11].

4.3 Diskrétní simulace (Petriho sítě)

Metoda Diskrétní simulace nahlíží na modelovaný systém jako na posloupnost kroků (proces). Tyto kroky jsou představovány událostmi v čase. Chápání času však v tomto případě není spojité, ale skokové (čas nabývá hodnot z předem určené diskrétní množiny).

Jinak řečeno, jestliže změna stavu systému nenastává průběžně, ale pouze v okamžiku výskytu z hlediska systému významné události, pak hovoříme o *diskrétní simulaci*[Dlo07].

Tato metoda je díky podobnosti svých modelů s vývojovými diagramy velice oblíbena mezi uživateli z byznys sféry.

Do rodiny metod diskrétní simulace patří Barevné Petriho sítě, které byly krátce představeny v 3.2.2, avšak pouze z hlediska použitelnosti pro modelování software procesů a z hlediska struktury. V sekci 6 jsou doplněny o formální popis. Cílem této kapitoly je popsat dynamiku Petriho sítí, tzn. popsat systém, jakým jsou jednotlivé přechody „uschopňovány“ k provedení a jakým způsobem Petriho sítě mění svůj stav.

Přechod je proveditelný, pakliže jsou splněny tyto podmínky (dle [Jen09]):

1. *Všechny příchozí hranové výrazy jsou ohodnotitelné* - aby byl přechod proveditelný, musíme být schopní najít takové ohodnocení (*binding*) proměnných, obsažených v hranových výrazech, aby se daný hranový výraz vyhodnotil jako *multiset* tokenů, který je již obsažen v přidruženém vstupním *místu*, tedy $\forall p \in P : E(p,t)\langle b \rangle \ll= M(p)$.

2. *Guard výraz je splnitelný* - pokud je přiřazen k přechodu *guard* (omezující podmínka), musí být tento výraz vyhodnocen jako *true* v daném *ohodnocení* (binding), tedy $G(t)\langle b \rangle$.

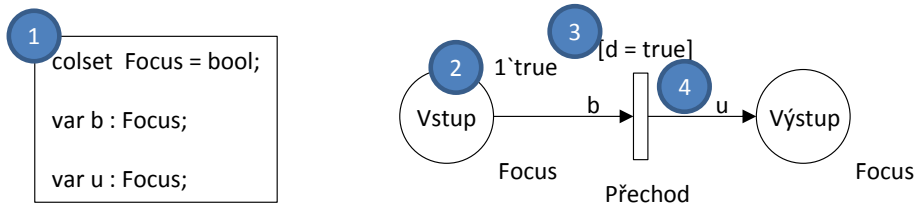
Poznámky ohledně návratových hodnot:

- Návratová hodnota hranového výrazu (tedy multiset) musí být stejného typu (colorset) jako přidružené místo dané hrany.

- Návratová hodnota guardu musí být typu *bool*.

Příklady proveditelnosti přechodu:

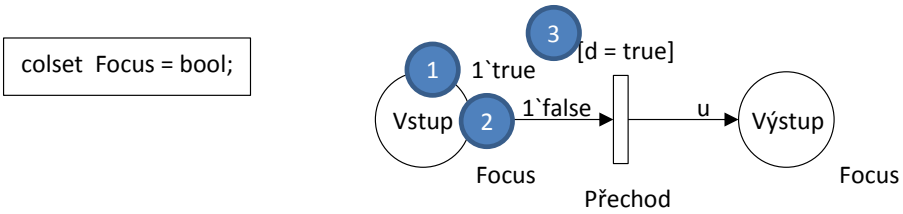
- *Proveditelný přechod*



Obrázek 8: Příklad proveditelného přechodu

1. Deklarace colorsetu a proměných.
2. Místo *Vstup* obsahuje jeden token s hodnotou *true*.
3. Přechod má přiřazen guard, který říká, že proměnná *d* musí být *true*.

- *Neproveditelný přechod*



Obrázek 9: Příklad neproveditelného přechodu

1. Místo *Vstup* obsahuje jeden token s hodnotou *true*.
2. Vstupní hranový výraz požaduje *multiset* o jednom tokenu s hodnotou *false*. Vstupní místo tyto tokeny neobsahuje a přechod tak není proveditelný.
3. I kdyby vstupní *místo* obsahovalo požadované tokeny, přechod je stále neproveditelný díky *guard* výrazu.

Petriho síť v každém kroku (stavu) vyhodnotí proveditelné přechody. Jeden přechod je poté náhodně vybrán a proveden, tím dochází ke změně stavu sítě. Běh Petriho sítě končí v případě, kdy není nalezen ani jeden proveditelný přechod.

Modelování v Petriho sítích je podobné (avšak přeci jen náročnější) jako modelování pomocí jiných metod založených na workflow modelování (UML, EPC a BPMN). Díky tomu je *modelování systému* a *následná simulace* pro byznys uživatele *snažší* než tomu bylo u metody systémové dynamiky. Petriho sítě nabízejí možnost *verifikace* modelovaného systému za použití *analýzy stavového prostoru* a simulace. Za nevýhodu této metody můžeme považovat to, že používá pro simulaci *diskrétní veličiny* (čas), což nemusí plně odpovídat realitně.

5 Výsledky zkoumání

5.1 Metody modelování

V kapitolách věnovaných metodám byznys (software) modelování jste měli možnost se krátce seznámit s nejpoužívanějšími zástupci. Z textu vyplývá, že ani jedna z metod není ideální. Toto tvrzení platí i pro odlišné přístupy (neformální, semi-formální, formální) metod - každý přístup má své silné a slabé stránky, ať už se jedná o srozumitelnost a použitelnost či precizně definovanou sémantiku.

Za nejuniverzálnější můžeme považovat rodinu metod IDEF a jazyk UML. Jelikož cílem této práce je vybrat pouze jednu z metod, ve výsledku se přikloníme k metodě UML. Důvodem pro toto rozhodnutí je fakt, že jazyk UML je modernější a populárnější než IDEF, neustále se vyvíjí a ve výsledku je srozumitelnější.

Shrnutí:

1. *UML je moderní*, s tím souvisí rozšířená podpora v softwarových nástrojích
2. *UML je populární* mezi experty v oboru modelování procesů
3. *UML se neustále vyvíjí*; tvůrci se snaží UML zdokonalit a odstranit nejasnosti
4. *Syntaxe UML je srozumitelná* a boří tak bariéry mezi inženýry a obchodníky

5.2 Metody simulace

V předchozích podkapitolách byly představeny v dnešní době nejznámnější simulační metody, každá se svou specifickou filozofií.

I přes její obrovskou univerzálnost se Systémová dynamika nejeví jako nejvhodnější metoda pro použití v rámci byznys (software) modelování a simulace, zejména z důvodu vysokých nároků na vytvoření simulačního modelu.

Metoda multiagentního modelování se jeví jako vhodný kandidát pro simulaci byznys procesů, zvláště s rostoucím trendem využití biologicky inspirovaných výpočtů pro řešení netradičních problémů současnosti, se kterými si analytické metody nedokáží poradit.

Proč ale nevyužít již zavedených standardů a nepoužít populární UML pro vytvoření semi-formálního modelu systému, následný automatizovaný převod do CPN (viz. kapitola 6) a následně tuto síť nepoužít pro simulaci a následnou analýzu vymodelovaného systému? Z důvodu přirozenosti, jakým jsou Petriho sítě podobné modelovaným workflow a také z důvodu možnosti převodu UML diagramu aktivit právě na CPN jsme vybrali Petriho sítě jako nejvhodnější metodu pro naše potřeby.

6 Formalizace modelů

V předchozích kapitolách jsme probrali výhody a nevýhody semi-formálních a formálních metod. Vyvodili jsme, že semi-formální metody jsou i přes jejich nevýhody lepší volbou (pro lidské uživatele). Co kdyby existoval způsob, jak tyto metody spojit a získat tak precizně definovou syntaxi a sémantiku bez nutnosti vytváření složitých a nesrozumitelných modelů? V této kapitole se pokusíme nastínit možnost automatizovaného převodu (formalizace) UML diagramu aktivit na Barevnou Petriho síť. Převod bude následně použit v nástroji Software Process Modeler.

Nejprve si formálně představme (nehierarchickou) CPN.

Definice 6.1 *CPN síť je uspořádáná n-tice* $N = (P, T, A, \sum, V, C, G, E, I)$*, kde (dle [Jen09]):*

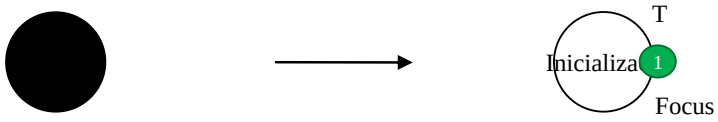
1. P *je konečná množina míst.*
2. T *je konečná množina přechodů, splňujících podmínku* $P \cap T = \emptyset$*.*
3. A *je množina orientovaných hran, přičemž* $A \subseteq (P \times T) \cup (T \times P)$*.*
4. $\sum$ *je konečná množina neprazdných colorsetů.*
5. V *je konečná množina typovaných proměnných, kde* $Type[v] \in \sum$ *pro všechny proměnné* $v \in V$*.*
6. C *je funkce* $C : P \mapsto \sum$*, mapující colorset na každé místo.*
7. G *je funkce* $G : T \mapsto EXPR_v$*, která přiřazuje guard každému přechodu* t*, přičemž platí* $Type[G(t)] = Bool$*.*
8. E *je funkce* $E : A \mapsto EXPR_v$*, přiřazující každé hraně* a *hranový výraz, kde* $Type[E(a)] = C(p)_{MS}$*, kde* p *je místo spojené s hranou* a*.*
9. I *je funkce* $I : P \mapsto EXPR_\emptyset$*, která přiřazuje inicializační výraz každému místu* p*, přičemž platí* $Type[I(p)] = C(p)_{MS}$*.*

UML diagram aktivit se ve verzi 2.0 nechal inspirovat sémantikou Petriho sítí a adoptoval myšlenku předávání tokenů pro řízení toku diagramu. Tokeny v tomto případě obsahují data nebo označují proveditelnou aktivitu (aktivity).

Na diagram aktivit můžeme nahlížet jako na orientovaný graf, díky tomu lze převod na CPN automatizovaně uskutečnit (vzhledem k faktu, že Petriho sítě jsou orientované grafy samy o sobě). Určitým typem algoritmu pro procházení grafu (například BFS nebo DFS) můžeme tento graf procházet a převádět elementy diagramu aktivit na elementy Petriho sítě.

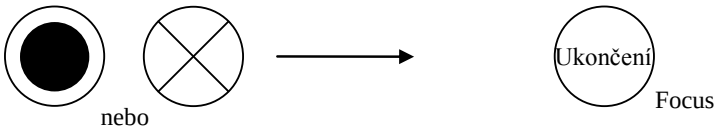
Ukažme si pravidla, pomocí kterých můžeme elementy diagramu aktivit převést na elementy Petriho sítě.

1. *Inicializace aktivity* - v tomto případě je převod poměrně zřejmý, element inicializace aktivity lze považovat za vstupní místo aktivity.



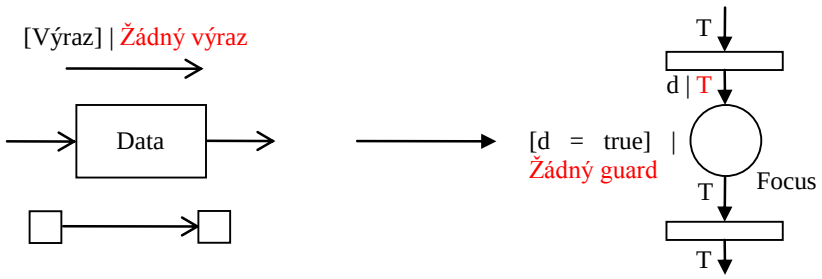
Obrázek 10: Mapování inicializace aktivity na odpovídající elementy CPN

2. *Ukončení aktivity* - viz. *inicializace aktivity* s tím rozdílem, že se jedná o místo výstupní.



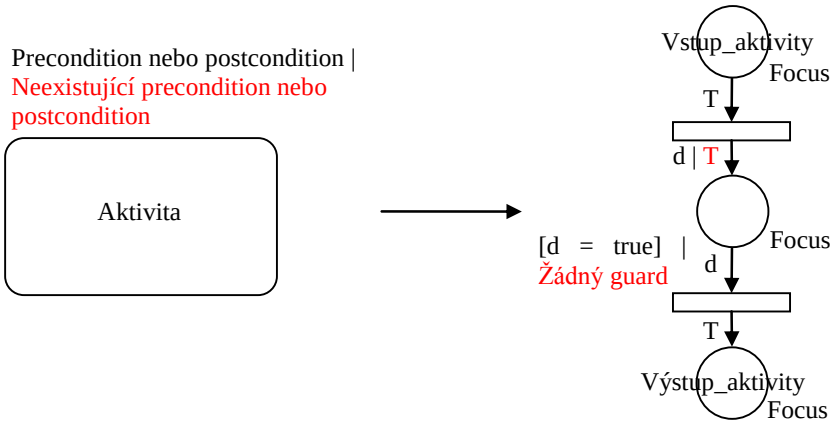
Obrázek 11: Mapování ukončení aktivity na odpovídající elementy CPN

3. *Toky* - toky (řídící, objektový) jsou převedeny na následující konstrukt CPN, přičemž první přechod slouží jako generátor náhodné hodnoty. Podmínka toku je reprezentována jako guard druhého přechodu.



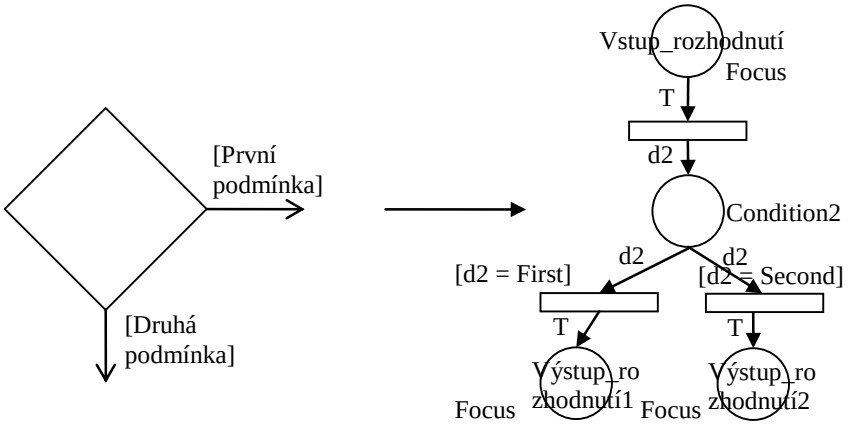
Obrázek 12: Mapování toků na odpovídající elementy CPN

4. *Aktivita* - na následujícím obrázku se nachází *nejjednodušší* forma převodu aktivity. Aktivita má vstupní místo, přechod symbolizující samotnou aktivitu, přechod symbolizující *precondition* a *postcondition* a výstupní místo. Tento typ převodu nepodporuje transakce.



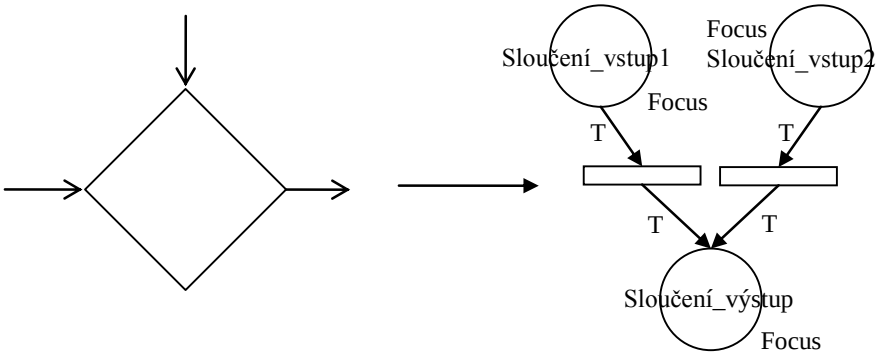
Obrázek 13: Mapování aktivity na odpovídající elementy CPN

5. *Rozhodnutí* - tento UML element je vyjádřen jedním přechodem, sloužícím jako generátor náhodné hodnoty a dále *n* přechody, jejichž spuštění je podmíněno výrazy z elementu rozhodnutí a dále *n* výstupními místy, jedno pro každou rozhodovací větev.



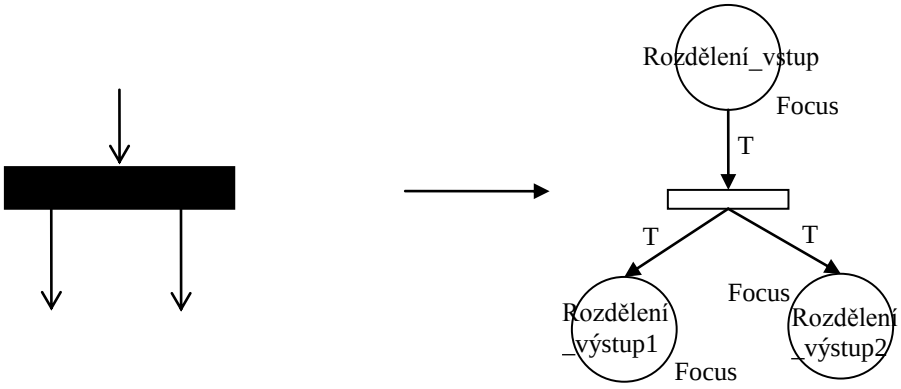
Obrázek 14: Mapování rozhodnutí na odpovídající elementy CPN

6. *Sloučení* - tento element slouží ke sloučení rozdělených toků za přepokladu, že alespoň jedna z připojených aktivit byla provedena.



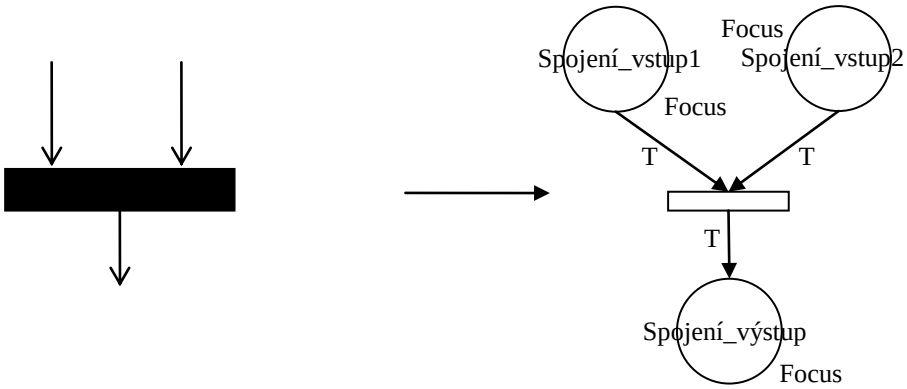
Obrázek 15: Mapování spojení na odpovídající elementy CPN

7. *Rozdělení toku* - rozdělení toku slouží k paralelizaci (možnost současného spouštění) dvou větví aktivit. Paralelizace se v CPN modeluje přechodem se dvěma (či více) výstupními místy, přičemž daný přechod vloží tokeny do obou míst a tím zajistí možnost současného provádění následujících přechodů.



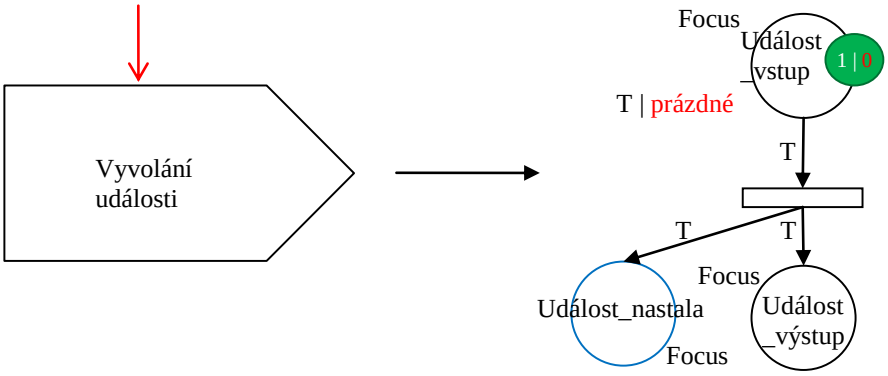
Obrázek 16: Mapování rozdělení toku na odpovídající elementy CPN

8. *Spojení toku* - tento prvek doplňuje element rozdělení toku a je jeho opakem. Jeho úkolem je sloučit (sesynchronizovat) dříve rozdělené toky. Toto je dosaženo *n* vstupními místy přechodu, jež bude proveden pouze v případě, že obě místa nebudou prázdná.



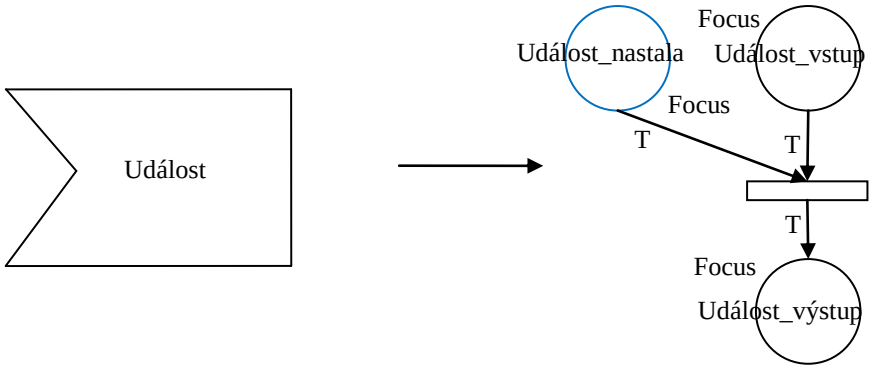
Obrázek 17: Mapování spojení toku na odpovídající elementy CPN

9. *Spuštění události* - prvek spuštění události nemá žádný vliv na tok diagramu, pouze vyvolává událost. Tento prvek je tedy převeden na přechod, který předává token beze změny řízení dál, avšak vkládá token do speciálního místa určeného pro danou událost.



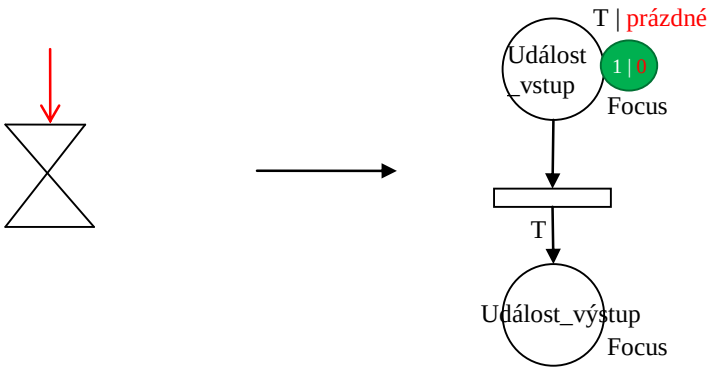
Obrázek 18: Mapování spuštění události na odpovídající elementy CPN

10. *Příchozí událost* - v tomto případě se jedná o element, který není spouštěn přímo v rámci toku diagramu svým předchůdcem, ale vyvoláním události v systému - buď zevnitř anebo zvenčí. V kontextu Petriho sítí můžeme toto chování modelovat přidáním speciálního vstupního místa, do kterého bude token vložen pouze v případě vyvolání události.



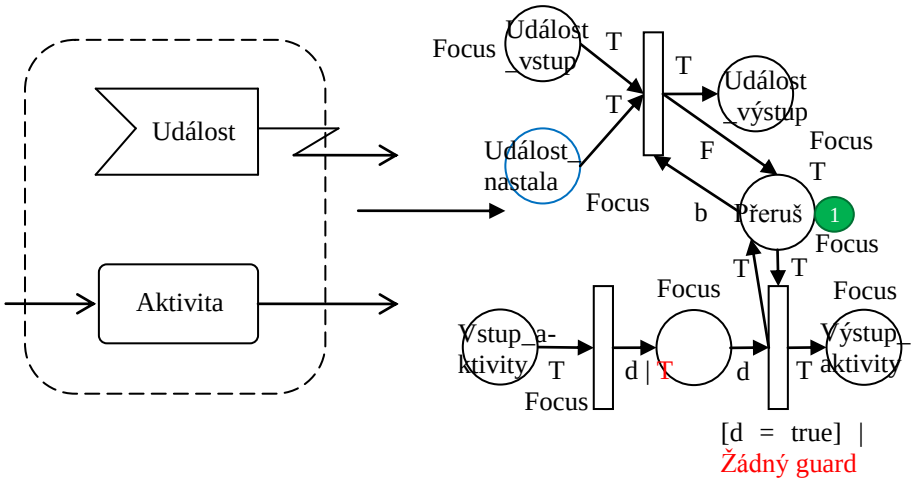
Obrázek 19: Mapování příchozí události na odpovídající elementy CPN

11. *Časovaná událost* - tento element je podobný elementu *příchozí událost* s tím rozdílem, že událost není spouštěna externě (elementem *spuštění události*). Událost je spuštěna na základě stanovené časové prodlevy. Při převodu je časová složka zanedbána.



Obrázek 20: Mapování časované události na odpovídající elementy CPN

12. *Blok přerušení* - tento element slouží k zastavení vykonávání posloupnosti aktivit, například z důvodu chyby. V Petriho síti je tento mechanismus modelován jedním společným *vstupním* místem všech aktivit, která jsou obsažena v bloku přerušení.



Obrázek 21: Mapování bloku přerušení na odpovídající elementy CPN

Pár poznámek k pravidlům představeným výše:

- každý element začíná a končí *místem*, jedinou výjimkou je element *řídící tok*, který v UML diagramu aktivit vždy spojuje dva ostatní elementy. Díky tomuto faktu vždy splníme body 2. a 3. z definice 6.1 při konstrukci Petriho sítě z diagramu aktivit.
- pakliže je *aktivita* následovníkem více než jedné aktivity, je pro každou „vstupní“ aktivitu (respektive *řídící tok*) vytvořeno separátní vstupní místo pro *přechod* aktivity. Z toho vyplývá, že přechod aktivity bude spuštěn až po provedení všech příchozích aktivit (přesně dle definice tohoto chování v UML).
- pravidlo pro převod *aktivity* platí i pro UML element *akce*.
- pakliže aktivita obsahuje další akce, popřípadě subdiagram, je tento subdiagram rozvinut (a převeden dle výše zmíněných pravidel) a nahrazuje rodičovskou aktivitu.

Doplňme nyní pravidla pro převod UML elementů na CPN elementy o pravidla pro colorsety, proměnné a výrazy:

1. *Colorsety* - colorsety budou deklarovány tyto:

```
colset Focus = bool;

colset Condition2 = with First | Second;
colset Condition3 = with First | Second | Third;
...
```

Výpis 1: Deklarace colorsetů

Všechny tokeny v síti tedy budou typu *bool* (true | false hodnota) a budou sloužit jen k řízení toku sítě s vyjímkou elementu *rozhodnutí*, kde budou použity tokeny typu Condition*N*. Tyto colorsety budou generovány na základě počtu použitých výstupních větví (podmínek) všech elementů *rozhodnutí*. Pokud se v diagramu budou nacházet *rozhodnutí* pouze s dvěma výstupy, bude vygenerován pouze colorset *Condition2*, atd.

2. *Proměnné a konstanty* - deklarovány budou následující proměnné a konstanty:

```
var b : Focus;
var d : Focus;
val T = 1`true;
val F = 1`true;

var d2 : Condition2;
var d3 : Condition3;
...
```

Výpis 2: Deklarace proměnných a konstant

- Proměnná *b* je použita v CPN konstruktu *blok přerušení*.
- Proměnná *d* slouží k vygenerování náhodné hodnoty ve *řídícím toku* a jako precondition a postcondition *aktivity*.
- Proměnná *d2* slouží k vygenerování náhodné logické hodnoty použité v guard výrazech přechodů převedeného *rozhodnutí* (pokud má rozhodnutí dvě výstupní větve).
- Proměnná *d3* slouží k vygenerování náhodné logické hodnoty použité v guard výrazech přechodů převedeného *rozhodnutí* (pokud má rozhodnutí tři výstupní větve).
- Proměnné typu Condition*N* budou generovány na základě počtu použitých výstupních větví (podmínek) všech elementů *rozhodnutí*.

3. *Výrazy* - výrazy řídících toků a výrazy rozhodnutí budou převedeny následovně:

   - *Výrazy řídících toků* budou převedeny na jednoduchý CPN ML:

     ```
     d = true
     ```

     Výpis 3: Převedený výraz řídícího toku

     Tento výraz bude použit v guardu *druhého* přechodu *řídícího toku*.

   - *Výrazy rozhodnutí* budou převedeny na následující CPN ML výrazy (pro rozhodnutí s dvěma podmínkami):

     ```
     d2 = First
     ```

     Výpis 4: Převedený výraz guardu pro kladnou větev

     Výše uvedený výraz bude sloužit jako guard pro kladnou větev *rozhodnutí*.

     ```
     d2 = Second
     ```

     Výpis 5: Převedený výraz guardu pro zápornou větev

     Výše uvedený výraz bude sloužit jako guard pro zápornou větev *rozhodnutí*.

Na začátku bude token nesoucí hodnotu *true* vložen pouze do míst reprezentujících element *inicializace aktivity* a do vstupních míst *odchozích událostí* (avšak pouze v případě, že jim nepředchází žádná aktivita). Zbytek míst bude prázdný.

Použitím metody převodu s použitím tzv. *focus* tokenů se soustředíme pouze na dynamiku Petriho sítě se zachováním její jednoduchosti. To nám umožňuje (v případě potřeby) jednoduše rozšířit model CPN o další colorsety reprezentující datovou složku diagramu aktivit.

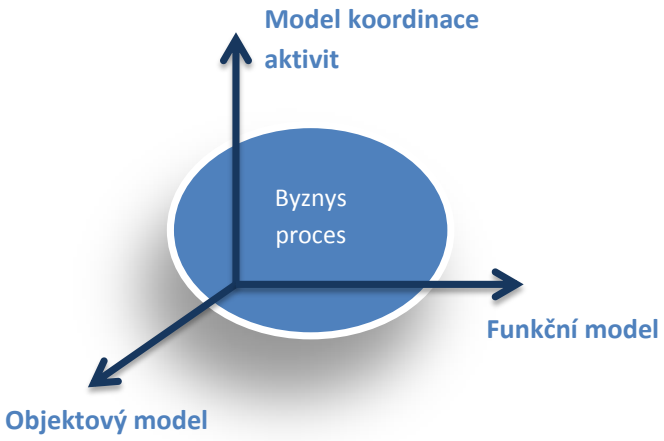
Z pravidel je zřejmé, že konstrukty CPN zavádí určitou redundantnost (model CPN po převedení obsahuje více prvků než model diagramu aktivit). V tomto případě (pro zachování co nejvyšší přehlednosti pro převod) není redundantnost na škodu. Pokud bychom však chtěli zvýšit efektivitu převodu (a následný celkový výkon CPN), tak existují metody na redukci počtu míst a přechodu Petriho sítě (např. [Eva05]).

Výše uvedená pravidla jsou založena na výzkumu [Gar02] a [Sta08], některá však byla vylepšena a byl doplněn způsob generování colorsetů, proměnných a hranových výrazů.

7 Software Process Modeler

Předešlé kapitoly byly zaměřeny především na teoretickou část softwarového procesu, metod modelování a simulace. Tato teoretická část sloužila k představení vybraných metod, k následnému výběru metod modelování a simulace, které budou použity a implementovány v rámci nástroje *Software Process Modeler* (zkráceně SPM). Primárním účelem SPM je podpora softwarových procesů s pomocí simulace.

SPM vychází z BP studia, což je balík nástrojů určených pro modelování, analýzu a simulaci byznys procesů. BP studio (respektive jeho modelovací nástroj) je komplexní software používající BPM metodu jako základ. Tato metoda je formalizována v podobě Petriho sítí, které jsou použity na pozadí (pro analýzu a simulaci). Na následujícím obrázku jsou vyobrazeny složky BPM (převzato z manuálu BP studia):



Obrázek 22: Tři složky BPM

SPM se zaměřuje na modelování složky *koordinace aktivit* ve formě diagramů aktivit. Podporuje však také modelování *objektové* složky s pomocí třídních diagramů. Pro vytváření modelů se tedy používá technologie UML (viz. 5.1). Simulační jádro tohoto nástroje je postaveno na Barevných Petriho sítích (viz. 5.2).

Shrňme si hlavní vlastnosti Software Process Modeler nástroje:

1. *Modelování* - SPM nabízí možnost modelovat softwarový proces pomocí třídních diagramů a diagramů aktivit.

2. *Simulace* - pro simulaci je použito simulační jádro postavené na Barevných Petriho sítích. SPM podporuje pouze neinteraktivní mód simulace s možností specifikovat počet opakování běhů simulace.

3. *Analýza* - SPM podporuje základní statickou analýzu modelovaného procesu. Analýzu lze také provést ze strukturovaných hlášení (výstupů) simulace.

Pro větší pohodlí uživatele je také možné jiným nástrojem již vymodelované UML diagramy (diagramy aktivit a třídní diagramy) importovat ve formátu XMI.

7.1 Analýza požadavků

Jelikož požadavky na software byly zadány nejprve pouze stručně formou zadání diplomové práce a obsahovaly jen dva základní požadavky (schopnost *modelovat* a *simulovat* softwarový proces) musel jsem v počátku prozkoumat konkurenční nástroje jako *Visual Paradigm*, *ADONIS*, *ARIS*, *MS Visio* a v poslední řadě také (v tomto případě ne konkurenční) *BP studio*. Seznámením se s těmito nástroji jsem byl schopen původní požadavky rozvést do detailu.

Celý vývoj probíhal agilně a požadavky tak byly doplňovány v průběhu vývoje aplikace. V tomto případě však požadavky nepocházely od zákazníka respektive *stakeholderů* (jak bývá obvyklé), ale od vedoucího diplomové práce. Dále vznikaly *vnitřní požadavky*, buď ode mě jako vývojáře nebo jako doporučení od kolegů specialistů, kteří se v oboru softwarového inženýrství pohybují.

Požadavky byly vždy roztříděny do dvou základních kategorií, na *funkční* a *nefukční* požadavky a posléze do podkategorií dle stupně důležitosti/náročnosti. Poté byly analyzovány, jsou-li *splnitelné*, zda *nejsou protichůdné* a v neposlední řadě, zda-li jsou *rentabilní*. Pokud požadavek splnil všechny tyto atributy, tak byl přijat k dalším fázím, tedy návrhu a následné implementaci.

Příklady funkčních požadavků, které byly implementovány:

- Uživatel by měl být schopen vymodelovat *dynamickou* stránku procesu pomocí jedné z konvenčních modelovacích metod.
- Uživatel by měl být schopen vymodelovat *strukturální* stránku procesu pomocí jedné z konvenčních modelovacích metod.
- Uživatel by měl být schopen odsimulovat proces a získat výsledky simulace.
- Uživatel by měl být schopen importovat již existující modely.
- Uživatel by měl být schopen exportovat výsledky simulace do HTML, popřípadě PDF.

Příklady nefunkčních požadavků, které byly implementovány:

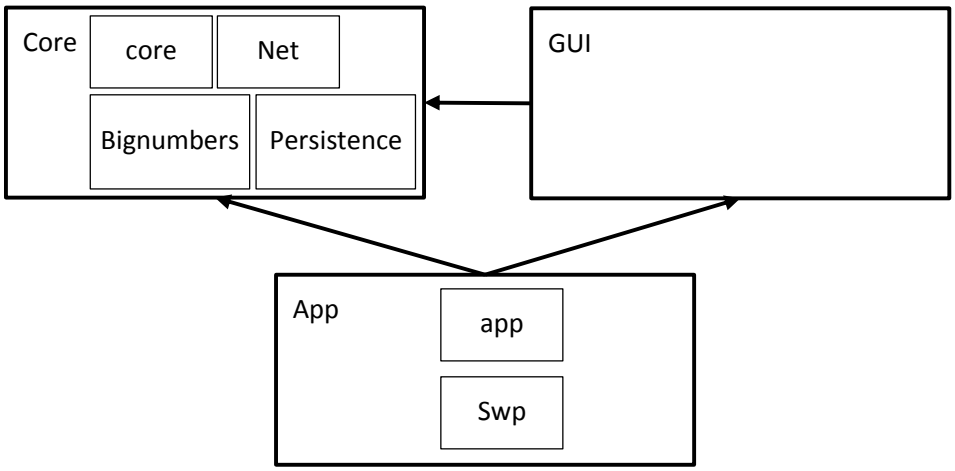
- Aplikace by měla být multiplatformní

7.2 Návrh

Návrh aplikace vycházel z nástroje, který jsem spolu se svými kolegy implementoval v rámci individuální odborné praxe (více ve [Czo11]) a byl koncipován jako návrh desktopové aplikace. Při designu architektury a jednotlivých komponent bylo využito zavedených (a ověřených) návrhových vzorů (viz. [Pec07]) a byla dodržována co nejvyšší míra znovupoužitelnosti komponent. Návrh také počítá s možným rozšířením o zásuvné moduly (angl. plugin) rozšiřující funkcionalitu aplikace bez nutnosti zasahovat přímo do kódu aplikace.

7.2.1 Systémová architektura

Aplikace byla rozdělena do logických celků (komponent a subkomponent) kvůli své komplexnosti, kvůli udržení přehlednosti a následné údržbě.



Obrázek 23: Architektura SPM

Na obrázku výše jsou znázorněny komponenty, ze kterých je SPM vytvořen. Nejdůležitější komponenta je *Core*, respektive její subkomponenty *core* a *Net*. Tyto subkomponenty dohromady tvoří simulační jádro aplikace.

Na *Core* balíku je závislá *GUI* část aplikace a na těchto dvou komponentách je závislá aplikační část, *App*.

Aplikační část se skládá z subkomponenty *app* a *Swp*, přičemž první zmíněná část obsahuje pouze *zdroje* (angl. resources) a vstupní smyčku aplikace. Detailnější popis zbývajících (sub)komponent je k nalezení níže.

7.2.2 Core komponenta

V této části se budeme věnovat subkomponentě *core*, jelikož balík *Core* slouží jen jako obálka pro komponenty, které vytvářejí jádro (nejen simulační) aplikace a které můžou ostatní vývojáři použít pro vývoj svých podobně založených aplikací bez ohledu na SPM. Pokud bychom tedy chtěli vytvořit modelovací a simulační aplikaci (čehokoliv), pak můžeme stavět právě na výše zmíněném balíku.

Primárním úkolem *core* knihovny je deklarovat rozhraní, na kterých bude aplikace stavět a zároveň poskytnout množinu tříd, nezbytných pro fungování základního rámce aplikace.

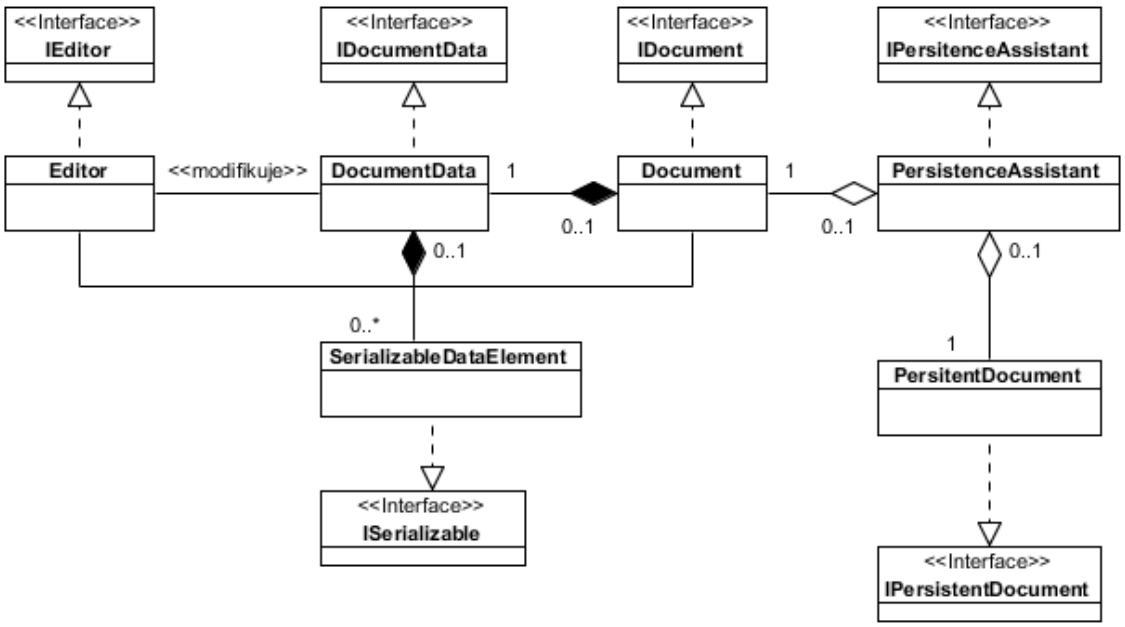
Rozhraní by se daly roztřídit do těchto kategorií:

1. *Grafy* - tato kategorie obsahuje rozhraní, která jsou implementována prvky CPN sítí, jelikož třída CPN sítě používá orientovaný graf jako svůj základ (viz. také 7.2.3).
2. *Dokumenty* - rozhraní spadající do této kategorie definují práci s dokumenty, přičemž jako dokument je považován objekt v paměti (více ve 7.2.4).
3. *Editory* - zjednodušeně řečeno editory slouží k úpravě obsahu *dokumentu* (o editorech se také dočtete ve 7.2.8).
4. *Perspektivy* - perspektivy jsou určeny pro rozlišení kontextu, v jakém se aplikace v danou chvíli nachází (o perspektivách se také dočtete ve 7.2.8).
5. *Perzistence* - do této kategorie spadají rozhraní poskytující mechanismy umožňující *dokumenty* uložit do trvalé paměti (viz. také 7.2.4).
6. *Serializace* - doplňková kategorie (pomocná) ke kategorii *perzistence*, obsahující rozhraní definující rámec serializace.
7. *Pomocné* - kategorie pro pomocná rozhraní, která logicky nespadají pod žádnou jinou kategorii, např. rozhraní pro implementaci *Observer* návrhového vzoru.

A množinu tříd můžeme rozdělit do těchto kategorií:

1. *Grafy* - třídy implementující orientovaný graf a jeho prvky.
2. *Mimetype* - každý dokument musí obsahovat informaci o svém typu, to nám zaručí třídy spadající do této kategorie.
3. *Perzistence* - množina tříd spravující perzistentní dokumenty (více ve 7.2.4).
4. *Serializace* - třída obsahující logiku serializace (manažer).
5. *Typy* - kategorie tříd, definujících základní typy (bool, string, number, enumeration) použité v simulačním jádru (viz. 7.2.3).
6. *Utility* - pomocné třídy (většinou třídy typu manažer).

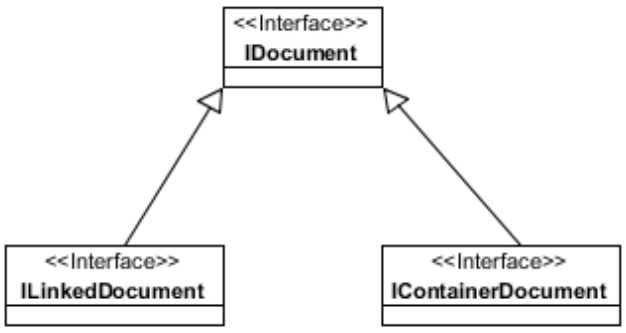
Následující třídní diagram znázorňuje to nejdůležitější z architektury *core* knihovny:



Obrázek 24: Důležitá rozhraní *core* componenty

Základ tvoří *Document*, který vlastní *DocumentData*. Tyto data jsou modifikovány *Editorem* a obsahují seznam objektů realizujících rozhraní *ISerializable*. Pro potřeby uložení dokumentu slouží *PersistenceAssistant*, který ví, jak a kam má dokument uložit.

Rozhraní *IDocument* má dvě specializace *ILinkedDocument* a *IContainerDocument*. *ILinkedDocument* rozhraní je implementováno dokumenty, které můžou být asociovány s jinými dokumenty (provázání dokumentů). Pro dokumenty, které mohou obsahovat jiné dokumenty (určují strukturu), je připraveno rozhraní *IContainerDocument* (obrázek 25).

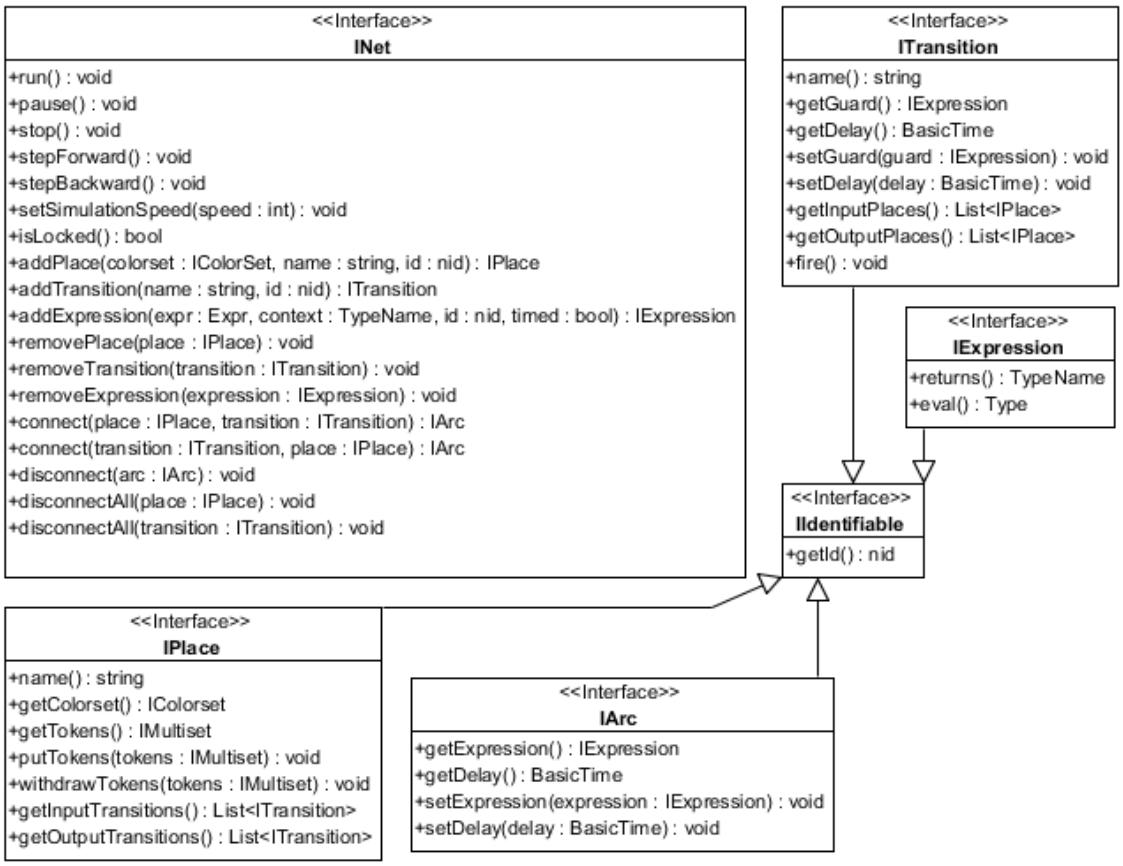


Obrázek 25: *IDocument* rozhraní a dědičnost

7.2.3 Net komponenta

Bezpochyby nejdůležitější knihovna v celém SPM nástroji tvořící mozek simulačního jádra. Je založená na Barevných Petriho sítích. Návrh a implementace této komponenty byl jednoznačně nejnáročnější částí SPM nástroje, jelikož jsem byl nucen navrhnout celou logiku Barevné Petriho sítě včetně interpreteru CPN výrazů. Postup při návrhu jsem rozdělil do těchto kroků:

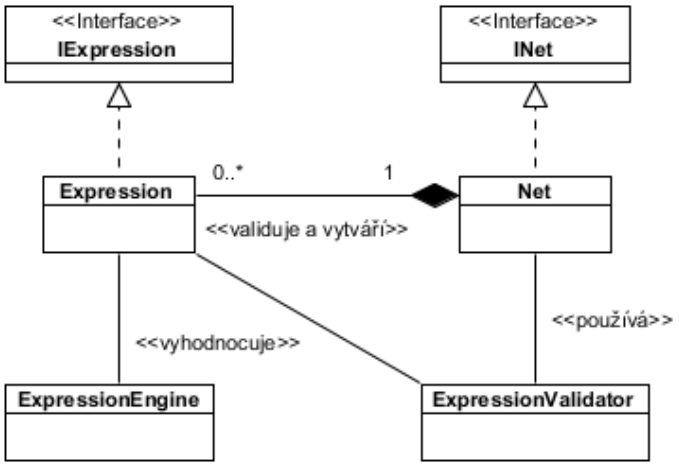
1. *INet rozhraní* - cílem tohoto kroku bylo navrhnout rozhraní Barevné Petriho sítě tak, abychom byli schopni vytvořit strukturu sítě přidáváním *míst* a *přechodů*, měli možnost nastavit hranové a guard výrazy, a abychom mohli běh sítě spustit, popřípadě zastavit. Po dokončení rozhraní bylo úkolem ho zrealizovat ve formě rozšiřitelné třídy *Net*.



Obrázek 26: Rozhraní CPN sítě

Obrázek 26 znázorňuje navržené rozhraní jak pro síť samotnou, tak pro pomocné objekty. Rozhraní je koncipováno tak, že jediným možným způsobem, jak vytvořit objekty tříd, implementujících rozhraní *ITransition*, *IPlace*, *IArc* a *IExpression*, je vytvořit tyto objekty pomocí metod rozhraní *INet*.

2. *Výrazy* - v tomto kroku byl navržen celý systém validace a vyhodnocování výrazů. Důvodem pro rozhodnutí navrhnout a implementovat vlastní řešení byla nezávislost na CPN ML interpreteru, který je součástí CPN tools.

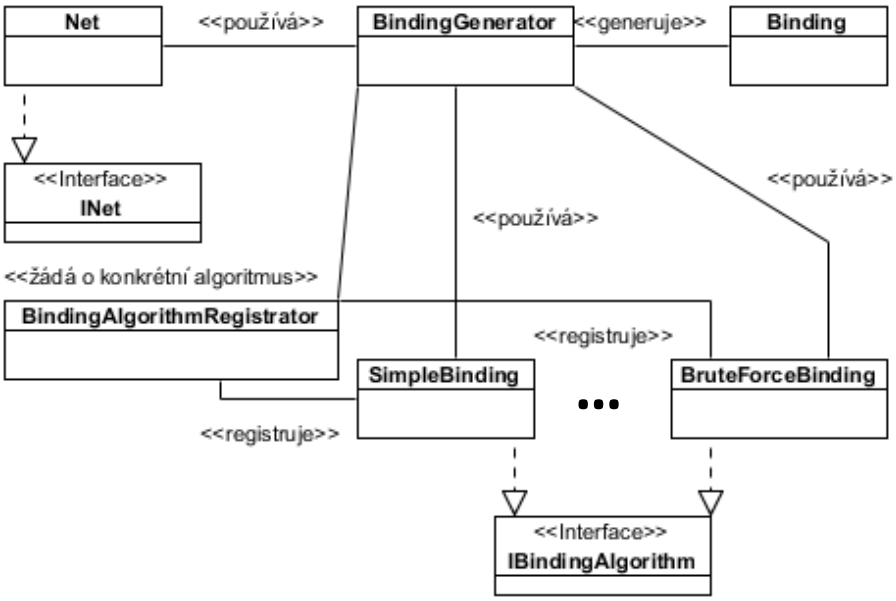


Obrázek 27: ExpressionEngine

Na obrázku výše můžete vidět návrh systému vyhodnocování. Z obrázku 26 je patrné, že jediným způsobem, jak přidat *Expression*, je přes rozhraní CPN sítě. Síť výraz předá třídě *ExpressionValidator*, která ho zvaliduje a pokud byla validace výrazu úspěšná, vygeneruje jeho systémovou interpretaci. Metoda *eval* výrazu poté interně volá *ExpressionEngine*, který vrátí výsledek.

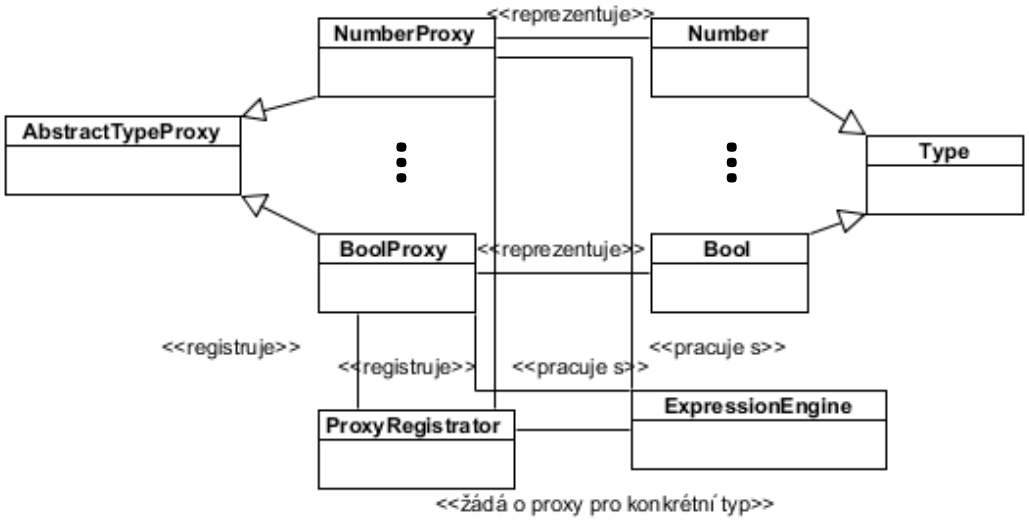
3. *Systém ohodnocování výrazů* - proměnné ve výrazech je třeba ohodnotit, aby mohl být výraz evaluován. Tomuto procesu se říká „binding“. Úkolem tedy bylo navrhnout množinu tříd, zajišťujících ohodnocování proměnných.

   Základem systému pro ohodnocování proměnných je třída *BindingGenerator*, která se rozhoduje, jaký ohodnocovací algoritmus je nejvhodnější pro konkrétní situaci, a která spolupracuje s třídou *BindingAlgorithmRegistrator*, od níž získává instance zaregistrovaných ohodnocovacích algoritmů (viz. obrázek 28).



Obrázek 28: Systém ohodnocování proměnných

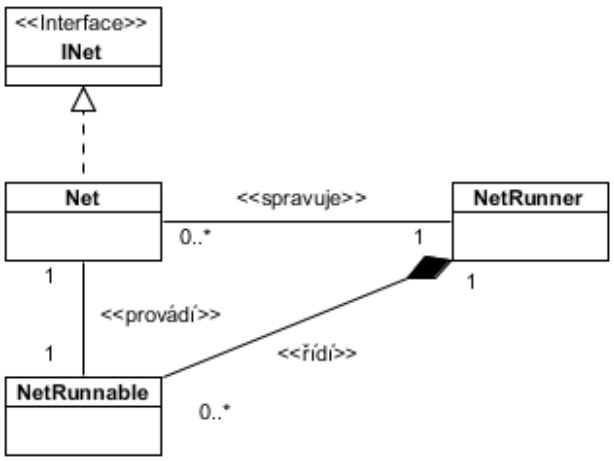
4. *Typy* - knihovna *core* obsahuje definici základních typů použitelných v CPN síti. Tyto typy ovšem nejsou kompatibilní s typy, používanými v systému pro vyhodonocování výrazů. Cílem tohoto kroku bylo navrhnout mechanismus, který toto omezení odstraní.



Obrázek 29: Typový proxy mechanismus

Tento mechanismus využívá systému proxy tříd, které reprezentují daný typ a zároveň jsou kompatibilní s třídou *ExpressionEngine*.

5. *Práce se sítí* - návrh tříd, zastřešujících běh a ovládání CPN sítí ve vláknech.

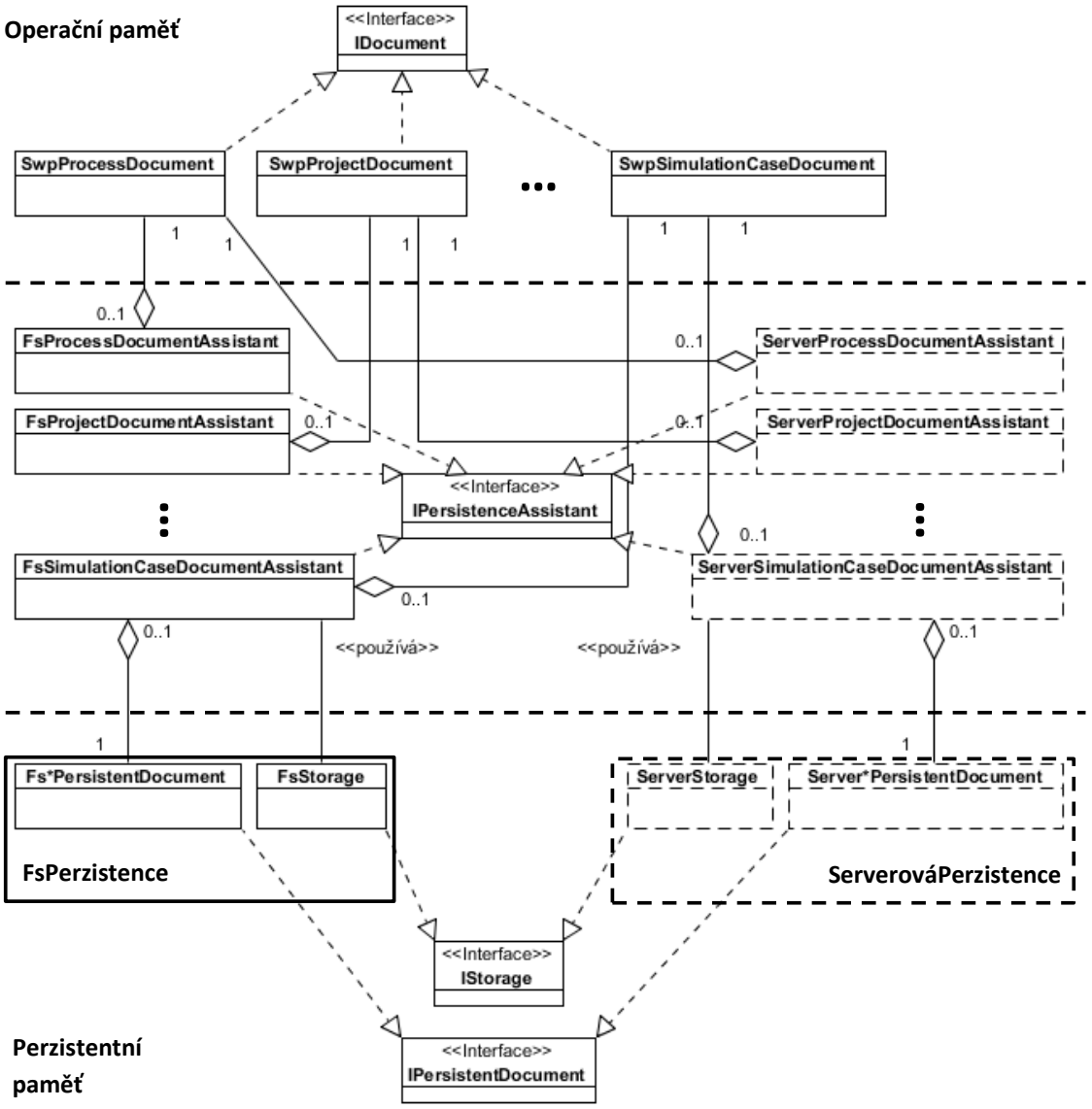


Obrázek 30: Systém pro ovládání chodu sítě

Hlavní roli hraje v tomto systému třída *NetRunner*, která spravuje instance tříd *Net*. Spuštěním sítě (metoda *run*) vnitřně dochází k zavolání API třídy *NetRunner*, která vytvoří pro spouštěnou síť novou instanci třídy *NetRunnable*. Tato třída představuje kontext, ve kterém bude síť běžet (samostatné vlákno). Třída *NetRunner* pak nepracuje přímo se sítí (například pokud přijde požadavek na zastavení běhu sítě), ale s přidruženou instancí třídy *NetRunnable*.

7.2.4 Persistence komponenta

Persistence poskytuje sadu tříd umožňujících uložit dokumenty (instance *IDocument*) do perzistetního úložiště (v základu jsou poskytovány pouze třídy umožňující uložit dokumenty na disk). Návrh této knihovny počítá s možností používat více typů úložišť. Pokud chceme ukládat dokumenty do databáze místo na souborový systém lokálního úložiště, můžeme implementovat rozhraní dle obrázku 31.



Obrázek 31: Architektura perzistence

Z diagramu nemusí být na první pohled patrné, že návrh *Persistence* knihovny (respektive rozhraní z *core* knihovny) počítá i se situací, kdy chceme dokument uložit jako jiný dokument (tedy *uložit jako*, popřípadě *export* funkcionalita). Tuto funkcionalitu zajišťuje třída *Persistence* z *core* komponenty, která spravuje zaregistrované třídy implementující rozhraní *IPersistenceAssistant*. Pro každý dokument může být zaregistrovaný jeden nativní asistent perzistence, který dokument ukládá do nativního perzistetního dokumentu (tedy *uložit* funkcionalita) a množina nenativních asistentů, které umožní uložit (převést) dokument do jiného typu dokumentu. Elementy obsahující * označují skupinu dokumentů se stejnou funkčností (slouží jako hromadný alias).

Třídy implementující rozhraní *IStorage* poskytují na míru vytvořené dialogy pro otevírání/ukládání dokumentů (pro uložení do databáze se může dialog značně lišit od dialogu určeného pro uložení na lokální úložiště). Tyto třídy jsou také přímo zodpovědné za zápis dat do perzistetního úložiště.

7.2.5 Bignumbers komponenta

Tato knihovna slouží k operacím nad velkými čísly. Jelikož vestavěné datové typy neposkytují možnost uložit a pracovat s libovolně velkým číslem, musíme velká čísla reprezentovat jiným způsobem a musíme také přizpůsobit operace s těmito čísly.

Knihovna podporuje celá a reálná čísla a používá se v simulaci pro výpočet nákladů a doby trvání aktivity.

7.2.6 GUI komponenta

Jak název napovídá, tato knihovna obsahuje třídy implementující prvky uživatelského rozhraní. Tyto prvky by se daly roztřídit do těchto kategorií:

1. *Dialogy* - tyto UI prvky slouží k dotázání se uživatele na potřebný vstup, pro SPM nástroj byly vyvinuty speciální dialogy, například dialog umožňující upravovat *odkazy* dokumentů na ostatní dokumenty a dialog pro správu nastavení aplikace (více ve 7.2.8).

2. *Widgety* - pro potřeby SPM byly navrženy některé vlastní widgety (UI komponenty) jako *dokovací widget* a *widget galerie* (viz. 7.2.8).

3. *Související s editory - editor* jako UI prvek sám o sobě musí být umístěn v jiném prvku, ideálně v takovém, který umožní přepínání mezi různými *editory* (respektive otevřenými dokumenty).

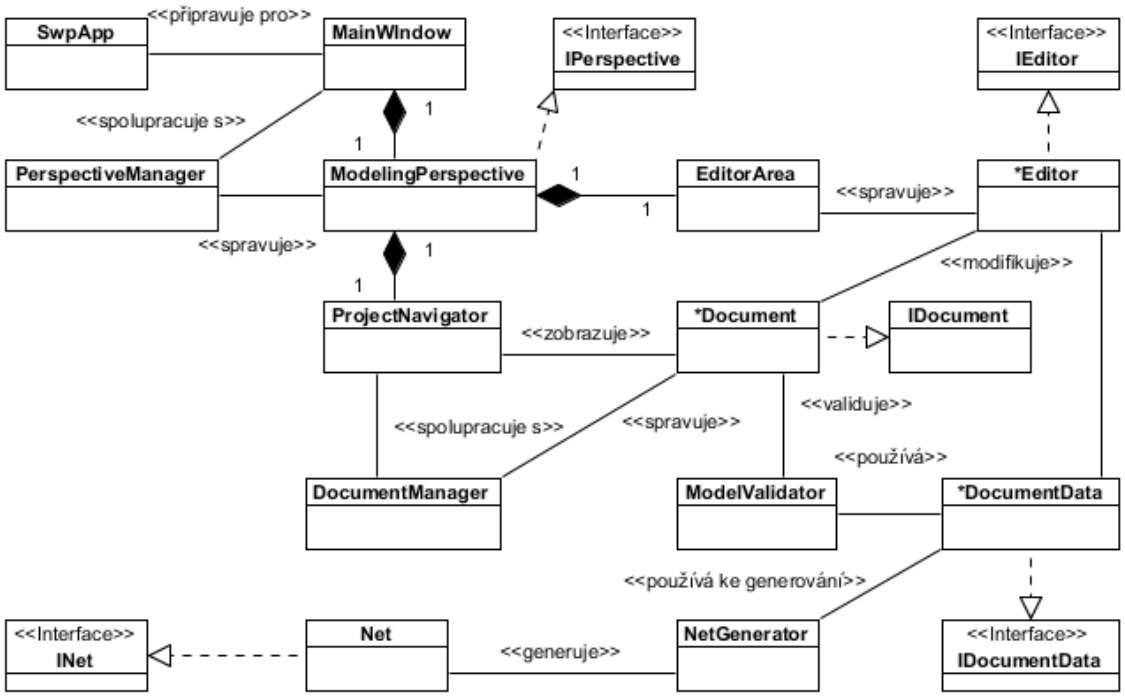
4. *Grafické prvky dokumentu* - tyto třídy slouží jako supertřídy (angl. base class) reprezentující základní grafické prvky dokumentu a jsou to tyto: polygon, obdélník, ellipsa, přímka, cesta.

5. *Události* - pro potřeby SPM byly implementovány speciální typy UI událostí

7.2.7 SWP komponenta

Jedná se o nejobsáhlejší knihovnu v celém nástroji. Třídy této knihovny implementují veškerá důležitá rozhraní z knihovny *core*. Pokud bychom nahlíželi na *core* jako na jedno obrovské rozhraní či abstraktní třídu, pak *SWP* by byla jeho implementace (specializace) pro softwarový proces (proto tedy zkratka SWP). SWP tedy obsahuje třídy všech dokumentů, dat dokumentů, editorů, grafických prvků dokumentů, perspektiv a pomocníků (manažerů) určených (specializovaných) pro softwarový proces.

Všechny koncepty použité v této komponentě vycházejí přímo z konceptů představených v minulých kapitolách.

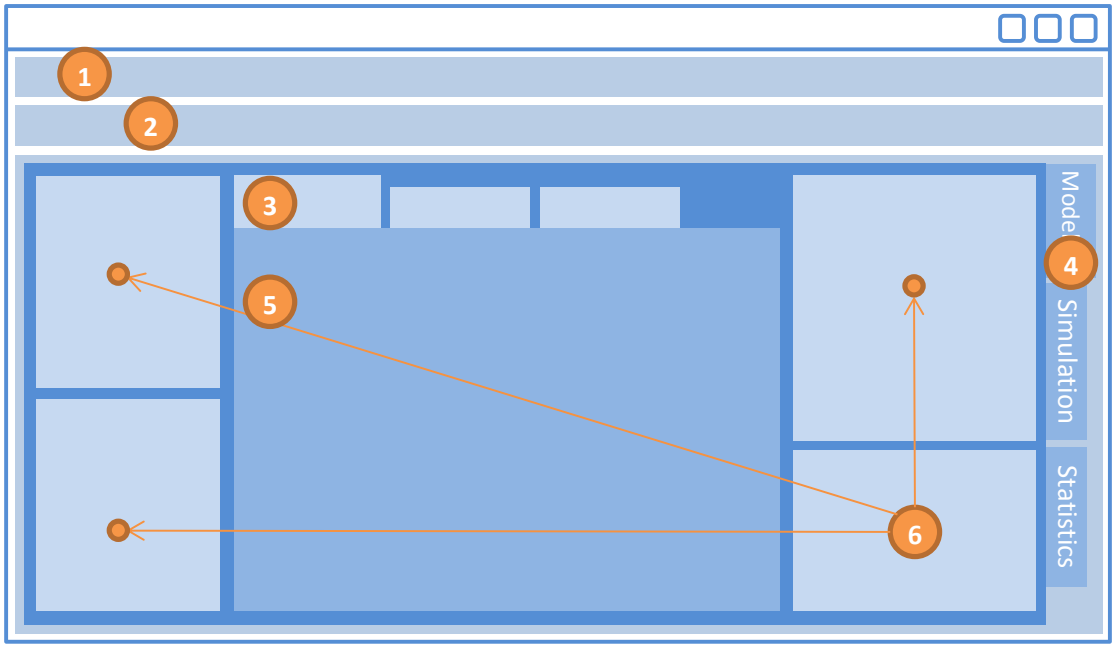


Obrázek 32: Architektura SWP

Na obrázku výše jsou zobrazeny nejdůležitější třídy *SWP* knihovny. Elementy diagramu začínající * nejsou skutečné třídy, ale sdružení tříd, které používají podobný princip (aliasy). Vazba *používá ge generování* mezi třídou *NetGenerator* a zástupnou třídou **DocumentData* je samozřejmě použitelná pouze pro konkrétní třídu. Důležitým prvkem je třída *SwpApp*, která inicializuje vše potřebné pro běh aplikace a proto tedy vazba *připravuje pro* mezi touto třídou a hlavní třídou aplikace *MainWindow*. Hlavní okno aplikace obsahuje tři perspektivy, přičemž na diagramu výše je vyobrazena pouze jedna a to *ModelingPerspective*. Každá perspektiva má svou vlastní instanci třídy *EditorArea*, která spravuje otevřené *editory* a také svého *navigátora* (více ve 7.2.8). Každý navigátor spolupracuje s třídou *DocumentManager*, která se stará o dokumenty (o jejich životní cyklus).

7.2.8 Koncept uživatelského rozhraní

Při návrhu SPM nástroje byla vzata do úvahy použitelnost aplikace, přičemž byl kladen důraz také na přehledné uživatelské rozhraní nástroje. Na obrázku níže jsou představeny hlavní prvky okna aplikace.

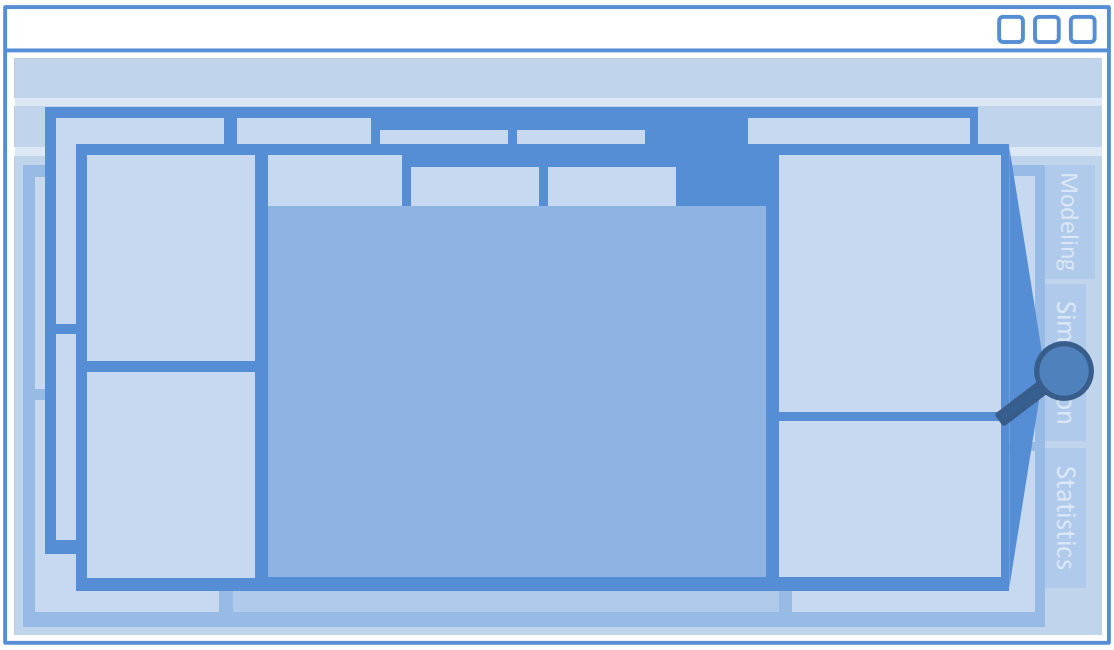


Obrázek 33: Hlavní prvky okna

1. *Hlavní menu aplikace* - slouží k přístupům k akcím aplikace. Tyto akce jsou roztříděny do jednotlivých kategorií, dle určení (submenu).
2. *Toolbar* - nejpoužívanější akce snadno přístupné na jednom místě.
3. *Taby editorů* - každý otevřený *editor* má své ouško (tab). Díky tomu je možné snadno přepínat mezi otevřenými dokumenty.
4. *Taby perspektiv* - slouží k přepínání mezi perspektivami.
5. *Okno editoru* - v tomto okně se zobrazuje obsah dokumentu.
6. *Dokovací okna* - tyto prvky slouží k zobrazení rozličných UI elementů (konkrétní příklady jsou k dispozici níže). Výhodou dokovacích oken je možnost měnit jejich rozestavení, můžeme si je tedy přizůsobit dle libosti.

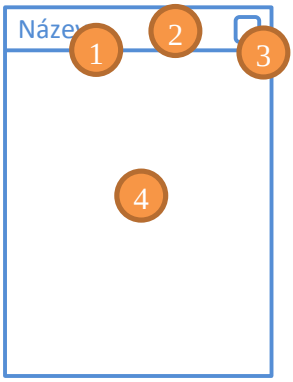
Hlavní okno aplikace je virtuálně rozděleno na tzv. *perspektivy*. Ty slouží k oddělení kontextu práce a ke zvýšení přehlednosti. Úkony prováděné v SPM se dají rozdělit do tří kategorií (modelování, simulace a analýza výsledků simulace). Každá kategorie má svá specifika (specifický workflow a UI prvky) a je vhodné tyto kategorie vizuálně oddělit. Jedním ze způsobů je právě použití perspektiv.

Obrázek 34 ilustruje princip perspektiv, které se chovají jako virtuální plochy. Přepínáním perspektiv můžeme také ovlivňovat některé kontextové akce v hlavním menu a v *toolbaru*.



Obrázek 34: Perspektivy

Jedním z hlavních prvků perspektiv jsou dokovací okna. Jejich anatomie je znázorněna níže.

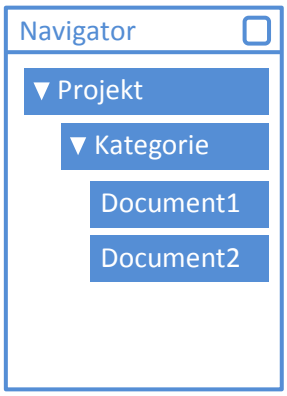


Obrázek 35: Dock widget

1. *Název dokovacího okna*.
2. *Dragovací oblast* - chycením okna za tuto část ho můžeme přesunout jinde.
3. *Tlačítko pro zavření*.

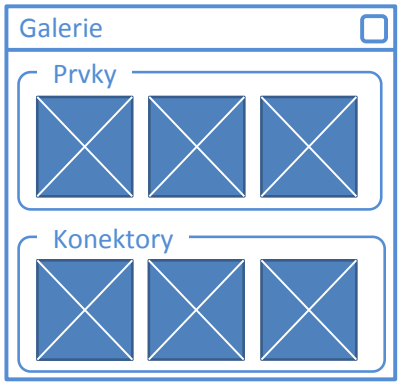
4. *Obsah doku* - jiný grafický prvek vložený do dokovacího okna.

Příkladem obsahu dokovacího okna je *navigátor* (každá perspektiva má svůj navigátor). Navigátor zobrazuje prvky ve stromové struktuře a slouží k rychlé navigaci v projektu, umožňuje také manipulaci s prvky přes kontextové menu.



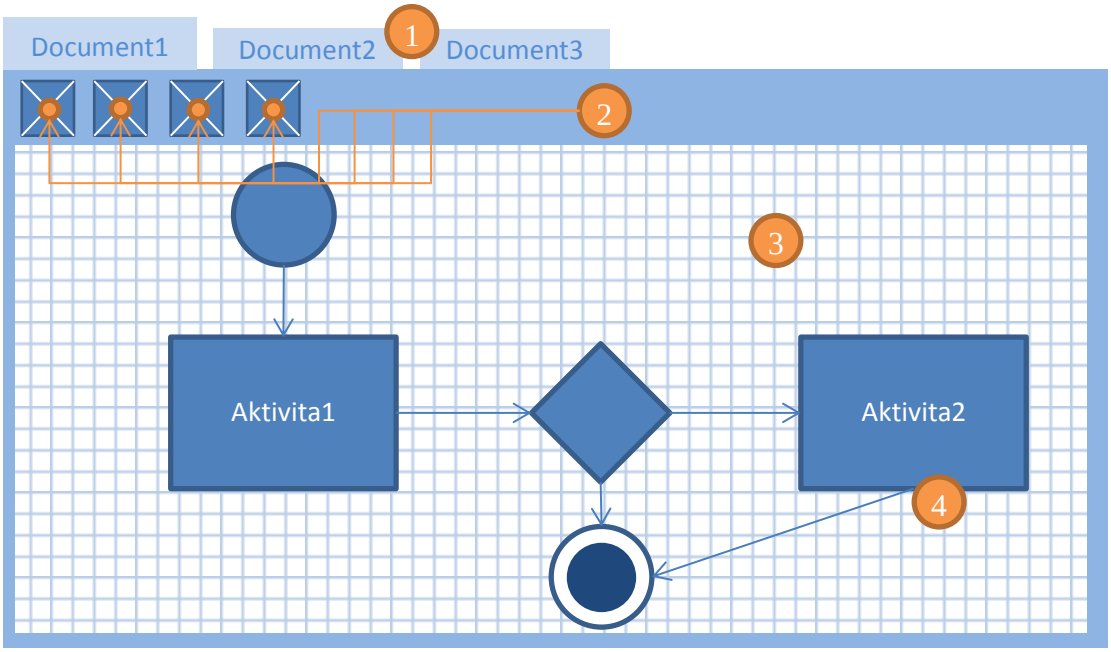
Obrázek 36: Navigátor

Dalším příkladem obsahu dokovacího okna je *galerie*. Tato komponenta slouží k pohodlenému výběru prvků, které chceme vložit do dokumentu. Uživatel může použít buď mechanismus *drag drop* nebo může jednoduše kliknout na konkrétní prvek galerie. Velkou výhodou galerie je, že uživatel vidí náhledy prvků, což pomáhá v rychlé orientaci a vytvoření mentálního modelu.



Obrázek 37: Galerie

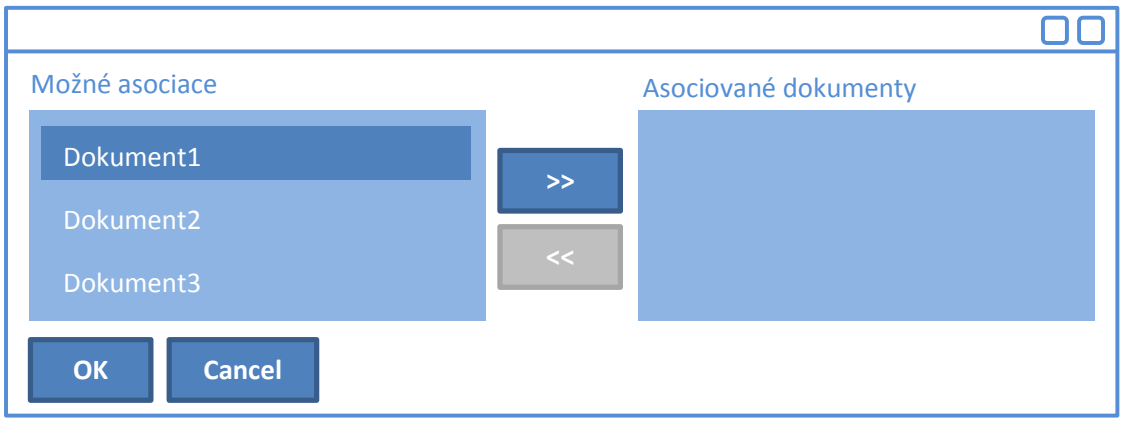
Centrem každé perspektivy je grafický prvek *EditorArea*. Tento prvek nám dovoluje přepínat mezi jednotlivými okny (editory).



Obrázek 38: Editory

1. *Taby pro přepínání mezi editory*.
2. *Akce editoru* - každý typ editoru (pro konkrétní typ dokumentu) obsahuje kontextové akce pro práci s prvky dokumentu.
3. *Oblast dokumentu*.
4. *Prvek dokumentu* - zobrazený v editoru.

V rámci SWP nástroje byly navrhnuty speciální, na míru šité dialogy. Na obrázku 39 je vyobrazen jeden z těchto dialogů. Jeho účelem je poskytnout komfortní způsob, jak upravovat asociované dokumenty konkrétního dokumentu. Na levé straně jsou v seznamu uvedeny dokumenty ještě neasociované s daným dokumentem. Na straně pravé jsou zobrazeny dokumenty, které již jsou s dokumentem asociovány. Dokumenty můžeme přemisťovat z jednoho seznamu do druhého pomocí tlačítek *přidat* a *odebrat* umístěných mezi seznamy.



Obrázek 39: Editor asociací

7.3 Implementace

Před samotnou implementací bylo nutné se rozhodnout pro technologii, na které bude SPM postaven. Kvůli svým předchozím pozitivním zkušenostem s Qt knihovnou jsem se rozhodl právě pro tuto knihovnu.

Qt je knihovna, umožňující vytvářet multiplatformní aplikace. Jako základ používá programovací jazyk c++ ve verzi 03 (od Qt verze 4.8 začaly být začleňovány určité prvky c++ 11). Tato knihovna nabízí obrovské množství tříd z různých odvětví vývoje a značně tím usnadňuje vývojářům práci. Jako nejužitečnější se dají označit třídy pro vytváření grafického uživatelského rozhraní, jelikož vývojáře zbavují nutnosti implementovat GUI nad různými technologiemi různých platforem (WinForms pro Windows, GTK pro Linux).

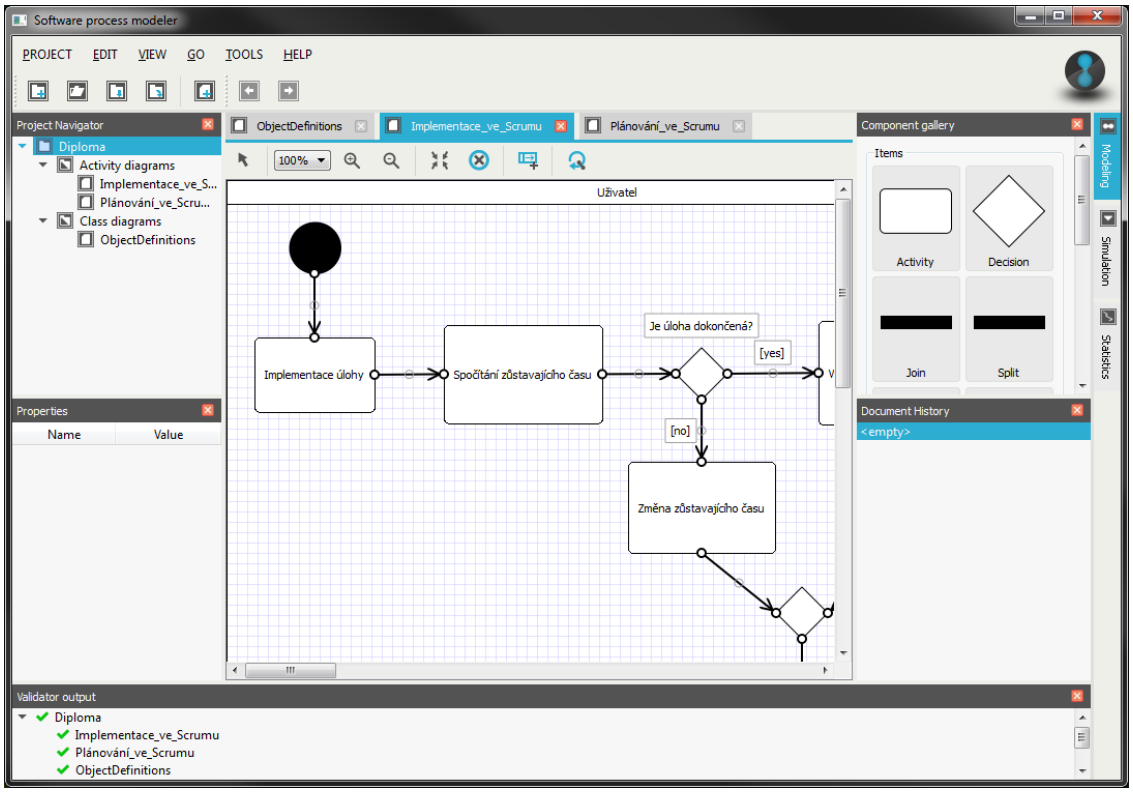
SPM tedy staví na Qt knihovně, konkrétně na verzi Qt 4.7.4, jelikož tato verze byla stabilní (bez chyb) v době začátku samotného vývoje. Technologie c++ standardně nenabízí vývojářům pracovat s typy dynamicky (protože se jedná o kompilovaný programovací jazyk) a nenabízí ani možnost dynamické evaluace výrazu (za chodu programu). Problém s dynamicky definovanými typy byl odstraněn vytvořením supertřídy *Type* a jejích potomků (viz. 7.2.3). Tento problém musel být vyřešen, jelikož *colorsety* a *multisety* představují dynamické typy (spolu s typy jako *enumerace*), které jsou požívány v CPN sítí. Druhý (a závažnější) problém byl vyřešen použitím třídy *ExpressionEngine* (popis ve 7.2.3), která slouží jako zástupce pro *Qt script engine* založený na ECMA skriptu, který je součástí Qt knihovny. Dříve zmíněná třída *ExpressionValidator*, jejímž úkolem bylo převést (nejen) validní CPN výraz na systémovou reprezentaci, tedy převáděla CPN na výraz ECMA skriptu. Průběh validace výrazu využíval podobného principu jaký používá teorie kompilátorů, a to rozložení výrazu na stromovou strukturu tokenů (v angličtině je tento proces známý jako *parsing*) s použitím zásobníku místo rekurzivního algoritmu. V projektu obecně nebyla použita rekurze, veškeré použité algoritmy rekurzivního charakteru byly přepsány do nerekurzivní podoby využívající smyček a pomocných struktur.

Simulace používá pro výpočet reálných hodnot *doby trvání* exponenciální rozdělení, kde hodnota těchto parametrů aktivity slouží jako λ parametr exponencionálního rozdělení, které je vhodné pro modelování teorie front (a procesů).

Důležitou třídou je také *NetGenerator*, která má za úkol převést neformální UML diagram aktivit na Barevnou Petriho síť. Tato třída využívá pravidel pro převod, které byly popsány v kapitole 6 s použitím algoritmu BFS (Breadth First Search, tedy procházení do šířky) pro procházení modelu diagramu aktivit.

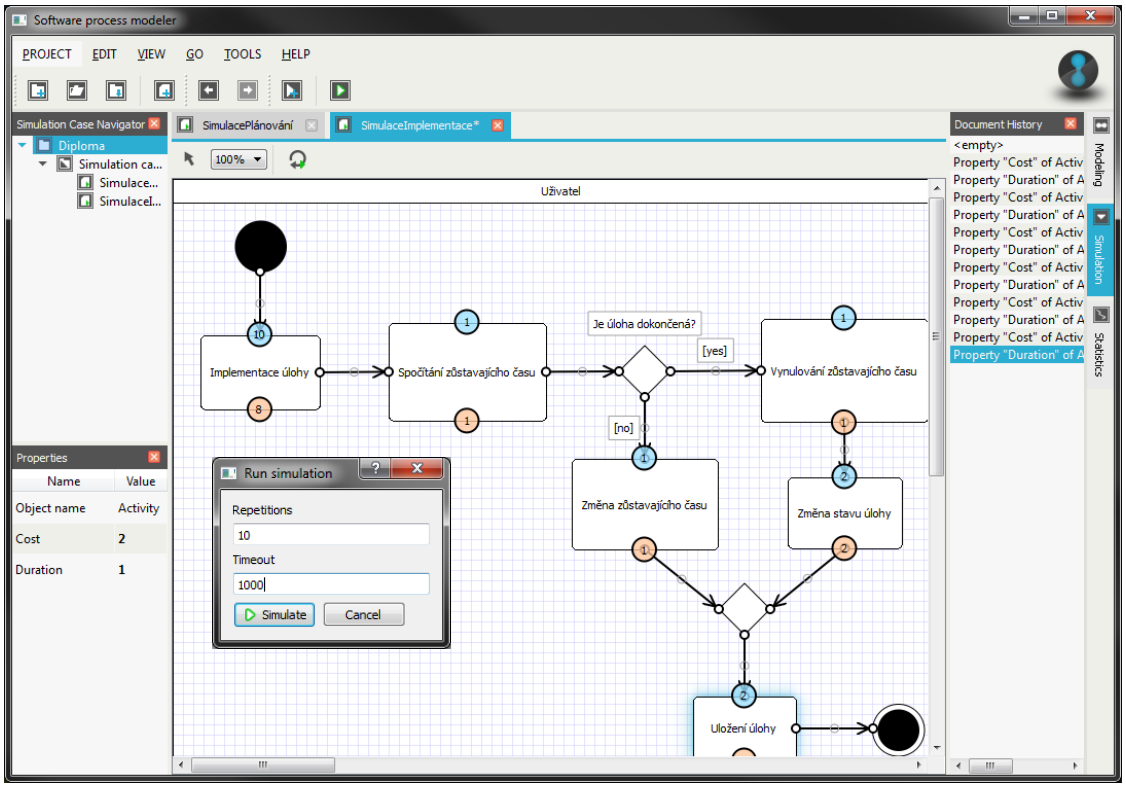
Jedním z cílů při implementaci SPM bylo zajištění kvality výsledného produktu. Kvalita samotného kódu byla zajištěna jak dodržováním kódovacích konvencí jazyka c++ a technologie Qt, tak použitím modulárních testů pro kritické komponenty systému. Aplikace byla testována v omezeném okruhu lidí. S každým milníkem byla také spuštěna diagnostika aplikace na odhalení úniků paměti (angl. *memory leaks*) s použitím nástroje *Dr. Memory* na Windows a *Valgrind* na Linuxu.

Na následujících obrázcích jsou k vidění jednotlivé perspektivy programu.



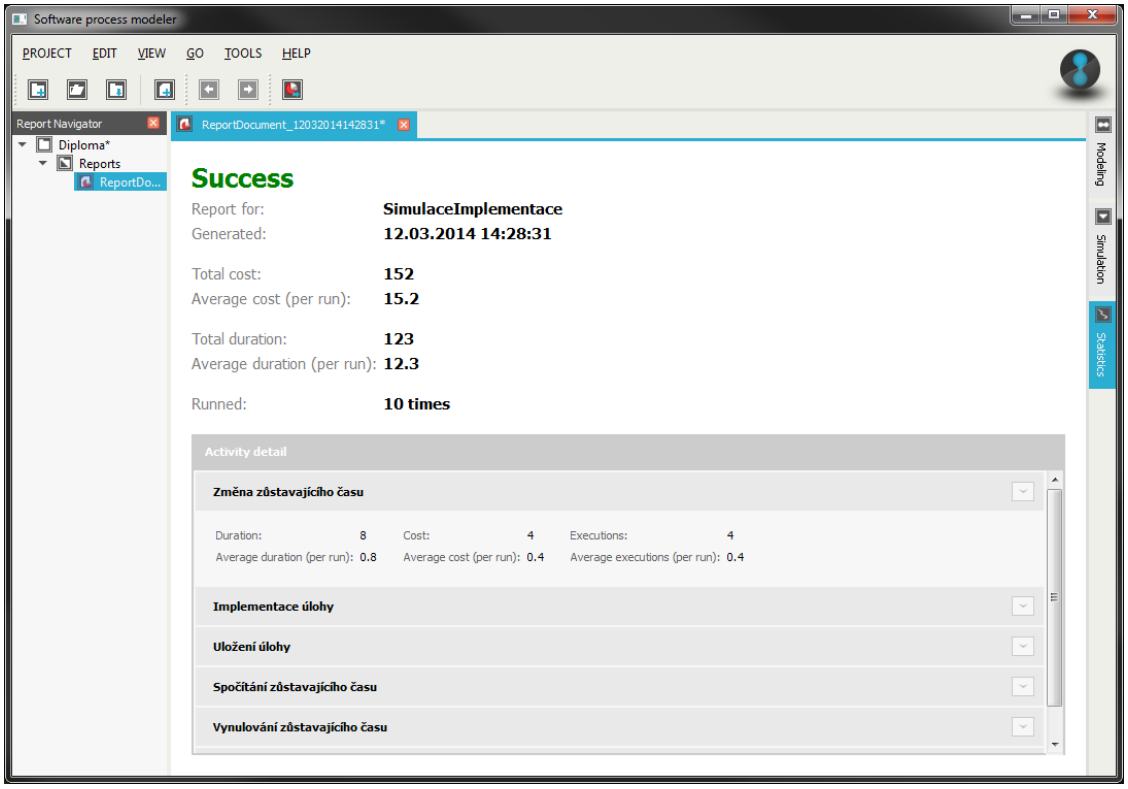
Obrázek 40: Modelovací perspektiva

Na obrázku výše je screenshot *Modelovací perspektivy*, ve které uživatel modeluje složky softwarového procesu pomocí UML diagramů (diagram aktivit a třídní diagram). Můžete zde vidět reálnou podobu prvků popisovaných v kapitole 7.2.8.



Obrázek 41: Simulační perspektiva

Simulační perspektiva je specifická v možnosti určit *cenu* a *dobu trvání* každé aktivity. S prvky simulačního diagramu nelze nijak manipulovat (přesouvat, mazat). Na screenshotu je patrné dialogové okno simulace, ve kterém uživatel nastavuje počet opakování, dobu vypršení a následně spouští simulaci.

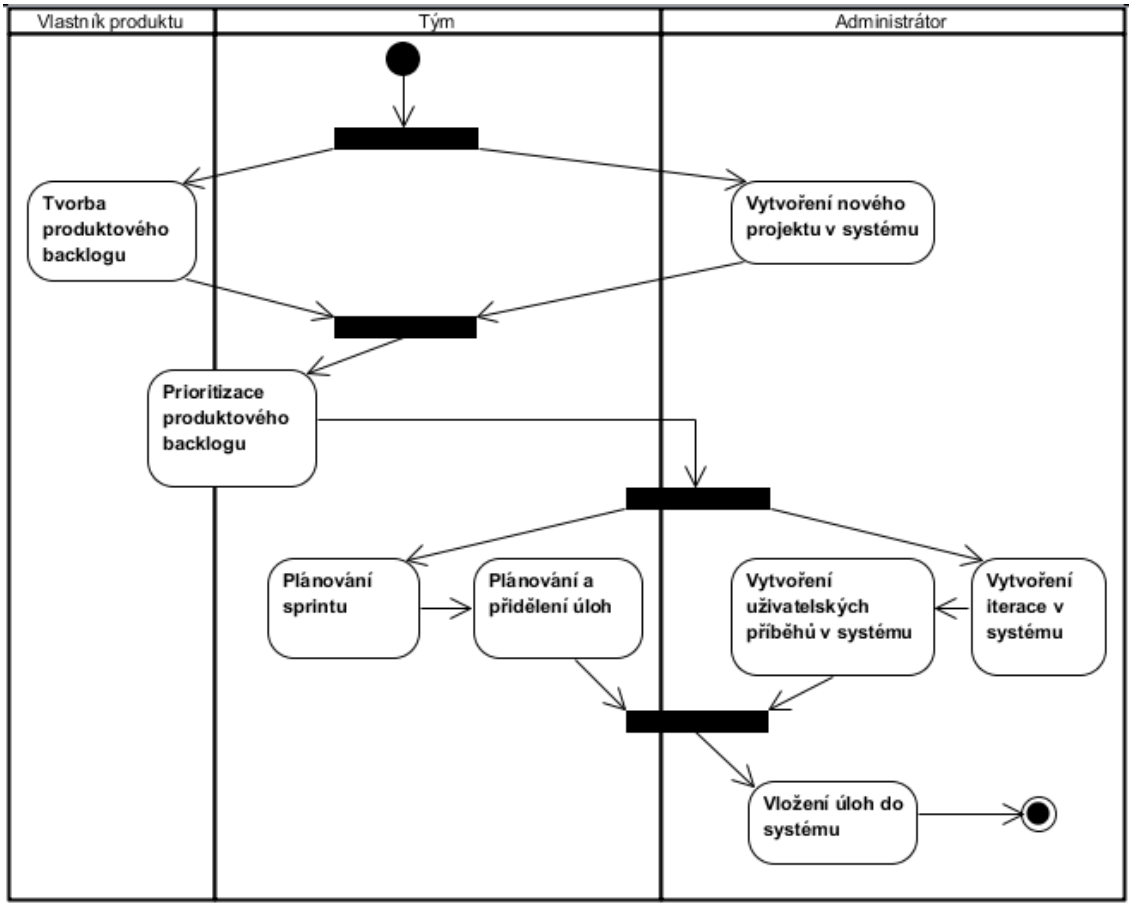


Obrázek 42: Perspektiva výsledků

Na obrázku je vidět výsledek simulace ukázkového procesu. K dispozici jsou údaje o výsledku (úspěch/neúspěch), ceně (celkové, průměrná), době trvání (celkové, průměrná) a detailní údaje každé aktivity. Tento výsledek je možné exportovat do přehledné HTML stránky (kterou je poté možné vytisknout například do PDF).

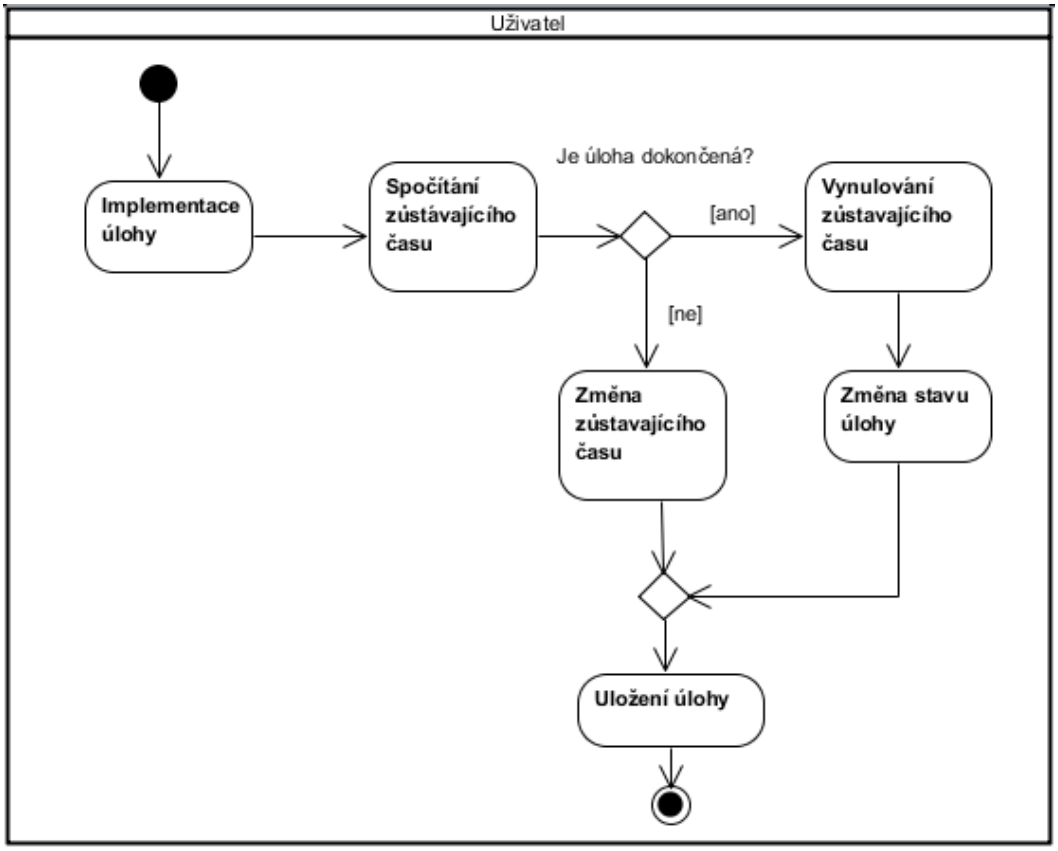
7.4 Ukázková data

Po domluvě s vedoucím diplomové práce a se svolením kolegy Radoslava Štrby jsem pro ukázku použil procesy popisující metodiku Scrum, jež zpracoval v rámci své diplomové práce ([Štr13]). Procesy bylo nutné převést z EPC diagramů do UML diagramů aktivit.



Obrázek 43: Specifikace procesu plánování ve Scrum metodologii

Na obrázku výše je vymodelovaný proces plánování ve Scrum metodologii. Z důvodu úspory místa byla vynechána datová složka. *Product owner* elementy EPC diagramu byly převedeny na plavecké dráhy.



Obrázek 44: Specifikace procesu implementace ve Scrum metodologii

Na diagramu výše je znázorněn proces implementace ve Scrum metodologii. Podobně jako u předchozího diagramu byla z důvodu úspory místa vynechána datová složka.

Procesům byly v SPM přiřazeny tyto hodnoty *ceny* a *doby trvání*:

Název aktivity	**Cena**	**Doba trvání**
Tvorba produktového backlogu	10	8
Vytvoření nového projektu v systému	2	1
Prioritizace produktového backlogu	20	10
Plánování sprintu	10	6
Vytvoření iterace v systému	2	1
Plánování a přidělení úloh	10	8
Vytvoření uživatelských příběhů v systému	10	8
Vložení úloh do systému	2	1

Tabulka 1: Hodnoty *ceny* a *doby trvání* procesu plánování ve Scrum metodologii

Název aktivity	Cena	Doba trvání
Implementace úlohy	10	8
Spočítání zůstavajícího času	1	1
Vynulování zůstavajícího času	1	1
Změna zůstavajícího času	1	1
Změna stavu úlohy	2	2
Uložení úlohy	2	1

Tabulka 2: Hodnoty *ceny* a *doby trvání* procesu implementace ve Scrum metodologii

Tyto hodnoty byly zvoleny po konzultaci s kolegy, aktivně používajícími Scrum (hodnota parametru *doba trvání* definuje hodnotu parametru λ exponencionálního rozdělení, viz. zmínka v kapitole 7.3).

Poté byly procesy simulovány s použitím deseti opakování. Tento (relativně) malý počet opakování byl zvolen díky faktu, že proces plánování neobsahuje žádné rozhodování a proces implementace pouze jedno rozhodování. Pokud bychom pracovali s komplexnějšími procesy s mnoha bloky rozhodování, bylo by vhodnější zvolit větší počet opakování (abychom prošli každou větev). Počet opakování by mohl být zvýšen také z důvodu zpřesnění výsledných hodnot *doby trvání*.

Výsledek	Úspěch
Celková cena	660
Průměrná cena	66
Celková doba trvání	479
Průměrná doba trvání	47,9

Název aktivity	Cena	Doba trvání	Počet spuštění
Tvorba produktového backlogu	10	8	10
Vytvoření nového projektu v systému	2	1	10
Prioritizace produktového backlogu	20	10	10
Plánování sprintu	10	6	10
Vytvoření iterace v systému	2	1	10
Plánování a přidělení úloh	10	8	10
Vytvoření uživatelských příběhů v systému	10	8	10
Vložení úloh do systému	2	1	10

Tabulka 3: Výsledek simulace procesu plánování ve Scrum metodologii

Výsledek	Úspěch
Celková cena	152
Průměrná cena	15,2
Celková doba trvání	123
Průměrná doba trvání	12,3

Název aktivity	**Cena**	**Doba trvání**	**Počet spuštění**
Implementace úlohy	10	7,3	10
Spočítání zůstavajícího času	1	2,1	10
Vynulování zůstavajícího času	0,6	0,9	6
Změna zůstavajícího času	0,4	0,8	4
Změna stavu úlohy	1,2	1,5	6
Uložení úlohy	2	1,3	10

Tabulka 4: Výsledek simulace procesu implementace ve Scrum metodologii

V detailu aktivit v tabulkách 3 a 4 jsou použity u *ceny* a *doby trvání* průměrné hodnoty ze všech opakování, u *počtu spuštění* jsou použity absolutní hodnoty (celkové).

U obou výsledků simulací se vyskytuje u *výsledku* hodnota „úspěch“, to značí, že simulace procesu proběhla ve všech opakováních v pořádku. Simulace má bezproblémový průběh pokud:

- Se proces nezacyklí.
- Pokud nenastane *zamknutí*, proces tedy nemohl skončit. Zamknutí může nastat v případě, kdy prerekvizitou aktivity bylo spuštění jiné aktivity a ta se nikdy neprovedla.
- Simulace skončí dřív, než vyprší časový limit.

8 Závěr

Moderní doba žádá moderní přístup k vývoji aplikací a klade čím dál větší nároky na kvalitu výsledné aplikace a rychlost vývoje. S těmito problémy nám mohou pomoci kvalitní softwarové procesy, podpořené formálními metodami pro modelování a simulaci jak statické, tak dynamické části softwarového procesu.

Tato diplomová práce se zabývá podporou dynamické části softwarového procesu. Postupně byly evaluovány různé metody modelování a simulace, abychom posléze vybrali nejvhodnější kandidáty pro podporu dynamické části softwarového procesu.

Dynamickou část softwarového procesu jsme podpořili použitím převodu ze semi-formální modelovací metody (UML) do formálních Barevných Petriho sítí, které jsou posléze použity pro simulaci a verifikaci procesu. Spojili jsme tak to nejlepší z obou světů. Výsledkem je možnost používat pohodlnou semi-formální metodu pro modelování procesu, bez nutnosti hlubokých znalostí formálních metod a následný automatizovaný (skrytý) převod na formální metodu, která nabízí silný matematický základ a je vhodná pro simulaci a analýzu procesu.

8.1 Dosažené výsledky

Primárním cílem diplomové práce byl návrh a implementace modelovacího a simulačního nástroje pro podporu softwarových procesů. Tento cíl se podařilo splnit a vznikl nástroj Software Process Modeler. Jedná se o nativní, *multiplatformní* aplikaci s přívětivým uživatelským rozhraním, používající rozšířenou a oblíbenou technologii UML (diagramy aktivit, třídní diagramy) pro modelování procesů a *formální* Barevné Petriho sítě jako simulační jádro. Jde o unikátní nástroj (z hlediska použitého simulačního jádra), který kráčí ve šlépějích výborného BP studia, určeného pro modelování byznys procesů. Dalšími cíly bylo nastudování a zhodnocení používaných metod modelování a simulace, vhodných pro byznys (respektive softwarové) procesy. V kapitolách 3 a 4 byli představeni nejvýznamnější zástupci metod modelování a simulace, spolu s výčtem jejich kladů a záporů. V kapitole 6 byl také popsán proces převodu vybrané semi-formální metody na metodu formální.

Shrnutí:

- Nastudování a výběr nejvhodnějších kandidátů metod modelování a simulace pro podporu softwarového procesu
- Převod semi-formálních UML diagramů aktivit na formální Barevné Petriho sítě.
- Simulační jádro používající Barevné Petriho sítě.
- Nativní, multiplatformní aplikace pro modelování, simulaci a podporu softwarového procesu.

8.2 Plány do budoucna

Ve vývoji nástroje Software Process Modeler bych rád pokračoval i nadále, primárně v rámci doktorského studia.

Jak bylo uvedeno na začátku této kapitoly, našim cílem jsou kvalitní softwarové procesy podpořené formálními metodami. V rámci této diplomové práce jsem se zaměřil na podporu dynamické části softwarového procesu. V budoucnu bych rád rozšířil SPM o formální modelování statické části softwarového procesu s použitím *ontologických slovníků*. Tímto rozšířením by vznikl jeden z nejkomplexnějších nástrojů pro modelování, simulaci a podporu softwarového procesu.

Rád bych také rozšířil a vylepšil stávající funkcionalitu nástroje o:

- Komplexnější možnosti modelování.
- Kompletní podporu UML2 diagramů aktivit a třídních diagramů.
- Možnost spustit více simulací ve stejném čase.
- Přidání *step-by-step* provádění simulace s vizuální odezvou (debugování).
- Možnost pracovat přímo s formálním modelem (Barevnou Petriho sítí) - vhodné pro zkušené uživatele.

9 Seznam literatury

[Czo11] CZOPIK, Jan. *Absolvování individuální odborné praxe*. Ostrava, 2011.

[Dlo07] DLOUHÝ, Martin, Jan FÁBRY, Martina KUNCOVÁ a Tomáš HLADÍK. *Simulace podnikových procesů*. Brno: Computer Press, 2007. ISBN 978-80-251-1649-4.

[Eva05] EVANGELISTA, Sami, Serge HADDAD a Jean-Francois PRADAT-PREYE. *Syntactical Colored Petri Nets Reductions*. Springer, 2005. ISBN 978-3-540-29209-8.

[Gar02] GARRIDO, José Luis a Miguel GEA. *A Coloured Petri Net Formalisation for a UML-Based Notation Applied to Cooperative System Modelling*. Springer, 2002.

[IDE10] IDEF. *IDEF* [online]. 2010. Dostupné z: `http://www.idef.com/`

[Jen09] JENSEN, Kurt a Lars Michael KRISTENSEN. *Coloured Petri Nets: modelling and validation of concurrent systems* [online]. Dordrecht: Springer, 2009. ISBN 978-3-642-00283-0. Dostupné z: `http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-00283-0`

[Kos10] KOŠINÁR, Michal. *Design and Utilization of Knowledge Bases for Software Processes*. Ostrava, 2010.

[Mil03] MILDEOVÁ, Stanislava a Viktor VOJTKO. *Systémová dynamika*. Praha, 2003. ISBN 80-245-0626-2.

[Mil13] MILDEOVÁ, Stanislava. *Systémová dynamika: discplína pro zkoumání měkkých systémů* [online]. Praha, 2013.
Dostupné z: `http://aip.vse.cz/index.php/aip/article/download/54/35`

[Nia11] NIAZI, Muaz a Amir HUSSAIN. *Agent-based computing from multi-agent systems to agent-based models: a visual survey*. Scientometrics, 2011.

[OWL12] . W3C. *OWL 2 Web Ontology Language Document Overview (Second Edition)* [online]. 2012. Dostupné z: `http://www.w3.org/2012/pdf/REC-owl2-overview-20121211.pdf`

[Pec07] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[Pet81] PETERSON, James Lyle. *Petri Net Theory and the Modeling of Systems*. N.J.: Prentice-Hall, 1981. ISBN 978-013-6619-833.

[Sta08] STAINES, Tony Spiteri. *Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets*. Belfast, 2008.

[Štr13] ŠTRBA, Radoslav. *Aplikace softwarové podpory SCRUM*. Ostrava, 2013.

[Von02] VONDRÁK, Ivo. *Úvod do softwarového inženýrství* [online]. Ostrava, 2002. Dostupné z: http://vondrak.cs.vsb.cz/download/Uvod_do_softwaroveho_inzenyrstvi.pdf

[Von04] VONDRÁK, Ivo. *METODY BYZNYS MODELOVÁNÍ* [online]. Ostrava, 2004. Dostupné z: http://vondrak.cs.vsb.cz/download/Metody_byznys_modelovani.pdf

[Wat08] WATSON, Andrew. *Visual Modelling: past, present and future* [online]. OMG, 2008. Dostupné z: http://www.uml.org/Visual_Modeling.pdf

[Whi04] WHITE, Stephen A. . *Introduction to BPMN* [online]. IBM, 2004. Dostupné z: http://www.omg.org/bpmn/Documents/Introduction_to_BPMN.pdf

[Wol98] WOOLRIDGE, Michael. *Agent-based computing. Interoperable Communication Networks* [online]. London, 1998. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.66.850&rep=rep1&type=pdf

[Wol09] WOOLRIDGE, Michael. *An introduction to multiagent systems - Second Edition*. London: Wiley, 2009. ISBN 978-0470519462.

Přílohy

Příloha ve formě CD obsahuje zdrojové kódy programu, distribuční balíčky Software Process Modeleru pro Windows a Linux, jednoduchý manuál ve formě HTML stránky a ukázkový projekt s modely diagramů, které byly představeny v kapitole 7.4.