VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

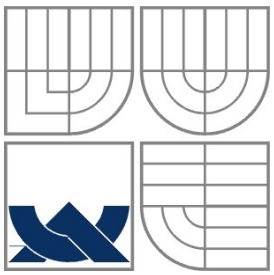
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELIGENT SYSTEMS

MODELEM ŘÍZENÝ NÁVRH KONFERENČNÍHO SYSTÉMU

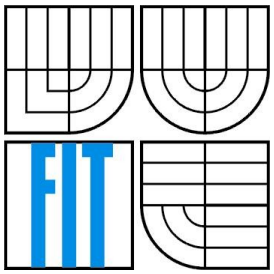
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. MATĚJ CAHA
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELIGENT SYSTEMS

MODELEM ŘÍZENÝ NÁVRH KONFERENČNÍHO SYSTÉMU

MODEL BASED DESIGN OF THE CONFERENCE SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MATĚJ CAHA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2013

Abstrakt

Modelem řízený návrh počítačových systémů je zaměřen na vytváření funkčních proveditelných modelů ve fázi návrhu systému, což vede k možnosti rychlého prototypování, testování a validace systému již v této fázi vývoje systému. Konkrétní implementace je poté vytvářena automaticky na základě překladových pravidel často specifikovaných vlastním formálním modelem. Místo transformace modelu na konkrétní implementaci může být funkční model zachován až k nasazení výsledného systému, čemuž se říká model continuity. Tímto způsobem lze efektivně využít metody modelování a simulace po celou dobu vyvíjení systému.

Pro tento účel je na fakultě informačních technologií na VUT v Brně vyvíjen nástroj SmallDEVS/PNtalk, který umožňuje programování modelů pomocí formalismů DEVS a OOPN. Objektově orientované Petriho sítě (OOPN) spojují výhody objektového programování a formálního aparátu Petriho sítí a dají se použít pro popis struktury a chování méně složitých systémů. Naopak DEVS je formalismus pro popis složitých diskrétních systémů řízených událostmi, kdy je možno systém skládat ze subsystémů (komponent), kde každý subsystém opět může být složen z dalších subsystémů. DEVS tedy definuje spojované komponenty, které mají propojení mezi svými vstupními a výstupními porty, a atomické komponenty, které mohou mít své chování popsány jiným formalismem, jako třeba OOPN, nebo konečný automat. Dohromady se jedná o silný modelovací aparát, jehož modelování/programování nám ulehčí nástroj SmallDEVS/PNtalk, postavený na čistě objektově orientovaném jazyce Smalltalk.

Abstract

Model-based design of computer systems is focused on creating functional executable models in the phase of design system, which leads to possibility of prototyping, testing and validation of the system at this stage of system development. The specific implementation is then created automatically based on the translation rules, which are often specified by its own formal model. The other way is to keep functional executable models in application, which is called model continuity. This is how you can effectively use the methods of system modeling and simulation throughout the developing process.

For this purpose a tool SmallDEVS/PNtalk was developed on faculty of information technology on VUT Brno. This tool allows programming models based on formalisms DEVS and OOPN. Object oriented Petri nets (OOPN) combine the advantages of OOP and formal apparatus of Petri nets. OOPN can be used to describe of structure and behavior of less complex systems. DEVS formalism is used to describe of complex discrete event systems, which may be composed of subsystems (components), where each subsystem can be an atomic component, or coupled component composed of other subsystems. Coupling of components is provided by input/output ports of component. Behavior of atomic components may be describe by other formalisms like OOPN or finite state machine. Altogether it is a powerful modeling apparatus and the framework SmallDEVS/PNtalk facilitates work on modeling and programming with it. SmallDEVS/PNtalk is based on purely object-oriented language Smalltalk.

Klíčová slova

Softwarové inženýrství, analýza a návrh počítačových systémů, , modelem řízený návrh, MBD, simulací řízený návrh, SBD, MDA, DEVS, OOPN, SmallDEVS, PNtalk, modelování a simulace.

Keywords

Software engineering, system analysis and design, model-based design, MBD, simulation-based design, SBD, MDA, DEVS, OOPN, SmallDEVS, PNtalk, modeling and simulation.

Citace

Matěj Caha: Modelem řízený návrh konferenčního systému, diplomová práce, Brno, FIT VUT v Brně, 2013

Modelem řízený návrh konferenčního systému

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Matěj Caha
xx.xx.2013

Poděkování

Děkuji svému vedoucímu práce panu Ing. Radku Kočímu, Ph.D. za uvedení do problematiky a za následné komentáře ke struktuře a obsahu této práce.

Obsah

1. Úvod

** zatím stejný jako v SEP. **
V této práci budou diskutovány aspekty návrhu softwarových systémů metodikou zvanou *modelem řízený návrh* (MBD - Model-based design). Shrneme zde dosavadní historii návrhu systémů, dále nastíníme princip metodiky MDA (Model Driven Architecture) dle specifikace skupiny OMG [1], poté navážeme na metodiku a technologii vyvíjenou na naší fakultě výzkumnou skupinou modelování a simulace systémů při ústavu inteligentních systémů (více v [2]).

Softwarové inženýrství se již spoustu let zabývá vývojem software a řešením problémů, které při vývoji vznikají, společně s vyvíjením nových technik k usnadnění a urychlení vývoje systémů. V současné době existuje mnoho metod vývoje SW, ze kterých si můžeme připomenout klasický Vodopádový model, pokročilejší Spirálový model, inkrementální iterační model vývoje, unifikovaný RUP nebo v neposlední řadě zmiňme agilní metodologie jako například Extrémní programování, nebo SCRUM.

Všechny zmíněné metodiky ve svých fázích vývoje více či méně využívají modely pro reprezentaci systému, nebo jeho částí. Modelování je pro nás důležité, neboť nám umožňuje postihnout ony vlastnosti systému, které nás zajímají. Pravděpodobně nejznámější modelovací prostředek je UML (Unified Modeling Language, [3]), který je však pouze informativní (vizualizační), lépe řečeno, neumožňuje ve své základní podobě provádění vytvořených modelů.

V oblasti softwarového inženýrství došlo v poslední době k rozvoji přístupu, který využívá modelování ve smyslu simulovatelných modelů a to v takém rozsahu, že je lze považovat za programovací jazyk [4]. Tento přístup dostál mnoha označení, z nichž některá jsou *Model-based design* (MBD), také *Model-driven design* (MDD) a *Model-driven architecture* (MDA), nebo trochu komplexnější název *Model and Simulation based design* (MSBD). Všechny ale v podstatě značí to, že vývoj počítačového systému je podložen proveditelným modelem, například *xUML* (Executable UML, viz. kap. 3.1), nebo jinými formalismy, jako *Petriho sítě*, případně *DEVS*, které jsou probrány v kapitole 4. Tento proveditelný model může být testován již v průběhu analýzy, což je nesporná výhoda oproti klasickým metodám vývoje, kdy se systém testoval až po fázi implementace. Fáze implementace u modelem řízených metod návrhu může být potom prováděna manuálně, kdy programátoři vytvářejí kód na základě odsimulovaného modelu systému, nebo také (polo)automaticky, kdy musí být prvkům modelu jasně stanoveny pravidla transformace na zdrojový kód. Tuto transformaci mohou IT analytici částečně uzpůsobovat svým potřebám pomocí zavedení *Metaúrovní*, jako například *Meta-Object Facility* (MOF) u UML. Tato problematika transformace modelu je částečně popsána v kapitole 3.

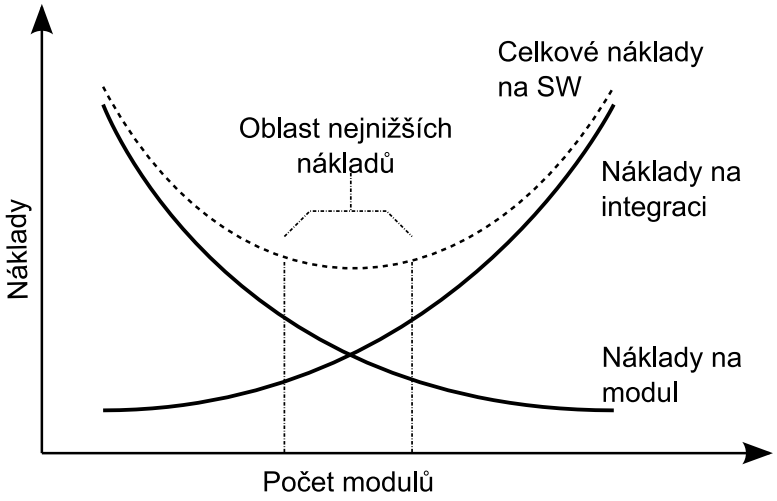
Kapitola 4 potom pojednává o významu modelování a simulace při vývoji systému a diskutuje výzkum probíhající na fakultě informačních technologií na VUT v Brně (viz. [4]).

2. Historie - cesta k modelem řízenému návrhu

Vývoj softwarového průmyslu od svého počátku dostál mnoha změn a postupně zde docházelo k vrstvení abstrakcí, díky kterých můžeme dnes psát programy o mnohem pohodlněji, než tomu bylo při psaní prvních počítačových programů. Mluvíme-li o programech ukládaných do paměti počítače, kdy měly počítače paměť počítanou v kB a výkon udávaný v kHz, existovala v podstatě pouze jedna vrstva abstrakce, která mapovala jedna k jedné instrukční sadu jazyka na přímé provádění strojového kódu [5]. Podobně tomu je i při návrhu počítačových systémů, kde také docházelo ke zvyšování počtu abstrakčních vrstev a vývoji různých metodik k ulehčení, ale hlavně urychlení, snížení chybovosti kódu a tedy snížení ceny vývoje. Pojďme si vývojem návrhu systémů tedy projít od začátku. Následující řádky popisující historii návrhu systémů jsou volně převzaty z [5].

Jak jsem psal v úvodu této kapitoly, na počátku vývoje počítačových systémů byly počítače a vývoj software dosti omezené. Každý programátor psal programy de facto ve strojovém kódu a vytvářel tak malé programy, které byly jednoduše srozumitelné jednomu člověku, který na nich pracoval. Počítače začaly být větší a výkonnější a společně s tím rostla i složitost vyvíjených systémů. S rostoucími nároky na složitost počítačových systému se programátoři museli spojit do týmů a pracovat na vývoji systému společně. Tím vznikla potřeba stanovit v týmu jistá pravidla psaní kódu. Program potřeboval být více strukturovaný čímž na začátku 70. let 20. století vznikl trend zvaný *Strukturované Programování*, který umožnil programátorům strukturovat kód do dílčích částí ve formě tzv. funkcí, případně procedur a tyto části používat na více místech programu. Pokročilejší strukturování vedlo ke vzniku ucelených funkčních bloků zvaných moduly. Programovací jazyky (jako třeba C, nebo C++) umožňující modulární programování jsou označovány jazyky třetí generace. Tento trend přinesl výhody dělení práce nad strukturovanými programy, avšak návrh systému zůstal tam kde byl, čili komplexní, nepřehledný, obtížný a náročný, což se s rostoucími programy ještě umocňovalo.

Možná tedy nebyl problém v implementaci kódu, ale v dřívější fázi životního cyklu softwaru (dále SW). Edward Yourdon a Larry Constantine to takto napadlo a uskutečnili posun směrem k *Strukturovanému Návrhu* SW. Navrhli dvě klíčová kritéria kvality návrhu: *propojení* a *koheze*. Obě dvě kritéria jsou míněna ve smyslu *nezávislosti*. Šlo jim tedy o to, aby při návrhu vznikly moduly, které jsou na sobě co nejméně závislé a to jak ve smyslu propojení, tak ve smyslu soudružnosti dat v modulu. Počet modulů je potřeba volit rozumně. Obrázek 2.1 ukazuje závislost ceny vývoje softwaru na počtu modulů, kdy s počtem modulů sice klesá cena za jeden vytvořený modulu, ale na druhou stranu roste cena za integrování těchto modulů do výsledného systému. Součtem těchto křivek dostaneme celkovou cenu vývoje, která je ve tvaru kolébky a je ideální volit takový počet modulů, který odpovídá dnu této kolébky (minimu funkce celkové ceny). Obě tyto vlastnosti jsou klíčové také pro modelem řízený návrh, ke kterému směřujeme. Dříve zmínění pánové také nabídli světu návrhový formalismus zvaný *Diagram Struktury,* který návrhářům umožnil grafickou vizualizaci navrhovaného kódu.



Obrázek 2.1 ukazuje závislost ceny vývoje SW (Cost or effort) na počtu modulů (Number of modules). Převzato z [6].

Zaměření na dřívější etapu životního cyklu SW pokračovalo dále a vyústilo ve *Strukturovanou Analýzu*, která byla popularizovaná lidmi Tom DeMarco (1978) a Edward Yourdon (1989). Principem bylo oddělení specifikace problému od jeho řešení, což vedlo k vytvoření dvou primárních modelů. První, zvaný *Základní Model*, sloužil jako reprezentace systému, bez zaměření na konkrétní implementaci. Druhý, zvaný *Implementační Model*, byl funkční realizací základního modelu s důrazem na organizaci hardwaru, softwaru a zdrojového kódu. Zde má základy metodika MDA se svým platformě nezávislým a platformě specifickým modelem, ale k tomu se dostaneme v kapitole 3. Při vytváření modelů v této etapě jsme se mohli setkat s diagramy typu *DFD* (Data Flow Diagram), *SD* (State diagram) a *ERD* (Entity Relationchip Diagram), které dnes v mírně upravené podobě potkáváme v modelovacím jazyce *UML* (Unified Modeling Language). V této době vznikl také *událostmi řízený* přístup, o který se zasloužili pánové Ward a Mellor (1985), kteří řekli, že systém nemusí být rozdělen s*hora-dolů*, ale *zvenku-dovnitř*. Jejich pohled byl založen na porozumění prostředí ve kterém systém běží a vytvoření specifikace chování tak aby vyhověla tomuto prostředí. Tento přístup se dá připodobnit k *Use-Case* diagramům, které opět najdeme v *UML*.

Mezitím, co byly vyvíjeny strukturované metody, si návrháři uvědomili, že mohou existovat i jiné způsoby rozdělení. Přišli na to, že objekty v systému jsou mnohem více stabilní, než funkce nad nimi prováděné. Vznikly tedy *Objektově Orientované* (OO) *Metody*, které se snaží přizpůsobit reálnému světu tím, že implementují objekty, které mají své vlastnosti a funkce, neboli metody, které mohou provádět. Knihy o OO programování autorů Booch (1986), Mayer (1988) a dalších byly následovány knihami o *OOD (Object-oriented Design)* a *OOA (Object-oriented Analysis)*. Tyto přístupy byly aplikovány hned v počátku životního cyklu SW, kde jsou prováděna nejdůležitější rozhodnutí ve formě specifikace chování a pravidel systému. MDA se ztotožňuje s touto aplikací a zavádí ji také do prvotní fáze vývoje.

Objektově orientovaný přístup se stal populárním a lidé začali vymýšlet, jak by šel vývoj za pomocí objektů ještě zjednodušit. Booch v publikaci *Object Oriented Analysis and Design with Applications* (1993) detailně popsal metodu OO vývoje. Principy užívání *vzorů*, nebo *archetypů*, jako základ pro platformě specifickou implementaci, byly obhajovány i mnoha dalšími (např.: Beck,

Shlaer, Mellor, Buschmann, Meunier, Coad). Celé to završila čtveřice Gamma, Helm, Johnson a Vlissides, kteří roku 1994 publikovali knihu *Design Patterns*, která je považována za nejznámější v oboru *návrhových vzorů*. Ivor Jacobson se svým přístupem řízeným *případy užití* (angl. Use-Case) roku 1992 pozvednul povědomí o důležitosti organizačních požadavků ve smyslu lehce modelovatelných a sledovatelných v průběhu životního cyklu. Roku 1991 vyvinul Dr. James Raumbaugh s kolektivem techniku zvanou *OMT (Object Management Technique)* ve které navrhli soubor notací pro podporu vytváření modelů analýzy a návrhu. Tato technika se v 90. letech stala dominantní.

Shlaer a Mellor s jejich *rekurzivním návrhem* zdůraznili význam vytváření precizních *PIM* (Platform Indipendent Model), které mohou být systematicky, případně automaticky, překládány až k vygenerování zdrojového kódu cílového systému. Dále prosadili nový přístup strukturování systému a to na základě tematických celků, čili sdružení tematicky shodných prvků do jednoho modulu zvaného *doména*. Výsledkem tohoto rozdělení systému jsou moduly vysoce nezávislé na ostatních, tak jak bylo požadováno již před 20 lety při *Strukturovaném Návrhu* pány Yourdon a Constantine. Podobným směrem se vydal kolektiv autorů v čele s Bran Selic, kteří v roce 1994 představili metodu nazývanou *Real-time Object-oriented Modeling (ROOM)* zaměřenou na vývoj RT systémů s důrazem na definování rozhraní a komunikačních protokolů. Oba tyto přístupy umožňují grafické znázornění, které je dnes standardizováno v UML.

Aktuální stav návrhu a vývoje systémů je považován za sjednocení těch nejlepších metod z historie a může být popsán následujícími vlastnostmi:

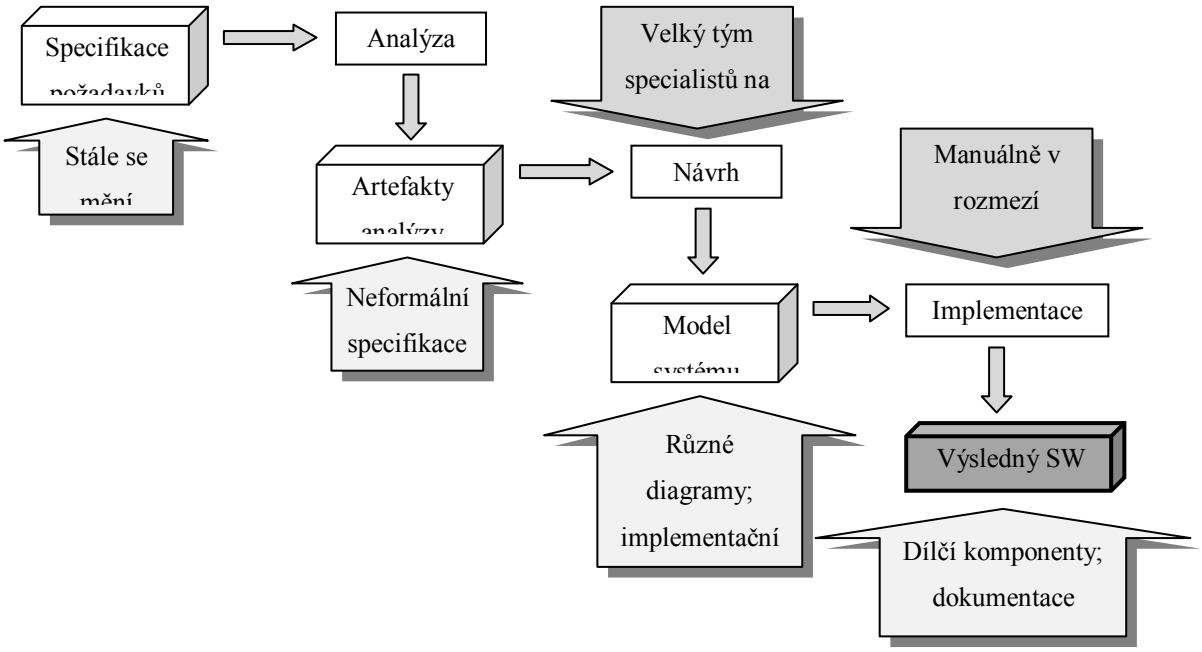
1. Rozdělení systému na domény, obsahující prvky/objekty zapadající do stejného tématu, které jsou téměř nepropojené a kohezní;
2. Oddělení platformě nezávislého chování od platformě specifického chování, jako rozšíření dříve definovaného oddělení *základního modelu* od *implementačního modelu*;
3. Definice vzorem řízeného mapování pro systematické vytváření *platformě specifických* modelů z *platformě nezávislých* modelů. Toto mapování může být manuální, ale je snaha o plnou automatizaci;
4. Používání abstraktního, ale úzce sémanticky definovaného formalismu *xUML* (executable UML), které zajišťuje modelům preciznost a možnost jejich testování/simulaci již v době modelování systému.

Tímto jsme prošli historií návrhu systémů a v následující kapitole se zaměříme na metodiku MDA.

3. Metodika Model Driven Architecture

MDA je metodika vývoje SW založená na proveditelných modelech. Je vyvíjena skupinou OMG [1], která má pod sebou i mnoho dalších metodik a technologií zaměřených na interoperabilitu a portabilitu objektově orientovaných aplikací, uveďme pár známějších např. UML, XML, XMI, COBRA.

Metodika MDA je tedy technika vývoje, kdy je striktně oddělen model systému ve smyslu požadovaného chování a funkcionality od implementačních detailů závislých na použitých technologiích. Model funkcionalit a chování systému, zvaný *PIM* (viz kap. 3.2), stačí vytvořit pouze jednou. Tento model je poté mapován pomocí překladových pravidel na *PSM* (viz kap. 3.3) a konkrétní implementaci na dané platformě, zvanou *PSI* (Platform Specific Implementation). Pro jeden *PIM* tedy může existovat mnoho *PSM*, které definují transformaci modelu *PIM* na platformě závislou implementaci *PSI*. Nutno říct, že princip MDA spočívá také právě ve formě modelu *PIM*, který je proveditelný a tedy testovatelný v průběhu vývoje.

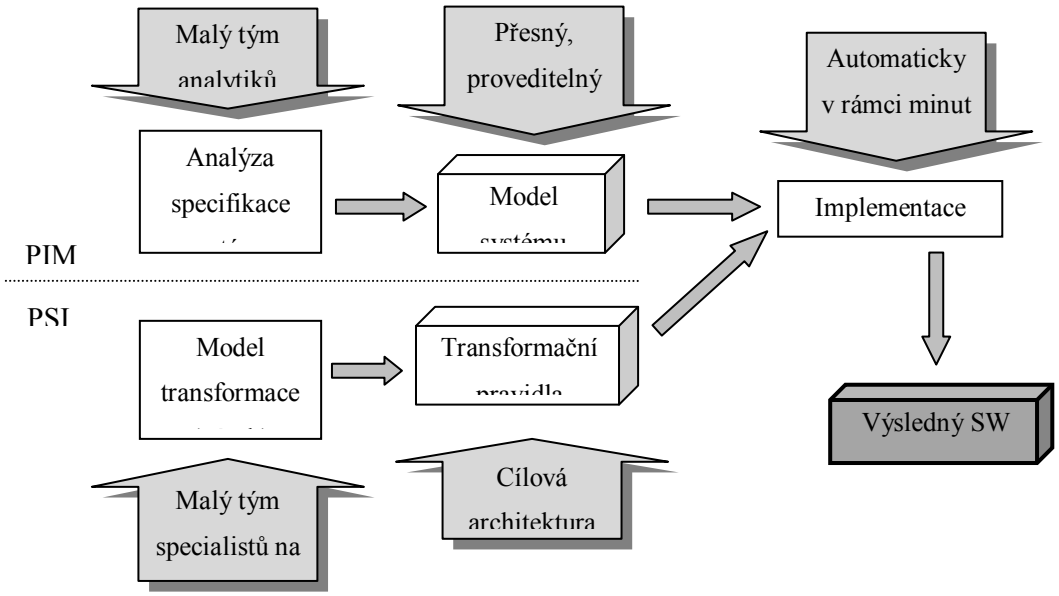


Obrázek 3.1 Diagram vývoje systému pomocí klasické vývojové metody Vodopád. Převzato z [5].

Klasické vývojové metody (např. Vodopád) stanoví problém (specifikace systému), který dále rozvíjí zaváděním návrhových a implementačních detailů, až dojdou k řešení, které představuje výsledek vývoje SW, čili konkrétní implementaci na dané platformě. Vývoj SW klasickou metodou Vodopád probíhá, jak znázorňuje Obrázek 3.1, postupně od specifikace požadavků (které se bohužel neustále mění), přes fázi analýzy, kdy jsou neformální požadavky na systém jasně a stručně přepsány do dokumentu, ze kterého návrhový tým vytvoří model systému, který specifikuje veškeré chování systému pomocí různých diagramů (např. diagram tříd, diagramy interakce, atp.). Tento model je

pouze ilustrativní a musí být manuálně programátory transformován do výsledného kódu systému. Je to rozvíjející proces (*Elaborative procces*).

Naopak v metodice MDA je stanoven problém ve formě formálního funkčního modelu *PIM*. Požadavky na systém jsou analyzovány a je z nich vytvořen funkční formální model, který se dá testovat. Současně s tím může být vytvořen (resp. zakoupen, nebo znovupoužit) model návrhu, přesněji model transformace funkčního modelu na konkrétní implementační kód, v závislosti na použité platformě. Tato transformace by měla být plně automatická, čímž zabere mnohem méně času, než manuální implementace u klasických metod. Tento překladem řízený proces (*Translation-based process*) je znázorňuje Obrázek 3.2. [5]



Obrázek 3.2 Diagram vývoje systému pomocí MDA. Znázorňuje oddělení funkčního modelu PIM od implementačně závislého transformačního modelu PSI. K jejich spojení dojde až při implementaci, respektive automatické transformaci do zdrojového kódu. Převzato z [5].

Dalším problémem klasických metod softwarového inženýrství je převod statického modelu systému (nejčastěji vytvořeného pomocí UML) na funkční implementaci SW. Tuto konverzi provádějí programátoři ve fázi implementace. V jejich práci jim mohou pomáhat různé (polo)automatizované transformace vybraných modelů, ze kterých lze získat kostru programu, či přímo kompletní kód nějakého modulu. Zde se objevuje jedno velké nebezpečí, které vzniká právě jednosměrností transformace [4]. Je velmi řídký jev, že programátor při dodatečných úpravách vyvíjeného systému zanese úpravy do všech artefaktů vývoje, čili analýzy, návrhu a samotné implementace. Tím vzniká nekonzistence vygenerovaného a upravovaného kódu vůči dříve navrhnutým modelům. U metodik řízených modelem se vždy upravuje model samotný, který je opět formálně verifikován a případně znovu generován do zdrojového kódu. [5]

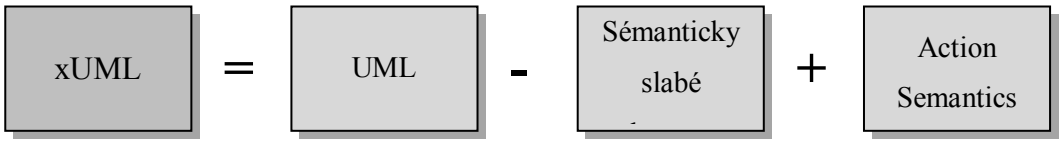
Koncept MDA využívá, rozšiřuje a integruje většinu již existujících a používaných specifikací skupiny OMG, jako třeba UML (Unified Modeling Language), MOF (Meta-Object Facility), XML (Extensible Markup Language) a IDL (Interface Definition Language). [7]

3.1 Srovnání UML a xUML

MDA se od klasických vývojových metod odlišuje také použitím modelovacích jazyků a modelu obecně. Zatímco u klasických metod je ve fázi analýzy a návrhu využíván model pouze coby reprezentativní a informativní formalismus, u MDA je tento model obdařen proveditelností, je tedy možné ho otestovat ihned po vytvoření. Tento přístup pomáhá odhalit chyby návrhu hned na počátku vývoje systému. [5]

UML (Unified Modeling Language) je modelovací jazyk, který poskytuje sjednocenou notaci pro reprezentaci různých aspektů objektově orientovaných systémů. Jazyk UML byl také specifikován skupinou OMG a jeho kompletní specifikace je dostupá zde [3]. Náplní této práce však není rozbor jazyka UML, přesto se na dalších řádcích se předpokládá alespoň základní znalost UML.

Jak už bylo řečeno, UML je pouze notační nástroj. UML samotné nemá přesně definovanou sémantiku značek, umožňuje použití různých značek v jiném významu v jiných diagramech (např.: stavový model může být použit k popsání Use-Case diagramů, diagramů objektu, nebo podsystémů) [3]. Tato vlastnost je pro MDA nežádoucí, modelování v MDA potřebuje striktní a precizní syntaxi a sémantiku notačních elementů, aby bylo u každého modelu přesně jasné co znamená a jaké chování reprezentuje. Toho bylo dosaženo vytvořením specifikace xUML (Executable UML), která je založena na jádře jazyka UML a doplněna Action Semantics (OMG, 2002), specifikující přesné chování používaných UML modelovacích elementů a odstraňuje mnoho nejasností elementů jazyka UML. Vzniká tak sjednocený modelovací jazyk, jehož pomocí lze vytvářet proveditelné modely. Vztah specifikace xUML s UML znázorňuje Obrázek 3.4. [5]



Obrázek 3.4 Vztah xUML a UML. Modelovací jazyk xUML čerpá z výhod klasického UML, ale odstraňuje diagramy a elementy, které nemají přesně danou sémantiku (chování), zato přidává a doplňuje přesnou specifikaci syntaxe a sémantiky zbylých elementů. Převzato z [5].

3.2 Platformě nezávislý model PIM

V MDA je pojem *platforma* používaná pro odkázání na technologické a návrhové detaily, které jsou irelevantní pro funkcionalitu SW. Platformě nezávislý model (PIM) je formální model popisující funkcionalitu systému, nebo jeho dané oblasti. PIM řeší koncepční otázky, ale nezohledňuje, jak bude vypadat technologická implementace. Pomocí dříve popsaného xUML tak můžeme vytvořit formální, precizní a proveditelný model PIM. Testování vlastností systému na úrovni PIM modelu přináší urychlení odladění případných chyb. K PIM modelu se často přidává PIM Metamodel do kterého mohou IT analytici vložit instrukce k překladu na PSM, aby výsledná implementace splňovala jejich výkonnostní požadavky. [5][7]

Základ PIM (jako modelu jedné domény) je diagram tříd, od kterého se odvíjejí další diagramy jako stavový diagram, nebo diagram aktivit [5]. V MDA je model PIM vytvářen pomocí xUML. Tento přístup ale obecně není jediný, jak uvidíme v kapitole 4.

3.3 Platformě specifický model PSM

Platformě specifický (závislý) model (PSM) reprezentuje veškerou funkčnost systému z PIM modelu s přidanou vazbou na konkrétní realizační platformu (např. COBRA, C++, JAVA). PSM je vytvořen z PIM aplikováním transformačních pravidel, tento proces se nazývá *mapování*. Mapování samotné je také definováno formálním modelem. Pro jeden PIM může existovat více PSM modelů. [5]

Existuje více přístupů generování PSM modelů. V prvním z nich, je PSM generován lidmi, týmem odborníků, který z PIM vytvoří PSM na základně cílové platformy. Tento přístup je však neefektivní. Přechodem k MDA jsme chtěli dosáhnout urychlení vývoje SW. Další přístup generování PSM je automatizované generování. Tento přístup již splňuje naše požadavky a je vhodný pro svou rychlost. Základem je, aby mapovací proces byl velmi dobře a precizně popsán/namodelován. Potom je možné úpravy v PIM modelu ihned přeložit a otestovat. Nedoporučuje se dělat zásah do vygenerovaného kódu, neboli PSI (Platform Specific Implementation), protože tím ztratí svou absolutní soudržnost s PIM modelem. [5]

3.4 Souhrn MDA

MDA je proces vývoje systémů, který je specifikovaný pod záštitou skupiny OMG, a ve světě používaný. Reprezentuje promyšlenou syntézu dobře otestovaných metodik softwarového vývoje. Proces vývoje systémů pomocí MDA se dá zjednodušeně popsat následujícími kroky:

1. Stanovit problém, získat požadavky na systém, identifikovat nebo znovupoužít funkční celky zvané *domény*
2. Vytvořit nebo znovupoužít precizní, prediktivní modely PIM odpovídající specifikací domén z předchozího kroku (diagramy tříd, interakce, chování elementů);
3. Podrobit modely PIM důkladnému testování a validaci před samotnou;
4. Vytvořit, znovupoužít nebo koupit modely PSM pro automatický převod (mapování) PIM modelů na konkrétní implementaci PSI;
5. Generovat výsledný systém z otestovaných a validovaných PIM modelů pomocí PSM modelu a provézt konečné testování na cílové platformě.

4. Modelování a simulace v procesu vývoje systémů

V předchozí kapitole jsme poukázali na existující uznávaný modelem řízený princip návrhu a vývoje SW zvaný *Model Driven Architecture*. V této kapitole se zaměříme na možnost zajít s teorií modelování a simulací ještě dále, než tomu je u MDA. Budeme vycházet z výzkumu a poznatků výzkumné skupiny Modelování a Simulace na fakultě FIT VUT v Brně [2]. Tato skupina, kromě jiného, zkoumá možnosti vývoje počítačových systémů přístupem kombinace modelem řízeného návrhu *MBD* (model-based design) s interaktivním inkrementálním vývojem (angl. exploratory programming), který spočívá v inkrementálním přechodu od modelu k realitě. Ve spojení se simulací daného modelu, případně jeho částí v průběhu vývoje, získáváme simulací řízený vývoj *SBD*, který umožňuje rychlé prototypování a simulování modelovaného systému, což je oproti klasické metodě obrovská výhoda. Formální model navíc umožňuje pokročilé funkční charakteristiky pomocí skládání spojitých a diskrétních stavebních bloků.

Jak již bylo naznačeno v kapitole 2, aktuální vývoj v oblasti softwarového inženýrství spočívá v posuvu od statických k dynamickým modelům a jejich simulaci, které umožňují efektivní analýzu procesů odehrávajících se ve vyvíjeném systému. Výše zmíněná výzkumná skupina se snaží dodržet tento trend s tendencí zachovat proveditelný model v průběhu celého vývoje až k cílovému nasazení. S tímto přístupem přišel již Ziegler v [8] a pojmenování tohoto přístupu se ustálilo na pojem *model continuity*. Zde si všimněme rozdílnost od metodologie MDA (popsané v kap. 3), která využívaný model systému nakonec transformuje do klasického zdrojového kódu (většinou OOP).

K udržení proveditelného modelu až po nasazení systému do reálného prostředí často pomohou techniky simulace *HIL (hardware-in-the-loop simulation)* a *SIL (software-in-the-loop simulation)*, které umožňují propojení reálných a simulovaných komponent v průběhu vývoje systému [4].

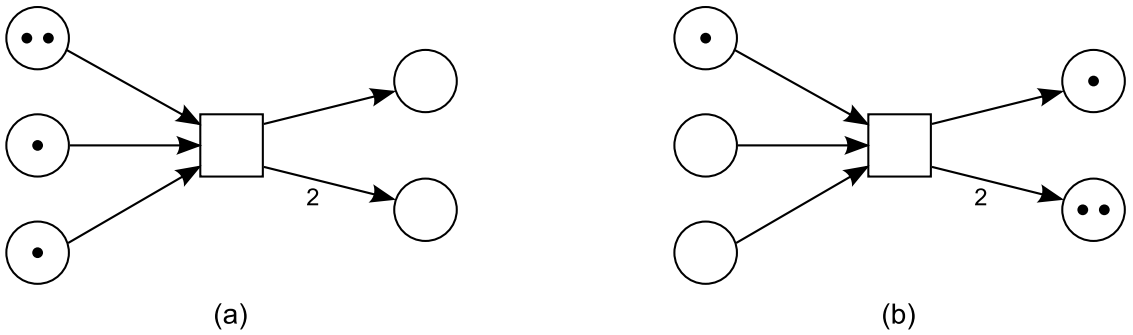
Vyvíjený model systému, který je v průběhu vývoje simulován, jak s virtuálními prvky, tak s reálnými, je potřeba formálně verifikovat. Výzkumná skupina Modelování a Simulace se proto snaží nahradit tradiční programovací jazyky formálními modely. Zaměřili se na formalismy jako *vysokoúrovňové objektově orientované Petriho sítě* (dále OOPN), *DEVS* a stavové diagramy (angl. state charts). Pro první dva formalismy implementovali nástroje, které kombinují programování a formální modelování, jde o *programovaní modely*, které umožňuje jednodušší práci s modely. [4]

Oba tyto formalismy a k nim vytvořené nástroje blíže proberu v následujících podkapitolách 4.2, 4.3 a 4.4. Začněme ale seznámením se základní strukturou a vlastnostmi jednoduchých Petriho sítí, ze kterých vycházejí rozšiřující typy Petriho sítí, včetně OOPN.

4.1 Petriho sítě

Petriho sítě představují vhodný formalismus pro modelování diskrétních systémů, nebo také workflow systémů, s možností grafického znázornění, analýzy a simulace. Populárnost Petriho sítí je dána jejich jednoduchostí. Model je sestaven z *míst*, které obsahují stavovou informaci ve formě

značek (tokenů) a mohou být kapacitně omezena (defaultně však omezená nejsou), *přechody* vyjadřující možné změny stavu a *hranami* propojující tyto *místa* a *přechody*. Hrany mohou být ohodnocené, kde ohodnocení hrany určuje kardinalitu přenášených značek. Provedení přechodu, též zvané *výskyt události*, u základních Petriho sítí je závislé na aktivaci přechodu. Přechod se stane aktivním, jakmile všechna vstupní místa, tzv. *preset*, obsahují potřebný počet značek, který je určen kardinalitou hran, a výstupní místa, tzv. *postset,* mohou pojmout tolik značek, kolik udává kardinalita odpovídající výstupní hrany přechodu. Ukázka provedení přechodu je vidět na následujícím Obrázek 4.1. [9]



Obrázek 4.1 Příklad přechodu a) => b) jednoduché Petriho sítě. Tečky představují značky, které jsou přechodem (čtverec) odebrány ze vstupních míst a generovány do míst výstupních. Přechod je aktivován pouze je-li ve všech vstupních místech potřebný počet značek a ve výstupních místech je prostor pro vygenerování nových značek. Převzato z [9].

Petriho sítě se dělí na různé typy, které vznikaly snahou zvýšit modelovací schopnosti tohoto formalismu při zachování koncepční jednoduchosti [9]. Různé typy tedy jsou: C-E (Condition-Evets) sítě, P/T sítě, dále vysokoúrovňové (High-Level) sítě, jako například predikátové (Predicate-Transition), nebo barvené (Coloured) sítě, které umožňují pohodlně modelovat i zpracování a tok dat [9].

4.2 Objektově orientované Petriho sítě

Protože Petriho sítě ve své původní podobě neposkytují strukturovací mechanismy známé z programovacích jazyků omezovalo se použití Petriho sítí pouze na nepříliš rozsáhlé systémy. Podíváme-li se na rozvoj programovacích jazyků a hlavně objektově orientovaného přístupu a postavíme jej vedle jednoduchých Petriho sítí, zjistíme, že tyto přístupy se zdají být ve svých výhodách a nevýhodách komplementární [9]. Není divu, že lidé začali vymýšlet jak tyto přístupy propojit a získat tak od každého výhody a odstranit nevýhody, čili získat dobře strukturovaný, paralelní a hlavně formální (de facto matematický) programovací nástroj. Jeden z možných způsobů zavedení objektové orientace do Petriho sítí je popsán v disertační práci [9], jejíž autorem je Vladimír Janoušek, který se ve své práci zabývá nejenom zavedením objektového paradigmatu do Petriho sítí, ale také vytvořením nástroje *PNtalk* pro práci s těmito OOPN. Spojením Petriho sítí s objektově orientovaným přístupem bylo dosaženo modelování aktivity objektu v objektu samotném, nikoliv

nadřazenou strukturou, jak je tomu u klasických OO programovacích jazycích jako třeba Java, nebo C++, které umožňují pouze kvaziparalelní[1] chování. Tímto se objekty stávají samostatnějšími a opravdu paralelními entitami, které spolu komunikují předáváním zpráv.

Dílčí konstrukce formalismu OOPN, jak jsou definovány v [9] a [10] a jak je budeme používat i my, si nyní popíšeme v následujících podkapitolách. Pro možnost pozdějšího využití výsledků práce v jiných rozšiřujících projektech, nebo studiích, budou zdrojové kódy modelu, uživatelské prostředí a diagramy psány v anglickém jazyce.

4.2.1 Objektová síť, atributy, stráž, metody

Zde si ukážeme jak budeme v OOPN modelovat princip objektové orientace. Objektové modelování pomocí Petriho sítí je umožněno zapouzdřením funkčnosti objektu do třídy. Specifikace třídy obsahuje definici objektu v podobě objektové *sítě*, která modeluje vnitřní aktivitu objektu a jeho vlastnosti, a definici *metod*, které jsou modelovány vlastní sítí, ale mají přístup k objektovým *atributům* a jeho dalším metodám. Objektová síť sestává z *míst* reprezentujících objektové *atributy* (privátní) a *přechodů* mezi těmito místy. Značky v místech OOPN již nejsou pouhé tečky, ale jsou to objekty. Přesto budeme používat notaci tečky (nebo symbolu *#e*) pro anonymní objekt, který slouží pouze k signalizaci změny stavu sítě. Místa mohou obsahovat počáteční značení určující obsah míst při inicializaci sítě.

Přechod kromě dříve zmíněných presetů a postsetů (viz kap. 4.1) může být na místa vázán tzv. *testovací hranou*, která naváže objekt z připojeného místa na proměnnou přechodu, ale objekt v místě nadále zůstává. Přechodům přibyla volitelná definice *stráže* a *akce*. Stráž přechodu přináší další podmínku proveditelnosti přechodu a lze s ní provádět testy na stav objektu a objektů navázaných na proměnné. Stráž se vyhodnocuje jako booleovský výraz, přičemž operandy mohou být kromě klasického vyhodnocování operací, také volání speciálních metod - *synchronního portu* nebo *predikátu*, které si popíšeme v podkapitole 4.2.2. V akci přechodu můžeme definovat sekvenci operací, jako třeba přiřazení proměnných, vytvoření nového objektu, nebo zasílání zpráv různým objektům. Rozhas platnosti jmen proměnných v akci přechodu zahrnuje stráž, akce a výrazy na okolních hranách. Zápis akce je od stráže oddělen vodorovnou čárou, pokud stráž není definována, pak se oddělující čára může vynechat. [9]

Hrany v OOPN neobsahují kardinalitu přenášených značek, ale tzv. hranové výrazy reprezentovány multimnožinami , které mají tvar

$$n_1{}'c_1, n_2{}'c_2, \dots, n_n{}'c_n$$

kde n_i koeficient násobnosti, předpokládá se nezáporné číslo, a c_i je term[2], nebo n-tice termů. Pokud je koeficient násobnosti roven 1, pak se nemusí zapisovat.

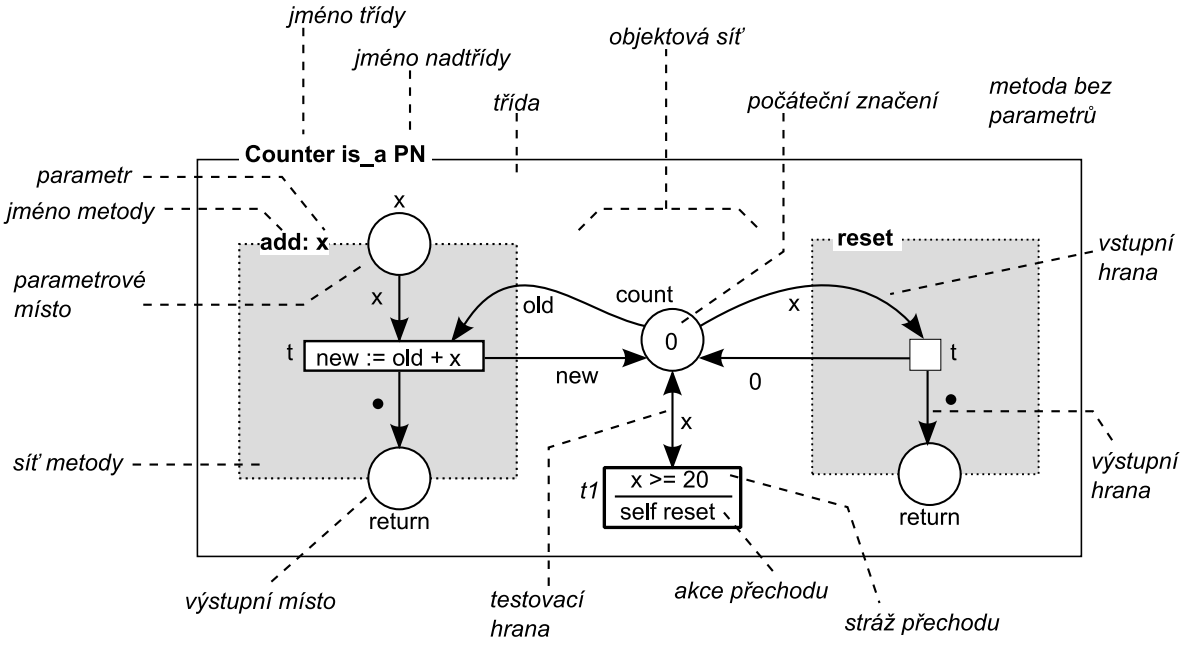
Ukázka specifikace jednoduché třídy OOPN je na obrázku Obrázek 4.2. Ukázka reprezentuje třídu `Counter` (počítadlo) s jedním atributem `count`, který při inicializaci obsahuje objekt *číslo* s

[1] Kvaziparalelní systém neumožňuje opravdový paralelní běh, ale simuluje paralelismus kontrolovaným přepínáním provádění jednotlivých procesů.

[2] Termy jsou nejjednodušší výrazy PNtalku. Reprezentují literály, proměnné, pseudoproměnné nebo jména tříd.

hodnotou 0, a metodami `reset` a `add:x`, kde `x` je parametrem metody a určuje kolik se má k počítadlu přičíst. Objektová síť obsahuje jeden přechod `t1` který je testovací hranou spojený s místem `count` a obsahuje stráž `x >= 0` a v akci probíhá volání své vlastní metody `reset`. Jakmile tedy bude atribut `count` obsahovat číslo větší než 19, aktivuje se přechod `t1` a následně bude provedena metoda `reset`, která z atributu `count` odebere aktuální číslo a vloží zpět číslo 0.

Když už známe základní konstrukce OOPN, řekněme si nyní o vytváření a komunikaci objektů. Instance objektu je kopie vzorové struktury popisující třídu, přesněji tedy objektové sítě. Komunikace objektů probíhá formou zasílání zpráv, která musí mít specifikovaného adresáta a předmět zprávy, kterým je nejčastěji volání metody nebo synchronního portu. Při zavolání metody objektu dojde k instanciaci sítě metody, na vstupní místa jsou navázány parametry metody a jako výstup slouží místo pojmenované *return*. K ukončení metody a zaniknutí sítě dojde v okamžiku, kdy se do místa *return* dostane jakákoliv značka, respektive objekt.

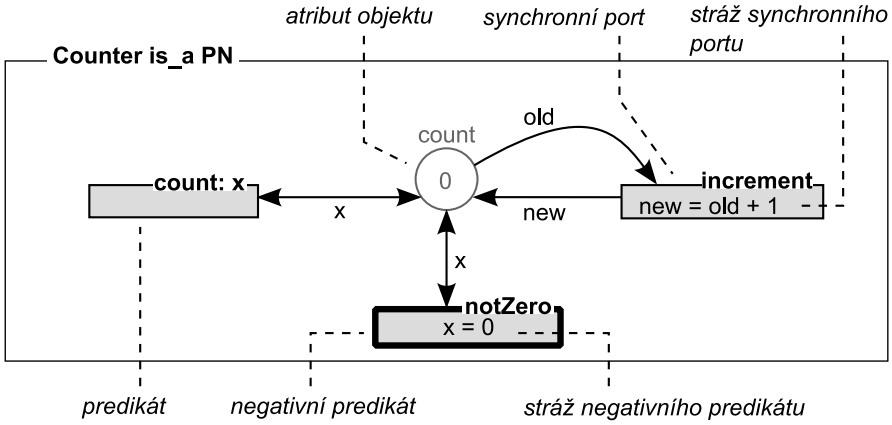


Obrázek 4.2 Ukázka grafické notace OOPN třídy *Counter* obsahující atribut *count* a metody *add(x)* a *reset*. Obrázek demonstruje jednotlivé prvky OOPN.

4.2.2 Synchronní port, predikát, negativní predikát

Na místa v objektové síti mohou být navázány *predikáty* a *negativní predikáty*, které mají testovací charakter, případně *synchronní porty* umožňující synchronní provedení přechodů více objektů najednou. Tyto prvky svým charakterem odpovídají přechodům i metodám, čemuž odpovídá jejich grafická notace (viz Obrázek 4.3). Porty, jak se jim zkráceně říká, mohou, podobně jako přechod, obsahovat vstupní, výstupní i testovací hrany a stráž, ale neobsahují akci, proto v grafické notaci pod stráží není oddělující čára. K synchronnímu portu je připojen vzor zprávy, na kterou reaguje a může být volán ze stráže libovolného přechodu. Tímto připomíná metodu, ale neobsahuje vlastní síť. Synchronní porty slouží k testování a změně stavu objektu. Mohou být volány s volnými

nebo navázanými proměnnými. Testování proveditelnosti portu, podobně jako přechodu, spočívá ve vhodném navázání proměnných. Výsledek volání synchronního portu je pravdivý, jestliže port je proveditelný v daném stavu objektu. Speciálním případem synchronního portu je *predikát*, který nikdy neovlivňuje stav objektu, pouze jej testuje pomocí testovacích hran. Predikát, který je platný právě pokud je neproveditelný, se nazývá *negativní predikát* [10]. Synchronní port je proveden společně s provedením přechodu v jehož stráži byl port testován. Platí tedy, že pokud je testovaný synchronní port proveditelný, ale testující přechod není proveditelný (např. kvůli vstupních podmínek), potom nedojde k provedení ani přechodu, ani portu. Provedení jakéhokoliv typu synchronního portu je vždy atomické. [9]



Obrázek 4.3 Ukázka grafické notace OOPN třídy *Counter* obsahující *synchronního portu* (increment), *predikátu* (count:x) a *negativního predikátu* (notZero). Obrázek demonstruje prvky OOPN.

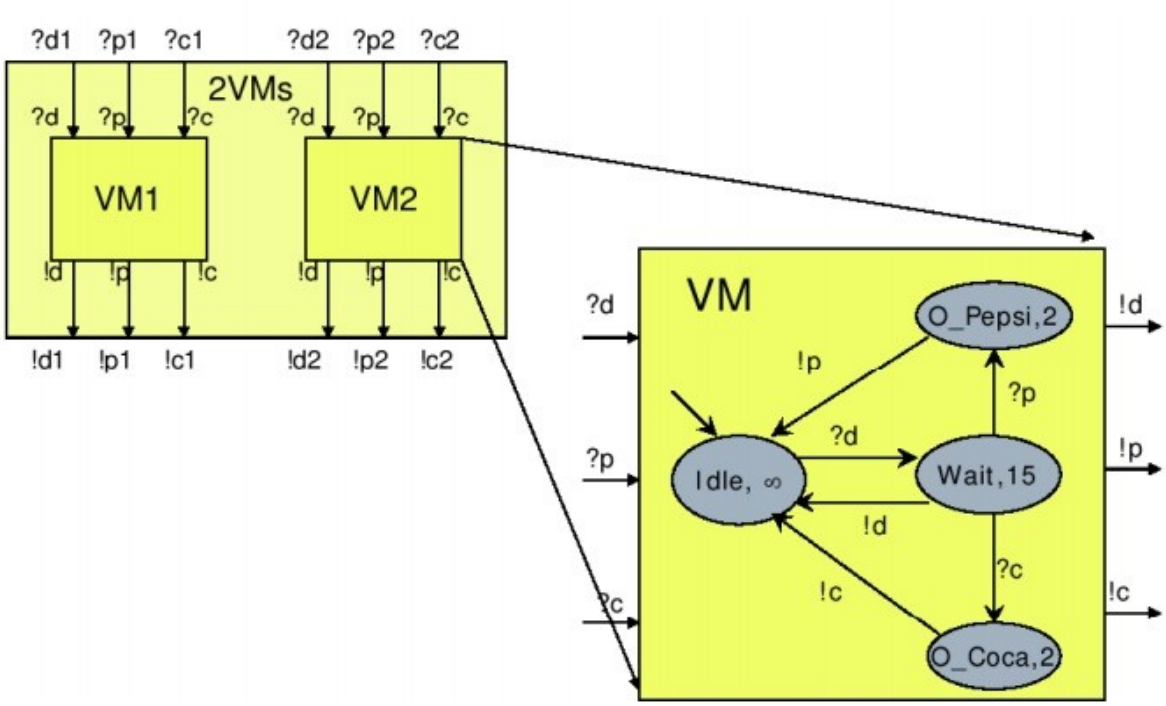
Grafickou notaci synchronního portu, predikátu a negativního predikátu si ukážeme na obrázku Obrázek 4.3, který zobrazuje rozšíření třídy `Counter` (počítadlo) z předchozí kapitoly 4.2.1. Pro přehlednost nebudeme zobrazovat všechny části třídy z obrázku Obrázek 4.2, pouze ty, na které navážeme naše nové prvky, a které zakreslíme šedou barvou. Zavedli jsme predikát `count:x`, který pokud zavoláme s volnou proměnnou, tak vrátí aktuální stav počítadla, a pokud jej zavoláme s navázanou proměnnou, tak vrátí `true` nebo `false`, podle toho, jestli je s tímto navázáním port proveditelný, čili jestli místo `count` obsahuje stejnou hodnotu, jako naše proměnná, nebo ne. Naopak negativní predikát `notZero` vrací pouze booleovskou hodnotu, která bude `true`, pokud bude negativní predikát neproveditelný, takže pokud navázaná proměnná `x` bude různá od 0. Uveďme zde, že zápis negativního predikátu `notZero` lze zjednodušit při zachování stejné funkčnosti tím, že odstraníme jeho stráž a hranový výraz testovací hrany přepíšeme na `1'0`. Potom zde ještě vidíme synchronní port `increment`, který je v dané situaci proveditelný vždy a mění stav počítadla jeho inkrementací. Všimněme si, že ve stráži portu `increment` není operace přiřazení, ale porovnání a tudíž dojde k navázání volné proměnné `new` na výsledek součtu `old + 1`.

V předchozí kapitole jsme se zmínili o tom, že atributy objektu jsou privátní, tedy nelze k nim přistupovat z venku. Pokud potřebujeme atribut udělat veřejným, aby k němu šlo přistoupit z venčí,

existují dvě možnosti jak toho docílit. První variantou je vytvoření dvou metod, tzv. *getter* a *setter*[3], pro přístup k atributu objektu. Výhodou tohoto přístupu je plná kontrola objektu nad svými atributy. Může měnit, kontrolovat, nebo dokonce zamezit, změně a získání atributu z venčí. Tyto metody budou znázorněny v kapitole 6.2. Druhou možností je navázání synchronního portu na místo reprezentující atribut objektu. Voláním portu s volnou proměnnou potom dojde k navázání obsahu místa na danou proměnnou. Pomocí synchronního portu se dá realizovat pouze operace *get*, nikoliv *set*. V praxi se často používají obě tyto varianty, které se navzájem doplňují.

4.3 DEVS

DEVS (Discrete EVent System specification) je formalismus pro specifikaci modelů založený na teorii systémů. DEVS formalismus rozlišuje atomické a složené komponenty (viz. Obrázek 4.4), kde složené komponenty mohou obsahovat další složené, nebo atomické komponenty. Atomické komponenty reprezentují jeden modul s rozhraním tvořeným vstupními a výstupními *porty*. Na DEVS komponenty mohou být mapovány i jiné formalismy (např. konečné automaty, Petriho sítě, atp.), vzniká tak hierarchicky definovaný systém popsaný různými formalismy, dle vhodnosti aplikace. [4]



[3] Getter a setter jsou pojmy obecně užívané v objektově orientovaném programování k pojmenování metod provádějící získání, respektive nastavení, určitého atributu objektu, který často bývá privátní. Tyto dvě metody podporují myšlenku zapouzdření objektu.

Obrázek 4.4 Příklad systému popsaného DEVS formalismem. Vlevo je spojovaný (hierarchický) model a vpravo je atomický model, který představuje jednu komponentu ve spojovaném modelu. Převzato z [11].

Systém (atomická komponenta také může být samostatný systém) modeluje vlastní aktivitu v podobě reakce na externí (vstupní) události externí přechodovou funkcí δ^{ext} a na interní (časově plánované) události interní přechodovou funkcí δ^{int}. Výstup systému (resp. komponenty) je závislý pouze na stavu systému (resp. komponenty). Formální popis atomického DEVS dle [11] je struktura

$$M = (X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, s_0)$$

kde

- X - množina vstupních hodnot (událostí)
- S - množina všech stavů systému
- Y - množina výstupních hodnot (událostí)
- $\delta_{int} : S \to S$ je interní přechodová funkce
- $\delta_{ext} : Q \times X \to S$ je externí přechodová funkce, kde $Q = \{(s, t_e) | s \in S, 0 \leq t_e \leq ta(s)\}$ je množina totálních stavů a t_e (elapsed time) je uplynulý čas od poslední události (vnitřní i vnější)
- $\lambda : S \to Y \cup \{e\}$ je interní výstupní funkce, kde e je prázdná událost
- $ta: S \to R_0^{+\infty}$ je funkce posuvu času (time advance function), kterou je při přechodu do nového stavu nastavena interní událost za daný čas
- $s_0 \in S$ - počáteční stav systému

Interpretace formalismu je poté následující: Systém se nachází ve stavu s. Pokud nenastane žádná externí událost $x \in X$, systém zůstává ve stavu s po dobu definovanou $ta(s)$. Při uplynutí času $t_e = ta(s)$ nastane vnitřní událost a provede se vnitřní přechodová funkce $\delta_{int}(s)$ a současně se vygeneruje výstupní hodnota pomocí výstupní funkce $\lambda(s)$. Pokud nastane externí událost $x \in X$ před uplynutím doby $ta(s)$, systém okamžitě přejde do stavu $s_{new} = \delta_{ext}(s, t_e, x)$. Při kolizi interní a externí události, čili v uplynulém čase $t_e = ta(s)$, má vždy přednost externí událost a interní událost se ignoruje. [11]

To byla definice atomického DEVS. Systémy jsou obvykle složeny z více subsystémů (komponent), které spolu interagují pomocí vstupních a výstupních portů, přičemž stav celého systému je dán stavem jednotlivých subsystémů. Při implementaci těchto složených systémů často množinu stavů a množiny vstupních a výstupních událostí obvykle specifikujeme jako strukturované množiny, což nám umožní používat libovolný konečný počet vstupních, výstupních portů a stavových proměnných. Strukturovaná množina obsahuje n-tice složek stavu nebo události, kde složka odpovídá dílčímu stavu (resp. události) v dílčí komponentě (resp. portu). Předcházející definice základního DEVS formalismu (atomické komponenty) je potom obohacena o množinu vstupních a výstupních portů. Spojovaný model je definován propojením submodelů (atomické nebo spojované modely) a vzniká nám tak možnost *hierarchického modelování*, z čehož plyne, že množina DEVS je uzavřena vůči skládání. Spojovaný model je dle [11] formálně definován strukturou

$$CM = (X, Y, D, \{M_i\}\, EIC, EOC, IC, select)$$

kde

- X a Y jsou množiny vstupních a výstupních hodnot (událostí),
- D je neprázdná konečná množina jmen komponent,

- $\{M_i\}$ je množina komponent, pro každé $i \in D$ je M_i atomický nebo spojovaný model,
- $EIC \subseteq X \times (\bigcup_{i \in D} M_i.X)$ množina připojení externího vstupu
- $EOC \subseteq (\bigcup_{i \in D} M_i.Y) \times Y$ množina připojení externího výstupu
- $IC \subseteq (\bigcup_{i \in D} M_i.Y) \times (\bigcup_{i \in D} M_i.X)$ množina interních propojení komponent
- $select: 2^D \rightarrow D$ je funkce výběru z konfliktních komponent (při výskytu události ve stejný čas)

Dle definice je propojení komponent (vstupů a výstupů) realizováno propojením událostí, avšak při použití portů jsou množiny strukturované a propojení je možné definovat mezi porty, kdy všechny události na port p_1 jsou mapovány na stejné události na portu p_2. [11] [12]

DEVS umožňuje simulaci modelu formou nadřazení simulátoru samotnému modelu v hierarchii komponent a propojením s jeho rozhraním. Lze tak simulovat celý model najednou, nebo pouze jeho části. Využívá se tzv. *Experimentální rámec*, který představuje zdrojový systém dat (reálné prostředí) a je napojen na vstupy a výstupy simulovaného modelu.

Při porovnání DEVS s OO přístupem lze dle [11] zjistit následující vlastnosti:

- OO zvládá dynamický vznik/zánik objektů, DEVS to řeší komplikovaně
- OO čitelně modeluje systém tříd, DEVS naopak pracuje na úrovní instancí
- OO komunikace objektů vyžaduje referenci na adresáta, DEVS definuje komunikační vazby jako relaci nad komponentami
- Některé komponenty systému mají dlouhou životnost, jiné nikoli. Komponenty s delší životností je vhodné modelovat komponentami DEVS, zbytek potom objekty uvnitř atomických komponent.

DEVS je založen na komponentním přístupu a objekty jsou zde používány na jemnější úrovní , čili v atomických komponentách. [11]

4.4 Nástroj PNtalk/SmallDEVS

V úvodu této kapitoly jsem zmínil výzkumnou skupinu Modelování a Simulace na FIT VUT v Brně [2]. Tato skupina se zabývá nejenom výzkumem v oblasti modelování a simulace, ale také vývojem nových formalismů (viz. OOPN v kapitole 4.1) a nástrojů pro práci s nimi. Pro výše uvedené formalismy to jsou nástroje *PNtalk* a *SmallDEVS*. Podobnost názvů těchto nástrojů s názvem programovacího jazyka *Smalltalk*, není náhodná, nýbrž opodstatněná, protože oba nástroje jsou založeny právě na tomto programovacím jazyku *Smalltalk*. Vývoj zmíněných nástrojů původně probíhal odděleně, nyní jsou však oba nástroje integrovány do implementace jazyka *Smalltalk* s názvem *Squeak* fungující jako virtuální operační systém. [4]

Jazyk *PNtalk* experimentální nástroj umožňující efektivní práci s OOPN a kromě aplikace v modelování a simulace se zaměřuje hlavně na možnost začlenění modelu do reálného prostředí. Takto pojatý model může sloužit jako část prototypu, případně přímo aplikace. Jazyk *PNtalk* umožňuje také grafickou práci s modelem OOPN.

Nyní si stručně popíšeme jeho syntaxi jazyka *PNtalk* a uvedeme názorný příklad definice třídy představené v kapitole 4.2. Následuje seznam klíčových slov jazyka a jejich popis:

- `is_a`, vyskytuje se v definici třídy a specifikuje rodičovskou třídu. Základní třídou OOPN je třída s názvem *PN*.
- `place`, definuje místo v síti. Umožňuje specifikovat počáteční značení.
- `sync`
- `inhibitor`
- `trans`, definuje přechod v síti. Udává jeho pojmenování a zavádí jeho možné komponenty, jimiž jsou hrany, stráž a akce, vypsány dále.
- `cond`, definuje testovací hranu přechodu. Udává místo na které je napojena a hranový výraz.
- `precond`, definuje vstupní hranu přechodu. Udává místo na které je napojena a hranový výraz.
- `postcond`, definuje výstupní hranu přechodu. Udává místo na které je napojena a hranový výraz.
- `guard`, definuje stráž přechodu.
- `action`, definuje akci přechodu.

Syntaxi jazyka znázorňuje následující kus kódu implementující třídu *Counter* z příkladu na obrázcích Obrázek 4.2 a Obrázek 4.3.

```
Counter is_a PN
    place count(1'0)
    trans t1
        cond count(1'x)
        guard { x >= 20 }
        action { self reset }
    sync count: x
        cond count(1'x)
    inhibitor notZero: x
        cond count(1'x)
        guard { x = 0 }
    sync increment
        precond count(1'old)
        guard { new := old + 1. }
        postcond count(1'new)
    method reset
        place return()
        trans t
            precond count(1'x)
            postcond count(1'0), return(1'#e)
    method add: x
        place x()
        place return()
        trans t
            precond count(1'old)
```

```
        action { new := old + 1. }
        postcond count(1'new), return(1'#e)
```

Struktury a principy vytváření objektů v OOPN popsaném v kapitole 4.2 umožňují zavést dynamické chování modelu. Dynamičností se rozumí změna struktury, nebo chování modelu za běhu. Tím že objekty (instance) jsou kopií vzorové třídy, je možné měnit buď chování objektu samotného, nebo změnit vzor třídy, přičemž každá další vytvořená instance této třídy bude kopírovat novou strukturu vzoru, ale instance již vytvořené zůstanou nezměněny. Toto platí nejenom pro objektovou síť, ale také pro definici sítí metod, jejichž kopie jsou vytvářeny při každém volání a zanikají při ukončení metody. [4]

V našem případě dynamičnost nevyužijeme, neboť provádíme demonstraci modelem řízeného vývoje na systému, jehož struktura zůstává statická.

Zatímco *PNtalk* je nástroj pro modelování OOPN, tak *SmallDEVS* je nástroj pro modelování a simulaci komplexních systému. *SmallDEVS* je dynamický, objektově orientovaný jazyk s otevřenou implementací založený na Zieglerově formalismu DEVS [8]. Modelování pomocí *SmallDEVS* je možné dvěma způsoby, stejně jako u *PNtalku*, kdy první je využití standardních nástrojů jazyka Smalltalk a knihovny tříd pro podporu DEVS formalismu (obsahuje třídy jako AtomicDEVS nebo CompositeDEVS) a druhý je využití specializovaného GUI[4] poskytovaného jazykem *SmallDEVS*, které je prozatím poněkud nestabilní, tudíž je tato možnost riskantnější. V obou případech je třeba ctít zásady DEVS (viz. 4.3).

Vývoj těchto experimentálních nástrojů je zaměřen na modelem a simulací řízený vývoj systémů, přičemž je kladen důraz na ponechání modelu a možnosti jeho simulace až do poslední chvíle, čili k aplikačnímu nasazení (*model continuity*). Z toho vyplývá další směr výzkumu a to optimalizace struktury modelů, transformace modelů do výkonnější podoby, případně možnost svázání s jiným programovacím jazykem. Vyvíjené nástroje jsou testovány na případových studiích jako třeba: konferenční systém řízený modelem toku dat (workflow), plánování v agentních systémech, řízení robotických systémů, nebo modelování a simulace senzorových sítí. [4]

[4] Zkratka z anglického *Graphical User Interface*, v češtině tedy Grafické uživatelské rozhraní.

5. Specifikace Konferenčního systému

Nyní se již dostáváme k aplikaci modelem řízeného návrhu na případové studii Konferenčního systému. V této kapitole si představíme požadovaný systém ve formě neformální specifikace, analýzy požadavků a diagramu případů užití. Nutno připomenout, že systém slouží pro demonstraci modelem řízeného návrhu, proto se nevyžaduje jeho komplexnost a přesnost zpracování.

5.1 Neformální specifikace

Konferenční systém by měl umožňovat vytvářet konference, jakožto události s daným datem a místem. Na takto vytvořenou konferenci poté autoři mohou přihlásit své vědecké články. Tyto články projdou hodnotícím procesem recenzentů a pokud bude vše v pořádku, bude článek redaktorem schválen k publikaci na konferenci. Po proběhnutí konference budou články zveřejněny online.

Každý uživatel si bude moci prohlédnout obsah konferencí a obsah těch článků, které již byly zveřejněny online. Může si také nechat vypsat všechny články najednou. Systém by měl zobrazovat konference a články v přehledné grafické podobě, např. v tabulce.

Systém by měl umožňovat správu uživatelů a přiřazení jednotlivých rolí, tj. redaktoři (správci), recenzenti, autoři, běžní uživatelé. Běžní uživatelé mají právo prohlížet konference a články dle výše popsaných kritérií. Autoři mohou navíc vytvářet své články a přihlašovat je k jednotlivým konferencím. Dále mohou tyto své články upravovat a mazat, dokud nejsou odeslány k hodnocení, nebo schváleny k publikaci. Autoři mohou pracovat pouze se svými články, nemohou například upravovat článek kolegy. Recenzenti mohou, navíc od běžných uživatelů, psát hodnotící recenze k článkům, ke kterým byli redaktorem přiřazeni. Tyto recenze jsou viditelné pouze aktérům přiřazeným k článku a redaktorovi. Redaktor (správce) pak může přidávat, upravovat a mazat konference, má na starost správu uživatelů a kontrolu nad akcemi těchto uživatelů. Má tedy právo přidávat, upravovat a mazat články, také upravovat a mazat recenze. Redaktor vybírá vhodné recenzenty k článku, vzhledem k jeho povaze. Dále může článek na konci cyklu schválit k publikaci, nebo jej zamítnout. Běžní uživatelé mají přístup do systému bez přihlašování. Autoři, recenzenti a redaktoři se pro využívání svých práv musí do systému přihlásit pomocí přihlašovacích údajů, které obdrží po vlastní registraci, nebo poté, co byli správcem vloženi do systému.

Uživatelské rozhraní bude implementováno jako webová aplikace, která bude komunikovat s modelem systému, od něhož obdrží potřebné informace. Systém by měl být intuitivní, vzhledově příjemný a uživatelsky přívětivý. Dále také dobře implementovaný a jednoduše udržovatelný.

5.2 Analýza požadavků

Z neformální specifikace plynou následující požadavky na systém:

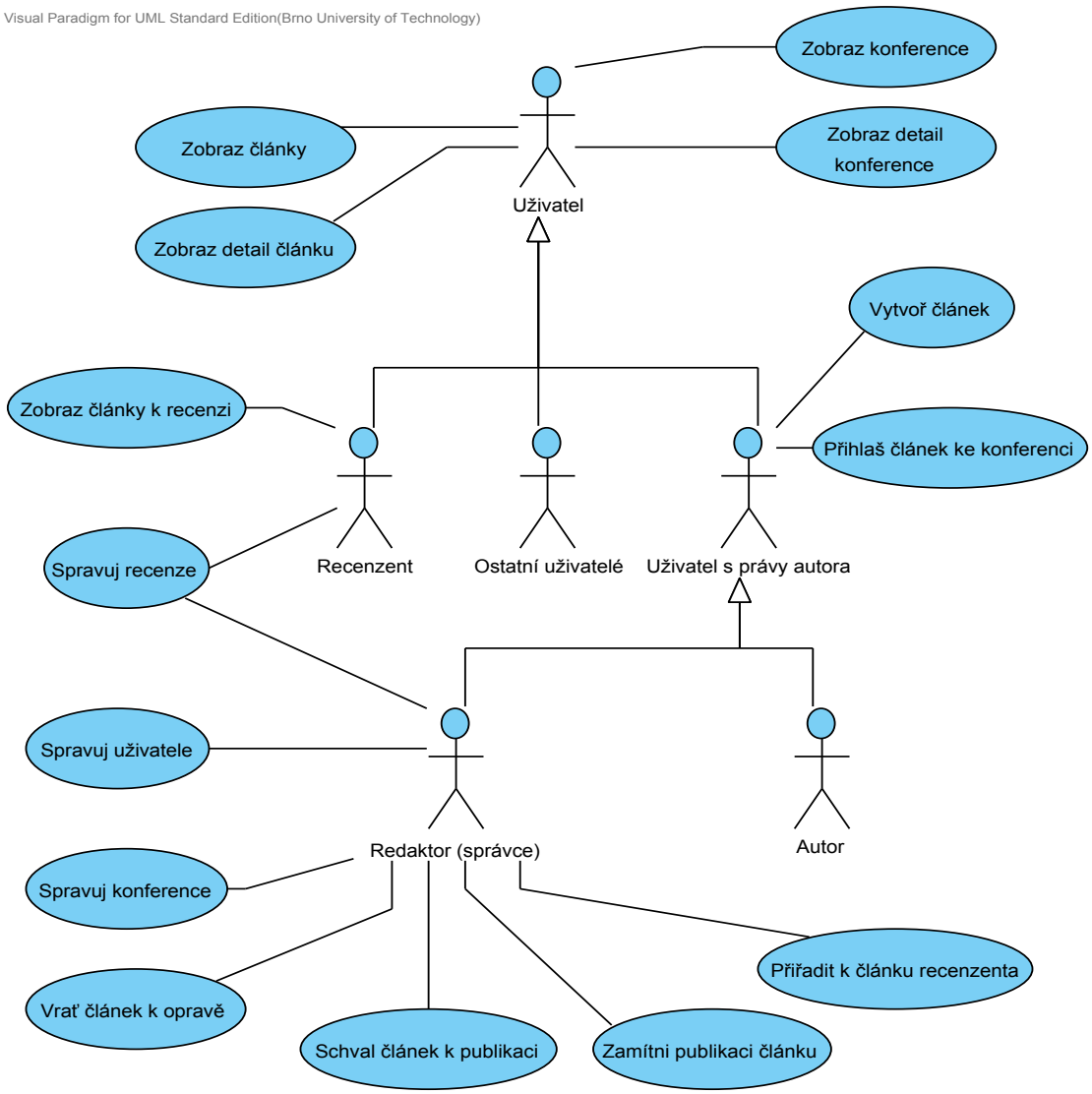
- abstraktní aktéři:

    - Uživatel - reprezentuje nejobecnějšího aktéra. Komunikuje s případy užití veřejně dostupnými, nemusí se tedy přihlašovat do systému. Jsou to pouze zobrazovací akce. Konkrétním potomkem je aktér z kategorie Ostatní uživatelé.
    - Uživatel s právy autora - reprezentuje aktéra, který má možnost spravovat články. Konkrétním potomkem je aktér Autor.
- speciální aktéři:
    - Recenzent - reprezentuje aktéra, který byl přiřazen k článku a požádán a napsání hodnotící recenze. Tento aktér nemá možnost články vytvářet, ale pouze zobrazovat a hodnotit.
    - Redaktor - je aktér s nejvyššími právy v systému. Může zobrazovat, upravovat a mazat libovolná data v systému.

Využití dědičností vyplynulo ze společných požadavků na aktéry v různých rolích. Následující diagram případu užití na obrázku Obrázek 5.1 zachycuje případy užití asociované s výše popsanými aktéry.

Diagram případů užití je diagram jazyka UML, pomocí něhož modelujeme rozvržení aktérů v systému a jejich možné akce nad systémem. Některé případy užití pokrývají několik samostatných případů užití, ale jejich zobrazením v diagramu by diagram ztratil na přehlednosti. Jedná se především o CRUD[5] akce.

[5] Pojem CRUD je zkratka čtyř anglických slov *Create*, *Read*, *Update* a *Delete*.



Obrázek 5.1 Diagram případů užití. Diagram znázorňuje rozvržení aktérů v systému a jejich možné akce.

6. Realizace modelu konferenčního systému

V této kapitole si ukážeme postup samotné realizace systému. Pro shrnutí si připomeňme, že systém bude modelován formalismem OOPN v nástroji PNtalk, který byl integrován do implementace jazyka Smalltalk, zvané *Squeak*. Implementaci celého systému můžeme rozdělit do dvou částí.

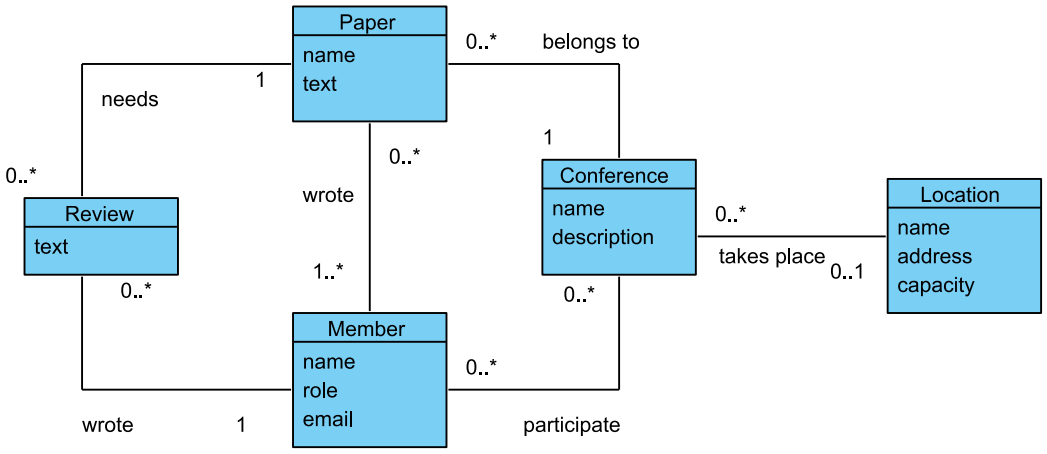
V první části, kterou popisuje tato kapitola, se zaměříme na vytvoření modelu systému představujícího entity vyskytující se v systému a jejich vazby. Nejprve vytvoříme konceptuální diagram tříd, který posléze bude sloužit jako předloha pro implementaci samotného modelu systému v OOPN. Vzhledem ke statičnosti modelu není jeho testování v této fázi příliš nutné. Přesto si v kapitole 8 ukážeme test reakce modelu na zasílané zprávy.

V další fázi provedeme nasazení modelu systému do reálné aplikace. Pro tyto účely zavedeme řídící sítě, které budou zprostředkovávat akce nad modelem systému a umožní tak nasazení do produkčního prostředí. Řídící sítě budou vykonávat netriviální operace a proto je potřeba provést jejich testování, které bude popsáno v kapitole 8. Poté vytvoříme jednoduché uživatelské rozhraní, popsané v kapitole 7.2.

Pro možnost pozdějšího využití výsledků práce v jiných rozšiřujících projektech, nebo studiích, budou zdrojové kódy modelu, uživatelské prostředí a diagramy psány v anglickém jazyce.

6.1 Návrh tříd

Pro návrh tříd využijeme diagram tříd modelovacího jazyka UML popisující typy objektů a statické vztahy mezi nimi. Vyskytuje se ve třech formách: konceptuální, specifikační, implementační. Jak již názvy napovídají, konceptuální diagram tříd slouží k vytvoření konceptu aplikační domény, bez vztahu k implementaci, je jazykově nezávislý. Specifikační diagram tříd znázorňuje pohled na aplikaci, specifikaci rozhraní bez specifikace implementace. Zato implementační diagram tříd zobrazuje i konkrétní implementační detaily. Pro naše účely se výborně hodí konceptuální diagram tříd, ve kterém si předvedeme strukturu modelu Konferenčního systému.



Obrázek 6.1 Konceptuální diagram tříd znázorňující vazby mezi objekty systému.

Obrázek 6.1 zobrazuje odpovídající konceptuální diagram tříd. Je zde vidět propojení jednotlivých tříd, respektive vazby jednotlivých objektů. Na obrázku můžeme vidět, že třída *Conference* (dále konference) kromě svého názvu a popisu bude obsahovat bude obsahovat maximálně jednu referenci na objekt třídy *Location* (dále místo), který svým jménem, adresou a kapacitou určuje místo, kde bude konference probíhat. Konference má také vazby na objekty tříd *Paper* (dále článek) a *Member* (dále účastník) s násobností 0..*, což znamená, že počet objektů, které budou náležet konferenci není nijak omezen. Naopak objekt třídy *Paper* patří vždy alespoň do jedné konference, má alespoň jednoho autora (instance třídy *Member*) a může obsahovat 0..* recenzí (instance třídy *Review*). Objekt třídy *Review* (dále recenze) tak vždy náleží jednomu článku a má jednoho autora. Nakonec účastník konference, jakožto objekt třídy *Member*, může a nemusí být přihlášen do více konferencí, může být autorem několika článků, nebo recenzí. V praxi to bude fungovat tak, že u účastníka záleží na jeho roli v systému, od které se odvíjí, jestli bude autorem článků, nebo recenzí.

Takto je znázorněn datový model systému. Nad tímto modelem později vytvoříme již dříve zmiňované řídící sítě, o kterých si povíme v kapitole 7.1.

6.2 Konverze diagramu tříd do objektově orientovaných Petriho sítí

Podle diagramu tříd z předchozí podkapitoly nyní vytvoříme definici tříd v OOPN. Požadované třídy a jejich atributy mají mnoho společných vlastností, jež představíme ještě před definicí samotných tříd na abstraktní třídě, kterou nazveme `Abstract`. Tyto vlastnosti potom budeme předpokládat u každé vytvářené třídy a budeme popisovat pouze odlišné a rozšiřující konstrukce. Může se stát, že následující definice nebudou kvůli jednoduchosti a čitelnosti přesně odpovídat implementovaným třídám, avšak bude zde zachyceno vše podstatné. Definice tříd budou, kromě slovního popisu, hojně znázorňovány diagramy, neboť grafická reprezentace OOPN je výstižnější a pro čtenáře přívětivější.

Připomeňme, že místa v sítích OOPN značí privátní atributy objektu, které lze pomocí synchronních portů, *getterů* a *setterů* zpřístupnit veřejně. Zaveďme tedy, že každý veřejný atribut níže představených tříd bude mít svůj getter a setter. Dále pro každý veřejný atribut `attr` bude specifikace třídy obsahovat stejnojmenný synchronní port, realizující operace get a test neprázdnosti, a negativní predikát pojmenovaný `attrEmpty`, realizující test prázdnosti atributu. Všechny tyto možnosti přístupu k jednoduchému atributu `attr` třídy `Abstract` demonstruje obrázek ***. Všimněme si, že porty `attr` a `attrEmpty` jsou komplementární a umožňují v metodách `getAttr` a `setAttr` výlučné provedení pouze jednoho z přechodů.

** TODO: ukázka predikátů attr, attrEmpty **
** TODO: ukázka getterů a setterů **

Pro atribut typu *kolekce*, potom místo getteru a setteru bude třída obsahovat metody pro přidání do kolekce, získání a odebrání z kolekce. Tyto metody se pro atribut `items` budou nazývat `addItem`, `getItem` a `removeItem`.

** TODO: ukázka add, get, remove pro kolekci dat **

Pokud třída obsahuje atribut typu kolekce, je pravděpodobné, že bude také potřebovat konstrukci pro provedení nějaké akce nad každým prvkem kolekce. Této konstrukci se v jiných objektově orientovaných programovacích jazycích říká cyklus *foreach* a je většinou zprostředkovaný jazykem samotným. PNtalk ale prozatím tuto konstrukci nezprostředkovává i přes to, že jazyk Smalltalk ano. Problém zřejmě tkví v tom, že v OOPN a PNtalku je kolekce chápána jako místo obsahující množinu objektů. Není tedy striktně určeno, které místo může obsahovat maximálně jeden objekt a které místo může obsahovat více objektů. Tato omezení závisí pouze na programátorovi, který si volí potřebný způsob přístupu k místu, tedy atributu objektu. Námi vytvořený cyklus *foreach* v potřebném rozsahu demonstruje obrázek ***.

** TODO: ukázka foreach zde, nebo až v kapitole testování, pro které jsem to tvořil **

Další teoretickou možností jak modelovat cyklus *foreach* je do místa reprezentujícího kolekci vložit Smalltalkovský objekt kolekce, nad kterým by byly prováděny operace add, get, remove a foreach. Toto řešení prozatím není možné, protože PNtalk neumožňuje v akcích přechodu použít bloky[6], které jsou nutné pro práci se Smalltalkovskou kolekcí.

[6] Blok je strukturální prvek jazyka Smalltalk (také mnoha jiných). Blok může obsahovat sekvenci příkazů, nebo další bloky, a používá se pro strukturní rozdělení zdrojového kódu. Jeho hranice jsou určeny hranatými závorkami - [..].
Pozn.: Například jazyk C++ také využívá bloky a jeho hranice jsou určeny otevírací, respektive uzavírací, složenou závorkou - { ... }.

Kromě výše zmíněných vlastností, definujme pro každou třídu metodu `print`, která vytiskne na standardní výstup[7] základní údaje o stavu objektu. Tato metoda bude využívaná pouze při testování a simulaci, pro prezentaci stavu objektu. O vypisování stavu objektu ve finální aplikaci se bude starat vrstva uživatelského rozhraní popsaná v kapitole 7.2.

Přestože je jazyk PNtalk (obecně Smalltalk) dynamicky typovaný[8], budeme níže u definice tříd specifikovat očekávaný typ proměnné daného atributu.

6.2.1 Conference

Třída *Conference* implementuje model konference tak, jak jej specifikace vyžaduje. Konference je základním prvkem konferenčního systému. Objektová síť této třídy nespecifikuje žádnou vlastní aktivitu objektu. Veřejnými atributy třídy jsou:

- `name`, obsahující *řetězec* se jménem konference;
- `description`, obsahující *řetězec* s popisem konference;
- `datetime`, obsahující *řetězec* udávající datum a čas konání;
- `location`, obsahující instanci třídy *Location;*
- `state`, obsahující *symbol* udávající stav konference;
- `papers`, obsahující *kolekci* článků (instancí třídy *Paper*) registrovaných na konferenci;
- `papersCount`, obsahující *číslo* udávající počet registrovaných článků v konferenci;
- `members`, obsahující *kolekci* účastníků konference (instancí třídy *Member*);
- `membersCount`, obsahující *číslo* udávající počet účastníku konference.

Třída *Conference* neobsahuje žádné nadstandardní metody nebo porty.

6.2.2 Location

Třída *Location* (dále místo) slouží k definici místa kde se bude konference odehrávat. Její atributy určují adresu místa, jeho kapacitu a pojmenování. Objektová síť třídy *Location* nespecifikuje žádnou vlastní aktivitu. Veřejnými atributy třídy jsou:

- `name`, obsahující *řetězec* udávající název místa;
- `capacity`, obsahující *číslo* udávající kapacitu místa;
- `address`, obsahující *řetězec* udávající adresu místa.

Třída *Location* neobsahuje žádné nadstandardní metody nebo porty.

6.2.3 Member

Třída *Member* (dále účastník, uživatel) slouží k definici osoby, jež se účastní dané konference ať už jako autor článků, nebo jejich recenzent. Její atributy udávají základní údaje o uživateli a jeho

[7] V prostředí *Squeak* je standardním výstupem myšlen modul *Transcript*.

[8] Dynamicky typované jazyky nepožadují specifikaci datového typu proměnných a ty mohou tudíž odkazovat na hodnoty jakéhokoli typu.

článcích a recenzích. Objektová síť třídy *Member* nespecifikuje žádnou vlastní aktivitu. Veřejnými atributy třídy jsou:

- `name`, obsahující *řetězec* udávající jméno uživatele, které je rovněž jeho přihlašovacím jménem do systému;
- `role`, obsahující *symbol* udávající roli uživatele v systému;
- `email`, obsahující *řetězec* udávající emailovou adresu uživatele;
- `papers`, obsahující *kolekci* článků (instancí třídy *Paper*) jichž je uživatel autorem;
- `papersCount`, obsahující *číslo* udávající počet napsaných článků.

Třída *Member* obsahuje jednu rozšiřující metodu `as:`*`role`* `name:`*`name`* a tři predikáty testující roli uživatele. Veškerá rozšíření jsou vidět na obrázku ***. Metoda `as:`*`role`* `name:`*`name`* provádí inicializaci objektu nastavením jeho role a jména. Můžeme říct, že je to pouze zkrácená verze dvou po sobě volaných setterů. Přídavné porty, resp. predikáty, testují roli uživatele pomocí symbolů reprezentujících jednotlivé role.

** TODO: graf. definice rozšíření **

6.2.4 Paper

Třída *Paper* (dále článek) slouží k definici článku registrovaného v systému. Její atributy udávají název a obsah článku, jeho autory, stav a recenzi. Objektová síť třídy *Paper* nespecifikuje žádnou vlastní aktivitu. Veřejnými atributy třídy jsou:

- `name`, obsahující *řetězec* udávající název článku;
- `text`, obsahující *řetězec* udávající obsah článku;
- `state`, obsahující *symbol* udávající stav článku;
- `authors`, obsahující *kolekci* uživatelů (instancí třídy *Member*) jež jsou autory článku;
- `authorsCount`, obsahující *číslo* udávající počet autorů;
- `review`, obsahující instanci třídy *Review*.

Třída *Paper* implementuje rozšířené chování metody `addAuthor:` *`author`*, ve které provádí volání zprávy `addPaper:` *`self`* nad instancí autora. Toto rozšíření mírně automatizuje vzájemné přiřazování autorů k článku a naopak, stačí článku přidat autora a obě přiřazení se provedou společně. Dále si všimněme, že atribut `review` není popsán jako kolekce, ale jednoduchý atribut. Tato vlastnost vyplývá ze specifikace systému a návrhu tříd.

** TODO: upravit podle konečné specifikace**

6.2.5 Review

Třída *Review* (dále recenze) slouží k definici recenze vytvořené k nějakému článku. Její atributy udávají obsah recenze a jejího autora. Objektová síť třídy *Review* nespecifikuje žádnou vlastní aktivitu. Veřejnými atributy třídy jsou:

- `text`, obsahující *řetězec* udávající obsah recenze;
- `author`, obsahující instancí třídy *Member*, reprezentující autora recenze.

Třída *Review* neobsahuje žádné nadstandardní metody nebo porty. Všimněme si atributu `author`, který reprezentuje jedinou instanci třídy *Member*, tak jak to vyplývá ze specifikace systému a návrhu tříd.

7. Nasazení modelu systému

Model vytvořený v předchozí kapitole nyní nasadíme do cílové aplikace. Forma aplikace byla určena zadáním této práce, jedná se o webovou aplikaci. Z možností nasazení vytvořeného modelu jsem zvolil použití modelu jako součásti aplikace (model continuity). Další možností byla transformace modelu do zdrojového kódu nějakého programovacího jazyka a následné využití kódu ve výsledné aplikace. Druhým zmíněným přístupem bychom se ale vraceli zpět ke klasickým metodám vývoje SW. Varianta přímého nasazené modelu byla zvolena díky tomu, že prostředí *Squeak* obsahuje volně dostupný framework pro tvorbu webových aplikací zvaný *Seaside*. Pomocí tohoto nástroje je možné z virtuálního prostředí *Squeak* jednoduše udělat webový server, který může hostovat více než jednu webovou aplikaci.

Architektura výsledné aplikace bude připomínat architekturu *Model View Controller* (zkr. MVC), která je v dnešní době webovými programátory hojně využívaná. Jedná se o architekturu aplikace rozdělenou do tří vrstev, jež mají zodpovědnost za jiné části aplikace. Spodní vrstva *Model* má zodpovědnost za práci s daty, jejich ukládání, vybírání a modifikování. Určuje také způsob rozložení dat a zapouzdřuje případné rozsáhlejší operace. Naopak horní vrstva *View* zajišťuje uživatelské rozhraní, včetně zobrazování dat a komunikace uživatele se systémem. Tyto dvě vrstvy jsou propojeny vrstvou *Controller*, která má na starosti vhodně zpracovávat požadované uživatelské akce, které přijímá od vrstvy *View* a naopak vyvolává odpovídající akce vrstvy *Model*. Architektura MVC tak odděluje ucelené části systému, které jsou potom jednodušeji udržovatelné a nahraditelné bez dopadu na zbytek systému. Jako příklad poslouží změna způsobu ukládání dat, potom stačí pozměnit implementaci vrstvy *Model*, která bude s daty pracovat novým způsobem, ale zachová si původní rozhraní, které využívá zbytek aplikace.

U malých a středních webových aplikací je úložiště dat nejčastěji realizováno *relační databází*. Při využití objektově orientovaného jazyka pro tvorbu webové aplikace potom vzniká problém ukládání zapouzdřených objektů do relační databáze, který se obvykle řeší zavedením ORM[9] (Object-relational mapping) nástrojů. V našem případě nebudeme využívat externího úložiště dat, protože nám prostředí *Squeak* umožňuje uchovávat data v sobě samém. Toto řešení přináší značné ulehčení, ale také poměrně nepříjemnou nevýhodu v podobě rizika ztráty dat při nečekaném pádu prostředí *Squeak*, ve kterém poběží jak model systému, tak webový server. Řešení dostupnosti dat napříč všech spuštěných instancí webové aplikace bude probráno v kapitole 7.2.

Vrstvu *Model* tedy bude zastávat model systému představený v kapitole 6 společně se řídícími OOPN sítěmi, o kterých si povíme v kapitole 7.1. Vrstvy *Controller* a *View* budou implementovány frameworkem Seaside a prezentovány v kapitole 7.2.

[9] ORM nástroje umožňují definovat mapování objektů aplikace do relačních databází a zpět. Jejich použití je téměř výhradně ve vrstvě Model

7.1 Řídící sítě

Nyní si popíšeme řídící sítě, které operují nad modelem systému z kapitoly 6 a zprostředkovávají tak akce modelu pro jakoukoliv zastřešující aplikaci (v našem případě webovou aplikaci). Nejdůležitější síť z této skupiny je nazvaná *SystemNet*, která, jak už název napovídá, zodpovídá za celý konferenční systém. Ostatní řídící sítě jsou vázány na roli uživatele a zprostředkovávají uživatelsky specifické akce (např. vložení článku autorem, vytvoření nové konference administrátorem). Pro představu jejich provázanosti mezi sebou a s původním modelem systému slouží diagram tříd na obrázku ***. Diagram je zjednodušený o detaily a vzájemné vazby tříd představených dříve v kapitole 6.1, ale přináší nový vztah - *Generalizaci*. Generalizace znázorňuje zobecnění skupiny tříd zavedením třídy obsahující společné vlastnosti podtříd. Generalizace modeluje dědičnost a v diagramu ji vidíme mezi třídami *AdminNet*, *AuthorNet*, *ReviewerNet* a *UserNet*. V diagramu také vidíme provázanost třídy *SystemNet* se všemi ostatními třídami modelu.

** TODO: Diagram řídící tříd **

7.1.1 SystemNet

Třídu *SystemNet* je nejdůležitější třídou celého Konferenčního systému. Tato třída specifikuje systém jako takový a její instance se bude na serveru vyskytovat jako singleton[10]. Veškeré spuštěné instance aplikace budou přistupovat k tomuto singletonu, jediné instanci třídy *SystemNet*, který budeme dále nazývat Systém. Singleton Systém slouží jako přístupový bod k modelu Konferenčního systému, uchovává všechna data systému a zajišťuje obsluhu základních operací jako například přihlášení/odhlášení uživatele do/ze systému, nebo tovární metody[11] pro vytváření nových objektů v systému. Jelikož jazyk PNtalk neumožňuje definici třídy jako singletonu, je tato skutečnost ošetřena v konstruktoru třídy webové aplikace (viz ***).

Třída *SystemNet* obsahuje následující veřejné atributy:

- `conferences`, obsahující *kolekci* všech instancí třídy *Conference* vyskytujících se v systému;
- ** pro ostatní entity systému **
- `users`, obsahující *kolekci* všech instancí třídy *Member* vyskytujících se v systému, reprezentující registrované uživatele;
- `loggedUserNets`, obsahující *kolekci* instancí třídy *Member*, reprezentující množinu přihlášených uživatelů.

Objektová síť třídy *SystemNet* má definovánu vlastní aktivitu, kterou obsluhuje registraci a přihlášení uživatelů. Adekvátní část specifikace třídy *SystemNet* je vyobrazena na obrázku *** a popsána v následujícím textu.

---

[10] Návrhový vzor *Singleton* (jedináček) zajišťuje, že aplikace bude obsahovat maximálně jednu instanci dané třídy.

[11] Tovární metody implementují návrhový vzor *Factory*.

Proces registrace, přihlášení a odhlášení vyžaduje metody:

- `newVerify:`*`name`* `as:`*`role`*, implementující registraci uživatele do systému, tzn. kontrolu unikátního jména, vytvoření nového uživatele systému a následné přihlášení;
- `verify:`*`user`*, provádějící přihlášení uživatele do systému, tedy vytvoření řídící sítě, její uložení do systému a podání její reference aplikaci;

predikáty:

- `user:`*`user`* `named:`*`name`*, testující stav systému na existenci registrovaného uživatele s určitým jménem;
- `login:`*`name`* `user:`*`user`*, testující, zda je v systému registrován uživatel s daným jménem a jeho navrácení přes volnou proměnou *`user`* (musí být volán s volnou proměnnou);

a synchronní port:

- `logout:`*`net`*, realizující test, jestli je uživatel s danou řídící sítí přihlášen, a jeho odhlášení.

Registrace je realizována voláním metody `newVerify`, přihlášení přes test na predikátu `login:user:` a následném volání metody `verify` a odhlášení přes synchronní port `logout`.

** TODO: obrázek výseku sítě **

Třída *SystemNet* dále zprostředkovává metody pro správu entit systému, jimiž jsou (znak * zastupuje název entity):

- `new`*, tovární metody pro vytváření nových entit systému;
- `add`*, metody pro přidávání entit do systému;
- `remove`*, metody pro odstraňování entit ze systému.

Všechny tři zmíněné metody jsou zobrazeny na obrázku ***, kde operují s instancí třídy *Conference*.

** TODO: obrázek ukázka tovární metody a add/remove metody **

7.1.2 UserNet

** Pravděpodobně zde zůstane pouze jedna uživatelská síť, která bude kontrolní bod pro všechny typy uživatelů. Myslím že to bude stačit. Rozdíly ve zobrazení budou řešeny v Seaside podle typu přihlášeného uživatele. Je zbytečné mít 3 třídy implementující skoro to stejné. **

** Mimochodem - zkoušel jsem v PNtalku vytvořit třídu dědící od již existující třídy ale nepovedlo se mi to. PNtalk tedy nepodporuje dědičnost mezi uživatelsky definovanými třídami? Nakonec nebudu potřebovat dědit, ale zajímalo by mě to. **

7.1.3 AdminNet

7.1.4 AuthorNet

7.1.5 ReviewerNet

7.2 Uživatelské rozhraní

Uživatelské rozhraní je podstatnou částí aplikace, která zajišťuje obousměrnou komunikaci s uživatelem. Pro jazyk Smalltalk existuje volně dostupný framework *Seaside* [13], který využijeme v naší aplikaci. Seaside je, stejně jako Smalltalk, čistě objektově orientovaný[12] framework, který byl navržen pro vývoj dynamických a komplexních webových aplikací. Obsahuje několik desítek předdefinovaných komponent, ze kterých lze stavět výslednou aplikaci.

Seaside je založen na komponentní architektuře využívající kontinuace, známé spíše z funkcionálních jazyků. Jedná se o přístup, kdy každá komponenta má své vlastní řízení a stav a umožňuje volání jiných komponent s tzv. *backtrackingem* (navracení se) zpět do místa volání. Struktura webové aplikace potom připomíná strom samostatných komponent řídících dílčí části aplikace. Tento přístup k tvorbě webových aplikací je mezi ostatními frameworky spíše neobvyklý.

Oproti jiným frameworkům, které využívají šablony, nebo organizaci pěkných URL adres, Seaside generuje obsah pomocí čistého Smalltalku a URL adresy jsou generovány dynamicky samotým Seaside, přičemž historie adres reflektuje předávání řízení mezi komponentami, na jejichž interakci je framework založen.

Životní cyklus každé komponenty sestává ze tří základních kroků:

- zobrazení dle svého stavu. Tuto akci obstarává metoda `renderContentOn:aCanvas`, pravděpodobně nejdůležitější metoda v Seaside. Metoda jako parametr přijímá objekt třídy *WACanvas*, který zastupuje vykreslovanou stránku[13] a zprostředkovává tvorbu obsahu stránky.
- Čekání na uživatelský vstup, kterým může být např. kliknutí na odkaz, potvrzení formuláře. Každá akce má definovanou reakci.
- Obsloužení provedené akce, kterým může být změna stavu systému, komponenty, nebo také volání jiné komponenty. Obsluha akce je v Seaside definována zprávou `callback`

[12] To znamená, že nabízí jen a pouze objektově orientovaný přístup. Dalšími čistými OO jazyky jsou například Ruby, JADE, Eiffel.

[13] Pojem stránka je zde myšlen ve smyslu webové stránky.

zaslanou aktivním prvkům stránky (např. odkaz, tlačítko). Zpráva `callback` přijímá jako parametr programový blok[14], který bude po provedení akce proveden.

Po dokončení obsloužení akce se komponenta dostane na začátek cyklu a znovu se vykreslí podle svého aktuálního stavu. Nejzajímavější akcí je právě volání jiné komponenty metodou `call:`, která provede vytvoření dané komponenty a předání řízení této komponentě. Nová komponenta opět provádí svůj cyklus dokud vlastní metodou `answer` nevrátí řízení původní komponentě. V procesu navrácení řízení může proběhnout také předání parametrů, např. výsledku běhu komponenty.

** mozna sekv. diagram? viz papir **

Kromě volání a předání celého řízení jiné komponentě umožňuje Seaside také vnořování komponent. Jedna komponenta potom může být poskládána z více komponent, které mají nezávislé řízení. Toho lze docílit metodou `render:` nad objektem třídy *WACanvas*, které dané komponentě vymezí prostor na stránce pro vykreslení a vlastní řízení. Stav vykreslené stránky je takto zachycen stavem všech dílčích komponent na stránce.

Výše popsané chování si nyní budeme demonstrovat na jednoduchém příkladu podobném *Hello World*[15]. Aplikace po nás bude vyžadovat jméno, pomocí něhož nás poté pozdraví. Představme si tedy definici kořenové třídy *WAHello*, která říká, že bude odvozena od třídy *WAComponent*, prozatím má pouze jednu instanční proměnnou `name`, se kterou budeme pracovat později. Poslední řádek určuje kategorii[16], ve které se třída nachází. Metoda `canBeRoot` přepisuje rodičovskou metodu a vrací hodnotu `true`, určuje tím že tato třída bude výchozím bodem aplikace a objeví se v nabídce web aplikací poskytovaných frameworkem Seaside.

```
WAComponent subclass: #WAHello
    instanceVariableNames: 'name'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PNtalk-DP-xcaham02'

canBeRoot
    ^ true.
```

Pro inicializační nastavení instance nám je poskytnuta metoda `initialize`, kterou ale prozatím nebudeme využívat. Pustíme se nyní do vytvoření obsahu stránky, tedy definice metody `renderContentOn:html` která je v následujícím úryvku kódu.

---

[14] Ve Smalltalku je programový blok objekt zvaný *BlockClosure*. Programový blok může obsahovat parametry, sekvenci příkazů, nebo další bloky.

[15] *Hello World* je mezi programátory označována typicky první aplikace, kterou programátor vytvoří v jazyce, nebo prostředí, se kterým se nově sezamuje.

[16] Kategorie ve Smalltalku jsoou pouze strukturálním rozdělením, nikoliv rozdělení do *Namespace (*jmenných prostorů).

```
WAHello >> renderContentOn: html
    html heading: 'SayHello app'.
    html form: [
        html text: 'Name'.
        html textInput
            on:#name of: self.
        html submitButton
            callback:[
                self call: (SayHello new to: name).
            ]; with: 'Say hello'.
    ]

WAHello >> name
    ^ name.

WAHello >> name: aName
    name _ aName.
```

Připomeňme si, že tato metoda přijímá jeden parametr jménem `html` reprezentující objekt třídy *WACanvas*. Obsah tvoříme zasíláním zpráv tomuto objektu, přičemž význam zpráv je často již ze zápisu pochopitelný, ale přesto si popíšeme alespoň ty zprávy, které budeme často využívat. Více informací najdete v knize o Seaside [13], nebo v dokumentaci na webových stránkách frameworku [14] v sekci dokumentace.

První použitou zprávou je zpráva `heading`, která přijímá řetězec a tvoří HTML element nadpisu - `h1`. Úroveň nadpisu lze specifikovat volitelným parametrem uvozeným klíčovým slovem `level:`. Další zprávou, přijímanou objektem `html` je zpráva `form` následována programovým blokem, ve kterém použijeme zprávu `text`, která svůj parametr vypíše do HTML jako text, a zprávy `textInput a submitButton`, popsané níže. Zpráva `form` vytvoří HTML formulář, přičemž v bloku, který přijímá jako parametr můžeme specifikovat další HTML prvky, hlavně bychom zde měli uvézt požadované formulářové pole a tlačítka.

V našem případě využijeme textové pole, kterému je navázána proměnná, kterou reprezentuje. Konstrukcí `html textInput on:#name of:self` tedy požadujeme od objektu `html` vytvoření textového vstupního pole, který bude obsluhovat atribut *name* objektu *self*. Tento atribut musí být v dané třídě specifikován a mít tam definovány své get/set metody, které také vidíme v představeném kódu výše. Při zobrazení našeho textového vstupu jej Seaside automaticky naplní hodnotou, kterou mu vrátí metoda `name`, a při potvrzení formuláře bude aktuální hodnota vstupu zpracována pomocí metody `name:`. Tento princip ulehčuje automatické zpracování formulářů, ale je také možné definovat si vlastní, typicky v *callbacku* potvrzovacího tlačítka. Dalším prvkem formuláře jsme zprávou `submitButton` definovali potvrzovací tlačítko formuláře, kterému jsme přiřadili obsluhu (programový blok) pomocí `callback` a popisek pomocí `with`. Obsluha tlačítka se provede kliknutím na něj. Dříve než si popíšeme co se vlastně stane, ukážeme si kód nové komponenty *SayHello*.

```
WAComponent subclass: #SayHello
    instanceVariableNames: 'name'
```

```
    classVariableNames: ''
    poolDictionaries: ''
    category: 'PNtalk-DP-xcaham02'

SayHello >> renderContentOn: html
    html heading: 'Hello ', name, '!'.
    html anchor callback: [ self answer. ]; with: 'Change name'.

SayHello >> to: aName
    name _ aName.
```

Co se tety stane? Vytvoří se nová komponenta třídy *SayHello*, které je hned zaslána zpráva `to:name`. Tato komponenta je pomocí `self call:` vyvolána do popředí a je jí předáno řízení aplikace. Znamená to, že se nám změní obsah stránky na takový, jak jej definuje metoda `renderContentOn:` komponenty *SayHello*. V této metodě vidíme opět definici nadpisu, tentokrát nejenom pouze s textem, ale také proměnnou, jejíž hodnota je konkatenována k textu. Hodnota proměnné `name` je nastavena metodou `to:` již při volání komponenty. Dále je zde pomocí zprávy `anchor` definován odkaz, který ve své obsluze provede kód `self answer` vedoucí k navrácení řízení původní komponentě *WAHello*. Posloupnost těchto akcí z pohledu uživatele znázorňuje obrázek ***. Všimněme si, že po návratu do úvodní komponenty zůstalo vstupní pole vyplněné, to proto, protože dříve při potvrzení formuláře se hodnota z pole uložila do proměnné komponenty a nyní je tedy načtena do vstupního pole, což dokazuje, že jsme se opravdu vrátili do původní komponenty, která nyní pokračuje ve svém cyklu a bylo znovu vykreslena.

** TODO: obr posloupností stránek **

7.2.1 Napojení na PNtalk

** počítám s tím, že v kap o PNtalk je popsáno vytvoření instance nějaké PNtalk třídy a její puštění v simulaci pomocí Workspace **

Nyní, když už máme představu o tom, jak funguje framework Seaside, popíšeme si jeho propojení s objekty vytvořenými v PNtalku, které budeme dále nazývat sítě. V kapitole 4.4 jsme si popsali instanciaci sítě a její spuštění v simulaci. Pokud chceme aby komponenta Seasidu mohla komunikovat s nějakou sítí, je třeba toto vytvoření a začlenění do simulace integrovat do její inicializační metody, přičemž si v nějaké proměnné ponecháme referenci na danou síť. V kapitole o řídících sítích 7.1 jsme zmínili, že základem celé aplikace z pohledu PNtalku bude síť *SystemNet*, která se bude v systému vyskytovat jako singleton, což musíme ošetřit při jejím vytváření v metodě `initialize` naší kořenové komponenty. Protože Seaside pro každé připojení k webové aplikaci vytváří novou instanci kořenové komponenty, je potřeba síť *SystemNet* uchovávat v třídní proměnné, kterou nazveme `system`, a zároveň kontrolovat jestli už nebylo instanciována. Teď trochu přeskočíme a prozradíme, že kořenová komponenta naší aplikace Konferenčního systému se bude jmenovat *WAConferenceSystemDP* a následující úryvek kódu je její metodou `initialize`.

```
WACoferenceSystemDP >> initialize
    | regCls |
    super initialize.
    system isNil ifTrue: [
        regCls _ MyRepository objectWithPathName: '/PNtalk
classes/ConferenceSystemDP/SystemNet'.
        system _ regCls newIn: (PNtalkSimulation default world).
    ].
```

Vytvořili jsme si proměnnou uchovávající síť *SystemNet*, která bude přístupná všem instancím naší webové aplikace. Po přistoupení k aplikaci přes webový prohlížeč uvidíme v prostředí Squeaku, že opravdu byla tato síť vytvořena a je připravena k simulaci. Zde přichází menší zádrhel, že tu simulaci musí manuálně spustit, aby se síť *SystemNet* stala aktivní a mohla reagovat na zaslané zprávy. Toto je bohužel další prozatimní vada experimentálního nástroje PNtalk.

Jelikož je síť PNtalku ve skutečnosti objekt, je možné ji zasílat zprávy a tím vyvolávat jeho metody. Toto se zatím od standardního použití Smalltalku. Sítě PNtalku ale mohou obsahovat synchronní porty a predikáty (viz kapitola 4.2.2), které se od klasických metod liší a je potřeba je volat speciálním způsobem, jak ukazuje příklad zdrojového kódu níže. Zasláním zprávy `asPort` objektu PNtalku získáme objekt (dále *port*), jehož metody odpovídají synchronním portům objektu PNtalku. Voláním metody objektu *port* se ve skutečnosti dotazujeme na stejnojmenný synchronní port původní sítě. Pokud má port nějaké parametry, můžeme jej volat, stejně jako při volání ze sítě PNtalku, s volnými, nebo vázanými proměnnými. Volání s vázanou proměnnou se neliší od jiného volání metod s parametrem, ale při volání s volnou proměnnou musíme na místo proměnné napsat `(PNVariable name:#var)`, kde `var` je jméno volné proměnné a bude na ni navázána hodnota, dle pravidel OOPN. Výsledkem volání portu je potom objekt, který nazveme *res*, a který umožňuje volání metod:

- `isTrue`, která vrací `true` pokud je synchronní port proveditelný;
- `perform`, která synchronní port provede[17];
- `variable:#var`, která vrátí výsledek navázání (objekt) na volnou proměnnou nazvanou `var`;
- `collectVariable:#var`, která vrátí kolekci všech možný navázání na volnou proměnnou `var`.

Pro názornost si ukážeme kus kódu, předpokládejme, že proměnná *`system`* je referencí na objekt OOPN třídy *SystemNet* představené v kapitole 7.1.1. V ukázce se vyskytuje také objekt OOPN třídy *Conference* popsané v kapitole 6.2.1.

V kódu provádíme výpis jmen všech konferencí obsažených v systému. Jelikož je synchronní port *conferences:* ve skutečnosti pouze predikátem, nevoláme nad výsledkem metodu `perform`.

```
renderContentOn: html
    system addConference: (system newConference name:'Conf1').
    system addConference: (system newConference name:'Conf2').
```

[17] Ačkoliv je v OOPN provedení portu automatické (viz 4.2.2), tak při přistupování ze Seaside komponenty, respektive z objektu Smalltalku, musí být provedení portu vyvoláno explicitě.

```
res _ system asPort conferences: (PNVariable name: #c).
res isTrue
    ifTrue: [
        (conferences collectVariable: #c) do:
            [:conf |
                res _ conf asPort name: (PNVariable name: #var).
                name _ res variable: #var.
                html text: name.
                html break.
            ].
    ]
    ifFalse: [
        html text: 'There is no conferences in system.'.
    ]
```

Úryvek kódu představuje metodu `renderContentOn` jejíž význam byl vysvětlen v úvodu této kapitoly 7.2. Nejprve se provede vložení dvou konferencí do systému pomocí volání odpovídajících metod třídy *SystemNet*. Tento kousek kódu by z hlediska dobrých programovacích praktik neměl být součástí vykreslovací metody, ale pro účel demonstrace jej zde necháme. Dále je proveden test portu `conferences` s volnou proměnnou `c`. Pokud je výsledek testu pravdivý, pak se přistoupí k větvi `ifTrue` (toto je podmiňovací konstrukce jazyka Smalltalk), ve které je pomocí metody `collectVariable` získána kolekce všech konferencí obsažených v systému nad kterou je proveden cyklus *do* (*foreach* smyčka Smalltalku), ve kterém je přes synchronní port *name* objektu konference získáno jméno konference, které je následně vypsáno na stránku a následováno zalomením řádku. Pokud by výsledek volání synchronního portu *conferences* byl nepravdivý, provedla by se větev `ifFalse` s výpisem informace o neexistenci žádné konference v systému.

** TODO: obrázek vrstev a naznačení role singletonu systém v aplikaci **

7.3 Vlastní provedení UI

** TODO: kapitola o mém provedení už. rozhraní?

8. Testování/simulace

8.1 Návrh testů

8.2 Výsledky testů

9. Závěr

** zatím stále stejný jako v SEP **

** napsat, že DEVS nebyl nakonec využit, sloužil by pouze ke strukturálnímu rozdělení **

V této práci jsem se zabýval *modelem řízeným návrhem* počítačových systémů. Stručně jsem prošel dosavadní historií vývoje softwarového inženýrství, a poté jsem se detailněji zaměřil na techniku modelem řízeného návrh (*MBD*), přesněji metodiku *MDA* (Model Driven Architecture) specifikovanou skupinou OMG (viz. [1]). MBD odstraňuje hlavní nevýhody klasických návrhových metod, které jsou nekonzistentnost návrhových modelů s výslednou implementací a nemožnost testovat a validovat systém ve fázi návrhu, tedy tvorby modelu systému. Návrh technikou *MBD* je založen na formálně popsaných funkčních modelech, díky kterým je možné systém prototypovat a testovat již ve fázi návrhu. Překlad na odpovídající implementační kód je oproti klasických metod, kdy implementační tým přepisuje model manuálně řádově několik měsíců, prováděn automaticky pomocí mapovacích pravidel odpovídajících dané cílové platformě, nebo je místo transformace funkční model zachován v aplikaci i při reálném nasazení (*model continuity*). Konkrétně u *MDA* je návrhový model popsán jazykem *xUML*, který je vytvořen z notačního modelovacího jazyka *UML*, přičemž odstraňuje sémanticky slabé elementy a zbylým přidává striktně specifikovanou syntaxi a sémantiku.

V další kapitole jsem probral obecný význam využití modelování a simulací ve vývoji systémů a poukázal jsem na výzkum probíhající na fakultě informačních technologií na VUT v Brně (viz. stránky výzkumné skupiny [2]), který je zaměřen na využití formalismů jako *DEVS* a *OOPN* (Objektově orientované Petriho sítě), vývoj nástrojů pro práci s těmito formalismy a aplikaci dříve zmiňovaného *model continuity*. Podrobněji jsem se seznámil s formalismem *DEVS*, který umožňuje hierarchické skládání podsystémů (komponent) v jeden komplexní událostně řízený diskrétní systém. Komponenty zde mohou být opět spojované, nebo atomické, kde se mimo jiných formálních popisu chování, jako například konečné automaty, uplatní hlavně formalismus *OOPN*, který spojuje výhody objektového programování a formálního matematického aparátu Petriho sítí. Teoreticky jsem se seznámil s vyvíjenými nástroji *SmallDEVS/PNtalk*, které umožňují programování modelu pomocí zmíněných formalismů, s důrazem na dynamičnost systému a možnost reálného nasazení modelu, respektive propojení simulovaných a reálných subsystémů, a kombinaci s jinými formalismy.

Doposud jsem provedl teoretickou přípravu pro modelem řízený návrh konferenčního systému, který je hlavním cílem mé diplomové práce. Pomocí uvedených formalismů a nástrojů budu vytvářet konferenční systém v základní funkčnosti. Účelem mé diplomové práce je na statickém, deterministickém informačním systému demonstrovat techniky vývoje zkoumané výzkumnou skupinou Modelování a Simulace při fakultě FIT VUT v Brně. Tento proces vývoje poté zhodnotím a porovnám s klasickou metodou vývoje informačních systémů.

Bibliografie

[1] **Object Management Group, Inc.** MDA. *Object Management Group.* [Online] 30. 10 2012. [Citace: 10. 12 2012.] http://www.omg.org/mda/.

[2] **Kočí, R., a další.** Modelování a simulace. *Modelování a simulace.* [Online] [Citace: 11. 12 2012.] http://perchta.fit.vutbr.cz/research-modeling-and-simulation.

[3] **Object Management Group, Inc.** UML Resource Page. *Object Management Group - UML.* [Online] 28. 11 2012. [Citace: 10. 12 2012.] http://www.uml.org/.

[4] **Kočí, R. a Janoušek, V.** Simulace a vývoj systémů. *Modelování a simulace.* [Online] 17. 9 2010. [Citace: 11. 12 2012.] http://perchta.fit.vutbr.cz/research-modeling-and-simulation/3.

[5] **Raistrick, Ch., a další.** *Model Driven Architecture with Executable UML.* London : Cambridge University Press, 2004. ISBN 0 521 53771 1.

[6] **Zendulka, Jaroslav.** Projektování programových systémů. [Online] [Citace: 10. 12 2012.] http://www.fit.vutbr.cz/study/courses/PPS/public/pdf/11_2.pdf.

[7] **Patočka, Miroslav.** *Řešení IS pomocí Model Driven Architecture.* [pdf] Brno : Masarykova Univerzita, 2008. Diplomová práce.

[8] **Ziegler, B.P., Preahofer, H. a Kim, T.G.** *Theory of Modeling and Simulation: Integrating discrete event and continuous complex dynamic systems.* Vyd. 2. San Diego : Academic Press, 2000. ISBN 01-277-8455-1.

[9] **Janoušek, V.** Modelování objektů Petriho sítěmi. *Disertační práce.* Brno : Fakulta informačních technologií, Vysoké učení technické v Brně, 1998. ISBN 978-80-214-3747-4.

[10] **Kočí, R. a V., Janoušek.** Enhancing the PNtalk Language with Negative Predicates. *MOSIS '08.* Ostrava, CZ : MARQ, 2008. stránky 28-34. ISBN 978-80-86840-40-6.

[11] **Janoušek, V.** Systémy s diskrétními událostmi, formalismus DEVS. *Prezentace k přednáškám.* [pdf].

[12] **Peringer, P.** Simulační nástroje a techniky. *Prezentace k přednáškám.* [pdf]. Brno : Fakulta informačních techologií, Vysoké učení technické v Brně, 2011.

[13] **Ducasse, S., a další.** *Dynamic Web development with Seaside.* Switzerland : Square Bracket Associates, 2010.

[14] seaside.st: Home. *Seaside.* [Online] [Citace: 10. 4 2013.] http://www.seaside.st/.

[15] **Girault, C. a Valk, R.** *Petri Nets for Systems Engineering. A Guide to Modeling, Verification, and Applications.* Berlin : Springer, 2003. ISBN 3-560-41217-4.

[16] Objektová Petriho síť. *akela.mendelu.cz.* [Online] [Citace: 11. 12 2012.] https://akela.mendelu.cz/~petrj/opnml/opn.html.