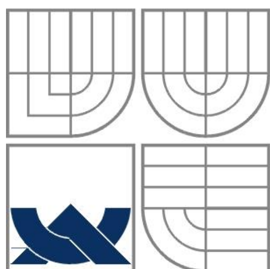
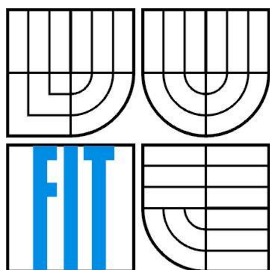


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

MODELEM ŘÍZENÝ NÁVRH SOFTWAROVÝCH SYSTÉMŮ

MODEL-DRIVEN SOFTWARE DEVELOPMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE — TOMÁŠ HANÁK
AUTHOR

VEDOUCÍ PRÁCE — ING. RADEK KOČÍ, PH.D.
SUPERVISOR

BRNO 2016

Abstrakt

Tato práce se zabývá moderním přístupem při vytváření softwarových systémů, který se nazývá modelem řízený návrh softwarových systémů. Jedná se o metodiku, která odstraňuje řadu kritických problémů, jež se vyskytují při konvenčním návrhu systémů. Hlavní myšlenka tohoto poměrně nového přístupu se zakládá na možnosti automatického, či poloautomatického generování výsledného kódu z konceptuálních modelů.

Abstract

This bachelor thesis deals with modern approach in creating software systems that is named Model-Driven Software Development. This metodology eliminates a lot of critical problems that can appear in conventional metodologies of creating software systems. Main idea of this approach is based on posibility to generate final code from conceptual models that can be done automatically or semi-automatically.

Klíčová slova

objektově orientované Petriho Sítě, DEVS, PNtalk, softwarové inženýrství

Keywords

object oriented Petri Nets, DEVS, PNtalk, software engineering

Citace

HANÁK, Tomáš. *Modelem řízený návrh softwarových systémů*. Brno, 2016. 32 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kočí Radek.

Modelem řízený návrh softwarových systémů

Prohlášení

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

……………………
Tomáš Hanák
16. 5. 2016

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Radku Kočímu, Ph.D., za odborné vedení, vstřícný přístup a cenné rady, které mi při psaní práce poskytl.

© Tomáš Hanák, 2016
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 3
1.1 Obsah bakalářské práce 3
1.2 Teoretické předpoklady a souvislosti 4
2 Model-driven engineering 5
2.1 Model Driven Architecture 6
2.1.1 xUML 6
2.2 Model-Driven Software Developement 7
2.2.1 Objektově orientované Petriho sítě 7
2.2.2 DEVS 7
3 Modeling and Simulation-Based Design 8
3.1 Využití OOPN 8
3.2 Modelovací technika 9
3.3 PNtalk 11
4 Případová studie 12
4.1 Případ užití (Use Case) 12
4.2 Aktivní subjekty 13
4.3 Specifikace rolí 15
4.4 Systémová role – společná síť 16
4.5 Sítě činností 17
4.5.1 Případ užití – Login 18
4.5.2 Případ užití – Editace rezervací 19
4.6 Aplikační síť 20
5 Aplikace 21
5.1 Smalltalk 21
5.2 Rezervační systém ve Smalltalku 21
5.2.1 Návrh aplikace 21
5.2.2 Data v aplikaci 22
5.2.3 Struktura a grafické uživatelské rozhraní 22
5.3 Rezervační systém řízený simulací OOPN 23
5.3.1 Aplikační síť 23
5.3.2 Inicializace aplikace 25
5.3.3 Volání GUI z OOPN 25
5.4 Možné změny 25

6 Shrnutí výhod a nevýhod metodiky 26
6.1 Efektivita..26
6.2 Vývojové prostředí 27
6.3 Změna vývojového cyklu 27
7 Závěr 28

1 Úvod

Softwarové inženýrství nabírá za posledních dvacet let na své důležitosti a vážnosti, neboť projekty jsou stále rozsáhlejší a komplikovanější. Klade se neustále větší důraz na kvalitnější technická řešení, která by navíc měla být v ideálním případě podložena precizní dokumentací, ovšem stále za předpokladu splnění dvou hlavních cílů softwarového inženýrství, a to snahy o dosažení co možná nejmenší ceny softwaru a zároveň co největší spolehlivosti. V neposlední řadě je kladen velký důraz na schopnost flexibilně reagovat na měnící se požadavky, například ze strany zákazníka, a také na čas, který je potřeba na zhotovení výsledného softwaru. Tyto dva požadavky značně ovlivňují vývoj softwarových systémů.

V současnosti se softwarové inženýrství rozvíjí dvěma směry. Prvním směrem je rozvoj agilních metodik, mezi které patří například metodika extrémního programování nebo metodika Scrum. Agilní metodiky jsou založeny na principu iterativního a inkrementálního vývoje, díky kterému dosahují flexibility při změně požadavků a zároveň jsou schopny rychlého vývoje softwaru. Druhým významným směrem jsou pak právě metodiky návrhu softwaru založených na modelech. Mezi tyto metodiky řadíme například metodiku Model Driven Architecture (MDA) nebo Model-Driven Software Developement (MDSD). Hlavní myšlenkou těchto metodik je možnost definovat systém pomocí modelů, ze kterých je možnost generovat výsledný kód automaticky, či poloautomaticky. Na rozdíl od konvenčních přístupů při vývoji softwarových systémů je výhoda těchto přístupů nejen ve flexibilitě při změně požadavků, ale navíc je výsledný kód izomorfní k modelům, neboť je generován přímo z modelů.

1.1 Obsah bakalářské práce

Práce se zabývá alternativními metodikami návrhu softwarových systému, které jsou založeny na modelech. Práce je strukturovaná následujícím způsobem.

Kapitola dvě se zaměřuje na dostupné metodiky, které v tomto odvětví doposud vznikly, a to především metodika MDA, která využívá striktně jazyk Executable UML (xUML), a metodika MDSD, jež může využívat jakýkoliv meta-modelový přístup, např. Objektově orientované Petriho sítě. Ve třetí kapitole si představíme techniku softwarového vývoje zvanou Modeling and Simulation-Based Design (MSBD). Čtvrtá kapitola pak představuje případovou studii druhé metodiky MDSD za pomocí Objektově orientovaných Petriho sítí (OOPN) a Discrete Event System Specification (DEVS). Vytvoření grafické aplikace pomocí jazyka Smalltalk a následná demonstrace propojení OOPN s grafickým uživatelským rozhraním jsou popsány v kapitole pět. Poslední kapitola se zaměřuje na použitelnost tohoto přístupu v praxi a jeho výhody a nevýhody.

Cílem bakalářské práce je tedy zhodnotit dosavadní přístupy v modelem řízeném softwarovém inženýrství a zaměřit se zejména na metodiku založenou na principu Objektově orientovaných Petriho sítí. Pokusíme se tedy tuto metodiku poupravit a provést případovou studii.

1.2 Teoretické předpoklady a souvislosti

V následujícím textu budou používány zkratky a obecné pojmy, které by bylo vhodné uvést do souvislosti, neboť jsou klíčové pro porozumění problematiky metodik softwarového inženýrství založených na modelech.

UML (Unified Modeling Language) – pravděpodobně nejrozšířenější standardizovaný grafický jazyk používaný v oblasti softwarového inženýrství.

Petriho sítě – matematická reprezentace systémů. Petriho síť lze použít v simulační oblasti jako matematický popis modelovaného systému.

Objektově orientované Petriho sítě – Petriho sítě jsou reprezentovány jako objekty, jedná se tedy o Petriho sítě doplněné o objektový přístup.

Model-driven engineering – odvětví softwarového inženýrství zabývající se návrhem softwarových systémů za pomocí modelů.

Model Driven Architecture – metodika, která vznikla z iniciativy uskupení jménem Object Managment Group (OMG), které stálo za standardizováním jazyka UML. Právě jazyk UML, resp. jeho varianta xUML, je v této metodice klíčový.

Model-Driven Software Developement – metodika vycházející z principů MDA, avšak jedná se o obecnější pojetí, bez nutnosti využívat xUML. Tato metodika může využít místo jazyka xUML jakýkoliv meta-modelový jazyk.

Smalltalk – interpretovaný, dynamicky typovaný, ale především hlavně čistě objektový programovací jazyk.

PNtalk – jazyk implementovaný pomocí jazyka Smalltalk. Jazyk slouží pro popis OOPN a je úzce spojen se simulátorem takto popsaných modelů.

2 Model-driven engineering

Modelem řízené inženýrství [1], tedy Model-driven engineering, je odvětví softwarového inženýrství, které se snaží o co nejefektivnější návrh softwarových systémů, a to prostřednictvím metodik založených na myšlence generování výsledného systému z modelů. Celá koncepce je postavena na ideji, že pomocí generátorů a transformačních prostředků se konečný zdrojový kód generuje automaticky, či poloautomaticky z konceptuálních modelů. Tento přístup nabízí jednu nesmírnou výhodu. V dnešní praxi se modely, které byly k softwaru vytvořeny, využívají spíše jako dokumentační prostředek, neboť při změně požadavků, například ze strany zákazníka, se úpravy systému provádějí převážně manuálně v již napsaném programu. Dochází tedy k situaci, kdy se výsledný kód začíná postupně vzdalovat od svých konceptuálních modelů, tedy vztah mezi konceptuálními modely a výsledným systémem není izomorfní. Na rozdíl od toho metodiky spadající do kategorie modelem řízeného inženýrství se pokoušejí o naprosto izomorfní vztah mezi modely a výsledným kódem. Tento stav je zaručen právě principem automatického generování kódu.

Jedná se tedy pouze o další stupeň abstrakce, která má pozornost programátora či návrháře směrovat od nepodstatných problémů k těm důležitým, a to ke strategickým problémům návrhu systému, jako je například správnost v rámci celého systému či výkon, kterého dosahuje. Stejně jako v historii došlo k abstrakci od jednotlivých instrukcí procesoru pomocí jazyka symbolických instrukcí (druhá generace jazyků), nebo v následující etapě se abstrakce posunula od symbolických instrukcí k vysokoúrovňovým programovacím jazykům (jazyky třetí generace), či v neposlední řadě rozvoj jazyků třetí úrovně o objektový přístup, mluvíme v poslední době o vzniku jazyků čtvrté generace.

Jazyky čtvrté, občas se označují i jako jazyky páté generace, se často srovnávají s jazyky Domain-specific language (dále DSL), které úzce souvisí s problematikou metodik spadajících pod modelem řízené softwarové inženýrství. Pojem DSL je základním stavebním kamenem pro metodiku Model-Driven Software Development (dále MDSD), o které se píše v následujících kapitolách a o které tato práce převážně pojednává.

Doménově specifické jazyky jsou zaměřeny spíše než na řešení obecných problémů, jako je tomu u jazyků třetí generace (C++, C#, Java, apod.), na konkrétní problémovou doménu. Díky tomuto přístupu lze dosáhnout značné abstrakce, která je na vyšší úrovni než u univerzálních jazyků (general-purpose languages). Tento postup lze aplikovat i u modelovacích jazyků a dostáváme se k pojmu tzv. doménově specifické modelování. Doménově specifické modelování je termín, který označuje metodologii zabývající se modelováním na vysoké úrovni abstrakce. Navíc často zahrnuje myšlenku automatického generování kódu, a to právě skrze modely, které byly vytvořeny pomocí doménově specifických jazyků.

Jednou ze známých metodik v modelem řízeném inženýrství je metodika Model Driven Architecture (dále MDA), která je založena na modifikaci známého univerzálního jazyka UML,

a to xUML (executable UML). Jazyk xUML zajišťuje podobné vlastnosti jako zmíněné doménově specifické jazyky, a to vysokou míru abstrakce, a obdobně se zaměřuje na specifické domény.

2.1 Model Driven Architecture

Model Driven Architecture [2], nebo také česky modelem řízená architektura je spíše než opravdová metodika obecný standard, který však spadá do kategorie Model-driven engineering (dále MDE). Za tímto standardem stojí konsorcium OMG (Object Management Group), které vydalo tento MDA v roce 2001. Jedná se tedy už o poměrně dlouhodobý koncept. Po prvním větším pokusu o automatické generování kódu nástrojem CASE v 80. letech se jedná o první větší krok v oblasti modelem řízeného inženýrství.

Tento standard je postaven na myšlence rozdělení návrhu do více úrovní abstrakce, které jsou reprezentovány různými typy modelů. Jedná se tedy o CIM (Computation Independent Model), PIM (Platform Independent Model) a v neposlední řadě PSM (Platform Specific Model), z něhož se následně může generovat kód pro libovolnou platformu. Modely CIM specifikují obecnou funkčnost navrhovaného systému. Zpravidla není zaručena možnost převádět modely CIM na PIM automaticky. Modely PIM jsou zaměřené mnohem více na konkrétní řešení navrhovaného systému, avšak s určitou nezávislostí na platformě, jak již vychází z názvu „platformě nezávislé modely“. Díky znovupoužitelnosti těchto modelů jsou využívány jako výchozí bod pro mnohá další obdobná zadání. Princip znovupoužitelnosti platformě nezávislých modelů je jednou z klíčových myšlenek standardu MDA, neboť zaručuje značné urychlení vývoje nových softwarových projektů.

MDA má však i své nevýhody. Zaprvé se nejedná o ucelený koncept, ale celá myšlenka je stále ve vývoji. Další nevýhodou je složitost celého konceptu a jeho nemožnost nasazení ve všech projektech, ale pouze na specifické zadání. Poslední nevýhodou je fakt, že celý standard byl vytvořen z iniciativy konsorcia OMG, které stojí i za standardizací jazyka UML, a tudíž i standard MDA je úzce spojen s jazykem UML, resp. jeho variantou xUML.

2.1.1 xUML

Jazyk UML (Unified Modeling Language) je v praxi běžně využívaný univerzální grafický jazyk pro popis softwarových systémů. Tradiční jazyk UML je nevhodný pro využití v modelem řízeném návrhu ze dvou důvodů. První důvod je ten, že jazyk UML nabízí široké množství diagramů a grafických reprezentací, které však nejsou jednoznačné. Například stavové diagramy mohou být použity hned v několika rozdílných reprezentacích. Druhým problémem je nedostatek sémantických akcí u všech elementů jazyka UML. Právě jazyk xUML je řešením těchto problémů, kdy je standard UML zbaven sémanticky slabých prvků a je doplněn o precizně definované sémantické akce. Vznikla tak podmnožina jazyka UML nazvaná xUML, díky níž jsme schopni definovat takzvané platformě nezávislé modely, které jsou nezbytné k popsání funkcionality cílového systému (algoritmy, pravidla,

atd.), a to tak, že jsou nezávislé na technologickém řešení. Z těchto platformě nezávislých modelů se pomocí systematického transformování, které se nazývá mapování, derivují platformě specifické modely a následně může být automaticky generována tzv. platformě specifická implementace, tedy cílový zdrojový kód.

2.2 Model-Driven Software Developement

Stěžejní princip pro tuto práci je metodika Model-Driven Software Developement [3] (dále MDSD). Jde o metodiku, která vychází z předchozích konceptů jako je např. MDA. Oproti MDA má však nesmírnou výhodu, že není svázána s konkrétním komerčním jazykem, jako je tomu u MDA - s jazykem UML, resp. xUML. Navíc dosavadní metodiky sice podporují automatické generování kódu, ale pořád je v mnoha případech nutné doladit finální kód manuálními úpravami a opět vzniká již zmiňovaný problém nesouladu kódu s konceptuálními modely. Pro modelování výsledného systému mohou být u metodiky MDSD použity meta-modely, např. lze použít doménově specifické jazyky. Díky této flexibilitě je na rozdíl od přístupu MDA možnost aplikovat MDSD na mnohem více projektů a zadání.

2.2.1 Objektově orientované Petriho sítě

V případě této práce se pro potřeby modelování, případně následného simulování, zaměříme na Objektově orientované Petriho sítě (dále OOPN), které budeme ke zmiňovaným potřebám využívat. OOPN jsou vhodné pro využití z mnoha důvodů. Hlavním důvodem je bezesporu to, že tento druh Petriho sítí podporuje návrh distribuovaných objektově orientovaných programových systémů. Dále umožňují reprezentaci tříd, a následně jejich instancí modelovaného objektově orientovaného programového systému s možností deklarace a definice datových položek, metod a událostí. V neposlední řadě je velice důležitý ten fakt, že objektové Petriho sítě jsou postaveny na formální matematické teorii, která podporuje prostředky pro modelování sítí, jež reprezentují výsledný systém a zároveň umožnují následnou analýzu systému.

2.2.2 DEVS

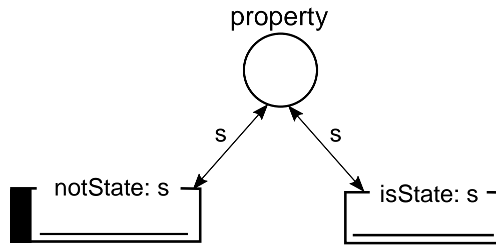
Při představení formalismu OOPN si nastíníme druhý formalismus, který se jmenuje DEVS [4] (Discrete Event System Specification). Formalismus OOPN je možno, jako jiné další formalismy, zaobalit pomocí DEVS. Samotné OOPN nemají totiž možnost hierarchizace. DEVS nabízí pomocí komponentového přístupu možnost zaobalovat jednotlivé sítě do komponent a ty pomocí portů propojit ve výsledný hierarchizovaný systém, resp. model OOPN může být rozdělen na komponenty a ty se mohou následně spojit pomocí kompatibilního rozhraní.

3 Modeling and Simulation-Based Design

Modeling and Simulation-Based Design [5] (dále MSBD) je technika založená na myšlence, že testování výsledného systému by mělo být proveditelné přímo v modelech, a to nejlépe tak, že bude testování umožněno v každém kroku vývoje výsledného systému. Pomocí tohoto přístupu lze ověřit správnost modelů již při návrhu systému, a tím předejít nesprávnosti systému v pokročilém stádiu vývoje. V následující části práce se bude k reprezentování systému, tedy jeho modelování, využívat Objektově orientovaných Petriho sítí, jejichž výhody byly zmíněny v kapitole 2.2.1. Hlavní výhodou této reprezentace je to, že se jedná o čistě formální modely. Tyto modely mají oproti méně formálním modelům, jako je například UML, výhodu právě v možnosti použití simulačních přístupů, které umožňují celkové testování a analýzu systému. Méně formální modely mají naopak oproti formálním modelům výhodu v tom, že se jedná o velice efektivní a rychlé prostředky pro návrh softwarových systému. Proces návrhu softwarových systémů pomocí MSBD se dělí se na modelační fázi a simulační fázi. Tyto dvě fáze se cyklicky opakují. V tomto cyklu se modely inkrementují, resp. v každém cyklu modely upravujeme a následně je pomocí simulačních experimentů testujeme. Tento cyklus se opakuje, dokud systém není ve výsledném požadovaném stavu. V následujícím textu se zaměříme na princip využití Objektově orientovaných Petriho sítí v oblasti modelování, tedy na modelační fázi MSBD.

3.1 Využití OOPN

Před předvedením Objektově orientovaných Petriho sítí [6] v konkrétním případu modelování softwarového systému je zapotřebí si představit základní vlastnosti těchto sítí. Objektově orientované Petriho sítě se v mnohém neliší od běžných Petriho sítí. Stejně jako v P/T Petriho síti – zkratka P/T značí *Place* (místo) a *Transition* (přechod) – se v OOPN objevují místa a přechody. Místa zde však nemusí obsahovat pouze obyčejné značky, které symbolizují jen procesy, ale mohou obsahovat objekty apod. Dále pak je důležité si povšimnout, že na rozdíl od P/T Petriho sítí se v těch objektových objevují kromě běžných přechodů tzv. synchronní porty a negativní predikáty. Synchronní porty jsou přechody, které se neprovádějí samy, ale jsou dynamicky spouštěny z jiných přechodů, se kterými jsou propojeny. Propojení synchronního portu a nějakého jiného přechodu je realizováno pomocí stráže. Stráž aktivuje synchronní port pomocí zaslání zprávy.



Obrázek 1: Negativní predikát a synchronní port

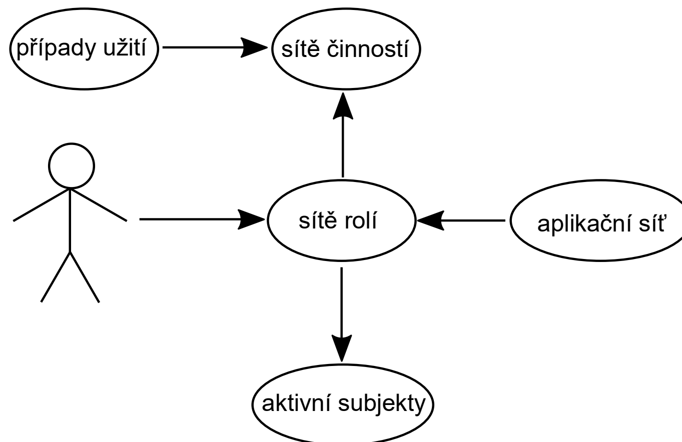
Na obrázku 1 je představen zmíněný synchronní port napravo, tedy *isState*, a negativní predikát vlevo, tedy *notState*. V tomto případě se jedná o zřetelný příklad realizace podmínky *If-Else*. Jde o kontrolu stavu systému. Místo *property* by mělo obsahovat symbol *s*. Pokud je zavolán ze stráže nějakého přechodu synchronní port *isState*, dojde ke zkontrolování, zdali se v místě tento symbol nachází. Původní přechod, který pomocí zprávy volal synchronní port *isState*, je proveditelný právě v okamžiku, kdy se v místě nachází symbol, a tedy synchronní port je proveditelný. Pokud je z nějaké stráže přechodu zavolán negativní predikát *notState*, je původní přechod proveditelný naopak tehdy, kdy se v místě *property* symbol *s* nenachází. Tímto způsobem lze testovat, zdali se systém nachází v nějakém daném stavu.

Obrázek 2: Volání synchronního portu

První zmíněný případ, tedy volání synchronního portu z jiného přechodu, resp. jeho stráže, je zjednodušeně naznačen na obrázek 2, kdy přechod *t1* volá ve své stráži již zmiňovaný synchronní port z obr. 1. Do místa *p2* se z přechodu *t1* dostaneme, pokud je proveditelný synchronní port. V přechodu je pod stráží naznačeno, že v rámci přechodu lze definovat další akce, které jsou proveditelné, pokud je splněna stráž. Zde popsaný princip je klíčový v návrhu softwarových systémů pomocí OOPN, neboť, jak již bylo zmíněno, umožnuje testování systému.

3.2 Modelovací technika

Modelovací technika popisuje způsob, kterým se vytváří výsledný systém, tedy jakým způsobem se využívají modely k popisu softwarového systému. Zaměříme se na nastínění tohoto principu, a to zejména na to, jakým způsobem jsou jednotlivé sítě zanořeny do sebe a jak na sebe navzájem navazují.



Obrázek 3: Schéma závislostí jednotlivých prvků návrhu

Obecné schéma, které zobrazuje jednotlivé vrstvy návrhu, je vyobrazeno na obrázku 3, přičemž šipky značí, která síť zapouzdřuje jinou. Návrh softwarového systému se skládá z následujících činností:

1. identifikace případů užití (obecně známých pod angl. názvem Use Cases),
2. specifikace rolí systému a aktivních subjektů,
3. specifikace sítě činností – dekompozice komplexní činnosti na jednodušší činnosti,
4. specifikace aplikační sítě – lze přirovnat k uživatelskému rozhraní.

Návrh softwarových systémů pomocí OOPN je úzce spojen s běžným návrhem systémů pomocí UML. Pro běžnou obecnou specifikaci případů užití lze použít Use Case diagram ze standardu UML za předpokladu, že později jsou tyto modely specifikovány pomocí OOPN. Dále je například úzká vazba mezi sítěmi rolí definovaných v OOPN a rolemi specifikovanými v UML. Role je spolu se subjekty a aktivitami základními prvky modelu. Princip je takový, že v systému existují tzv. subjekty. Subjekt reprezentuje individuální prvek systému. Může to být například uživatel vyskytující se v systému. Tento uživatel může nabývat více rolí. Role zde dodatečně definuje, které funkce může daný uživatel v rámci systému plnit. Např. vysokoškolský student může nabývat v rámci systému vysoké školy více rolí. Může být zároveň běžným studentem, členem studentského senátu a zaměstnancem pracovní skupiny. V následující kapitole si představenou architekturu nastíníme v praktických ukázkách, řekněme tedy v případové studii. Projdeme si klasickou sekvenci softwarového návrhu. Nejprve si představíme Use Case, poté si ukážeme určení rolí a subjektů, dále pak sítě činností a nakonec si ukážeme aplikační síť. Právě při této sekvenci se uplatňuje již zmíněný inkrementální postup návrhu, který byl představen v úvodu třetí kapitoly. Systém je tedy vyvíjen inkrementálně a v každém kroku se modelují vybrané sítě. Sítě jsou proveditelné, resp. jsme schopni je simulovat a provádět nad nimi simulační experimenty, které mohou sloužit jako vhodná forma

testování. Ze statistik a sesbíraných dat se designér následně rozhoduje, které sítě by bylo vhodné modifikovat.

3.3 PNtalk

Pro simulační a modelovací účely je vhodné použít aplikační framework PNtalk [7]. PNtalk kombinuje vysokoúrovňové Petriho sítě, tedy OOPN, s objekty jazyka Smalltalk. Objekty formalismu OOPN popsané pomocí PNtalku jsou přímo dostupné v aplikaci napsané ve Smalltalku (viz podkapitola 5.1) a naopak objekty Smalltalku jsou dostupné, skrze aplikační framework, v OOPN. V rámci PNtalku je realizován i simulátor, který je založený na formalismu DEVS. Díky tomuto spojení PNtalku a DEVS lze modely vytvářet, ale také pohodlně testovat a simulovat. Syntaxe jazyka je následující. Předpokládejme OOPN z obr. 1.

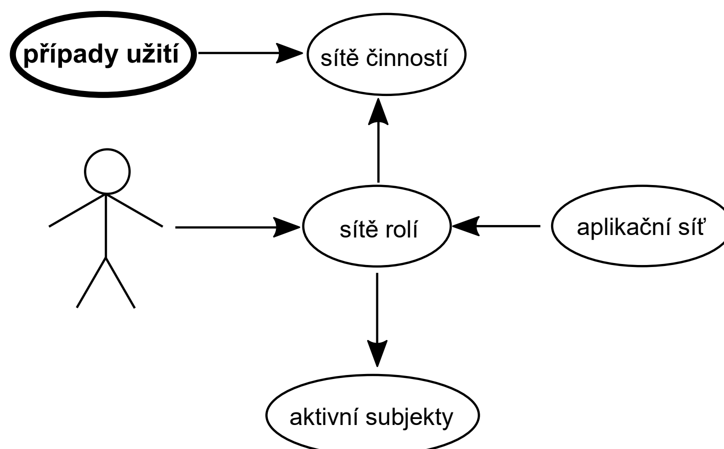
```
class ExampleNet is_a PN
object
        place property()
        inhibitor notState: s
                cond property(1`s)
        sync isState: s
                cond property(1`s)
```

Nejprve si musíme definovat, že se jedná o třídu Petriho sítí, a následně se musí uvést klíčové slovo *object*. Následuje definice všech míst, které se budou v síti vyskytovat. Poté je zapotřebí definovat všechny přechody pomocí klíčového slova *trans* (v ukázce není představeno) a následuje uvedení všech negativních predikátů (*inhibitor*) a synchronních portů (*sync*). U přechodů, negativních predikátů a synchronních portů můžeme definovat, jaké jsou vstupy (klíčová slova *cond, precond*), výstupy (*postcond*) a lze jim také přidat stráže a definovat akce. V neposlední řadě PNtalk umožnuje definovat ke každé třídě vlastní metody tak, jak jsme tomu zvyklí z běžných objektově orientovaných jazyků.

4 Případová studie

Postupy představené v předchozí kapitole se pokusíme demonstrovat na případové studii. Případová studie navazuje na předchozí práci v této oblasti softwarového návrhu [5]. Jedná se spíše o předvedení jednotlivých klíčových prvků modelovací techniky a nastínění obecného postupu, než o komplexní rozsáhlou případovou studii. Jde o velice jednoduchý rezervační systém, který obsahuje opravdu základní a jednoduché funkce, jako je například přihlášení uživatelů, či prohlížení a změny rezervací. V následujících podkapitolách bude vždy nejprve znázorněno schéma obecné architektury návrhu s vyznačeným prvkem, na který se momentálně zaměříme.

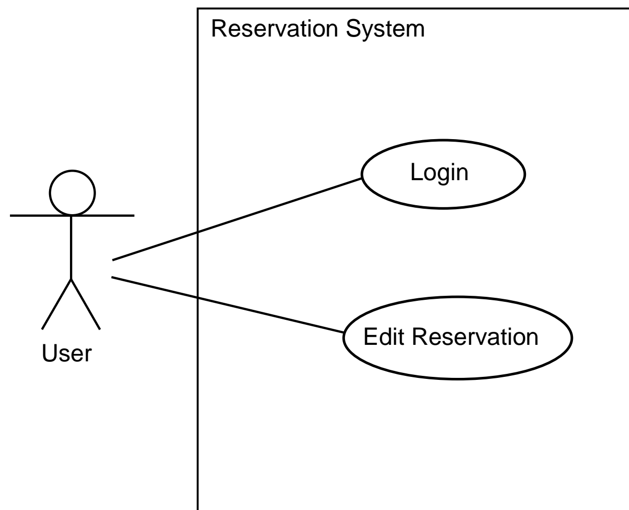
4.1 Případ užití (Use Case)



Obrázek 4: Schéma závislostí – případy užití

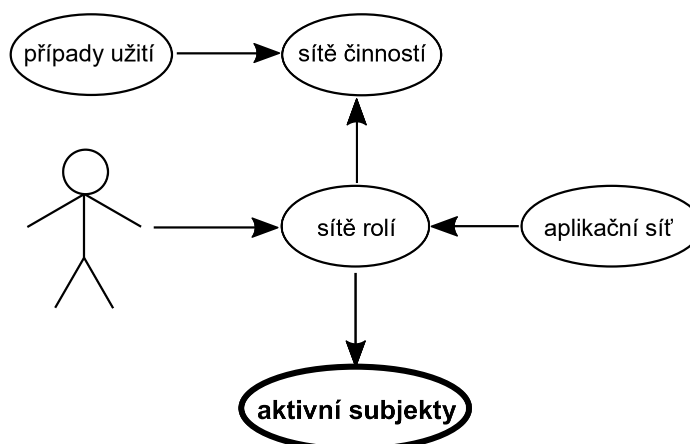
Z představeného konceptu sekvenčního návrhu systému je patrné, že prvním stavebním kamenem pro návrh je určení si případů užití. Diagram případů užití, který je definován ve standardu UML, je klíčovým diagramem v oblasti stanovení požadavků na systém. Diagram samotný nepojednává o tom, jakým způsobem bude systém řešit jednotlivou funkcionalitu, ale definuje pouze to, že daná funkcionalita je v systému obsažena. Z tohoto hlediska je to nejdůležitější diagram v oblasti prezentace vytvářeného systému, neboť je přehledný pro široké množství lidí a vytváří se v drtivé většině jako první, jelikož za pomoci tohoto diagramu dochází k dohodě na tom, co systém bude nabízet. Množina jednotlivých diagramů případů užití dohromady by měla dávat výsledné chování systému. Diagram obsahuje skupinu aktérů, ke kterým jsou definovány operace, jež konkrétní aktéři mohou provádět. Jako příklad diagramu případu užití v našem teoretickém rezervačním systému si uvedeme diagram obsahující jednoho aktéra jménem *User* a dva případy užití, a to *Login* (pro přihlášení uživatele) a *Edit*

Reservation (pro editaci rezervací). Diagram případu užití tedy znázorňuje jednoho uživatele, který je schopen vykonávat dvě zmiňované činnosti. Popsaný diagram je ukázán na obrázku 5. V následujících kapitolách si ukážeme, jak se v této prezentované modelovací technice případy užití specifikují pomocí OOPN.



Obrázek 5: Diagram případu užití

4.2 Aktivní subjekty

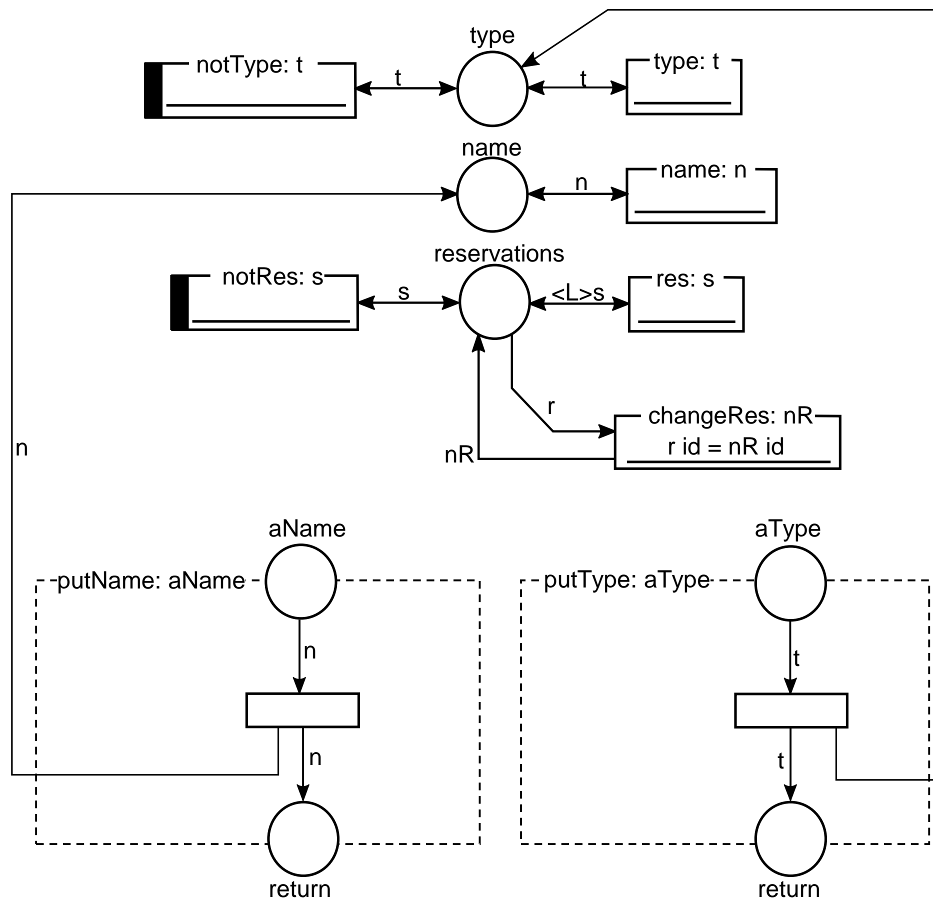


Obrázek 6: Schéma závislostí – aktivní subjekty

Aktéři, kteří jsou definováni v diagramech případů užití, mají většinou vlastní podmnožinu případů užití, se kterými jsou schopni pracovat. Jedná se o to, že systém může rozeznávat mnoho rozdílných aktérů. Ti mají však většinou podobný základ. Jsme tedy schopni specifikovat aktéry podle tzv. aktivních subjektů a rolí, které mohou tito aktéři nabývat. Například můžeme mít osobu, tedy

aktivní subjekt, a ta může nabývat rolí jako např. ředitel, správce a vedoucí. Modelováním těchto aktivních subjektů nedosahujeme určitých akcí, ale jedná se pouze o vyjádření současného stavu. Dále tímto modelem znázorňujeme možné akce, které jsou schopny subjekty vykonávat skrze své role.

Na obr. 7 můžeme vidět aktivní subjekt *Member*. Tento model uchovává o subjektu všechny elementární informace jako je jméno, typ a rezervace (v modelu *name*, *type*, *reservations*) a jejich uložení je realizováno pomocí příslušných míst. Typ reprezentuje seznam rolí, které k subjektu přísluší. Jméno a rezervace reprezentují přesně svůj název.

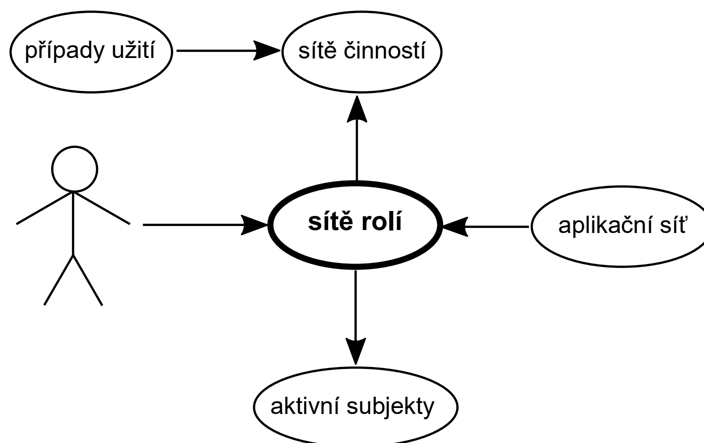


Obrázek 7: Aktivní subjekt Member – OOPN

Jak je vidět na modelu modelovaného pomocí OOPN k hodnotám uloženým v místech, jsme schopni se dostat pomocí synchronních portů. Negativní predikáty naopak fungují jako testovací prostředek, který je schopen říci, zdali je nějaká hodnota nastavena, či nikoliv. V místě *type* jsou tedy uloženy role, v místě *name* jméno subjektu a v neposlední řadě je zde znázorněno místo *reservations*. Toto místo zde slouží k ukládání rezervací. Aby bylo možné rezervace jednoduše a efektivně zobrazit, je využito mezi místem a synchronním portem *res* symbolu <L>. Symbol značí, že k proměnné *s* se naváže celý obsah místa *reservations*. Tedy po zavolání synchronního portu ze stráže nějakého přechodu (například z jiné Petriho sítě) dojde k přeposlání a například zobrazení všech rezervací

uložených v již zmiňovaném místě. Jako zajímavý prvek v tomto modelu je třeba zmínit synchronní port *changeRes*, který slouží k editování rezervace. Pokud tento synchronní port obdrží zprávu zaslanou z nějakého přechodu, nejprve zkontroluje, zdali se mezi rezervacemi nachází rezervace se stejným *id*, jako má příchozí požadavek na aktualizování rezervace. Když je tato podmínka splněna, přechod synchronní port je proveditelný a dochází k nahrazení staré rezervace za novou. Síť aktivního subjektu obsahuje také tři sítě metod (angl. *method nets*) pro vkládání hodnot na příslušná místa (*name*, *type*, *reservations)*. Dvě z nich jsou na obr. 7 znázorněny. Ve všech třech případech se nejedná o nic jiného než o tzv. „*setter*“, tedy metodu, která slouží pouze pro nastavení hodnoty. Metoda má standardně své jméno a případně parametr či parametry. Mějme například metodu *putName: aName*. Metoda má jako startovací bod první místo, tedy místo *aName*. Při zavolání metody se do tohoto místa vloží parametr metody a metoda je poté vykonána. Zde se nejedná o nic jiného než o přechod, který na příslušné místo předává hodnotu parametru. Metoda je také vybavena patřičným místem, do kterého lze případně vracet například hodnota *true/false* apod. V našem případě se vrací zadaný parametr. Stejným způsobem funguje metoda *putType:* a totožná je i metoda *addReservation:*, která zde není modelována.

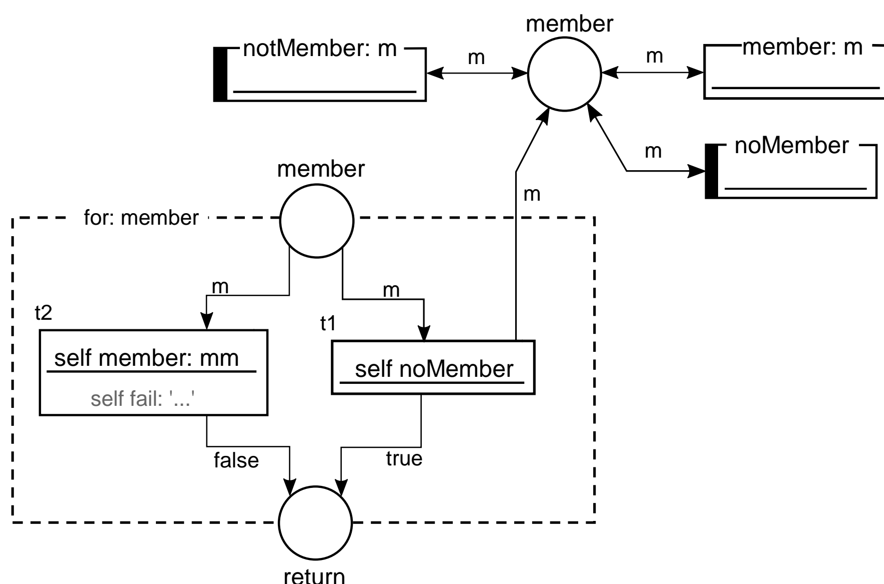
4.3 Specifikace rolí



Obrázek 8: Schéma závislostí – sítě rolí

Jak bylo zmíněno, popsaný *member* může nabývat rozdílných rolí. Jednotlivé role lze namodelovat pomocí OOPN. Základní myšlenkou je, že role by měla vědět, ke kterému subjektu patří. Na obr. 8 si ukážeme případ modelu role, konkrétně role *User*. V našem případě bude informace o tom, ke kterému subjektu role patří, uložena v místě *member*. Tato síť je opět vybavena negativními predikáty a synchronními porty. Negativní predikát *notMember* modeluje test na to, zdali role nereprezentuje daný subjekt a právě v tuto chvíli je pravdivý a provede se. Přesným opakem je synchronní port *member*, který je proveditelný, pokud *member* reprezentuje předaný subjekt. Negativní predikát *noMember* je proveditelný, pokud v danou chvíli není žádný subjekt reprezentovaný touto rolí. Atributy

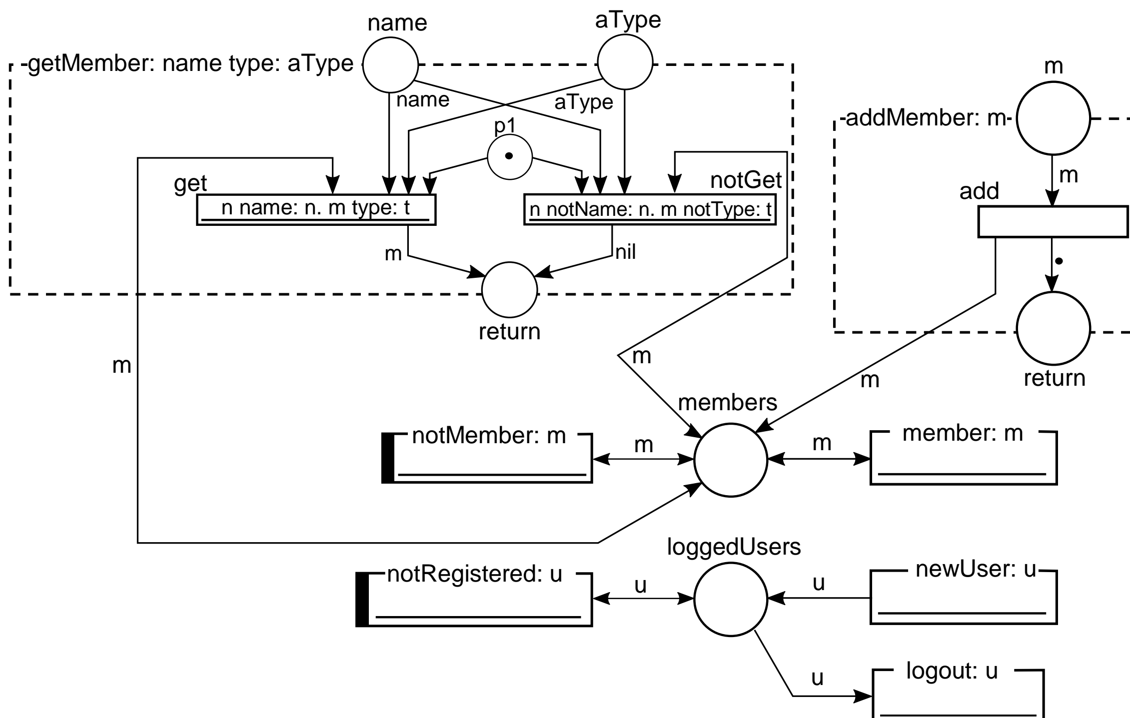
by měly být inicializovány skrze sítě metod. V modelu role *User* je jedna taková síť metody naznačena. Jedná se o síť *for: member*, která inicializuje atribut *member*. Při inicializaci dochází ke kontrole neboli testu na to, zdali atribut *member* byl již inicializovaný, či nikoliv. Zmiňovaný požadavek se namodeluje pomocí zavolání negativního predikátu *noMember* ze stráže přechodu, který je označen jako *t1*. Když je negativní predikát pravdivý, *member* není reprezentován, dojde k inicializaci atributu a zároveň je vygenerována hodnota *true*. Naopak pokud byl již *member* inicializován, dochází k tomu, že přechod *t2* je proveditelný, neboť atribut *member* byl nalezen. V tomto momentě je vrácena hodnota *false* a je možno například v rámci přechodu *t2* vygenerovat výjimku, která může být opět použita jako vhodný prostředek pro testování modelu.



Obrázek 9: Role User

4.4 Systémová role – společná síť

Když mluvíme o rolích, tak si můžeme předvést jeden speciální typ role, a tím je systém samotný (viz obr. 10). Role, která vyjadřuje systém, je modelována z důvodu perzistence, ale také z důvodu přístupu ke sdíleným objektům apod. Mějme například situaci, kdy je systém ve stavu čekání na přihlášení. Žádný uživatel není přihlášený, tedy místo *loggedUsers* je prázdné. Místo *members* zde pak slouží jako databáze všech uživatelů, kteří jsou v systému zaregistrováni. Při pokusu o přihlášení je třeba právě v tomto místě ověřit, zdali je přihlašovaný uživatel shodný s některým v databázi, kde jako složený klíč k přihlášení slouží hodnota *type* a *name*. Na obr. 10 je uveden koncept, jak by mohl model pro náš rezervační systém vypadat. Dovoluje systému uchovávat informace o *members*, které zpřístupňuje pomocí synchronního portu anebo také dovoluje zpracovávat, resp. uchovávat přihlášené uživatele

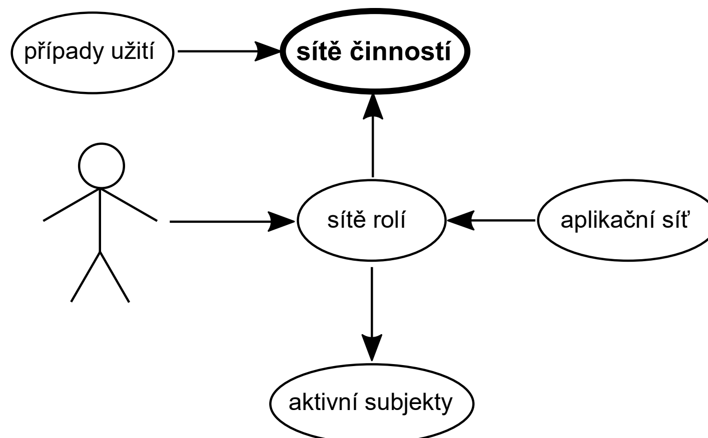


Obrázek 10: Role jako systém

v podobě místa *loggedUsers*. Uživatele se do systému přihlašují, odhlašují se a může být například přidán test na to, zdali daný uživatel v systému registrován není. Společná síť obsahuje také dvě metody. Jednodušší metodou je metoda na vložení uživatele do databáze. Tato metoda (*addMember: m*) dostává jako parametr instanci uživatele (instanci sítě subjektů) a vkládá ho na místo *members*. Druhá metoda (*getMember: name type: aType*) je implementována z důvodu ověření, zdali uživatel v databázi existuje. Jako parametry dostává klíč složený z hodnot *name* a *type*. Metoda má dva přechody, které se snaží navázat uživatele z databáze a ověřuje je na dané hodnoty. Pro přístup k hodnotám *name* a *type* lze využít synchronních portů a negativních predikátů definovaných v aktivním subjektu *Member* (obr. 7). Tyto porty jsme schopni volat ze stráže jednotlivých přechodů. Pokud byl uživatel nalezen, vrací metoda instanci uživatele. V opačném případě vrací hodnotu *nil*.

4.5 Sítě činností

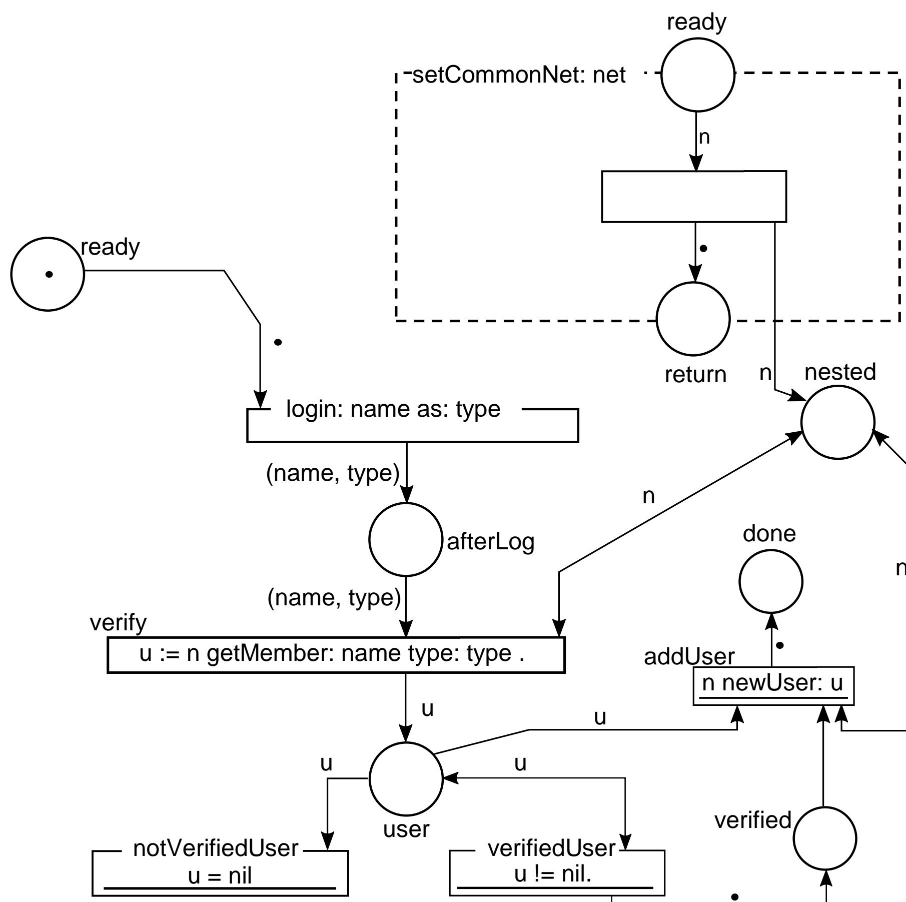
Jako třetí bod se v našem sekvenčním konceptu návrhu nacházejí tzv. sítě činností. V úvodních kapitolách jsme zmínili, že každý případ užití musí být následně popsán pomocí modelu, v našem případě musí být namodelován pomocí OOPN. K modelování využijeme běžných prostředků, jako jsou místa, přechody, synchronní porty, ale z důvodu testování mohou být doplněny o negativní predikáty. Přechody modelují události interní a synchronní porty poslouží k reprezentaci externích událostí. Místa slouží k uchování informace o tom, v jaké části procesu činnosti se právě nacházíme. Tato místa mohou být pak následně testována na své hodnoty pomocí synchronních portů a negativních predikátů.



Obrázek 11: Schéma závislostí – Sítě činností

V následujících dvou podkapitolách se zaměříme na dva případy užití, které byly představeny v diagramu v podkapitole 4.1.

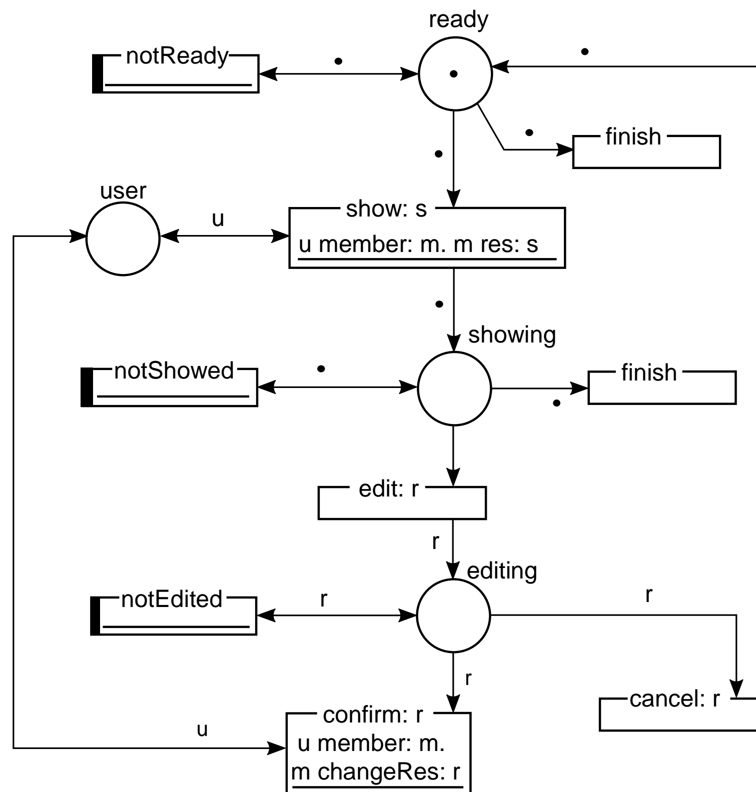
4.5.1 Případ užití – Login



Obrázek 12: Případ užití Login modelován pomocí OOPN

Prvním případem užití z diagramu, který je ukázán jako model třídy OOPN na obrázku 12, je *Login*. Elementární činnost každého systému je proces, kdy je uživatel schopen se do systému přihlásit, přičemž většinou se uživatel přihlašuje pod nějakou rolí, tedy v tomto případě pomocí hodnoty *type* a dále pomocí svého jména. Získání těchto údajů je realizováno synchronním portem (tedy externí událost) *login:as:*. Tento port získá do OOPN reprezentující případ užití potřebné informace k ověření. Po získání hodnot *type* a *name* je v přechodu *verify* volána metoda instance tzv. společné sítě, tedy databáze, která byla představena na obr. 10. Po provedení tohoto přechodu můžeme zavolat jeden ze synchronních portů, a to buď *notVerifiedUser*, nebo *verifiedUser*. V prvním případě se jedná o selhání autorizace uživatele a činnost přihlašování končí. V druhém případě však činnost proběhla úspěšně a role, pod kterou se uživatel snažil přihlásit, je pomocí zavolání synchronního portu *newUser*, který je znázorněn v kapitole 4.4 na obrázku 10, přidána do systému. Tímto způsobem se uživatelé mohou přihlašovat do systému a systém si následně přihlášené uživatele pamatuje ve speciální systémové roli.

4.5.2 Případ užití – Editace rezervací

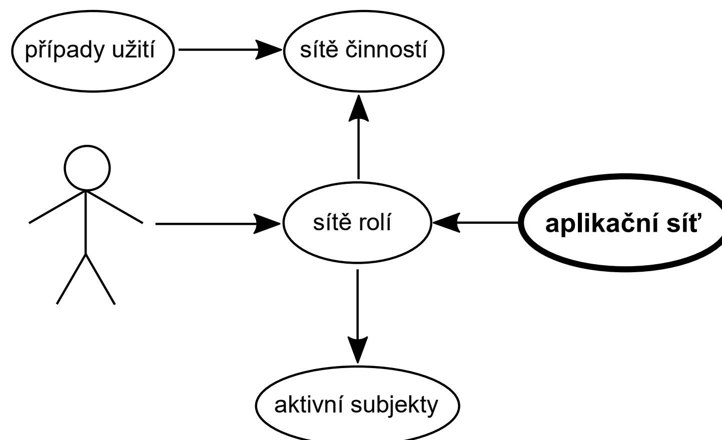


Obrázek 13: Případ užití EditReservation modelován pomocí OOPN

Druhým případem užití je editace rezervací, která je znázorněna pomocí *EditReservations*. Tento případ užití je opět popsán pomocí objektové sítě ve formalismu OOPN na obr. 13. V případě editace rezervací

je zapotřebí, aby si síť činnosti byla vědoma toho, ke které roli patří. Tento fakt je modelován pomocí místa *user*. Tento atribut by měl být nejlépe inicializován metodou sítě OOPN. Celá činnost rezervace se skládá z několika následujících základních stavů. Prvním stavem je proces prohlížení rezervací. Pomocí navázaní uživatele (volání synchronního portu *member*) a následného zobrazení všech existujících rezervací pro daného uživatele se dostaneme do stavu prohlížení (*showing*) rezervací, které patří k danému uživateli. Tuto akci můžeme provést ze stráže synchronního portu *show*. Ten zjišťuje z přilehlého místa, o jakého uživatele se jedná, a následně ve své stráži provede zmíněné akce. Poté může dojít k editaci (externí událost *edit*) a nakonec ke zrušení pomocí externí události *cancel* nebo potvrzení pomocí externí událost *confirm*. Zavoláním synchronního portu *cancel* dochází tedy k navrácení do výchozího stavu *ready*. Je dobré si povšimnout, že změny stavů, tedy kromě okolí stavu *editing*, jsou modelovány pouze jako jednoduché procesy, neboť síť činností v těchto stavech neprovádí změny v systému. Synchronní port *confirm* pak ve své stráži volá synchronní port *member* pro získání instance konkrétního uživatele a následně volá nad tímto uživatelem synchronní port pro změnu rezervace. Negativní predikáty, o které byl model doplněn, jsou opět použity jako prostředek pro testování, v kterém stavu se činnost právě nachází, a jsou vždy platné, pokud se právě činnost v daném stavu nenachází. Prázdné synchronní porty *finish* slouží k ukončení případu užití, tedy činnosti. Operaci ukončení lze provést ze stavů *ready* a *showing*.

4.6 Aplikační síť



Obrázek 14: Schéma závislostí – Sítě činností

Jedná se o řídící síť, kterou si lze představit jako grafické uživatelské rozhraní. Tato síť spouští všechny prvky interakce uživatele s daným systémem. Konkrétní ukázku aplikační sítě si představíme v následující kapitole, konkrétně při demonstraci návrhu aplikace pomocí OOPN.

5 Aplikace

V následující kapitole bude představen koncept aplikace, která odpovídá případové studii popsané v kapitole číslo 4. Jedná se o rezervační systém se standardním grafickým uživatelským rozhraním. Z důvodu předešlé práce na aplikačním frameworku, který se jmenuje PNtalk, byla aplikace navržena v jazyce Smalltalk. Celý framework je totiž navržen taktéž v jazyce Smalltalk.

Demonstrace celého konceptu je rozdělena do dvou částí. Nejprve je v podkapitole 5.2 představena struktura aplikace, která byla pro účel demonstrace navržena a naprogramována. V podkapitole 5.3 se poté využije naprogramované aplikace, resp. pouze jejího grafického uživatelského rozhraní a dojde k propojení s OOPN. Představíme si tedy princip realizace aplikace pomocí OOPN.

5.1 Smalltalk

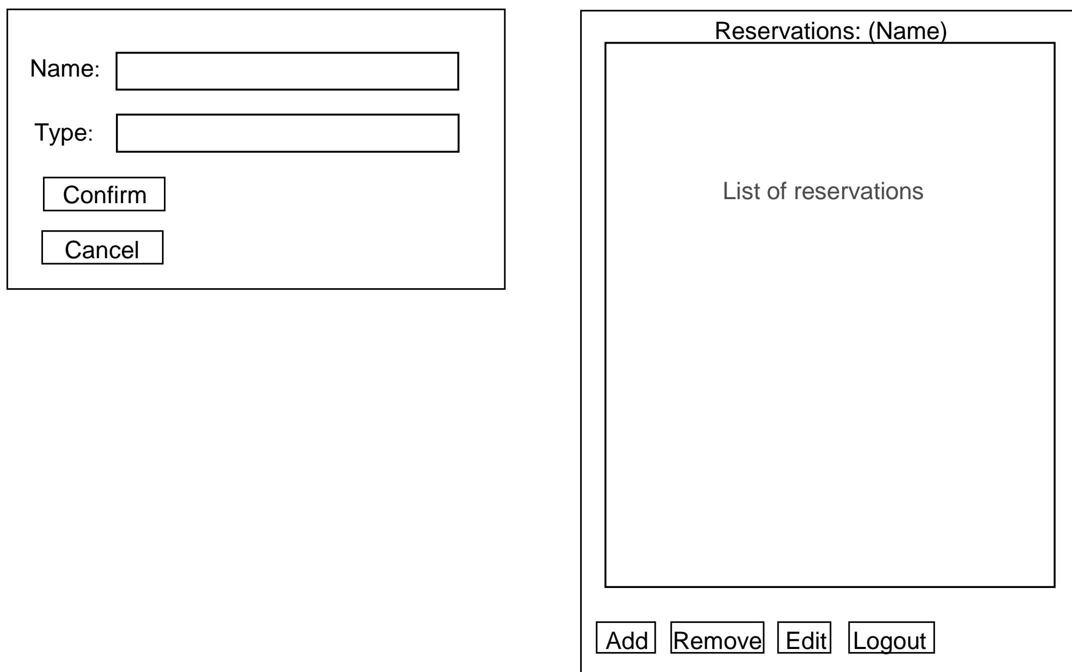
Krátkou zmínku věnujme implementačnímu jazyku Smalltalk, ve kterém jsou klíčové prvky této práce implementovány. Jedná se o interpretovaný, dynamicky typovaný, ale především hlavně o čistě objektový programovací jazyk. Všechny datové typy jsou v tomto jazyce objekty. Smalltalk se dnes vyskytuje v mnoha svobodných implementacích. Pro následující aplikaci, stejně jako pro framework PNtalk, bude použita implementace Pharo [8]. Jde o svobodnou implementaci jazyka Smalltalk v kombinaci s integrovaným vývojovým prostředím, které podporuje tzv. „*live programming*".

5.2 Rezervační systém ve Smalltalku

Podkapitola 5.2 popisuje implementaci rezervačního systému v jazyku Smalltalk. Jedná se o klasický přístup při návrhu aplikace pomocí objektově orientovaného jazyka. OOPN v této fázi nejsou použity a celá aplikace je realizovaná čistě pomocí jazyka Pharo Smalltalk a jeho vestavěných knihoven.

5.2.1 Návrh aplikace

Aplikace jako taková je navržena pro demonstrační účely a je tedy poměrně jednoduchá. Jak můžeme vidět na obr. 12, jedná se o dvě standartní obrazovky. První by měla sloužit k přihlášení uživatele, tedy k běžné kontrole na to, jestli zadané údaje odpovídají některému ze zaregistrovaných uživatelů. Je vybavena dvěma tlačítky a dvěma textovými poli. Do textových polí se vyplňuje jméno uživatele spolu s typem uživatele, resp. jeho rolí v systému. Poté lze pomocí tlačítek „*Confirm*" a „*Cancel*" potvrdit, či zrušit přihlašování. Druhá obrazovka pak slouží k zobrazení rezervací, které přísluší konkrétnímu uživateli, jenž se právě přihlásil. Rezervace lze přidávat, editovat a popřípadě mazat. Lze se také pomocí tlačítka „*Logout*" odhlásit a tím se vrátit na úvodní přihlašovací okno.



Obrázek 15: Mock-up výsledné aplikace ve Smalltalku

5.2.2 Data v aplikaci

Aplikace obsahuje dvě databáze *databaseUser* a *databaseReservation*. Nejedná se o databáze jako takové, ale jsou realizovány pomocí *OrderCollection* na straně třídy, resp. tříd *User* a *Reservation*. Obě jmenované třídy pak mají na straně třídy i přístupovou metodu, která je schopna vrátit celou databázi. Pomocí třídní metody lze k databázi uživatelů, či rezervací přistupovat z libovolné třídy a metody, a to jejím voláním. Databáze uživatelů obsahuje všechny uživatele, kteří jsou v systému vytvořeni, resp. všechny instance třídy *User* s jejími proměnnými *Name* a *Type*, které reprezentují jednotlivé uživatele. Databáze rezervací naopak reprezentuje všechny rezervace, které systém zaregistroval. Rozlišení toho, která rezervace patří ke kterému uživateli, je realizována přidáním, kromě standardních informací o rezervaci (*thing*, *time*, *purpose*), informace o tom, který uživatel rezervaci vytvořil. K rezervaci jsou tedy přidány opět proměnné *Name* a *Type*.

5.2.3 Struktura a grafické uživatelské rozhraní

Jak již bylo zmíněno, architektura programu obsahuje dvě třídy *User* a *Reservation*. Tyto třídy slouží k reprezentaci dat. Pro reprezentování grafických prvků programu byly vytvořeny dále třídy: *Logger*, *ReservationEditor* a *ReservationListEditor*.

První třída *Logger* funguje jako přihlašovací okno. Reprezentuje tedy první obrazovku z podkapitoly 5.2. Při spuštění aplikace se objeví právě toto okno. Třída implementuje de-facto pouze chování tlačítek pro potvrzení a zrušení. Po naplnění textových polích a zmáčknutí tlačítka se volá

právě třídní metoda třídy *User*. Pomocí ní se ověří, zdali je nebo není uživatel v databázi. Při nalezení uživatele je zavolána metoda třídy *ReservationListEditor*, která zobrazuje okno s rezervacemi.

ReservationListEditor je tedy hlavní jádro aplikace, která realizuje rezervační systém. Dovoluje uživateli provádět veškeré důležité operace nad jeho rezervacemi. Implementuje grafické zobrazení seznamu rezervací pro daného uživatele a intuitivně pomáhá uživateli spravovat jeho rezervace pomocí vybrání některé z položek seznamu a následnou akcí nad ní anebo je pomocí tohoto okna schopen uživatel vytvořit novou rezervaci. Pro editaci, či vytvoření rezervace byla vytvořena poslední třída *ReservationEditor*.

ReservationEditor zastává funkci modálního okna. Instance tohoto modálního okna je vytvořena v případě pokusu o editaci existující rezervace nebo za předpokladu, že uživatel chce vytvořit rezervaci novou. Okno obsahuje vyplnění údajů o rezervaci a samozřejmě tlačítka na potvrzení a zrušení rezervace.

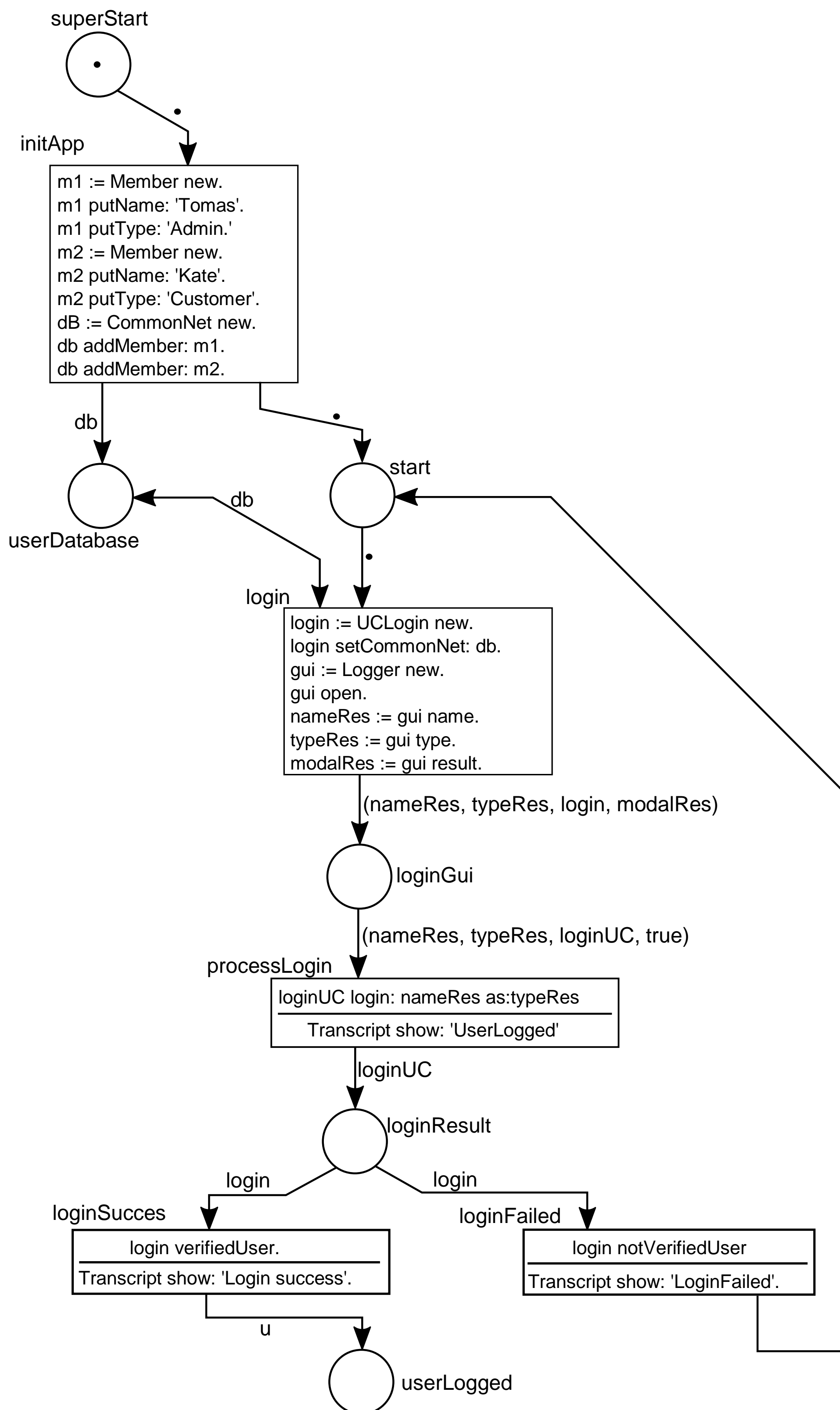
Tyto tři popsané třídy dohromady tvoří grafické rozhraní výsledné aplikace. Pro realizaci grafických prvků, které se v systému vyskytují, byly použity standardní třídy a metody, které nabízí repozitář vývojového prostředí Pharo Smalltalk.

5.3 Rezervační systém řízený simulací OOPN

V minulé podkapitole 5.2 byla představena konvenční aplikace realizovaná pomocí jazyka Smalltalk bez využití formalismu OOPN. Nyní si představíme koncept, jakým způsobem lze aplikace realizovat pomocí OOPN. Využijeme pouze grafických tříd navržených v klasické aplikaci. Celý běh aplikace je řízen v rámci Objektově orientovaných Petriho sítí, které pouze v určitých bodech spouštějí grafické prvky pro získání dat od uživatele. V následujícím textu budeme předpokládat, že sítě představené v kapitole 4 byly popsány jazykem PNtalk. Představíme si, jakým způsobem lze navrhnout aplikační síť pro prezentovaný rezervační systém. Aplikační síť bude hlavní síť pro vytvoření simulace a bude řídit celý tok programu.

5.3.1 Aplikační síť

Aplikační síť je jediná síť, která nebyla detailněji představena v případové studii v kapitole 4. Jedná se o řídící síť, která je vytvořena jako prvotní síť celé simulace, tato síť tedy inicializuje celou aplikaci a následně ji řídí. Tato síť se stará o tok programu a vyvolává grafické prvky, případně spouští jednotlivé aktivity definované pomocí jednotlivých sítí činností. V naši aplikační síti, resp. v celé simulaci systému, je startovním bodem místo *superStart* a přechod *initApp*. Za předpokladu, že je aplikační síť vložena do simulace, dojde po spuštění simulace k vykonávání sítě vyobrazené na obr. 16. V tomto momentě je připravena značka na místě *superStart* a přechod *initApp* je proveditelný.



Obrázek 16: Aplikační síť pro přihlášení uživatele

Pro demonstraci je představena část aplikační sítě, která řídí interakci uživatele při pokusu o přihlašování. Nejprve si představíme inicializační část a následně část sítě, kdy se předávají získané informace od uživatele do toku programu pomocí grafických prvků.

5.3.2 Inicializace aplikace

Přechod *initApp* slouží k vytvoření dvou subjektů definovaných pomocí sítě aktivních subjektů *Member*, která je popsána v podkapitole 4.2. Tato síť disponuje metodami na vložení atributů jména a typu, kterých subjekt nabývá. Druhá důležitá akce je vytvoření instance tzv. společné sítě, která je popsána v podkapitole 4.4. Tato síť modeluje databázi systému. Modeluje uložení aktivních subjektů, tedy těch, kteří jsou schopni do systému vstoupit. Dále si uchovává databázi uživatelů, kteří se úspěšně do systému přihlásili. Do instance společné sítě jsou na místo *members* vloženi dva právě vytvoření uživatelé (dvě instance třídy *Member*). Databáze (společná síť) zůstává dostupná na úrovni aplikační sítě skrze místo *userDatabase*, do které je uložen odkaz na právě vytvořenou společnou síť.

5.3.3 Volání GUI z OOPN

Po inicializování aplikace se aplikace dostává do stavu *start*. Zde začíná běh aplikace tak, jak ji vidí uživatel. Nyní přichází na řadu grafické uživatelské rozhraní navržené v klasické aplikaci napsané v jazyku Smalltalku. Jelikož OOPN jsou popsány pomocí aplikačního frameworku PNtalk, lze v sítích jednoduše volat třídy a jejich metody napsané v jazyce Smalltalk. Mějme například aktivitu přihlášení uživatele do systému. Na obr. 16 je vidět přechod *login*. Tento přechod slouží k započetí aktivity přihlášení. Je vytvořena instance třídy *UCLogin*, tedy instance případu užití *Login* z podkapitoly 4.5.1. Místo *nested* obsažené v této instanci sítě slouží pro přístup k databázi. Je nutné tedy pomocí metody *setCommon*, která je definována v *UCLogin*, vložit databázi do právě spuštěného případu užití. Pro získání dat od uživatele lze využít třídy *Logger* z klasické aplikace jenom s tím rozdílem, že nyní třída *Logger* nevykonává žádné akce, ale pouze získává hodnoty od *name* a *type* od uživatele. Dále je okno třídy *Logger* upraveno na modální. Přechod tedy čeká na odezvu od uživatele. Pomocí přechodu *processLogin* dochází ke spuštění synchronního *login:as:*. Tento port má za účel navázat do případu užití *Login* hodnoty zadané uživatelem. Následně je spuštěna sekvence přechodů na ověření uživatele. Simulace se poté může odvíjet dvěma způsoby. Buď uživatel nebyl nalezen a vrací se do místa *start,* nebo může pokračovat v běhu aplikace pro právě přihlášeného uživatele. Obdobným způsobem lze tedy namodelovat i další sekvenci přechodů, které řídí ostatní sítě při případu užití editace rezervací.

5.4 Možné změny

Jelikož se jedná o jednoduchý editační systém, který uvažuje pouze jeden druh uživatele, lze přenechat většinu řízení čistě na aplikační síti. Pokud by byl projekt rozsáhlejšího rázu, je nutnost dodržení konceptu spouštění případů užití z rolí nezbytná. Při rozdílné úrovni práv přihlašovaných uživatelů dojde totiž k nabízení rozdílných funkcionalit a modelování této skutečnosti čistě v aplikační síti by se stalo neúnosným.

6 Shrnutí výhod a nevýhod metodiky

Metodika, kterou se tato práce zabývala, tedy Model-Driven Software Developement realizovaná pomocí využití formalismu Objektově orientovaných Petriho sítí v kombinaci s druhým formalismem nazvaným DEVS (Discrete Event System Specification), se řadí mezi moderní a revoluční přístupy v oblasti vývoje softwarových systémů. Jedná se o realizování myšlenky, kdy modely, které doposud sloužily spíše jako stavební plány k tvorbě systému, budou navíc sloužit i jako přímá implementace výsledného systému. Navazuje na principy a poznatky získané z předchozích prací v oblasti modelem řízeného návrhu softwarových systému, jako je například obecný standard Model Driven Architecture, který využívá pro modelování systému poupraveného jazyka UML, tedy xUML.

Model-Driven Software Developement je na rozdíl od Model Driven Architecture metodika osvobozená od konkrétního modelačního prostředku. Pro modelování lze využít jakýchkoliv formálních modelů. V našem případě byly použity OOPN. Jedná se o čistě formální objektově orientované modely. Modely OOPN zde slouží k vyjádření jednotlivých sítí, které reprezentují subjekty, role a aktivity potřebné k definování systému (viz podkapitola 3.2).

Metodika Model-Driven Software Developement založená na modelech OOPN je postavena na myšlence Modeling and Simulation-Based Design, která předpokládá, že validita modelů může být testována přímo v modelech samotných. Pro tento účel byl využit aplikační framework PNtalk. Tento framework propojuje modely formalismu OOPN a jazyka Smalltalk. Dále dovoluje modely simulovat a tedy i testovat. Pro testování se v OOPN využívá vlastnosti synchronních portů a negativních predikátů (podkapitola 3.1), které umožňují testovat, zdali se systém nachází v určitých stavech. Modeling and Simulation-Based Design také předpokládá, že jsme schopni měnit prvky modelů při běhu simulace. Na tomto místě byl využit formalismus DEVS, který umožňuje modely rozdělit do tzv. komponent. Tyto komponenty lze v DEVS libovolně zaměňovat, aniž by došlo ke změně ostatních komponent, za předpokladu, že komunikují pomocí stejného rozhraní, tedy kompatibilních portů. DEVS také umožňuje hierarchizovat modely OOPN.

6.1 Efektivita

Zmíněný postup zajišťuje realizaci hlavních myšlenek, které jsou pro Model-driven engineering (viz kapitola 2) charakteristické. Největším přínosem je urychlení vývoje softwarových systémů, avšak společně s faktem, že nedochází ke ztrátě kvality v oblasti dokumentace vyvíjeného softwaru. Modely totiž slouží jako implementace samotná. Není nutné modely transformovat ve výsledný kód, neboť v tomto případě jsou modely implementací a naopak, tedy vztah mezi modely a implementací je izomorfní. Teoretické urychlení vývoje spočívá v možnosti simulovat a testovat modely v inkrementačním cyklu.

Právě popsaný postup má však i své úskalí. Hlavním problémem jsou zvýšené nároky na návrháře systému. Návrháři by si museli osvojit poměrně značně odlišný způsob návrhu softwarových systémů. V dnešním světě informačních technologií se jeví standard UML, který slouží právě k popisu softwarových systému, jako nerozšířenější a nejpoužívanější grafický jazyk. V našem případě by si návrhář musel osvojit Objektově orientované Petriho sítě. Pro tyto účely by bylo vhodné zpracovat rozsáhlý soubor příkladů modelování reálných konkrétních zadání pomocí OOPN, například v porovnání s modelováním totožného systému pomocí notace jazyka UML. Kromě zvládnutí problematiky okolo OOPN se musí v současném stavu práce návrhář vypořádat s jazykem Smalltalk, který je poměrně jedinečný svým čistě objektově orientovaným přístupem. Jazyk Smalltalk je důležitý z důvodu, že aplikační framework PNtalk pro OOPN je úzce spojen s implementací jazyka Smalltalk, Pharo Smalltalk.

6.2 Vývojové prostředí

S vytvářením softwarových systémů je neoddělitelně spojen i podpůrný software pro rychlejší a efektivnější tvorbu softwarových řešení. V tomto případě se jako největší slabina jeví absence propracovaného nástroje na modelování a simulování OOPN. Kvalitní a intuitivní grafický nástroj pro tvorbu OOPN, který by navíc dokázal spolupracovat s aplikačním frameworkem PNtalk, by značně usnadnil a urychlil vývojový cyklus. Návrhář by mohl být schopen graficky vytvářet a na místě ihned testovat a propojovat modely OOPN.

V tomto bodě je dobré zmínit problematiku hledání chyb v logice programu. V simulačním prostředku lze sice pozorovat, v jakém stavu síť skončila, případně kde se právě nachází. Chybí však možnost krok po kroku procházet modely a sledovat jejich chování. V případě chyby je navíc velice obtížné chybu odhalit, protože překladač jazyka PNtalk vrací pouze obecné informace o vzniklé chybě.

6.3 Změna vývojového cyklu

Dále se rozšiřuje množství práce, kterou musí návrhář práce zvládnout. Nejedená se už o pouhý návrh, ale jedná se o konkrétní implementaci, jejíž validita je ověřena dostatečným a kvalitním simulováním, resp. testováním zhotovených modelů. Oblast návrhu zde začíná splývat s oblastí implementace výsledného systému. Otázkou například zůstává adaptabilnost tohoto přístupu na velké projekty, na kterých se podílí mnoho rozdílných návrhářů a vývojářů a je potřeba často pracovat paralelně na totožném problému v pracovních týmech, které jsou od sebe mnohdy poměrně vzdálené.

7 Závěr

Předmětem této bakalářské práce bylo nejprve analyzovat současný inovativní směr softwarového inženýrství, který je založen na myšlence řízení návrhu softwarových systémů pomocí modelů. V první části bylo nutné prozkoumat dosavadní práci v tomto odvětví softwarového inženýrství. Druhá část se zabývala konkrétním přístupem v oblasti modelem řízeného inženýrství, a to metodikou založenou na Objektově orientovaných Petriho sítích.

Druhá část byla rozdělena na dvě poloviny. V první části došlo k představení principu návrhu softwarových systémů pomocí OOPN, resp. k představení jednoduché případové studie, která se zabývala prostým rezervačním systémem. V této části jsme se zaměřili na všechny klíčové prvky a na celkovou dekompozici systému a jeho aktérů na jednotlivé OOPN. V druhé části pak byla případová studie uvedena do praktického využití. Nejprve byla vytvořena konvenční grafická aplikace v jazyce Smalltalk. Jazyk Smalltalk byl daný předchozí prací v oblasti využití OOPN v návrhu softwarových systémů. Jednalo se o velice jednoduchou aplikaci podporující základní grafické prvky. Poté byly do konvenční aplikace pomocí aplikačního frameworku PNtalk zakomponovány OOPN, resp. grafické prvky byly demonstračně propojeny s OOPN. Podařilo se tedy demonstrovat použitelnost OOPN v oblasti návrhu softwarových.

Jako poslední část této práce se nacházela krátká úvaha výhod a nevýhod návrhu softwarových systémů pomocí formálních modelů, v našem případě tedy OOPN. Jedná se o zkušenost získanou při návrhu jednoduché aplikace rezervačního systému pomocí modelů a vestavěných grafických tříd jazyka Smalltalk.

Literatura

[1] SCHMIDT, Douglas C. Model-driven engineering. *IEEE Computer*. 2006, 32(2), 25-31.

[2] RAISTRICK, Chris. *Model driven architecture with executable UML*. New York: Cambridge University Press, 2004. ISBN 0521537711.

[3] TEKINERDOGAN, Bedir. Experiences in teaching a graduate course on model-driven software development. *Computer Science Education*. 2011, 21(4), 363–387. ISSN 0899-3408.

[4] KOČÍ, Radek a JANOUŠEK, Vladimír. System Composition Using Petri Nets and DEVS Formalisms. In: *The Ninth International Conference on Software Engineering Advances*. Nice: Xpert Publishing Services, 2014, 309-315. ISBN 978-1-61208-367-4.

[5] KOČÍ, Radek a JANOUŠEK, Vladimír. Modeling and simulation-based design using Object-oriented Petri nets: a case study. In: *Proceeding of the International Workshop on Petri Nets and Software Engineering 2012*, vol. 851. CEUR, 2012, 253–266.

[6] JANOUŠEK Vladimír. *Modelování objektů Petriho sítěmi*. Brno: Ústav informatiky a výpočetní techniky FEI VUT, 1998.

[7] PNtalk Project [online]. [cit. 2016-05-10]. Dostupné na URL: <http://perchta.fit.vutbr.cz:8000/projekty/12/>

[8] Pharo [online]. [cit. 2016-05-10]. Dostupné na URL: <http://pharo.org/>

Seznam příloh

Příloha A. Obsah CD s přiloženým obrazem Pharo Smalltalk obsahující aplikaci Reservačního systému

Příloha B. Návod pro spuštění demonstrační aplikace Reservačního systému

Seznam Ilustrací

Obrázek 1: Negativní predikát a synchronní port 9

Obrázek 2: Volání synchronního portu 9

Obrázek 3: Schéma závislostí jednotlivých prvků návrhu 10

Obrázek 4: Schéma závislostí – případy užití 12

Obrázek 5: Diagram případu užití 13

Obrázek 6: Schéma závislostí – aktivní subjekty 13

Obrázek 7: Aktivní subjekt Member – OOPN 14

Obrázek 8: Schéma závislostí – sítě rolí 15

Obrázek 9: Role User 16

Obrázek 10: Role jako systém 17

Obrázek 11: Schéma závislostí – Sítě činností 18

Obrázek 12: Případ užití Login modelován pomocí OOPN 18

Obrázek 13: Případ užití EditReservation modelován pomocí OOPN 19

Obrázek 14: Schéma závislostí – Sítě činností 20

Obrázek 15: Mock-up výsledné aplikace ve Smalltalku 22

Obrázek 16: Aplikační síť pro přihlášení uživatele 24

A Obsah CD

- /Pharo 4.0/ – složka s vývojovým prostředím Pharo Smalltalk obsahující obraz s rezervačním systém
- /BP_PDF/ – bakalářská práce ve formátu PDF
- /BP_WORD/ – bakalářská práce ve zdrojovém formátu Word
- licence.txt – licence spojené s vývojovým prostředím
- README.txt – návod na spuštění vývojového prostředí Pharo a rezervačního systému

B Návod pro spuštění

Na CD je přiloženo Pharo. Pharo Smalltalk je svobodná implementace jazyka Smalltalk. Kromě implementace jazyka se jedná o ucelené vývojové prostředí pro tento čistě objektově orientovaný jazyk. Více informací o Pharo lze najít na: http://pharo.org/. Pharo lze na systémech Windows spustit z CD pomocí /Pharo4.0/Pharo.exe.

Návod na spuštění Rezervačního systému (operační systém Windows):

1. Otevřít složku Pharo 4.0.
2. Spustit vývojové prostředí pomocí Pharo.exe.
3. Systém automaticky otevře obraz obsahující Rezervační systém.
4. V levém panelu (*My Repository*) lze prozkoumat OOPN pro rezervační systém.
5. V okně *Playground* je předpřipravena třída *simulation* a *ReservationListEditor*.

Spuštění aplikace založené na simulaci OOPN:

1. Pro spuštění simulace zatrhněte řádek "simulation startSimulation." a klikněte pravým tlačítkem na tento řádek. Vyberte možnost "Do it". Nyní můžete vyzkoušet aplikaci. Pro testování jsou vytvořeni dva uživatelé: Name: Tomas
Type: Admin

Name: Kate
Type: Customer

2. Pro zastavení simulace zatrhněte řádek "simulation stopSimulation." a opět klikněte pravým tlačítkem a vyberte možnost "Do it". Nyní lze zavřít modální okno.

Spuštění výchozí aplikace napsané ve Smalltalku:

1. Zatrhněte řádek "ReservationListEditor startApplication." a pravým kliknutím vyberte možnost "Do it". Opět lze využít stejné jména a typy uživatelů jako u aplikace OOPN.

2. Při změně okna Playground na GUI-ModelDriven lze prohlížet třídy definované v jazyce Smalltalk.