VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

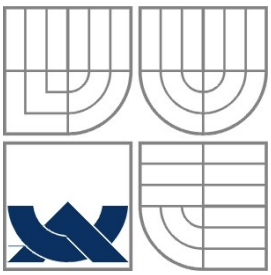
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KNIHOVNA VZORŮ PETRIHO SÍTÍ

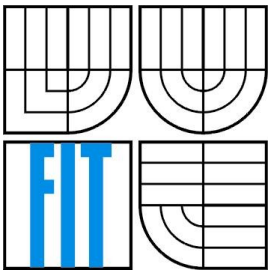
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN HANÁK
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KNIHOVNA VZORŮ PETRIHO SÍTÍ

PETRI NETS PATTERN LIBRARY

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN HANÁK
AUTHOR

VEDOUCÍ PRÁCE — ING. RADEK KOČÍ, PhD.
SUPERVISOR

BRNO 2011

Abstrakt

Tato bakalářská práce se zabývá vytvářením modelů pomocí formalismu Objektově orientovaných Petriho sítí (dále jen OOPN). Rozebírá význam a možnosti tohoto formalismu a ve své druhé části se snaží identifikovat užitečné nebo často se opakující vzory. Tyto vzory jsou inspirovány konstrukcemi klasických programovacích jazyků. Jde především o manipulaci s daty a řízení běhu programu. Nalezené vzory popisují a diskutují možnosti modelování takových konstrukcí pomocí OOPN.

Abstract

This thesis concerns creating models using formalism of Object Oriented Petri Nets (OOPN). It concludes with various usability of such formalism and in the second part of the thesis it indentifes useful and often repeated patterns of OOPN. These patterns are strongly inspired by classic programing languages, mainly by constructions concerning data manipulation and program flow control. Indentified patterns describes options for modelling these construction in OOPN.

Klíčová slova

objektově orientované Petriho sítě, PNtalk, knihovna vzorů, konstrukce v programovacím jazyku, workfow

Keywords

object oriented PetriNets, PNtalk, pattern library, constructs of programming language, workflow

Citace

Martin Hanák: Knihovna vzorů Petriho sítí, bakalářská práce, Brno, FIT VUT v Brně, 2011

Knihovna vzorů Petriho sítí

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Radka Kočího, PhD.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.
Použité ilustrace jsou vlastní výroby, vyjímky jsem uvedl u seznamu ilustrací.

……
Martin Hanák
17.5.2011

Poděkování

Chtěl bych poděkovat vedoucímu práce Radku Kočímu za ochotu při uvádění do problematiky světa PNtalk.

© Hanák, Martin 2011
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

1 Úvod

Se stoupajícím rozvojem softwarového inženýrství stoupá také význam návrhu softwaru. Jak důraz na návrh roste, rostou i tendence vytvářet návrhy čím dál komplexnější. A platí, že čím je návrh dokonalejší, tím více je možné ovlivnit implementaci, popř. dokonce zavést postupnou automatizaci generování implementace. Za předpokladu, že nástroj pro návrh zároveň může sloužit jako simulátor naskýtá se možnost si nejdřív software vyzkoušet pomocí simulace. Pokud do návrhu dostatečně popíšeme i chování, naskýtá se možnost mít kompletní aplikaci popsanou modelem. Pro to, aby tato myšlenka byla životaschopná, musí nejdřív vzniknout prostředek pro popis modelu, který bude v zásadě velmi jednoduchý, bude však skýtat možnosti popsané výše.

Jedním takovým projektem je i formalismus OOPN. OOPN je formován a vyvíjen na Fakultě informačních technologií již od roku 1993 prostřednictvím projektu *PNtalk* [1]. Je to prozatím experimentální projekt. Jeho základní myšlenka je, že vyšší modelovací nástroje mohou sloužit podobně jako klasické programovací jazyky. Má ambice stát se součástí implementace projektů a svým konceptem nabídnout odlišný přístup k tvorbě software.

1.1 Předmět bakalářské práce

Moje práce spočívala ve vytvoření knihovny vzorů OOPN. Tyto vzory jsou vetšinou takového charakteru, že popisují základní prvky klasických programovacích jazyků. K dispozici jsem měl implementaci *PNtalku* s hotovými ukázkami systémů a také rozpracovanou případovou studii „Konferenční systém“, kterou jsem dokončil. Při dokončování této studie jsem měl nejlepší příležitost vzory identifikovat. Dokončená studie je tedy součástí této práce a je přiložena. Slouží také pro demonstraci použití vzorů v praxi. Díky této práci jsem se také zevrubně seznámil s celým konceptem *PNtalku*.

1.2 Pojmy a jejich vztah

Během projektu budeme operovat s několika termíny, nástroji a jazyky, které spolu úzce souvisí a proto je vhodné je na začátku vyjmenovat, oddělit a určit mezi nimi vztahy. Jsou to:

- *Workflow* – metodika využívaná v managementu, rozděluje procesy na podprocesy z hlediska toku práce (angl. work-flow).
- *Petriho sítě* – modelovací nástroj, jeden z mnoha způsobů jak popsat workflow.
- *Objektově orientované Petriho sítě (OOPN)* – rozšíření Petriho sítí o objektový přístup.
- *PNtalk* – jazyk pro popis OOPN, v rámci implementace *PNtalku* je implementován také simulátor modelů popsaných tímto jazykem.
- *Smalltalk* – objektově orientovaný jazyk, ve kterém je implementován *PNtalk*. *PNtalk* zachovává možnost použití *Smalltalku* při tvorbě systému.

- *Squeak* – nástroj pro programování ve *Smalltalku*. Není to pouze vývojové prostředí, ale zároveň virtuální stroj, ve kterém je *Smalltalk* implementován.

1.3 Struktura práce

V první části práce se seznámíme se s prostředky využitých při tvorbě práce. Kapitola 2 pojednává o workflow, jeho využití a použití. Kapitola 3 nás pak seznamuje s formalismem *Objektově orientovaných Petriho sítí* (OOPN). Nabízí stručný úvod do tohoto formalismu a popis jeho nejvýznačnějších prvků, ale také jeho reprezentaci v jazyce *PNtalk*.
Druhá část práce, kterou tvoří kapitola 4 se bude zabývat vzory v OOPN. Ty reflektují klasické struktury procedurálního programovacího jazyka. Proto nejdřív připomenu základní vlastnosti obou jazyků.

Programovací jazyk je vysokoúrovňovou abstrakcí strojového kódu. Jako takový je zpravidla strukturován sekvenčně. V současné době však s nástupem vícejádrových procesorů vzrůstá význam vícevláknových aplikací a tím paralelního zpracování. Podmíněné zpracování se realizuje skoky ve strojovém kódu.

Modelovací jazyk oproti tomu musí reflektovat modelovanou skutečnost, popř. její vlastnosti. Model musí také reflektovat změny systému, a logiku těchto změn. Změny mohou být doprovázeny akcemi, nejčastěji sekvencí akcí. Změny systému jsou nezřídka prováděny podmíněně.

Identifikovali jsme tedy dva společné prvky těchto jazyků. Podmíněné řízení běhu nebo změny stavu a sekvenční vykonávání kódu/akcí.

2 Workflow

Workflow je pojem spjatý s managementem. Označuje se tak technika, při které se složitý proces rozdělí na menší procesy, se kterými se poté pracuje. Na systém se potom nahlíží skrze vztahy mezi těmito procesy, hledají se závislosti. Někdy se pojem workflow blíží pojmu model a někdy se dokonce používá spojení „workflow model“. Z pohledu simulačních systémů je však workflow abstraktním systémem, tj. zobecněnou realitou za účelem zkoumání jejích vlastností. Workflow samo o sobě není modelem, vyžaduje tedy reprezentaci nějakým modelovacím nástrojem nebo jazykem.

2.1 Užití v managementu

Jak už bylo řečeno, workflow slouží nejčastěji k vytvoření modelu a identifikaci jeho vlastností a vztahů. Toto použití pro IT není příliš zajímavé. IT návrhář často dostane workflow už definované a na něm je vytvořit jeho model nebo na jeho základě začít navrhovat. Nicméně tyto managerské techniky jsou velmi užitečné pokud IT specialista začne navrhovat vlastní workflow pro svoje účely, protože zpravidla musí řešit podobné problémy.

Workflow se často využívá pro identifikaci slabin procesu a poté k jejich odstranění. Nejklasičtější použití je pak následující [2]:

1. Identifikace procesů
2. Vytvoření workflow současného procesu (as-is process)
3. Návrh nového procesu (to-be process) pomocí workflow
4. Aplikace nového návrhu

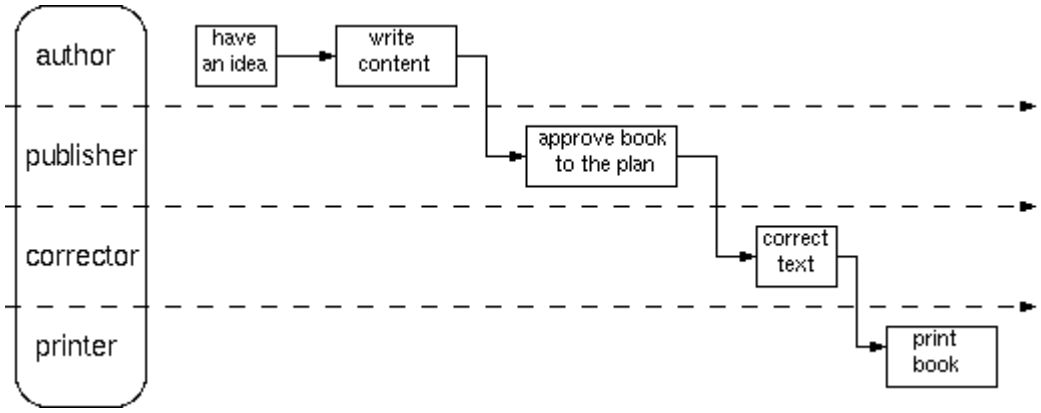
Tento postup se často nazývá reengineering a slouží k čistě managerským potřebám, jako je zefektivnění výroby, odstranění slabých článků procesu apod.

Nicméně následující rady je dobré vzít v potaz. Jde především o výčet prvků, které každé workflow obsahuje a pokud se vše hlídá, pak se zvyšuje šance, že navržené workflow bude reflektovat skutečnost. Nejde o ucelené rozdělení, spíše ukázku význačných prvků, na které bývá kladen důraz při workflow managementu.

- Lidé – role: činnost je vždy vykonávána nebo řízena lidskými zdroji. Obecné pravidlo managementu říkající, že pokud role nejsou správně definovány, dojde dříve nebo později ke konfliktům a nedorozuměním
- Zdroje: vykonávání může být velmi často podmíněno dostupností zdrojů pro tuto činnost. Jako zdroj může vystupovat leccos, např. lidské zdroje, stroje nutné pro činnost, materiál, apod.
- Pravidla: chování systému může být podmíněno různými okolnostmi a pravidly nastavenými tvůrcem nebo správcem systému. Typicky „výdaje nad 25000,- musí schválit vedoucí oddělení“.
- Metrika posuzování: bývá kritickým krokem při reengineeringu. Zpravidla jde o to stanovit kritéria pro posuzování efektivity tak, aby tyto kritéria byla relevantní.

2.1.1 Příklad – *swimlane diagram*

Nejtypičtějším nástrojem pro modelování workflow pro účely managementu je tzv. *swimlane diagram*. Do češtiny by se přeložil jako „digram plaveckých drah“. To proto, že hlavní roli v něm hrají řádky, přičemž každý řádek značí jinou roli v systému. Samotný proces je pak reprezentován rámečky, které jsou umístěny na řádcích. Rámečky mají definovanou posloupnost a také mají přiřazeny popisy definující konkrétní činnost. Na pomyslné ose x buď může nebo nemusí být reprezentace času. Rámečky by pak byly dlouhé podle toho, jakou dobu se činnost vykonává. Použití času je třeba zvážit, někdy může být kritické sledovat na modelu předpokládaný čas, jindy může být hlavním účelem diagram samotná posloupnost činností. Ukázka bez modelování času je na obrázku 1.



Ilustrace 1: Ukázka workflow - swimlane diagram

Obliba swimlane diagramů spočívá v jejich jednoduchosti, a to především z hlediska čitelnosti. Řádková reprezentace bývá zpravidla dobře přijímána a skýtá dobrý přehled o procesu jako celku.

Ještě stojí za zmínku trochu přepracovaná varianta swimlane diagramu nazývaná *Ganttův diagram*. *Ganttův diagram* se vyznačuje především tím, že řádek nereprezentuje roli, ve které se činnost provádí, ale činnost samotnou. Hlavním významem této úpravy je zpravidla možnost strukturování činností do větších významových celků. *Ganttův diagram* je hlavní součástí většiny softwaru pro řízení projektů, např. Microsoft Project.

2.2 Význam pro IT

Pro IT má workflow samozřejmě také vzrůstající význam. Tak, jak jsem popsal workflow v předchozí kapitole, tedy z hlediska rolí, zdrojů apod., se využívá především ve fázi analýzy. S využitím swimlane diagramu může často sloužit jako komunikační prostředek pro analytika a zadavatele nějakého informačního systému. Avšak pro tvorbu software řádková reprezentace není nejvhodnější. Důvodem je především fakt, že při tvorbě softwaru se na role (nebo spíše oprávnění) neklade takový důraz.

Z hlediska návrhu softwaru je podstatnější logická struktura, akce které uživatel provádí, podmínky proveditelnosti těchto akcí, výsledky provedení apod. Pro tyto účely jsou vhodné jiné

nástroje, které by se logikou neomezovaly na posloupnost řádků. O nástrojích vhodných pro modelování workflow pro IT však pojednám více v následující kapitole

Co je daleko zajímavější, je možnost zapojit workflow do řízení logiky aplikace. Ve své praxi jsem se s tímto využitím již setkal. Představme si, že máme v systému objekt, který definuje význačný stav, ale co je důležitější, změna tohoto stavu má za následek vykonání nějaké akce, nebo dokonce posloupnost akcí. Změny těchto stavů mají nějakou logiku, která může být vnější. Typickým příkladem může být datum, jako v následujícím příkladu z mé praxe.

2.2.1 Příklad z praxe

Příklad z mé praxe se týkal evidence smluv. Smlouva měla jistou platnost od-do. Ve většině případů však firma tyto smlouvy prodlužovala. Vznikl tedy požadavek na to, aby se vedoucímu nějakou dobu před ukončením platnosti odeslal varovný email o tom, že smlouvě končí platnost. Toto se mělo dít naprosto automaticky, aniž by aplikaci evidence těchto aplikací kdokoli spouštěl.

Takže se použilo workflow založené na čase. Pro tyto účely jsme měli vytvořené třídy sloužící k definici workflow. Ve workflow byly nadefinovány stavy a logika přechodů mezi nimi. Vyjadřovací silou to tedy byl konečný automat (více v kapitole 2.3). Nadefinovali jsme akce tak, aby se při změně stavu odeslal email o této změně. Stavy smlouvy byly nadefinovány následovně: nová, platná, končící, ukončená. Možnost přechodu mezi stavy byla v pořadí tak, jak jsem je uvedl a tyto změny byly řízeny časem (pro kontrolu jsme použili robota, který toto pravidelně hlídal). Navíc existovala možnost přechodu ze stavu končící do platná. Tento přechod však byl podmíněn akcí uživatele na prodloužení smlouvy.

Toto řešení se ukázalo být velmi elegantním především z následujících důvodů:

- workflow se definuje jednodušeji, než implementace KA přímo v kódu
- všechny akce jsou v definici workflow, nejsou po celém zdrojovém kódu
- velmi dobrá znovupoužitelnost – jednu definici workflow můžeme použít vícekrát
- možnost snadného redesignu definice workflow pro adaptaci na nové podmínky/jiný objekt

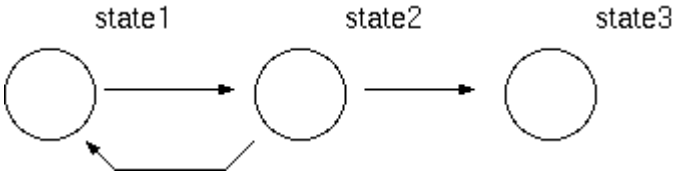
2.3 Možnosti jak modelovat workflow

Jak jsem naznačil v předchozím příkladě, způsob reprezentace workflow může mít různou vyjadřovací sílu. V příkladu byl použit konečný automat a na řešený problém to stačilo. Avšak v mnoha jiných případech toto nebude stačit. Proto jsem uznal vhodné uvést základní rozdíly v modelovacích systémech. Uvedu 3 příklady: Konečný automat, Petriho sítě, Objektově orientované Petriho sítě.

Konečný automat

Základem konečného automatu je stav. Automat sestává z množiny stavů, jichž může nabývat. Automat má vždy jen jeden aktuální stav. Aktuální stav se změní tím, že automat přejde do jiného stavu na základě splnění podmínek přechodu. Častý požadavek na konečný automat je, že má být

deterministický. Determinismus u něj znamená, že z každého stavu, který není konečný, musí být v danou chvíli maximálně jeden přechod uskutečnitelný.



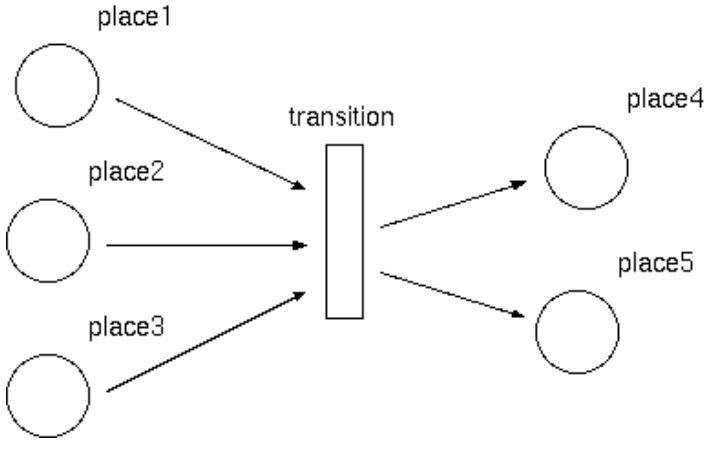
Ilustrace 2: Příklad konečného automatu

Z naposledy jmenovaného plyne asi největší nevýhoda konečného automatu, tj. nemožnost modelovat paralelní procesy. Další nevýhodou je, že pokud budou vlastnosti systému nějak strukturované a my budeme chtít sledovat jen nějakou konkrétní vlastnost, musíme pro ni definovat všechny stavy, které tuto vlastnost splňují.

Petriho sítě

Pokud chceme silnější nástroj, zvolíme Petriho sítě. Petriho sítě sestávají z míst a přechodů, přičemž místo však nereprezentuje stav ale pouze vlastnost systému v knize *Petriho sítě* [3] se dokonce hovoří o tzv. parciálním stavu. Označení místa se provádí graficky tečkou, ta se nazývá *token*. Přechod je pak proveditelný, pokud systém splňuje výčet parciálních stavu, tzv. *Preset*. Přechod však také definuje *Postset*, což je množina parciálních stavů, jež systém nabude po provedení přechodu. Vazby *postsetu* a *presetu* k místům se nazývají hrany

Fakt, že v jednu chvíli může být označeno více míst, nám tedy dává možnost modelovat paralelismus, což je asi největší přínos Petriho sítí. Avšak z hlediska čitelnosti modelu je dekompozice stavu na množinu vlastností, tedy míst, také nespornou výhodou. Na obrázku 3 je ukázka přechodu Petriho sítě, adaptována z [3].



Ilustrace 3: Přechod v Petriho síti

Pro Petriho sítě existuje celá řada rozšíření , uvedu nejklasičtější:

- *prioritní přechody* – pokud jsou v jednu chvíli proveditelné dva přechody současně, vznikl by nedeterminismus, řeší se prioritním přechodem
- *přechod s pravděpodobností* – pokud je proveditelných více přechodů, lze jim stanovit pravděpodobnost, s jakou se provedou. Pokud povolíme nedeterministický stav, je to v podstatě ekvivalent pravděpodobných přechodů se stejnou pravděpodobností.
- *váha hrany* – dovoluje aby pro proveditelnost musel přechod odčerpat více než jeden token a také mohl po provedení distribuovat více než jeden token.
- *kapacita míst* – dovoluje stanovit maximální počet tokenů, které se v daném místě mohou vyskytovat

Jedno z rozšíření také bývá vnesení objektově orientovaného přístupu, o tom však pojednává celá další kapitola.

3 Objektově orientované Petriho sítě

Existuje snaha v klasickém programování, aby struktury v programech co nejvíce odrážely skutečnost, což vedlo k objektově orientovaném přístupu k programování. Využití této abstrakce má význam především pro lepší strukturování kódu, důsledkem čehož se zlepší čitelnost a přehlednost nebo znovupoužitelnost programů.

Modelování systémů Petriho sítěmi je sice elegantní, avšak při nárůstu složitosti systému se síť stává příliš složitá, nepřehledná a prakticky nečitelná. Při vhodném zapouzdření však lze dosáhnout značné abstrakce a tím i zjednodušení. To je hlavní důvod snahy o zavedení objektového přístupu i do Petriho sítí. Jedním takto vytvořeným formalizmem je OOPN, definovaný v disertační práci *Modelování objektů Petriho sítěmi* [4].

OOPN je matematický formalismus a proto vyžaduje nějakou reprezentaci, a to buď příslušným jazykem (*PNtalk*, definován také v [4]), nebo grafickou reprezentací. Pro účely popisu vzorů v kapitole 4 poslouží grafický popis. K projektu je však přiložena implementace případové studie (kapitola 4.1), a ta je pochopitelně psána pomocí jazyka *PNtalk*. Proto představím obě reprezentace konstrukcí OOPN.

3.1 Vlastnosti OOPN

Nejdůležitějším atributem OOPN je, že samotná Petriho síť je objekt. Vytvořením objektu nějaké třídy (zavoláním konstruktoru new) se automaticky aktivuje jeho *objektová síť*, tj. Petriho síť popisující hlavní logiku objektu. Kromě ní může mít třída metody a každá metoda má svou vlastní Petriho síť, tzv. *síť metody*. Ta vzniká zavoláním metody a jejím skončením zase zaniká.

Tokeny nejsou už pouze značky, ale reference na objekty. Objekt může být buď datové primitivum (číslo, symbol, řetězec, apod.) nebo objekt Petriho sítě. *Token* může reprezentovat n-tici objektů. Zůstává zachována sémantika místa, tedy že přítomnost *tokenu* v místě značí parciální stav.

Přechody mezi místy jsou proveditelné, pokud platí podmínky *preset*, *postset* a nově i *stráž přechodu*. V přechodu lze definovat akci, tj. posloupnost příkazů a volání, které se při přechodu zavolají. V přechodu lze manipulovat s proměnnými, které do přechodu vstupují pomocí hran. Lokální proměnné není třeba deklarovat, OOPN je volně typovaný a jmenný prostor se váže dynamicky.

Hrany jednak vyjadřují kardinalitu objektu, s nímž se manipuluje a také slouží k navázání objektu na proměnnou tím, že nese název proměnné a tento název pak lze v přechodu používat. Hrany vyjadřují podmínky *preset* a *postset*. V *PNtalku* jsou vyjádřeny pomocí konstrukcí `precond` (objekty vstupující do přechodu), `postcond` (objekty vystupující) a `cond` (objekt se naváže na proměnnou přechodu, ale v místě zůstává).

V sítích lze také namodelovat nedeterministické chování, tj. vnesení náhody do vykonávání programu. Existují dva způsoby jak toho lze dosáhnout. Prvním je výběr *tokenu* z místa, kde se nachází více *tokenů*, přesněji existuje více možných navázání proměnných. Druhou situací je, že

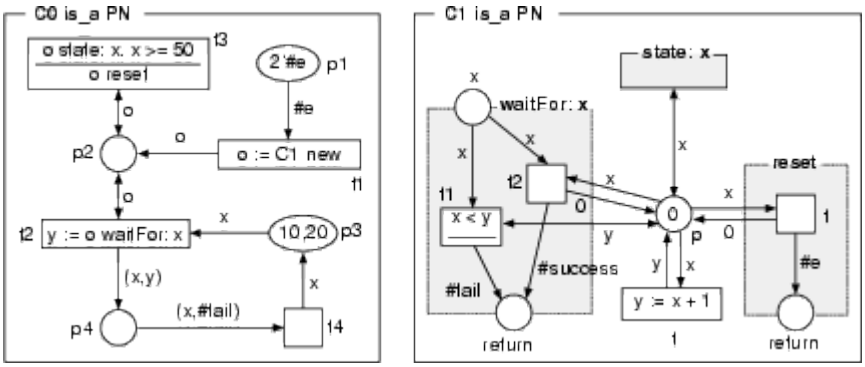
v jednu chvíli je z jednoho místa proveditelných více přechodů. Simulátor si pak musí jeden vybrat a tato činnost se děje náhodně. Nedeterminismu se však doporučuje vyhýbat. Ve většině případů je nežádoucí a jeho užití pro modelování pseudonáhodného procesu je stejně závislé na implementaci simulačního nástroje.

Z hlediska omezení jsou v OOPN nejdůležitější tyto vlastnosti:

- Hranám nelze definovat prioritu ani pravděpodobnost
- Místům nelze definovat omezení kapacity

3.2 Důležité prvky a konstrukce v OOPN

Nejdůležitější prvky demonstrujeme na příkladu použití, k čemuž poslouží Ilustrace 4. Nebudeme se však zabývat modelem, jež reprezentuje. V následujících podkapitolách nás zajímají nejdůležitější konstrukce, které identifikujeme, popíšeme, popř. interpretujeme jejich sémantiku. Na příkladu si také můžeme všimnout grafické reprezentace, stejná reprezentace je použita také u vzorů v kapitole 4.



Ilustrace 4: Příklad konstrukcí OOPN

3.2.1 Stráž (guard)

Stráž je součást přechodu a odděluje se od akce přechodu vodorovnou čarou, přičemž stráž je nad čarou. Pomocí stráže lze realizovat testy na stav objektů. Toho lze dosáhnout voláním *synchronních portů* a *inhibitorů* a také vyhodnocovacích operací. Pokud lze provést nějaké navázání proměnných a zároveň všechny vyhodnocovací výrazy jsou úspěšné, pak lze přechod provést. Jinak se objekt vrátí do stavu před zavoláním portu.

Příklad: v síti `C0` v přechodu `t3` je stráž: „`o state: x. x >= 50.`“. Její sémantika je: pokus se najít v místě `p2` objekt `o`, jehož *synchronní port* `state: x` vrátí objekt, jehož hodnota je větší než 50.

3.2.2 Synchronní port

Zjednodušeně, *synchronní port* (nebo zkráceně jen *port*) slouží k propojení s jiným objektem. Může definovat `precond, cond, postcond` a `guard.` Od přechodu se liší tím, že nedefinuje akci

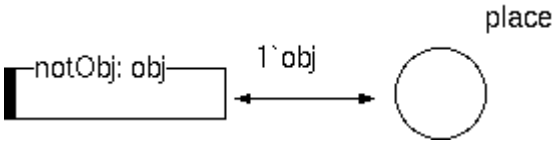
a především jeho provedení je podmíněno voláním z jiného objektu. Port je tak napůl metodou a napůl přechodem. Pokud je portu argumentem předána již navázaná proměnná, port je proveditelný pouze když je schopen nalézt navázání, tj. funguje jako test na přítomnost nějakého objektu v síti. Pokud je proměnná v argumentu nenavázaná, port provede některé možné navázání, pak slouží jako nástroj pro získání reference. V případě, že je možných více navázání, vybere se jedno náhodně, což je nedeterministické chování.

Příklad: síť `C1` definuje port `state: x`, který exponuje objekt z místa `p` a opět ho vrací. Tento port je volán ze stráže přechodu `t3`, třídy `C0` pomocí nenavázané proměnné x. Port tedy naváže proměnnou `x` na některý objekt z místa `p`.

3.2.3 Inhibitor

Tato struktura byla do OOPN dodefinována v [5]. *Inhibitor* je negativní port. Opačná sémantika takového portu spočívá v tom, že port je proveditelný, pouze pokud nelze nalézt příslušné navázání. Ostatní vlastnosti jsou shodné se synchronním portem. Význam inhibitoru je, že pokud máme dvojici inhibitor a synchronní port se stejně definovanými podmínkami, tvoří spolu vzájemně komplementární rozhodovací dvojici. Takovýchto dvojic se pak užívá v kapitole 4.2.2. Vzhledem k tomuto vztahu používáme často označení *port* bez specifikace sémantiky a můžeme tím myslet pozitivní i negativní port (v situacích, kdy na tom nezáleží).

Grafická reprezentace je na obrázku 5.



Ilustrace 5: Příklad inhibitoru

3.2.4 Metody

Metoda je jedna z nejtypičtějších konstrukcí v objektově orientovaném přístupu. Také třídy v OOPN mají definované metody. Každá metoda má definovanou svoji síť metody. Instance sítě je vázána na jedno volání metody, tzn. při každém volání vznikne nová síť a po skončení metody zase zaniká.

Každá metoda musí ve své síti definovat návratové místo `return`. Přítomnost tokenu v tomto místě indikuje konec vykonávání metody. Pokud je v místě `return` více tokenů, jako návratová hodnota slouží pouze jeden token. Ten by se vybral náhodně a když by objekty nebyly stejné, vzniklo by nám nedeterministické chování.

Metoda může definovat argumenty, ale také nemusí. Volání metody je možné z jakéhokoli přechodu programu a z metody je možno přistoupit ke všem místům objektu. V *síti metody* však nelze definovat porty.

Příklad: třída `C1` definuje dvě metody. Metodu `waitFor: x` s argumentem x a metodu `reset` bez argumentu.

3.3 PNtalk

K vytváření modelů pomocí OOPN nám slouží jazyk pro jeho zápis *PNtalk*. Příklad použití pak nalezneme v příloze . Následuje přehled klíčových klíčových slov *PNtalku* a jejich zásady použití.

- `is_a` – používá se v definici třídy a specifikuje rodičovskou třídu. Základní třída pro popis OOPN je PN. Syntax:
  `<NewClass> is_a <ParentClass>`
- `place` – místo v Petriho síti. Místu lze definovat počáteční značení. Syntax:
  `place startPlace(1`#e)`
- `trans` – klíčové slovo pro přechod. Může obsahovat následující konstrukce (v daném pořadí): `precond`, `cond`, `guard`, `action`, `postcond`. Každá konstrukce je uvozena maximálně jednou. Pokud obsahuje více položek, píší se za sebe. Všechny konstrukce jsou nepovinné, přechod však musí mít smysl.
- `precond` – specifikuje hrany přechodu pro navázání objektu v určitém místě na proměnnou. Při provedení přechodu se objekt na místo nevrátí. Např.
  `precond place1(1`obj)`
- `cond` – syntakticky totožná konstrukce jako `precond`, pouze po provedení přechodu se objekt na místo vrátí.
- `postcond` – specifikuje místa, kam simulátor vloží objekty po provedení přechodu. Syntax je stejná jako u `precond` a `cond`.
- `guard` – definuje stráž přechodu. Syntax:
  `guard{volání portů a výrazy.}`
- `action` – definuje akci spojenu s provedením přechodu. Syntax:
  `action{sekvence příkazů.}`

Následuje ukázka z částí implementace Ilustrace 4. Slouží jako demontrace konstrukcí, není to kompletní zápis systému.

```
C0 is_a PN
place p1(2`#e)
place p2()
trans t1
     precond p1(1`#e)
     action {o:= C1 new.}
     postcond p2(1`o)
trans t3
     cond p2(1`o)
     guard{o state: x. X >= 50.}
     action{o reset}
```

```
C1 is_a PN
place p(0)
sync state: x
     cond p(1`x)
method reset
     place return()
     trans t
          precond p(1`x)
          postcond p(0)
return(1`#e)
```

4 Knihovna vzorů

V následující části konečně přistoupíme k samotným konstrukcím v OOPN. Některé byly už implementovány a publikovány[6,7], avšak vždy pro konkrétní příklad. Některé pokročilejší vzory jsem identifikovat při potřebě modelovat takovou situaci pro Konferenční systém. Všechny vzory jsou popsány pokud možno co nejobecněji se snahou o diskuzi jejich použití, popř. odlišností při různých situacích.

Problematika OOPN řešená v této práci je demonstrována na případové studii Konferenčního systému. Tento model byl navržen pro potřeby demonstrace modelovacích technik v rámci *PNtalku*. Příklad Konferenčního systému byl částečně popsán v [6,7].

4.1 Popis příkladu: Konferenční sytém

Hlavním třídou je `Conference`, která udržuje stav systému, udržuje kolekci členů konference a také kolekci prací. Má synchronní port pro získání stavu `phase:`. Stavy konference jsou open, review, final a close. Ve fázi open mohou autoři do systému vkládat své práce a upravovat ty, které už vložili. Fáze review umožňuje administrátorovi přidělit recenzentovi práci k recenzování a recenzent může k práci autora připojit svoji recenzi. Následuje fáze final, kdy si autor může přečíst recenzi na svou práci a poté odevzdat finální produkt. Ve fázi close je konference zavřená a nikdo nemůže dělat nic.

Stav lze měnit dvěma způsoby. Buď jej nastavíme ručně metodou `setState:` nebo můžeme spustit předefinovaný cyklus metodou start.

Každý člen konference je reprezentován objektem třídy `Member`. Ta ukládá jeho jméno, které slouží zároveň jako klíč k identifikaci. Členové mohou vystupovat v různých rolích a podle role jsou oprávněni k různým akcím podle stavu konference. Role se uchovává v místě type a je k dispozici celá řada portů na testování role. Člen si taky uchovává své práce v místě `papers`.

Uživatel systému se nejdříve musí přihlásit pomocí instance třídy `RegistrationNet`. Pokud je síť připravena k přihlášení, je proveditelný port `login:as:`, kde prvním argumentem je přihlašovací jméno a druhým je role ve které chce uživatel vystupovat. V případě úspěchu bude registrační síť ve stavu přihlášení a pomocí portu user můžeme získat síť pro chování uživatele.

Pro každou roli existuje jedna síť chování (`AuthorNet, AdminNet, ReviewerNet`). Každá síť má uložen odkaz na konferenci a na člena, jehož chování popisuje. Podle typu sítě jsou pak k dispozici metody a porty umožňující dané roli provádět příslušné akce.

Třída `Paper` reprezentuje práci autora. Má místo `document`, kde je uložen obsah práce, místo `authors`, kde mohou být uloženi autoři práce, a konečně místo `review`, kam ukládáme recenzi práce. Pro recenzi je navržena speciální třída `Review`, která může ukládat obsah recenze a recenzenta.

Konečně je k dispozici testovací síť `TestingNet`. Tato třída simuluje život systému tím, že provádí operace nad objekty a tím simuluje uživatelské akce.

4.2 Obecné prerekvizity

Vzhledem k vlastnostem OOPN tak, jak jsou popsány v kapitole 3.1, je třeba mít při tvorbě systému na paměti několik zásad. Jedná se spíše o rady týkající se přístupu k programování, pojmenování, práce s místy apod. Některé se pak týkají toho, jak pracovat s vlastnostmi tak, abychom jich mohli při budoucí práci s výhodou využít.

4.2.1 Vlastnosti × kolekce

Tato část vyplývá z faktu, že místu nelze nastavit maximální kapacitu. Místo se totiž dá používat buď jako vlastnost, tedy jeho kapacita je jediný objekt, nebo jako kolekce. Tento rozdíl nijak nelze hlídat, takže záleží na tvůrci, jak s místem nakládá.

Slině však doporučuji mezi těmito přístupy rozlišovat. Hlavním důvodem je totiž fakt, že pokud vyberu jeden prvek z kolekce tak tato operace se provede, systému to nevadí, avšak tato operace je čistě náhodná. Je třeba dát velký pozor jestli je to zamýšlené chování, nebo zda nejde o nežádoucí nedeterminismus.

Dále je třeba mít na paměti i to, aby byl systém čitelný. Pokud ze systému nebude jasné, jak se dané místo používá, značně se tím zhoršuje schopnost systému porozumět.

V neposlední řadě objekty, které shromažďujeme na místě jako kolekci by měly implementovat nástroje pro vlastní identifikaci v rámci této kolekce. Pro identifikaci je třeba zvolit vhodný klíč, podle kterého jsme pak schopni objekt znovu vyhledat.

Lze však však podniknout kroky, kterými lze těmto nepříjemným stavům předcházet nebo je zcela eliminovat:

1. Pojmenování míst. Zvolte si systém pojmenování a striktně ho dodržujte. Lze využít jednotné a množné číslo, např. „member – members“. Záměrně jsem užil angličtiny, protože je vidět, jak ten rozdíl může být malý. Proto doporučuji spíše lépe viditelné předpony nebo přípony, vhodnější je tedy „col_member“ nebo „memberCol“ apod.
2. Při manipulaci s daty využívat metody, které jsou určeny pro daný typ místa. V kapitole 4.3 jsou pro manipulaci vzory a vždy se vyskytují ve dvojici podle užití. Přehled vzorů je v následující tabulce.

	Přidat	**Vybrat**	**Odebrat**
vlastnost	Setter	Getter	Clear
kolekce	Add	Find/GetAll	Remove/ClearAll

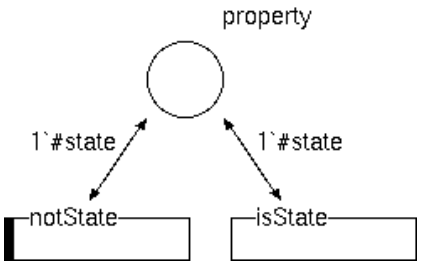
3. Pro kolekci implementovat speciální objekt. Toto opatření je poněkud náročnější a má charakter spíše speciální knihovny. Ale spočívá v tom, obalit kolekci a všechny metody pro práci s ní do jednoho objektu. Kolekci pak získáme jako referenci na takový objekt, dostupnou z místa používaného jako vlastnost.

4.2.2 Požadavky na objekt

Je vhodné, aby objekt zveřejňoval své důležité vlastnosti pomocí portů. Mnoho z nich se také vyžaduje jako prerekvizity pro některé vzory, tzn. objekt musí tyto porty implementovat, aby šel daný vzor použít. Tyto požadavky vycházejí ze způsobu, jakým OOPN vyhodnocuje proveditelnost přechodu a taky díky neexistenci *prioritních přechodů.* Jedná se zpravidla o vyhodnocovací dvojice, protože se většinou vyžaduje komplementarita testu.

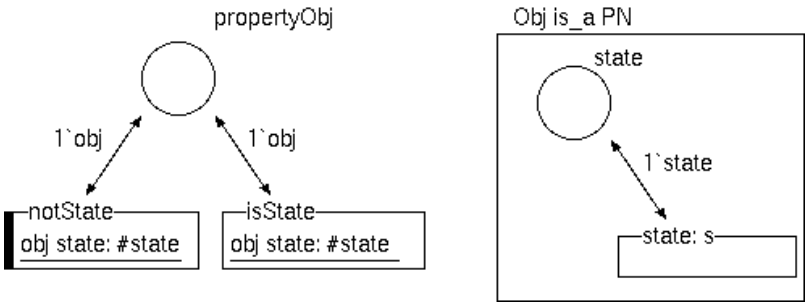
Test stavu (*State test*)

Pokud na nějakém místě uchováváme nějaký stav, nejčastěji reprezentovaný symbolem, např. `#state`, pak je vhodné zajistit kontrolu, zda objekt je v takovém stavu, zároveň ale můžeme požadovat i zjištění, jestli tato podmínka neplatí. Typické užití je pak ve vzoru *If-Else.*



Ilustrace 6: State test

Realizace tohoto testu, nebo lépe řečeno dvojice testů, je zajištěna synchronním portem a inhibitorem se stejně definovaným tělem a hranami. Na příkladu obrázku 6 jsou těla prázdná, protože k navázání dojde pomocí hrany.

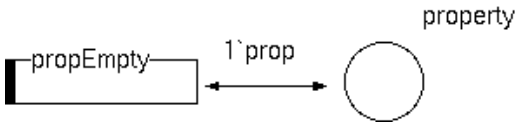


Ilustrace 7: State test jiného objektu

U tohoto příkladu demonstruji ještě případ, kdy se zjišťování stavu týká jiného objektu. Toto zanoření je dobře patrné z obrázku 7. V místě `propertyObj` je objekt jiné třídy (třídy *Obj*), který uchovává požadovaný stav. Ve zjišťovací dvojici získáme odkaz na tento objekt a teprve přes jeho synchronní port `state: s` se pokusíme získat stav.

Test prázdnosti (*Empty test*)

Tento test se často vyskytuje nepárově a to z toho důvodu, že svůj komplement zpravidla nepotřebuje. Jeho užití je zpravidla v situaci If-else pokud chceme s nějakým objektem něco provést. Pokud místo není prázdné, pak jej hranou jednoduše vytáhneme. Pokud však je prázdné, přechod by se stal blokujícím a čekal by na objekt. Proto implementujeme *Empty test* který se pak uplatní ve větvi *else.* Užití např. ve vzoru *Getter.*

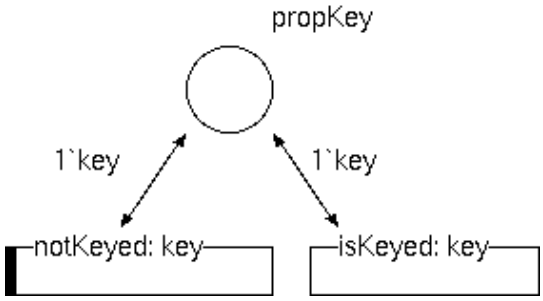


Ilustrace 8: Empty test

Je třeba dodat, že není vyloučeno, že pokud to situace bude vyžadovat, lze implementovat i *notEmpty test.* Líší se pouze tím, že není použito inhibitoru, ale synchronního portu, a tak ho tu explicitně neilustruji.

Test identifikace objektu (*Is test*)

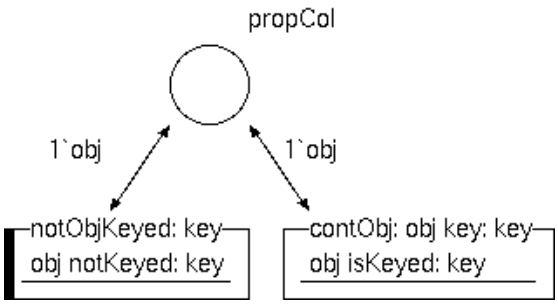
Nejčastěji se tento test využívá při identifikaci objektu v rámci kolekce. Objekt by měl obsahovat nějaký klíč, kterým ho lze jednoznačně identifikovat v rámci kolekce. Tím se myslí vybrat jednu vlastnost a tu používat jako primární klíč. Pak je třeba zajistit, aby tento klíč byl v rámci kolekce unikátní. *Is test* je typický tím, že dvojice synchronní port a inhibitor mají jeden argument a to klíč. Při použití se předpokládá, že klíč nebude nenavázaná proměnná.



Ilustrace 9: Is test

Test přítomnosti objektu podle klíče (*Contains test*)

Tento test využíváme při kontrole, zda je se v místě obsahujícím kolekci vyskytuje objekt s určitým klíčem. Tento test má sám prerekvizitu, a sice objekty v kolekci musí implementovat *Is test*, aby je bylo možné porovnat na shodu. Synchronní port `contObj: key:` má dva parametry. Jako první parametr se předává nenavázaná proměnná. Pokud chceme jen zjištění, zda je hledaný objekt přítomen, pak tuto proměnnou hned zahodíme. Lze ji však použít i k získání reference a s objektem dále pracovat.

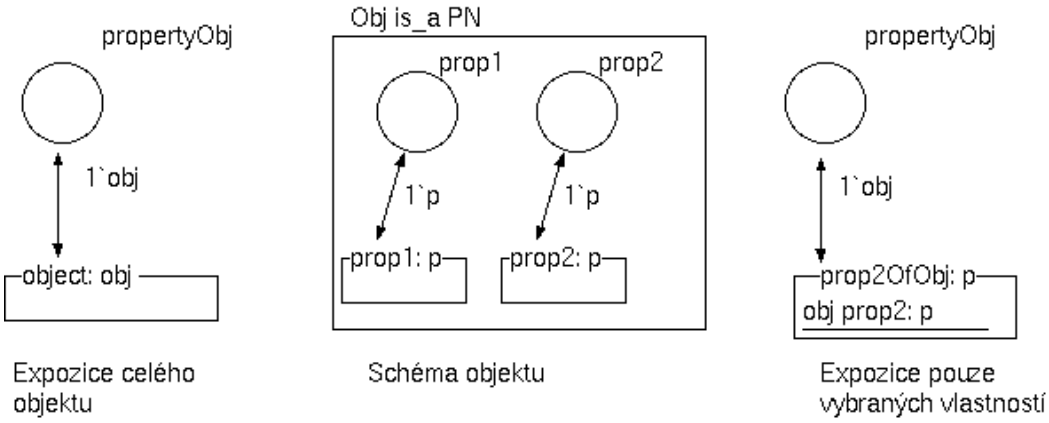


Ilustrace 10: Contains test

4.2.3 Skrývání a expozice portů vnořených objektů

To, co označuji jako skrývání, by se dalo považovat už za vzor. Týká se expozice vlastností vnořených objektů. Na obrázku 11 máme definovaný objekt třídy Obj, který má dvě vlastnosti `prop1` a `prop2`. Tento objekt uchováváme v místě `propertyObj`. Když budu chtít exponovat celý objekt, tak zavedu port, který mi naváže celý objekt na proměnnou, jak je vidět na obrázku vlevo (port `object: o`). V tomto případě budu mít referenci na objekt mohu libovolně přistupovat ke všem jeho portům a metodám.

Může však nastat situace, kdy například `prop1` obsahuje citlivé informace, my nechceme aby k nim šlo přistoupit a zajímá nás pouze `prop2`. V tomto případě zavedu port, který bude až ve stráži volat požadovaný port uchovávaného objektu, jak je patrné napravo Ilustrace 11 (port `prop2OfObj: p`).



Ilustrace 11: Skrývání a expozice vlastností

Obdobným způsobem lze exponovat i metody objektu `Obj`. Tento postup poněkud připomíná techniku *private* (skryté) a *public* (veřejné) známou z klasických OOP jazyků. Není to však úplně přesné, protože *private* a *public* se nenastavuje přímo v definici třídy `Obj`, ale až při jeho použití. Přísně vzato každé místo je přirozeně *private* a přítomnost portu s odkazem na jeho obsah je ekvivalentem direktivy *public*.

4.3 Manipulace s daty

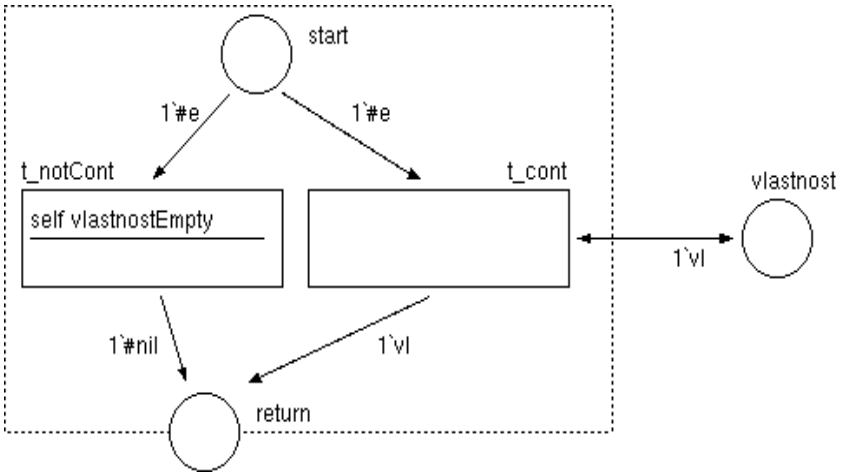
Druhá oblast vzorů pojednává o manipulaci s daty. V případě, že se na vzory díváme objektově orientovaně, pak je třeba zajistit, aby objekt nabízel možnost získání vlastností, které reprezentuje a naopak nastavení těchto vlastností. Typicky se metody pro tyto operace nazývají getter a setter. Protože v OOPN nelze místu stanovit kapacitu, je třeba dbát na to, aby bylo jasné, kdy se místo používá jako vlastnost a má tedy kapacitu=1, a kdy je místo používáno pro kolekci objektů. Práce s daty se v těchto případech pak poněkud liší.

4.3.1 Získání objektu – *Getter*

Na začátek si dovolím poznámku k využití těchto metod. Vzory jsou koncipovány jako čisté gettery, tedy mají získat referenci na objekt. Těla přechodů jsou proto prázdná. Nic ovšem nebrání tomu přepsat metody tak, aby mohly objekt měnit a upravovat. Jen připomenu zásadu, že getter by měl vždy objekt, se kterým pracuje opět vrátit na místo, odkud jej vzal.

Getter vlastnosti

Pokud je getter implementován pro vlastnost, je použití jednodušší. Jedinou prerekvizitou je fakt, že objekt musí implementovat indikátor stavu, kdy je vlastnost prázdná, tedy *Empty test*.



Ilustrace 12: Getter vlastnosti

Typický getter je na obrázku 12. Metoda nemá žádný vstup, výstupem je buď odkaz na objekt v případě úspěchu, nebo symbol indikující neúspěch(v našem případě `#nil`). Proveditelnost přechodu pro neúspěch *t_notCont* je realizována inhibitorem indikujícím stav prázdného místa *vlastnost*. Proveditelnost přechodu *t_cont* je zprostředkována přítomností nějakého objektu v místě *vlastnost.* Je důležité, aby přechod na toto místo odebraný objekt zase vrátil nebo ideálně použít konstrukci *PNtalku* `cond`, která objekt v místě ponechává, pouze naváže referenci.

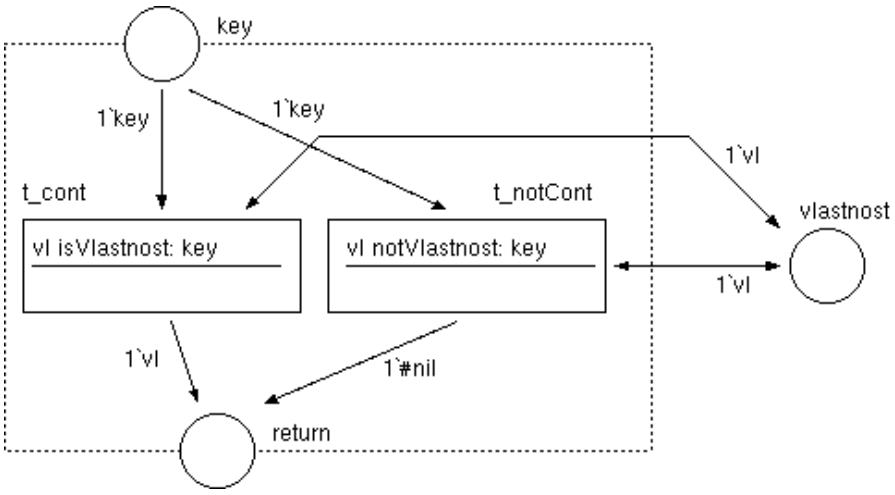
Getter a kolekce

Pro práci s kolekcí můžeme požadovat dva druhy přístupu. První je získat všechny prvky, které místo obsahuje. Tato operace však obsahuje průchod všemi prvky, který je popsán v kapitole 4.5.2, proto je tato operace popsána v diskuzi této kapitoly.

Druhou možností je nalezení určitého prvku v kolekci. Tato metoda by se mohla jmenovat spíše Find, avšak zařadil jsem ji ke Getteru, protože je blízká svoji logikou.

Metoda je na první pohled podobná metodě předchozí, jak je vidět z obrázku 13. Požadovaný prvek je v kolekci, proto je třeba ho vyhledat pomocí určitého klíče, který je parametrem této metody. Je tedy nezbytné, aby prvek implementoval *Is test* pro tento klíč (v našem případě *isVlastnost:key* a *notVlastnost:key*). Kolekce také musí zajišťovat, že každý klíč je v kolekci unikátní. Pokud by tomu tak nebylo, je výběr nedeterministický.

Stráž musí být proto realizována v obou přechodech.



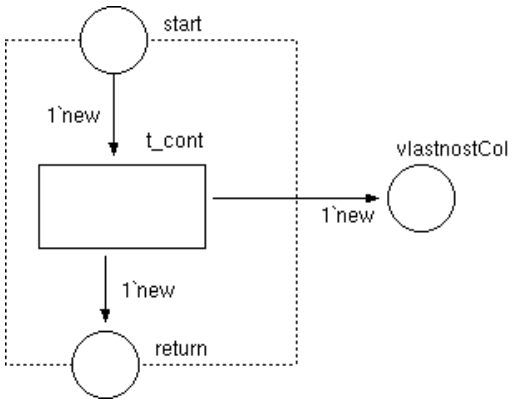
Ilustrace 13: Getter v roli Find

4.3.2 Přidání – *Setter* a *Add*

Další funkcí, kterou by měl objekt poskytovat je možnost na nějaké místo přidat jiný objekt nebo nastavit nějakou vlastnost, jež bývá také reprezentována objektem.

Add

Následující metoda je implementována pro kolekci, tedy případ, kdy v místě uchováváme více než jeden objekt. Její činnost je velice prostá, jak naznačuje obrázek 14.

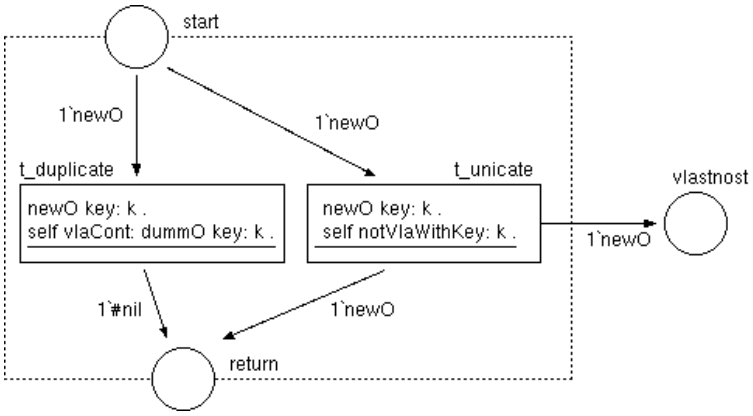


Ilustrace 14: Metoda Add

Diskuze

U této metody je vhodné použít kontrolu na unikátnost. Týká se to koncepce modelu a použití místa jako kolekce, protože z kolekce většinou budeme chtít vyhledat podle klíče, viz. Kapitola 4.2.1.

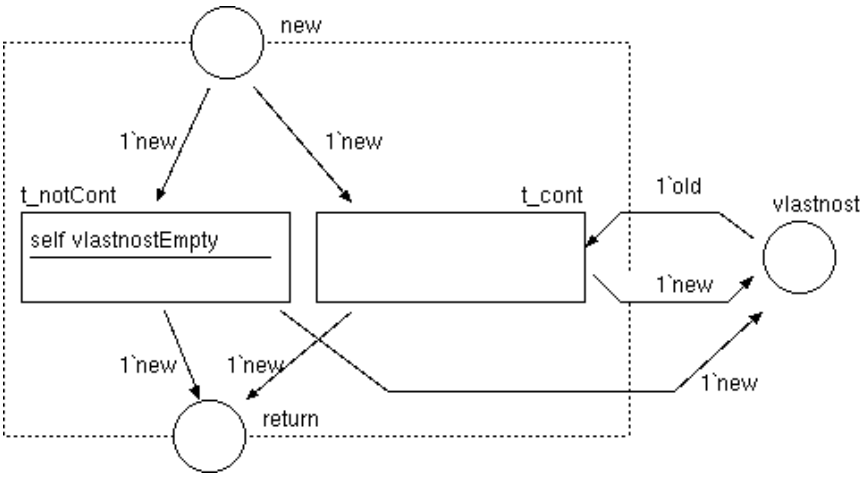
Kontrola na unikátnost je ukázána na obrázku 15. Pro tuto kontrolu musí objekt implementovat *Contains test*



Ilustrace 15: Metoda Add s testem na unikátnost

Setter

Pro klasickou vlastnost však využijeme vzor *Setter.* Při tomto vzoru musíme hlídat, aby na místě, ke kterému se setter váže, nebyl nikdy více než jeden objekt. Čili může tam být jeden nebo žádný. Když místu nastavujeme novou hodnotu, musíme tu starou zahodit. Pro testování přítomnosti prvku nám postačí, aby objekt implementoval *Empty test,* neboť test na přítomnost zajišťuje hrana.



Ilustrace 16: Setter

4.3.3 Clear

Posledním problémem spojeným s manipulací daty je odstranění objektu. Tento problém je pro vlastnosti poměrně jednoduchý, naproti tomu pro kolekce je výrazně složitější.

Protože pro odstranění prvku z kolekce, popřípadě vyčištění kolekce, je zapotřebí průchod všemi prvky, který řeší až vzor *Foreach,* je tato operace popsána až v diskuzi ke kapitole 4.5.2. Nyní popíšu pouze vzor Clear pro vyprázdnění vlastnosti.

Operace *Clear* má dvě části: zjistit zda je vlastnost prázdná a pokud ne, odčerpat objekt a zahodit jej. Pro první krok potřebujeme *Empty test* a pro druhý pouze jednu hranu odčerpání. Princip operace je dost podobný *Getteru* na ilustraci 12,pouze z přechodu `t_cont` se objekt už do místa vlastnost nevrací.

4.4 Podmíněné řízení

Je nezbytnou součástí jakéhokoli programu. Bez podmíněného řízení můžeme vytvářet nanejvýš sekvenční program, což je pro drtivou většinu programu nedostačující.

4.4.1 Modelování podmínky

První, co musíme udělat, je identifikovat, co vlastně v OOPN je podmínka. Z nejzákladnějšího hlediska lze říci, že podmínka vykonání (vykonání je reprezentováno přechodem) je proveditelnost přechodu. Proveditelnost lze ovlivnit v zásadě 3 způsoby:

1. Vhodným *Presetem*. To je způsob plynoucí z obyčejných Petriho sítí. Přechod je v tomto případě podmíněn přítomností tokenů na více místech nebo dokonce v určitém počtu.
2. Specifickým tokenem. Díky tomu, že tokeny jsou v OOPN objekty, lze pak hranu k přechodu definovat nějakým specifickým objektem. V OOPN jsou však proměnné

netypované, proto se v tomto ohledu využívají pouze symboly. Takže je možné podmínit přechod odčerpáním objektu `1`#state`, jakýkoli jiný objekt pak hrana nepustí. Jenom připomenu, že předat lze i n-tice, např. `1`(someObj, #state)`.

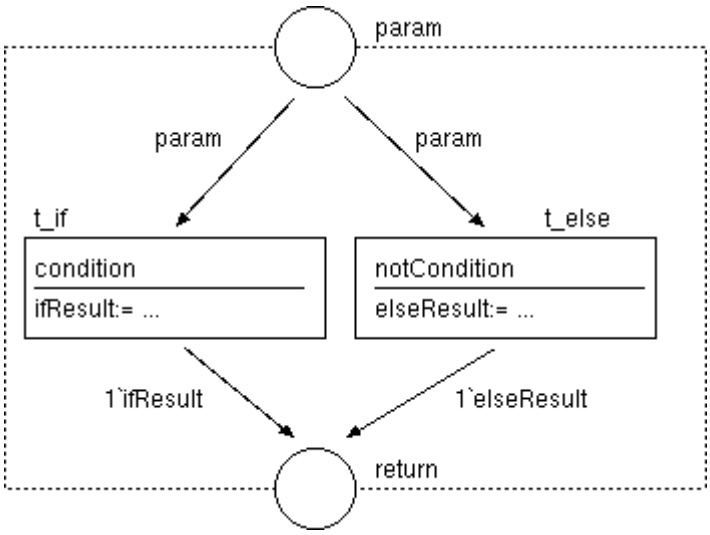
3. Implementace stráže přechodu. Nejčistší způsob z hlediska OOPN, ale také s nejširšími možnostmi, protože můžeme volat synchronní porty a inhibitory, nebo také provádět testy na rovnost či nerovnost.

Největším problémem podmínek proveditelnosti však je, že jsou všechny blokující. Při jejich nesplnění se model na daném místě zastaví a čeká na jejich splnitelnost. To může být výhodné, pokud modelujeme nějaké paralelní procesy a v jistém bodě musí jeden proces počkat na druhý. Ale když máme sekvenční systém, což je u aplikací poměrně běžné, musíme zajistit další běh programu. K tomu slouží konstrukce *If-else* a *Switch.*

4.4.2 Dvoucestné řízení – *If-else*

Sémantika podmínky If-else je: „pokud podmínka platí (*if*), proveď akci 1, pokud neplatí (*else*), proveď akci 2.“

Při porovnání podmínek s klasickými programovacími jazyky, u klasických jazyků nevzniká na podmínce blokování, takže podmínka *if* může vypadat jakkoli a větev *else* bude vždy jejím komplementem.



Ilustrace 17: If-else

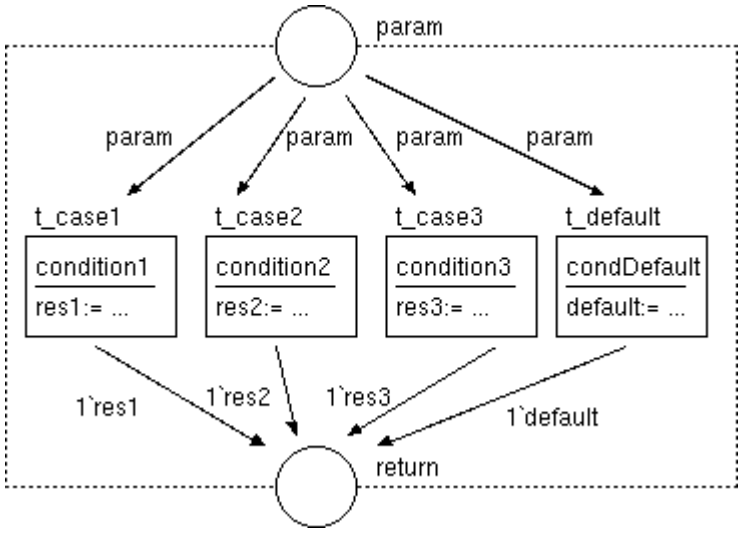
Ze samé podstaty Petriho sítí však musíme v OOPN implementovat podmínky pro všechny větve, a to tak, aby pokryly všechny možné případy. Pro *If-else* musí být tedy dvě podmínky k sobě komplementární. Lze například použít některý z testů uvedených v kapitole 4.2.2, jen pozor na komplement u *Empty testu.*

Diskuze

V OOPN nelze implementovat samotné *If* jinak, než že akci větve *else* necháme prázdnou. U klasických jazyků tato konstrukce existuje, protože pak implicitně existuje větev *else* „pokud *if* neplatí, nedělej nic a pokračuj“. V OOPN však tuto větev musíme explicitně implementovat.

4.4.3 Vícecestné řízení – *Switch*

Switch je definovaný jako sekvence testů nějaké proměnné na různé hodnoty. Z obrázku 18 je patrné, že *Switch* není nic jiného, než *If-else* s více větvemi. Důležitá je opět zachovat determinističnost. Jednotlivé podmínky se nesmí překrývat a musí dohromady pokrývat všechny možné případy vyhodnocení. Jinými slovy u S*witche* musí být vždy proveditelný právě jeden přechod. Bohužel přechod *default* sice lze použít, ale podmínku proveditelnosti musíme stejně nadefinovat, takže toto řešení pro OOPN není tak elegantní. Jen dodám, že tvorba podmínek může být proto dosti ošidná, lze se však inspirovat v kapitole 4.2.2 nebo příkladem v případové studii.



Ilustrace 18: Switch

Diskuze

Formálně bývá také *Switch* definován jako sekvence konstrukcí *If-else*, vnořených tak, že další test se nachází v těle předchozího *else.* Testy potom vytvoří kaskádu. V některých případech může být výhodné test ze S*witche* vydělit a udělat z něj předtest *If-else*. Tato operace může být výhodná především na počet exekucí stráže, pokud předtest filtruje významný počet dotazů.

Příklad: u třídy `RegistrationNet` porovnejte metody `verify: as:` a `newverify: as:`.

4.5 Cykly a kolekce

Následující kapitola se zaobírá cyklením, tj. opakováním nějaké akce většinou na základě platnosti nějaké podmínky.

4.5.1 Obecný *For* (*While*)

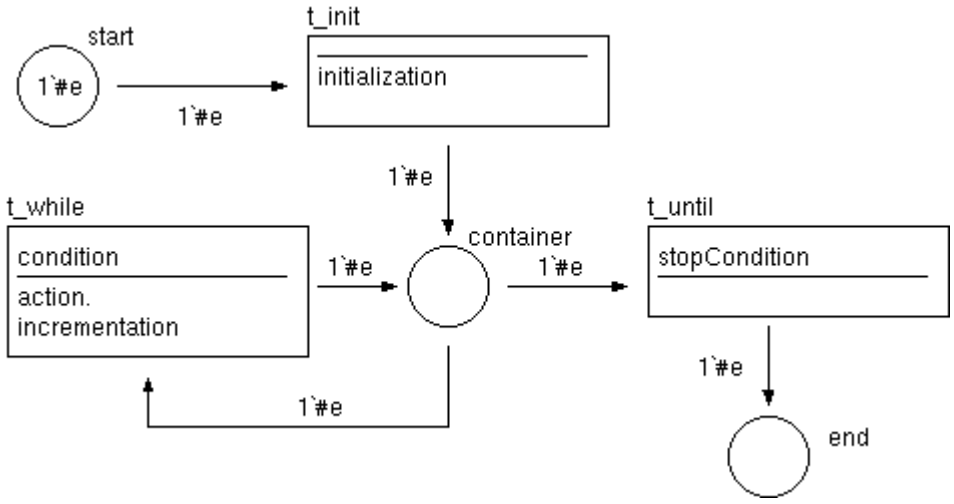
Tyto dva typy cyklů jsem dal k sobě z jedno prosté důvodu. Cyklus *For* bývá zapisován ve tvaru:

```
For (inicializace; podmínka platnosti; inkrementace)
{akce;}
```

Ze své definice je *For* však ekvivalentem následujícího pseudokódu:

```
inicializace;
While(podmínka platnosti)
{
      akce;
      inkrementace;
}
```

Uvedu tedy pouze vzor *For*, aplikovat *While* odstraněním inicializace a inkrementace je pak triviální operace. Na obrázku 19 je obecný cyklus *For*.



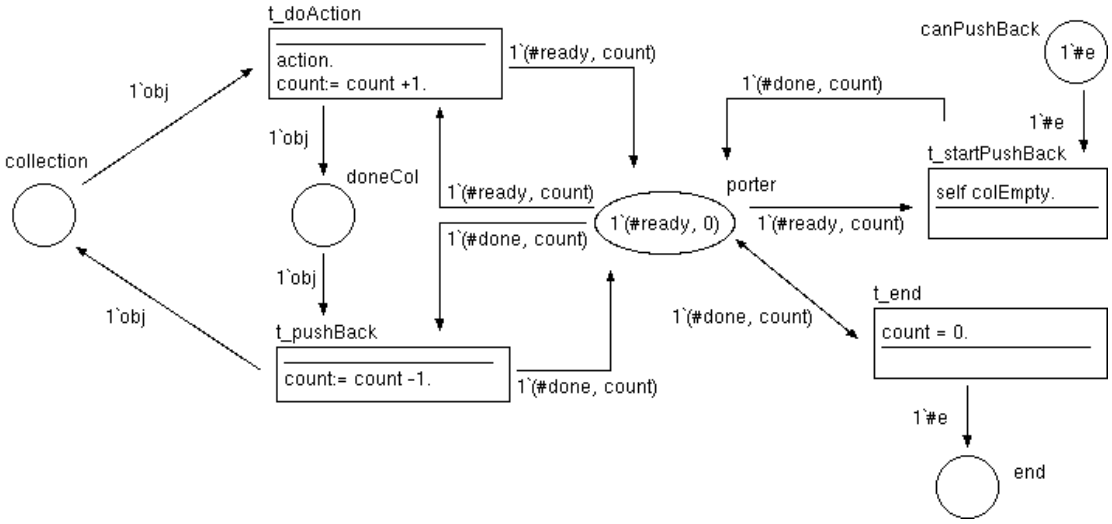
Ilustrace 19: Cyklus For

V celém cyklu putuje pouze token `1`#e`, záleží však na logice aplikace, mohou tu putovat i objekty a v přechodech pak můžeme volat jejich metody nebo v podmínkách jejich porty. Po startu musí vždy proběhnout přechod `t_init`, ve kterém je inicializace. Pokud do cyklu vstupují nějaké parametry, pak je vhodné je navázat už zde. V tomto portu nedoporučuji používat stráž, aby nedošlo k blokování.

Po inicializaci naplníme místo `container`. Když je zde přítomen objekt, pak je třeba rozhodnout, zda platí podmínka pro cyklus, nebo má cyklus skončit. To se provede dvojicí přechodů `t_while` a `t_until` se vzájemně komplementárními podmínkami přechodu (`condition` a `stopCondition`). Po provedení přechodu `t_until` cyklus končí.

4.5.2 Vzor *Foreach*

Foreach je nejčastější operace prováděná nad kolekcí, protože znamená projít všechny prvky kolekce a nad každým prvkem provést akci tak, aby pro každý prvek byla akce provedena právě jednou. Z kolekce, kterou mám v nějakém místě (viz kapitola 4.2.1) se však prvek vybere vždy náhodně. Proto nezbývá, než všechny objekty z místa odebrat, při tom nad nimi provést nějakou akci a poté je zase všechny vrátit.



Ilustrace 20: Foreach

Vzor na obrázku 20 je *Foreach* s použitím počítadla (o *Foreach* bez počítadla více v diskuzi). Motorem celého vzoru je místo `porter` (vrátný). Toto místo obsahuje dvojici objektů, přičemž první reprezentuje stav a druhý je číslo udávající počet zpracovaných položek, popř počet objektů, které zbývá vrátit do kolekce. Stav je na počátku nastaven na `#ready` (může čerpat). Akci nad prvkem provádí přechod `t_doAction`, který po každém zpracování zvýší počet zpracovaných objektů a umístí objekt do shromaždiště zpracovaných objektů `doneCol`. Musíme hlídat kdy se takto zpracují všechny objekty, a k tomu nám poslouží *Empty test* pro místo `collection`. Jakmile je tedy test proveditelný, proběhne přechod `t_startPushBack`, který nám změní stav vrátného na `#done` (v počítadle je teď celkový počet objektů v kolekci). Po této změně je proveditelný přechod `t_pushBack`, který postupně vrátí objekty do kolekce a při každém vrácení sníží počet objektů k vrácení i ve vrátném. Celou operaci pak ukončí přechod `t_end`, který je podmíněn stavem `#done` a tím, že již není žádný objekt k vrácení (opět platí `count=0`).
Příklad: v konferenčním systému metoda `print` třídy `Conference`.

Diskuze

Uvedený vzor je univerzální a funguje vždy. Přítomnost počítadla nám však může připadat zbytečná, když nepotřebujeme znát počet prvků kolekce. Alternativní řešení je použít *Empty test* nad místem `doneCol` pro indikaci vrácení všech objektů zpět. Potom bychom mohli počítadlo odstranit. K testu však potřebujeme *inhibitor* a ten lze použít pouze v síti objektu, v síti metody však ne.

Problém, který jsem však podrobněji neřešil, vyvstává při opakovaném spuštění, nebo při paralelním spuštění. Opakované spuštění není problém při použití v metodě. Voláním metody je

Upozorním ještě na jedno zjednodušení. Uvedený příklad má koncový stav `end`, abychom mohli indikovat konec provedení *Foreach*. Typické je to v metodě, protože ta se ukončí přítomností objektu v místě `return`.

4.5.3 Použití *Foreach* pro práci s kolekcemi

Zde uvedu výčet a popis zajímavých modifikací vzoru Foreach pro práci s kolekcemi. Předpokládám využití v metodách:

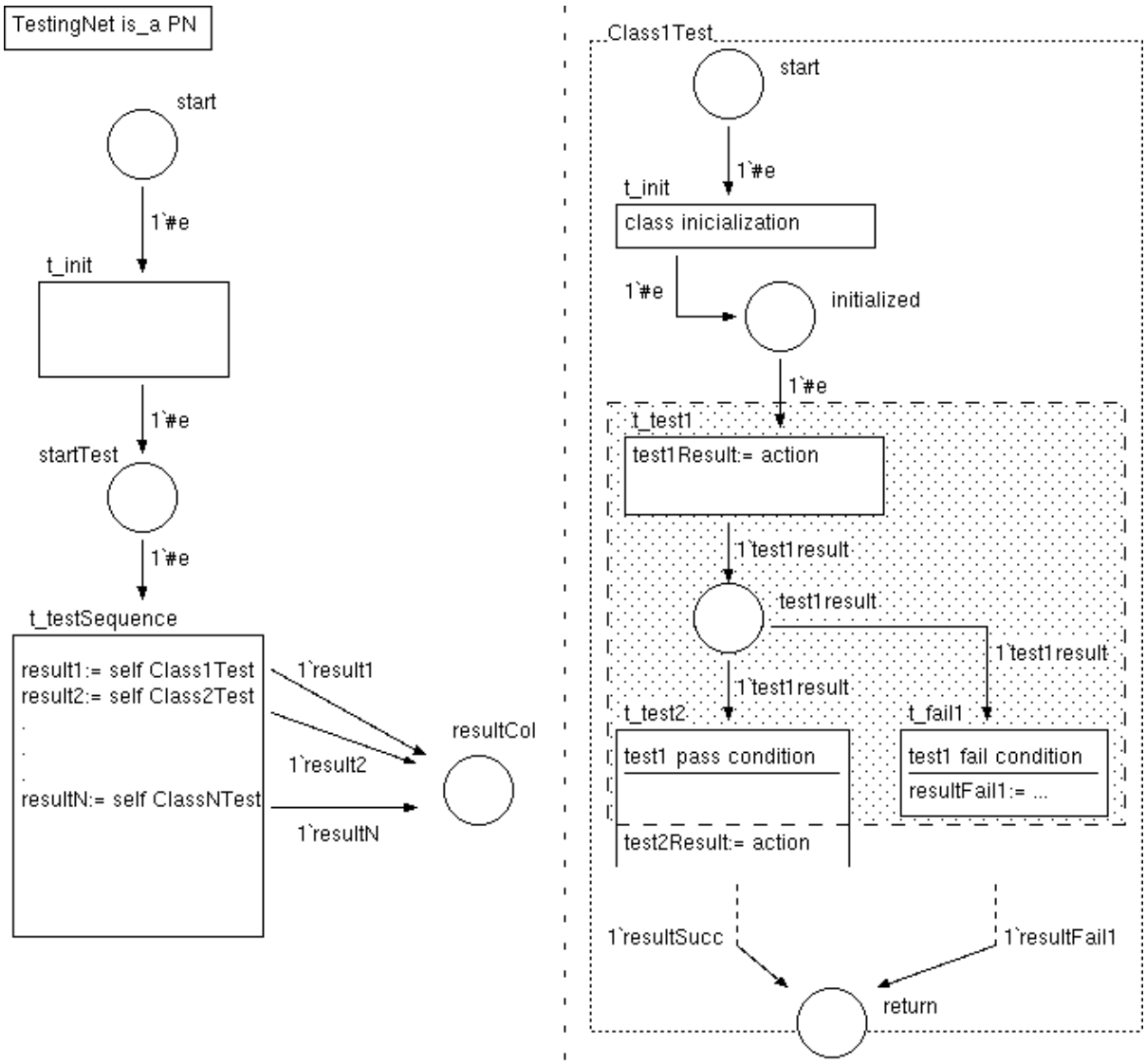
- Clear – zahození všech objektů v kolekci. Pro tento vzor stačí vypustit přechod `t_pushBack`.
- Remove – zahození jednoho konkrétního prvku podle klíče. Stačí přidat argument a mezi přechodem `t_doAction` a místem `doneCol` implementovat *If-else*, který v případě shody klíče objekt nepředá dál.
- Count – zjistí počet prvků. Ve chvíli provedení přechodu `t_startPushBack` si někam uložíme kopii count, abychom ji mohli v přechodu `t_end` zase vyzvednout.

4.6 Testování

Následují vzor je spíše námět pro systematické testování. Systém který jsme vytvořili je také dobré otestovat. A protože OOPN sestává ze tříd, můžeme testovat každou třídu zvlášť. Při dobré struktuře testů tedy můžeme nad testováním získat lepší kontrolu.

Návrh metodiky testování je na obrázku 21. Testovací třída je na obrázku vlevo, pojmenujme ji např. `TestingNet`. Průběh této sítě je poměrně jednoduchý: po startu se inicializuje, a poté provede přechod, ve kterém je sekvence testů jednotlivých tříd. Test pro jednu třídu je realizován jednou metodou třídy `TestingNet`. Tyto metody doporučuji pojmenovávat ve stylu `<jméno třídy>Test`, např. `Class1Test`. Jiné formy mohou být matoucí, např. `TestClass1` spíše evokuje, že jedná o testovací třídu.

Výsledky jednotlivých testů pak skladujeme v úložišti výsledků (na obr. `ResultCol`). V takovém případě však potom u výsledku musí být jasné, ke kterému testu patří. Popřípadě můžeme mít pro každý výsledek vlastní místo.



Ilustrace 21: Příklad testu

Samotný test je realizován v metodě (na příkladu `Class1Test`, v pravé části Ilustrace 21). Nejdříve je vhodné inicializovat testovanou třídu. Jeden test pak sestává z hlavního přechodu testu a vyhodnocení na úspěch/neúspěch. V hlavním přechodu (`t_test1`) se provede akce, jejíž výsledek se pošle dál. Výsledek se pak distribuuje dvěma přechodům, přičemž provedení přechodů je podmíněno vzájemně komplementárními podmínkami (úspěch/neúspěch). Pokud dojde na neúspěch, metoda může skončit a vrátit objekt indikující chybu. Naproti tomu úspěch testu je možno indikovat už ve stráži následujícího testu, protože splnění podmínky úspěchu má za následek pokračování v sekvenci testů. Na obrázku 21 vpravo je v rámci metody vyznačen rozsah jednoho testu (obdélník s tečkovaným podkladem). V případě, že metoda projde všechny testy, může vrátit objekt indikující úspěch testu třídy.

Diskuze

V rámci testovací metody byly jednotlivé testy řazeny sekvenčně. Nabízí se možnost i paralelního řazení, test by potom běžel ve více nezávislých větvích. Jednoznačnou výhodou takového přístupu by byla možnost sledovat i při neúspěchu parciální úspěchy. Je však třeba počítat s jistými obtížemi při vyhodnocení celé metody. Jednotlivé větve mohou skončit různě a přítomnost pouze jediného neúspěchu znamená neúspěch celé třídy. Další věc, kterou by bylo třeba zajistit, je synchronizace větví před opuštěním metody, aby některá větev neukončila celou metodu, i když se některý test ještě provádí.

5 Závěr

Předmětem práce bylo vytvořit knihovnu často se opakujících vzorů v OOPN (Object Oriented Petri Nets). K tomu bylo zapotřebí se seznámit s celým formalismem a pochopit základy modelování tímto nástrojem. Nejlepší cestou, jak toho dosáhnout bylo při práci na dokončení případové studie Konferenční systém.

Vzory, které jsem nakonec identifikoval jsou zpravidla konstrukce známé z klasických programovacích jazyků. Je to dáno především tím, že jsem zvyklý těmito jazyky přemýšlet, ale také proto, že konstrukce takových jazyků, jsou pro algoritmické problémy velmi vhodné.

Původní myšlenkou OOPN však bylo dokázat, že modelovací nástroj lze použít i k tvoření programů a že pro tento účel může být stejně efektivní. Paralele těchto dvou přístupů se tedy nevyhneme.

Význam práce a náměty pro navazující projekty

Při studiu OOPN a *PNtalku* jsem narazil na problém pochopit jejich filozofii. Tato práce tím, že identifikuje známé konstrukce může posloužit jako úvod do tvorby programů v OOPN především pro programátory navyklé klasickým programovacím jazykům.

V práci však velmi často zaznával i problém rozlišení kolekce a vlastnosti. Pro kolekci vždy existovaly trochu jiné verze metod. Nejlepším způsobem, jak tento problém vyřešit, je vytvořit robustní třídu pro práci s kolekcemi a implementovat všechny metody naznačené v této práci. Pokud mluvíme o kolekcích tak součástí takové práce by bylo například i vyřešit indexování prvků v kolekci, v případě obecné kolekce problém generického klíče apod.

Jiné využití knihovny je doimplementovat do *PNtalku* „syntaktický cukr“, tedy nadefinovat klíčová slova, která se při překladu rozgenerují na navržený vzor.

Velmi podobné využití, avšak na jiném principu je implementovat pro vzory třídy, které budou mít požadovanou logiku, ale jejich použití (například přes metody) by bylo značně jednodušší, než implementovat logiku znovu.

Knihovnu lze však využít i pro sofistikovanější problémy. Identifikací základních konstrukcí se totiž přibližuje možnost automaticky generovat kód. A to jak ve směru OOPN na nějaký klasický jazyk, tak ve směru z klasického jazyka do OOPN. V delším horizontu si tak lze představit nástroj, který udržuje návrh modelu pomocí OOPN i jeho implementaci nějakým programovacím jazykem v ekvivalentním stavu. Za takové situace pak dobrý návrh generuje kvalitní kód a změna kódu se automaticky reflektuje v modelu.

Literatura

[1] PNtalk Project [online]. [cit. 11-05-2011]. Dostupné na: <http://perchta.fit.vutbr.cz:8000/projekty/12>

[2] Sharp, A., McDermott, P: Workfow modeling: Workflow modeling: tools for process improvement and application development, Norwood, Artech House, 2001, ISBN 1-58053-021-4

[3] Češka, M.: Petriho sítě, Brno, Akademické nakladatelství CERM, 1994, s.9-10 ISBN 80-85867-5-4

[4] Janoušek, V.: Modelování objektů Petriho sítěmi, Brno, CZ, UIVT FEI VUT, 1998

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

[6] Kočí, R., Janoušek, V., Zbořil, F.: Object Oriented Petri Nets -- Modelling Techniques Case Study, In: Second UKSIM European Symposium on Computer Modeling and Simulation, Liverpool, GB, IEEE CS, 2008, s. 165-170, ISBN 978-0-7695-3325-4

[7] Kočí, R., Janoušek, V.**:** System Design with Object Oriented Petri Nets Formalism, In: The Third International Conference on Software Engineering Advances Proceedings ICSEA 2008, Los Alamitos, US, IEEE CS, 2008, s. 421-426, ISBN 978-0-7695-3372-8

Seznam příloh

Seznam ilustrací

Ilustrace 3 je adaptace obrázku Obr.2 z knihy Petriho sítě[3].

Ilustrace 4 je použita z internetové stránky projektu PNtalk
<http://www.fit.vutbr.cz/~janousek/pntalk/node8.html>

Příloha 1. Návod pro spuštění Konferenčního systému

V příloze je k dispozici image pro virtuální stroj *Squeak*, který obsahuje implementaci *PNtalku* v jazyce *Smalltalk*, a v rámci příkladů také Konferenční systém. Je připravena i spouštecí třída ve *Smalltalku* `ConfTestWrapper`. K projektu je přibalena i spustitelná verze pro Windows ve složce */PNtalk/SqueakVM-Win32-3.11.5-bin/Squeak.exe*. Návod na spuštění naleznete v souboru *readme.txt.*

Jak spustit test

1. Spusťte *Squeak* nad imagem umístěným v */PNtalk/PNtalk-hanak.image*
2. Otevřete okna `Transcript` a `Workspace` (z lišty `Tools`)
3. Do okna `Workspace` napište následující kód:

   ```
   test := ConfTestWrapper new.
   test init.
   ```

   Tento kód označte a z nabídky akcí (střední nebo pravé tlačítko myši) vyberte akci „*do it"*
4. Pokud se v okně `Transcript` objevilo pouze „Inicializace testu", otevřete okno `My Repository` (když není otevřené, naleznete ho na liště *SmallDEVS*), a zkontrolujte stav simulace *Root/PNtalk simulations/Default*. V případě, že má příznak [S], z nabídky akcí vyberte „*Start simulation*".
5. V okně `My Repository` pod simulací *Root/PNtalk simulations/Default* by měly vzniknout objekty simulace, kde si můžete prohlédnout stav ve které se simulace nachází. V okně `Transcript` je pak výstup, který se produkuje při provádění simulace a který je přidán do logiky aplikace.

Jak lze systém prohlížet/editovat

1. Spusťte *Squeak* nad imagem umístěným v */PNtalk/PNtalk-hanak.image*
2. Otevřete okno `My Repository`, pokud není otevřené, naleznete ho na liště *SmallDEVS*.
3. Konference je umístěna v *Root/PNtalk Classes/Conference Example*
4. Vyberte třídu a z nabídky akcí (střední nebo pravé tlačítko myši) vyberte akci „*Open*"
5. V textovém editoru můžete upravovat logiku a stiskem *Ctrl+S* práci uložíte a zároveň zkompilujete