VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO PRÁCI S OBJEKTOVĚ ORIENTOVANÝMI PETRIHO SÍTĚMI

OBJECT ORIENTED PETRI NET TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ANTONÍN NEUŽIL**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Neužil Antonín, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Nástroj pro práci s Objektově orientovanými Petriho sítěmi**
Object Oriented Petri Net Tool
Kategorie: Softwarové inženýrství

Zadání:

1. Prostudujte formalismus Objektově orientovaných Petriho sítí (OOPN) a jazyk PNtalk. Seznamte se s aktuální implementací simulačního serveru OOPN a s jeho komunikačním protokolem.
2. Navrhněte nástroj pro editaci a ladění modelů OOPN. Nástroj musí umožnit ukládání a načítání modelů ve vhodném formátu a ve formátu jazyka PNtalk, komunikovat se simulačním serverem, spouštět simulace modelů, krokovat simulaci, definovat body zastavení a modifikovat stav simulace.
3. Implementujte nástroj v jazyce Java. Rozšiřte nástroj o možnost exportu modelů do formátu SVG (Scalable Vector Graphics).
4. Vytvořte manuál a sadu příkladů demonstrující možnosti nástroje, zejména komunikaci se simulačním serverem.
5. Diskutujte dosažené výsledky a navrhněte možná rozšíření nástroje. Výsledky také prezentujte formou posteru.

Literatura:

- Janoušek, V.: Modelování objektů Petriho sítěmi, Brno, CZ, UIVT FEI VUT, 1998

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 20. května 2020
Datum schválení: 31. října 2019

Abstrakt

Tato práce pojednává o vývoji a použití grafického editoru objektově orientovaných Petriho sítích implementovaných v jazyce PNtalk. Nejprve jsou zde popsány Petriho sítě rozšiřující se s postupem času. Navazuje na ně popis objektově orientovaného paradigma. Dále je zde popsána struktura objektově orientovaných Petriho sítí v implementaci PNtalku. Dále práce uvádí popis návrhu aplikace, kde je uvedena struktura celé aplikace z několika pohledů. Hlavní motivací pro vytvoření tohoto nástroje je snaha zpřístupnit objektově orientované Petriho sítě jejich uživatelům. Aplikace byla implementována ve frameworku JavaFX a celý postup je uveden v následující kapitole spolu s testováním. Výsledek práce je prezentován v příkladech obsažených v poslední kapitole, která obsahuje i podrobný manuál pro ovládání aplikace. Výsledný nástroj ve spolupráci s PNtalk serverem dovoluje uživatelům pohodlnější práci s OOPN a jejich vizuální podobou.

Abstract

This work deals with the development and use of a graphical editor of object-oriented Petri nets implemented in the PNtalk language. First, Petri nets that expand over time are described here. They are followed by a description of the object-oriented paradigm. Next, the structure of object-oriented Petri nets in the implementation of PNtalk is described here. Further work presents a description of the application design, which shows the entire structure of the application from the perspective of views. The main motivation for creating this tool is the effort to make object-oriented Petri nets accessible to their users. The application was implemented in the JavaFX framework and the whole procedure is given in the following chapter together with testing. The result of the work is presented in the examples contained in the last chapter, which also contains detailed manuals for controlling the application. The tool in cooperation with the PNtalk server enables users to work more pleasantly with OOPN and their visual appearance.

Klíčová slova

Petriho síť, Objektově orientované Petriho sítě, PNtalk, PNML, SVG, PNtalk server, grafický editor, Java, JavaFX

Keywords

Petri net, Object oriented Patri nets, PNtalk, PNML, SVG, PNtalk server, graphic editor, Java, JavaFX

Citace

NEUŽIL, Antonín. *Nástroj pro práci s Objektově orientovanými Petriho sítěmi.* Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Nástroj pro práci s Objektově orientovanými Petriho sítěmi

Prohlášení

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

.....
Antonín Neužil
3. června 2020

Poděkování

Chtěl bych poděkovat vedoucímu mé práce Ing. Radku Kočímu, Ph.D. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu práce věnoval. Také bych chtěl poděkovat všem, kteří se účastnili testování.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Petriho sítě jsou stále efektivním nástrojem pro modelování a simulaci systému. Jejich grafická podoba není nikterak složitá a snadno uchopitelná. S dlouhodobou dominancí objektově orientovaného přístupu k vývoji systémů jsou Petriho sítě vhodným nástrojem pro jejich návrh a simulaci. Obohacení Petriho sítí o objektově orientované paradigma je činí o něco složitější, ale po jejich správném využití mnohem efektivnější a přehlednější. Implementace objektově orientovaných Petriho sítí známa pod názvem PNtalk má přesnou specifikaci, která je činí lehce pochopitelné. Práce se zabývá tvorbou nástroje, který bude umět pracovat s danou implementací jazyku PNtalk. Tento nástroj si bere za úkol zpříjemnit uživateli práci s OOPN.

Nejdříve jsou v kapitole 2 popsány jednotlivé druhy Petriho sítí, které se postupem času vyvíjely až k objektově orientovaným. Jsou zde stručně vysvětleny základy objektově orientovaného paradigmatu. Rovněž je v kapitole popsána základní struktura objektově orientovaných Petriho sítí v PNtalku.Následující kapitola 3 obsahuje specifikaci požadavků a návrh cílové aplikace. Kapitola obsahuje všechny provedené analýzy z hlediska vývoje aplikace tak, aby výsledná struktura před implementací byla zjevná. Také jsou zde rozbory dalších technologii, které se podílejí na jejím vývoji. Zbylé dvě kapitoly 4 a 5 se zaměřují na konkrétní způsob implementace, testování a ovládání aplikace. V implementaci jsou uvedeny veškeré technologie, které byly použity pro její vývoj. Kapitola 5 obsahuje návod pro obsluhu aplikace. Kromě návodu jsou zde uvedeny také konkrétní příklady, kterými se může uživatel inspirovat.

Kapitola 2

Objektově orientované Petriho sítě

V této kapitole jsou nejdříve uvedeny obecné informace o různých typech Petriho sítí. Dále se zde kapitola zaměřuje na konkrétní vlastnosti objektově orientovaných Petriho sítích. Je zde také popsána konkretní implementace objektově orientovaných Petriho sítí známa jako PNtalk, který byl vyvinut na FIT VUT v Brně. Petriho sítě jsou matematický nástroj pro reprezentaci distribuovaných systémů. Vytvořil jej německý matematik a vědec Carl Adam Petri ve své disertační práci v roce 1962. Z počátku byl pojem Petriho síti pouze orientovaný bipartitní graf, zobrazující strukturu systému. Později byl rozšiřován tak, aby vyhověl praktickým potřebám při vývoji systému. Při těchto začátcích Petriho sítě narážely na chybějící datový a hierarchický koncept. Chybějící datový koncept zapříčinil nárůst velikosti sítě, protože všechny její operace bylo nutné vytvářet přímo na struktuře. Bez hierarchického modelu nebylo možné vytvořit velký model pomocí několika menších propojených modelů přes definované rozhraní[8]. Tyto nedostatky vyřešil v 80. letech dánský vědec Kurt Jensen, který vytvořil barevné Petriho sítě. V podstatě tím položil základy k vývoji vyšší úrovně Petriho sítí. Zavedením barevných tokenů vytvořil datové typy. Později také přidal další prvky jako jsou proměnné, deklarace typů, stráže a akce přechod[4]. Pro rozsáhlejší modely bylo nutné zavést hierarchii do Petriho sítí. Vznik hierarchických Petriho sítí umožnil vícenásobnému využití jednotlivých komponent, paralelní práci při návrhu modelu a snadnější údržbu systému[8]. Vývoj, který započal v 60. letech, se stále ještě nezastavil. Důkazem je konference nazvaná *Petri net 2020*, která je součástí série konferencí *Paris Nord Summer of LoVe (Logic and Verification)* odehrávající se na *Université Paris XIII Nord* v průběhu léta roku 2020. Tato konference je v pořadí již 41[10].

2.1 Petriho sítí

Jednotlivé typy Petriho sítí korespondují s časovým horizontem, jak probíhal vývoj. Sítě se postupně obohacovaly o modelovací schopnosti, které řešily praktické potřeby při návrhu systémů. Následující typy postupují od jednodušších ke složitějším.

- C/E Petriho sítě jsou výpočetně nejslabší variantou. Vyjadřovací silou odpovídají konečnému automatu.
- P/T Petriho sítě jsou podobné předchozímu typu. Hlavním rozdílem je vlastnost místa, která umožňuje obsahovat více než jeden token.

- Petriho sítě s inhibiční hranou přináší velkou výpočetní sílu. Inhibiční hrana, která může směřovat pouze od místa k přechodu, dává výpočetní sílu úrovně Turingova stroje.
- Petriho sítě s prioritami řeší problém nedeterminismu. Každý přechod má přidělené nezáporné celé číslo, které značí jeho prioritu. Pokud z jednoho místa lze provést více přechodů, vždy se provede ten s vyšší prioritou.
- Časované Petriho sítě přenášejí diskrétní systémy do spojitých. Zároveň kromě využití deterministicky zadaného času, lze využít i stochastických událostí. Samotné připojení časového paradigmatu k Petriho sítím se také liší v závislosti k danému prvku, u kterého je využito. .
  - V případě přidání času k přechodu, čas značí dobu provedení.
  - V případě místa, čas vyjadřuje dobu pobytu tokenu v daném místě.
  - Časové razítko u tokenu představuje dobu po které bude možno token znova použít. Tato doba se obvykle obnoví po provedení přechodu
  - Čas uvedený u hrany představuje dobu pro přesun tokenu.
- Barevné Petriho sítě obohacují modely o různé typy tokenů. Dále se rozšiřují o proměnné, deklarace typů, multimnožiny, podmínky u přechodů a další. Multimnožiny se mohou nacházet u míst a hran k nim připojených, které značí dané tokeny, se kterými je povoleno pracovat.
- Hierarchické Petriho sítě umožnily strukturování modelů. Model se dělí na sítě a podsítě. Síť může být rozšířena jednou či více podsítěmi pomocí hierarchických konstruktorů.
  - Substituce míst představuje nahrazení místa příslušnou podsítí.
  - Substituce přechodu je obdobné nahrazení přechodu příslušnou podsítí.
  - Slučování míst pomáhá k vytváření přehlednějších sítí.
  - Slučování přechodů představuje stejný mechanismus, jak u předchozího případu, ale pouze pro jiný element.
- Objektově orientované Petriho sítě existují v různých variantách. Všechny se snaží k synchronizačním mechanismům Petriho sítí připojit objektově orientované paradigma, které zahrnuje dědičnost, polymorfismus apod. Na podrobný popis objektově orientovaných Petriho sítí se zaměřuje podkapitola 2.2.

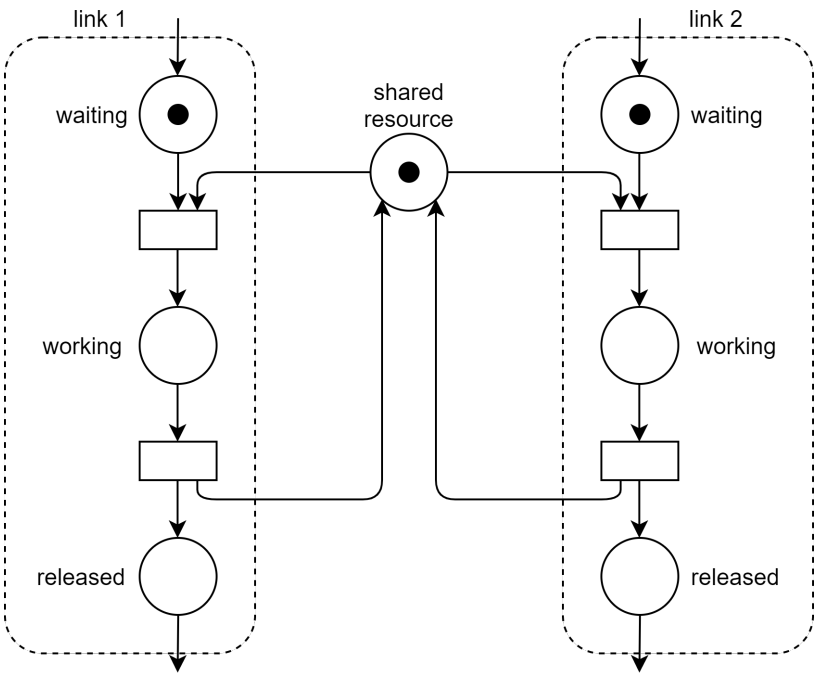
2.1.1 Základní konstrukce Petriho sítí

Před samotným zaměřením se na objektově orientované sítě je potřeba uvést základní kostru společnou pro všechny typy Petriho sítí. I přes četná rozšíření, které se nabízejí, je původní koncept po vizuální stránce téměř netknutý. Základní strukturou je orientovaný bipartitní graf sestávající ze čtyř elementů.

- Prvním elementem se nazývá místo nebo také stav v případně C/E Petriho sítí podmínka. Po vizuální stránce se značí kruhem nebo elipsou.
- Druhým elementem je přechod v případě C/E Petriho sítí události. Spolu s místy tvoří uzly grafu. V síti je značen jako obdélník, čtverec nebo také krátká svislá úsečka.

- Jednotlivé uzly jsou spojovány pomocí orientovaných hran. Tyto hrany představují šipky směřující vždy pouze od místa k přechodu nebo naopak.
- Tokeny nebo také značky se zakreslují jako tečky do míst. Jejich průchod sítí představuje simulaci daného modelu.

Obrázek 2.1 reprezentuje příklad klasického sdílení zdroje mezi dvěma linkami. V úvodním značení se nedeterministicky rozhodne, která linka bude obsloužena. Po vybrání např. první linky a jejím provedením, se sdílený zdroj vrátí a je dostupný pro druhou linku, jestliže mezi tím nepřišel další token do čekacího místa první linky. Poté by opět došlo k nedeterministickému rozhodnutí. Problém nedeterminismu řeší Petriho sítě s prioritami, které mohou při takovém rozhodování udělit přechodům rozdílné priority. Ten s vyšší prioritou by poté dostal přednost.



Obrázek 2.1: **Obecná Petriho síť. Převzato z** [6]

2.2 Základy Objektové orientace

Petriho sítě jsou dobře známým grafickým a modelačním nástrojem pro souběžné a distribuované systémy. Jelikož byla pro programovací jazyky přijata objektová orientace, je více než přirozené, že i Petriho sítě se inspirovaly objektově orientovaným programováním[12]. Existuje více navržených druhů Petriho sítí zaměřených na objektovou orientaci. Tato práce je zaměřena na objektově orientované Petriho sítě spojené s nástrojem *PNtalk* navrženým na FIT VUT.

Před samotnými objektovými Petriho sítěmi je vhodné zopakovat, co se pod pojmem objektová orientace myslí. Objektově orientované programování je již nedílnou součástí vývoje různých systémů. Jedná se o způsob myšlení a implementace, kde se klade důraz

na znovupoužitelnost. Následující podkapitola má za cíl pouze stručně vysvětlit základní koncept objektové orientace.

2.2.1 Třídy a objekty

Základní myšlenkou objektové orientace je pohled na program jako na systém objektů, který se pro uživatele jeví více intuitivně, neboť i reálný svět je tvořen objekty. Po formální stránce lze objekt definovat jako ucelenou jednotku, která obsahuje následující tři části.

- Vnitřní stav je soubor atributů, které mohou být proměnnými, případně dalšími objekty.
- Vnitřní chování je soubor metod. Metody lze chápat jako procedury, které pracují s vnitřním stavem objektu.
- Protokol zpráv udává veřejné rozhraní a způsob jeho mapování na vnitřní procedury objektu[2].

Třída je základní konstrukční prvek v objektově orientovaném programování a lze ji chápat jako schéma případně vzor. Třída sama o sobě popisuje vnitřní strukturu objektu a jeho vnější rozhraní. Vznik objektů probíhá instancí třídy. Obrázek 2.2 představuje jednoduchý případ pro objektově orientované programování. Máme zde například základní třídu *User*, která má atributy identifikátor, jméno a email. Jediná metoda *changeName*, která se zde nachází, představuje vnitřní chování. Dále se zde nachází další dvě třídy *Customer* a *Administrator*. Instancí těchto dvou tříd vznikly dva objekty *c1* a *a1*. Každý objekt má již definovaný konkrétní vnitřní stav. Konkrétní vnitřní chování je definované v rámci třídy.

2.2.2 Dědičnost

Dědičnost je klíčová vlastnost v objektově orientovaném programování. Hlavním účelem je tvorba nových datových struktur na základě starých. Pomocí dědičnosti je možné vytvořit hierarchii tříd se společnými vlastnostmi, které se mohou dále specifikovat. Konkrétně dědičnost umožňuje rozšiřovat nebo modifikovat atributy a metody daných tříd. Nové třídy tedy mohou vznikat specifikací starých tříd, což vede k tvorbě hierarchie.

V příkladu na obrázku 2.2 je hierarchie tvořena třemi třídami. Původní třída *User* obsahuje společné atributy a metodu pro dvě dědící třídy *Customer* a *Administrator*. Třída *Customer* rozšiřuje třídu uživatele o atribut pole objednávek, nazvaných *orders*. Také je zde přidaná metoda *sendOrder*, která má představovat možnost zákazníka provést odeslání objednávky. Podobně třída *Administrator* se rozšiřuje o další vlastnosti odpovídající jejímu významu.

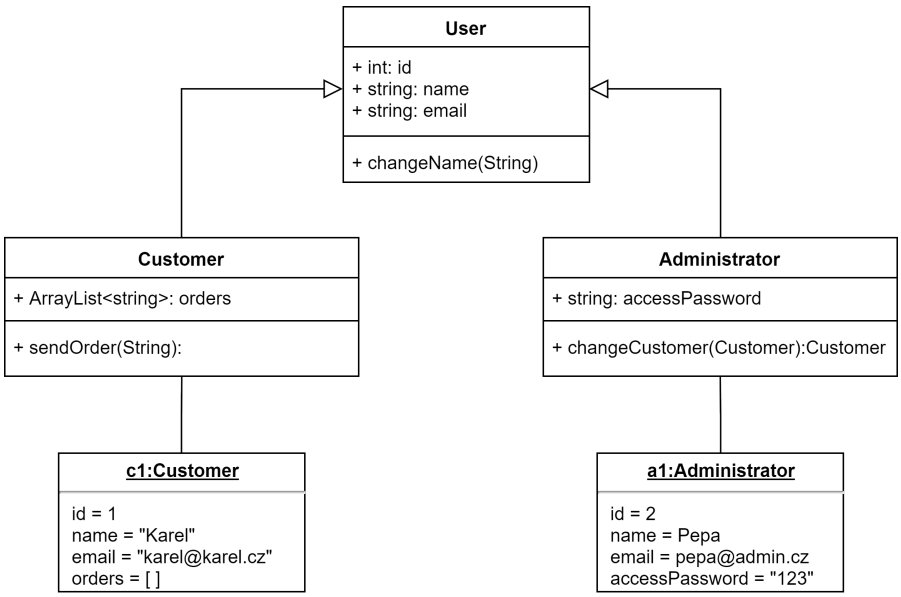
2.2.3 Zapouzdření

Koncept zapouzdření má za význam zabalit, případně skrýt určitá data nebo procesy daného objektu. Objekty mají definované rozhraní, které zpřístupňuje komunikaci s daným objektem. Pokud je potřeba např. změnit některá data v objektu, je zapotřebí takto učinit pomocí rozhraní, které musí být správně definováno, aby neohrozilo konzistenci objektu. V případě na obrázku 2.2 je v třídě *Customer* definovaná metoda *sendOrder*, která již podle názvu má za význam vytvořit v systému objednávku daného zákazníka. Je tedy zřejmé, že proces vytvoření objednávky bude obsahovat více než jen přidání objednávky do pole *orders*. Pokud tento příklad představuje pouze zlomek nějakého systému, tak si lze představit,

že dále je třeba vytvořit objekt objednávku, která se zobrazí administrátorovi k vyřízení. Může následovat mnoho dalších skrytých procesu, které ale samotný zákazník nemusí vidět.

2.2.4 Polymorfismus

Dalším důležitým aspektem objektově orientovaného programování je polymorfismus. Polymorfismus umožňuje používat jednotné rozhraní pro práci s různými typy objektů. V principu umožňuje zavolat různé metody se stejným jménem. Rozhodnutí, která metoda má být zavolána, je provedeno dynamicky v závislosti na typu objektu. Rovněž dovoluje volat stejnou metodu s různými parametry.



Obrázek 2.2: **Příklad objektově orientované struktury**

2.3 Základní konstrukce OOPN

Podobně jak u ostatních objektově orientovaných programovacích jazyků jsou i zde hlavními prvky objekty a třídy. Celý model je poté tvořen hierarchicky organizovanou množinou tříd, ve které je jedna třída označená příznakem *main*. Po spuštění celého modelu je na úvod vytvořena instance této třídy, ve které se aktivuje objektová síť. Z této objektové sítě dále pokračuje běh celého modelu. Před vysvětlením konceptu tříd je třeba uvést elementární prvky PNtalku a jejich význam.

2.3.1 Termy

Termy jsou nejjednoduššími výrazy, které se mohou objevit v PNtalku. Oproti originálním Petriho sítím termy nahradily tokeny. V PNtalku je definováno několik typů termů[3].

1. Literály jsou základními primitivními objekty, které se v síti mohou vyskytovat. Existuje několik druhů literálů. Kromě níže zmíněných literálů existují v definici Smalltalku i další, které jsou platné i pro PNtalk. Toto jsou základní literály.

- Řetězce. Jedná se o posloupnost jednoho nebo více znaků, které jsou ohraničeny apostrofy. Příklady řetězců jsou 'w', 'hello', atd.
- Znaky. Znaky jsou primitivními objekty zastupující symboly abecedy. Jejich literály jsou uvozeny symbolem *$*. Příkladem znaku je $c.
- Symboly. Symboly jsou primitivní objekty používané především jako jména. Jejich zápis je s prefixem *#*. Příklady symbolů jsou #h, #D12, atd.
- Čísla. Jedná se o primitivní objekty reprezentující číselnou hodnotu. Mohou mít prefix znaménka - pro záporná čísla. Rovněž mohou obsahovat desetinou tečku a znak *e* pro reprezentaci exponenta. Příklady čísel jsou 1, 1.2, -2.3, 15.3e-55.
- Booleovské konstanty. Podobně jak v ostatních jazycích existují pouze dvě *true* a *false*.
- Nedefinované objekty. Jedná se o prázdný výraz, reprezentovaný identifikátorem *nil*. Příkladem jsou neinicializované proměnné, které mají hodnotu právě *nil*.

2. Proměnné. Proměnné podobně jak v jiných programovacích jazycích mohou zastupovat různé objekty ať už primitivní nebo neprimitivní. Jsou značeny sekvencí znaků začínající malým písmenem například a, aBC.

3. Pseudoproměnné. V PNtalku se mohou objevit pouze dvě, *self* a *super*. Jejich hodnota vychází z kontextu modelu.

4. Jména tříd. Podobně jak proměnné, jedná se o identifikátory začínající velkým písmenem. Jsou vyhrazeny pro jména tříd a dynamicky se dále nemohou měnit. Příkladem jsou *PN*, *AB1, atd.*

2.3.2 Zasílání zpráv a inskripce přechodu

Zaslání zprávy je výraz, který se může objevit jako součást stráže i akce přechodu[3]. Tvar zprávy se sestává z adresáta a konkrétní zprávy. Konkrétní zpráva pak je tvořena selektorem a argumenty. Rozdílnými selektory se zprávy dělí na tři druhy[3].

- Unární zpráva. Příkladem je *A new*. Unární operace neobsahuje argumenty. *A* je adresát a v případě unárních zpráv je identifikátor selektorem.

- Binární zpráva. Selektory zpráv jsou tvořeny symboly. Jedná se například o symboly *+*, -, *=*, atd. Všechny symboly, které lze použít pro binární operace lze nalézt v obecné definici Smalltalku. Příkladem je *9 * 7*, kde *9* představuje adresáta, *** je selektor a *7* reprezentuje argument.

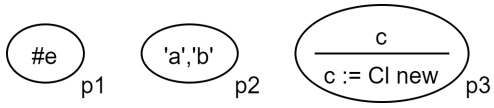
- Zprávy s klíčovými slovy. Zpráva obsahuje alespoň jeden klíč, který představuje identifikátor zakončený dvojtečkou.

Zasílání zpráv probíhá ve stráži a akci přechodů. Stráž přechodu tvoří sekvence výrazů, které jsou odděleny tečkou. Jednotlivé výrazy značí samotné zaslání zprávy. Jednotlivé výrazy v sekvenci jsou konjunkcí k vytvoření výsledku celé stráže určitého přechodu. Akce přechodu může mít na rozdíl od stráže výraz přiřazení. Je zde možné utvořit sekvenci výrazů, které pospolu představují sekvenční provádění.

2.3.3 Místa, přechody a hrany

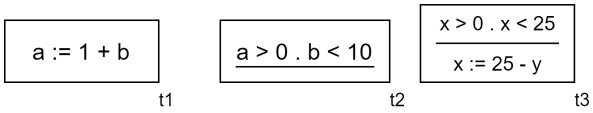
Základní prvky v PNtalk síti mají jasně danou grafickou podobu. Sítě jsou tvořeny třemi základními prvky.

Místo nebo také stav je zobrazeno kružnicí nebo elipsou. Každé místo musí obsahovat identifikační jméno, které pokud není specifikováno programátorem, je automaticky generováno. Místo se může skládat ze dvou částí. První části je počáteční značení. Počáteční značení může obsahovat term, seznam nebo více do sebe zanořených seznamů obsahujících termy. Rovněž je v PNtalku povoleno použít proměnnou jako značení. Definici proměnné má za úkol počáteční akce místa, která tvoří jeho druhou část. Počáteční akce místa je syntakticky stejná s akcí přechodu. Obrázek 2.3 obsahuje tři různá místa. Místo označené *p3* obsahuje v počátečním značení proměnnou *c* a pod ní se nachází počáteční akce místa, která ji definuje[3].



Obrázek 2.3: **Místa**

Přechody jsou graficky znázorněné jako čtyřúhelníky. Podobně jako u místa obsahuje jednoznačný identifikátor. Rovněž se přechod skládá ze dvou částí, stráž a akce. Stráž obsahuje sekvencí výrazů. Akce na rozdíl od stráže může kromě sekvence výrazu obsahovat i výraz přiřazení. Graficky je stráž přechodu vždy podtržena. Obrázek 2.4 představuje tři varianty rozdílných zápisu stráží a přechodů. Přechod *t1* zobrazuje přechod pouze s akcí. Naopak přechod *t2* obsahuje pouze stráž. Poslední přechod *t3* obsahuje obě zároveň.



Obrázek 2.4: **Přechody**

Posledními základními prvky jsou hrany, ke kterým je připojený hranový výraz. Hranový výraz syntakticky odpovídá počátečnímu značení místa. Stále se jedná o bipartitní[1] graf, takže hrany mohou spojovat pouze přechody s místy. Hrany jsou rozlišeny na tři druhy podle orientaci vůči přechodu[3].

- Vstupní hrana směřuje z místa do přechodu a tvoří vstupní podmínku pro přechod. Hranový výraz představuje značení, které je odebráno z místa pro provedení přechodu.
- Výstupní hrana směřuje naopak z přechodu do místa. Obdobně hranový výraz představuje značení, které se ale vloží do místa po provedení přechodu.
- Testovací hrana směřuje oběma směry. Testovací hrana je podobná vstupní hraně jen s rozdílem, že nepřemisťuje značení míst. Hrana pouze kontroluje přítomnost značení v propojeném místě.

[1] Prvky v grafu můžeme rozdělit na dvě disjunktní množiny, kde lze propojit pouze dva prvky z rozdílných množin.

2.4 Třídy a sítě OOPN

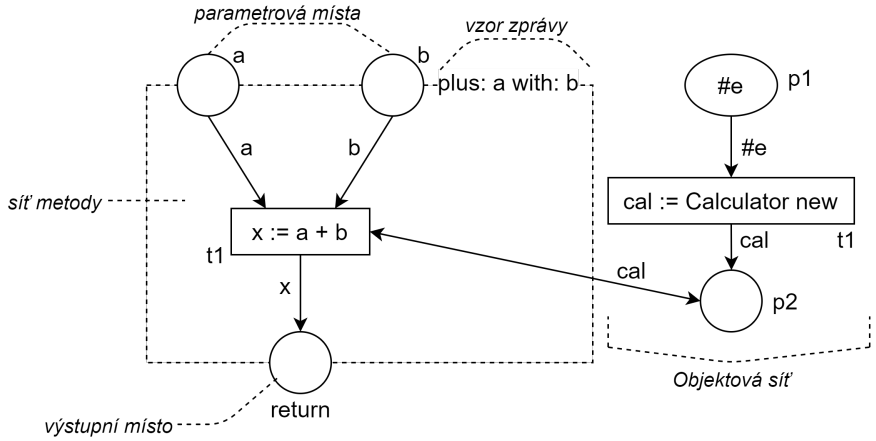
Podkapitola se zaměřuje na vyšší elementy v objektově orientovaném paradigmatu. Těmi jsou především třídy a objekty. Spolu s třídami jsou uvedeny principy dědičnosti v OOPN.

2.4.1 Sítě

Síť je tvořena hranami propojenými místy a přechody. Třída může obsahovat dva druhy sítí.

- Objektová síť představuje vnitřní stav objektu. Místa v objektové síti jsou atributy objektu. Třídy mohou obsahovat pouze jednu objektovou síť.
- Síť metody představuje vnitřní chování objektu. Třída může obsahovat více metod, které představují reakci objektu na zaslanou zprávu. Metoda se obdobně skládá z míst, přechodů a hran. Kromě základních prvků sítě musí metoda obsahovat *vzor zprávy*, který definuje selektor zprávy pro její zavolání. Kromě selektoru vzor obsahuje i parametry zprávy. Množina parametrů poté představuje odpovídající množinu *parametrových míst*, které slouží k jejich předání metodě. Jména parametrů musí odpovídat jménům parametrových míst. Každá metoda obsahuje jedno místo, které se jmenuje *return*. Toto místo vrací výsledek metody volajícímu objektu[3].

V rámci jedné třídy mohou být obě typy sítí propojeny pouze přes místa objektové sítě a přechody metod. Takto metody přistupují k vnitřnímu stavu objektu.



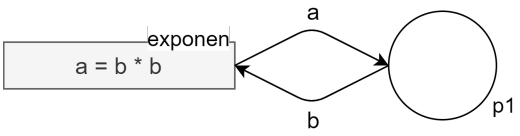
Obrázek 2.5: **Přechody**

Obrázek 2.5 zobrazuje metodu, která sčítá vstupní parametry a vrací jejích výsledek. Tato operace je provedena pouze pokud v objektové síti je ve stavu *p2* přítomna proměnna *cal* představující instanci třídy *Calculator*.

2.4.2 Synchronní porty a konstruktory

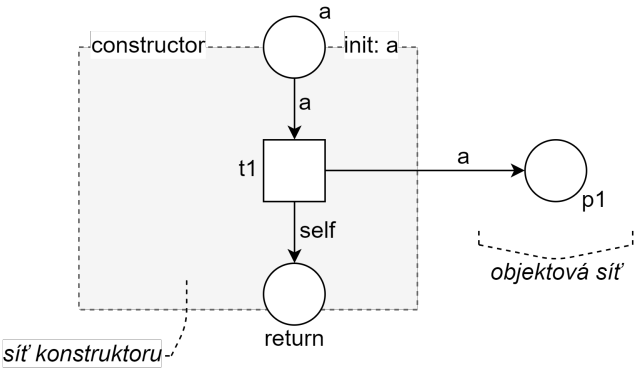
Kromě objektové sítě a metod může třída obsahovat také *synchronní porty*, které představují kombinaci metody a přechodu. Graficky jsou v PNtalku znázorněny jako přechody čtyřúhelníkem, ale také obsahují vzor zprávy, která má stejnou funkci jako u metod. Synchronní porty nemohou obsahovat místa ani přechody. Jediné, co může synchronní port

obsahovat, je stráž, která má stejnou syntaxi jak u přechodu a může tak měnit stav v objektové síti[3]. Obrázek 2.6 představuje příklad synchronního portu *exponen*, který provádí umocňování.



Obrázek 2.6: **Synchronní port**

Konstruktory, podobně jak v jiných objektově orientovaných programovacích jazycích, slouží k vytváření objektů. Konstruktory jsou implicitně součástí všech tříd a volají se pomocí zprávy *new*. Reakcí na zaslání této zprávy je inicializace objektové sítě třídy a jejího počátečního značení. V případě potřeby je možné přidat k této inicializaci metodu konstruktor s klíčovým slovem *constructor*, která může přijímat parametry při vytváření objektu a podle nich měnit jeho vnitřní stav. Od metod se liší také návratovou hodnotou. Konstruktor vždy vrací hodnotu *self*. V ostatních ohledech může konstruktor obsahovat stejnou vnitřní síť jako jiné metody. Příklad na obrázku 2.7 představuje jednoduchý konstruktor, který přijímá parametr a pak jej ukládá do místa *p1* v objektové síti dané třídy[3].



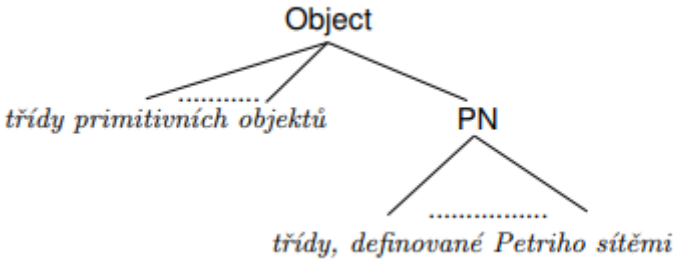
Obrázek 2.7: **Konstruktor**

2.4.3 Třídy

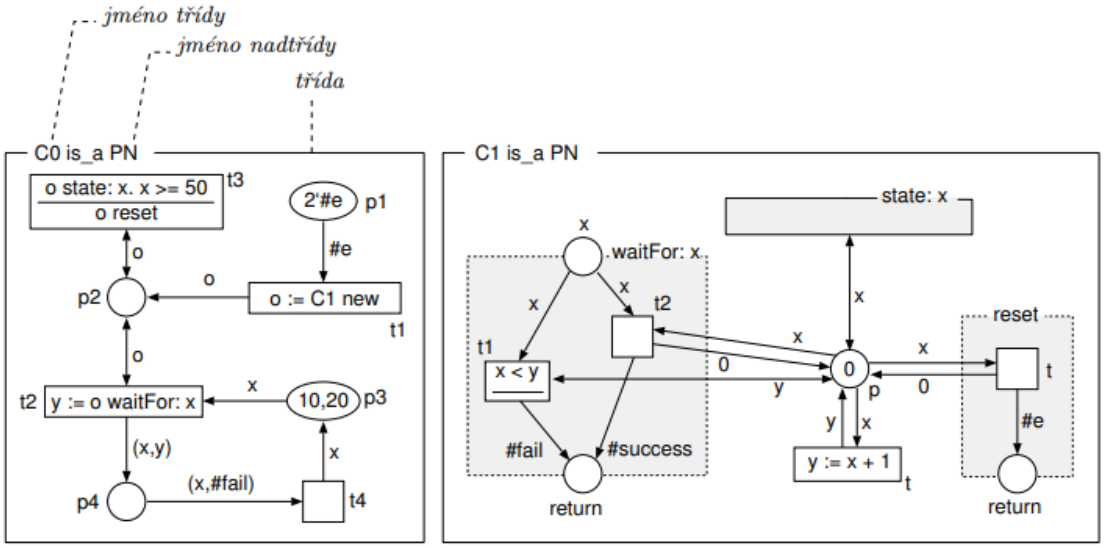
Základní hierarchie v PNtalku je tvořena pomocí tříd. V každém modelu musí existovat alespoň jedna třída s příznakem *main*, která je při jeho spuštění inicializována. Třída se skládá z maximálně jedné objektové sítě a množin metod, konstruktorů a synchronních portů. Každá třída musí mít jednoznačný identifikátor a označení svého předchůdce. Hierarchii tříd je možné si vytvořit libovolně, kromě jejího vrcholu. Ten je tvořen pomocí abstraktní třídy *PN* a jejím předchůdcem *Object*, který obsahuje všechny primitivní objekty[3]. Vrchol hierarchie popisuje obrázek 2.8.

V příkladě na obrázku 2.9 jsou definovány dvě třídy *C0* a *C1*. Obě třídy dědí z abstraktní třídy *PN*. Přičemž ve třídě C0 dochází k dvojí instanci třídy C1 v přechodu *t1*. Průběh výpočtu modelu se poté neustále cyklí.

Pokud třída dědí od nadtřídy, tak přejímá strukturu objektové sítě a všechny její metody, konstruktory a synchronní porty. Objektová síť může být předefinovaná po jednot-

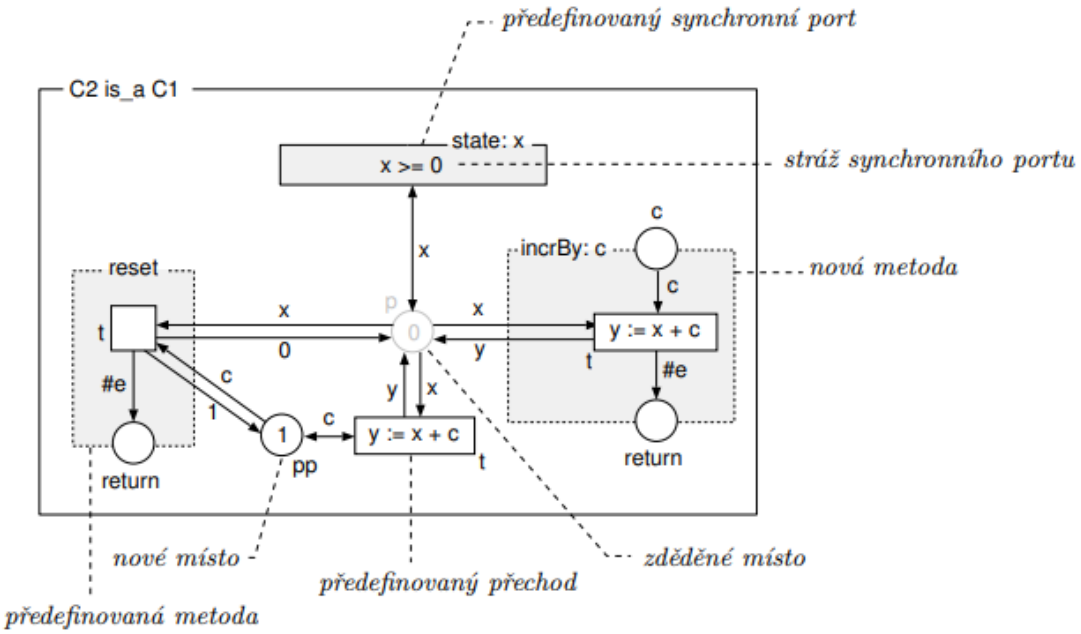


Obrázek 2.8: **Hierarchie dědičnosti tříd v PNtalku**. Převzato z [3]



Obrázek 2.9: **Dvě třídy, tvořící OOPN**. Převzato z [3]

livých místech nebo přechodech a to uvedením stejného jména jako v nadtřídě. Metody, konstruktory a synchronní porty lze předefinovat použitím nových definic pouze se stejným selektorem zpráv, jak u nadtřídy[3]. V příkladě na obr. 2.10 dochází k dědičnosti mezi třídami. Třída *C2* dědí od třídy *C1* z příkladu na obr. 2.9. Je zde vidět, jak lze předefinovat různé komponenty, které obsahuje třída. Také lze přidat nové místo *pp* jako z příkladu 2.10.



Obrázek 2.10: **Ilustrace dědičnosti a předefinování zděděných prvků třídy C1 z obr. 2.9**. Převzato z [3]

Kapitola 3

Specifikace požadavků a návrh aplikace

Proč vlastně děláme specifikaci a návrh aplikace? Před samotným vývojem softwaru je potřeba si ujasnit několik základních myšlenek a odhalit kritická místa pro průběh implementace. Při specifikaci se musíme zaměřit na několik klíčových bodů.

- Ujasnit si co přesně bude program dělat. Mít představu o potřebách uživatele a jeho přístupu k aplikaci.
- Vědět, s jakými daty bude aplikace pracovat. Předpokládat, která data bude mít aplikace k dispozici, případně které data jí uživatel poskytne.
- Předpokládat využití vstupních dat a jejich práci aplikace s nimi. Lze také předpokládat práci uživatele s daty v rámci aplikace.
- Vědět, jaké výstupní data bude aplikace pro uživatele poskytovat. Z těchto výstupních dat zjistit, zda budou požadované výsledky relevantní pro uživatele.

Při analýze se snažíme naleznout nejvhodnější možný způsob a metodu řešení našeho zadání. Analýza může být několika druhů, které se mohou zaměřovat na různé aspekty vývoje. Je potřeba si vybrat relevantní druhy analýz, které se zaměřují na předpokládané řešení. Z analýzy poté vyplývá návrh řešení, který poslouží k ujasnění dílčích cílů pro vývoj aplikace. S návrhem řešení je spojeno několik diagramů, které slouží k získání nadhledu při implementaci.

3.1 Neformální specifikace

Je potřeba vytvořit aplikaci, která bude umět vytvářet a editovat objektově orientované Petriho sítě implementované v jazyku PNtalk. Vytvořené Petriho sítě jsou zobrazeny po třídách a spolu dohromady musí tvořit samostatné modely, které aplikace musí umět načíst a opět uložit. V ideálním případě by byly zobrazeny třídy se všemi komponenty objektové sítě včetně metod, synchronních portů a konstruktorů.

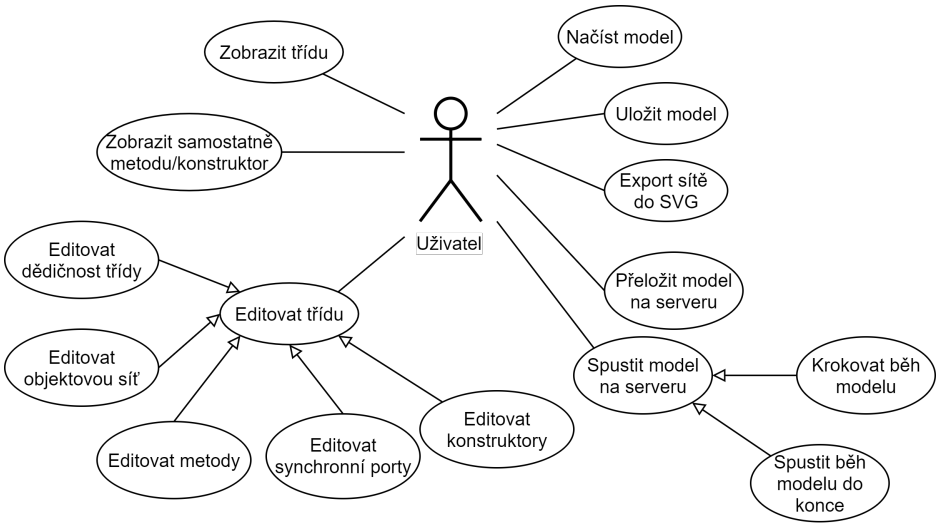
Aplikace musí umět jednoduše měnit jednotlivé komponenty sítě dle uživatelských požadavků. Kromě možnosti měnit vlastnosti přechodu, míst a hran je potřeba i měnit jejich rozložení v sítí, které ji činí přehlednou. Vhodné je učinit i metody, synchronní porty a

konstruktory snadno přemístitelné. Kromě zobrazení tříd se všemi komponenty je potřeba dovolit i samostatné zobrazení metod a konstruktorů.

Pro práci se serverem je potřeba mu odeslat všechny třídy v aktuálním modelu a poté model spustit. Server komunikuje na určité adrese a portu. Data mu jsou posílaná v jazyce PNtalku. Ve stejném jazyce pak data vrací zpět. Server sám o sobě reaguje na několik druhů příkazů tak, aby si uživatel mohl vybrat jestli chce nechat dojet síť do konce nebo si krokovat jednotlivé kroky své sítě.

3.2 Diagram případu užití

Aplikace nepředpokládá žádný velký koncept rozdílných uživatelů, kteří budou aplikaci používat. Proto v případu užití figuruje pouze jeden základní uživatel, který nemá omezené práva a může aplikaci užívat v plném rozsahu. Diagram nepředstavuje všechny podrobné operace, které může uživatel používat. Některé operace jsou abstraktní, pod kterými se skrývá více dílčích operací. Příkladem jsou editace různých sítí, které zahrnují více menších operací. Diagram případu užití je možné vidět na obr. 3.1.



Obrázek 3.1: **Diagram případu užití**

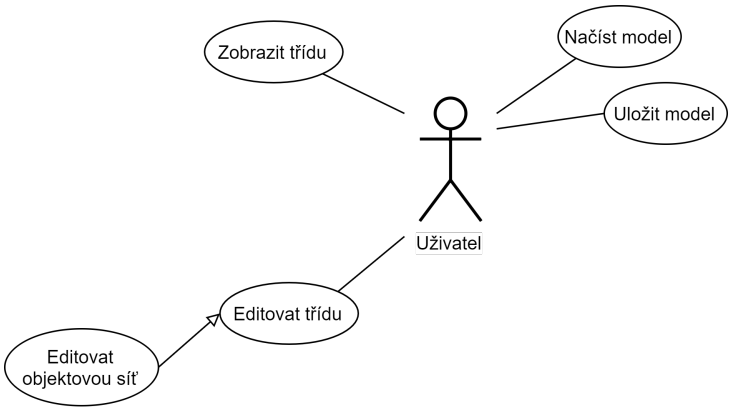
3.2.1 Plán projektu

Z prvotní analýzy vyplývá, že aplikace bude mít hodně funkcionalit, z niž některé spolu souvisí. Tyto funkcionality jdou rozdělit do několika skupin, které mají podobné chování. Je tedy vhodné rozdělit vývoj do několika iterací, které budou postupně uzavírat část funkcionality aplikace. Je vhodné rozdělit vývoj do tří oddělených iterací tak, aby první iterace měla již minimální užití, které se dá prezentovat.

První iterace

První iterace je zaměřena na minimální funkčnost. Diagram užití je vykreslen na obr. 3.2. I když se může zdát, že první iterace toho uživateli příliš nenabízí, tak opak je pravdou. V první iteraci je zahrnuto nejdůležitější téma, práce s Petriho sítí. Samotná implementace

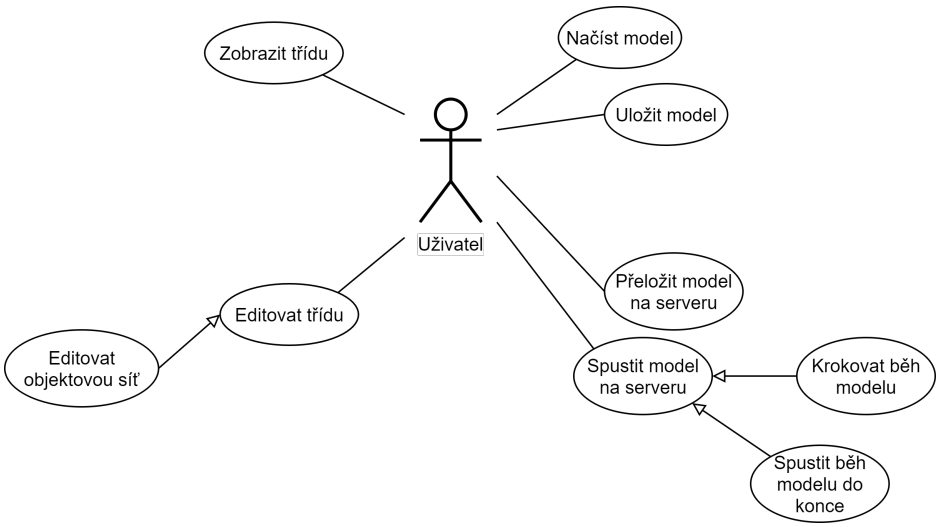
editace je poměrně náročná. Práce s Petriho sítěmi je v tomto ohledu naprosto klíčová. Jestliže je v první iteraci zahrnuta práce se sítěmi, tak zákonitě musí být vytvořené i grafické uživatelské rozhraní. Je zde také požadavek na uložení hierarchie tříd do textové podoby a opětovné načtení. Tento fakt určuje potřebu mít vytvořené pozadí aplikace tak, aby vstupní a výstupní data měli strukturu do které se mohou nahrát. Je tedy zřejmé, že první iterace položí základní kameny k tvorbě funkcionalit pro ostatní iterace.



Obrázek 3.2: **Diagram případu užití 1. iterace**

Druhá iterace

V druhé iteraci je aplikace obohacena o práci se serverem. Aplikace v první řadě musí poslat všechny nahrané třídy na server tak, aby byl server schopen je přeložit a později spustit. I když se jeví odeslání dat na server jednoduše, není tomu tak. Je potřeba vnitřní interpretaci dat z pozadí aplikace převést do jazyku PNtalk. Poté se výsledný text PNtalku pošle na server. Pokud jej server příjme bez chyb, pak může uživatel spustit daný model. Diagram užití je zobrazen na obr. 3.3.



Obrázek 3.3: **Diagram případu užití 2. iterace**

Třetí iterace

Třetí a zároveň poslední iterace má za úkol doladění zbytku aplikace. Dochází zde především k doladění práce se všemi komponenty, které obsahují třídy. Důležité je poté dořešit samostatné i společné zobrazení těchto komponent vzhledem k objektové síti dané třídy. Tento proces nemusí být tolik složitý, jelikož základy pro něj by měli být podloženy a mělo by být potřeba jen nastavit správné zobrazení. Zároveň je potřeba doladit detaily ohledně zobrazení hierarchie tříd tak, aby byly uživateli dostatečně přehledné. Posledním bodem je tvorba exportu grafické podoby sítě do SVG vektorové grafiky. Výsledný diagram užití je k dispozici na obr. 3.1.

3.3 Architektonický návrh aplikace

Po ujasnění specifikací a vytvoření plánu vývoje je potřeba si vytvořit architektonický návrh aplikace. Plán vývoje 3.2.1 určuje postup, kterým je potřeba se řídit a pro dekomponované problémy vytvořit návrh jejich řešení. Dekomponované problémy vedou k tvorbě dílčích podsystémů, které je řeší. Jednotlivé podsystémy musí být řádně definovány včetně jejich společných vztahů. Vytvoření podrobného návrhu může do budoucna ušetřit zbytečné problémy, které by mohly nastat.

3.3.1 Model životního cyklu

Již ve specifikaci byl určen druh vývoje softwaru po několika iterací. Je tedy vhodné si vybrat iterativní model životního cyklu softwaru. Při vývoji celkem dojde ke třem iteracím, ve kterých se opakují specifikace, návrh, implementace a testování. Iterativní model životního cyklu softwaru rychleji poskytuje hotové částečné řešení, které pomůže postupně rozvíjet funkcionalitu aplikace dle plánu projektu[11].

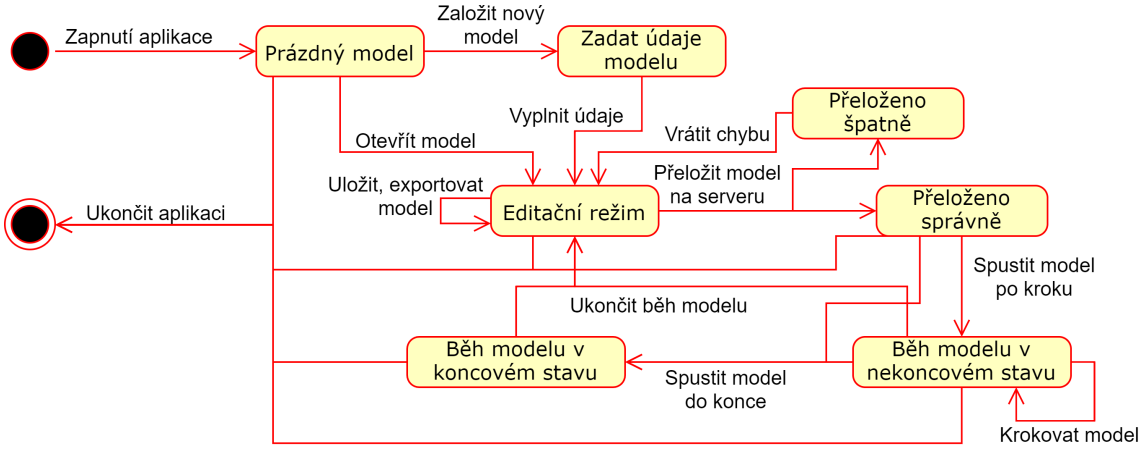
3.3.2 Diagram stavů

Stavový diagram je další grafický způsob zápisu systému. V tomto případě je diagram použit pro upřesnění všech možných stavů a akcí s nimi spojených. Pomocí tohoto diagramu je možné odhadnout chování uživatele při používaní aplikace. Rovněž je možné použít diagram k pozdějšímu rozvinutí pro modelování podrobnějších operací. Obr. 3.4 představuje diagram stavů pro touto prací popisovanou aplikaci. Je takto snadno vidět veškeré stavy, ve kterých se může aplikace nacházet, a které operace mohou být pro dané stavy použity. Téměř ve všech stavech lze ukončit aplikaci. *Editační režim* se nazývá klíčový stav, ve kterém se odehrává většina operací spojená s modelováním Petriho sítí. Tento stav může sloužit k rozvinutí pro samostatný stavový diagram, který se zaměří na podrobnější chování aplikace v tomto režimu.

3.3.3 Architektura MVC

Dalším krokem je zaměřit se na podrobnou strukturu aplikace, ze které může vycházet diagram tříd. Pro popis struktury aplikace byl vybrán typ architektury MVC.[1] Tento architektonický model rozděluje aplikaci do třech nezávislých komponent tak, aby změna v jedné z nich neměla velký vliv na ostatní. Zároveň rozdělení modelu do tří komponent odpovídá přibližnému rozložení v technologii *Java*, která byla určena pro vývoj aplikace v zadání

[1] MVC zkratka pro Model-View-Controller



Obrázek 3.4: **Diagram stavů aplikace**

diplomové práce. Tyto vlastnosti jsou hlavním důvodem výběru právě tohoto modelu pro aplikaci. Architektura aplikace je zobrazena na obr. 3.5.

3.3.4 Diagram tříd

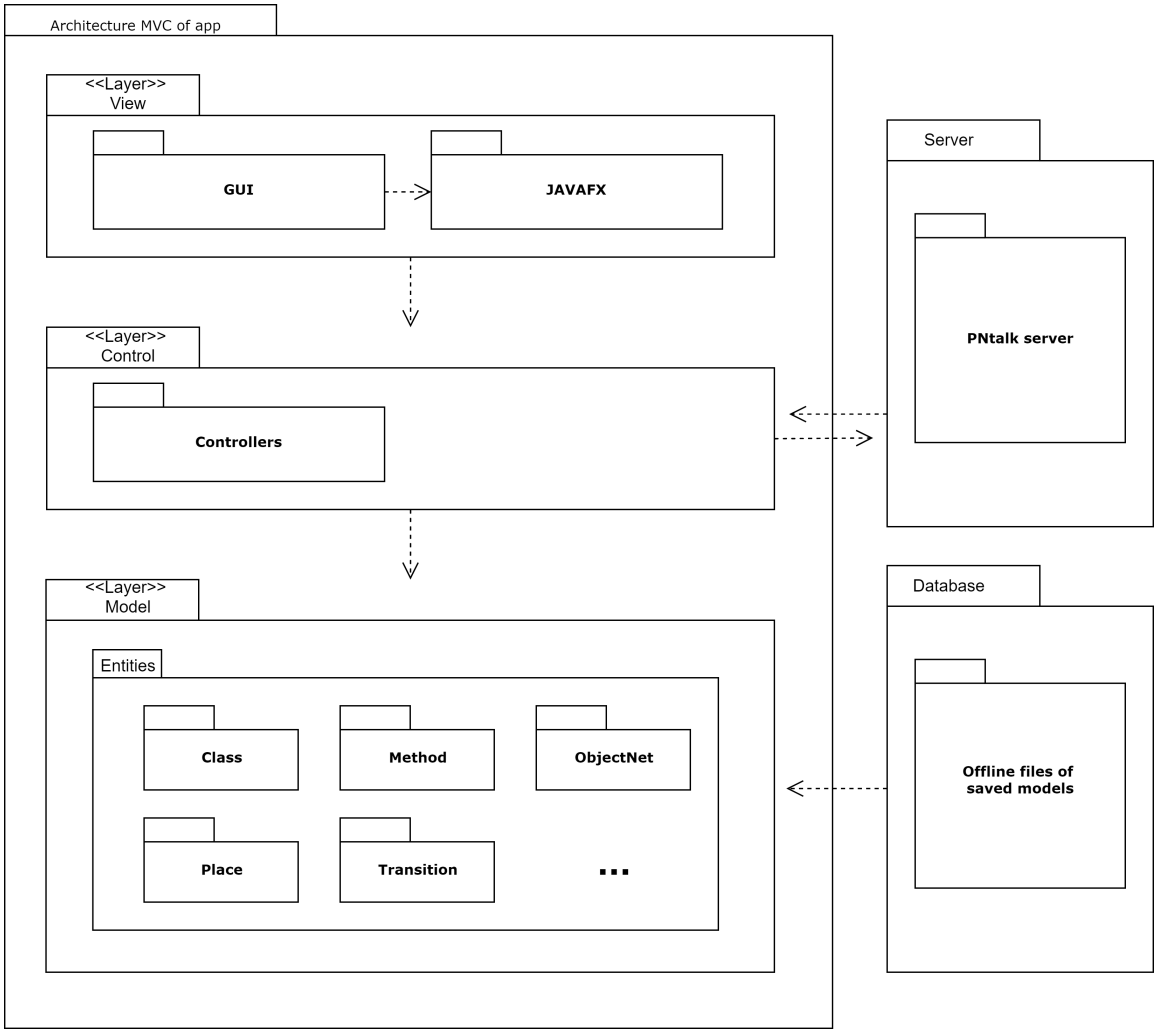
Dalším nástrojem pro upřesnění návrhu struktury aplikace je diagram tříd. Tento diagram popisuje statickou strukturu aplikace. Struktura je zobrazena po třídách, které mohou mít mezi sebou vztahy. Každá třída je zobrazena s atributy třídy a metodami.

Nemá smysl, aby diagram tříd mapoval celou architekturu z obr. 3.5. Klíčovým faktorem jsou zde *Entities*, které obsahují v podstatě celé pozadí aplikace. Proto diagram tříd z obr. 3.9 zobrazuje celou strukturu pozadí aplikace, která má za úkol pracovat s daty, které popisují objektově orientované Petriho sítě v PNtalku.

3.3.5 Grafické uživatelské rozhraní

Jelikož nemá velký význam popisovat vrstvu řadiče z MVC architektury na obr. 3.5, tak poslední popisovanou vrstvou se stává *View*. Tato vrstva se stará o grafické uživatelské rozhraní a její vstup je poslán do řadiče, který vyvolá příslušnou reakci v aplikaci. Je nezbytné vytvořit grafický návrh tohoto rozhraní. Při tvorbě je nutné dbát na jednoduchost a přehlednost tak, aby uživatel neměl problém s orientací v aplikaci. Pro případ aplikace je vytvořen grafický návrh na obr. 3.6. Aplikace nabízí pouze jedno rozložení, které zůstane téměř stejné po celou dobu běhu aplikace.

Grafické rozhraní se skládá z několika ovládacích panelů. Prvním z nich je *Menu* panel, který ovládá celou aplikaci. Zde bude například otevíraní a ukládání modelů, nastavení, ukončení aplikace atd. Na stejné úrovni v opačném rohu je panel pro ovládání komunikace aplikace se PNtalk serverem. Na levé straně rozhraní jsou k dispozici vyjíždějící okna, která znázorňují strukturu aktuálně nahraného modelu. Podél menu panelu se táhne panel pro záložky. Každá záložka představuje, podobně jak v internetovém prohlížeči, jedno aktuálně otevřené zobrazení sítě. Samotná síť je zobrazena v největším okně v celém rozhraním, které se nazývá *Work space*. Toto okno představuje pracovní plochu, ve které se zobrazují Petriho sítě a ve kterém jsou editovány. Vpravo od pracovní plochy je okno s názvem *Tool box*. Okno obsahuje veškeré elementy objektově orientovaných Petriho sítí, které může uživatel



Obrázek 3.5: **MVC architektura aplikace**

přidat do své sítě zobrazené na pracovní ploše. Posledním panelem je *Event Log*. Tento panel zobrazuje instrukce pro uživatele. Těmito instrukcemi mohou být například zprávy o provedeném uložení modelu, nahráním modelu na server, případně chybové hlášky atd.

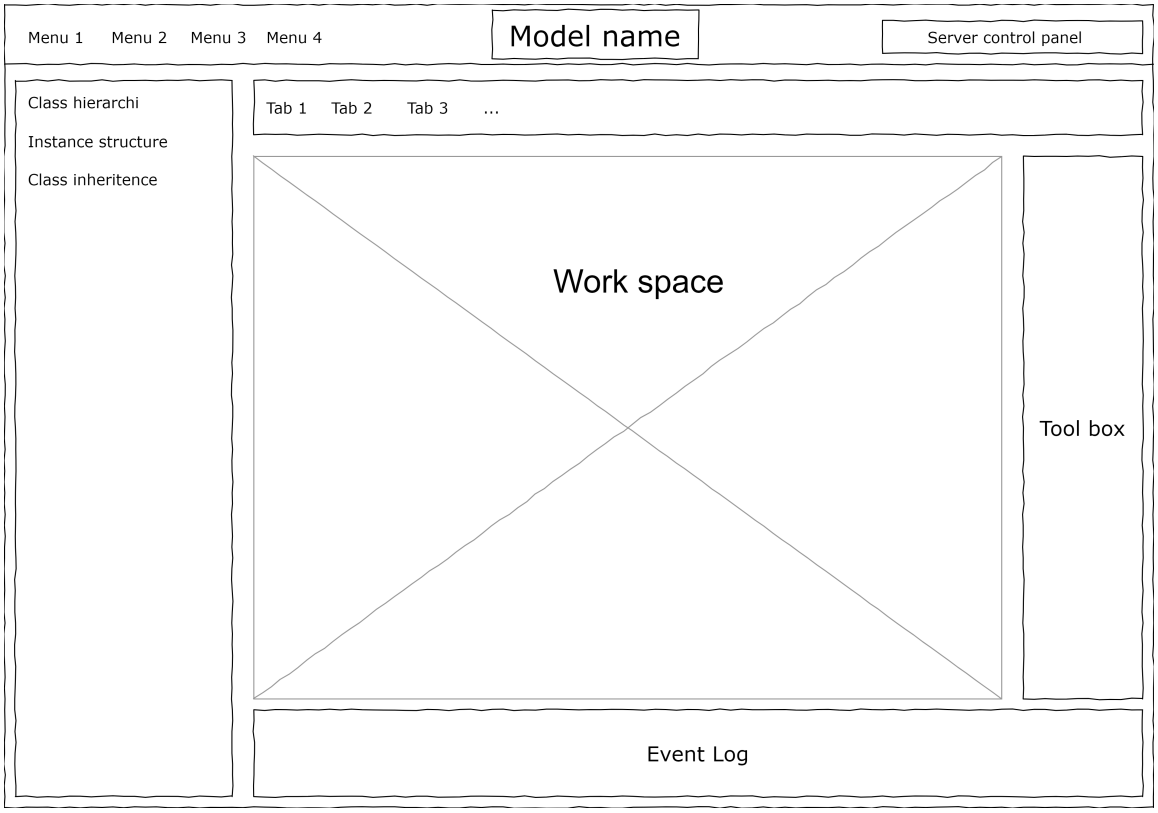
3.4 Jazyk PNML pro uložení dat

Jazyk *Petri Net Markup Language* zkráceně PNML slouží pro textový zápis Petriho sítí. Jazyk je založen na podobném principu značkovacích jazyků jako XML.[2] Jeho hlavními výhodami jsou především přenositelnost a rozšiřitelnost. Právě rozšiřitelnost hraje roli v případě využití PNML v aplikaci, která je předmětem této práce. Aplikace musí umět své modely uložit do modifikovaného zápisu PNML, který si lze přizpůsobit podle své potřeby.

3.4.1 Vlastnosti PNML

Jazyk PNML se opírá především o následující tři základní vlastnosti.

[2]XML(eXtensible Markup Language) je rozšířený jazyk, který se používá k serializaci dat.



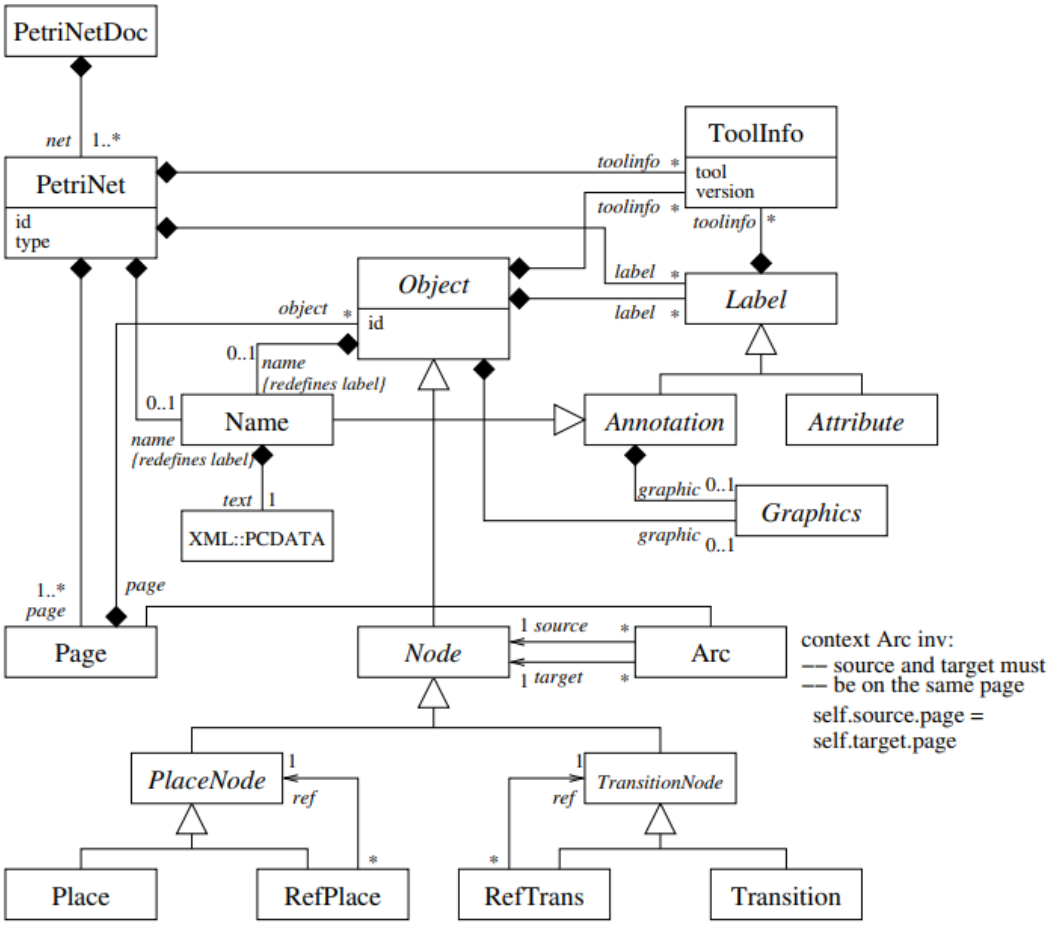
Obrázek 3.6: **Návrh grafického uživatelského rozhraní**

- Flexibilita v kontextu PNML znamená schopnost reprezentovat jakýkoliv druh Petriho sítí se svými specifickými rozšířeními a vlastnostmi. PNML by nemělo omezovat vlastnosti některých druhů Petriho sítí, ani je nutit, aby se při jejich převodu na PNML ignorovaly nebo abstrahovaly určité informace Petriho sítí[1].
- Nejednoznačnost je z formátu odstraněna zajištěním toho, že původní Petriho síť a její konkrétní druh lze jednoznačně určit z její reprezentace PNML. Za tímto účelem PNML podporuje definici různých typů Petriho sítí. Definice typu Petriho sítě (PNTD) určují štítky pro konkrétní typ Petriho sítě. Přiřazením fixního typu ke každé Petriho síti bude popis jednoznačný[1].
- Kompatibilita znamená, že mezi různými typy Petriho sítí lze vyměňovat co nejvíce informací. Za účelem dosažení kompatibility přichází PNML s konvencemi o tom, jak definovat štítek se zvláštním významem. V konvenčním dokumentu je předefinována syntaxe i zamýšlený význam všech druhů rozšíření. Při definování nového typu Petriho sítě mohou být štítky vybrány z tohoto konvenčního dokumentu[1].

3.4.2 Struktura PNML

Různé části PNML a jejich vztahy jsou znázorněny na obr. 3.7. Meta model definuje základní strukturu souboru PNML, rozhraní pro definici typu umožňující definici nových typů Petriho sítí, které omezují legální soubory meta modelu a rozhraní pro definici funkcí umožňujících definovat nové funkce pro Petriho sítě. Tyto tři části jsou trvale zafixovány. Další část PNML a Konvenční dokument se vyvíjí. Model obsahuje definici sady standardních

funkcí Petriho sítí, které jsou definovány podle rozhraní pro definici prvků. Navíc bude existovat několik standardních typů Petriho sítí, které používají některé funkce z Konvenčního dokumentu a případně i jiné. Pokud jsou ve společném zájmu, mohou být do konvenčního dokumentu přidány nové funkce a nové typy Petriho sítě ke standardním typům. Tyto dokumenty jsou díky své vyvíjející se povaze nejlépe publikovány a udržovány prostřednictvím webové stránky[1].



Obrázek 3.7: **PNML Core Model**. Převzato z [7]

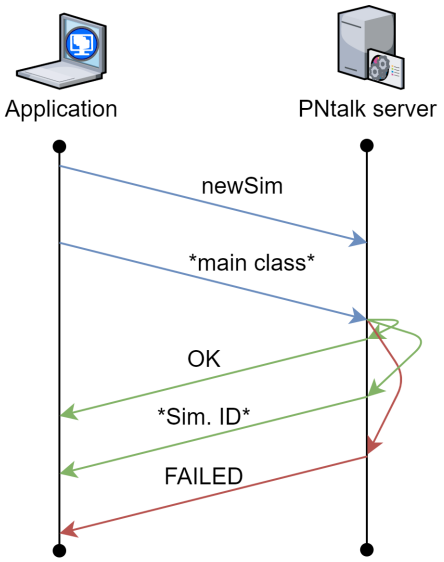
3.5 Komunikace s PNtalk serverem

Aplikace sama o sobě nedokáže provádět výpočty spojené se simulací OOPN. Z tohoto důvodu je nutné zajistit komunikace se serverem, který dokáže tyto simulace provádět. PNtalk server je již vytvořený software, který funguje na principu spuštění v prostředí aplikace *Pharo*.[3] V principu aplikace počítá se zásahem uživatele, který si na stejném zařízení PNtalk server otevře v prostředí Pharo a spustí jej.

[3]Pharo je open source implementace objektově orientovaného programovacího jazyka a prostředí Smalltalk.

Komunikace se serverem může probíhat dvěma způsoby. Uživatel má možnost s aktuálním stavem na serveru manipulovat pomocí prostředí Pharo. Prostředí v aplikaci Pharo poskytuje uživateli veškerou potřebnou funkcionalitu pro práci s OOPN. Bohužel, ale toto prostředí není uživatelsky přívětivé. Tento problém si dává za úkol vyřešit aplikace, která je cílem této diplomové práce. Aplikace musí komunikovat se serverem přes lokální adresy na daném zařízení. Komunikační protokol s PNtalk serverem je jasně definovaný. V obrázcích B.1, B.2 a B.3 v příloze je rozebrán celý komunikační protokol, který je pro účely aplikace klíčový. Po jednotlivém komunikačním procesu se spojení se serverem ukončí a pro další komunikaci je potřeba se k serveru znova připojit.

Obrázek 3.8 představuje konkrétní případ pro vytvoření nové simulace na serveru. Po inicializaci komunikačního kanálu aplikace pošle příkaz *newSim*, který upozorní server na potřebu vytvoření nové simulace. V následující zprávě aplikace sděluje serveru název hlavní třídy, odkud započne simulace. Server může odpovědět pouze dvěma způsoby. Pokud vše na server projde v pořádku, pak pošle aplikaci odpověď *OK* a v následující zprávě je číslo, které slouží jako unikátní identifikátor pro novou simulaci. Aplikace pak může pracovat se simulací pomocí identifikátoru, který se na ní odkazuje. Druhý způsob odpovědi je *FAILED*, který oznamuje aplikaci, že pokus o vytvoření simulace proběhl neúspěšně.



Obrázek 3.8: **Komunikační protokol první část**

3.6 SVG

Značkovací jazyk a zároveň formát souborů SVG[4] slouží k ukládání informací o grafické reprezentaci objektů. Spolu s formáty PDF, EPS, SVG a dalšími spadá do vektorové grafiky. Na rozdíl od rastrové grafiky, kde je obrázek popsán pomocí hodnot jednotlivých pixelů, SVG popisuje obrázek skrze přesně definované 2D objekty, ze kterých se obrázek skládá. Mezi tyto objekty patří body, křivky, mnohoúhelníky apod. Textová reprezentace těchto útvarů je v SVG standardu provedena prostřednictvím jednotlivých značek, které jsou defi-

[4]SVG(Scale Vector Graphics) standard pocházející z roku 2001.

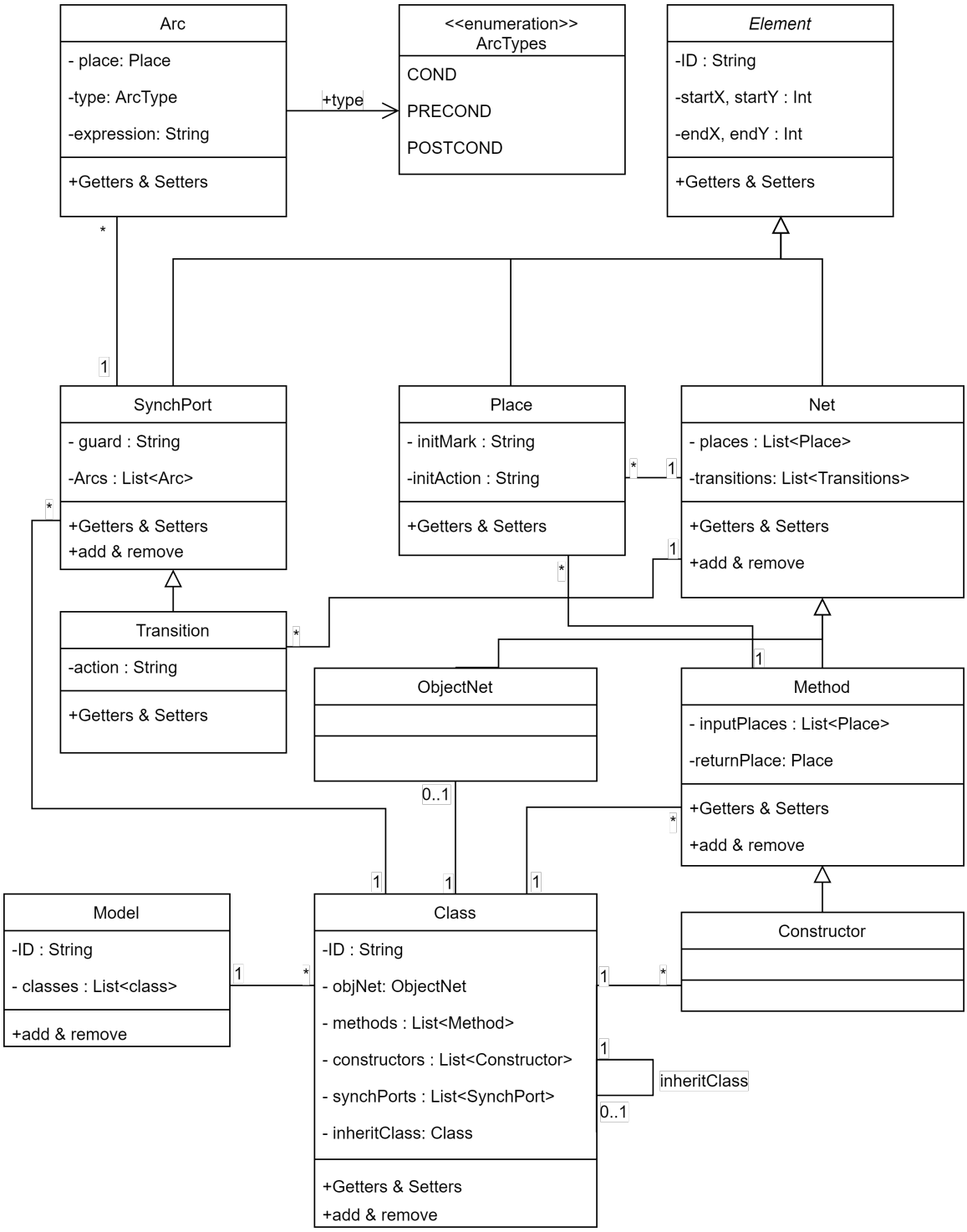
novány. Dílčím úkolem vytvářené aplikace je exportovat grafickou podobu OO Petriho sítí. Tento export musí být právě ve formátu SVG.

3.6.1 Vlastnosti SVG

Největší výhodou SVG formátu je jeho nezávislost na rozlišení a device-pixel-ratio.[5] Zároveň se dá snadno editovat nebo generovat, jelikož je obrázek v podstatě XML kód. Lze využít rovněž jeho dynamickou editovatelnost pomocí CSS[6] a JavaScriptu. I přes vysokou kvalitu obrázku, není zde nárok na vysoký objem dat. Z neposledních výhod je rovněž možnost idexovatelnosti včetně roboty Googlu[9].

Naproti tomu samotné pořízení obrázku je složitější než v rastrové grafice, kde stačí pořídit fotografii nebo sken. Zároveň SVG není vhodný pro zápis složitějších barevných ploch, kterými jsou právě například fotografie. Překročí-li grafický objekt určitou složitost, stane se náročnější na procesor a operační paměť.

[5] Device-pixel-ration je poměr mezi fyzickými a logickými pixely.
[6] CSS(Cascading Style sheet) jazyk pro stylistický popis elementů v jazycích HTML, XHTML a XML



Obrázek 3.9: **Diagram tříd pro backend**

Kapitola 4

Implementace a testování

Tato kapitola popisuje implementaci cílové aplikace, jejíž návrh byl popsán v kapitole 3. Pro implementaci byl použit framework JavaFX,[1] který je postavený na bázi platformy Javy. Kromě jazyku Java pracuje aplikace rovněž s XML a PNtalk. Oba tyto podpůrné jazyky slouží pro komunikaci aplikace s externímy zdroji.

4.1 Backend

V prvních fázích implementace byla naprogramována backend aplikace, která tvoří základní kostru aplikace. Na obrázku 3.9 je zobrazena základní struktura backendu aplikace. Oproti původnímu návrhu jsou do backendu aplikace přidány další třídy, které doplňují aplikaci o klíčové funkce. Největší podíl tvoří skupiny tříd, které zaštiťují komunikaci s PNtalk serverem. Další pak pomáhají ke konverzi vnitřní struktury modelu na externí přepis, který se používá k ukládání, exportování a překládání na serveru.

4.1.1 Model

Model je obalující prvek, se kterým aplikace pracuje. Aplikace může vždy pracovat v daném okamžiku pouze s jedním modelem. Jeho backendová reprezentace je třída, která obsahuje název modelu, kolekci PN tříd,[2] označení hlavní třídy a další proměnné pro mapování průběhu práce aplikace s modelem. Celkovou strukturu modelu aplikace lze rozdělit do třech základních konstrukčních prvků. Prvním z nich jsou již zmíněné PN třídy. Druhým prvekem jsou obecné sítě, které dále můžeme rozlišit na metodové, konstrukční a objektové. Posledním prvkem jsou definované elementy, které může síť obsahovat. Každý z těchto prvků je implementován vlastní třídou, která zapouzdří veškeré jeho vlastnosti a chování. Následující popis začíná u jednodušších prvků, které postupně umožňují rozvíjet popis celkové struktury, která končí až u modelu.

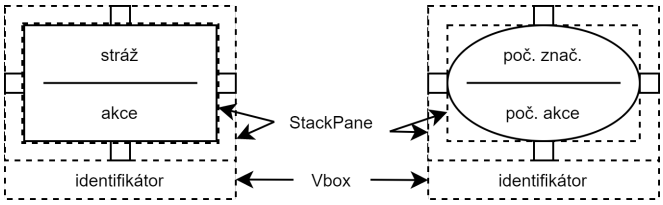
4.1.2 Místa a přechody

V Petriho sítích místa a přechody tvoří uzly bipartitního grafu. Lze je rozlišit dle tvaru jejich ohraničení. Místa jsou vytvořeny tak, aby jejich ohraničení tvořila elipsa. Naproti tomu přechod je ohraničen obdélníkem. Místa i přechody obsahují dělící línii, která rozděluje

[1] JavaFX je moderní framework používaný pro tvorbu okenních aplikací založený na jazyce Java.

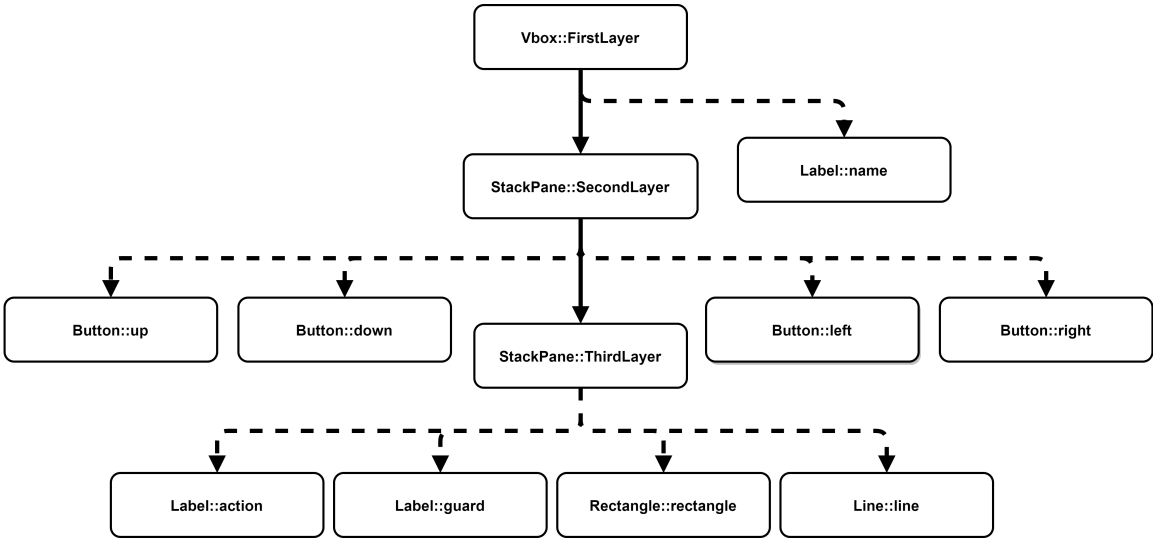
[2] PN třída je označení třídy v objektově orientovaných Petriho sítí, které slouží k rozlišení tříd v jazyce Java a tříd v jazyce PNtalk.

jejich obsah na horní a spodní polovinu. Místo obsahuje v horní polovině počáteční značení a ve spodní počáteční akci. Na rozdíl od toho, má přechod v horní polovině stráž a ve spodní akci. Pod oběma elementy se vždy nachází jejich jednoznačný identifikátor.



Obrázek 4.1: **Struktura elementů**

Struktura místa a přechodu je zobrazena na obrázku 4.1. Oba elementy mají stejný základ tvořený z kontejneru označeného v Javě *VBox*.VBox kontejner všechny své grafické objekty skládá vertikálně za sebou. VBox místa i přechodu se používá pro práci s celým elementem. Například, pokud uživatel chce daný element v síti přesunout, tak právě VBox má na sobě nastavený *action listener*,[3] který uživateli umožňuje pohybovat s VBoxem a tudíž i s všemi jeho vnořenými prvky.



Obrázek 4.2: **Hierarchie grafických objektů místa a přechodu**

Celková hierarchie grafických objektů přechodu je zobrazena na obrázku 4.2. Hierarchii grafických objektů pro místo je téměř shodná, pouze obsahující jiné názvy. Nejvýše v hierarchii je postaven VBox. VBox obsahuje vždy pouze další dva grafické objekty. Prvním je *Label*,[4] který zobrazuje název daného elementu a je na něm nastavené *kontext menu*.[5] Druhým objektem je *StackPane*,[6] který zajišťuje připojovací body pro daný element. Tento StackPane druhé vrstvy obsahuje čtyři tlačítka. Tyto tlačítka představují malý čtverečky

[3]Action listener je metoda, která se zavolá v případě provedení nastavené akce, kterou může provést uživatel nebo program sám.

[4]Label je grafický objekt v jazyce Java, který se nejčastěji používa pro zobrazení určité informace například textu nebo obrázku

[5]Kontext menu ve frameworku JavaFX představuje vyskakovací menu, které se zobrazí po kliknutím pravého tlačítka myši na určený grafický objekt

[6]StackPane představuje ve frameworku JavaFX kontejner, kde se může skládat grafické objekty na sebe.

na obrázku 4.1, které jsou ve výchozím stavu skryty. V případě, kdy uživatel chce přidat hranu spojující dva elementy, se tyto tlačítka zobrazí a jakmile se provede spojení případně zrušení, tak jsou opět schovány. Další vnořený StackPane představuje třetí vrstvu elementu. Ve třetí vrstvě jsou uloženy dva labely, které představují vnitřní informace pro daný typ elementu. Také obsahuje dělící línii, která odděluje tyto dva labely a obdélník, který je obaluje. Celkově element obsahuje tři labely, kde každý label má přiřazen vlastní kontextové menu. Všechny tyto meny obsahují akci pro změnu dané informace, kterou label představuje. Jediný label *name* obsahuje v kontextovém menu akci pro smazání celého elementu. Kromě těchto možných akcí, které uživatel může provádět s elementem, je zde přidán action listener pro zvýraznění objektu. Pokaždé když uživatel najede s kurzorem na objekt, dojde k jeho obarvení. Při opětovném vyjetí kurzoru dojde k zrušení zvýraznění.

4.1.3 Synchronní port

Synchronní port obsahuje velmi podobnou strukturu jako přechod. Jednotlivé vrstvy, které jsou uvedeny pro přechod, synchronní port také obsahuje. Hlavním rozdílem synchronního portu je absence labelu, který představuje akci přechodu. Stráž je zobrazena opět labelem a dělící línie je posunuta níže. Obdélník, který představuje hranu synchronního portu, má lehce šedé pozadí pro jednodušší rozlišení od přechodu.

4.1.4 Hrany

Pro objektově orientované Petriho sítě rozlišujeme tři typy hran. *Precond* hrana směřuje z místa do přechodu a *postcond* opačně. Poslední speciální typ hrany je *cond*, který se také nazýva testovací hrana. Precond a postcond jsou zobrazeny šipkou mířící mezi elementy. Cond je zobrazen obousměrnou šipkou. Při přidání hrany do sítě, výchozí struktura se sestává ze tří hlavních části. První částí je label, který zobrazuje její hranový výraz. Tento label je přesouvatelný a tedy nezávislý na ostatní části. Je tedy na uživateli kam si ho umístí. Další částí je množina menších úseček, které jsou pospojovány tak, aby vytvořily požadovanou šipku. K nastavení přehlednosti v grafu je poslední částí množina kontrolních bodu, pomocí kterých uživatel může upravit cestu hrany mezi elementy. Kontrolní body jsou tvořeny labely ve tvaru čtverce s černou výplní. Hrana ve výchozím stavu obsahuje pouze jeden kontrolní bod. Avšak každému labelu je přidáno kontextové menu, které dovoluje odstraňovat a opět přidávat kontrolní body. Label, který představuje hranový výraz, také obsahuje kontextové menu. Toto kontextové menu umožňuje odstranit celou hranu, případně změnit hranový výraz. Na všechny části hrany je vytvořen action listener, který po najetí myší obarví celou hranu na červenou barvu. Hrana si vždy pamatuje připojené dvě tlačítka a jejich polohu. V případě přemístění připojeného elementu se vždy provede celé přepočítaní hrany tak, aby i po přesunutí šipka směrovala do správného místa.

4.1.5 Sítě

V backendu jsou sítě vytvořeny tak, aby vždy obsahovaly množinu míst a přechodů. Sítě nemusí obsahovat množinu hran, jelikož každý přechod případně synchronní port si uchovává všechny připojené hrany. Další vlastnosti sítí poté závisí na jejich typu. V PNtalku existují tři typy sítí. První typem je objektová síť. Tato síť jako jediná může obsahovat synchronní porty. Dalším typem je metoda. Na rozdíl od objektové sítě metoda obsahuje zprávu, kterou je volána. Rovněž obsahuje speciální množinu vstupních míst případně portů, které slouží pro přijetí parametrů metody. Poslední specialitou je místo *return*, které slouží pro

návratovou hodnotu. Posledním typem sítě je konstruktor, který může být volán v případě instance třídy. Ve všech ostatních ohledech je téměř identický s metodou.

4.1.6 Třídy

Každá třída v backendu může obsahovat objektovou síť. Je pouze na uživatelově rozhodnutí zda třída bude obsahovat objektovou síť. Dále třída obsahuje množinu metod a konstruktorů. Všechny tyto sítě obsahují zpětný odkaz na třídu, ve které jsou obsaženy. Každá třída vlastní atribut, který se odkazuje na otcovskou třídu. V případě, že tento odkaz je nulový, pak je otcovská třída PN. Všechny tyto třídy jsou obsaženy v modelu. Hierarchii dědičnosti si řeší třídy mezi sebou.

4.2 Frontend

V podkapitole 4.1 je lehce přiblížen vzhled a struktura jednotlivých elementů, které se mohou vyskytovat v síti. V následující podkapitole budou upřesněny části, které tvoří celý frontend. V případě frameworku je frontend napsán v jazyce *FXML*, který slouží pro tvorbu formulářových aplikací. Zároveň ve FXML podporuje tvorbu grafického uživatelského rozhraní pomocí *Java Scene Builderu*. Lze pomocí něj lehce vytvořit základní koncept uživatelského rozhraní, které se dá dynamicky upravovat za běhu aplikace.

4.2.1 Záhlaví aplikace

Grafické uspořádání aplikace lze rozdělit do několika částí. První částí grafického rozhraní lze identifikovat jako záhlaví. Záhlaví je zobrazeno na obrázku 4.3. Je zde zobrazen název aktuálně otevřeného modelu. V tomto případě *modelX*. Na dalším řádku je v levé části umístěno ovládání celé aplikace. Záložka *File* je implementována jako vyskakovací menu, kde jsou uvedeny funkce pomocí kterých uživatel může například ukládat a načítat modely atd. Další záložka *Help* nasměruje uživatele k pomoci. Na stejném řádku ve střední části je vytvořeno tlačítko *Build on server*, které vedle sebe skrývá další tlačítka. Tyto skrytá tlačítka představují ovládací prvky pro práci se PNtalk serverem. Poslední část je jednoduché označení, které informuje uživatele o aktuálním stavu aplikace.

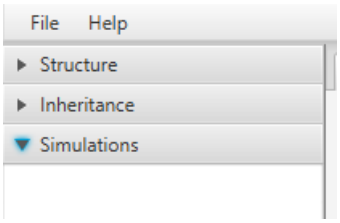
Obrázek 4.3: **Záhlaví aplikace**

4.2.2 Accordion

Na dalším obrázku 4.4 je zobrazena levá část aplikace, kde se nachází *accordion*.[7] Tento accordion obsahuje tři základní okna, které jsou *Structure*, *Inheritance* a *Simulations*. Všechny okna obsahují stromovou strukturu, ve které jsou zobrazena související data. V okně *Structure* jsou zobrazeny data o struktuře daného modelu z pohledu tříd a jejich sítí. Okno Inheritance zobrazuje data o hierarchii dědičnosti tříd v otevřeném modelu. Poslední okno se aktivuje při přeložení modelu na serveru, zobrazuje aktivní simulace a všechny jejich

[7] Accordion je rozjíždějící okno ve frameworku JavaFX.

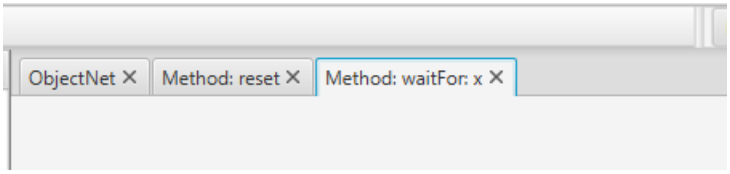
aktivní instance. Pro každé okno je vytvořené kontextové menu, které umožňuje uživateli měnit a zobrazovat části modelu na úrovni sítí a tříd.



Obrázek 4.4: **Accordion**

4.2.3 Pracovní plocha

Největší část aplikace zabírá pracovní plocha, která se nachází uprostřed aplikace. Aplikace pracuje se záložkami podobně jako internetový prohlížeče. Jedná se o *TabPane*, který dokáže zobrazovat záložky a přepínat mezi nimi. Lehce tak lze jednotlivé záložky přehazovat prohazovat a zavírat. Jejich grafická podoba je zobrazena na obrázku 4.5. Otevírání nových záložek probíhá v accordionu.



Obrázek 4.5: **Záložky pracovní plochy**

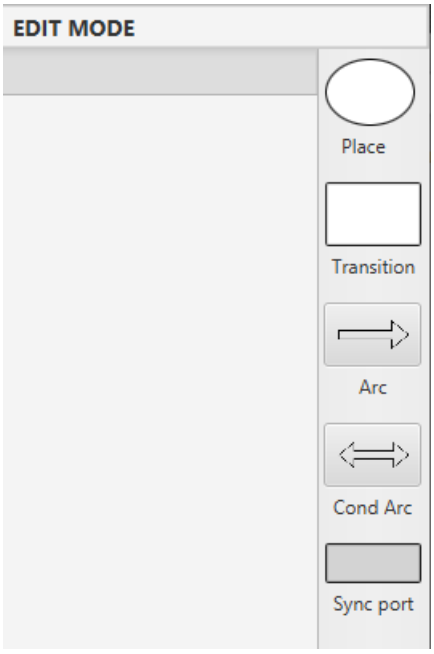
4.2.4 Toolbox

Poslední částí je *toolbox* na obrázku 4.6, který se nachází na pravé straně. Je implementován pomocí java *Toolbar*, který seřadí grafické objekty za sebe. Obsahuje tři labely, který představují funkcionalitu pro přidání míst, přechodů a synchronních portů do právě zobrazené sítě na pracovní ploše. Toolbox také obsahuje dvě tlačítka. První s jednosměrnou šipkou a názvem *Arc*. Při kliknutí na tlačítko se na pracovní ploše, v právě otevřené síti, objeví kolem jednotlivých elementů připojovací tlačítka, které uživateli umožní vytvořit hranu. Druhé tlačítko toolboxu s obousměrnou šipkou obdobně umožňuje vytvořit testovací hranu.

4.3 Kontrolér

Propojení frontendu a backendu je vytvořeno java třídou *Controller*.[8] Všechny reakce na uživatelské akce jsou zachyceny v této třídě. V případě chyby nebo doplňující otázky controller vytváří modální okna, kterými aplikace komunikuje s uživatelem. Controller zároveň dynamicky mění frontend. Controller tedy řídi veškeré změny, které se dynamicky projevují v grafickém uživatelském rozhraní aplikace.

[8]Controller ve frameworku JavaFX slouží k propojení grafického uživatelské rozhraní v jazyce FXML s pozadím programu v jazyce Java.



Obrázek 4.6: **Toolbox pro práci se sítí**

4.4 Ukládání a načítání modelu

Aplikace musí dokázat uživatelem vytvořené a upravené modely ukládat a znovu načíst. K tomuto účelu se v podkapitole 3.4 lze dočíst o jazyce PNML, který je předlohou pro strukturu ukládání celého modelu. I když jazyk PNML má možnosti pro uchování grafických vlastností uvedených v této bakalářské práci [5], tak pro účely aplikace jsou pravidla tohoto jazyka omezující. Hlavním důvodem jsou doplňující grafické vlastnosti sítí, které si musí aplikace pamatovat. Z tohoto důvodu jsou výsledná data pro uložení modifikována tak, aby bylo možné zahrnout grafické prvky, které jsou nezbytné pro aplikaci. Výsledná data jsou založena na jazyku PNML, ale také obohacena dalšími prvky, které hranice jazyk PNML překračují. Tyto data pak lze s jistotou nazvat jako XML nebo jako modifikovaný PNML. Příkladem těchto přidaných modifikací jsou například tagy *expression*, který představuje hodnotu hranového výrazu a určuje jeho polohu v síti.

Ukládání modelu probíhá vždy do stejné složky s názvem *export* umístěnou v rootu, odkud je aplikace otevřena. Kromě modelů jsou zde uloženy také exporty dílčích části modelů například tříd, metod, objektových sítí a dalších. Aplikace tedy je implementována tak, aby kromě celkového uložení modelu mohla exportovat některou z jeho částí a poté ji importovat do jiného případně toho samého modelu. K rozlišení, zda se jedná o celý model nebo jen o jeho část, jsou exportované soubory ukládány s názvem obsahujícím typ dat uložených v souboru. Příkladem takového názvu souboru je *modelX.model.xml*, kde tečka rozděluje název od typu. Obdobně vypadají názvy exportovaných částí modelu.

Příklad uložené struktury modelu je zobrazen v obrázku 4.7. Struktura uloženého modelu v jazyce XML je obdobná jako struktura načteného modelu v backendu. Jednotlivé elementy, které se v modelu mohou vyskytnout, jsou v XML uloženy jako tagy.[9] Jejich jednotlivé vlastnosti jsou uloženy v atributech tagu. V případě, kdy například objektová

[9]Tag je značka ohraničená ostrými závorky, ze které sestává program v značkovacích jazycích jako je HTML, XML, atd.

síť obsahuje více míst, tak do jejího obsahu tagu jsou vloženy další tagy, které představují místa. Tímto způsobem je zmapován a převeden celý model. K automatické tvorbě tagů je použita knihovna *jdom-2.0.6*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<model mainClass="ClassA" name="inherModel">
    <class father="ClassB" name="ClassA">
        <objectNet>
            <place id="p1" inheritance="False" initAction="" initMarking="2`#e" x="701.0" y="231.0"/>
            <place id="placeObjB" inheritance="True" initAction="" initMarking="" x="928.0" y="437.0"/>
            <transition action="" guard="" id="transObjB" inheritance="True" x="1310.0" y="335.0">
                <arc cond="False" endPos="right" extern="False" inheritance="True" place="placeObjB" startPos="left"
                     type="POSTCOND">
                    <expression value="#e" x="1208.0" y="458.0"/>
                    <controlPoint id="0" x="1217.0" y="427.0"/>
                </arc>
            </transition>
        </objectNet>
        <method name="hello">
            <place id="return" inheritance="True" initAction="" initMarking="" x="884.0" y="635.0"/>
            <place id="input" inheritance="True" initAction="" initMarking="2`#e" x="1030.0" y="158.0"/>
```

Obrázek 4.7: **Příklad uložených dat modelu v XML**

4.5 SVG export

Export do jazyka SVG probíhá podobně jak u ukládání modelů, jelikož je podobně založen na bázi jazyku XML. Využívá se pro to stejná knihovna *jdom-2.0.6*. Rozdíl je v míře exportu. Při exportu do SVG je jako zdroj použita pouze právě zobrazena síť z pracovní plochy. Na obrázku 4.8 je zobrazen úryvek z programu v jazyce SVG, který aplikace exportuje. Aplikace projde všechny elementy zobrazené v síti na pracovní ploše a převede je na ekvivalenty v jazyce SVG. SVG export neobsahuje kontrolní body, kterými uživatel upravuje cestu hran. Je tím tak export zpřehledněn.

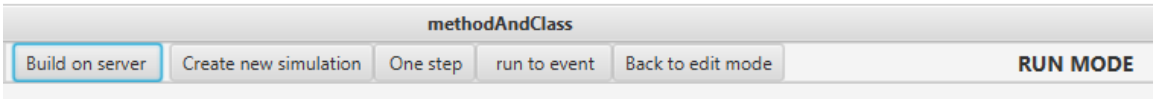
```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 1648 946">
    <ellipse cx="782" cy="128" fill="white" fill-opacity="1" rx="55" ry="22" stroke="black" stroke-width="1"/>
    <text dominant-baseline="middle" font-size="15px" text-anchor="middle" x="762" y="170">p01</text>
    <text dominant-baseline="middle" font-size="13px" text-anchor="middle" x="782" y="141"/>
    <text dominant-baseline="middle" font-size="13px" text-anchor="middle" x="782" y="126"/>
    <line stroke="black" x1="732" x2="832" y1="128" y2="128"/>
    <ellipse cx="574" cy="280" fill="white" fill-opacity="1" rx="55" ry="22" stroke="black" stroke-width="1"/>
    <text dominant-baseline="middle" font-size="15px" text-anchor="middle" x="554" y="322">p02</text>
    <text dominant-baseline="middle" font-size="13px" text-anchor="middle" x="574" y="293"/>
    <text dominant-baseline="middle" font-size="13px" text-anchor="middle" x="574" y="278">1`3</text>
    <line stroke="black" x1="524" x2="624" y1="280" y2="280"/>
    <ellipse cx="792" cy="392" fill="white" fill-opacity="1" rx="55" ry="22" stroke="black" stroke-width="1"/>
    <text dominant-baseline="middle" font-size="15px" text-anchor="middle" x="772" y="434">p03</text>
    <text dominant-baseline="middle" font-size="13px" text-anchor="middle" x="792" y="405"/>
    <text dominant-baseline="middle" font-size="13px" text-anchor="middle" x="792" y="390">1`1</text>
    <line stroke="black" x1="742" x2="842" y1="392" y2="392"/>
    <ellipse cx="583" cy="510" fill="white" fill-opacity="1" rx="55" ry="22" stroke="black" stroke-width="1"/>
```

Obrázek 4.8: **SVG export**

4.6 Spolupráce se PNtalk serverem

Před samotnou komunikací se serverem je potřeba model exportovat do jazyka PNtalk. Podobně jak při ukládání do XML je nutné převést model do jazyka PNtalk. Opět podle struktury uložené v backendu je potřeba všechny elementy převést do jejich ekvivalence v

PNtalku. Takto se vytvoří jeden dlouhý řetězec, který je předložen serveru. Celá tato akce probíhá po stisknutí tlačítka *Build on server*, které se nachází na horní liště na obrázku 4.3. Po jejím stisknutí se naváže komunikace se serverem.Všechny komunikační protokoly jsou uvedeny v příloze B.



Obrázek 4.9: **Záhlaví v run módu**

Server komunikuje s aplikací přes *localhost IPv4*[10] a port 9999. Pro každou komunikaci se vytvoří socket,[11] který tvoří komunikační proces. Po stisknutí tlačítka pro překlad se provede export do PNtalk programu a provede se komunikační protokol *Build*. Ten pošle serveru požadavek na přidání třídy a hned po něm pošle i data třídy v jazyku PNtalk. Dle protokolu server odpoví *OK* nebo *FAILED*, což jsou standardní odpovědi pro úspěch nebo neúspěch. V případě úspěchu se odkryjí další tlačítka z lišty v záhlaví na obrázku 4.9 a změní se mód aplikace.

V případě správného překladu je možné vytvořit novou simulaci. Při vytváření simulace se aplikuje komunikační protokol z *New simulation*. Pokud i tato komunikace proběhne v pořádku, pak následují další komunikační procesy *New event* a *Get state*. Pokud aplikace má definován bod zastavení, pak se na server nahraje *event*,[12]. Poté dojde k druhému komunikačnímu procesu *Get state*, který získá stav dané simulace. Při každé další změně v simulaci se aktivují poslední dva komunikační procesy *Simulate one step* a *Simulate to event* a dochází k opětovnému získání stavu. Přijatá data ze serveru ohledně aktuálního stavu simulace se vždy musí řádně zpracovat tak, aby mohli být jednotlivé instance nahrány do struktury simulace v backendu. Následně se struktura simulace nahraje do okna *Simulations* v accordionu. Zde v stromové struktuře si uživatel může otevřít instance jednotlivých sítí, dále provádět kroky simulace a sledovat změny, které se okamžitě projevují na pracovní ploše.

4.7 Testování

Tato podkapitola popisuje testování aplikace. Samotné testování lze rozdělit do dvou fázích. První fázi je průběžné testování.[13] Druhou fází je uživatelské testování, které bylo prováděno za účasti dalších osob.

4.7.1 Průběžné testování

Průběžné testování lze rozdělit na testování backendu a frontendu. Toto testování začalo po první fázi vývoje softwaru, kdy byla vytvořena základní struktura backendu a trvala téměř do konce vývoje aplikace. Backendové testování bylo provedeno řešitelem na ukázkových příkladech. Tyto příklady byly navrženy tak, aby bylo možné pozorovat chování aplikace

[10]Localhost IPv4 je speciální adresa 127.0.0.1, kterou většinou používají aplikace daného zařízení pro komunikaci.

[11]Socket je v informatice bod připojení pro počítačovou síť.

[12]Event představuje v PNtalku bod zastavení simulace, který lze nastavit pouze na přechod.

[13]Průběžné testování je typ testů, které se provádí na straně organizace nebo osob vyvíjejících daný software

na vznikající strukturu vytvořenou v backendu. Sledování vzniklé struktury bylo promítáno do frontendu, kde řešitel mohl sledovat, zda je struktura v backendu správně vytvořena.

Průběžné testování na úrovni frontendu je odlišný případ. Pro tento druh testů lze uplatnit pouze testování na základě pozorování osoby. Proto průběžné testování frontendu zůstalo čistě na řešiteli této diplomové práce a jeho vizuálním vnímáním vytvářené aplikace. Je vhodné zde připomenout Java Scene Builder. Tento nástroj vývojáři umožní vytvořit vzor určitého grafického elementu a nastavit jeho vlastnosti. V případě dynamického přidávání grafických objektů je tak tento vzor vhodnou předlohou, která slouží k ověření správné funkcionality nově přidaných grafických objektů. Tento způsob testování je efektivnější díky rychlosti, jakou lze v Java Scene Builderu grafický objekt vytvořit.

Výsledky průběžného testování nelze jednoduše prezentovat. Je to pouze subjektivní pohled řešitele, který si průběžné testování stanovil. Java Scene Builder byl vhodným pomocníkem při testování a usnadnil řešiteli práci s testováním dynamicky přidaných grafických elementů.

4.7.2 Uživatelské testování

Uživatelské testování začalo v posledních fázích vývoje. Uživatelské testování bylo zaměřeno pouze na frontend. Bylo potřeba vyzkoušet zejména funkčnost na dalších platformách a intuitivnost, kterou aplikace působí na lidi. Aplikace je plně otestována na platformě Windows 10, kde lze zaručit plnou funkčnost. Aplikace je zároveň otestována pro operační systém na bázi linuxu, konkrétně pro Ubuntu. Testování pro Ubuntu je složitější z hlediska požadovaných knihoven pro překlad. V některých případech bylo nutné doinstalovat určité knihovny pro framework JavaFX. Zároveň i samotná instalace prostředí Pharo je na Ubuntu složitější, jelikož je obtížné na 64 bitovém operačním systému provést instalaci starší verze Pharo, která je pouze 32 bitová.

Jelikož je aplikace speciální v náročnosti na uživatele, který musí znát alespoň základy Petriho sítí, nebylo možné provádět testování s kýmkoliv. Z tohoto důvodu byli k Uživatelskému testování přizváni dva kolegové ze studia na fakultě řešitele práce, kteří měli zkušenosti pouze s Petriho sítěmi nikoli s OOPN. Testování mělo za úkol odhalit nedostatky v přehlednosti aplikace a otevřít možnosti pro vylepšení aplikace. Zároveň lze považovat za Uživatelské testování i poznámky vedoucího práce, kde vliv na výslednou aplikaci je mnohem větší.

Výsledky Uživatelského testování jsou oproti průběžnému lépe prezentovatelné. Největší podíl na změnách ve vývoji měl vedoucí práce, jehož testování aplikace mělo vliv na celkový vzhled a funkčnost. Ve směs šlo hlavně o rady k správnému pochopení OOPN ve spojení s PNtalkem a jejich promítnutí do aplikace. Také zde byly připomínky k správnému umístění jednotlivých funkčních prvků aplikace ve frontendu. Všechny tyto připomínky byli řešitelem akceptovány a implementovány.

Výsledky testů dvou přizvaných studentů byly méně směrodatné. I když oba se orientují v Petriho sítích, tak pochopení OOPN bylo pro ně mnohem náročnější. Oba studenti měli problém se orientovat ve struktuře modelu a významu v obsahu elementů sítě. Testování funkčnosti z pohledu OOPN a PNtalku je v tomto případě nepoužitelné. Oba studenti alespoň provedli testování grafického uživatelského rozhraní a jeho působení na ně. Některé jejich poznámky byly přijaty a implementovány v aplikace.

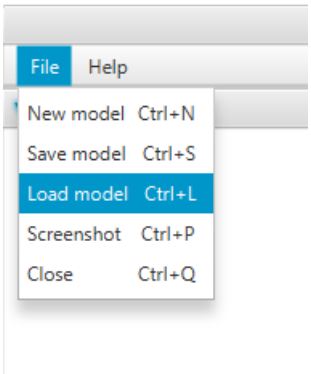
Kapitola 5

Příklady použití aplikace

Tato kapitola pojednává o práci uživatele v aplikaci. Pro instalaci je vytvořen návod v příloze A, kde lze nalézt informace o podpoře, zdrojích ke stažení, instalačních procesech a častých problémech. Dále lze najít v této kapitole několik vzorových příkladů, které ilustrují jejich simulaci.

5.1 Práce s modelem

Po zapnutí se aplikace vždy nachází v prázdném stavu. Nejdříve je potřeba otevřít nebo vytvořit model. Načítání, ukládání a otevírání lze nalézt ve vysouvacím menu *File*, které je zobrazeno na obrázku 5.1. Kromě zmíněných funkcí menu obsahuje také uzavření celé aplikace a pořízení snímku. U všech těchto funkcí jsou zároveň nastaveny klávesové zkratky, které uživateli usnadní práci s aplikací. V případě načítání modelu pomocí tlačítka *Load model* nebo klávesové zkratky Ctrl + L se otevře modální okno pro výběr souboru. Výchozí složka pro ukládání a načítání modelu se nazývá *export*. Je nutné vybrat soubor, který v názvu obsahuje řetězec „.model“. Ostatní soubory slouží pro import a export, který se provádí v jiné části aplikace.

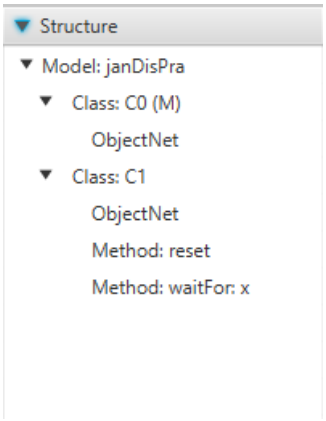


Obrázek 5.1: **File menu**

5.2 Struktura modelu

Po načtení modelu se zobrazí v accordionu okno *Structure*, kde se zobrazí kompletní struktura tříd se všemi jejími sítěmi. Příklad reprezentující tuto strukturu lze najít na obrázku

5.2. Jsou zde zobrazeny dvě třídy *C0* a *C1*. U třídy C0 je zobrazen příznak „(M)“. Tento příznak značí hlavní třídu modelu.[1] Ve stromové struktuře jsou také zobrazeny jednotlivé sítě všech tříd.



Obrázek 5.2: **Struktura modelu**

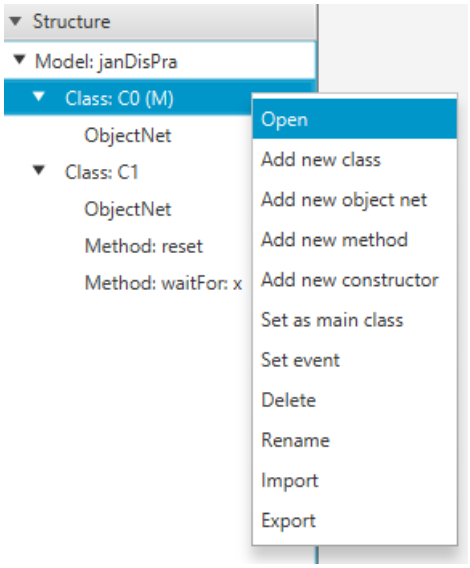
Kontextové menu pro okno Structure z obrázku 5.3 umožňuje uživateli hned několik funkcionalit. Lze pomocí něj otevřít danou sít na pracovní ploše. Mimo to síť lze otevřít také dvojklikem. Dále kontextové menu obsahuje přidání nových sítí a tříd. Pro přidání sítě je důležité mít vybranou správnou třídu, kam chce uživatel síť přidat. Když uživatel chce přidat metodu pro třídu C0, je nutné mít vybranou odpovídající položku ve stromové struktuře. Na obrázku 5.3 je právě tento případ zobrazen, kde je vybrána třída C0. Mazání ve struktuře modelu probíhá podobným způsobem. Pro přidání třídy není nutné vybrat položku modelu. Lze tak učinit po kliknutí kamkoliv v daném okně.

Mezi další možnosti kontextového menu patří přejmenování tříd, metod a konstruktorů pomocí tlačítka *Rename*. Také se zde dá nastavit hlavní třída tlačítkem *Set as main class*. Tlačítko *Set event* dovoluje uživateli nastavit přechod vybrané sítě jako bod zastavení při simulaci. Dvě poslední tlačítka *Import* a *Export* umožňují separátní uložení a načtení dílčích částí modelu. Je zde potřeba zdůraznit, že pro obě tlačítka je nutné mít vybrán správný element. Například pokud uživatel chce přidat metodu, musí mít označenou třídu, která bude vlastníkem importované metody. Export lze provést u všech části modelu kromě jej samotného. Pro export modelu se standardně používá ukládání pomocí akce z obrázku 5.1.

5.3 Editace sítě

Po otevření sítě a její nahrání do pracovní plochy, může uživatel provádět její editaci. Příkladem takové sítě je obrázek 5.4. Jedná se o objektovou síť třídy C1 z příkladu na obrázku 5.2. V síti je možné vidět od každého elementu jeden kus. Je zde zobrazeno místo *p1*, přechod *Transition* a synchronní port *state: x*, kde jsou spolu všechny elementy propojeny pomocí všech typů hran. Hrana směřující z místa p1 do Transition je červeně zvýrazněná. K tomuto efektu dojde, když uživatel přejede kurzorem po kterékoliv její části. Pro přidání jednotlivých elementů je nutné, z toolboxu přetahovat elementy do sítě. Na místě, kde uživatel odklikne, se zobrazí přetahovaný element ve výchozím stavu. V ideálním případě by uživatel měl nově vzniklý element editovat tak, aby nevznikl konflikt se stejnými jmény.

[1] Server PNtalk vždy na úvod simulace vytvoří instanci objektové sítě hlavní třídy.



Obrázek 5.3: **Kontextové menu okna Structure**

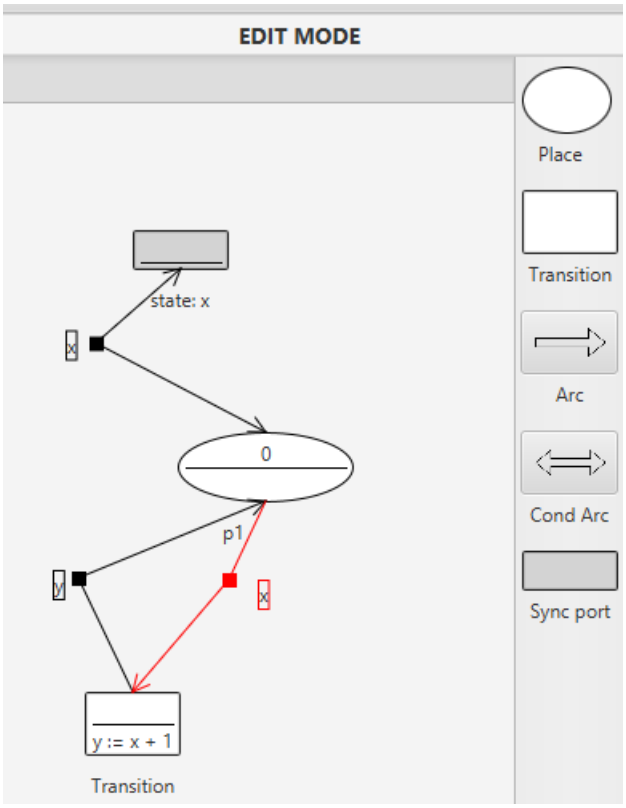
Pro přidání klasických hran je nutné kliknout na tlačítko *Arc* a poté vybrat zdroj a cíl hrany pomocí tlačítek, které se u všech elementů zobrazili. Přidání testovací hrany probíhá stejným způsobem.

V případě editace sítě metody musí aplikace umožnit tuto síť propojit s objektovou sítí dané třídy. K tomuto účelu je nutné kliknout pravým tlačítkem na pracovní plochu sítě, kde se zobrazí kontextové menu s jedinou akci *Show/Hide places from objectnet*. Po kliknutí na tuto akci se zobrazí místa objektové sítě zvýrazněna šedým pozadím. Příklad této akce je zobrazen na obrázku 5.5, který je příkladem metody *waitFor: x*. Místo p1 představuje místo z objektové sítě.

5.4 Dědičnost modelu

Jak pracovat s dědičností v aplikaci? Pro úpravu dědičnosti je vytvořeno extra okno *Inheritance* v accordionu. Opět se zde dá nalézt stromová struktura tříd, která nyní značí dědičnost tříd. V obrázku 5.6 jsou zobrazeny třídy *CL1* a *CL2*. Z jejich postavení ve stromové struktuře lze vyvodit, že třída CL1 dědí z CL2. Pro nastavení otce třídy je nutné opět použit kontextové menu, kde se nachází akce *Set class parent*. Pomocí této akce se zobrazí modální okno, kde bude na výběr ze všech tříd. Po vybrání je nutné ještě u dědící třídy použít druhou akci z kontextového menu, která se nazývá *Reload inheritance from parent class*. Tato akce provede aplikaci nastavené dědičnosti u vybrané třídy. V případě, kdy obě třídy obsahují objektové sítě, je nutné rozhodnout, co se stane s objektovou sítí u dědící třídy. Uživatel dostane na výběr ze tří možných akci. Může dědící objektovou síť přepsat, spojit nebo ponechat. V případě spojení dochází k smíchání zděděných elementů spolu s původními.

Příkladem tohoto spojení je obrázek 5.7, kde jsou rozlišeny dva druhy elementů. Původní elementy mají klasické zobrazení jako z předchozích případů. Zděděné elementy se vyznačují přerušovaným ohraničením. Například místo *CL2place* je zděděné místo. Dále je možné provádět editace, které spojují oba typy elementů.



Obrázek 5.4: **Objektová síť**

5.5 Modální okna

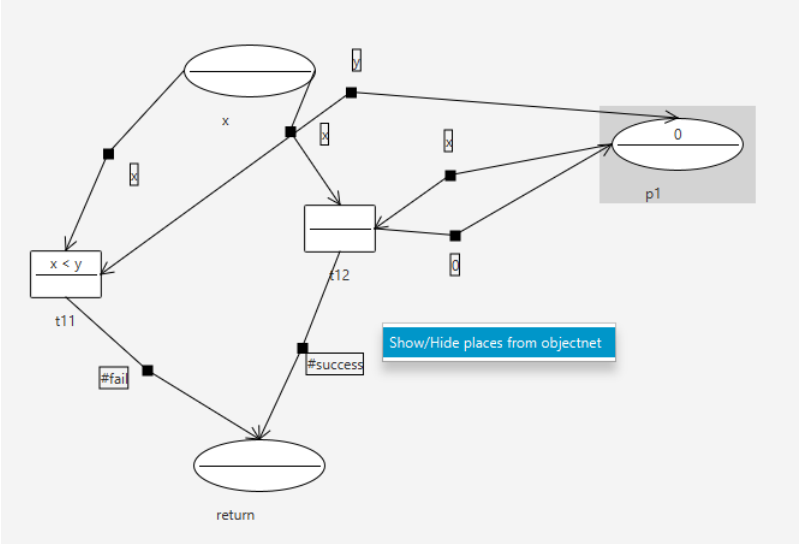
Interakce mezi uživatelem a aplikací je provedena přes modální okna, která vždy žádají po uživateli dodatečné informace. Také tyto modální okna informují uživatele o akci, kterou se pokoušel provést. Může se zobrazit okno o úspěšném provedením akce v případě, že výsledek není pro uživatele na první pohled patrný. Rovněž se může zobrazit okno, které oznamuje chybu nebo upozornění, které uživatele informuje o vzniklém problému.

Modální okna mohou po uživateli chtít doplnit určitou informaci. Tyto okna lze rozdělit na dva typy. Prvním je *fulltext* okno, které přímo vyžaduje po uživateli napsaní například názvu nebo hodnoty pro danou akci. Druhým typem okna je *select*, kde uživatel vybírá z několika možností. Příkladem, kdy se objevuje toto okno, je výběr dědičnosti, kdy uživatel musí vybrat z předem definovaných možností.

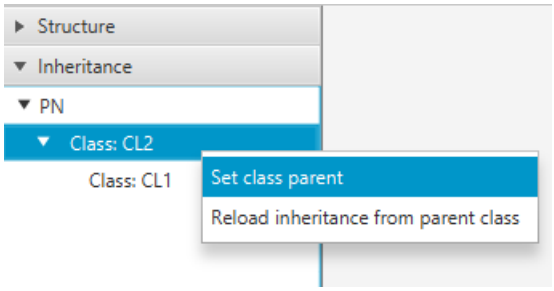
5.6 Simulace

Před samotným ovládáním simulace je důležité připomenout, že PNtalk server musí být zapnutý. V opačném případě můžou vzniknout chyby, které aplikace nemusí detekovat a zamrzne. V případě, kdy se snažíte na vypnutém serveru přeložit model, dojde k upozornění aplikace o daném problému.

Pokud uživatel má model vytvořen a chce provést simulaci musí stisknout tlačítko *Build on server*. Pokud překlad na serveru bude v pořádku, tak se aplikace přepne do *RUN MODE*. V tomto módu nemůže uživatel editovat model. V accordionu se uživateli otevře



Obrázek 5.5: **Propojení metody s objektovou sítí**



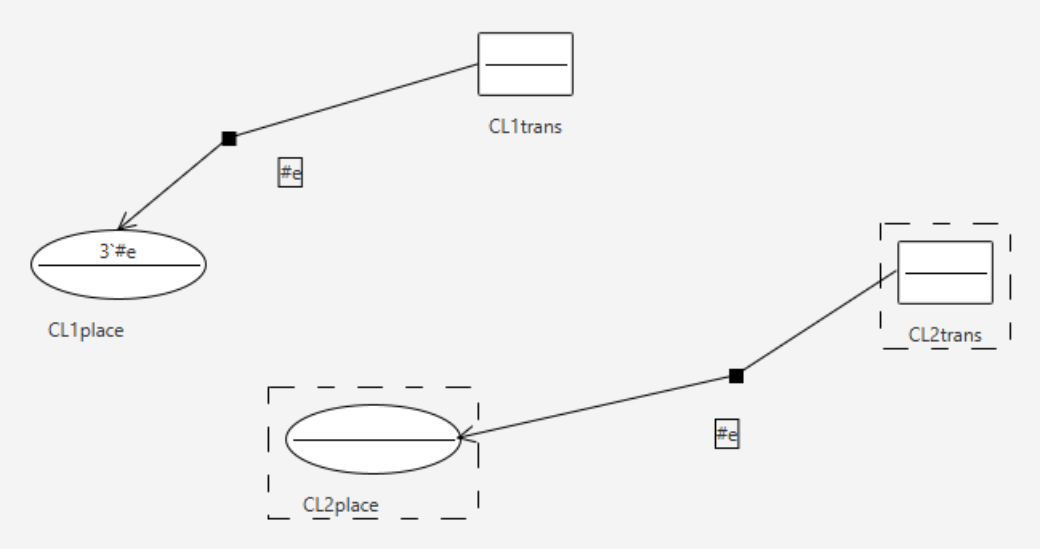
Obrázek 5.6: **Dědičnost modelu**

okno *Simulations* a ostatní okna se skryjí. Rovněž se skryje toolbox. V následujícím kroku musí uživatel vytvořit první simulaci. Těchto simulací může uživatel vytvořit kolik jen chce.

Po vytvoření simulace se uživateli objeví její struktura. Při vytvoření simulace se v prvním kroku vždy vytvoří instance objektové sítě hlavní třídy. Příkladem takové struktury je obrázek 5.8, kde jsou zobrazeny hned dvě běžící simulace. Každá simulace má své určité ID, které ji jednoznačně identifikuje. Podobný identifikátor mají i ostatní instance. Lze si tedy všimnout, že po simulaci se vždy nachází instance tříd a v nich se teprve nachází instance sítí. Pouze instance sítí lze otevřít a sledovat, jak probíhá simulace.

Pro posun v simulaci existují v aplikaci dvě tlačítka. Lze je najít v záhlaví celé aplikace. Jmenují se *One step* a *run to event*. Jak je z názvu patrné, tak tlačítko One step provádí vždy pouze jeden krok v aplikaci. V případě použití druhého tlačítka run to event, dochází k provedení jednoho nebo více kroků. Počet provedených kroků se odvíjí od nastaveného bodu zastavení. V případě, kdy uživatel nenastaví tento bod, dojde k posunu v simulaci do konce.

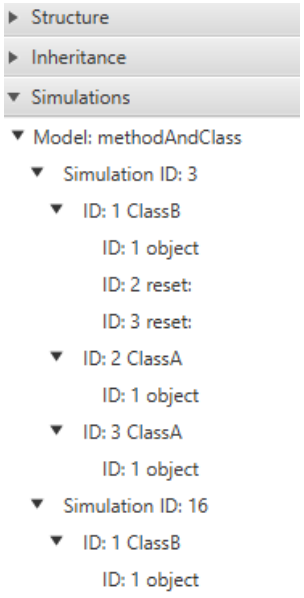
Kromě dvou tlačítek pro posun v simulaci existují i tlačítka v kontextovém menu v okně pro zobrazení struktury simulace. Tyto tlačítka jsou tam navíc. Jejich původní smysl, ale upozorňuje na situaci, kdy uživatel má otevřených více simulací v jednom okamžiku. Pokud uživatel chce provádět simulace paralelně, musí vždy kliknout levým nebo pravým



Obrázek 5.7: **Dědičnost v objektové síti**

tlačítkem myši na jakoukoliv instanci vybrané simulace. Tím se uloží daná simulace jako výchozí a tlačítka v záhlaví aplikace budou provádět akce nad ní.

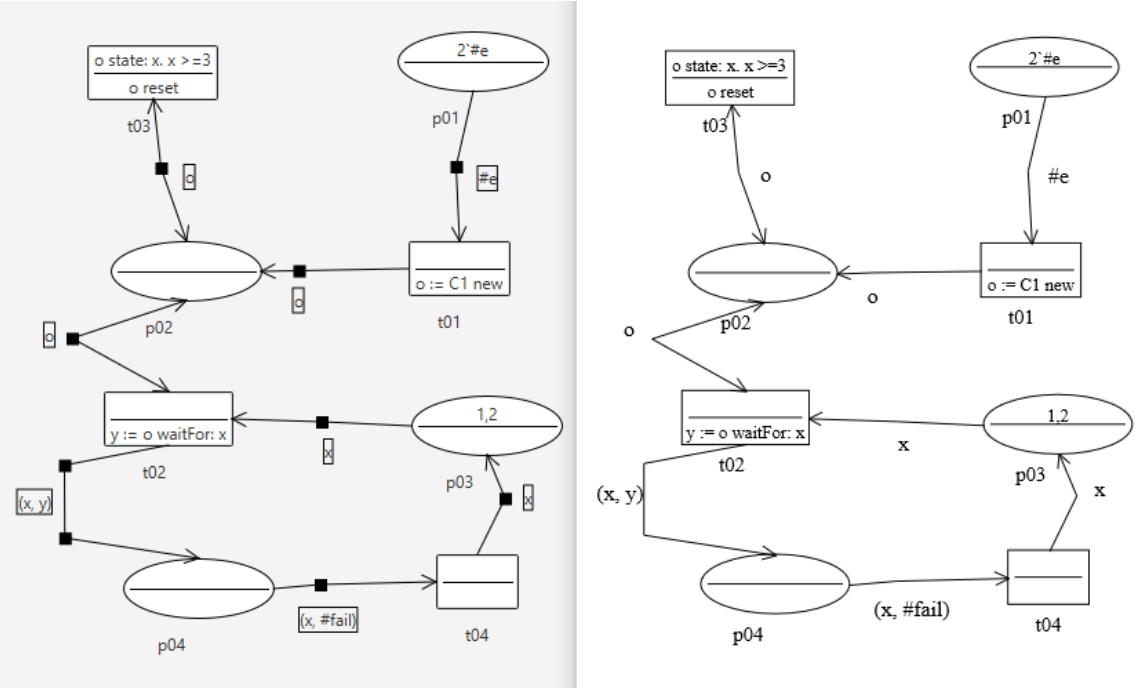
Pro ukončení simulace musí uživatel stisknout tlačítko *Back to edit mode*. Po tomto kroku se aplikace opět vrátí do editačního stavu. Před tímto návratem musí uživatel zvážit, zda nechce ukončit běžící simulace. Pokud uživatel simulace neukončil na serveru nebo nevypnul aplikaci, tak i když skryty jsou simulace stále dostupné. Pokud uživatel tyto simulace chce ukončit, tak musí pravým tlačítkem kliknout na dané poličko simulace a vybrat akci *Close simulation*. Kliknutím na instancí třídy nebo sítě nelze simulaci ukončit.



Obrázek 5.8: **Simulace**

5.7 Export do SVG

Pro export do SVG obrázku je nutné použít akci *Screenshot* z menu, které je zobrazeno na obrázku 5.1. Lze pro tuto akci použít také klávesovou zkratku Ctrl + P. SVG obrázek se uloží do složky *screenshots*. Příkladem takové export je obrázek 5.9. Jeho levá část představuje vzor z aplikace, který v ní byl vytvořen. Pravá strana obsahuje SVG export. Jsou zde zjevné rozdíly, které výsledný export zjednodušují. Zjednodušený výsledek je tak pro uživatele přehlednější.



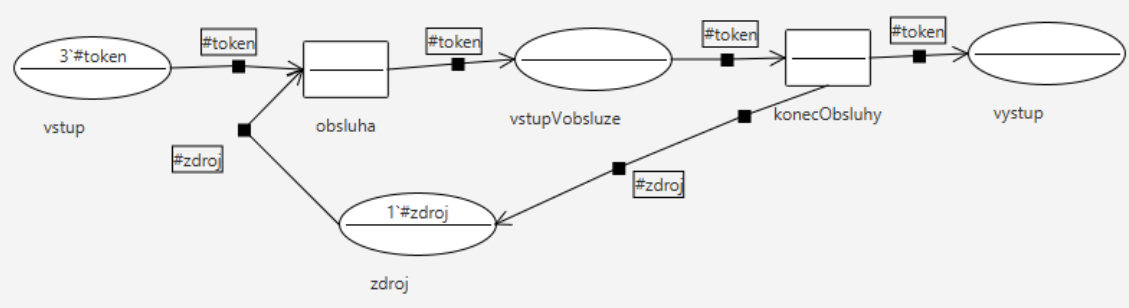
Obrázek 5.9: **SVG export**

5.8 Příklady

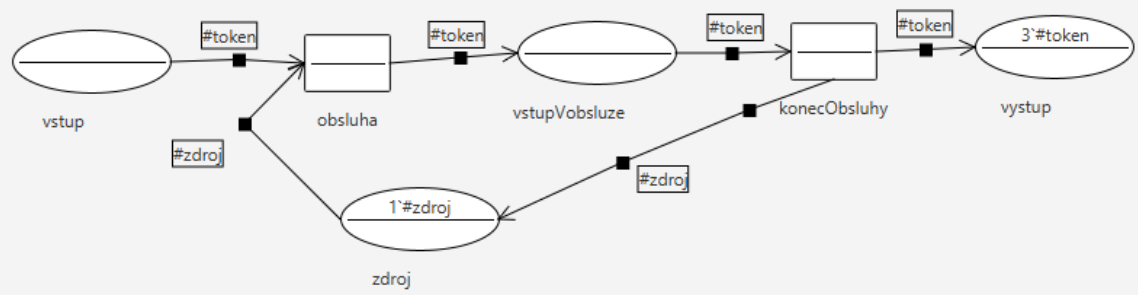
Následující podkapitola obsahuje několik příkladů OOPN. Z příkladu je patrný postup změny v aplikaci při postupu v simulaci. Příklady začínají od jednodušších a postupně se jejich složitost zvyšuje.

5.8.1 Základní instance objektové sítě

První příklad demonstruje základní simulaci v síti. V příkladu se nachází pouze jedna třída se svojí objektovou sítí, kde probíhá simulace. Obrázek 5.10 představuje počáteční stav v síti. Příklad si lze představit jako postupné zpracování požadavků ve frontě. V místě *vstup* jsou ve frontě 3 tokeny k vyřízení. Pokud je na místě *zdroj* přítomno značení *#zdroj* pak je možné provést přechod *obsluhu*. V následujícím kroku se provede přechod *konecObluhy* a vyřízený požadavek v podobě tokenu se přesune do místa *vystup*. Uvolněný zdroj se přesune zpět do původního místa a pokud bude na vstupu další token k vyřízení, tak se celý cyklus opakuje, dokud nebude vstup prázdný.



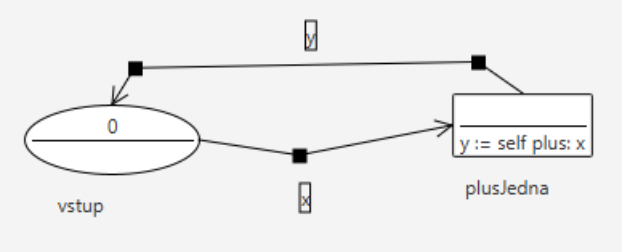
Obrázek 5.10: **Příklad: Základní instance objektové sítě - stav na počátku**



Obrázek 5.11: **Příklad: Základní instance objektové sítě - koncový stav**

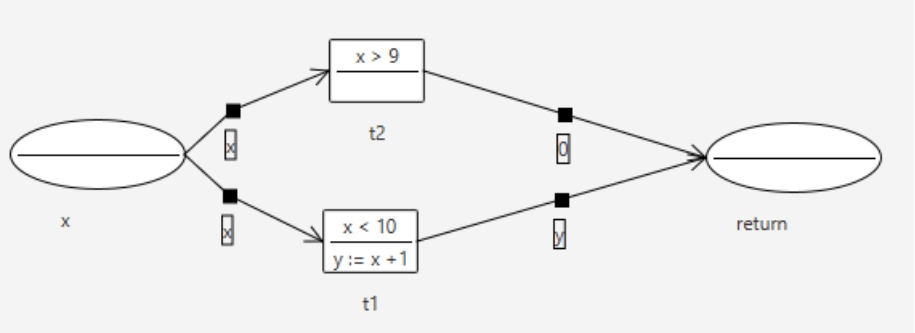
5.8.2 Volání metody

Druhý příklad demonstruje volání metody na obrázku 5.12. Struktura třídy sestává z objektové sítě a metody *plus: x*. Objektová síť na počátku obsahuje pouze jedno místo, ve které je počáteční značení nula, a jeden přechod. Tento přechod provádí volání vlastní metody s parametrem *X*, který přebírá od místa. Výsledek poté vrací v proměnné *Y*.



Obrázek 5.12: **Příklad: Volání metody - objektová síť**

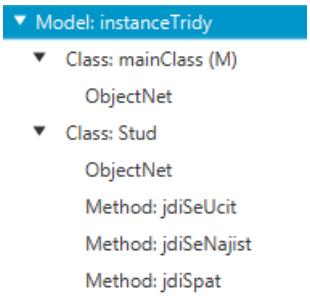
Metoda *plus: x* z obrázku 5.13 představuje počitadlo, které číslo ze vstupního místa *X* zvětší o 1 v případě, kdy je číslo menší než 10. Tento proces provádí přechod *t1*. V opačném případě se provede přechod *t2*, který číslo nuluje. Lze si tedy tento celý případ představit jako počitadlo, které se po dosažení hodnoty deset vždy v následujícím kroce vynuluje. Tento model je tedy nekonečný a neexistuje jeho finální podoba.



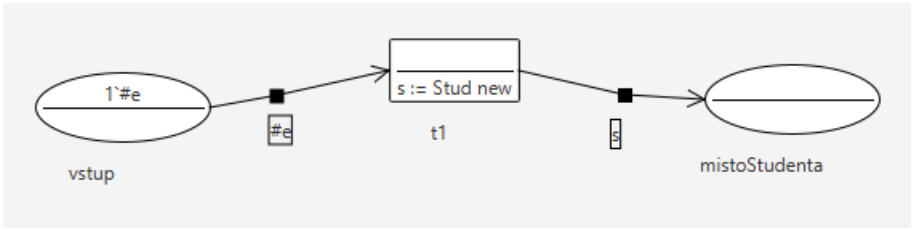
Obrázek 5.13: **Příklad: Volání metody - metoda**

5.8.3 Instance třídy

Následující příklad demonstruje způsob vytvoření instance jiné třídy z hlavní. Tento příklad se skládá ze dvou tříd, kde v hlavní třídě je vytvořena instance nové třídy *Stud*. Obrázek 5.15 představuje objektovou síť hlavní třídy, kde dochází pouze k vytvoření nového objektu. Při této instanci se automaticky vytvoří instance objektové sítě třídy Stud. Kromě objektové sítě třída obsahuje tři další metody uvedené ve struktuře modelu na obrázku 5.14.

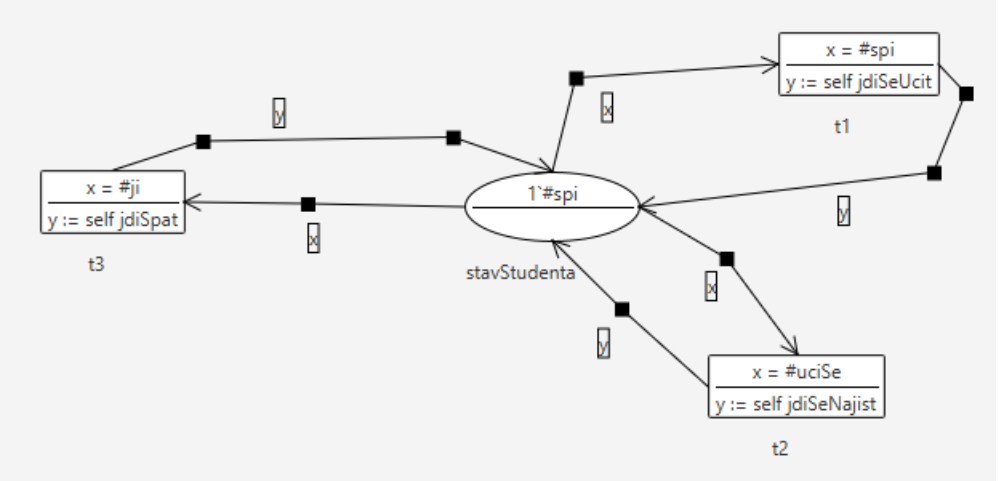


Obrázek 5.14: **Příklad: Instance třídy - Struktura modelu**

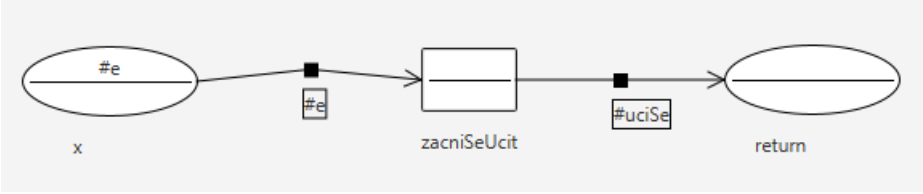


Obrázek 5.15: **Příklad: Instance třídy - hlavní třída**

Objektová síť třídy Stud představuje denní cyklus studenta. Vizuální podoba této sítě je zobrazena na obrázku 5.16. Zde se nachází jedno místo *stavStudenta*, které vždy obsahuje aktuální činnost studenta. Okolo místa jsou rozmístěny tři přechody. Každý přechod kontroluje ve stráži přechodu, zda daný stav studenta je ten, který odpovídá požadovanému stavu pro jeho provedení. Takto se stále dokola střídají studentovy činnosti. Síť jednotlivých metod je dost podobná. Obrázek 5.17 představuje metodu *jdiSeUcit*, která vrací nový status pro studenta.



Obrázek 5.16: **Příklad: Instance třídy - objektova síť třídy Stud**



Obrázek 5.17: **Příklad: Instance třídy - metoda jdiSeUcit**

Kapitola 6

Závěr

Cílem diplomové práce bylo provést studii objektově orientovaných Petriho sítí a vytvořit návrh řešení aplikace. Ve studii v kapitole 2 bylo provedeno zmapovaní různých druhů Petriho sítí a především podrobnější rozbor objektově orientovaných Petriho sítí v jazyce PNtalk. Návrh řešení zaznamenaný v kapitole 3 obsahuje několik pohledů, které předcházeli vývoji aplikace. Je zde rozdělení do iterací, které představuje plán vývoje aplikace. Dále jsou zde diagramy stavů, tříd a architektury, které popisují strukturu a chování aplikace.

Dalším cílem diplomové práce je vytvoření funkční aplikace, která zjednoduší uživatelům práci s objektově orientovanými Petriho sítěmi. Celkový popis implementované aplikace je uveden v 4 a je podrobně rozebrán. Rovněž jsou zde uvedeny informace o provedeném testování související s implementaci. Výsledek aplikace je prezentován na konkretních příkladech z kapitoly 5. Příklady prokazují funkčnost aplikace, která uživateli umožní pracovat s objektově orientovanými Petriho sítěmi.

Celkový výsledek této práce lze zhodnotit jako kvalitní. Návrh aplikace byl vhodně vytvořen a posloužil jako dobrý bod, o který se implementace opírala. Výslednou aplikaci lze považovat jako splnění zadání práce o implementaci nástroje. I když aplikace obsahuje pár drobných mezer především v uživatelské přívětivosti, tak tyto nedostatky mohou být omluveny složitostí OOPN. Proto lze celkový výsledek hodnotit kladně.

Aplikace sama o sobě představuje vhodný bod pro její další rozvoj. V ideálním případě by šel provést grafický upgrade s pomocí dalších frameworků, kterými by se zvýšila uživatelská přívětivost, jelikož ta je v moderních softwarových systémech alfa a omega. Další možností pro rozvoj může nastav v případě rozšíření PNtalk serveru o další funkcionalitu. Komunikace se serverem je od jádra aplikace oddělena a její rozšíření by šlo snadno přidat k aplikaci.

Literatura

[1] Billington, J., Christensen, S., Hee, K. van, Kindler, E., Kummer, O. et al. *The Petri Net Markup Language: Concepts Technology and Tools* [online]. University of South Australia, březen 2003 [cit. 2020-1-11]. Dostupné z: http://www.pnml.org/papers/PNML_CTT.pdf.

[2] Hordějčuk, V. *Objektově orientované programování* [online]. [cit. 2020-1-3]. Dostupné z: http://voho.eu/wiki/oop/.

[3] Janoušek, V. *Modelování objektů Petriho sítěmi*. Brno, CZ, 1998. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: http://www.fit.vutbr.cz/~janousek/publications/phdthesis.pdf.

[4] Jensen, K. *Coloured Petri Nets*. Springer-Verlag, 1996. ISBN 978-3-662-03241-1.

[5] Josefík, M. *Nástroj pro práci s objektově orientovanými Petriho sítěmi*. Brno, CZ, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://core.ac.uk/download/pdf/44404929.pdf.

[6] Kavi, K., Moshtaghi, A. a Chen, D.-J. Modeling Multithreaded Applications Using Petri Nets. *International Journal of Parallel Programming*. Říjen 2002, roč. 30, s. 353–371.

[7] Kindler, E. *Concepts, Status, and Future Directions* [online]. University of Paderborn, květen 2006 [cit. 2020-1-14]. Dostupné z: http://www.pnml.org/papers/PNML-EKA06.V3.pdf.

[8] Kochaníčková, M. *Petriho sítě* [online]. Univerzita Palackého, Přirodovědecká faktulta, leden 2008 [cit. 2019-12-27]. Dostupné z: http://phoenix.inf.upol.cz/esf/ucebni/petriho_site.pdf.renamed.

[9] Michálek, M. *SVG: vektorový formát, který na weby chyběl* [online]. vzhuruadolu.cz, září 2014 [cit. 2020-5-17]. Dostupné z: https://www.vzhurudolu.cz/prirucka/svg.

[10] Petrucci, L. a André Étienne. *Petri nets 2020* [online]. Université Paris 13, leden 2020 [cit. 2020-6-31]. Dostupné z: https://lipn.univ-paris13.fr/petrinets2020/CfP-Petri-Nets-2020.pdf.

[11] Radek Kočí, B. K. *Úvod do softwarového inženýrství Studijní opora* [online]. 2010 [cit. 2020-1-13]. Dostupné z: https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FIUS-IT%2Ftexts%2FIUS_opora.pdf&cid=9410.

[12] Toshiyuki MIYAMOTO, S. K. *A Survey of Object-Oriented Petri Nets and Analysis Methods* [online]. IEICE TRANS. FUNDAMENTALS, listopad 2005 [cit. 2019-12-28]. Dostupné z: https://www.researchgate.net/publication/31307320_A_Survey_of_Object-Oriented_Petri_Nets_and_Analysis_Methods.

Seznam příloh

Příloha A

Návod pro instalaci

Tato příloha obsahuje návod na instalaci aplikace. Aplikace byla vytvořena na platformě Windows 10 a její funkčnost je zde zaručena. Podpora pro systém na bázi Linux byla přidána, ale mohou se zde vyskytnout menší grafické rozdíly. Pokud by balíček aplikace nebyl k dostaní spolu s touto prací, lze jej stáhnout z tohoto sdíleného uložiště:
https://drive.google.com/drive/folders/1aJdyQArBHrn80NG6aUao6Z1zHd8KTqVt?usp=sharing

A.1 Instalace pro Windows

1. Stáhněte si a nainstalujte si aplikaci Pharo verze 5.
2. Stáhněte si balíček aplikace.
3. Ujistěte se, že máte nainstalovaný jdk verze 11 nebo novější.
4. Spusťte Pharo a jako image použijte z balíčku aplikace soubor *PNtalk-2.1-server.image*.
5. Na spuštěném Pharu označte do bloku příkaz *PNtalkServer start.* a klikněte na něj pravým tlačítkem. Otevře se nabídka a vyberte příkaz „Do it“.
6. Použijte příkazový řádek a přesuňte se přímo do složky v balíčku aplikace, kde se nachází soubor *GEOOPN_WIN.jar*. Proveďte spuštění pomocí příkazu *java -jar GEOOPN_WIN.jar*.

A.2 Instalace pro Linux

1. Stáhněte si a nainstalujte si aplikaci Pharo verze 5. Pozor pokud vlastníte 64 bit verzi systému, tak použijte následující sekvenci příkazů pro instalaci podpory 32 bit verze Phara:
   - sudo dpkg –add-architecture i386
   - sudo apt-get update
   - sudo apt-get install libx11-6:i386
   - sudo apt-get install libgl1-mesa-glx:i386
   - sudo apt-get install libfontconfig1:i386
   - sudo apt-get install libssl1.0.0:i386

2. Stáhněte si balíček aplikace.

3. Ujistěte se, že máte nainstalovaný jdk verze 11 nebo novější. Zároveň v některých případech je nutné si nainstalovat JavaFX sdk.

4. Spusťte Pharo a jako image použijte z balíčku aplikace soubor *PNtalk-2.1-server.image*.

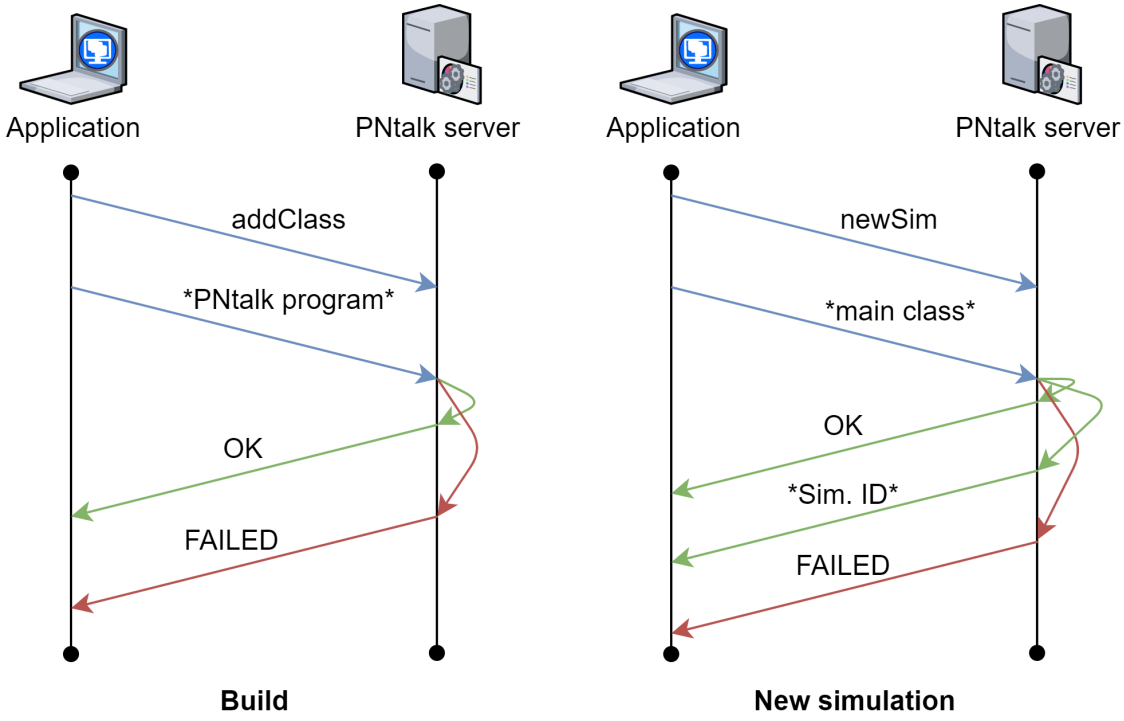
5. Na spuštěném Pharu označte do bloku příkaz *PNtalkServer start.* a klikněte na něj pravým tlačítkem. Otevře se nabídka a vyberte příkaz „Do it“.

6. Použijte terminál a přesuňte se přímo do složky v balíčku aplikace, kde se nachází soubor *GEOOPN_LINUX.jar*. Proveďte spuštění pomocí příkazu *java -jar GEOOPN_LINUX.jar*.

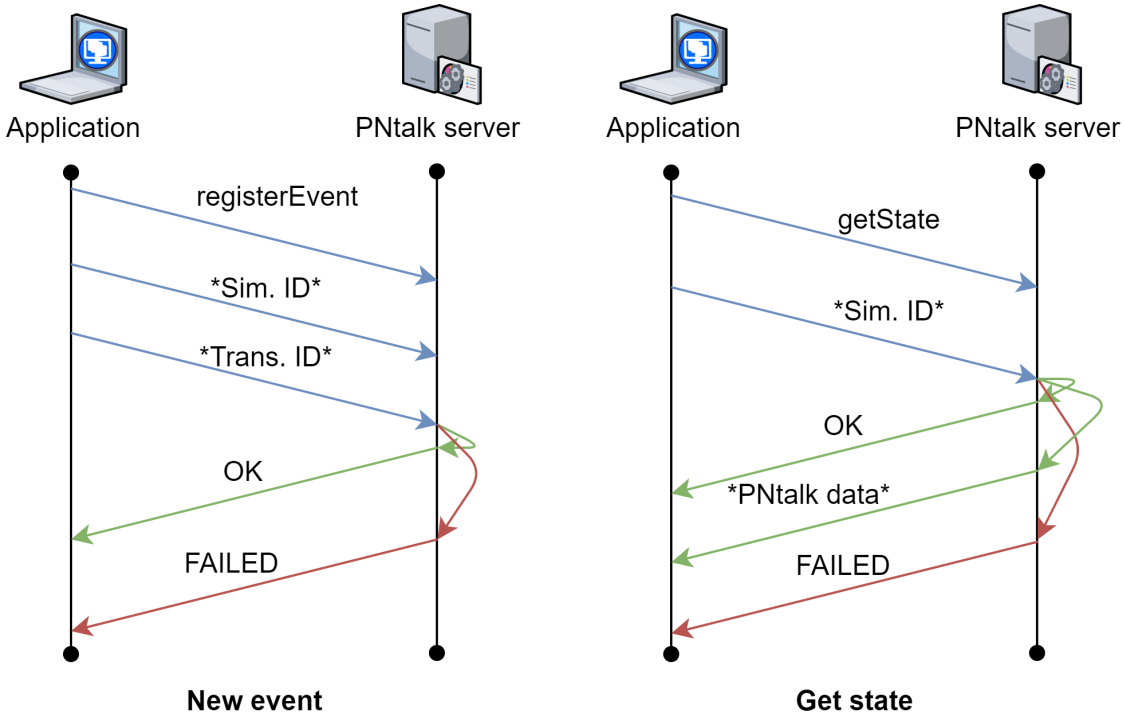
Příloha B

Komunikační protokoly s PNtalk serverem

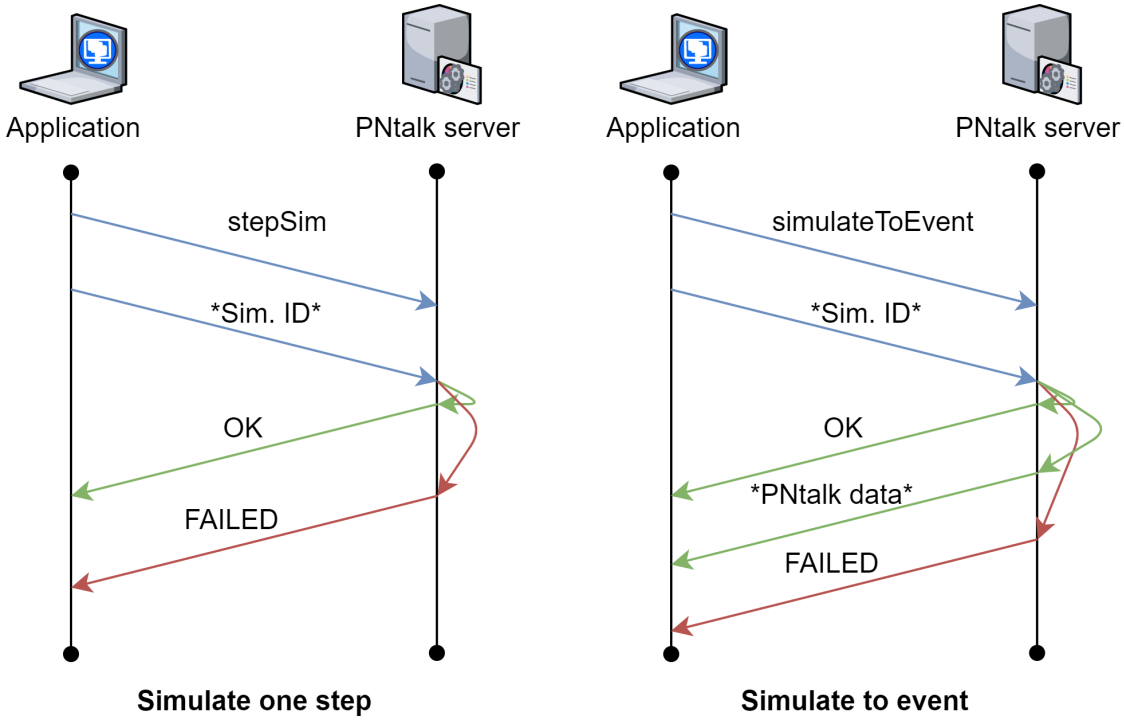
Následující příloha obsahuje obrázky všech komunikačních procesů mezi aplikací a PNtalk serverem. Každý obrázek obsahuje dva tyto procesy, které vznikají a ukončují se odděleně.



Obrázek B.1: **Komunikační protokol první část**



Obrázek B.2: **Komunikační protokol druhá část**



Obrázek B.3: **Komunikační protokol třetí část**

Příloha C

Obsah na přiloženém CD

Obsah aplikace je zabalen v archívu diplomova_prace_xneuzi05. Po jeho rozbalení se zobrazí následující obsah:

- Složka *export* s uloženými modely.
- Složka *lib* s externími knihovny.
- Složka *META-INF*.
- Složka *out*.
- Složka *screenshots* s SVG exporty.
- Složka *src* s zdrojovými kódy.
- Soubor GEOOPN_LINUX.jar spustitelný v prostředí linux.
- Soubor GEOOPN_WIN.jar spustitelný v prostředí windows.
- Soubor DP.pdf obsahuje elektronickou formu diplomové práce.
- Soubor file
- Soubor GEOOPN.iml
- Soubor README.txt s návodem instalace aplikace a jejím spuštění.