VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

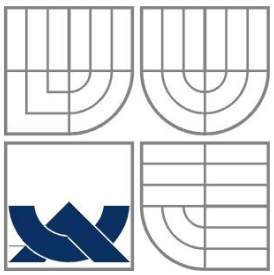
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

EDITOR OBJEKTOVĚ ORIENTOVANÝCH PETRIHO SÍTÍ

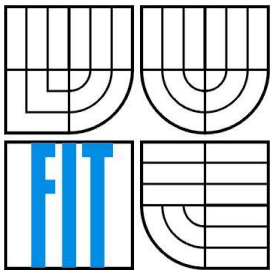
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Zoltán Kovács
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

EDITOR OBJEKTOVĚ ORIENTOVANÝCH PETRIHO SÍTÍ

EDITOR OF OBJECT ORIENTED PETRI NETS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. Zoltán Kovács
AUTHOR

VEDOUCÍ PRÁCE — Ing. Radek Kočí, Ph.D.
SUPERVISOR

BRNO 2010

Abstrakt

Práce se věnuje návrhu a implementace editoru OOPN popsaných jazykem PNTalk. Vytvoří se kompatibilní zobrazení PNTalk a formát uložení založený na technologií XML. V návrhu jednotlivých částí editoru je kladen důraz na vnitřní uložení objektů, grafického uživatelského rozhraní a funkcionalitu. V závěru práce jsou demonstrovány praktické ukázky z aplikace.

Abstract

The thesis deals with design and implementation of the editor object-oriented Petri nets described by language PNTalk. A PNTalk compatible representation and storage format based on XML technologies will be created. The design of the individual parts of the editor focuses on the internal storage facilities, on the graphical user interface and its functionality. In the final part there are practical demonstrations of the application.

Klíčová slova

Editor, PNTalk, objektově orientované Petriho sítě, vektorová grafika, PostScript

Keywords

Editor, PNTalk, object oriented Petri nets, vector graphics, PostScript

Citace

Kovács Zoltán: Editor objektově orientovaných Petriho sítí, diplomová práce, Brno, FIT VUT v Brně, 2010

Editor objektově orientovaných Petriho sítí

Prohlášení

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

……
Zoltán Kovács
26. 5. 2010

Poděkování

Rád bych poděkoval panu Ing. Radkovi Kočímu, Ph.D. za vedení mé diplomové práce, odbornou pomoc a připomínky.

Obsah

1 Úvod

V roku 1993 sa začal vývoj projektu, ktorého cieľom bolo priblížiť objektovo orientované Petriho siete (OOPN) k programovacím jazykom. Ako výsledok sa zrodil jazyk PNTalk. Časom sa vytvorilo niekoľko plnohodnotných simulačných a zobrazovacích nástrojov. PNTalk má niekoľko verzií, avšak stále chýba plnohodnotný grafický editor, pomocou ktorého by programátor bol schopný vizuálne tvoriť OOPN siete.

Práca nadväzuje na túto skutočnosť a cieľom je statický editor, ktorý pomáha tvoriť OOPN siete. To znamená, že sa musí zobraziť sieť a musí sa sprístupniť užívateľovi editácia siete. Existuje niekoľko nástrojov pre vytváranie PNTalk sietí, avšak ani jedna z nich nie je navrhnutá z vizuálneho pohľadu. Táto práca sa snaží preto brať inšpiráciu z funkcionalít vektorových editorov pri návrhu. To znamená, že vytvárať OOPN siete nebudeme pomocou kódu, ale pomocou komponentov s tým, že sa následne dokáže vygenerovať kód, ktorý odpovedá textovej reprezentácii.

Práca v sebe zahrňuje vytvorenie vlastného formátu, ktorý by mal byť kompatibilný s existujúcim formátom PNTalk, a súčasne by mal byť rozšírený o grafické informácie. Výsledný formát bude používať technológiu XML.

Práca musí mať ďalej možnosti exportu výsledného zobrazenia OOPN triedy do vektorového formátu. Doporučený formát je PostScript.

Dokumentácia bude pozostávať z niekoľkých častí. Najprv sa zoznámime s formátom PNTalk, v rámci toho s jednotlivými OOPN objektmi. Táto kapitola vlastne bude inšpirovaná dizertačnou prácou autora PNTalku Vladimíra Janouška.

V ďalšej kapitole sa budeme zaoberať existujúcimi nástrojmi, ktoré nám poskytujú podrobné popisy funkcionalít, ktoré nám budú užitočné a z ktorých sme čerpali podnet pre návrhovú časť.

Nasleduje samotná návrhová časť, kde sa už podrobne zaoberáme tým, ako sme doplnili pôvodný formát OOPN objektov tak, aby s nimi bolo možné pracovať v rámci našej aplikácie. Nasleduje popis nami vytvoreného XML formátu, a vytvorenie návrhu GUI časti. Samotná funkcionalita zaberie ďalšiu kapitolu, do ktorej patrí hlavne návrh exportovacích a importovacích funkcií a reakcie na užívateľské udalosti.

Ďalšia časť sa zaoberá samotnou implementáciou, kde sa už popisuje samotné riešenie z pohľadu programového. Túto kapitolu môžeme považovať ako rozšírenie a objasnenie programovej dokumentácie JavaDoc.

V záverovej časti komentujeme dosiahnuté výsledky a načrtneme možnosť rozšírenia aplikácie.

2 PNTalk

V tejto kapitole sa zameriame na jazyk PNTalk. Rozoberieme dôležité rysy, ktoré nám poslúžia ako základ grafických elementov.

Petriho siete slúžia ako prostriedok pre popis paralelných činností asynchrónnych udalostí. Je to matematický model založený na modelovaní kauzálnych vzťahov medzi jednotlivými udalosťami a ich vplyvom na stav systému [1]

Existuje mnoho konceptov, ako zaviesť objektovú orientáciu do Petriho sietí, my sa však v rámci diplomovej práci budeme sústrediť na jazyk PNTalk, vytvorený na FITe docentom Vladimírom Janouškom. Je to kompletná implementácia OOPN siete, a ďalej pod pojmom OOPN sieť budeme rozumieť sieť definovanú jazykom PNTalk.

Pôvodný koncept jazyku PNTalk je definovaný v Ph.D. práci docenta Vladimíra Janouška.[2] Tento jazyk poslúži ako základ nášho editora. Obecné rysy jazyka sú[3]:

- Práca s triedami s dedičnosťou,
- Objektové siete, – v každej triede je popísaná štruktúra stavov, ich inštancie a a prípadné aktívne chovanie
- Metódy– ktoré môžu byť vyvolané k asynchrónnej komunikácii s objektmi,
- Synchrónne porty – zvláštna forma prechodu aktivovaná volaním a umožňujúcu synchrónnu komunikáciu s objektmi,
- Objekty ako inštancie tried a s bežiacimi inštanciami metód,

V ďalších bodoch bude teda rozobraná definícia jazyka PNTalk, podľa pôvodnej definície. Bude kladený veľký dôraz na zachovanie vierohodnosti pôvodnej definície. [2]

2.1 Termy

Najjednoduchšie výrazy jazyku PNTalk sa volajú Termy. Delíme ich na niekoľko skupín:

1. **Literály**, ktoré definujú významné primitívne objekty:
   a) **Čísla** sú objekty reprezentujúce číselné hodnoty a reagujúce na správy, ktoré vyžadujú výsledky matematických operácií. Literál reprezentujúce číslo je sekvencia číslic, ktoré môžu predchádzať znamienko mínus. Je možné v literále použiť desatinnú čiarku, ktorá rozdeľuje sekvenciu čísiel na celú a desatinnú časť. Taktiež je možné použiť notáciu s exponentom, ktoré je vyjadrené znakom „e“ v rámci literálu.
   b) **Znaky** sú objekty, ktoré reprezentujú jednotlivé symboly abecedy. Ich literály sú uvedené symbolom dolár.
   c) **Reťazce** sú objekty reprezentujúce sekvenciu znakov. Tieto znaky reagujú na správy a požadujú prístup pre prácu s jednotlivými znakmi ako aj porovnanie s inými reťazcami. Značenie literálu reťazca je sekvencia znakov, ktorá je uzavretá v apostrofoch. Ak sa vyskytuje apostrof vnútri reťazca, tak musí byť zdvojený.

d) **Symboly** sú objekty používané ako mená. Symbol je reprezentovaný prefixom mriežky. Dva identické symboly reprezentujú ten istý objekt.

e) **Booleovské konštanty** sú reprezentované identifikátormi *true* a *false*. Tieto identifikátory sú vyhradené na tento účel.

f) **Nedefinovaný objekt** je reprezentovaný výhradne identifikátorom *nil*. Tento identifikátor je vyhradený na tento účel.

2. **Premenné** v dobe behu výpočtu môžu reprezentovať rôzne objekty, ktorých hodnota sa programovateľne dokáže meniť v čase. Premenné sú reprezentované sekvenciu znakov, ktoré začínajú malým začiatočným písmenom abecedy. Identifikátory ako *true*, *false* a *nil* sú rezervované a nemožno ich použiť ako premenné.
3. **Pseudopremenné**. Existujú dve pseudopremenné a to *self* a *super*. Ich hodnota závisí vždy na kontexte a ich hodnota nie je programovo ovplyvniteľná.

2.2 Zasielanie správ

Zasielanie správy je výraz, ktorý sa môže objaviť ako súčasť stráži aj akcie prechodu. Špeciálny prípad zasielania správy je term, ale inak má syntax správy nasledujúci charakter:

<adresát> <správa>.

Správa zostáva zo selektora a argumentov ak existujú. Podľa tvaru správy rozlišujeme tri druhy správ, a to:

1. **Unárna správa**. Selektor správy je identifikátor, ktorý obsahuje dvojbodku. Správa nemá žiadne argumenty.
2. **Binárna správa**. Selektor správy je jeden z týchto symbolov:

   + - / * = < > ~= <= >= & | // \\ , == ~==

   Po symbole nasleduje jeden argument.
3. **Správa s kľúčovými slovami**. Správa obsahuje jeden alebo viacero kľúčov, ktoré sú identifikátory ukončené dvojbodkou s vlastnými argumentmi.

Ak sú adresát aj argumenty termy, tak sa jedná o **jednoduché zasielanie správy.** PNTalk pripúšťa aj **zložené zasielanie správy,** kde ako príjemca alebo argument správy je uvedené znovu zasielanie správy. Zložené zasielanie správ sa vyhodnocuje v poradí:

4. Unárna správa – zľava doprava
5. Binárna správa – zľava doprava
6. Správy s kľúčovými slovami – zľava doprava

Toto poradie je možné zmeniť použitím zátvoriek.

2.3 Inskripce prechodov

Zasielanie správ, obecne zložené zasielanie správ sa vyskytuje vo výrazoch, ktoré špecifikujú stráže a akcie prechodov.

Stráž prechodu je tvorená sekvenciou výrazov, ktoré sú oddelené bodkou. Každý čiastkový výraz je zasielanie správy.

Sekvencia výrazov v stráži má význam konjunkcie výsledkov čiastkových výrazov. Zložené zaslanie správy je ekvivalentné so sekvenciou výrazov.

Akcia prechodov na rozdiel od stráži môže obsahovať výraz priradenia. Tento výraz má nasledujúci tvar:

<premenná> := <zaslanie správy>

Rozšírenie PNTalku oproti definícii tu pripúšťa aj sekvenciu takýchto výrazov, ktoré má význam sekvenčného prevádzania.

Rozsah platnosti mien premenných zahrňuje stráž, akcie a výrazy na okolitých hranách prechodov. U miest zahrňuje počiatočné značenie a počiatočnú akciu.

2.4 Hranové výrazy

Hranové výrazy sa vyskytujú na hranách grafov Petriho sietí a ako počiatočné značenie miest. Hranové výrazy reprezentujú multimnožinu tvaru:

n_1'c_1, n_2`$c_2$, …, $n_m$`c_m,

kde n_i je term a c_i je term alebo zoznam. Ak sa jedná o zoznam, tak tvar zoznamu je nasledujúci:

$(e_1,e_2,...,e_m)$,

kde e_i je term alebo zoznam. PNTalk pripúšťa aj zápis zoznamu známy z Prológu:

$(h_1,h_2,...,h_m \mid t)$,

kde h_i reprezentuje prvky na začiatku zoznamu a t reprezentuje zvyšok.

Na rozdiel od definície sa v zápise multimnožiny používa namiesto symbolu + čiarka. Ak je v zápise multimnožiny koeficient ni=1 môže byť vynechaný.

Prázdnu multimnožinu môžeme definovať niekoľkými spôsobmi:

0'#e

kde symbol #e používame v roli ako anonymnej, čiernej resp. bezfarebnej značky.

nil

ako druhá možnosť pre účel definovania prázdnej multimnožiny.

2.5 Miesta, prechody hrany

Siete sa v PNTalku špecifikujú graficky. Pre potreby strojového spracovania existuje možnosť textovej špecifikácie sieti.

Miesto Petriho siete je v PNTalku reprezentované kružnicou alebo elipsou. Každé miesto má jedinečné meno a môže mať **počiatočné značenie**. Podľa definície, ak nie je meno definované programátorom, tak sa priradí automaticky pri preklade. Počiatočné označenie miesta syntakticky odpovedá definícii hranovému výrazu. PNTalk rozširuje definíciu OOPN o možnosť požitia v počiatočnom mieste premennej, pričom vyčíslenie hodnôt premenných je úlohou počiatočnej akcie miesta. Táto akcia je syntakticky taká istá ako akcia prechodu.

Prechod je v PNTalku reprezentovaný štvoruholníkom. Každý prechod má jednoznačné meno, ktoré ak nie je definované programátorom, tak podľa definície sa priradí automaticky v čase prekladu. Prechod môže mať definovanú **stráž** a **akciu**. Syntax stráže a akcie bola vysvetlená vyššie. Podľa definície stráž musí byť vždy podčiarknutá.

Miesta a prechody môžu byť prepojené hranami. Ku každej hrane je pripojený **hranový výraz.** Z hľadiska prechodu môžeme rozlíšiť tri typy hrán:

Vstupná hrana, ktorá je orientovaná z miesta do prechodu. Reprezentuje vstupnú podmienku prechodu, špecifikujúcou značkou, odobranou prechodom z príslušného miesta.

Výstupná hrana, ktorá je orientovaná z miesta do prechodu. Reprezentuje výstupnú podmienku prechodu, špecifikujúcou značkou, odobranou prechodom z príslušného miesta.

Testovacia hrana, ktorá je obojstranne orientovaná a reprezentuje podmienku prechodu, ktorými prechod len testuje prítomnosť značiek v príslušnom mieste.

Rozsah platnosti mien miest a prechodov je obmedzený sieťou, v ktorej sa vyskytujú. Výnimkou sú zdieľané miesta siete objektu sietí metódy. Všetky miesta siete objektu sú viditeľné aj v sieti objektu. Mená uzlov sa uplatňujú taktiež pri inkrementálnej špecifikácie sieti, keď sa použije dedičnosť.

2.6 Siete

Miesta a prechody prepojené hranami tvoria sieť. Siete sú súčasťou špecifikácie tried objektov, komunikujúce predávaním správ. V PNTalku sú definované dva druhy sietí:

Sieť objektu reprezentuje atribúty objektu v podobe miest a jeho vlastnú implicitnú aktivitu.

Sieť metódy špecifikuje reakciu objektu na prichádzajúcu správu. Definícia triedy môže obsahovať niekoľko sietí a metód. Sieť metódy pozostáva z miest a prechodov, ktoré sú prepojené hranami, tak isto ako sieť objektu, avšak ku každej sieti metódy je priradený **vzor správy**, ktorého prijatie objektom vyvolá dynamické vytvorenie inštancie siete metódy. Vzor správy je zložený z **selektore správy** a formálnych parametrov. Podmnožiny miest siete objektu, ktoré môžu byť aj prázdne, sú parametrové miesta, ktoré slúžia k predaniu parametrov pri volaní metódy. Ich mená musia odpovedať menám formálnych parametrov vo vzore správy. Každá sieť obsahuje jedno **výstupné miesto** nazývané *return,* ktoré slúži na predanie výsledku a vyhodnotenie správy volajúcemu objektu.

Siete metód môžu byť v rámci definície triedy prepojené so sieťami objektu a hranami medzi miestami objektu a prechodmi siete metódy. Týmto spôsobom môže metóda pristupovať k dátam objektu a v prípade potreby ich zmeniť.

2.7 Synchrónne porty

Okrem siete objektov a sieti metód sú súčasti špecifikácie tried aj **synchrónne porty.** Tieto porty majú charakter prechodov aj metód. Preto tomu odpovedá grafická podoba v PNTalku Nie sú to siete, takže neobsahujú miesta ani prechody. Synchrónny port podobne ako prechod, môže byť prepojený hranami a miestami siete objektu a taktiež môže obsahovať stráž. Aj u synchrónneho portu platí, že stráž musí byť podčiarknutá. K synchrónnemu portu, je podobne ako k sieti metódy, pripojený vzor správy, na ktorú reaguje a ktorá môže byť volaná zaslaním správy zo stráže ľubovoľného prechodu.

Synchrónne porty slúžia k testovaniu a prípadnej zmene stavu volaného objektu. Môžu byť volané s dopredu vyhodnotenými parametrami, ale je taktiež možné volať ich pomocou voľných

premenných. Synchrónny port je v danom stave buď prevediteľný alebo neprevediteľný pre určité naviazanie premenných. Hľadanie vhodného naviazania premenných prebieha tak isto ako v prípade prevediteľnosti prechodov.

Špeciálny prípad synchrónneho portu je predikát, ktorý neovplyvňuje stav objektu, teda je prepojený s miestami siete len testovacími hranami.

2.8 Konštruktory

Vytvorenie objektu sa v PNTlaku realizuje zaslaním správy new príslušnej triede. Mená tried sú globálne dostupné. Implicitný konštruktor *new* je pevne zabudovaný do jazyka. Správe new rozumie každá trieda a ako reakcia na ňu tvorí inštanciu triedy, inicializovanou začiatočným stavom siete objektu.

Pre dodatočnú a prípadne parametrizovanú inicializáciu objektov slúžia špeciálne metódy nazývané **konštruktory.** Syntakticky sú podobné ostatným metódam objektu, avšak v ich špecifikácii sa objavuje kľúčové slovo *constructor***.**

Ak je v rámci definície triedy špecifikovaný konštruktor, tak ten príslušnej správe rozumie ako trieda tak aj jej inštancia. Trieda na túto správu reaguje tak, že najprv vytvorí inštanciu implicitným konštruktorom *new,* a potom zašle tú istú správu takto vzniknutom objektu. Objekt na túto správu reaguje štandardným spôsobom. Konštruktor vždy vracia hodnotu *self*, teda meno vytvoreného objektu.

2.9 Triedy a dedičnosť

Model v PNTalku pozostáva z množiny **tried.** Jedna z nich je deklarovaná ako **počiatočná trieda**. Tá je implicitne inštancovaná pri zahájení simulácie modelu.

Trieda je zložená zo siete objektov, množiny sietí metód konštruktorov a synchrónnych portov.

Každá trieda OOPN má práve jedného predchodcu v hierarchii dedičnosti. Vrchol hierarchie tried definovaných Petriho sieťami je trieda *PN,* ktorá sama už nie je popísaná Petriho sieťami. Jej sieť objektov je prázdna a je priamym následníkom triedy *Object*, ktorá tvorí vrchol hierarchie všetkých tried PNTalku. Okrem abstraktnej triedy *PN* sú ďalšími následníkmi abstraktnej triedy *Object* taktiež triedy primitívnych objektov PNTalku. Nazývajú sa tiež ako konštanty, z ktorých sú jazykovo dostupné ako literáli PNTalku.

Abstraktná trieda Objekt definuje reakcie na správy, ktorými musia rozumieť všetky objekty. Ide hlavne o porovnávanie identity objektov.

Každá trieda dedí od svojej nadtriedy štruktúru siete objektov a všetky metódy, konštruktory a synchrónne porty. Pritom všetko čo zdedí trieda od triedy PN a ich potomkov je možné predefinovať. Metódy zdedené od triedy *Object* však predefinovať nemôžeme. Metódy, konštruktory a synchrónne porty môžu byť predefinované uvedením nových definícií pre ten istý selektor ako u nadtriedy.

Sieť objektu môže byť predefinovaná po častiach redefinícou jednotlivých prechodov a miest, teda uvedením miest a prechodov uvedených tým istým názvom ako v nadtriede. V prípade redefinície prechodu musí byť znovu uvedená aj definícia okolitých hrán. V prípade redefinície miesta hrany zostávajú nezmenené. Toto odpovedá tej skutočnosti, že podľa definície hrany sú súčasťou prechodov.

Okrem redefinície zdedených metód, konštruktorov, synchrónnych portov a uzlov siete objektov, môžu byť tieto prvky pridávané, teda prepojované s zdedenými prvkami siete objektu.

V grafickej reprezentácii OOPN môžeme ale nemusíme zobraziť zdedené prvky nadtriedy, ktoré nie sú priamo prepojené s novo definovanými.

3 Vektorové editory

Nakoľko sa práca zaoberá hlavne vytvorením editora, ktorý dokáže manipulovať s OOPN sieťami, musíme sa zamierať všeobecne na už existujúce editovacie nástroje. Táto kapitola nám dá jednak základ pre estetický a prehľadný návrh užívateľského rozhrania, a na druhej strane komfortný formát pre navrhnutie a uloženie dát.

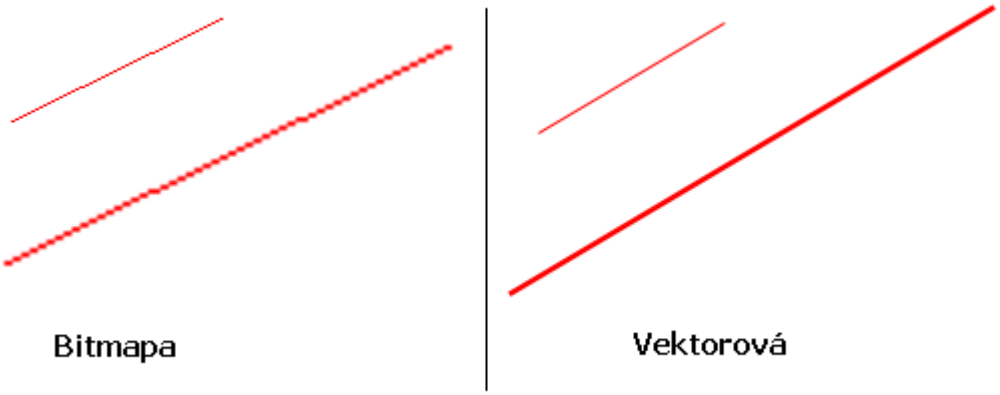
3.1 Grafika

V tejto časti si rozoberieme grafické typy, ktoré sú potrebné pre pochopenie a pokračovanie v našej práce. Nakoľko potrebujeme jednotný formát, ktorý je dôležitý pre uchovanie informácií a následne pre zobrazenie celkového kontextu, musíme sa najprv zoznámiť s existujúcimi formátmi a technikami, aby sme z nich mohli čerpať.

3.1.1 Rastrový formát

Rastrová grafická informácia spočíva v tom, že body sa ukladajú jednotlivo po grafickej mriežke. Každý bod má vlastné parametre, ako súradnice, že presne o ktorý bod sa jedná, informáciu o farbe a iné atribúty. Na tomto princípe pracujú skoro všetky zobrazovacie zariadenia. Obrázok sa vykresľuje ako celok a nenesie sémantické informácie. Je to potom na pozorovateľovi, aby z celkového obrazu vystihol informácie pre neho potrebné. Rastrová grafika je objemovo náročná na ukladanie, nakoľko musíme všetko do posledného detailu uložiť zvlášť. Existujú síce kompresné metódy, ktoré nám napomáhajú pri redukcii objemu dát, avšak veľkosť dát môže byť ešte stále obrovská.

Ďalšou nevýhodou je, že ak rastrovo popísaný obrázok zväčšíme, tak sa rozmazáva. Čím väčšie je rozlíšenie tým sa rozmazanie prejaví pri väčšom priblížení.



Obrázok 4.1: Rozdiel medzi bitmapovým a vektorovým obrázkom

3.1.2 Vektorový formát

Vektor je popísaný pomocou matematických funkcií. Tieto matematické funkcie sa používajú pri vykresľovaní vektorového formátu. Objekty popísané matematickými funkciami nazývame grafické primitíva. Obsahujú presne definované grafické prvky, ako sú krivky, a rôzne tvary, ako mnohouholníky či elipsy. Zoznam grafických primitív je nasledujúci:

- body
- úsečky a segmenty priamok
- plochy
- kružnice elipsy a kruhy
- trojuholníky alebo obecne iné polygóny

Každý grafický primitív má presne definovanú farbu. Každej z týchto častí sa priradí kódová informácia o tom ako má byť zobrazená, aký útvar definuje. Taktiež je možné tieto grafické primitíva kombinovať a dostať tak zložitejší tvar.

Grafické primitíva sú pamäťovo nenáročné na uloženie, ale náročné na výpočet. Keďže väčšina zobrazovacích jednotiek používa rastrové zobrazenie, tak pri zobrazovaní sa vypočítavajú rastrové údaje. Výhodou je, že keď sa zmení veľkosť zobrazovania, obrázky ostávajú ostré.

Ďalším kladom je, že uloženú kresbu je veľmi jednoduché editovať. Stačí zmeniť parametre ako kód funkcie, kód použitej farby, alebo spôsob zobrazenia. Nezanedbateľné je, že s jednotlivými objektmi pracujeme ako celkom, popritom ostatné objekty zostávajú nezmenené.

Úpravy, ktoré dokážeme aplikovať na vektorové objekty sú nasledujúce:

- otočenie
- posun
- prevrátenie
- rozťahanie / zúženie
- skosenie
- obecná afinitná transformácia (posunutie pod uhlom)
- zmena Z súradnice
- kombinácia primitív do komplexných objektov
- spojenie
- rozdelenie
- prienik

Pomocou grafických primitív a operáciami s nimi sa dá dosiahnuť použiteľný obraz. Vektorovú grafiku je teda možné použiť všade, kde nie je potrebné realistické zobrazenie.

3.1.3 Formáty

Existuje niekoľko súborových formátov ako vyriešiť uloženie grafických primitív a príslušné transformácie. Keďže sa jedná o takú zobrazovaciu formu objektov, ktorá je nezávislá na zobrazovacom zariadení, tak sa vývoj tohto formátu stal dôležitým, keď sa počítačové videnie začalo z textového zobrazovacieho formátu preorientovávať na obrázkové.

3.1.3.1 PostScript

PostScript[6], sa považuje za programovací jazyk. Slúži pre grafický popis dokumentov. Pôvodný koncept bol navrhnutý v 80. rokoch 20. Storočia. Vtedy sa považoval za revolučný pre prístup, pretože zobrazovanie nezávisí na zariadení. Zjednotil formát tlače dokumentov. Dovtedy sa

uchovávali len ASCII znaky, a každá tlačiareň ich interpretovala inak. Riešenie celého formátu tak bolo evidentné a jednoznačné.

PostScript prešiel cez svoj vývoj troma fázami. Pôvodný návrh bol zameraný na fonty písmen, a ich zobrazenie. Neskôr sa doplnila podpora bitmapových obrázkov, a súčasné riešenie rozšírilo možnosti o paletu farieb, tvarov, transformácií a zobrazovacích možností.

Licenciu PostScript formátu drží Adobe, a naliezame preto silný vplyv PostScriptu v obľúbenom PDF formáte. PDF alebo Portable Desktop Format pridáva aj použité fonty k textu, aby sa zobrazenie celkového dokumentu na všetkých platformách javilo ako také isté. Je to jeden z najobľúbenejších formátov, hlavne pre uloženie textu.

PostScript je zameraný na zobrazovanie textu, avšak tento formát sa rozšíril o možnosť reprezentácie ľubovoľného vektorového grafického formátu. Tento formát sa menuje Encapsulated PostScript[7]. Celková funkčnosť je prevedená tak, že do PostScriptu je zapuzdrený obrázok, ktorý sa zobrazí ako bitmapa pri nahliadnutí.

EPS formát používa programovací jazyk PostScript, pomocou ktorého sa najprv definujú medze obrázku, ktorý má byť zakreslený do neho. Táto definícia sa javí ako rámec celej grafickej časti, a všetko čo je mimo toho sa zahadzuje. Táto definícia je povinná v hlavičke súboru.

Ďalej pomocou grafických primitív sa definujú jednotlivé geometrické objekty. Môže sa použiť transformácia, a rôzne úpravy, ktoré sú zaznačené vo formátovacom texte. Niektoré operátory sú zakázané, iné striktne obmedzené.

Celkový formát je zdrojový kód programovacieho jazyka, a vyžaduje tak program, ktorý dokáže z údajov vykresliť obrázok. Existuje mnoho konvertovacích nástrojov pre konverziu EPS formátu do rastrového alebo do iného vektorového. Preto formát EPS ako aj PostScript obecne je použiteľný aj v rámci tejto práce, ako vhodný výstup pre export vykresleného výsledku.

3.1.3.2 Ostatné

Okrem formátu EPS existuje mnoho iných vektorových formátov, ktoré sú založené na ukladaní primitívnych objektov s atribútmi a prevedenými úpravami s nimi. Ako najznámejší je formát Scalable Vector Graphics[8]teda SVG. Tento formát sa s veľkou obľubou používa na prezentáciu na webe. Je založený nad XML a bol vyvíjaný konzorciom W3C. Skoro všetky webové prehliadače majú natívnu podporu pre tento formát, takže je možné zamyslieť sa aj nad implementáciou exportu do tohto jazyka.

Existuje ešte rada ďalších formátov, ktoré sú však väčšinou združené ku komerčným vektorovým editorom. Zoznam je nasledovný:

- AI – natívny súbor pre Adobe Illustrator
- CDR – natívny súbor pre CorelDraw!
- SWF – exportovaný finálny súbor z FLA – Macromedia Flash

3.2 Editory

Máme definovanú funkčnosť vektorových editorov, a máme všeobecný prehľad o formátoch. Existuje niekoľko pohodlných a jednoduchých nástrojov, ktoré používajú klasické aj revolučné riešenia či rozloženia. Rozoberme ich z hľadiska funkčnosti, funkcionality a výzoru - niektoré konkrétne. Pri pojme editor sa však nemusí jednať o vyslovene grafické editory, nakoľko navrhujeme editovací nástroj pre programovací jazyk. Tieto informácie o Existujúcich prvkoch nám pomôžu pri vytváraní nášho editora.

3.2.1 Corel Draw

CorelDraw je jeden z prvých naozaj úspešných vektorových editorov. Od roku 1989 sa uplatňuje vo vektorovej grafike. Prvé verzie editora rozkladali jednotlivé objekty na grafické primitíva, a tie potom rozložili po dvojdimenziálnom súradnicovom priestore a to po ose X a Y. Používal sa popri základných úpravách objektov aj spojenie i kombinovanie objektov, aby sa dosiahli zložitejšie tvary.

Mal niekoľko verzií a vyvíja sa do dnes. V súčasnosti je najnovšia verzia X5, ktorá je pätnásta v poradí. Čo nás z CorelDraw-u zaujíma, to je hlavne užívateľské rozhranie a práca s ním. Ako pri väčšine grafických programov je rozloženie tlačidiel jednoduché. Na hornej časti sa nachádza štandardný nástrojový panel. Tento nástrojový panel zahrňuje ikony ako

- možnosť vytvorenia nového výkresu
- otvorenie existujúceho výkresu
- uloženie existujúceho výkresu
- tlačidlá kopírovania vystrihnutia a vloženia,
- tlačidla pre späť a dopredu
- tlačidlá pre importovanie a exportovanie
- ďalšie voliteľné a doplňovacie nastavenia

Ďalší nástrojový panel vidíme na pravej časti editora. Tu sú zahrnuté všetky nástroje, ktoré potrebujeme k samotnej práci s pracovnou plochou. Nástroje sa v jednotlivých verziách menili, ale tie najzákladnejšie boli vždy zahrnuté. Zoznam najdôležitejších nástrojov si rozoberieme v nasledujúcich bodoch.

Nástroj výberu

Nástroj výberu slúži na vybranie existujúceho objektu na pracovnej ploche. Keďže sme si vedomí toho, že grafické primitíva ako čiara, elipsa mnohouholník sa chová ako jeden objekt, tak tento nástroj umožňuje manipuláciu s ním.

Nástroj tvarovania objektu

Nástroj tvarovania objektu je určený na to, aby sa doladili primitívne tvary. Dokážu sa pridať nové body k existujúcim tvarom, napríklad z jednoduchej úsečky dokážeme spraviť trojuholník, alebo sekvenciu úsečiek.

Nástroj čiary

Nástroj čiary je jednoduchý nástroj, ktorý slúži pre vytvorenie úsečky, a rôznych mutácií.

Nástroj obdĺžnika

Nástroj obdĺžnika je jeden z najzákladnejších a najpoužívanejších primitívnych tvarov. Definuje sa kliknutím od jedného rohu podržaním a pustením v druhom protiľahlom rohu. Tieto dva údaje stačia na to, aby sa vytvoril obdĺžnik.

Nástroj elipsy a kruhových tvarov

Elipsa sa kreslí tým istým spôsobom ako obdĺžnik. Tvar však dostáva z obálkového charakteru. To znamená, že sa berú dve protiľahlé súradnice, jedna pri stlačení myši a druhá pri pustení, o ktoré sa nakreslí imaginárny obdĺžnik. Do tohto obdĺžnika sa vkreslí elipsa tak, že sa priamo dotkne zvnútra k stene obdĺžnika.

Nástroj mnohouholníkov

Nástroj mnohouholníku je tvar, ktorý sa podobne ako elipsa vmestí do imaginárnej obálky.

Nástroj textu

Nástroj textu slúži pre vykreslenie textu. Text defaultne je definovaný fontmi, podobne ako v textových editoroch, a nie grafickými primitívmi. V prípade potreby je možné previesť text jednosmerne na primitívne tvary a editovať ich na grafickej úrovni.

Nástroj výplne

Vo vektorovej grafike je možnosť uzavretým objektom priradiť informácie o farbe výplni. Výplň sa pritom pridáva k vybranému objektu.

Nástroj obrysu

Nástroj obrysu je podobne ako nástroj výplne taký nástroj, ktorý len modifikuje atribúty už existujúceho objektu. V tomto prípade sa zameriava na definovanie najmä:

- hrúbky obrysu,
- farby obrysu
- ukončenie obrysov, čo znamená, že môžeme z dopredu definovaných tvarov určiť ako má vyzerať koniec. Je tak možnosť pridania napríklad šípky na koniec obrysu.

3.2.2 Microsoft Visual Studio

Ako ďalší editor vystihneme Microsoft Visual Studio. Visual Studio je nástroj vývojára. Jedná sa o integrované vývojové prostredie, čo znamená, že podporuje mnoho riešení a programovacích jazykov a pritom sa nejedná len o funkcie editácie, ale aj o možnosti spúšťania a ladenia.
Moderné vývojové nástroje už väčšinou dokážu spolupracovať s návrhom GUI aplikácie a kódom súbežne. V rámci nášho projektu rozoberieme Visual Studio z pohľadu funkčnosti a samotné jeho užívateľské rozhranie.

Vieme, že Visual Studio je vyostrené na riešenia Microsoft. Vymedzuje širokú škálu možností od návrhu konzolových aplikácií cez kompletné programy v C++ alebo .NET až po návrh rôznych pluginov až po mobilné alebo webové riešenia. Toľko mnoho možností je síce veľmi úctyhodných, avšak pre nás nezaujímavých. Zamerajme sa v ďalších bodoch na rysy, ktoré budú inšpiráciou pre náš projekt. Jedná sa hlavne o možnosti editácie a následného generovania kódu z vizuálnej časti a z editácie a zobrazenia príslušných vlastností. Táto časť sa menuje dizajnérska časť a v ďalších bodoch budú rozobrané hlavné rysy tejto časti.

Panel nástrojov

Pri vytvorení nového projektu musíme vždy určiť presne o aký projekt sa jedná. Ak si zvolíme taký typ projektu, ktorý implementuje GUI, tak sa ako prvé zobrazí vizuálne okno na novom ušku. Na pravej strane máme nástrojový panel, ktorý nám dáva možnosť pridania nových komponentov.

Panel hierarchie

Pre už vytvorené komponenty Visual Studio, podobne ako mnoho iných IDE nástrojov, ukladá jednotlivé komponenty do úrovne hierarchií. Takýto hierarchický panel nájdeme na pravej strane vývojového prostredia. Každé okno má svoju vlastnú hierarchiu, a môže byť pod nimi niekoľko komponent.

Tabuľka vlastností

Ako vieme každý komponent môže disponovať celou škálou vlastností. Medzi tieto vlastnosti patria napríklad atribúty ako meno komponentu, ikona, farba pozadia alebo defaultná akcia pri manipulácii s komponentom. Tieto vlastnosti sa vo Visual Studiu zapisujú do tabuľky vlastností, ktorý je na ľavej strane užívateľskej aplikácie. Tabuľka má dva stĺpce. Prvý stĺpec reprezentuje názov vlastnosti a druhý stĺpec hodnotu. Hodnotu je možné dynamicky upravovať, a tak zmeniť komponent.

Ostatné časti

Ako sme už spomenuli, tak Visual Studio je integrované vývojové prostredie, a teda má v sebe veľmi veľa užitočných možností a okien pre kompiláciu, spúšťanie, ladenie a vývoj aplikácie. Tieto možnosti môžu byť zaujímavé ak uvažujeme o vývoji nášho editora ako plnohodnotného vývojového prostredia.

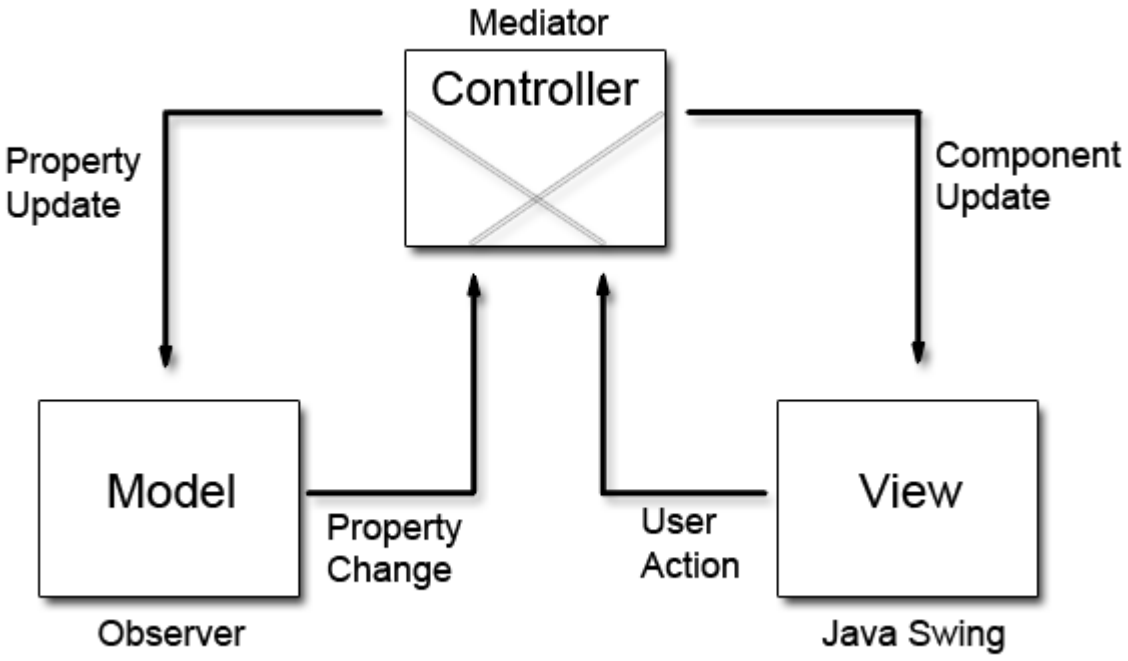
4 Špecifikácia a návrh editoru

V návrhovej fáze sa musíme sústrediť na to, ako by mal výsledný program vyzerať, zamerať sa na detaily, vymyslieť a presne definovať dátové štruktúry. Základom dobrej aplikácie je dobrý návrh, takže tejto kapitole budeme venovať veľký dôraz.

V časti analýzy sme sa zoznámili s požiadavkami, ktoré zvyknú byť špecifikované zákazníkom, v našom prípade našim konzultantom, ktorý danú aplikáciu bude používať, avšak to, ako to celé bude implementované už nemusí byť pre koncového zákazníka až tak dôležité. Návrhová časť však musí byť dôležitá hlavne preto, aby sa kód aplikácie mohol doladiť poprípade rozšíriť o ďalšie funkcie a moduly. A taktiež, aby projekt mohol byť ďalej rozširovaný iným riešiteľom.

Pri návrhu aplikácie budeme postupovať tak, ako pri návrhu ľubovoľného grafického editoru. Prístup k návrhu bude mať dve hlavné časti. Prvá časť bude zameraná na interné zobrazenie, uchovanie grafických primitív ako objektov OOPN. Tu sa zameriame na to, aby tie objekty niesli minimum informácií, avšak všetko potrebné k vykresleniu celkovej OOPN siete na ľubovoľný grafický kontext. Druhá časť návrhu bude pozostávať zo samotného editoru, teda možnosťami vykreslenia objektov a zásahom do OOPN siete, či už pridávaním, vymazaním alebo editáciou OOPN objektov.

Výsledné riešenie by malo používať obdobu modelu view kontrolóra (MVC), kde model odpovedá internému uloženiu OOPN objektov, View by bola časť GUI a kontrolór by bola funkcionalita.



Obrázok 5.1: MVC model

4.1 Špecifikácia požiadaviek

Zo zadania diplomovej práce a následných konzultáciách sme vytvorili zoznam požiadaviek na funkčnosť výsledného editora. Tieto požiadavky sú zhrnuté v nasledujúcich bodoch.

- Zobrazenie OOPN Triedy a jej objektov
- Možnosti pridávania, odoberania a editovania OOPN objektov na grafickej úrovni.
- Vytvorenie príslušného formátu pre uloženie a načítanie siete . Formát musí byť založený na technológii XML
- Import z PNTalk jazyka
- Export do PNTalk jazyka
- Export to vektorového grafického formátu. Podľa pokynov musí byť použitý formát PostScript.

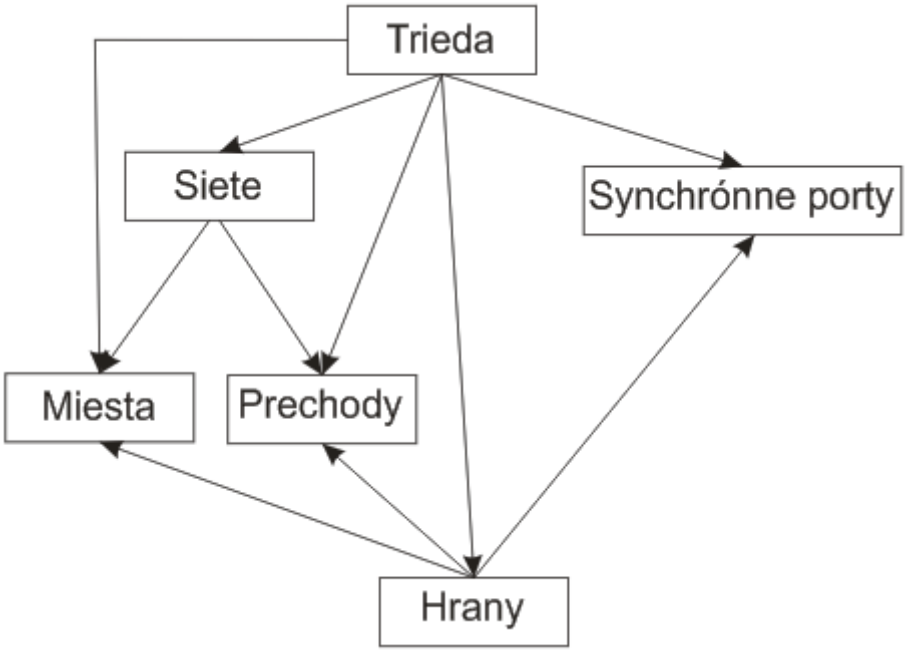
Ďalej sa budeme zaoberať s návrhom editora tak, aby sa stanovené požiadavky premietli do výsledného riešenia.

4.2 Interné zobrazenie objektov

Z teoretickej časti PNTalku máme dostatočné vedomosti k tomu, aby sme dokázali OOPN objekty dať do takého kontextu, aby s nimi náš editor dokázal pracovať čo najefektívnejšie.

Najprv sa však musíme zamyslieť nad tým, že aká jednotka celkového PNTalk systému je ešte použiteľná ako celok pre zobrazenie. Odpoveďou je jednoznačne to, že najefektívnejšie pre zobrazenie je zobraziť jednu triedu aj so všetkými zdedenými OOPN objektmi. Tento prístup bude preferovať aj interné zobrazenie objektov. Teda musíme do interného zobrazenia vložiť všetky informácie OOPN triedy. Bol by možný aj prístup že by sa uchovala len sieť, avšak ťažko by sa pri takomto prístupe uchovávali hrany, ktoré spájajú OOPN objekt vo vnútri siete s objektom mimo.

Samotná implementácia bude vykonaná v jazyku Java. Môžeme tak uvažovať o tom, že pre OOPN Objekt sa vytvorí samostatná trieda, ktorá bude definovať, podobne ako v PNTalku, jednotlivé objekty. Tieto triedy sa potom môžu inštancovať a tým sa vytvárajú samotné OOPN objekty, ako miesta, prechody, hrany, synchrónne porty, konštruktory či metódy.



Obrázok 5.2:Náčrt referencií jednotlivých OOPN elementov

4.2.1 Siete

Jedná sa o definíciu OOPN siete. Ako vieme trieda v OOPN definuje niekoľko sietí. Podobne ako funkcie pri funkcionálnom, alebo metódy pri objektovo orientovanom programovaní, aj v tomto prípade majú vlastný vymedzený definičný priestor.

4.2.1.1 Typ

Typ objektu je používaný k rozlišovaniu podobných typov. Ako sieť môže byť Objektová sieť, ktorá pokrýva celý definičný priestor triedy. Je jedinečná v rámci triedy.

Ako ďalší typ siete môže byť metóda. Metóda je previata z definície PNTalku a plne odpovedá jej funkcionalite.

Ako ďalší typ je použitý konštruktor, ktorý sa považuje taktiež za sieť. Technicky to sieť nie je, avšak ak použijeme istú abstrakciu, tak ju môžeme považovať za sieť. Definícia konštruktora dokonca vystihuje, že sa jej zobrazenie líši len použitím kľúčového slova.

4.2.1.2 Meno

Sieť musí byť jednoznačne identifikovateľná v rámci OOPN triedy. K tomu poslúži meno, ktoré ho robí jedinečným. Meno sa skladá zo vzoru správy, presnejšie zo sektoru správy a formálnych parametrov definovaných v PNTalku.

Výnimkou je sieť typu Objektová sieť, ktorá v sebe má zabudované ako meno, meno triedy.

4.2.1.3 Grafické informácie

Siete implicitne nenesú v sebe žiadne grafické informácie, ktoré by sa mali uchovať. Avšak budeme musieť previesť výpočet grafického zobrazenia, odvodených z objektov spadajúcich do definičného priestoru siete.

4.2.2 Miesta

Miesto považujeme za jeden zo základných konštrukčných kameňov pri vytvorení grafického znázornenia OOPN triedy. To je dané tým, že tento objekt je znázornený samostatne, a nie je závislý od okolitých objektov. V nasledujúcich častiach si vymedzíme hlavné rysy a atribúty, ktoré by sa jednoznačne mali objaviť v internom zobrazení miesta.

4.2.2.1 Meno

Každé miesto má jednoznačné meno. Toto meno je v rámci definovaného kontextu jedinečné. Keďže miesto môže byť súčasťou aj samotnej objektovej siete alebo rôznych iných sietí, napríklad siete metódy či konštruktora, tak kontext mena sa môže meniť. Ak je definované miesto v metóde pomocou mena, tak je viditeľné v rámci metódy a v objektovej sieti. V ostatných sieťach metód alebo konštruktorov už nie je viditeľné. Toto sa vzťahuje napríklad pre miesto *return*.

Z pohľadu návrhu má táto vlastnosť za účel to, že v rámci rôznych metód sa môže použiť to isté meno a bude pri tom značiť úplne odlišný OOPN objekt, v tomto prípade miesto. Dôležitá skutočnosť je, že nemôžeme meno chápať ako jednoznačný identifikátor OOPN miesta v rámci celého interného zobrazenia, keďže sa zobrazuje trieda.

4.2.2.2 Nadradená sieť

Aj z dôvodu, že miesto nemôže byť samostatne menom jednoznačne definovateľné, musíme použiť referenciu na sieť. Táto sieť je buď samostatná objektová sieť, alebo metóda či konštruktor. Pomocou tohto údaja môžeme rozšíriť identifikáciu miesta tak, aby sieť a meno jednoznačne vymedzovali OOPN miesto.

4.2.2.3 Počiatočné značenie

OOPN miesto má ďalší atribút, a to počiatočné značenie, ktoré je však voliteľné. To znamená, že sa vôbec nemusí vyskytnúť žiadna hodnota. Počiatočné značenie je definované vlastne ako hranový výraz. Otázkou je, ako definovať v jazyku Java toto počiatočné značenie. Java na rozdiel od SmallTalku má dopredu definované dátové typy a potrebujeme volať nejakú prevádzaciu funkciu, aby sme prekonvertovali číselný typ na reťazový. Vieme z definície, že počiatočné miesto môže byť literál, a preto najjednoduchšie je použiť reťazový dátový typ pre uchovanie tejto hodnoty. Ďalšia dôležitá otázka je, ako sa má chovať tento atribút ak nie je definované počiatočné značenie. Použiť null ukazovateľ nie je najlepšie z pohľadu neskoršieho spracovania, tohto údaju, takže najlogickejšie je použiť prázdny reťazec.

4.2.2.4 Typ

Pri definícii miesta ako aj pri ostatných objektoch definujeme tento identifikátor. Vždy bude definovaný na typ miesto. Je však tento identifikátor dôležitý, ak zovšeobecníme OOPN objekty a chceme sa dozvedieť, s akým objektom práve pracujeme.

4.2.2.5 Grafické informácie

Z definície vieme, že miesto sa graficky značí elipsou alebo kružnicou. Berme do úvahy, že keď výšku a šírku elipsy nastavíme na tú istú hodnotu, tak dostávame kružnicu. Môžeme tak teda pristúpiť k tomu, že každé miesto budeme značiť elipsou. V Jave sa pre vykreslenie elipsy používajú 4 hlavné údaje:

- súradnica, ktorá značí ľavú hornú súradnicu na ose X

- súradnica, ktorá značí ľavú hornú súradnicu na ose Y
- výška elipsy.
- šírka elipsy

Tieto údaje by nám mali postačovať pre jednoznačné určenie polohy a veľkosti miesta na grafickom kontexte. Ako otázka je tu tiež predurčenie použitia typov v Jave. Keďže sa jedná o číselné údaje, mali by sme sa rozhodnúť medzi typom celočíselným alebo s pohyblivou rádovou čiarkou. Z dôvodu, že vykreslenie grafických primitív používa celé číslo, tak aj pre uchovanie našich grafických informácií použijeme int typ.

Je tu jedna dôležitá časť, s ktorou sa musíme zaoberať. Jedná sa o to, že súčasťou grafického zobrazenia objektu je aj názov objektu a počiatočné značenie. Vykreslenie týchto údajov musí byť závislé na pozícii celkového miesta. Netvoríme úplne voľný grafický editor, kde by sme mohli text uložiť tam kde chceme, ale zavedieme striktnú definíciu pozície textov a to nasledovne:

- Meno sa vždy zobrazí v pravej hornej časti miesta.
- Počiatočné značenie je uprostred elipsy

Dôležité je, že ak je veľkosť elipsy príliš malá na to, aby sa do nej zmestil vykreslený text, tak sa veľkosti dynamicky zväčšia a uložia do interného zobrazenia.

4.2.3 Prechody

Podobne ako miesta, tak aj prechody sú z grafického hľadiska najzákladnejšie objekty. Prechod nie je závislí od okolitých objektov a preto sa môže zobrazovať na grafickom kontexte samostatne.

4.2.3.1 Meno

Taktiež ako miesto, aj prechod má meno, ktoré ho jednoznačne definuje v rámci siete. Avšak stretávame sa aj v tomto prípade s tým, že meno samostatne nedokáže vymedziť OOPN objekt v rámci celej triedy, teda ani v našom internom zobrazení triedy. Táto vlastnosť je daná znovu tým, že aj prechod môže byť súčasťou metódy a konštruktora, ktoré mu dávajú charakter akoby zaškatuľkovaného do bloku.

4.2.3.2 Nadradená sieť

Keďže vieme, že prechod musí byť súčasťou niektorej siete, buď objektovej siete alebo siete metódy či konštruktora, musíme tento údaj uchovať v internom zobrazení prechodu. Účelom je jednoznačná identifikácia OOPN prechodu.

4.2.3.3 Stráž

Podľa definície, stráž je sekvencia príkazov oddelených bodkou. Pri vizuálnom zobrazení je dôležité, aby sme si všimli, že všetky tieto príkazy sa chovajú samostatne. Otázkou je, či uložiť tieto príkazy spoločne ako stráž alebo jednotlivo ako pole príkazov. Pre jednoduchšiu reprezentáciu je jednoduchšie uložiť pole príkazov, avšak pri ukladaní, editovaní a konvertovaní sa zdá efektívnejšie použiť prístup uloženia stráže ako celok, teda sekvenciu príkazov.

Pre uloženie sa použije dátový typ reťazec kvôli dynamickým vlastnostiam a univerzalite uložených dát.

4.2.3.4 Akcia

Pri prístupe návrhu uloženia akcie do vnútorného zobrazenia sa budeme riadiť prístupom ako pri stráži. V definícii je podotknuté, že akcia má podobnú syntax ako stráž, avšak je obohatená

o možnosť priradenia. Jednotlivé výrazy sú znovu oddelené bodkou a spracovávajú sa sekvenčne. Ukladať ich budeme do dátového typu reťazca.

4.2.3.5 Typ

Podobne ako pri mieste, tak aj pri prechode je nutné definovať typ. Typ bude vždy jednoznačne prechod, ale kvôli zovšeobecneniu tu musí byť.

4.2.3.6 Grafické informácie

OOPN objekt prechod sa graficky zobrazuje pomocou obdĺžnika. Je jednoznačné, že pre grafické zobrazenie obdĺžnika potrebujeme uchovať 4 údaje.

- súradnica, ktorá značí ľavý horný roh obdĺžnika na ose X
- súradnica, ktorá značí ľavú horný roh obdĺžnika na ose Y
- výška obdĺžnika
- šírka obdĺžnika

Použitý dátový typ pre uchovanie týchto údajov tak znova bude tvoriť int. Dôvodom je, že funkcia v Jave pre vykreslenie grafických primitív používa int, takže spracovanie je efektívnejšie, keď sa údaj udáva v celom čísle definujúcom pixle.

Aj pri grafickej reprezentácii prechodu musíme dbať na skutočnosť, že prechod ako grafický celok v sebe má zahrnuté meno, stráž aj akciu. Pri grafickom znázornení budeme preferovať nasledujúce body:

- Meno sa vždy zobrazí v pravej hornej časti miesta.
- Stráž sa nachádza vnútri obdĺžnika, a dodržuje odstup pár pixlov od steny. Stráž musí byť podčiarknutá, aby sa jednoznačne odlíšila od akcie.
- Akcia sa nachádza vo vnútri obdĺžnika pod strážou. Text nie je však nijak výrazne modifikovaný.

Aj tu platí zásada, že sa veľkosti obdĺžnika dynamicky môžu zväčšovať, ak meno, stráž alebo akcia presahuje aktuálne hranice.

4.2.4 Synchrónne porty

Synchrónne porty z hľadiska zobrazenia sú taktiež základným a samostatným grafickým prvkom. Ich veľkosť a umiestnenie je jednoznačné v rámci celej triedy.

4.2.4.1 Meno

Synchrónne porty sa taktiež definujú jednoznačným identifikátorom, ktorým je meno. V tomto prípade, keďže sa synchrónny port môže nachádzať iba mimo sietí, je toto meno jedinečné v rámci celej triedy. Nemusíme sa teda zaoberať s nadradenou triedou.

4.2.4.2 Stráž

U synchrónnom porte použijeme taktiež definíciu, stráže. Je jednoznačné, že použijeme ten istý prístup ako sme použili pri stráži definovanej u prechodu. Pre uloženie dátového typu teda použijeme reťazec s tým, že stráž sa uloží ako celok.

4.2.4.3 Typ

Pri definovaní typu synchrónneho portu zvolíme konštantu synchrónneho portu a to preto, aby sme pri zovšeobecnení OOPN objektu dokázali jednoznačne určiť typ.

4.2.4.4 Grafické informácie

OOPN objekt synchrónny port sa graficky zobrazuje pomocou šedého obdĺžnika. Je jednoznačné, že pre grafické zobrazenie obdĺžnika potrebujeme uchovať 4 údaje.

- súradnica, ktorá značí ľavý horný roh obdĺžnika na ose X
- súradnica, ktorá značí ľavý horný roh obdĺžnika na ose Y
- výška obdĺžnika
- šírka obdĺžnika

Grafické znázornenie synchrónneho portu teda v sebe zahrňuje aj meno aj stráž. Zobrazovanie používa znovu pravidlá že:

- Meno sa vždy zobrazí v pravej hornej časti miesta.
- Stráž sa nachádza vnútri obdĺžnika, a dodržuje odstup pár pixlov od steny. Stráž musí byť podčiarknutá.

Aj tu platí zásada, že sa veľkosti obdĺžnika dynamicky môžu zväčšovať, ak meno alebo stráž presahuje aktuálne hranice.

4.2.5 Hrany

Keď navrhujeme prechod a synchrónny port, tak sa môžeme zamyslieť nad otázkou, či hrany máme zahrnúť do definície týchto objektov, alebo ich máme skôr považovať za samostatné objekty. Z pohľadu zobrazovacej funkcie editora je lepšie, ak hrany sú definované nezávisle od objektov, ktoré spájajú.

Hrany nie sú graficky samostatný prvok, nakoľko ich vykreslenie záleží hneď na niekoľkých atribútoch:

- na synchrónnom porte alebo prechode s ktorým je spojený
- na mieste s ktorým je prepojený
- na bodoch cez ktoré sa tiahne čiara hrany
- na type hrany

Ako ďalší problém, ktorému musíme čeliť ak použijeme hranu ako samostatný objekt je, že nemáme jednoznačnú identifikáciu hrany. Hrana nemá meno v rámci systému ani metódy, lebo podľa definície PNTalku patrí k OOPN objektu synchrónneho portu alebo prechodu.

4.2.5.1 Miesto

Ako sme už znázornili vyššie, tak miesto, s ktorým je prepojená hrana odohráva jednu z kľúčových úloh. To platí pravdaže pri grafickej definícii objektu, nakoľko koniec hrany by sa mal presne dotýkať okrajov elipsy. Z toho vyplýva, že ak sa dynamicky mení elipsa, to znamená presun alebo zmenu jej veľkosti, musí sa meniť aj koncový bod hrany.

Keďže sme zahrnuli do definície miesto, nie je potrebné implicitne prikladať štartovací bod hrany, nakoľko sa dynamicky môže dopočítať.

4.2.5.2 Prechod a synchrónny port

Ak berieme do úvahy definíciu hrany, tak je jednoznačné, že podľa hierarchie v OOPN patrí k prechodu alebo k synchrónnemu portu. Preto musíme uchovať odkaz na tieto objekty. Avšak prechod a synchrónny port sa navzájom vylúčia, takže buď použijeme generalizáciu synchrónneho portu alebo použijeme jednu z dvoch a tú druhú nastavíme na neinicializované. Využijeme ale oba prístupy, kvôli ľahšej manipulácii.

Ani tu nemusíme implicitne ukladať koncový bod hrany. Pri dynamických zmenách prechodu alebo synchrónneho portu sa údaj dá jednoducho dopočítať.

4.2.5.3 Lomená čiara

Pri kreslení hrany musíme dbať na to, že hrana nie je len jedna priama čiara spojujúca dve objekty, ale môže mať niekoľko častí, čiar. Ako sme už vyššie uviedli, k týmto bodom nepatria body na hranice miesta objektu alebo synchrónneho portu, ale tie body, ktoré sú len na grafickom kontexte, a v ktorých bodoch sa hrana nalomí. Definovaním týchto bodov tak dokážeme hranu rozdeliť na sekvenciu čiar. Musíme však dbať na poradie bodov, takže ako najpohodlnejšie je zvoliť ako dátovú štruktúru zoznam.

4.2.5.4 Hranový výraz

Hranový výraz je vymedzený v definícii PNTalku. Našou úlohou je teraz dosadiť túto definíciu do praktickej formy. Keďže hranový výraz je skoro identický s počiatočným zaučením miesta, preto ako dátový typ znovu zvolíme reťazec kvôli univerzalite. Aj hranový výraz môže byť bez textu, v tom prípade použijeme prázdny reťazec.

4.2.5.5 Typ

Tento údaj je dôležitý pri hrane, a to z niekoľkých dôvodov. Ako prvý musíme rozoznať hranu a k tomu nám poslúži typ. Typy hrany rozdelíme na tri, a to podľa pôvodnej definície PNTalku na:

- vstupnú hranu
- výstupnú hranu
- testovaciu hranu

Táto informácia potom napomáha grafickému znázorneniu, keď sa na príslušný koniec hrany dokreslí šípka a to podľa definície.

4.2.5.6 Grafické informácie

Pri hrane už nepotrebujeme žiadne ďalšie grafické informácie, nakoľko dva koncové body sa dajú vypočítať z objektov, ktorými sú spojené a body sa majú uložiť zvlášť do dátovej štruktúry.

Výpočet koncových bodov hrán by sme mali realizovať s dvoma funkciami:

- Prvá funkcia pre dotyk s elipsou, teda miestom by mal byť výpočet priesečníka úsečky a elipsy. Úsečka by mala byť čiara z prostriedku elipsy smerom k nasledujúcemu bodu, pričom ak nasledujúci bod by mal byť objekt prechod alebo synchrónny port, tak sa použije prostredný bod toho objektu.
- Druhá funkcia vypočítava dotyk synchrónneho portu alebo prechodu s úsečkou. Pri návrhu tejto funkcie nie je potrebné použiť komplikovanú matematickú funkciu. Stačí definovať osem bodov a na základe rozloženia nasledujúceho bodu čiary sa vypočíta uhol čiary. Na základe uhla sa vypočíta, že na ktorý z ôsmych bodov bude koncový bod pripojený. Pritom definovaných osem bodov je na obdĺžniku, a to nasledovne: pravý horný, ľavý horný, ľavý dolný, pravý dolný roh a prostriedky hornej, dolnej, ľavej a pravej steny.

Ďalšou otázkou zostáva zobrazenie hranového výrazu. Keďže sa jedná o text spojený s hranou, môžeme brať do úvahy niekoľko možností. Ako prvé treba definovať, kde presne sa má zobrazený text objaviť, aby bol chápaný ako súčasť hrany. Asi najlogickejšie je ak sa priamo vykreslí tak, aby sa buď začiatkom alebo koncom dotýkal hrany. Avšak samotná hrana môže pozostávať z niekoľkých čiar, rozdelených bodmi. V tomto prípade berieme riešenie, že sa bude brať do úvahy prostredná časť hrany. A aj pri tejto prostrednej časti prostredný bod. Ešte podľa sklonu hrany musíme definovať, či

sa má použiť text napravo alebo naľavo vykreslený a tak dostávame priamu pozíciu vykreslenia textu na kresliacej ploche.

4.2.6 Elementy OOPN

4.2.6.1 Typy

Pre jednoduchšiu manipuláciu s objektmi môžeme v rámci celej aplikácie definovať vlastný dátový typ, ktorý jednoznačne dokáže z OOPN objektu vyviesť o aký objekt sa jedná. Toto rozlíšenie typov je dôležité pri objektoch, ktoré majú tú istú Java triedu. Ako príklad môžeme uviesť napríklad hrany, ktoré sú v pôvodnej definícii odlišné len v tom, na ktorom konci hrany sa nachádza šípka. Taktiež toto rozlíšenie typov môže byť vhodné, keď rozlišujeme metódy a konštruktory. Oba objekty majú podobné zobrazenie, akurát vo funkcionalite sa rozlišujú. Pre uchovanie kompatibility však definujeme aj tento typ pre miesta, prechody, synchrónne porty, kvôli možným neskorším rozšíreniam.

Pre definíciu typov použijeme triedu s vyčísleným typom. Do nej zadefinujeme všetky možné typové značenia objektov. Takto sa definícia typu stáva jednoznačná v rámci celej aplikácie.

4.2.6.2 XML identifikátor

Ako v vidíme z predchádzajúcich OOPN Objektov, tak len ťažko sa definuje taká jednoznačná identifikácia, ktorá by jednotným spôsobom dokázala rozlíšiť jeden objekt od druhého objektu. Napríklad prechody a miesta používajú k jednoznačnej identifikácii meno s kombináciou referencie na sieť, sieť a synchrónny port používa k výhradnej identifikácii meno, kým hrany nemajú implicitný identifikátor. Z tohto dôvodu potrebujeme zaviesť k jednotlivým objektom jednoznačné identifikátory, s ktorými sa na ne neskôr môžeme odkazovať a vyhľadávať ich. Vieme, že v rámci Java aplikácie majú jednotlivé Java objekty jednoznačnú referenciu. Táto referencia sa môže previesť z číselnej formy do textovej podoby ak použijeme toString funkciu. Formát má nasledujúci:

<meno_balíčka>.<meno_triedy>@<číslo_inštancie_triedy>

Takto definovaný identifikátor je jednoznačný v rámci celého systému a je teda použiteľný napríklad ak danú OOPN triedu budeme ukladať do XML formátu. Je tak vylúčené, že sa referencie môžu stratiť.

4.2.6.3 Zdedenie

Ak ďalej rozoberieme definíciu triedy OOPN, musíme počítať ešte s niekoľkými atribútmi, ktorými by mal OOPN objekt disponovať. Definícia OOPN zavádza aj pojem dedičnosť tried. To znamená, že aktuálny objekt nemusí byť definovaný priamo v tejto triede, ale môže byť definovaný v nadradenej triede. Tento atribút musíme definovať pre všetky objekty. Isté vymedzenie by tu mohlo byť, lebo napríklad prechody a miesta sa dedia samostatne, len keď sú súčasťou objektovej siete. Inak sa môžu dediť len celkovo s metódou alebo konštruktorom. Avšak pre vykreslenie je jednoduchšie ak pre daný element presne vieme určiť, či sa jedná o zdedený alebo o priamo definovaný OOPN objekt. Rozlišujeme pri tom ešte jednu skutočnosť, a to, že dedený objekt môže byť predefinovaný. V tomto prípade musíme poznačiť element, že je síce zdedený, ale je aj predefinovaný a tak musí byť zahrnutý do definície aktuálnej triedy.

Najjednoduchšie je definovať triedu, ktorá presne vymedzuje ako môže vyzerať zdedenie. Rozlišujeme 3 druhy:

- Zdedená – kde objekt je súčasťou nadradenej triedy

- Natívna – kde objekt nie je súčasťou ani jednej nadradenej triedy, a je definovaná len v tejto triede.
- Preťažená – ktorá znamená, že objekt je síce definovaný v nadradenej triede, ale je predefinovaný v aktuálnej triede.

4.3 Interné zobrazenie triedy

V predchádzajúcich častiach sme definovali jednotlivé OOPN objekty s atribútmi, parametrami a grafickými informáciami. Ďalej potrebujeme navrhnúť spôsob ako budú tieto samostatné objekty interaktívne spolupracovať, aby výsledok bol zobrazenie triedy ako celku. Samozrejme jednotlivé OOPN objekty sú hierarchicky poskladané a niektoré majú aj referencie na ostatné objekty, napríklad, miesta a prechody majú referencie na siete, alebo hrany majú referencie na synchrónne porty, prechody, alebo miesta. Tieto údaje však nie sú dostatočné aby sme tieto objekty mali pokope. K tomu potrebujeme vytvoriť jeden kontajner, ktorý nám poslúži ako skladište týchto OOPN objektov.

V predchádzajúcich bodoch sme rozdelili OOPN objekty do piatich definičných skupín s vlastnou grafickou reprezentáciou. Pravdaže sme sa zvlášť nezaoberali primitívnymi typmi, pre ktoré sme vo väčšine prípadoch použili reťazový dátový typ. Tieto primitívne typy sú zahrnuté ako súčasť vyšších OOPN objektov a netvoria samostatnú zobrazovaciu jednotku.

4.3.1 Úložné jednotky

Definícia pätice objektov je teda postačujúca aby sme dokázali vykresliť ľubovoľnú OOPN sieť. Musíme však tieto objekty zorganizovať do štruktúrovanej formy. Môžeme sa zamyslieť nad dvoma formami uloženia objektov. Ako prvý prístup by mohol byť, že všetko uložíme do jednej spoločnej úložnej jednotky, alebo vytvoríme päť rôznych úložných jednotiek, kde separovane uložíme jednotlivé Java inštancie OOPN objektov. Druhý prístup sa javí ako logickejší a preto pristúpime k nemu.

Ako dátový typ skladišťa, musíme navrhnúť taký typ, nad ktorým sme schopný vkladať, vyberať, a rýchlo vyhľadávať prvky. Ako najjednoduchšie sa zdá použiť pole elementov. Avšak pole elementov nemusí vždy stačiť, hlavne keď sa mažú prvky, vtedy sa pole musí preindexovať. Je oveľa pohodlnejšie použiť zoznam. V Jave Existuje typ ArrayList čo je vlastne pole, ktoré v sebe má implementované funkčnosť zoznamu. Tento typ sa zdá byť pre naše účely vhodný. Môžeme ho použiť pri všetkých piatich Java tried OOPN objektov.

4.3.2 Atribúty OOPN triedy

Z pôvodnej PNTalk definície triedy sú dôležité základné údaje o názvoch tried. Tieto údaje a zahrnuté objekty do triedy sú postačujúce pre simuláciu a základné vykreslenie modelu, ale nezabezpečujú nám škálovateľnosť zobrazenia, ktoré sa od editora očakáva. Musíme tak doplniť existujúce údaje o nové atribúty, ktoré sa budú viazať k triede.

4.3.2.1 Meno aktuálnej triedy

Toto meno je reprezentované ako reťazec. Vieme podľa definície, že mená tried začínajú veľkým počiatočným písmenom, takže aj v tomto prípade si na to potrpíme. Meno triedy je taktiež nezávisle prezentované v objektovej sieti.

4.3.2.2 Meno nadradenej triedy

Meno nadradenej triedy je znova reťazec, a aj tu sa dbá na to, aby sa meno začínalo s veľkým začiatočným písmenom.

4.3.2.3 Minimálna veľkosť objektov

Ako prvá grafická informácia k triede je minimálna veľkosť objektov. Keďže vieme že miesta, prechody a synchrónne porty majú tvary podobnej veľkosti, môžeme tak zaviesť implicitné značenie veľkosti týchto objektov. Tento údaj je celočíselná hodnota vyjadrujúca veľkosť v pixloch. Je použitá ako pre výšku tak aj pre šírku objektu. Objekty môžu byť väčšie, ak sa do nich text nezmestí, inak nie menšie.

4.3.2.4 Veľkosť použitého fontu

Výpočet veľkosti objektov záleží aj na veľkosti použitého fontu. Súčasný návrh nezahrňuje možnosť použitia ľubovoľného fontu, avšak toto rozšírenie je ľahko implementovateľné v budúcnosti. Avšak do návrhu priradíme veľkosť implicitne použitého fontu. Táto veľkosť sa udáva v bodoch, použitím celočíselnej hodnoty.

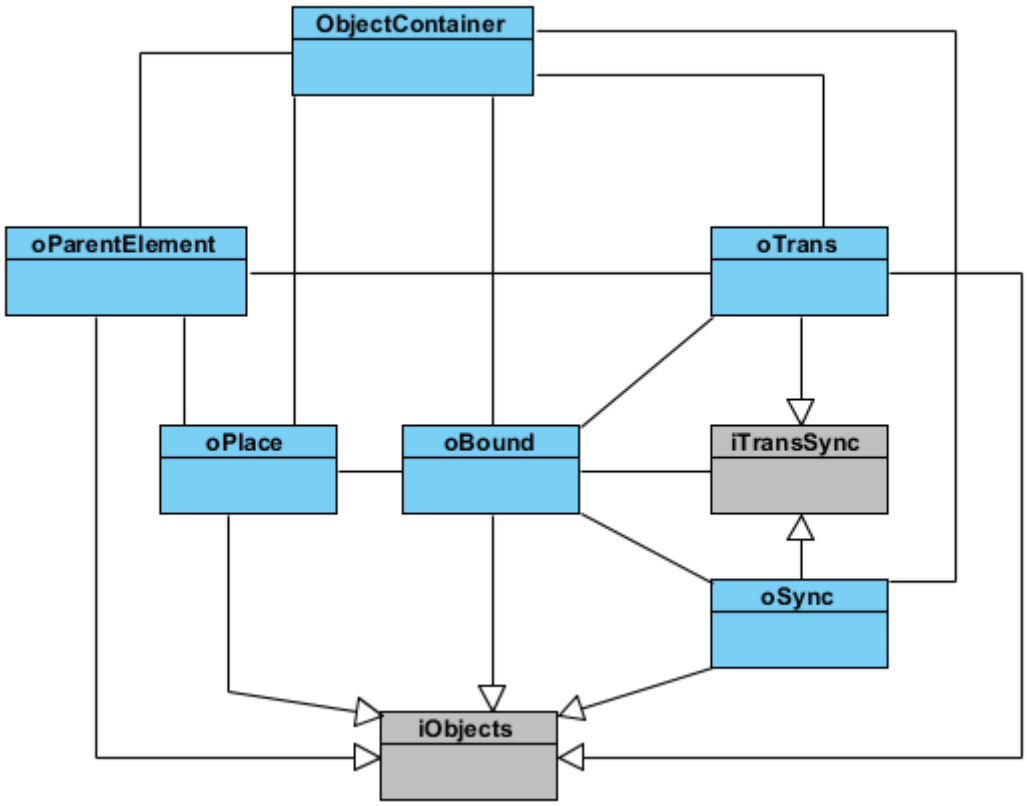
4.3.2.5 Veľkosti celkového modelu

Jedná sa o dve číselné údaje, ktoré udávajú veľkosť kresliacej plochy, presnejšie výšku a šírku. Sú tieto údaje užitočné, ak musíme daný model rozložiť na dopredu definované miesto, alebo ak chceme dostávať výslednú grafiku dopredu definovanej miery.

4.3.2.6 Informácie o použitých farbách

Návrh sa zaoberá aj použitými farbami. Základný návrh PNTalku aj obecne Petriho siete okrem farebného typu nedefinujú farebné značenia. Väčšinou sa používajú stupnice sivej pri znázornení PNTalk triedy, ale naša definícia dokáže pridať farebnosť do celkovej editácie. Tieto farby sú jednotlivo uchované, a každý objekt môže mať niekoľko farebných parametrov. Tieto parametre budú definované v neskorších častiach, avšak zoznam vykreslených možností farbami si uvedieme v nasledujúcich bodoch:

- Farba vykreslenia objektov,
- Farba siete metódy
- Farba siete konštruktora
- Farba vybraného objektu
- Farba objektu v pohybe
- Farba vykreslenia zdedených objektov
- Farba vykreslenia preťažených objektov
- Farba výplne zdedeného synchrónneho portu
- Farba výplne synchrónneho portu
- Farba výplne zdedeného objektu, ako miesto alebo prechod
- Farba výplne objektu, ako miesto alebo prechod
- Farba pozadia kresliacej plochy



Obrázok 5.3: UML graf spolupráce tried v internom zobrazení.

4.4 XML formát

Máme definované interné zobrazenie formátu, avšak interné zobrazenie musíme uložiť, aby sa manipulácia s aplikáciou neobmedzovala na jednorazovú činnosť, ale rozrobenú OOPN sieť bolo možné editovať aj inokedy. K tomu musíme vytvoriť formát súboru, podľa ktorého dokážeme uložiť a znova načítať všetky interné práce.

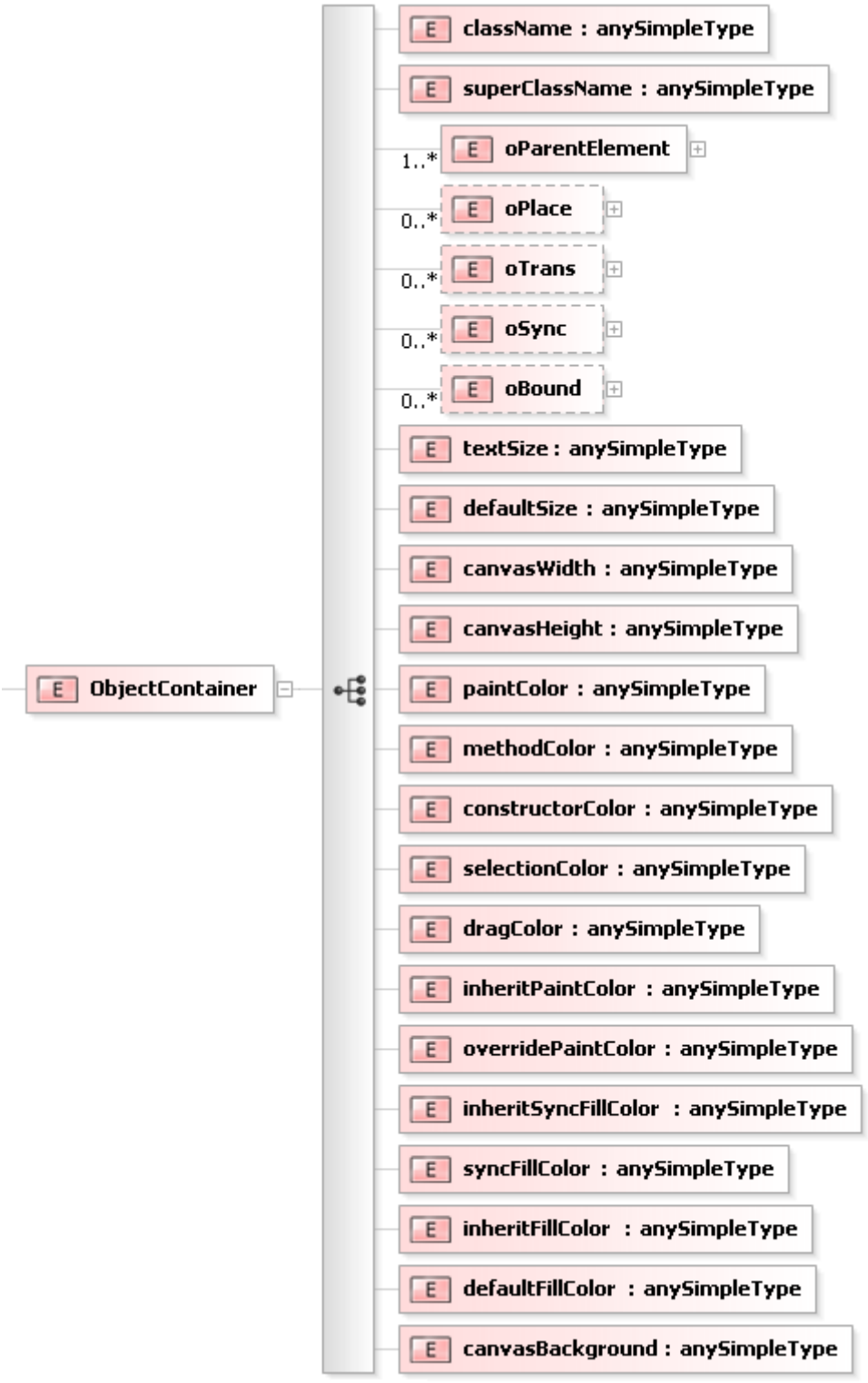
4.4.1 Trieda

Ako základ celej aplikácie je uloženie a manipulácia s jednou OOPN triedou. Samozrejme vieme, že celkový OOPN systém pozostáva z niekoľkých tried, avšak jednotka pre zobrazenie vytvárania sa považuje ako jedna trieda. Keďže aplikácia teda pracuje v jednom čase s jednou triedou, tým pádom považujeme triedu ako základný prvok, teda ako hlavný element v XML triede. Môžeme navrhnúť XML formát pre uloženie niekoľkých tried, avšak teraz pre jednoduchosť predpokladajme, že jeden XML súbor bude ukladať presne jednu triedu.

Keďže formát XML pozostáva z jedného koreňového elementu, ktorý sa v anglickej terminológii značí root, musíme definovať tento element. Všetky dokumenty, ktoré slúžia ako úložné dátové jednotky majú definovaný koreňový element tag **<xml>**. Tento tag je použitý ako koreňový aj v našom prípade.

Z predchádzajúcich bodoch vieme, že obdobou triedy je akoby úložná jednotka s pridanými atribútmi. Túto úložnú jednotku zadefinujeme ako **ObjectContainer**, a priradíme mu tag s identickým názvom. Keďže sme si uložili restrikciu, že jedna trieda je súčasťou jedného XML

súboru, musíme teda povinne definovať, že v našom XML súbore bude práve jeden tag ObjectContainer.



Obrázok 5.4: XML elementy OOPN triedy.

V budúcnosti môžeme pouvažovať nad tým, že túto restrikciu prepíšeme, a tak celý XML formát by bol schopný uložiť celý systém PNTalk.

K triede však patria aj ďalšie atribúty, ktoré sa musia pripojiť k nej. Ako najdôležitejšie atribúty sú meno triedy a meno nadradenej triedy. Tieto informácie musíme pripojiť s ďalšími tagmi,

pričom platí znova pravidlo, že sa tieto atribúty môžu vyskytnúť práve jedenkrát. Tentoraz však sú tieto atribúty zapuzdrené ako jednotlivé uzly pod uzlom ObjectContainer. K definovaní tagov použijeme anglické názvy **className** a **superClassName**.

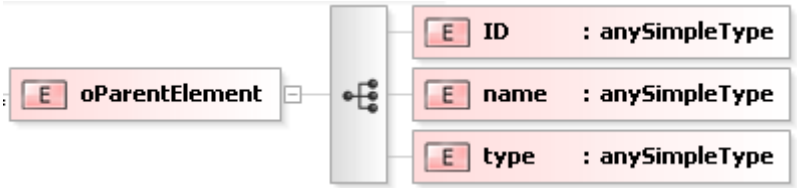
Ďalšie atribúty nesúce grafické informácie musíme tiež uložiť. Avšak tieto atribúty definujeme tak, že sa nemusia implicitne prikladať k XML súboru. To znamená, že ak tieto atribúty nie sú zahrnuté, tak sa použije defaultné nastavenie. Jedná sa o veľkosť textu, minimálnu veľkosť objektu a informácie o farbách. Tagy, ktoré sa použijú sú nasledujúce:

- **textSize**
- **defaultSize**
- **canvasWidth**
- **canvasHeight**
- **paintColor**
- **methodColor**
- **constructorColor**
- **selectionColor**
- **dragColor**
- **inheritPaintColor**
- **overridePaintColor**
- **inheritSyncFillColor**
- **syncFillColor**
- **inheritFillColor**
- **defaultFillColor**
- **canvasBackground**

Máme tak definovanú triedu x XML tagmi. Samotná trieda však nenesie žiadne relevantné informácie o jednotlivých objektoch. Tieto informácie sú zahrnuté v ostatných elementoch. Musíme teda špecifikovať tieto OOPN objekty.

4.4.2 Sieť

Ako prvé objekty ktorých definície musíme uložiť, sú OOPN objekty sietí. Samotná sieť nemá žiadnu zobrazovaciu silu. Sémanticky sa jedná o dôležitú časť v OOPN, avšak v grafickom znázornení slúži skôr pre zoskupenie objektov, konkrétne miest a prechodov. Preto použijeme namiesto výrazu sieť tag **oParentElement**. S týmto znázorníme, že tieto objekty sú vlastne logické celky, pod ktoré patrí mnoho iných OOPN objektov.



Obrázok 5.5: XML elementy OOPN siete.

Keďže siete majú taktiež ďalšie atribúty, tieto musíme uchovať do ďalších uzlov pod elementom oParentElement. Asi najdôležitejší parameter musí byť jednoznačný identifikátor. Tento identifikátor uložíme do taku **ID**. Ostatné atribúty sú meno siete, definovaný anglickým názvom **name**. Ako ďalší parameter musíme uložiť typ siete. K použitému tagu priradíme názov **type**. Môže nadobúdať tri hodnoty, a to :

- *OBJECT* - pre určenie objektovej siete
- *METHOD* – pre určenie siete metódy

- *CONSTRUCTOR* – pre určenie siete konštruktora

Každý objekt siete popritom môže byť zdedený. To, či sa jedná o zdedenú sieť naznačuje atribút. Tento atribút sa uchová pomedzi tagy **inheritance**, a môže nadobúdať hodnoty:

- *INHERITED* – ak je sieť zdedená
- *OVERRIDED* – ak je sieť preťažená
- *NATIV* – ak je sieť definovaná len v tejto triede

V rámci triedy sa môže vyskytnúť až niekoľko elementov siete, s odlišnými parametrami, ako názov metódy alebo konštruktora, avšak objektová sieť musí byt vždy práve jedna.

4.4.3 Miesto

Ako ďalšie dôležité objekty musíme uložiť informácie o miestach. Máme tu niekoľko možností ako ukladať objekty miest. Jednak, môžeme tieto objekty pridať ako uzly pod oParentElement, a môžeme ich uložiť aj samostatne pripojené k ObjectContainer uzlu. Z dôvodu, že miesta sa uchovávajú v internom zobrazení v jednej veľkej úložnej jednotke, bolo logické použiť prístup, že sa aj v XML formáte budú naväzovať na ObjectContainer, avšak budú mať uloženú referenciu na sieť.

Ako tag pre miesto použijeme kľúčové slovo **oPlace**. Môže sa vyskytnúť nula až n-krát, v rámci XML súboru, kde n značí presný počet miest v rámci celej triedy.

Miesto má taktiež atribúty, ktoré musíme uložiť. Ako jeden z najdôležitejších je znova jednoznačný identifikátor značený tagom **ID**. Ako sme spomenuli musíme uložiť aj identifikátor na sieť, ku ktorému miesto patrí. Tento údaj bude tag **oParent**. Aj miesto má definované meno, pre ktoré použijeme kľúčový výraz **name**. Ako ďalší atribút je typ značený tagom **type.** Tento atribút striktne nadobúda hodnotu PLACE. Ďalší atribút určí dedenie objektu a je uložené medzi tagy **inheritance**, a môže nadobúdať hodnoty:

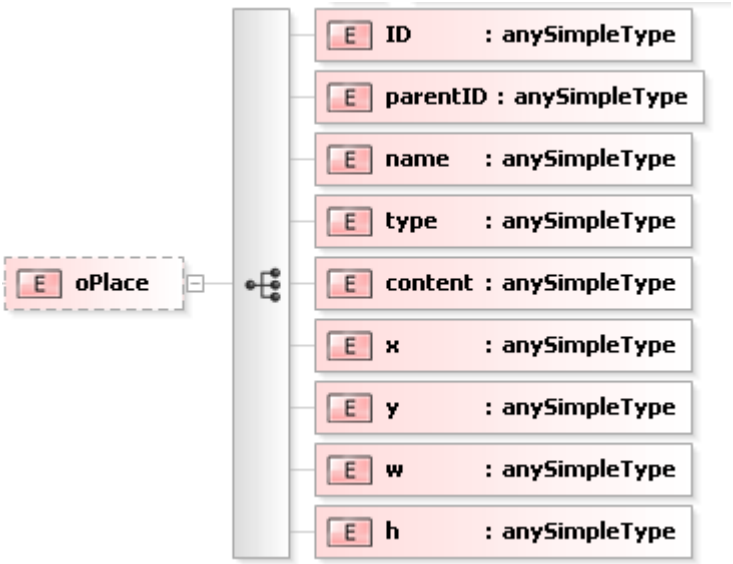
- *INHERITED* – ak je miesto zdedené
- *OVERRIDED* – ak je miesto preťažené
- *NATIV* – ak je miesto definované len v tejto triede.

Miesta môžu mať ďalej počiatočné značenie. Toto počiatočné značenie sa graficky zobrazuje dovnútra miesta a je tak jeho obsahom. Preto použijeme pre značenie z anglickej terminológie „reťazec obsahu“ teda **content**. Tento tag sa musí povinne vyskytovať, avšak môže byť prázdny, nakoľko nemusí byť definované žiadne počiatočné značenie.

Ďalšie informácie, ktoré sa uchovávajú, sú grafické informácie. Ako sme už vysvetlili pri internom zobrazení, že miesto je samostatný grafický celok, definovaný štyrmi údajmi. Tieto údaje definujeme tagmi:

- **x**, súradnica ľavej strany
- **y,** súradnica hornej strany
- **w,** šírka
- **h,** výška

Definícia miesta sa môže objaviť nula až n-krát. Presný počet je daný počtom miest v danej triede a to aj zdedenými miestami.



Obrázok 5.6: XML elementy OOPN miesta.

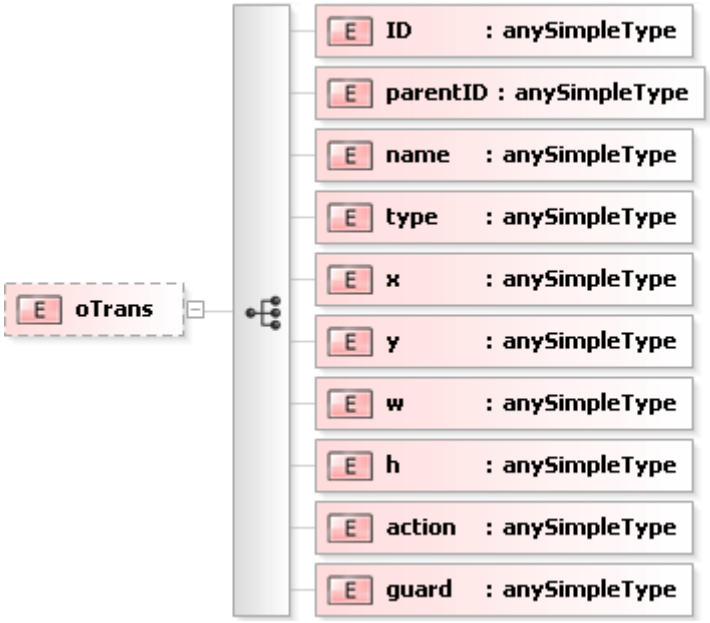
4.4.4 Prechod

Prechod, podobne ako miesto je samostatná jednotka vykreslenia. Prechod je tiež súčasťou siete, avšak kvôli internému zobrazeniu ho napájame na uzol ObjectContainer. Prechod bude uložený do tagu definovaného názvom **oTrans**. Jednotlivé atribúty musíme uchovať ako ďalšie uzly napojené na oTrans. Tieto atribúty sú jednoznačný identifikátor definovaný tagom **ID**. Nakoľko aj prechod je súčasťou vždy niektorej triedy, preto musíme uložiť identifikátor tej triedy do tagu **parentID**. Aj prechod má meno, pri ktorej použijeme definíciu tagu **name**. Ďalej aj prechod sa musí identifikovať typom, uzavretým do tagu **type**. Hodnota je vždy TRANS. Ďalší atribút je dedenie objektu, ktoré ako pri mieste aj sieti je uložené medzi tagy **inheritance**, a môže nadobúdať hodnoty:

- *INHERITED* – ak je prechod zdedený
- *OVERRIDED* – ak je prechod preťažený
- *NATIV* – ak je prechod definovaný len v tejto triede .

Prechod má stráž a akciu. Stráž sa ukladá do tagu **guard** a akcia do tagu **action.** Tieto tagy sa musia povinne vyskytovať ako súčasť oTrans, avšak nemusia nadobúdať hodnotu. Ako posledné musíme uložiť grafické informácie a to podobne ako pri mieste aj tu si vystačíme štyrmi údajmi a to:

- **x**, súradnica ľavej strany
- **y,** súradnica hornej strany
- **w,** šírka
- **h,** výška



Obrázok 5.7: XML elementy OOPN prechodu.

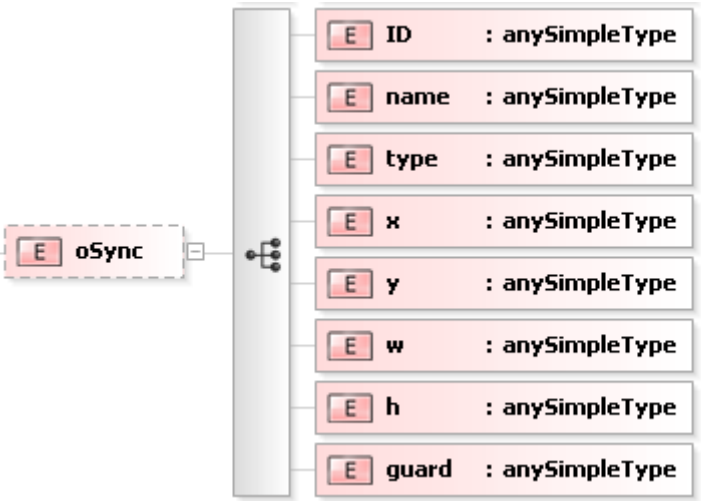
4.4.5 Synchrónny port

Keďže synchrónne porty sa budú definovať v XML dokumente tagmi **oSync**. Tieto uzly musíme jednoznačne napojiť na ObjectContainer. Atribúty, ktoré majú byť definované sú podobné ako pri prechode. Aj synchrónny port musí zahrňovať jednoznačný identifikátor, a to v tagu **ID**. Taktiež má jednoznačné meno, definovaný pomocou tagu **name**. Ďalej aj synchrónny port musíme poznačiť typom. Ukladáme ho medzi tagy **type** a priraďujeme hodnotu SYNC. Ako ostatné objekty, tak aj synchrónny port má uzol **inheritance** definujúcu dedičnosť nasledujúco:

- *INHERITED* – ak je synchrónny port zdedený
- *OVERRIDED* – ak je synchrónny port preťažený
- *NATIV* – ak je synchrónny port definovaný len v tejto triede.

Z definície synchrónneho portu vieme, že jej súčasťou môže byť stráž, pre ktorú použijeme tag **guard.** Podobne ako pri prechode, aj tu sa stráž musí definovať ale nemusí nadobúdať žiadnu hodnotu. Ako posledné sa aj tu musia použiť grafické informácie:

- **x**, súradnica ľavej strany
- **y,** súradnica hornej strany
- **w,** šírka
- **h,** výška



Obrázok 5.8: XML elementy OOPN synchrónneho portu.

4.4.6 Hrana

Ako sme už vysvetlili, hrana sa definuje ako samostatný objekt, a preto aj formát XML ho vymedzuje tak. Napája sa podobne ako ostatné objekty na uzol definovaný ObjectContainer tagom. Samotná hrana sa uzatvára do tagu **oBound**. Ostatné atribúty sa pridávajú do uzlov viazajúce sa na oBound. Ako prvý musíme definovat identifikátor, ktorý ukladáme pomedzi tagy **ID**. Ako ďalšie sa musia uložiť väzby na miesta a prechody alebo na synchrónne porty. Väzby na miesta uložíme do tagov **parentPlaceID**. Keďže vieme, že hrana sa môže napájať buď na prechod, alebo aj na synchrónny port, takže by sme mali rozoznávať o čo sa jedná. Keďže však používame jednoznačné identifikátory, ktoré rozlíšia objekt aj typovo, nemusíme sa zaoberať týmto krokom, a môžeme tak previesť priamo identifikátor prechodu alebo synchrónneho portu. Pri spätnej rekonštrukcie sa tak len vyhľadá príslušný objekt a nemusíme sa zaoberať jeho typom. Tento identifikátor uložíme do tagu **parentTransSyncID**.

Podobne ako pri ostatných objektoch, tak aj pri hrane je dôležité uchovať typ hrany. K tomu použijeme tag **type.** Hodnoty tohto tagu tak môžu nadobúdať hodnoty:

- COND – pre testovaciu hranu
- PRECOND – pre vstupnú hranu
- POSTCOND – pre výstupnú hranu

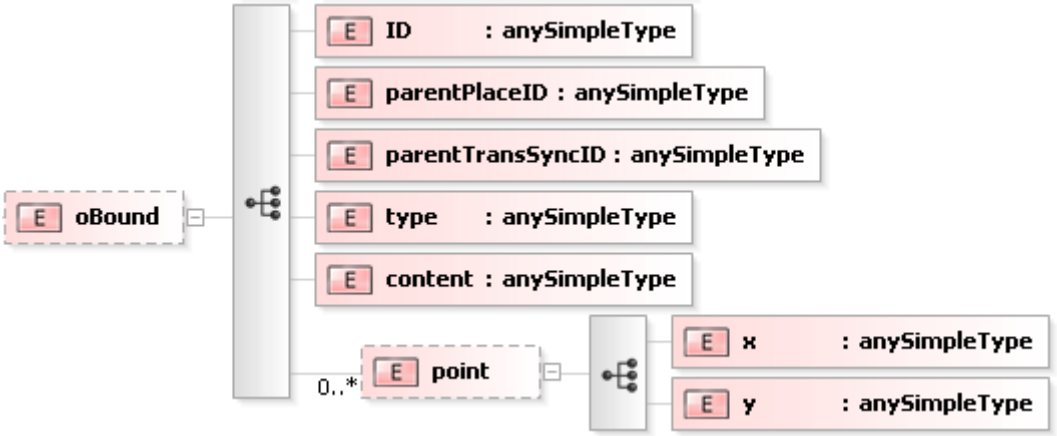
Taktiež ako všetky ostatné objekty, tak aj hrany majú určenú dedičnosť. Táto dedičnosť je v prípade hrany totožná s dedičnosťou nadradeného objektu, teda, prechodu alebo synchrónneho portu. Pre prehľadnejšie spracovanie sa však uvádza aj tu a to pomedzi tagy **inheritance,** ktorá môže nadobúdať nasledujúce hodnoty:

- *INHERITED* – ak je synchrónny port zdedený
- *OVERRIDED* – ak je synchrónny port preťažený
- *NATIV* – ak je synchrónny port definovaný len v tejto triede .

Každá hrana má popri týchto informácií aj definovaný hranový výraz. Tento hranový výraz používa identickú syntaktiku, ako počiatočné značenie pri mieste. Je preto logické použiť znovu tag **content,** pre značenie hranového výrazu.

Pri hrane sa musí uložiť ešte jedna dôležitá informácia, a tou je zoznam bodov cez ktoré prechádza. Tento zoznam pritom môže byť prázdny. S týmto sa vysporiadame tak, že každý bod budeme definovať zvlášť pomedzi tagy **point.** Každý uzol point naväzuje na uzol oBound, pričom point sa môže vyskytnúť nula až n-krát. Keď sa teda ukladá informácia o hrane, ktorá priamočiarovo

spojuje miesto s prechodom, tak sa tag point nevyskytuje. Vnútorné zobrazenie tagu point, bude určené nasledujúcu. Ak berieme dvojdimenzionálny súradnicový priestor, tak každý bod je definovaný súradnicou x a súradnicou y. Aj v našom prípade preto každý uzol point bude mať element s tagom **x** a element s tagom **y**.



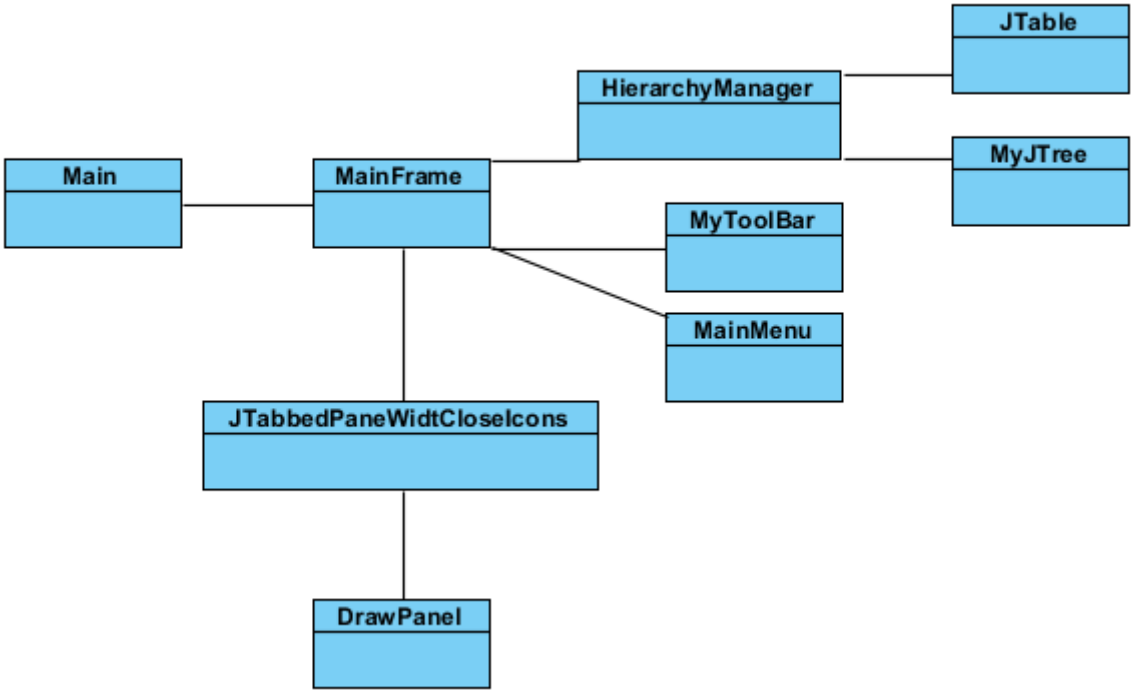
Obrázok 5.9: XML elementy OOPN hrany.

5 Návrh GUI

Pri návrhu GUI aplikácie si musíme uvedomiť, že sa jedná vlastne o kľúčovú časť práce. Všetky editory sú silne založené na funkcionalite užívateľského rozhrania. V prechádzajúcich kapitolách sme rozobrali dva existujúce nástroje, ktoré nám ponúkajú dobrý základ k tomu, aby sme docielili užitočné jednoduché a prehľadné grafické užívateľské rozhranie.

Grafické užívateľské rozhranie v nasledujúcich bodoch rozoberieme na niekoľko častí. Každá časť je samostatná logická jednotka, jej návrh budeme realizovať tak, aby sa dostala do samostatnej triedy.

Samotná implementácia prebehne v programovacom jazyku java. Tento jazyk má niekoľko knižníc, pomocou ktorých sa môžu vytvárať GUI komponenty. Najznámejšie a najpoužívanejšie knižnice sú AWT a SWING. Nedoporučuje sa kombinovať tieto knižnice v rámci vytváraní GUI, takže si vyberieme jednoznačne novšiu knižnicu SWING.



Obrázok 6.1: UML diagram spolupráce Java tried, popisujúcich vizuálne komponenty aplikácie.

5.1 Hlavné okno

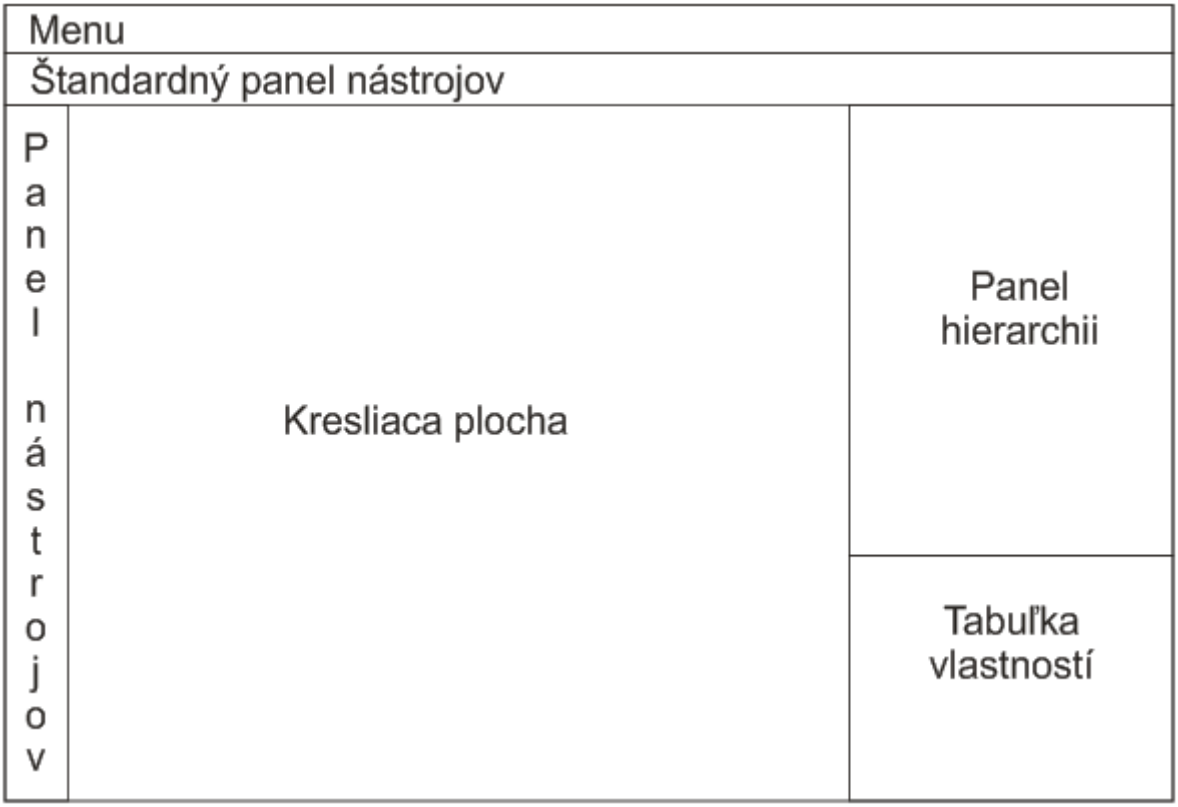
Existuje mnoho revolučných prístupov ako vytvoriť okno aplikácie. Ako jeden z extrémov je napríklad GIMP, ktorý má všetky nástroje, panely vlastností v samostatnom okne. Tento prístup však pri návrhu našej aplikácie nemusí obstáť.

Kvôli tomu zvolíme radšej klasický vzhľad všetko v jednom okne, a v rámci okna budú rozdelené jednotlivé panely. Toto okno nazveme preto hlavným oknom, a tvorí jadro celej aplikácie.

Musí v sebe uložiť všetky referencie na panely, pomocou ktorých užívateľ interaktívne ovláda aplikáciu.

Hlavné okno v návrhu rozdelíme na 6 častí. Tieto časti sa v nasledujúcich bodoch budú rozoberať ďalej. Vizuálne uloženie však načrtneme teraz.

Na hornú časť okna vložíme klasicky menu aplikácie. Pod menu uložíme štandardný panel nástrojov. Na ľavej strane sa bude nachádzať panel nástrojov. Uprostred aplikácie bude telo, teda samotná kresliaca plocha. Pravá strana aplikácie bude rozčlenená na hornú a dolnú časť. V hornej časti sa bude nachádzať panel hierarchií a dolnú časť bude tvoriť tabuľka vlastností. Náčrt návrhu okna uvidíme na nasledujúcom obrázku.



Obrázok 6.2: Náčrt GUI aplikácie.

Pre implementačnú časť zvolíme komponent z knižnice SWING JFrame. Tento komponent je dobre škálovateľný a preto bude slúžiť ako základ našej aplikácie.

5.2 Kresliaca plocha

Kresliaca plocha je jadrom celej aplikácie. Je to tá časť, na ktorú zobrazíme celú triedu v rámci PNTalk systému. Keďže zobrazenie a manipulácia s celým PNTalk systémom by bola príliš neprehľadná, tak ako najkomplexnejšia jednotka pre zobrazenie použijeme triedu. Kresliacu plochu pritom upravíme tak, aby naraz bolo možné súbežne pracovať s niekoľkými triedami naraz. To dáva užívateľovi možnosť nepriamo pracovať s celkovým PNTalk systémom.

Aby sme mohli docieliť realizáciu kresliacej plochy, musíme vybrať vhodný komponent. Ako ideálné sa zdá použiť JPanel. Avšak samotný JPanel je veľmi základný komponent. V moderných aplikáciách je už samozrejmosťou, že plocha s ktorou pracujeme môže byť posunutá. K posunutí sa

používa rolovací pruh, ktorý je implementovaný v Jave ako JScrollBar. Do JScrollBar zapuzdríme JPanel a máme už celkom slušne manipulovateľnú plochu.

V ďalšom kroku sa musíme zamyslieť nad tým, že ako realizujeme súbežnú manipuláciu s viacerými triedami naraz. K tomu môžeme využiť ďalší komponent knižnice SWING a to JTabbedPane. Jedná sa o panel so záložkami. Jednotlivé záložky popritom môžu byť dynamické a tak dokážu celkovo pokryť celý systém tried v PNTalku.

Samotný základ kresliacej plochy pre jednu OOPN triedu tak vždy bude tvoriť jeden JPanel. Tento panel sa bude považovať ako obrázok, ktorý je realizovaný na základe vnútorného zobrazenia OOPN Triedy. Musí tak existovať metóda, ktorá dokáže previesť to externé zobrazenie do vizuálnej podoby. Keďže panel má rastrový základ, v určitom slova zmysle sa jedná o rasterizáciu objektov. Návrh jednotlivých algoritmov a funkcií bude rozobraný v neskorších častiach.

K tomuto JPanelu môžeme pripojiť reakcie na rôzne interakcie pôsobené užívateľom. Tie interakcie budú hlavne reakcie na udalosti myšou. Samozrejme tieto reakcie budú záležať aj od vonkajších okolností, napríklad aký nástroj je vybraný. Obecne však platí pravidlo, že pri kliknutí myšou sa prevedie akcia napríklad výberu existujúceho objektu, a poznačenie súradníc, pri ťahaní myšou sa len prekresľuje plocha na základe práve určenej akcie a pri pustení sa aplikuje permanentná zmena.

5.3 Štandardný panel nástrojov

Z mnoho aplikácií poznáme štandardný panel nástrojov. Panel nástrojov je v Java knižnici SWING definovaný triedou JToolBar. Táto trieda v sebe nesie možnosť uloženia tlačidiel, ktorým môžeme definovať funkcie. Tieto funkcie potom interaktívne zasahujú do behu aplikácie. Tento panel sprístupňuje dôležité akcie celej aplikácie. Jedná sa o zoznam akcií:

- Nový súbor
- Otvorenie súboru
- Uloženie súboru
- Import z PNTalk formátu
- Export do PNTalk formátu
- Export do obrázkového formátu
- Export do EPS formátu
- Undo
- Redo
- Ďalšie dôležité akcie

Všetky nástroje sú reprezentované pomocou tlačidiel. Tlačidlo sa definuje v SWING knižnici triedou JButton. Popritom táto knižnica nám ponecháva širokú možnosť škálovateľnosti tlačidla. Použijeme tak možnosť bez vykreslenia textu len s použitím obrázku. Použitie obrázku musí byť jednoznačné a preto pri výbere musíme byť opatrní. Avšak môžeme si napomôcť použitím alternatívneho textu. Tento text vysvetľuje funkcionalitu tlačidla a zobrazí sa ak nad tlačidlom postojíme s myšou.

5.4 Panel nástrojov

Panel nástrojov je umiestnený na ľavej strane aplikácie a definuje hlavne použité nástroje, pomocou ktorých užívateľ dokáže zasahovať do kresliacej plochy. Môže byť znovu zrealizovateľný pomocou komponentu JToolBar a pridaných tlačidiel. Pri tomto panelu však je praktickejšie pre tlačidlá použiť

komponenta JToggleButton. Rozdiel je v tom, že JToggleButton oproti JButton dokáže uchovať stav. To znamená, že ak užívateľ spraví výber nástroja, tak tento výber zostáva permanentný, čo reprezentuje tlačidlo, ktoré zostáva zatlačené.

Jednotlivé nástroje sa identifikujú pomocou obrázkov a alternatívneho textu. Užívateľ tak pri trocha šikovnosti intuitívne príde na funkčnosť tlačidiel. Zoznam nástrojov, ktorý panel nástrojov implementuje je nasledujúci:

- Nový objekt synchrónny port
- Nový objekt miesto
- Nový objekt prechod
- Nový objekt hrana
- Tvarovanie hrany
- Výber a posunutie objektu
- Vymazanie objektu

Ak sa zamyslíme nad zoznamom, tak vidíme, že chýba možnosť vytvorenia siete. Táto možnosť však nevyžaduje grafickú interakciu, nakoľko grafická podoba OOPN siete záleží na rozložení objektov. Možnosť vytvorenia novej siete aplikácia musí povoliť, avšak k tomu použijeme iný komponent.

5.5 Menu

Väčšina aplikácií z pravidla implementuje menu. Do menu sa dávajú prvky pre lepšiu orientáciu, ďalej tie prvky a akcie, ktoré sú druhoradé alebo sa týkajú nastavení alebo ktoré sa inde nezmestili. Ani naša aplikácia nebude výnimkou. Menu implementuje všetky akcie zo štandardného panelu nástrojov, a ešte aj príkazy navyše.

Menu sa rozdeľuje spravidla na niekoľko hlavných častí. V našej aplikácii týchto hlavných častí budú štyri časti. K zobrazeniu použijeme JMenuBar, pre vytvorenie celkovej lišty, ďalej rozčleníme jednotlivé časti na JMenu s položkami JMenuItem.

5.5.1 File

Prvý člen je ponuka Súboru. Pod túto ponuku zaraďujeme manipulácie so súborom a to presne:

- Vytvorenie nového súboru
- Otvorenie súboru
- Uloženie súboru
- Importovanie z PNTalk formátu
- Export do PNTalk formátu
- Export do Obrázkového formátu
- Export do EPS formátu
- Ukončenie aplikácie

Ako vidíme tieto funkcie sú všetky pridružené k celkovej práci so súborom alebo teda OOPN triedou.

5.5.2 Edit

Ďalší člen je ponuka Editácie. Pod týmto bodom sa skrývajú tie možnosti, ktoré majú priamo vplyv na aktuálne editovanú OOPN triedu. Do tejto ponuky zaraďujeme nasledujúce operácie:

- Undo

- Redo
- Validáciu predkov OOPN triedy
- Panel nastavenia OOPN vlastností

Z ponuky vyčnieva zobrazenie panela nastavenia, avšak ak sa logicky zamyslíme, že sa vlastne možnosťami nastavenia editujú grafické vlastnosti triedy, tak právom zaraďujeme túto ponuku pod bod editácie.

5.5.3 View

Pod túto ponuku uložíme tie nastavenia, ktoré majú priamy vplyv na celkový výzor editora. Patria sem hlavne zobrazenia a skrývania jednotlivých častí GUI. Zoznam je nasledujúci:

- Zobrazenie panelu hierarchie
- Zobrazenie tabuľky vlastností
- Zobrazenie okna zdrojového kódu

5.5.4 Help

Do tejto položky sa pridávajú základné informácie o aplikácii, ako ju používať a klasické položky nápovede.

5.6 Panel hierarchii

Tento panel slúži na zobrazenie jednotlivých objektov tak, ako sa logicky skladajú v rámci celkovej triedy. To znamená, že na vrcholu je vždy zobrazená objektová sieť, pod nej sa zobrazujú siete metód a konštruktorov, a pod nimi synchrónne porty. Tieto objekty sú si rovnocenné v rámci OOPN triedy, a tak sa zobrazujú aj v paneli hierarchii na tej istej úrovne. Podradené objekty sú prechody a miesta. Tieto sa zobrazujú vždy tak, že sa nadväzujú na niektorú sieť. Tvoria preto úroveň druhú, podradenú.

Pre zobrazenie hierarchii poslúži najlepšie komponent JTree. Jedná sa o stromové zobrazenie, kde sa jednotlivé OOPN objekty môžu mapovať ako uzly. Pritom SWING komponent JTree má taktiež dobrú možnosť škálovateľnosti a dynamického prekresľovania. Môžeme tak jednoduchú stromovú štruktúry prekresliť ako komplexný zdroj informácií. Môžeme napríklad zobraziť objekty, ktoré sú zdedené, zobraziť inou farbou. Taktiež aj objekty preťažené môžeme odlíšiť farbou. Podobne môžeme zaviesť aj vlastný obrázok podľa typu objektu. Ak to presnejšie rozoberieme, tak každý OOPN objekt v zobrazení hierarchii má vlastnú ikonu. Táto ikona ju odlišuje, a tak sa jednoznačne už na prvý pohľad dá určiť, či sa jedná o miesto, prechod, synchrónny port, alebo o objektovú sieť alebo o sieť konštruktora či metódy. Taktiež farebné značenie bude odpovedať dedičnosti.

- Ak je meno OOPN objektu na bielom pozadí s čiernym písmenom, tak sa jedná o objekt definovaný v aktuálnej triede.
- Ak je meno OOPN objektu na čiernom pozadí bielym písmenom, tak sa jedná o zdedený objekt z inej triedy.
- Ak je meno OOPN objektu na zelenom pozadí so žltým písmom, tak sa jedná o preťažený objekt.
- Ak je meno OOPN objektu na modrom pozadí tak je aktuálny objekt vybraný.

5.6.1 Popup Menu

Ako ďalšia možnosť je panel hierarchii urobiť interaktívny. To znamená, že pridať vyskakovacie menu k paneli, ktoré sa zobrazí kliknutím pravého tlačidla myši. K vytvoreniu tohto menu môžeme znova použiť JMenu s položkami JMenuItem. Keďže toto menu má byť interaktívne, tak sa presne musí vedieť aj na ktorý objekt sa kliklo.

Ako ďalšia možnosť využitia menu je, že ako sme spomenuli pri paneli nástrojov, tak nemáme zahrnutú nikde možnosť pridávania novej siete. Tak ju pridáme teraz do interaktívneho menu. Ďalšie funkcie sú v nasledujúcom zozname:

- Odstránenie OOPN objektu
- Pridanie nového prechodu
- Pridanie nového miesta
- Pridanie nového konštruktora alebo metódy
- Pridanie nového synchrónneho portu
- Preťaženie OOPN objektu

5.7 Tabuľka vlastností

Tabuľka vlastností je vlastne komponent JTable. Je uložená v pravej dolnej časti aplikácie a interaktívne spolupracuje s dianím v celej aplikácii. To znamená, že sa na základe aktuálne vybraného objektu naplnia atribúty, teda vlastnosti objektu. Ako vieme, objekty OOPN sa môžu líšiť v celej hromade vlastností. Napríklad niektoré objekty majú meno, iné objekty nie, niektoré definujú prechod, iné hranové výrazy. Taktiež niektoré objekty môžeme editovať, a iné nie. To všetko sú vlastnosti a musíme ich zobraziť. Taktiež tabuľka je ten nástroj, ktorý nám umožňuje tieto vlastnosti meniť. Musíme teda definovať vlastnosti, ktoré sa pripíšu do tabuľky. Základné rozloženie nám poskytne typ vybraného objektu.

V nasledujúcich bodoch rozdelíme presné znázornenie riadkov tabuľky, podľa vybraného typu objektu.

5.7.1 Trieda

Priamo vybrať celú triedu je najlogickejšie asociovať s vybratím Objektovej siete. V tomto prípade sa môžu modifikovať vlastnosti celkovej triedy. Keď je vybraná OOPN trieda, tak sa vždy zobrazia nasledujúce vlastnosti v tabuľke:

- Identifikátor vybraného elementu, ktoré v tomto prípade bude vždy trieda.
- Typ objektu bude trieda
- Meno triedy
- Meno nadradenej triedy

Zo spomenutých vlastností je možné modifikovať len meno triedy. Ostatné vlastnosti sú striktne dané.

5.7.2 Sieť

Keď máme vybranú sieť, tak je dôležité uvedomiť si, že ktoré vlastnosti, a na základe čoho sa majú modifikovať. Ako vieme, celková sieť môže byť zdedená a v tom prípade sa nemôže modifikovať nič. V prípade ak je sieť preťažená v aktuálnej triede, tak sa modifikácie prevádzať môžu. Preťaženie

siete však musí byť len implicitné, nakoľko názov siete musí byť identický, inak sa nejedná o preťaženie ale o definíciu novej siete.

Objektovú sieť považujeme za jedinečnú a pre zobrazenie ju asociujeme s triedou. Do kategórie sietí tak zahrňujeme len sieť metódy a sieť konštruktora.

Vlastnosti, ktoré sa zapisujú do tabuľky sú nasledujúce:

- Identifikátor siete, ako je uložený v rámci triedy.
- Typ siete. Môže mať hodnotu METHOD alebo CONSTRUCTOR.
- Pomenovanie triedy.
- Informáciu o tom, či je sieť zdedená preťažená alebo nová.

Pri sieti sa môže meniť názov, ak sa nejedná o zdedenú alebo preťaženú triedu. Ostatné atribúty sú striktne dané.

5.7.3 Miesto

Miesta sú základné prvky a nie sú nezávislé, takže musíme zahrnúť aj vlastnosť závislosti na inom OOPN objekte, ktorý je sieť. Ak je miesto ešte aj v objektovej sieti, tak môže byť zdedené priamo, kým ak je súčasťou inej triedy, tak je dedičnosť nepriama, a dedí sa z dedičnosti siete. Zobrazené vlastnosti sú nasledujúce:

- Identifikátor miesta.
- Typ, jednoznačne označený značkou PLACE
- Názov miesta
- Počiatočné značenie
- Názov nadradenej siete
- Dedičnosť, ktorá môže značiť zdedený sieť, alebo definovaný v rámci aktuálnej triedy.

Pri mieste je možné modifikovať názov miesta a počiatočné značenie. Ostatné atribúty sa menia buď nepriamo alebo sú striktne dané.

5.7.4 Prechod

Prechod podobne ako miesto je taktiež závislé na sieti. Ak je súčasťou objektovej siete, tak dedičnosť sa kontroluje nezávisle, kým pri ostatných sieťach je priamo spojená s dedičnosťou sietí. Zobrazené vlastnosti sú nasledujúce:

- Identifikátor prechodu.
- Typ, ktorý je označený značkou TRANS
- Názov prechodu
- Stráž prechodu
- Akcia prechodu
- Názov siete, do ktorej prechod patrí.
- Dedičnosť, ktorá môže značiť zdedený prechod, alebo definovaný v rámci aktuálnej triedy.

Dokážeme z uvedených vlastností modifikovať vlastnosti názov, stráž a akciu..Ostatné atribúty sú pevne dané alebo spojené s iným objektom.

Pri prechode by sme mohli byť schopný zahrnúť aj zoznam hrán, ktoré k nemu patria, avšak keďže podľa interného zobrazenia budeme brať hrany ako nezávislé, tak sa k nim budeme správať aj tu tak isto, a neuvádzame ich ako súčasť prechodu.

5.7.5 Synchrónny port

Synchrónny port je nezávislý OOPN element triedy, jeho vlastnosti sú dané nasledujúcimi vlastnosťami v tabuľke:

- Identifikátor synchrónneho portu
- Typ, ktorý je označený názvom SYNC
- Meno synchrónneho portu
- Stráž
- Dedičnosť

Meniť môžeme názov synchrónneho portu a stráž. Podobne ako pri prechode, tak ani u synchrónneho portu nezobrazujeme zoznam hrán.

5.7.6 Hrana

Hrana taktiež nie je samostatný objekt, ale patrí k prechodu alebo k synchrónnemu portu. Zobrazujeme nasledujúce vlastnosti:

- Keďže hrana nemá názov, tak len označenie že máme vybranú hranu.
- Typ hrany, v tomto prípade to bude vždy jedna z trojíc COND, PRECOND alebo POSTCOND
- Hranový výraz
- Dedičnosť, priamo odvodená od prechodu alebo od synchrónneho portu.

5.8 Okno zdrojového kódu

Ako ďalší prívlastok grafického užívateľského rozhrania špecifikujeme samostatné okno pre textové zobrazenie. Keďže PNTalk trieda má presnú syntax jazyka, pomocou ktorého sa môže definovať, zabudujeme možnosť zobraziť aktuálnu triedu v tomto jazyku. K tomu potrebujeme plochu, kde môžeme vložiť text. V Jave existuje komponent zvaný JTextArea, ktorý postačuje k tomuto cieľu. Potrebujeme teda vytvoriť okno, ktoré bude mať v sebe zapuzdrený tento komponent.

5.9 Okno nastavenia vlastností

Potrebujeme vytvoriť v rámci našej aplikácii možnosť, ako zmeniť dynamicky vlastnosti triedy a jednotlivých OOPN objektov. V časti o atribútoch triedy sme zaviedli presne, že o aké vlastnosti sa jedná, teraz ich rozoberme podľa komponentov ktoré použijeme. Celé okno nastavenia by malo pripomínať klasické nastavovacie okno, prehľadné a jednoznačné, že čo sa práve mení. Hlavný komponent je JFrame. Keďže vieme, že okno zobrazuje vlastnosti jednej triedy, tak vlastnosti musia dynamicky prispôsobovať zmenám. Jednotlivé časti nastavenia sú nasledujúce.

5.9.1 Nastavenia veľkosti textu a minimálnej veľkosti objektu

Nastavenie veľkosti textu a minimálnej veľkosti objektu sú celočíselné údaje, a potrebujú jemné ladenie. Každý pixel je rozhodujúci, lebo rázne sa môže zmeniť vykresľovanie. Preto použijeme behúň alebo aj nastaviteľný kontakt pre nastavenie týchto vlastností. Behúň sa v Jave reprezentuje komponentom JSlider. Môžeme nastaviť hranice, ako minimálnu a maximálnu hodnotu. Taktiež pre

vykreslenie týchto vlastností dodáme aj JTextField, do ktorého sa dynamicky dopisuje aktuálne nastavená hodnota. Takto sa jednoducho dokážeme orientovať v aktuálne nastavených hodnotách.

5.9.2 Nastavenie veľkosti kresliacej plochy

Nastavenie veľkosti kresliacej plochy sú celočíselné údaje. Tieto údaje nie sú až také citlivé, a ich rozsah môže byť pomerne veľký. Nebolo by praktické aj tu použiť JSlider, tak od toho odstúpime, a namiesto toho pridáme len JTextField, kde užívateľ z klávesnice dokáže priamo zadať hodnoty výšky výkresu a šírky výkresu.

5.9.3 Nastavenie farieb

Jednotlivé farebné údaje budú zobrazené zvlášť v samostatných riadkoch. Pre každý farebný údaj použijeme návesti, reprezentované komponentmi JLabel , aby sme rozlíšili o čo sa vlastne jedná. Samotná farba bude dokreslená do komponentu JButton. Užívateľ tak už na prvý pohľad dokáže rozoznať aktuálnu farbu. Zmenu farby dokážeme previesť kliknutím na JButton. Vtedy sa zobrazí zabudovaný JColorChooser, ktorý nám pomôže vo výbere farby.

5.9.4 Tlačidlá

Pri paneli nastavení použijeme niekoľko štandardných tlačidiel, ktoré poslúžia pre manipuláciu s nastaveniami. Použijeme klasické tlačidlá ako:

- OK – aplikuj nastavenia a zavri okno
- Cancel - Stornuj nastavenia a zavri okno
- Apply - Aplikuj aktuálne nastavenia ale ne
- Default – Aplikuj predvolené nastavenia, pevne definované v aplikácii ale nezatvor okno

6 Funkcionalita

Z hľadiska funkcionality musíme rozlíšiť možnosti, ktoré má aplikácia zvládnuť. V tejto časti sa nezameriame len na grafické zobrazenia, ale taktiež na ukladanie, a konvertovanie do nového vektorového formátu alebo import z PNTalk.

Tieto časti sú celým jadrom aplikácie. Zahrňujú hneď niekoľko rôznych manipulácií s komponentmi. Na jednej strane sa musí vždy aktualizovať kresliaca plocha, ďalej sa musí aktualizovať panel hierarchií a ako tretí dôležitý prvok sa musí podľa potreby aktualizovať aj tabuľka možností. Tieto tri časti rozoberieme zvlášť v rámci jednotlivých podkapitol.

6.1 Vykreslenie kresliacej plochy

Kresliaca plocha vlastne reprezentuje hlavný komponent aplikácie. Chová sa jednak ako pasívny prvok, teda slúži len na to, aby vierohodne previedla vizualizáciu interného zobrazenia, popritom však musí zachovať dynamiku pri aktualizácii jednotlivých OOPN objektov. To značí prekresľovanie viackrát. Avšak musí implementovať aj aktívne chovanie, nakoľko pri kreslení, teda ťahaním drag and drop nového objektu, užívateľ priamo interaguje s kresliacou plochou, a núti ju dynamicky sa prekresľovať.

Ako prvý z návrhových otázok sa nám javí, že ako presne interpretovať interné zobrazenie objektov na kresliacu plochu. Ak musíme zodpovedať túto otázku, tak najprv si musíme vyjasniť, ako by sme kreslili jednotlivé objekty. Vieme, že v Jave sa pre vykreslenie používa mnoho grafických primitív. Tieto primitíva použijeme aj mi. Existuje mnoho tvarov, ale v ďalších bodoch vyberieme len tie, ktoré nám poslúžia v našej aplikácii. Jedná sa o nasledujúce tvary:

- Vyplnená elipsa
- Obrys elipsy
- Vyplnený obdĺžnik
- Obrys obdĺžnika
- Text
- Čiara
- Tvary –nami definované

Keďže máme k dispozícii sadu grafických primitív, môžeme porozmýšľať nad tým, ako pomocou nich interpretujeme naše OOPN objekty. Vieme, že objekty máme uložené v jednotlivých skladištiach, teda v zoznamoch. Prechádzame tak cez zoznamy, a vypočítame tvary jednotlivých OOPN objektov.

6.1.1 Permanentné zobrazenie

Jedná sa o zobrazenie objektov, ktoré sú uložené v internej databáze. Tieto objekty sa väčšinou zobrazujú vždy na kresliacej ploche, akoby priamo splynuli s kresliacou plochou. Je ich však nutné prekresliť ak sa niečo zmení v databáze, alebo zmenia sa grafické vlastnosti OOPN triedy, ako farby veľkosti textu, objektov, alebo miery kresliacej plochy.

Objekty miesta znázorníme pomocou vyplnenej elipsy a dvoch textových prvkov. Jeden textový prvok bude znázorňovať názov OOPN miesta, a druhý bude znázorňovať počiatočné značenie. Toto počiatočné značenie pritom dokáže dynamicky meniť veľkosť miesta. To je dané

nasledujúco. Ak minimálna veľkosť elipsy, definovaná globálne, je taká, že sa do nej zmestí vykreslenie počiatočného značenia, tak sa nič nemení, avšak keď veľkosť počiatočného značenia presahuje veľkosť elipsy, tak sa miery elipsy dynamicky zväčšujú, pokiaľ sa do nej text nezmestí. Tento údaj sa uchová aj v internom zobrazení.

Objekty prechodu, sú znázornené pomocou vyplneného obdĺžnika, a troch textových údajov. Tu platí taktiež, že názov prechodu sa vykresľuje z pravého horného rohu obdĺžnika. Ďalej texty stráže a akcia sa vkresľujú do tela obdĺžnika, pričom veľkosť obdĺžnika dokážu dynamicky zväčšiť, ak sa nezmestia do minimálne definovanej veľkosti. Stráž popritom musíme odlíšiť od akcie. To sa prevedie modifikáciou fontu. K štandardnému fontu pridáme podčiarknutie, ako je to aj v definícii OOPN.

Objekt sietí vykresľujeme len v špeciálnych prípadoch. Objektovú sieť nekreslíme vôbec, kreslíme len siete konštruktora a metódy. Siete nemajú pevne stanovené hranice. Tie sa vždy pri každom vykreslení počítajú z objektov, ktoré do siete zapadajú. K vykreslení siete použijeme vyplnený obdĺžnik. Súradnice získame tak, že nájdeme najprv objekty, ktoré majú byť zobrazené na najpravejšej, najľavejšej, najhornejšej a najdojnejšej strane, nasledujúco k týmto súradniciam pridáme odstup, aby objekty neboli priamo prilepené na stenu siete, a máme výsledné súradnice pre vykreslenie. Použijeme aj textový údaj, pomocou ktorého vykreslíme aj názov siete, ktoré pripíšeme k ľavej hornej súradnici.

Ďalší objekt pre vykreslenie je synchrónny port. Pomerne sa podobá na prechod, avšak sú znaky, ktoré ho odlišujú. Použije sa pri vykreslení vyplnený obdĺžnik a dve textové údaje. Prvý textový údaj sa vykresľuje do ľavého horného rohu a udáva názov synchrónneho portu. Druhý textový údaj sa vkresľuje do teľa obdĺžnika a označuje stráž synchrónneho portu. Ak by stráž bola graficky väčšia ako samotný obdĺžnik, tak sa obdĺžnik dynamicky zväčšuje.

Ako posledné sa vykresľujú hrany. Hrany z pohľadu grafického sú reprezentované čiarami, presnejšie sekvenciou čiar. Ďalej jedným textovým údajom. Pri každej hrane je počet sekvencií čiar minimálne jedna. Ak sú definované dodatočné body, tak sa počet čiar zvyšuje na n+1, kde n je počet definovaných bodov, cez ktoré hrana prechádza. Pritom kvôli lepšej identifikácii okolo týchto bodov prikreslíme malý štvorec, aby bolo jednoznačné, že sa jedná o zlom v hrane. Vykreslenie textu je pomerne zložitá záležitosť. Najprv musíme špecifikovať na ktorý segment prikreslíme text, teda hranový výraz. Pre jednoduchosť určíme vždy prostredný segment. Orientácia textu je vždy daná vodorovne. Ak je aktuálny stredný segment stúpajúci, tak sa hranový výraz vykreslí pred segment, keď má klesajúcu tendenciu, tak sa vykreslí za segment. Ako posledné je odlíšenie typu hrany. Testovacia hrana má šípky na oboch koncoch, vstupná len pri obdĺžniku, teda synchrónnom porte alebo prechode, kým výstupná hrana pri mieste. Šípku znázorníme ako trojuholník, presne na konci čiary, dotýkajúcej sa okraja OOPN objektu. Smer šípky je predurčený podľa smeru segmentu.

Atribúty farieb vykreslenia sú vždy definované pomocou aktuálnej farby vykreslenia. Tento údaj je pevne definovaný medzi atribútmi OOPN triedy. Keďže pre všetky objekty sú priamo definované tieto údaje tak sa len preberú z vlastností OOPN triedy, a nastavia sa pre aktuálne vykresľovanie. Takto sa odlišujú napríklad zdedené objekty od natívnych, definovaných v aktuálnej triede.

6.1.2 Dočasné vykreslenie

Jedná sa o také zmeny v kresliacej ploche, ktoré priamo súvisia s interakciou užívateľa. Jedná sa hlavne o pridanie nového objektu alebo o posunutie existujúceho. Ten časový okamih, pokiaľ sa prevádza drag objektu. To znamená, že tlačidlo myši je stlačené, teda sa ťahá. Rozlišujeme hneď niekoľko typov tejto interakcie.

Prvý typ je, ak je vybrané vykreslenie nového objektu. Môže sa jednať o vykreslenie miesta objektu alebo synchrónneho portu. Pri stlačení myši sa prevedie vykreslenie obrysu objektu podľa výberu, teda buď elipsa alebo obdĺžnik. Ten sa následne kreslí po ploche v tom smere kadiaľ užívateľ ťahá myš. Kreslenie je dočasné, lebo vždy sa musí odstrániť stopa po predchádzajúcej. Po pustení tlačidla myši sa potom predá riadenie vytvoreniu nového objektu a následne sa zapíše objekt do interného zobrazenia. Tým pádom sa už nebude chovať ako dočasný.

Druhý typ je nová hrana, graficky znázornená ako vykresľovanie spojovacej čiary. Tu sa kladie dôraz na interakciu s kresliacou plochou, nakoľko sa musí určiť objekt, ktorý je pri potlačení vybraný. Tento objekt bude potom jedným koncom hrany, a objekt pri upustení myši zas druhým. Dočasné vykreslenie sa tu deje tak, že sa vykreslí čiara od objektu zvolenom pri stlačení k aktuálnej pozícii myši.

Tretí typ dočasného vykreslenia sa týka segmentu čiary. Slúži na to, aby sa pridal nový bod a teda aj segment alebo premiestnil existujúci bod. V tomto prípade nás tiež zaujíma kliknutie myšou, avšak teraz vyhľadávame čiaru v blízosti súradnice miesta kliknutím myšou. Ak máme súradnicu, tak väčšinou máme aj segment. Tento segment eliminujeme z permanentného zobrazenia, a to tak, že predáme funkcii permanentného zobrazenia výnimku a následne plochu prekreslíme. Máme tak zobrazenie, bez tej segmenty hrany, s ktorou ideme pracovať. Musíme brať ešte dve súradnice, ktoré boli pôvodné konce segmentu. Tieto súradnice budú teraz koncové body nášho dočasného zobrazovania. Počiatočný bod bude aktuálna pozícia myši.

Ako posledný typ dočasného zobrazenia je posunutie objektu. Táto funkcia je zameraná hlavne na posunutie OOPN miesta, prechodu a synchrónneho portu. Ak presúvame tieto objekty, najprv ich musíme odstrániť z permanentného zobrazenia. To spravíme znova pomocou výnimky. Avšak musí si dávať pozor, nakoľko pri presune sa aj hrany dynamicky prispôsobujú novej pozícii objektu. Takže vynechávame aj hrany spojené s aktuálne vybraným objektom z permanentného vykreslenia. Následne musíme poznať koncové body segmentov. Tie zistíme ľahko pomocou zoznamu, ktorý nám udáva tie body, s ktorými sa vkreslenie musí spojiť. Dočasné vykreslenie tak zahrňuje pri pohybe myši jednak tvar aktuálne posúvaného objektu a spojenie s koncovými bodmi segmentov hrán. Takto môžeme ľahko vidieť výsledok posúvaného objektu, ktorý po uvoľnení tlačidla myši sa stáva opäť permanentným.

6.2 Prekreslenie panela hierarchii

Ako sme spomenuli panel hierarchií je vlastne stromová štruktúra. Sem sa zapisujú OOPN objekty. Presná štruktúra je znázornená tak, že na vrchole je vždy objektová sieť. Táto sieť je jediná, ktorá je akoby vytvorená od začiatku, a nesmie sa odstrániť. Ako ďalšie sú siete metód a konštruktorov. Ak sa jedná o nededené siete, tak sa môžu odstrániť, inak sa môžu len preťažiť. Pod siete zaraďujeme OOPN objekty prechodov a miest. Tieto objekty sú vždy v druhej úrovni zobrazovania. Ako posledné objekty v prvej úrovni zobrazovania sú synchrónne porty. Do úrovni hierarchii nezahrňujeme hrany, nakoľko by sa neprehľadne zobrazovali, a z definičného hľadiska PNTalku patria k prechodom.

Manipulácia s panelom hierarchií môže byť založená s menu na stlačenie pravého tlačidla myši. Jednotlivé funkcie si rozoberme.

Odstránenie OOPN objektu je položka, ktorá slúži na odstránenie elementu z interného zobrazenia. Musí sa najprv zistiť, či je objekt odstrániteľný. Položka sa zobrazí len vtedy, ak je objekt definovaný v tomto trende. Zdedené objekty sa nemôžu odstrániť, nanajvýš preťažiť. Ak je možné previesť odstránenie objektu, tak sa to prevedie a prekreslí sa kresliaca plocha.

Pridanie nového prechodu sa zobrazí, ak sa klikne pravým tlačidlom myši na niektorú sieť. Ak nie je vybraná sieť, tak táto možnosť sa nezobrazuje. Následne sa zobrazí dialóg pridania nového prechodu a prevedie sa pridanie do interného zobrazenia.

Pridanie nového miesta podobne ako pri prechode sa zobrazí len pri kliknutím tlačidla myši nad sieťou. V tomto prípade pri kliknutí sa taktiež zobrazí dialógové okno, tentokrát okno pridania nového miesta.

Pridanie nového konštruktora alebo metódy, keďže sa jedná o priame napojenie na triedu, sa zobrazí všade keď v paneli hierarchií vyvoláme menu pravého tlačidla. Aj tu sa zobrazí príslušné dialógové okno, ktoré slúži pre pridanie OOPN siete .

Pridanie nového synchrónneho portu taktiež je spojené priamo s triedou. Zobrazí sa teda táto položka taktiež kdekoľvek v paneli hierarchií, keď sa zobrazí menu. Následne po vybratí tejto položky sa taktiež zobrazí dialógové okno, špecifikované pre pridanie synchrónneho portu.

Preťaženie OOPN objektu sa zobrazí len keď sa vyberie buď miesto alebo prechod definovaný v objektovej sieti a tento OOPN objekt je zdedený z inej triedy, alebo ak sa klikne pravým tlačidlom na zdedenú sieť metódy alebo konštruktora.

6.3 Tabuľka vlastností

Interaktivita tabuľky vlastností spočíva hlavne v tom, že ak sa prevedie v rámci celej aplikácie výber niektorého objektu, tak sa dynamicky prispôsobí a vyplní svoje bunky na základe prevedeného výberu.

Výber sa môže realizovať dvoma spôsobmi. Prvý spôsob je, že sa z panelu nástrojov vyberie nástroj pre výber, a následne sa na kresliacej ploche klikne na niektorý objekt. Vlastnosti toho objektu sa okamžite premietajú do tabuľky vlastností.

Ako druhá možnosť je, že sa v paneli hierarchií klikne na niektorý objekt. Ten objekt sa následne považuje za vybraný a jeho vlastnosti sa zapíšu do tabuľky vlastností.

V časti zobrazení sme si definovali, že aké vlastnosti sa zapíšu do tejto tabuľky. Interakcia je jednoduchá. Ak nie je objekt zdedený z inej triedy, alebo ak je, ale súčasne je aj preťažený, tak sa jeho vlastnosti môžu meniť. To naznačíme v tabuľke vlastností tak, že umožňujeme bunky editovať. Následne len užívateľ prepíše požadované hodnoty, ktoré sa premietajú do interného zobrazenia a na vykresľovaciu plochu.

6.4 Undo a Redo

Všetky operácie, ktoré užívateľ previedol musíme ukladať pre prípad, že ich užívateľ chce vrátiť späť. Operácia návratu sa obecne volá undo, kým znovu prevedenia redo. Aby sme mohli definovať operáciu undo a redo, najprv si musíme vyjasniť operácie, ktoré môžeme previesť nad OOPN objektmi. Rozdeľujeme ich do troch základných kategórií a to:

- Vytvorenie objektu
- Vymazanie objektu
- Editácia vlastnosti objektu.

Pritom za editáciu vlastnosti objektu môžeme považovať aj posunutie, keďže vlastne nemeníme nič iné len súradnice objektu, avšak do časti editácie objektu nezapadá manipulácia s jednotlivými hranovými bodmi. Body cez ktoré hrana prechádza je zoznam súradníc. Tento zoznam musíme ošetriť zvlášť a preto zavedieme ďalšie tri typy operácií:

- Vytvorenie hranového bodu

- Vymazanie hranového bodu
- Posun hranového bodu.

Aby sme ďalej boli možní spraviť komplexné operácie, teda sekvenciu hore uvedených operácií, tak nadefinujeme ešte dve značky a to

- začiatok komplexnej operácie
- koniec komplexnej operácie

Keď máme definované operácie, musíme vymyslieť spôsob, ako ich uložiť tak, aby boli reverzibilné. Vytvoríme preto niekoľko zásobníkov, do ktorých budeme ukladať jednak operácie a ďalej objekty, ktoré boli modifikované. Počet zásobníkov sa líši pri každej operácii. Operácia vymazanie a pridanie OOPN objektu len zruší referenciu na objekt. Preto tu stačí len uložiť na vrch zásobníka referenciu na objekt, a následne previesť reverz operácií, teda pri pridaní odobrať a pri mazaní pridať.

Pri editácii vlastnosti objektu použijeme špeciálny spôsob, ktorým vytvoríme klón objektu s pôvodnými atribútmi a potom pozmeníme originál. Pri tomto prístupu musíme uložiť dva objekty. Kvôli prehľadnejšej manipulácii pridáme nový zásobník pre klonované objekty. Pri undo operácie sa len prekopírujú vlastnosti klonovaného objektu späť do originálneho objektu, čím sme dosiahli reverzáciu operácie.

Ďalšie zásobníky sa vytvoria pre body, presnejšie pre súradnice bodov, cez ktoré body prechádzajú. Potrebujeme tu taktiež dve zásobníky, jeden ktorý budeme používať pri všetkých operácií. Tento ukladá referenciu na body. Ďalší zásobník sa použije pri zmene súradníc bodov. Tento zásobník ukladá body, ktoré uchovávajú originálne súradnice pred presunutím bodu.

Kompletný zoznam zásobníkov je tak nasledujúci:

- Ukladanie kódu prevedenej operácie
- Ukladanie OOPN objektu
- Ukladanie klonovaného OOPN objektu
- Ukladanie bodov
- Ukladanie klonovaných bodov

Z každého typu zásobníka spravíme dva kusy, jeden pre operácie undo a druhý pre operáciu redo. Musíme pritom brať do úvahy, že pri každej prevedenej operácii undo sa musia uložiť informácie pre redo a opačne. Následne, ak užívateľ prevedie nejakú operáciu, tak sa záznamy z redo zásobníkov vymažú.

6.5 Import z PNTalk

Jedná sa o dôležitú časť aplikácie. Táto funkcia má za úlohu prekonvertovať zobrazenie OOPN objektov definovaných pomocou jazyka PNTalk do nášho interného zobrazenia. Musí sa preto pridať informácia o grafickom rozložení.

Musíme si uvedomiť, že pre analýzu PNTalku v prostredí SmallTalku existuje vyspelý lexikálny analyzátor, simulátor, ktorý realizuje simuláciu siete. V Jave však nemáme žiadne triedy, ktoré by nám mohli poslúžiť ako základ. Musíme tak implementovať všetko od začiatku.

Aby sme mohli navrhnúť prevod PNTalk kódu do nášho interného zobrazenia, musíme rozobrať syntax jazyka PNTalk. To nám poslúži k tomu, aby sme mohli vytvoriť čo najlepší lexikálny, analyzátor. Cieľom je prekonvertovať informáciu z PNTalku. Nepotrebujeme zložité metódy, ale také postupy, ktoré nám dokážu udeliť očakávaný výsledok, ktorým je úložisko naplnené objektmi OOPN.

6.5.1 Systém PNTalk

Ako sme z predchádzajúcich častiach videli, každý OOPN systém sa skladá z niekoľkých tried. Naše interné zobrazenie pracuje s jednou triedou ako celkom. Prvé kroky teda vedú k tomu, aby sa systém rozobral na jednotlivé triedy a následne spracoval tieto triedy jednotlivo. Kľúčové slová sú *class*. Preto naša aplikácia bude vyhľadávať tieto kľúčové slová a systém podľa nich rozoberie na menšie celky, teda triedy.

6.5.2 Trieda PNTalk

V tejto časti už pracujeme len s jednou triedou. Musí sa uložiť meno triedy, ktoré je priamo definované za kľúčovým slovom *class* a musí sa uložiť aj meno nadradenej triedy. Ak detekujeme, že nadradená trieda je niečo iné ako *PN*, tak musíme spracovať najprv tú triedu. Toto sa môže diať do hĺbky pomocou rekurzie. Vždy sa vracia pritom naplnené úložisko *ObjectContainer*. Všetky elementy v tomto úložisku sa označia ako zdedené a pokračuje sa naplňovaním OOPN objektov aktuálnej triedy.

6.5.3 Siete

Ako prvé časti, ktoré vyhľadáme v kóde PNTalk, budú názvy sietí. Predovšetkým by sa mal vyskytovať názov *OBJECT*, ktorý znázorňuje objektovú sieť. Objektová sieť nemá vlastný názov, takže testovanie ďalších parametrov nie je opodstatnené. Následne siete metódy označené kľúčovým slovom *METHOD* a siete konštruktorov označené kľúčovým slovom *CONSTRUCTOR*. Tieto siete majú parametre podľa definície PNTalk. Tieto parametre ako názov metódy použijeme ako názov nášho objektu **oParentElement**. Globálne sa poznačí že v presne v ktorej sieti sme, aby sa nestratili napojenia s na sieť napojenými objektmi.

6.5.4 Miesta

Ako ďalšie dôležité prvky musíme spracovať miesta. Miesta sa odlišujú kľúčovým slovom *PLACE*. Následne nasleduje názov a v zátvorke počiatočné značenie. Tieto údaje spracujeme samostatne. Názvy ukladáme ako názov miesta nového objektu **oPlace** a počiatočné značenie ako **content string.** Samozrejme uchovávame si aj to, že ktorá sieť je pre aktuálne miesto definovaná, a tak nestrácame konzistenciu.

6.5.5 Prechody

Prechody sa značia kľúčovým slovom *TRANS*. Tieto kľúčové slová nasleduje názov prechodu. Vytvorí sa v internom skladišti príslušný **oTrans** objekt s definovaným názvom. Aj tu sa zachováva odkaz na sieť, aby sa nestrácala konzistencia. Globálne sa poznačí, že práve ktorý prechod bol spracovaný. Tento krok je dôležitý, aby sme dokázali napojiť nasledujúce inskripcie prechodov, a hrany.

6.5.6 Synchrónne pory

Kľúčové slovo naznačujúci, že sa bude jednať o nový OOPN objekt synchrónneho portu je *SYNC*. Za týmto kľúčovým slovom nasleduje názov. Z týchto údajov sa vytvorí nový element **oSync**, ktorý sa

následne uloží do úložiska. Následne sa globálne poznačí, že posledný element bol synchrónny port, aby sme boli schopný napojiť stráž.

6.5.7 Stráž

Stráž je značená kľúčovým slovom *GUARD*. Vždy musí mať napojenie na synchrónny port alebo na prechod, preto sa najprv zistí posledný element, ktorý bol vybraný a následne sa k nemu pripojí hodnota stráže, ako **guard string**. Stráž sa tak stáva neoddeliteľnou súčasťou OOPN objektu.

6.5.8 Akcia

Akcia sa môže vyskytnúť len ako atribút prechodu. V tomto prípade sa akcia značí kľúčovým slovom *ACTION* a nasledujúce riadky definujú akcie. Tieto hodnoty sa uložia k príslušnému prechodu, ktorý bol zadefinovaný pred ňou.

6.5.9 Hrany

Jeden z najcitlivejších operácií je detekcia hrán. Hrany sú z definičného hľadiska v PNTalku napojené na prechody alebo synchrónne porty. Musíme teda brať do úvahy presne OOPN objekt, ktorý bol vybraný pred nimi. Tento objekt, ktorý musí byť typu **oSync** alebo **oTrans**, bude jeden z koncových objektov hrany. Ako sme spomenuli, existujú tri typy hrán, a každé z nich sa líši v kľúčovom slove. Hrana testovacia používa kľúčové slovo *COND*, hrana vstupná *PRECOND* a hrana výstupná *POSTCOND*. Keďže objekty hrany sú objekty typu **oBound**, tak na základe kľúčového slova sa explicitne definuje typ hrany. Po kľúčovom slove nasleduje miesto, na ktoré je hrana napojená. Toto miesto musíme vyhľadať spomedzi uložených miest. Pritom musíme dbať na to, aby sme vybrali presne tú hranu, ktorá je v súlade s objektom prechodu alebo synchrónneho portu. Ako posledné sa ukladá aj hranový výraz, ktorí sa doplní ako **conetnt string.**

6.6 Export do PNTalk

Export objektu do PNTalku používa jednoduchú metódu. Nakoľko interné zobrazenie je veľmi podobné zobrazeniu PNTalk a manipulácia s interným zobrazením OOPN objektov je taktiež pomerne jednoduchá, nerobí žiadny problém pretransformovať dáta.

6.6.1 Meno triedy

Máme v internom zobrazení uchované meno triedy, aj meno triedy z ktorej je zdedená aktuálna trieda. Tieto kľúčové informácie len doplníme pomedzi kľúčové slová

class <meno_triedy> *is_a* <meno_nadradenej_triedy>

6.6.2 Siete

Siete v každom prípade musíme označiť kľúčovým slovom. Kľúčové slovo *object* sa zobrazí hneď po definícii triedy. Následne sa naplnia všetky elementy, spojené s touto sieťou. To sú prechody a miesta. Keď sme vyčerpali všetky prechody a miesta patriace k tejto sieti, až vtedy zapíšeme ďalšiu sieť podľa typu. Ak je to sieť metódy, tak pridáme kľúčové slovo *method*, ak sieť konštruktora, tak *constructor*. Následne zapíšme meno ako parameter.

6.6.3 Miesta

Miesta vždy hľadáme podľa aktuálnej siete. Prejdeme zoznam, a ak natrafíme na takú sieť, ktorá je definovaná ako nadradená pre dané miesto, tak to miesto vypíšeme v nasledujúcom tvare s kľúčovými slovami a zátvorkami:

place <meno>*(* <content_string> *)*

6.6.4 Prechody

Podobne ako pri miestach, tak aj zoznam prechodov prehľadávame v závislosti na sieťach. Ak nájdeme prechod, ktorý korešponduje s aktuálnou sieťou tak ju vypíšeme.

trans <meno>

Ak je definovaná stráž alebo akcia pri prechodoch, tak si aj tú hneď môžeme vypísať v ďalších riadkoch:

guard <guard string>

action <action string>

Ako ďalšie údaje sa potom vypisujú hrany, ktoré si uvedieme neskôr.

6.6.5 Synchrónne porty

Synchrónne porty sú nezávislé objekty. Začíname s vypisovaním, keď sme vypísali už aj posledný sieťový objekt aj so svojimi prvkami. Výpis synchrónneho portu vyzerá nasledujúco:

sync <meno>

Podobne ako pri prechode, tak aj pri synchrónnom porte vypíšeme priamo stráž za definíciou samotného synchrónneho portu.

guard <guard string>

Podobne ako pri prechode, aj tu nasledujú hrany spojené so synchrónnym portom.

6.6.6 Hrany

Zoznam hrán prechádzame niekoľkokrát, a to vždy, keď sa volá výpis prechodu alebo synchrónneho portu. Ak natrafíme na korešpondujúcu hranu, tak musíme rozlíšiť dve veci. Prvá vec je, typ hrany. Podľa typu totiž zvolíme kľúčové slovo, ktoré bude hranu predchádzať. Druhá vec je, či sa jedná o prvý výskyt hrany toho typu alebo nie. Ak sa jedná o prvý výskyt, tak pripíšeme kľúčové slovo, ale ak nie, tak kľúčové slovo vynechávame a pripisujeme údaj len s čiarkou k predchádzajúcemu údaju. Celková definícia tak vyzerá nasledujúco:

- Prvé výskyty:
  cond <meno_miesta> *(* <content string> *)*
  precond <meno_miesta> *(* <content string> *)*
  postcond <meno_miesta> *(* <content string> *)*
- Ostatné výskyty:
  , <meno_miesta> *(* <content string> *)*

Takto sa vybuduje celá definícia všetkých hrán.

6.7 Export EPS

Jedná sa o zaujímavú funkciu. Nakoľko vieme, že encapluslated postscript formát je vlastne v istom slova zmysle taktiež len grafický kontext, preto s ním môžeme zaobchádzať takým spôsobom. K tomuto použijeme externé knižnice k Java, od Apache. Názvy balíčkov sú org.apache.xmlgrpahics.ps. Jedná sa o API vrstvu, ktorá prekonvertuje do súboru všetko, čo sa jej zadá pre vykreslenie. Keďže máme navrhnutú funkciu, ktorá nám spraví vykreslenie na obrazovku, tak s malou modifikáciou tejto funkcie dokážeme vytvoriť súbor vo formáte EPS, zodpovedajúci štandardom.

7 Príklad

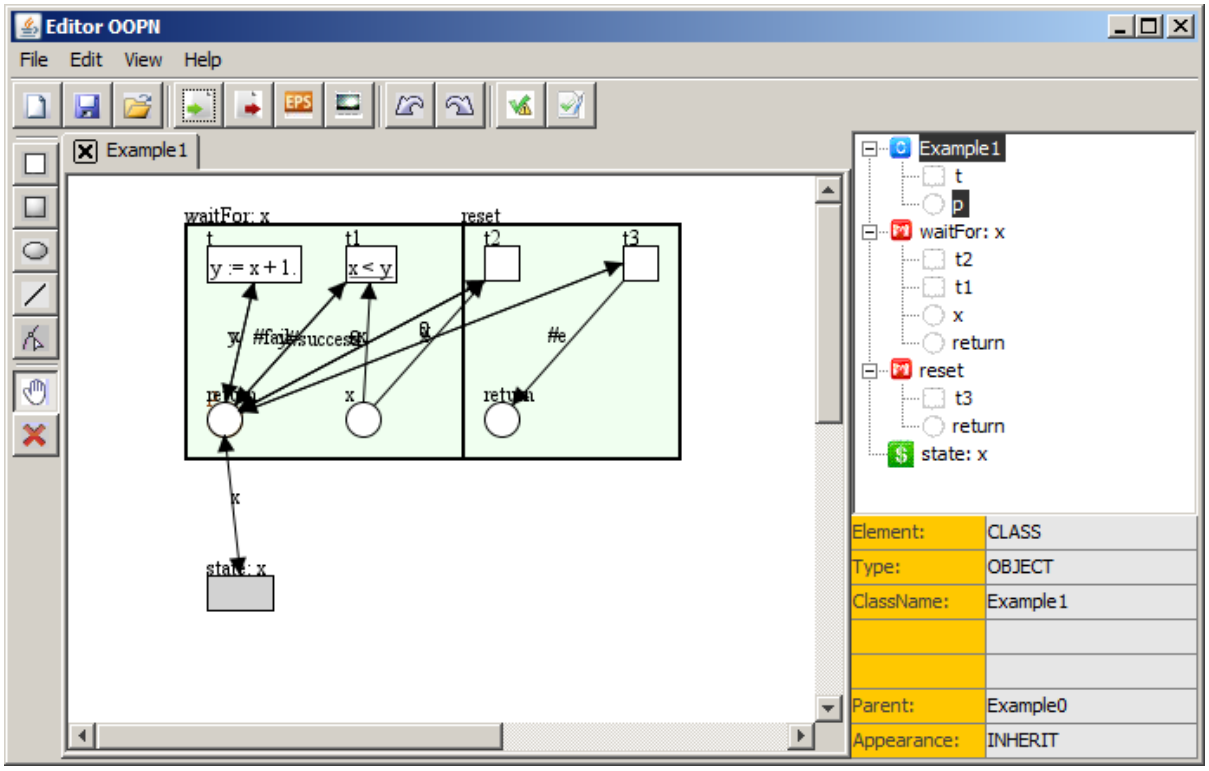
V tejto kapitole si ukážeme praktický príklad aplikácie. Ako príklad si berieme dve triedy napísané v PNTalk jazyku. Trieda Example0 je uložená v súbore *Example1.txt* a má nasledujúci text:

```
class Example0 is_a PN
 object
      place p(0)
```

Trieda Example1 je zdedená z Example0, je uložená v súbore *Example1.txt* a má prepis nasledujúci:

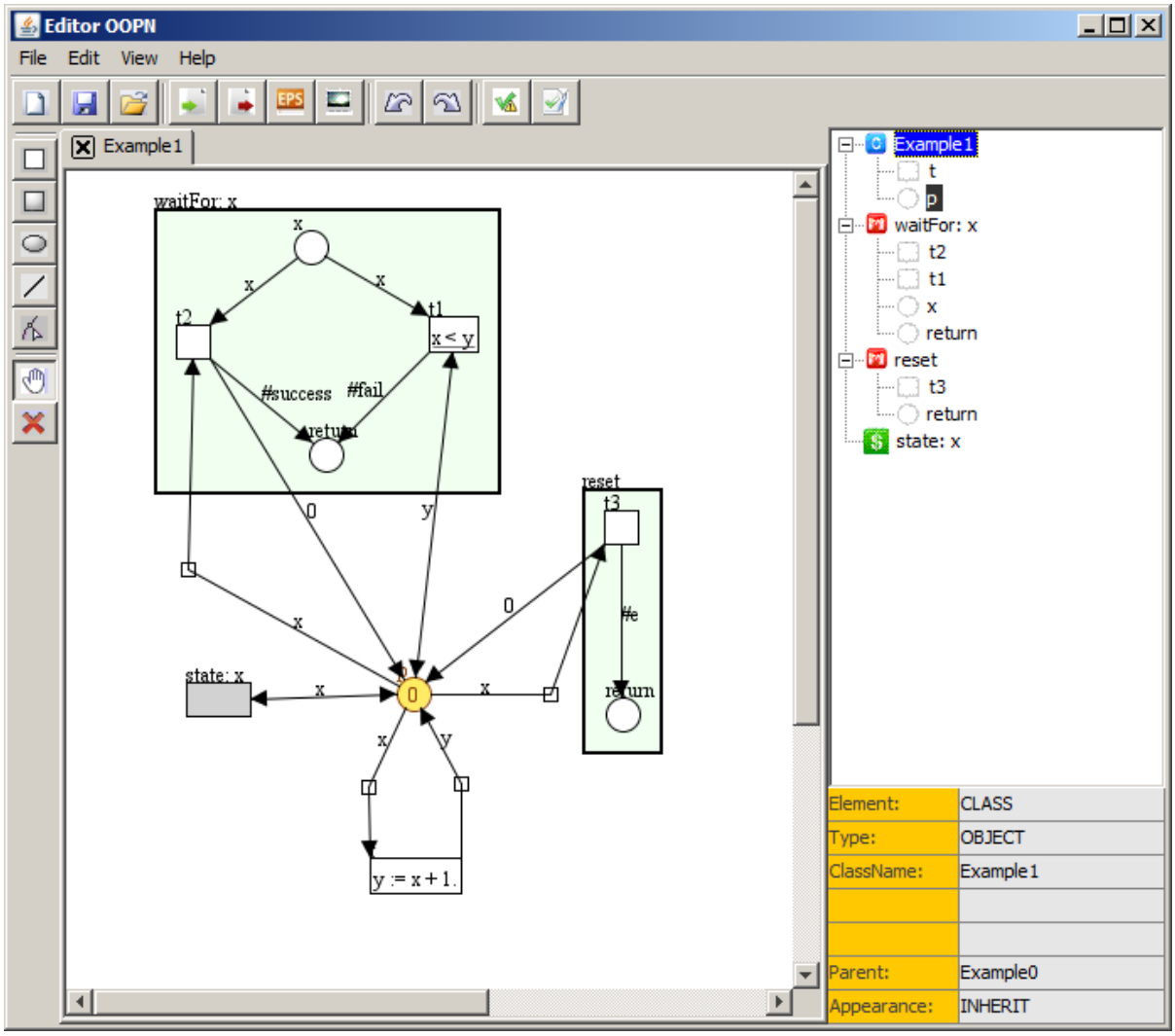
```
class Example1 is_a Example0
 object
      trans t
           precond p(x)
           action {y := x + 1.}
           postcond p(y)
 method waitFor: x
      place return()
      place x()
      trans t1
           cond p(y)
           precond x(x)
           guard {x < y}
           postcond return(#fail)
      trans t2
           precond x(x), p(x)
           postcond return(#success), p(0)
 method reset
      place return()
      trans t3
           precond p(x)
           postcond return(#e), p(0)
 sync state: x
    cond p(x)
```

V aplikácii sa vyberie import *Example1.txt* triedy. Tá automaticky vyhľadá *Example0.txt* a najprv importuje objekty odtiaľ a až potom OOPN objekty z triedy Example1.txt. Výsledok je pomerne neprehľadný, avšak všetky objekty sú už v aplikácii.



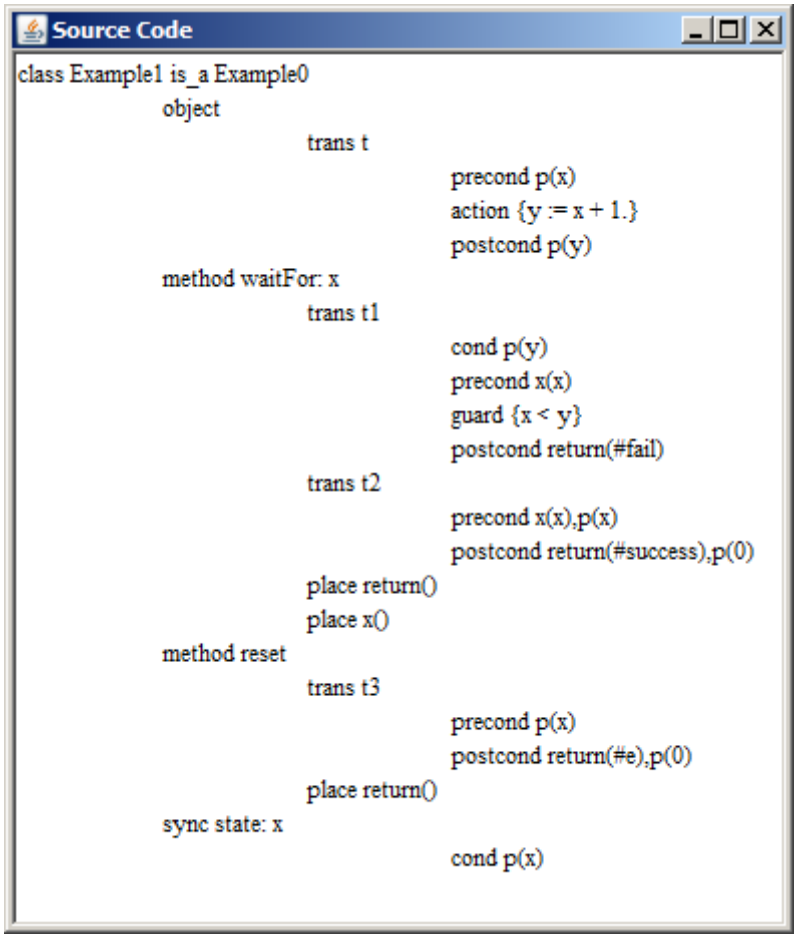
Obrázok 8.1: Aplikácia po importe triedy z PNTalk-u.

Manuálne poopravíme objekty, aby bol prehľadnejší a dostávame nasledujúci obraz:



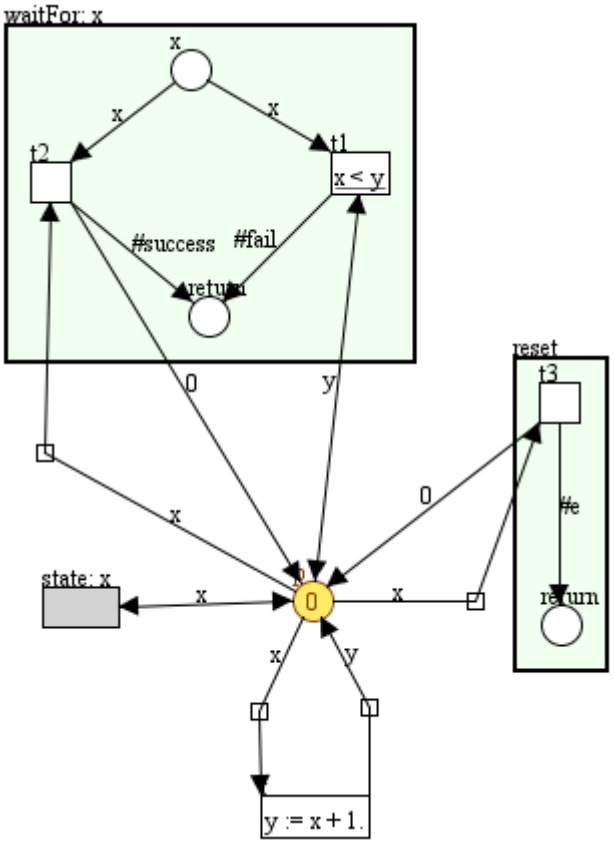
Obrázok 8.2: Aplikácia po vizuálnej úprave OOPN objektov.

Ak sa teraz pozrieme na okno Source Code, vidíme tak PNTalk prepis aktuálnej triedy. Keďže sme ešte nič nemenili, tak prepis je identický s textovým súborom *Example1.txt*



Obrázok 8.3: Okno zobrazujúce zdrojový kód OOPN triedy.

Výsledok dokážeme tak pomocou funkcií aplikácie exportovať do EPS alebo PNG formátu. Výsledok je nasledujúci:



Obrázok 8.4: Exportovaný EPS obrázok-

Ako posledné znázorníme ukážky z XML dokumentu:
Ukážka miesta ***P:***

```
<oPlace>
    <ID>editorOOPN.oPlace@3eb68e0e</ID>
    <parentID>editorOOPN.oParentElement@412beeec</parentID>
    <name>p</name>
    <type>PLACE</type>
    <inheritance>1</inheritance>
    <content>0</content>
    <x>111</x>
    <y>341</y>
    <w>20</w>
    <h>20</h>
</oPlace>
```

Ukážka synchrónneho portu ***State: X:***

```
<oSync>
    <ID>editorOOPN.oSync@4e9722c9</ID>
    <name>state: x</name>
    <inheritance>0</inheritance>
    <type>SYNC</type>
    <x>270</x>
    <y>445</y>
    <w>38</w>
    <h>20</h>
    <guard/>
</oSync>
```

Ukážka hrany ***X*** spájajúce miesto ***P*** so synchrónnym portom ***State: X***

```
<oBound>
    <ID>editorOOPN.oBound@246ce26e</ID>
    <parentPlaceID>editorOOPN.oPlace@3eb68e0e</parentPlaceID>
    <parentTransSyncID>editorOOPN.oSync@4e9722c9</parentTransSyncID>
    <type>COND</type>
    <inheritance>0</inheritance>
    <content>x</content>
</oBound>
```

8 Záver

Cieľom diplomovej práce bolo vytvoriť Editor, pomocou ktorého dokážeme staticky editovať OOPN siete. Základ tvorili siete popísané jazykom PNTalk, takže editor musel zvládnuť importovanie a exportovanie do tohto formátu. Aplikácia tak zvláda tieto operácie bezproblémovo pomocou funkcií. Tento formát je doplnený o grafické atribúty a zobrazený. Editor podporuje editáciu a široké možnosti nastavenia zobrazenia a editácie OOPN triedy, pritom však berie ohľad na to, aby bola ponechaná kompatibilita s formátom PNTalk. Ako ďalšia úloha bola vytvorenie samostatného XML formátu, ktorý dokáže uložiť všetky informácie a grafické atribúty OOPN objektov. Definícu XMS formátu sme pridali ako prílohu. Aplikácia ďalej podporuje možnosti exportovania zobrazenia OOPN triedy do vektorového formátu EPS a do bitmapového formátu PNG. Samotná implementácia prebehla v jazyku Java, takže výsledná aplikácia je platformovo nezávislá.

Zadanie tak bolo splnené. Tento projekt v sebe nesie veľkú škálu možností. Jednak návrh bol presadený podľa pôvodnej špecifikácie PNTalku. Časom sa však táto špecifikácia pozmenila, pribudli nové objekty, a vlastnosti. Návrh konceptu tak nie je ťažké pozmeniť, aby aplikácia bola schopná pracovať s novými objektmi. Možné rozšírenia aplikácie sú dosadenie grafového algoritmu, ktorý dokáže optimálne rozostaviť jednotlivé OOPN objekty, tak aby výzor celkovej triedy bol čo najoptimálnejší. Ďalšie možné rozšírenie je zamyslieť sa nad simulačnými možnosťami aplikácie. Táto úloha by však skôr potrebovala redizajnovania celkového návrhu, nakoľko sa jedná o statický editor a nie simulátor. Potrebovali by sa dosadiť triedy pre simuláciu, ktoré by boli prepojené s vizuálnym výstupom. Ako alternatívu by sme mohli použiť simulačné výstupy z SmallTalk implementácie PNTalku. Týmito krokmi by sme mohli povýšiť náš editor na integrované vývojové prostredie s možnosťami ladenia.

Literatúra

[1] Doc. Ing. Zdena Rabova, CSc.,Prof. RNDr. Milan Češka, CSc., Doc. Ing. Jaroslav Zendulka, CSc.,Dr. Ing. Petr Peringer, Ing. Vladimir Janoušek Ph.D. *Skripty Modelovani a simulace* Dokument dostupný na URL: https://www.fit.vutbr.cz/study/courses/IMS/private/SkriptaMS.pdf 20.5.2010

[2] Doc. Ing. Vladimír Janoušek, Ph.D. *Modelování objektů Petriho sítěmi* Dokument dostupný na URL: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=6001&shortname=1 20.5.2010

[3] RNDr. Milan Češka, CSc., Skripty Petriho síte: *Barvené Petriho síťe* Dokument dostupný na URL: http://www.fit.vutbr.cz/study/courses/PES/public/Prednasky/PES-09-CPN_OOPN.pdf 20.5.2010

[4] Martin Farář *Prednáška ČVUT Vektorová grafika* Dokument dostupný na URL: http://xvrg.webst.fd.cvut.scz/phprs/storage/download/vektor.ppt 20.5.2010

[5] *Oficiálne stránky PNTalk* Dokument dostupný na URL http://perchta.fit.vutbr.cz:8000/projekty/12 20.5.2010

[6] Adobe Systems: *PostScript® Language Reference Supplement* Dokument dostupný na URL: http://partners.adobe.com/public/developer/en/ps/PS3010and3011.Supplement.pdf 20.5.2010

[7] Adobe Systems: *Encapsulated PostScriptFile Format Specification* Dokument dostupný na URL: http://partners.adobe.com/public/developer/en/ps/5002.EPSF_Spec.pdf 20.5.2010

[8] W3C *Scalable Vector Graphics (SVG)* Dokument dostupný na URL: http://www.w3.org/Graphics/SVG/ 20.5.2010

[9] *Corel Draw Tutorial* Dokument dostupný na URL: http://designer-info.com/Draw/corel_draw_tutorial.htm 20.5.2010

[10] Algorithms in Java: Graph Algorithms. Robert Sedgewick. ISBN: 0- 201-36121-3. Publisher: Addison-Wesley

Zoznam príloh

Príloha A. Syntax PNTalk

Pôvodná syntax jazyka PNTalk je definovaná v PH. D. práci [2]. Keďže aj táto definícia bude odohrávať kľúčovú úlohu, musíme ju uviesť aj v rámci tejto práce.

Textový popis štruktúry objektovo orientovanej Petriho siete je v pôvodnom návrhu obvykle automaticky generovaná nástrojom pre vizuálne programovanie v jazyku PNTalk.

Syntax popisu siete, syntax štruktúry siete a celého systému tried je formálne definovaná rozšírenou Backus-Naurovou formou.

- Zápis [...] znamená, že „...“ sa môže vyskytnúť nepovinne,
- Zápis [...]* znamená, že „...“ sa môže vyskytnúť 0 až ľubovoľne krát,
- Zápis [...]* znamená, že „...“ sa môže vyskytnúť 1 až ľubovoľne krát
- Zápis ... | … [| ...]* znamená výber z jednej z variant
- Reťazce v úvodzovkách odpovedajú lexikálnym symbolom
- Počiatočný symbol je *classes*

```
classes: [class]* "main" id [class]*
class: "class" classhead [objectnet] [methodnet|constructor|sync]*
classhead: id "is a" id

objectnet: "object" net
methodnet: "method" message net
constructor: "constructor" message net
sync: "sync" message [cond] [precond] [guard] [postcond]
message: id | binsel id | [keysel id]+
net: [place|transition]*

place: "place" id"(" [initmarking] ")" [init]
init: "init" "f" initaction "g"
initmarking: multiset
initaction: [temps] exprs
transition: "trans" id [cond] [precond] [guard] [action] [postcond]
cond: "cond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
precond: "precond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
postcond: "postcond" id"(" arcexpr ")" ["," id"(" arcexpr ")"]*
guard: "guard" "f" expr3 "g"
action: "action" "f" [temps] exprs "g"
arcexpr: multiset

multiset: [n "`"] c ["," [n "`"] c]*
n: [dig]+ | id
c: literal | id | list
list: "(" [c ["," c]* ["|" [id | list] ]] ")"

temps: "|" [id]* "|"
unit: id | literal | "(" expr ")"
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
exprs: [expr "."]* [expr]
expr: [id ":="]* expr2
expr2: primary | msgexpr [ ";" cascade ]*
expr3: primary | msgexpr
msgexpr: unaryexpr | binexpr | keyexpr
```

```
cascade: id | binmsg | keymsg
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: selchar[selchar]
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel expr2]+
keysel: id":"
literal: number | string | charconst | symconst | arrayconst
arrayconst: "#" array
array: "(" [number | string | symbol | array | charconst]* ")"
number: [-][[dig]+ "r"] [hexDig]+ ["." [hexDig]+] ["e"["-"][dig]+].
string: "'"[char]*"'"
charconst: "$"char
symconst: "#"symbol
symbol: id | binsel | keysel[keysel]* | string
id: letter[letter|dig]*
selchar: "+" | "-" | "*" | "/" | "~" | "|" | "," | "<" | ">" | selchar2
selchar2: "=" | "&" | "n" | "@" | "%" | "?" | "!"
hexDig: "0".."9" | "A".."F"
dig: "0".."9"
letter: "A".."Z" | "a".."z"
char: letter | dig | selchar | "[" | "]" | "f" | "g" | "(" | ")" | char2
char2: " " | "^" | ";" | "$" | "#" | ":" | "." | "_" | "`"
```

Príloha B. XSD Šablóna vytvoreného XML formátu

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema elementFormDefault="qualified"
                    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ObjectContainer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="className" type="xs:anySimpleType" />
        <xs:element name="superClassName" type="xs:anySimpleType" />
        <xs:element minOccurs="1" maxOccurs="unbounded"
                                                name="oParentElement">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:anySimpleType" />
              <xs:element name="name" type="xs:anySimpleType" />
              <xs:element name="type" type="xs:anySimpleType" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="oPlace">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:anySimpleType" />
              <xs:element name="parentID" type="xs:anySimpleType" />
              <xs:element name="name" type="xs:anySimpleType" />
              <xs:element name="type" type="xs:anySimpleType" />
              <xs:element name="content" type="xs:anySimpleType" />
              <xs:element name="x" type="xs:anySimpleType" />
              <xs:element name="y" type="xs:anySimpleType" />
              <xs:element name="w" type="xs:anySimpleType" />
              <xs:element name="h" type="xs:anySimpleType" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="oTrans">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:anySimpleType" />
              <xs:element name="parentID" type="xs:anySimpleType" />
              <xs:element name="name" type="xs:anySimpleType" />
              <xs:element name="type" type="xs:anySimpleType" />
              <xs:element name="x" type="xs:anySimpleType" />
              <xs:element name="y" type="xs:anySimpleType" />
              <xs:element name="w" type="xs:anySimpleType" />
              <xs:element name="h" type="xs:anySimpleType" />
              <xs:element name="action" type="xs:anySimpleType" />
              <xs:element name="guard" type="xs:anySimpleType" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
```

```
        <xs:element minOccurs="0" maxOccurs="unbounded" name="oSync">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:anySimpleType" />
              <xs:element name="name" type="xs:anySimpleType" />
              <xs:element name="type" type="xs:anySimpleType" />
              <xs:element name="x" type="xs:anySimpleType" />
              <xs:element name="y" type="xs:anySimpleType" />
              <xs:element name="w" type="xs:anySimpleType" />
              <xs:element name="h" type="xs:anySimpleType" />
              <xs:element name="guard" type="xs:anySimpleType" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element minOccurs="0" maxOccurs="unbounded" name="oBound">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ID" type="xs:anySimpleType" />
              <xs:element name="parentPlaceID" type="xs:anySimpleType" />
              <xs:element name="parentTransSyncID" type="xs:anySimpleType"
                                                   />
              <xs:element name="type" type="xs:anySimpleType" />
              <xs:element name="content" type="xs:anySimpleType" />
              <xs:element minOccurs="0" maxOccurs="unbounded"
                                                   name="point">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="x" type="xs:anySimpleType" />
                    <xs:element name="y" type="xs:anySimpleType" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

1 Príloha C. – Časť implementácie

Vo fáze návrhu sme popísali koncový návrh aplikácie, so všetkými prvkami, ktoré sa majú implementovať. Časť implementácie prechádzalo cez vývojové štádiá, kde sa vždy niečo implementovalo, otestovala sa funkčnosť a keď všetko sedelo, tak sa implementovalo niečo ďalšie, ktoré dopĺňalo aplikáciu. Takto sme postupne dospeli k výslednej aplikácie. Jednotlivé fázy rozoberieme v tejto kapitole.

Táto kapitola však neponúka podrobný popis všetkých funkcií. Na to slúži technická dokumentácia JavaDoc, vygenerovaná priamo zo zdrojového kódu. JavaDoc pre svoju mohutnosť je priložená na prílohe CD spoločne so samotnými zdrojovými kódmi..

1.1 Globálne identifikátory

Programátorsky je neprehľadné v rámci celej aplikácie pre identifikáciu jednotlivých atribútov používať čísla. Takéto atribúty môžu byť typy vlastností. Taktiež môže byť neprehľadné použiť napríklad atribút definovaný číslom 5. Preto vytvoríme triedy, ktoré dodávajú sémantiku takýmto atribútom, pomocou enumerácií.

1.1.1 MyObjectNames.java

Táto trieda je trieda enumerácii, ktorá slúži nato, aby sme objekty OOPN dokázali oddeliť. Je to vlastne identifikátor typu. Hodnoty sú nasledujúce:

NOTOBJECT,
PLACE,
TRANS,
SYNC,
INHIBITOR,
COND,
PRECOND,
POSTCOND,
OBJECT,
METHOD,
CONSTRUCTOR,

1.1.2 InheritanceType.java

Musíme taktiež oddeliť typy zdedenia objektov. Rozlišujeme tri typy:

INHERIT
NATIV
OVERRIDE

1.1.3 DrawPanel.java

Atribúty nemusíme vždy definovať do oddelenej triedy. Stačí ak im pridáme statický atribút, a tak sa môžeme na tie typy odkazovať, aj keď nemáme vytvorenú inštanciu triedy. Tento postup definujeme v prípade nástrojov. Sú priamo definované v triede pre vykreslenie. Hodnoty sú nasledujúce

SYNC
TRANS
PLACE
LINE
LINESELECT
SELECT
DELETE

1.2 Základné časti

V tejto časti predstavíme triedy, ktorých inštancie bude reprezentácia jednotlivých OOPN objektov. V návrhovej časti sme už podrobne definovali, že ktoré objekty majú aké vlastnosti. V implementačnej časti je potrebné tieto triedy vytvoriť.

1.2.1 iObjects

Obecne sa nejedná o objekt, jedná sa len o rozhranie. Všetky objekty však povinne implementujú toto rozhranie. Slúži nato, aby sme v jednotnej forme mohli pracovať priamo s OOPN objektmi, nezávisle na typu OOPN objektu.

Rozhranie priamo ponúka definíciu metód:
- *getType* – slúžiace na určenie typu objektu
- *setType* – slúžiace na nastavenie typu objektu
- *getInheritanceType* – slúžiace na zistenie dedičnosti
- *setInheritanceType* - slúžiace na nastavenie dedičnosti
- *getName* – slúžiace na zistenie mena objektu
- *setName* – slúžiace na nastavenie mena objektu
- *getXMLID* – slúžiace na získanie XML Identifikátora
- *renewXMLID* – slúžiace na obnovenie XML Identifikátora
- *setSelected* – nastavenie či je objekt vybraný
- *isSelected* – zistenie či je objekt vybraný
- *clone* – vytvorenie kópii objektu kopírovaním všetkých vlastností
- *updateDataFromClone* – nastavenie vlastností z kópii objektu

1.2.2 oParentElement

Je to trieda, ktorej inštancie reprezentujú siete. Má niekoľko konštruktorov. Vždy musíme udať typ triedy. Sú akceptované typy METHOD, OBJECT a CONSTRUCTOR. Ďalej sa musí udať meno objektu a dedičnosť, ktorá je elementom z triedy InheritanceType. V špeciálnom prípade konštruktor triedy zahrňuje aj XML Identifikátor.

Trieda implementuje niekoľko setterov a getterov. Settery sú metódy, ktoré nastavujú atribúty a gettery sú zas metódy, ktoré vracajú tieto atribúty. Jedná sa obecne o:

- dedičnosť funkciami *getInheritanceType* a *setInherintanceType*

- typ funkciami *getType* a *setType*
- meno funkciami *getName* a *setName*
- XML Identifikátor funkciou *getXMLID* a *renewXMLID*

Ďalej musíme použiť atribút, ktorý označuje či je objekt práve vybraný alebo nie. Tento atribút nazveme selected a nastavovaciu funkciu setSelected(boolean), kde ak je parameter true, tak je nastavený, ak false tak nie.

Keďže aj siete v sebe môžu niesť grafické informácie, takže musíme mať metódu, ktorá vypočíta súradnice siete. Takúto metódu pomenujeme ako *initAllCoordinates*, a ako parametre jej predáme ObjectContainer. Metóda si potom sama zvolí objekty, presnejšie prechody a miesta, ktoré pripadajú k nej. Svoje súradnice nastaví tak, aby všetky objekty boli graficky vnútri.

Následne potrebujeme metódy, ktoré nám tieto súradnice pre vykreslenie dokážu dať späť. Tieto metódy sú:

- *getX*
- *getY*
- *getWidth*
- *getHeight*

Ako posledné potrebujeme metódu ktorá vytvorí kópiu objektu. Táto metóda sa nazýva *clone*. Nekopíruje len jednotlivé odkazy na vnútorné parametre, ale kopíruje hodnoty samotné. To je dôležité pri operáciách Undo a Redo. Musíme tak spraviť funkciu, ktorá aktualizuje dáta z kópii objektu. Túto metódu aktualizácie nazveme *updateDataFromClone*.

1.2.3 oPlace

Trieda, ktorá definuje objekty miesta, implementuje rozhranie iObjects. To znamená, že aj tu sa vyskytnú všetky settery a gettery, a metódy definované tam. Ako typ akceptuje PLACE. Miesto má v sebe informáciu o počiatočnom značení. To je ContentString. Operácie s tým atribútom definujeme ako

- *getContentString*
- *setContentString*

Ďalej potrebujeme grafické informácie uložiť a vybrať. K tomuto musíme implementovať funkcie

- *setX* a *getX*
- *setY* a *getY*
- *setWidth* a *getWidth*
- *setHeigth* a *getHeight*

K štandardným getter a setter metódam pridáme niekoľko špeciálnych metód. Jedna metóda nastaví všetky súradnice naraz a ponesie názvov *setAllCoordinates*. Druhá metóda vypočíta a vráti prostredný bod miesta popisujúcej elipsy, a menuje sa *getCenterCoordinate*.

Ďalšia funkcia berie ako parameter súradnice a ak sú tieto súradnice v objekte, tak vracia hodnotu true, inak false. Táto metóda sa volá *isOnCoordinates*.

1.2.4 iTransSync

Je to ďalšie rozhranie. Ako aj meno naznačuje slúži pre objekty OOPN prechody a synchrónne porty. Základný koncept je dôležitý, aby hrany sme dokázali jednotným spôsobom spojiť. Nestačí nám v tomto prípade iObjects rozhranie, lebo nenesie základné grafické informácie, ktoré sú potrebné pre prepojenie s hranou. Tieto informácie sprístupňuje toto rozhranie cez definíciu nasledujúcich metód

- *getX* - získanie súradnice ľavej časti

- *getY* - získanie súradnice hornej časti
- *getWidth* – získanie šírky objektu
- *getHeigth* – získanie výšky objektu
- *getType* – získanie typu objektu
- *getCenterPoint* – získanie súradnice uprostred objektu
- *setSelected* – nastavenie, či je objekt vybraný
- *isSelected* – získanie informácie či je objekt vybraný

1.2.5 oTrans

Táto trieda slúži pre vytvorenie OOPN prechodu. Ako všetky triedy implementuje rozhranie iObjects a všetky tam definované metódy. Tieto metódy však rozširuje o vlastné metódy. Ako bolo v návrhu určené, tak atribútmi tohto objektu sú stráž a akcia. Musíme mať metódy, ktoré nastavia tieto hodnoty a získajú hodnoty. Tieto metódy sú

- *setGuardString* a *getGuardString*
- *setActionString* a *getActionString*

Ako bolo taktiež spomenuté tento objekt v sebe nesie kopu grafických informácií. Tieto grafické informácie sa nastavujú a taktiež vyčítajú metódami:

- *setX* a *getX*
- *setY* a *getY*
- *setWidth* a *getWidth*
- *setHeigth* a *getHeight*

Je možné nastaviť všetky parametre naraz jednou metódou. Táto metóda je *setAllCoordinates.*

Taktiež potrebujeme metódu, ktorá vypočíta a vracia súradnicu prostredného bodu. Tá metóda sa volá *getCenterCoordinate.*

Potrebujeme aj metódu, ktorá nám určí či sa objekt nachádza na daných súradniciach. Ako parameter musíme predať súradnice, a vrátená hodnota nám definuje, či objekt vpadá do daných súradníc hodnotou true. Ak nepadá tak návratová hodnota je false. Metóda nesie názov *isOnCoordinates*.

Nakoľko objekt prechodu nie je samostatný objekt, a musí byť vždy k nemu definovaná sieť, tak teda definujeme *getoParentElement* a *setoParentElement* metódy.

Prechod navyše implementuje aj rozhranie iTransSync. Toto rozhranie slúži na to, aby zjednotilo koncový OOPN objekt pre hrany.

1.2.6 oSync

Synchrónny port potrebuje zvlášť triedu, nakoľko sa nenapája na žiadne objekty, je súčasťou triedy. Pri synchrónnom porte, ako aj pri ostatných OOPN objektoch potrebujeme implementovať metódy z rozhrania iObjects. Tieto metódy udávajú základ, a k nemu pridáme aj implementáciu metód z rozhrania iTransSync. Tieto metódy rozširujú možnosti pre jednotnú grafickú manipuláciu z pohľadu hranového výrazu.

Synchrónny port má ako atribút stráž, takže implementujeme metódy pre manipuláciu so strážou. Tieto metódy sú:

- *setGuardString* a *getGuardString*

Ostatné metódy sú hlavne implementácia metód z rozhraní, rôzne settery, gettery a komplexnejšie metódy pre získanie grafických súradníc alebo iných informácii z objektu. Zoznam týchto metód je nasledujúci:

- *setX* a *getX*

- *setY* a *getY*
- *setWidth* a *getWidth*
- *setHeigth* a *getHeight*
- *setType* a *getType*
- *getCenterPoint*
- *setSelected*
- *isSelected*
- *getName* a *setName*
- *getInheritanceType* a *setInherintanceType*

1.2.7 oBound

Hrana je pomerne zložitý objekt oproti ostatným. Jednak implementuje rozhranie iObjects, avšak metódy *getName* a *setName* sú len prázdne metódy. Metóda *getName* vracia vždy reťazec „*bound*".

Funkcie *getInheritanceType* a *setInherintanceType* korešpondujú vždy s nadradeným elementom, teda synchrónnym portom alebo prechodom. Keď sa mení dedičnosť jednej, tak sa automaticky zmení aj typ dedičnosti hrany.

Hrana má dva konce naviazané jednak na objekty miesta a druhoradé na objekt prechodu alebo synchrónneho portu. Tieto atribúty sa priamo uchovávajú v konštruktore objektu, ale môžeme ich aj explicitne nastaviť metódami. Taktiež potrebujeme metódy, ktoré nám vracajú tieto elementy. Je možné sa na synchrónne porty a prechody odkazovať pomocou jednotného rozhrania iTransSync, ale aj jednotlivo. Pritom platí, že ak je hrana naviazaná na sychrónny port, a žiadame metódu, ktorá nám vráti napojený prechod, tak návratová hodnota je null. To platí aj opačne. Zoznam týchto funkcií setterov a getterov je nasledujúci:

- *setoPlace* a *getoPlace*
- *setoTrans* a *getoTrans*
- *setoSync* a *getoSync*
- *setiTransSync* a *getiTransSync*

Hrana pritom implementuje funkciu *isMissingParentObject*, ktorá akoby kontroluje či sú oba konce nastavené na validný objekt. Metóda vracia false, ak je všetko v poriadku, ak chýba jeden alebo oba koncové objekty tak true.

Hrana v sebe má ako neoddelitelnú časť hranový výraz. Tento hranový výraz musíme nejak uchovať. Použijeme setter a getter, aby sme dokázali uchovať hranový výraz ako parameter. Použijeme meno atribútu ContentString, podobne ako pri mieste. Metódy sú:

- *setContentString* a *getContentString*

Ako ďalšie metódy sú implementované metódy pre grafické zobrazenie. Musíme brať do úvahy, že hrana je vlastne čiara, a potrebujeme funkcie, na modifikáciu a pridanie hranových bodov. K tomu najprv potrebujeme funkciu, ktorá nám vyberie pri kliknutí hranu. Metóda sa menuje *isOnCoordinates*, a vracia hodnotu true, keď zadaná súradnica je poblíž hrany. K výpočtu musíme použiť algoritmické riešenia *isLineIntersectingRectangle*, ktoré ďalej rozoberieme metódou *isLineIntersectingLine*. Návratová funkcia sa určí pomocou toho, či sa obe úsečky pretínajú. Základom bolo použitie Sedgewickovho algoritmu. [10]

Ako ďalšie potrebuje získať zoznam bodov cez ktoré prechádza hrana. Dve koncové body sú vždy závislé na objektoch, ktorými je hrana spojená. Metóda *getWholeRoute* tak vracia úplný zoznam, aj s bodmi priamo spojenými s objektmi. Metóda *getPoints* vracia len súradnice objektov, ktoré sú vložené explicitne. Pri týchto objektoch musíme mať na mysli vždy to, že záleží na poradí, v ktorom prechádzame cez tento zoznam bodov. Preto sa interne musí uchovať aj index, že kde

presne bol bod vybraný, a následne keď sa vkladá nový bod, tak ho musíme dodať na ten index. Preto sme implementovali niekoľko metód. Zoznam je nasledujúci:

- *addPointToSelectedIndex* – pridá sa bod na aktuálne nastavený index
- *getSelectedIndex* – vráti sa aktuálne nastavený index
- *addPointToIndex* – musí sa explicitne ako parameter funkcie definovať index, na ktorý sa pridáva bod
- *addPointSequentially* – špeciálna funkcia, ktorá pridá bod na posledné miesto a nemení index
- *removePoint* – vymaže bod zo zoznamu

Nakoľko vykreslenie takejto hrany nie je vždy najjednoduchšie, preto musíme zaviesť niekoľko špeciálnych funkcií, ktoré nám pomáhajú pri vykresľovaní. Taká funkcia je napríklad funkcia, ktorá presne udá počiatočnú súradnicu textu. Volá sa *getDrawTextStartPoint*. Vypočíta sa presne na základe grafických nastavení, že presne na ktorú stranu hrany sa má text vykresliť, podľa uhlu čiary. Ak sa má text posunúť tak, že koniec textu sa má dotýkať čiary, tak sa začiatok posunie o šírku textu. Šírka textu je pritom vypočítaná z grafických metód.

1.2.8 ObjectContainer

ObjectContainer je vlastne trieda, ktorá slúži ako úložisko všetkých OOPN objektov a ich atribútov. Je to vlastne implementácia OOPN triedy.

Keďže ObjectContainer reprezentuje vlastne OOPN triedu musíme nastaviť mená triedy a mená nadradenej triedy ako atribúty. K tomu slúžia metódy:

- *setOOPNClassName* a *getOOPNClassName*
- *setOOPNSuperClassName* a *getOOPNSuperClassName*

Explicitne v rámci celej triedy vytvoríme OOPN element objektovej siete. K tomu použijeme metódu setParentElement_OBJECT a k získaniu getDefault_OBJECT.

1.2.8.1 Polia

Ako hlavné komponenty použijeme objekty typu ArrayList pre rôzne typy objektov, špecifikovaného v návrhu Existuje päť ArrayListov a to:

- ArrayList<oPlace> oPlaceArray
- ArrayList<oTrans> oTransArray
- ArrayList<oBound> oBoundArray
- ArrayList<oSync> oSyncArray
- ArrayList<oParentElement> oParentElementArray

1.2.8.2 Pridávanie OOPN objektov

Potrebujeme však celú sadu metód, ktoré nám pomáhajú s manipuláciou nad týmito poľami. Ako prvé typy operácií je pridanie a vymazanie objektov. Implementujeme metódu pre jednotné pridanie, ktoré podľa typu sa rozhodne, že o ktorý objekt sa jedná a pridá do príslušného poľa. Táto metóda sa nazýva *addIObjects*. Ostatné metódy sú špecifické typom parametru a nesú názov *add*.

1.2.8.3 Vyhľadávanie OOPN objektov

Ak máme polia k dispozícii, musíme mať možnosť aj vyhľadávania objektov. Implementujeme tak množstvo metód, ktoré slúžia k tomu, aby sme dokázali vyhľadať jednotlivé objekty podľa rôznych atribút.

Rozoberme najprv vyhľadávanie objektov podľa mena. Okrem hrany dokážeme všetky objekty vyhľadať podľa mena. Metódy ktoré nám to umožnia sú nasledujúce:

- *getoParentElementByName*
- *getoTransByName* – existujú dve variácie, jedna používa samotné meno, druhá využíva meno plus meno siete
- *getoPlaceByName* – existujú dve variácie, jedna používa samotné meno, druhá využíva meno plus meno siete
- *getoSyncByName*

Ako ďalšie vyhľadávanie je vyhľadávanie podľa XML identifikátora. Táto funkcia je dôležitá hlavne pri importovaní. Metódy, ktoré nám napomáhajú sú:

- *getoParentElementByXMLID*
- *getoTransByXMLID*
- *getoPlaceByXMLID*
- *getoSyncByXMLID*

Ďalšie dôležité vyhľadávacie metódy sú vyhľadávania podľa súradníc. Implementujeme dva typy metód. Jedna implementácia vracia prvý výskyt daného objektu a druhá implementácia vracia zoznam všetkých objektov, ktoré sa nachádzajú na danej súradnice. Metódy nesú nasledujúce mená:

- *getoSyncOnCoordinates*
- *getoTransOnCoordinates*
- *getoPlaceOnCoordinates*
- *getoBoundOnCoordinates*
- *getoSyncListOnCoordinates*
- *getoTransListOnCoordinates*
- *getoPlaceListOnCoordinates*

Vyhľadávacie funkcie prešli obecne cez optimalizáciu. Za výsledok takejto optimalizácie považujeme aj to, že chýbajú jednotlivé funkcie, ktoré však nie sú použité v samotnej aplikácii.

1.2.8.4 Odstraňovanie OOPN objektov

Odstránenie objektov je podobná operácia ako pridávanie, avšak musíme mať dopredu vyhľadanú referenciu na objekt. K tomu nám napomáha mnoho metód, popísaných vyššie.

Existuje odstránenie ľubovoľného typu objektu pomocou metódy *removeIObjects*. Ostatné odstránenia však odlišujeme rôznymi menami metód:

- *removeParentElementFromArray*
- *removePlaceFromArray*
- *removeTransFromArray*
- *removeSyncFromArray*
- *removeBoundFromArray*

Musíme popritom dávať pozor, že pri odstraňovaní vyšších objektov sa odstraňujú aj naväzujúce objekty. Ak sa teda odstraňuje sieť, tak sa volá funkcia odstránenia všetkých pridružených prechodov a miest. Ak sa volá odstránenie prechodov, alebo synchrónnych portov, tak sa odstránia aj príslušné hrany. Takto zostáva celok konzistentný.

1.2.8.5 Hromadné preťaženie objektov

Za hromadné preťaženie objektov považujeme hlavne preťaženie sietí. Obecne, ak sa preťaží zdedená sieť, tak sa vytvorí nová sieť od začiatku. Je však jednoduchšie, ak všetky zahrnuté v nadradenej triede, prekopírujeme ako vlastné objekty novej implementácie. K tomu nám poslúžia nasledujúce metódy:

- *overrideIObject*
- *overrideParentElement*

- *overridePlace*
- *overrideTrans*
- *overrideSync*

1.2.8.6 Ostatné jednotné metódy

Potrebujeme zo zoznamu vybrať ešte objekty, ktoré sú poznačené pre výber. Toto sa detekuje metódou *isSelected* implementovanou v každom objekte. Jednotná funkcia s manipuláciou nad tým je *getSelectedObject*.

Podobne ako dokážeme vyhľadať vybraný objekt dokážeme vymazať tento atribút zo všetkých objektov. K tomuto slúži funkcia *removeAllSelection.*

Takisto potrebujeme funkciu, ktorá jednotne obnoví XML atribút objektov po importe, aby bolo v korešpondujúcom stave s novými objektmi. Táto funkcia sa volá *renewAllXMLID.*

Taktiež jedna hromadná metóda, je *setAllObjectsAsInherited*. Táto metóda označí všetky prvky ako zdedené. Slúži nato, aby sme použili ObjectContainer ako nadradenú triedu pre novú triedu alebo pri importe z PNTalku.

1.2.8.7 Validácia dedených objektov

Potrebujeme metódu, ktorá jednotným spôsobom dokáže určiť či všetky zdedené objekty sú v poriadku. V PNTalku dedené objekty nie sú priamo v danej triede, a vždy pri každej simulácii sa prevedie určenie dedičnosti. V našej aplikácii to tak nie je, a preto ak sa niečo zmení v nadradenej triede, tak sa to musí prejaviť aj v podradenej triede. Validácia má tri kroky, implementované do rôznych metód. Prvý krok je vytvoriť vnútorné zobrazenie nadradenej triedy. Následne sa dve zobrazenia konfrontujú. Objekty, ktoré majú v podradenej triede atribút dedičnosti INHERIT alebo OVERRIDE sa nájdu v nadradenej triede a vymažú. V aktuálnej triede sa poznačia ako vybrané metódou *setSelected*. Toto použijeme preto, aby sme si poznačili, že ktoré objekty boli už kontrolované. Ak sa nenájde objekt v nadradenej triede, a má atribút INHERIT, tak sa odstráni z aktuálnej, podradenej triedy. Ak má atribút OVERRIDE, tak sa prepíše tento atribút ako NATIV. Ak sa ešte v ObjectContainer-y nadradenej triedy nachádzajú objekty, tak to znamená, že sa medzičasom nadradená trieda privlastnila nové objekty, ktoré sa následne pridajú aj do aktuálnej triedy s atribútom dedičnosti INHERIT. Túto celú činnosť vykonáva metóda *reValidateContainerByParent,* ktorej ako parameter odovzdáme ObjectContainer nadradenej triedy.

1.2.8.8 Vykreslenie

V rámci ObjectContainer-u implementujeme funkciu, ktorá dokáže vykresliť všetky jeho komponenty na základe daných parametrov a zadaného grafického kontextu. Aby sme mohli takú funkciu implementovať, musíme implementovať aj funkciu vypočítania súradníc.

Prvá funkcia sa menuje *setProperObjectSize* a nastaví rozmery objektov uložené v ObjectContainer-i na také, aby korešpondovali s aktuálnym nastavením. Tu sa vypočítava veľkosť objektov sietí, veľkosť jednotlivých prechodov, miest a synchrónnych portov na základe textu, ktoré je do nich napísané.

Ak máme túto funkciu, tak samotné objekty už disponujú požadovanými mierkami. Teraz sa už len musia vykresliť na grafický kontext zadaný ako parameter metódy *paintAllobjects.* Metóda vyžaduje ako ďalšie parametre objekt, ktorý má byť vynechaný z kreslenia. Neskôr toto vykreslenie budeme nazývať ako permanentné kreslenie a dôvod bude vyjasnení tam. Jednoducho je tu možnosť vynechať jeden objekt zo zobrazenia celku. Ako návratová hodnota tejto funkcie je zoznam bodov, s ktorými je tento vynechaný objekt spojený. Ani tieto hrany sa nevykresľujú. Samotné vykreslenie

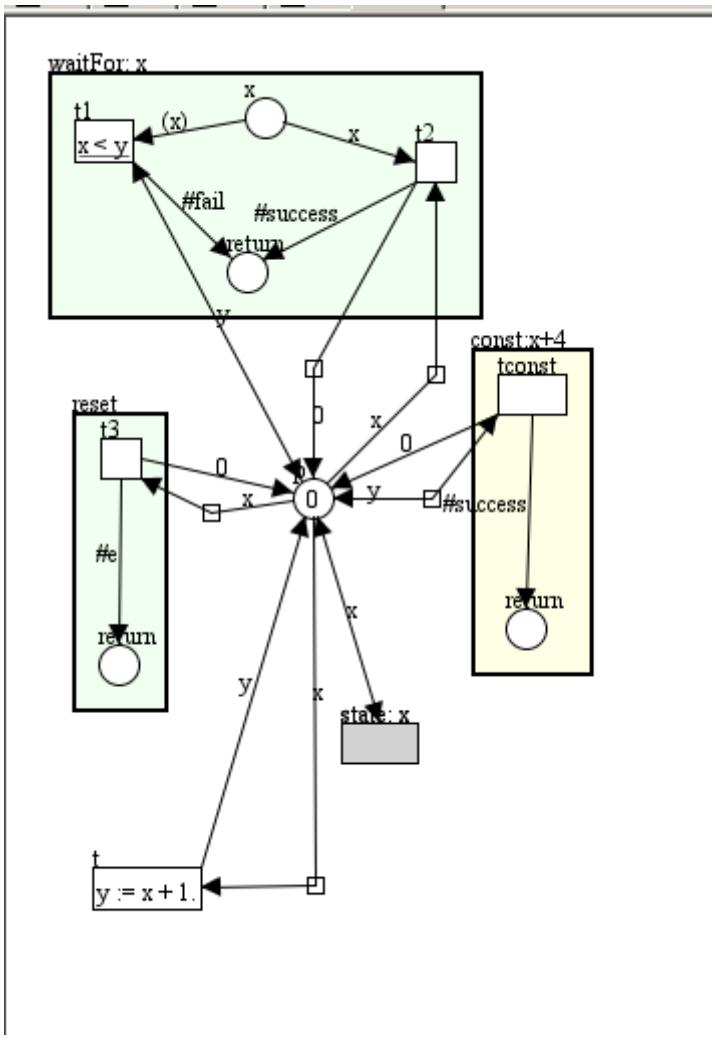
má päť fáz, pomocou ktorých sa prechádza cez jednotlivé zoznamy objektov a vykresľujú sa príslušné objekty.

V prvej fáze sa prejde zoznam oParentElement a vykreslia sa siete. Pritom sa berú farebné atribúty *MethodColor* alebo *ConstructorColor* pre pozadie a *PaintColor* pre text a okraje. Ak je sieť zdedená, tak vykreslenie textu a pozadia sa deje s atribútom *InheritPaintColor* a ak je sieť preťažená tak *OverridePaintColor*.

Ako ďalšie sa kreslia oPlace a potom oTrans objekty. Používaním farieb *PaintColor, InheritPaintColor,* alebo *OverridePaintColor* v závislosti na dedičnosti. Telo sa vykreslí buď s *DefaultFillColor* alebo s *InheritFillColor*, pričom aj pri preťažení sa použije *InheritFillColor*.

Ako ďalšie sa vykreslia oSync objekty. Na rozdiel od miest a prechodov, tu je výplň daná atribútmi *SyncFillColor* a *InheritSyncFillColor*.

Ako posledné sa vykreslia hrany a to sekvenciou čiar po celej ceste hranou, pričom okolo všetkých bodov lomu nakreslíme malé štvorčeky, aby sme sa dokázali lepšie orientovať pri výbere. Použité farby sú *PaintColor, InheritPaintColor* alebo *OverridePaintColor* podľa nastavenej dedičnosti.



1.2.8.9 Atribúty

Ako bolo naznačené ObjectContainer v sebe nesie aj hromadu grafických atribútov, ktoré slúžia pre vykreslenie. Tieto atribúty musíme nastaviť pomocou funkcií setterov a getterov. Pre každý atribút existuje pár týchto funkcií. Kompletný zoznam je nasledujúci:

- *setTextSize* a *getTextSize*
- *setDefaultSize* a *getDefaultSize*
- *setPaintColor* a *getPaintColor*
- *setMethodColor* a *getMethodColor*
- *setConstructorColor* a *getConstructorColor*
- *setSelectionColor* a *getSelectionColor*
- *setDragColor* a *getDragColor*
- *setCanvasBackground* a *getCanvasBackground*
- *setDefaultFillColor* a *getDefaultFillColor*
- *setInheritFillColor* a *getInheritFillColor*
- *setSyncFillColor* a *getSyncFillColor*
- *setInheritSyncFillColor* a *getInheritSyncFillColor*
- *setOverridePaintColor* a *getOverridePaintColor*
- *setInheritPaintColor* a *getInheritPaintColor*
- *getCanvasWidth* a *setCanvasWidth*
- *getCanvasHeight* a *setCanvasHeigth*

1.3 Hlavné triedy aplikácie

V tejto časti sa budeme venovať tým triedam, ktoré sú zodpovedné za konštrukciu hlavného okna a všetkých komponentov. Vyjasníme si, že inštancie tried, ktoré sa vytvoria pri spustení aplikácie, aby užívateľ mohol pracovať s aplikáciou.

1.3.1 Main.java

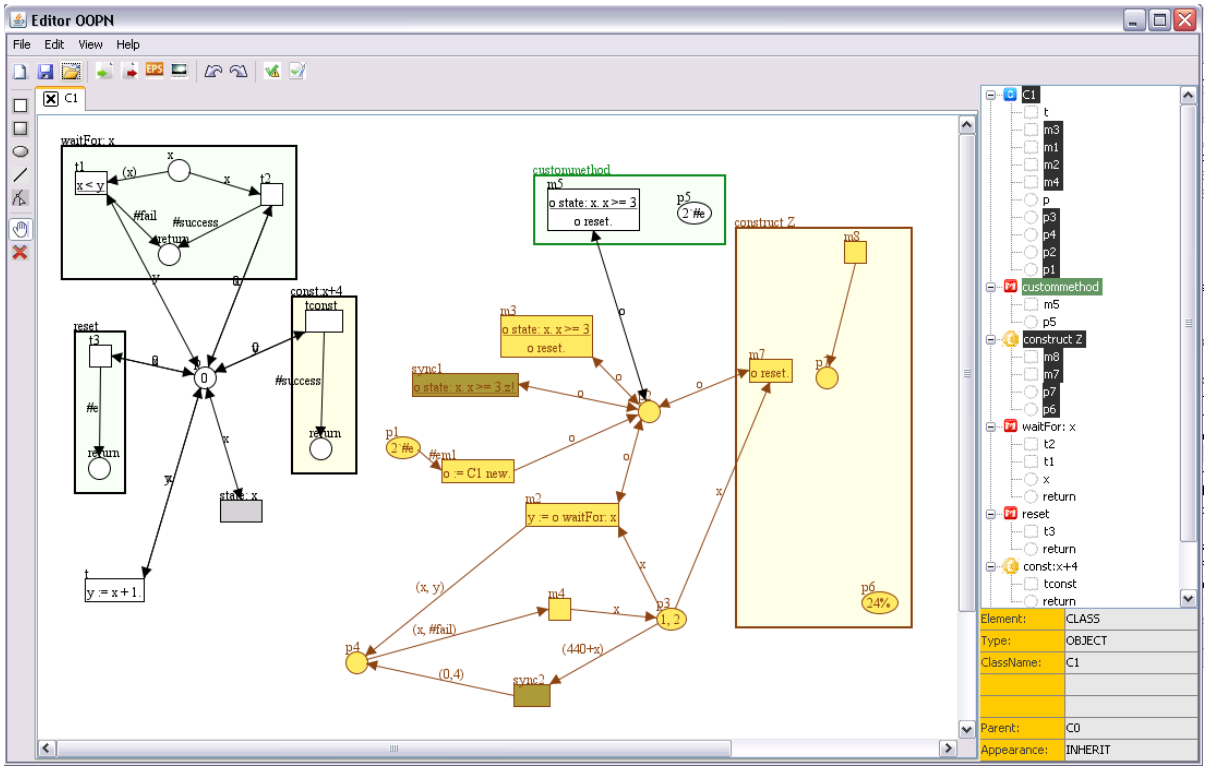
Z implementačného hľadiska sme potrebovali vstupný bod aplikácie. Tento vstupný bod je vždy funkcia main. Aj my sme teda implementovali triedu, ktorá nič iné nespraví, len predá riadenie triede MainFrame, teda konštruktoru okna.

1.3.2 MainFrame.java

Táto trieda je hlavná trieda aplikácie, akoby komponent, ktorý zodpovedá za to, aby všetko bolo dostupné a zorganizované. Pri volaní konštruktora, vytvorí JFrame aplikácie, a vytvorí inštancie tried ostatných komponentov. Referencie na tieto komponenty si zachováva v premenných a tak cez túto triedu sa dokážu vyhľadať odkazy na ostatné komponenty. Zoznam je nasledovný:

- Hlavné menu reprezentované triedou MainMenu.java
- Panely násrojov reprezentované triedou MyToolBar.java
- Panel hierarchií reprezentované triedou HierarchyManager.java.
- Okno pre zobrazenie zdrojového kódu reprezentované triedou ViewCodeWindow.java
- Trieda spracovania PNTalk formátu reprezentované triedou RawClassFileProcesser.java
- Trieda spracovania XML formátu reprezentované triedou XMLFileProcesser.java
- Okno nastavenia reprezentované triedou Settings.java
- Panel zo záložkami pre kresliace plochy reprezentované triedou JTabbedPaneWithCloseIcons.java.

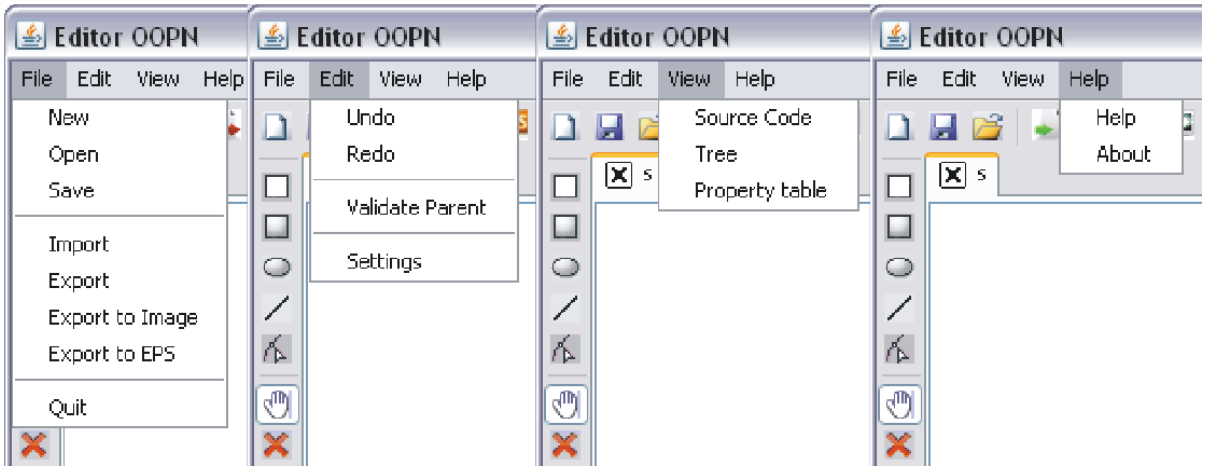
Táto trieda definuje aj vizuálne rozloženie komponentov v okne, teda takzvaný layout, podľa ktorého sa určí, že nasledujúce komponenty kam budú uložené.



Obrázok 10.1: Výsledná aplikácia.

1.3.3 MainMenu.java

Táto trieda slúži pre vytvorenie menu aplikácie. Menu uložíme na hornú časť okna a naplníme položkami podľa anglickej terminológie. Výsledok je tak nasledujúci:



Obrázok 10.2: Menu aplikácie.

Položky menu sú vložené tak, že každá položka v sebe definuje príkaz. Tento príkaz je vyvolaný výberom položky z menu. Príkazy sú oddelené na základe znaku. Pre položky platí, že napríklad výberom položky *New* sa spustí príkaz nadefinovaný na znak N, nakoľko príkaz *New* je spárovaný s ním. Obecne platí, že sa ako kľúčový znak použije začiatočné písmeno. Neplatí to v prípade *Export to Image*, kde je kľúčový znak X, v prípade *Export to EPS* kde kľúčový znak je Y, v prípade *Source Code*, kde kľúčový znak je K a v prípade *Settings*, kde kľúčový znak je G.

Príkazy v MainMenu korešpondujú s príkazmi na paneli nástrojov, teda v triede MyToolBar.java.

1.3.4 MyToolBar.java

Táto trieda v sebe zapuzdruje dve štandardné panely nástrojov JToolBar. Jeden je štandardný panel nástrojov, ktorý pomocou layout managera JFrame-u uložíme na pozíciu *Borderlayout.NORTH*. Zahrňuje položky po poradí, definované v návrhovej časti.

Obrázok 10.3: Štandardný panel nástrojov aplikácie.

Položky u realizované tlačidlami, teda komponentom JButton. Ku každej položke je pridaná ikona. Tieto ikony sú uložené ako zdroje priamo v aplikácii. Presne po poradí sú:

- Tlačidlo vytvorenia novej OOPN triedy
- Tlačidlo uloženia OOPN triedy do XML formátu
- Tlačidlo načítania existujúcej OOPN triedy z XML formátu
- Tlačidlo importu existujúcej OOPN triedy z PNTalk formátu
- Tlačidlo exportu OOPN triedy do PNTalk formátu
- Tlačidlo exportu grafického zobrazenia OOPN triedy do EPS formátu
- Tlačidlo exportu grafického zobrazenia OOPN triedy do obrázkového PNG formátu
- Tlačidlo prevedenia akcie späť
- Tlačidlo prevedenia akcie znovu
- Tlačidlo validácie zdedených prvkov OOPN triedy
- Tlačidlo vynútenej aktualizácie okna zdrojového kódu aktuálnej OOPN triedy

Všetky tieto tlačidlá majú definovanú vlastnú akciu, ktorá sa vykoná pri stlačení tlačidla.

Ako ďalší JToolBar je definovaný panel nástrojov na pravej strane. Používa rozloženie *Borderlayout.RIGHT*. Samotné rozloženie použije horizontálne rozloženie. Takže jednotlivo ako vkladáme tlačidlá sa pekne zaraďujú pod seba. Výsledok sú nástroje v komponentoch JToggleButton. Navyše implementujeme funkciu automatického prepínania, to znamená, že ak sa stlačí jedno tlačidlo, tak zostáva zapnuté, kým ostatné tlačidlá sa vypnú. Toto presne korešponduje s tým, že na kresliacej ploche aký nástroj je práve používaný.

- Nový Synchrónny port
- Nový Prechod
- Nové Miesto
- Nová hrana
- Nový hranový bod
- Presun objektu
- Vymazanie objektu

Obrázok 10.4: Panel nástrojov.

Pri kliknutí na tieto tlačidlá sa v aktuálnej kresliacej ploche nastavia.

1.3.5 ViewCodeWindow.java

Je to vlastne jedno samostatné okno, do ktorého sa vykresľuje text. Text berieme z funkcie, ktorá spraví výstup pre export do formátu PNtalk.

Trieda má jednu jedinú metódu a to aktualizácia textu vnútri textového poľa.

1.3.6 RawClassFileProcesser.java

V rámci celej aplikácii spravíme jednu triedu, ktorá bude zodpovedná za prevod z PNTalk formátu do nami definovaného formátu. Do tejto triedy implementujeme aj funkcie otvorenia súboru. Pôvodný koncept počíta s tým, že všetky triedy sú zvlášť uložené a tak musíme načítať jednotlivé textové znázornenia a až potom ich spracovať. Funkcia načítania zo súboru, tak nič iné nespraví, len otvorí súbor a načíta celý obsah do reťazca. Tento reťazec sa ďalej spracováva. Podobne pri exporte, sa vytvorí jeden reťazec s výslednou podobou a uloží sa do špecifikovaného súboru.

1.3.6.1 Import

Samotná funkcia importu je teda konverzia medzi PNTalk textom a nami definovaným interným zobrazením. Skladá sa z lexikálneho analyzátora, takzvaného pharsera, ktorý detekuje kľúčové slová. Samotná funkcia je popísaná v návrhovej časti .

V implementácii sme rozobrali najprv celý text na jednotlivé tokeny. Tieto tokeny, sú vlastne jednotlivé riadky súboru, ktoré sme upravili tak, že na jednom riadku sa vyskytuje vždy na prvom mieste kľúčové slovo. Takto sme eliminovali rozdelenie akcie, stráže alebo hranových výrazov po niekoľkých riadkov. Keďže máme na začiatku každého riadku kľúčové výrazy, tak pharser, veľmi ľahko prechádza cez celý text. Ak nájde výraz triedy, tak nastaví parametre triedy. Ak nájde kľúčové slovo pre niektorú sieť, prechod, miesto synchrónny port alebo hrany, tak sa vytvorí objekt a doplnia atribúty.

Výsledok je naplnené vnútorné zobrazenie ObjectContainer, pomocou ktorého môže aplikácia ďalej pracovať.

1.3.6.2 Export

Funkcia exportu taktiež kopíruje návrh. Prepíše vnútorné zobrazenie objektov na kód PNtalk. Pritom používa princíp, že kľúčový výraz definovaný je vždy na novom riadku s odstupom niekoľkých tabulátorových medzier, daný hierarchickým vnorením atribútu. Výsledok je tak pekne sformovaný text.

1.3.7 XMLFileProcesser.java

V implementácii XML procesoru nám pomôžu knižnice javax.xml štandardné buildery. Použijeme objekty typu Document , DocumentBuilderFactory, DocumentBuilder, DOMImplementation, TransformerFactory a Transformer. Pomocou týchto objektov, dokážeme vytvoriť vlastnú kostru XML súboru, do ktorého dokážeme potom jednotlivé elementy a uzly pridávať.

1.3.7.1 Zapísanie

Element značí značku XML ktorá zaväzuje na nejaký uzol, teda hlavný element. Celý dokument má ako najvyšší uzol <xml>. Do tohto uzla sa pridávajú jednotlivé Elementy podľa špecifikácie v návrhovej časti. Do elementov sa pridávajú textové uzly, zvané TextNode. Tieto textové uzli nesú hodnotu, ktorá má byť zapísaná do XML súboru. Po dokončení zápisu sa potom XML formát prevedie na stream znakov, a zapíše do súboru.

1.3.7.2 Načítanie

Pri načítaní používame taktiež podobný princíp. Musíme však vytvárať jednotlivé objekty pomocou špeciálneho konštruktora, ktorý explicitne deklaruje XML identifikátor pre všetky objekty.

Tento postup je nutný kvôli tomu, aby sa zachovali referencie na objekty, a tak konzistencia celej OOPN siete.

1.3.8 HierarchyManager.java

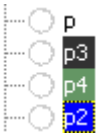
Trieda HierarchyManager je vlastne jednotná trieda pre celú pravú stranu aplikácie. To znamená, že je tu zhrnutá hierarchia aj tabuľka vlastností. Samozrejme obe komponenty majú vlastné triedy, ale patria z logického hľadiska k tejto triede. Konštruktor triedy vyvolá a nastaví základné vlastnosti oboch tried ako aj výzor. Ďalej tu máme funkciu *init*, ktorá resetuje tabuľku a nastaví aj strom hierarchie. V podkapitolách rozoberieme obe komponenty tejto triedy.

1.3.8.1 JTree

Tento objekt je vytvorený na základe TreeModelu. Dopredu máme definované, že pre aké komponenty ako sa renderuje strom. Renderovanie spočíva v triede *MyJTree.java* súboru. Je to externý súbor, ktorým preťažíme javax.swing.JTree triedu. Hlavné zmeny sú napríklad vytvorenie ikony pre všetky typy objektov ako objektová sieť, synchrónny port, metóda, konštruktor prechod a miesto.:

Obrázok 10.5: Značenie obrázkov jednotlivých OOPN elementov v strome hierarchie.

Ďalšie možnosti renderovania je vykreslenie textu v strome hierarchii. Rozlišujeme štyri typy. Prvý typ je, ak sa jedná o natívny objekt, vtedy nastavíme renderovanie ako čierny text na bielom podklade. Druhý typ je zdedený objekt, ktorý značíme ako biely text na čiernom pozadí. Textovú časť preťaženého objektu implementujeme ako zelené pozadie a biely text, a vybraný objekt je znázornený modrým pozadím.

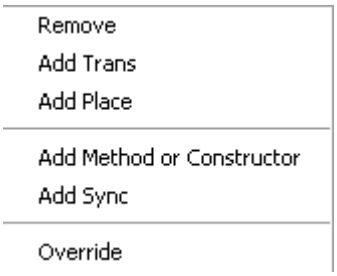


Obrázok 10.6: Značenie typov dedičností v v strome hierarchie.

Ako ďalšia funkcia celkového stromu je interaktívna práca s objektmi. Musíme preto implementovať menu systém z návrhovej časti. Toto menu v sebe nesie v sebe komplexnú funkcionalitu. Pre vytvorenie bol použitý podobný systém ako pri vytvorení položky hlavného menu avšak hlavný komponent je JPopupMenu. Samotné menu má šesť položiek pomenovaných:

- Remove – volá removeIObjects funkciu v aktuálnom ObjectContainer-y
- Add Trans – volá dialógové okno pridania nového prechodu
- Add Place – volá dialógové okno pridania nového miesta
- Add Method or Constructor – volá dialógové okno pridania novej siete

- Add Sync – volá dialógové okno pridania nového synchrónneho portu
- Override – volá Override funkciu v aktuálnom ObjectContainery



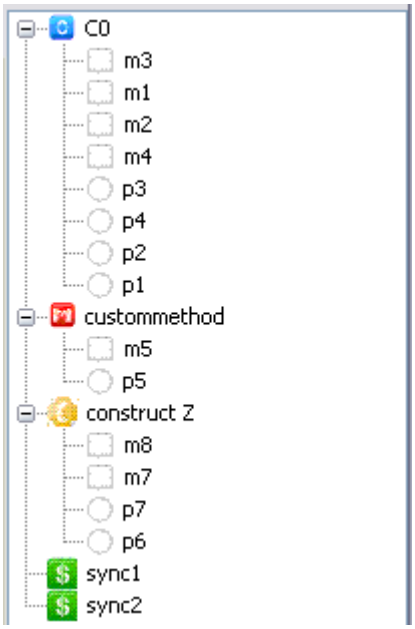
Obrázok 10.7: Menu ľavého tlačidla v strome hierarchie.

Ďalej potrebujeme metódu, ktoré dynamicky prepíšu obsah celého stromu podľa aktuálneho ObjectContaineru. Táto metóda je *fillAllObjects* v samotnom HierarchyManager.java triede, ktorá následne naplní strom OOPN objektmi.

Ako ďalšiu metódu potrebujeme dynamicky meniť práve vybraný objekt. K tomu skonštruujeme metódu, ktorá jednak nastaví vybraný objekt aj v tabuľke aj v strome. Menuje sa *setNodeAndTableToSelect* a nastaví vizuálne pozadie pre aktuálne vybraný objekt.

Ďalej potrebujeme aj metódu, ktorá nastaví v ostatných komponentoch, ako na kresliacej ploche, tak aj v tabuľke vlastností, ak užívateľ interaktívne vyberie objekt v strome. K tomuto slúži funkcia *paintSelectionToCanva*, ktorá použije vizuálne vykreslenie.

Ako posledné metódy slúžia pre nastavenie viditeľnosti celkového stromu. Preto implementujeme funkciu *isTreeVisible* a *setTreeVisible* metódy, pracujúce s boolean parametrami. Prvá metóda vracia aktuálny stav viditeľnosti, a druhá mení tento stav.



Obrázok 10.8: Strom hierarchii.

Strom sa musí vysporiadať aj zo základnými operáciami, ako pridanie alebo vymazanie OOPN objektov. Pridanie môžeme jednoducho skonštruovať funkciami *add*, ktoré pridávajú nové uzly do existujúcej štruktúry. Pri mazaní je to o trochu ťažšie. Je lepšie prekresliť celý strom, nakoľko mazanie môže vždy so sebou niesť odstránenie viacerých objektov.

1.3.8.2 JTable

JTable reprezentuje tabuľku vlastností. Jej renderovanie je podobne ako funkcionalita veľmi dynamická. Použijeme pár, meno atribútu v prvom stĺpci a hodnotu atribútu v druhom stĺpci. Pozadie stĺpcov vyfarbíme rôznymi farbami. Pritom povolíme editáciu len pre niektoré atribúty.

Element:	m5
Type:	TRANS
Name:	m5
Guard:	o state: x. x >= 3
Action:	o reset.
Parent:	custommethod
Appearance:	NATIV

Obrázok 10.9: Tabuľka vlastností.

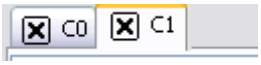
Keďže teoreticky pri každom kliknutí, teda výbere nových objektov sa musí prekresliť celková tabuľka vlastností. Základné renderovanie je dané triedou *MyColorRenderer*. Táto trieda je súčasťou triedy HierarchyManager, dokáže tak pristúpiť k premenným potrebným pre renderovanie. V rámci tejto triedy sa nastavia farby pozadia jednotlivých buniek tabuľky.

Musíme vytvoriť metódu, ktorá bude zodpovedná za naplnenie atribútov objektov do riadku tabuľky na základe výberu. Táto metóda sa bude menovať *setTableToSelect* a má parameter OOPN objekt.

Ako ďalšiu vnorenú triedu vytvoríme *MyEditor*, ktorý slúži pre editáciu parametrov. Samotné riadky tabuľky nie sú len pasívne bunky, ale slúžia pre zmenu jednotlivých atribútov. Takto dokážeme napríklad zmeniť názov objektu, stráž, hranový výraz alebo akciu. *MyEditor* trieda preťažuje DefaultCellEditor triedu, a tak nám umožní aby sme dokázali rozšíriť všetky jej editačné funkcie. Editované atribúty tak priamo môžeme aplikovať na OOPN objekty.

1.3.9 JTabbedPaneWithCloseIcons.java

Nakoľko aplikácia dokáže naraz pracovať s niekoľkými triedami, potrebujeme vytvoriť jednotný komponent, ktorý dokáže uchovať a umožniť prepínanie medzi jednotlivými kresliacimi plochami. K tomu by nám poslúžil aj komponent *JTabbedPane*. Tento komponent však musíme doladiť niekoľkými drobnosťami a preto vytvoríme novú triedu rozširujúcu *JTabbedPane.*



Obrázok 10.11: Ukážka záložiek jednotlivých tried.

Ako prvú zmenu spravíme možnosť uzatvorenia okien s ikonou X na záložkách Toto bude vlastná trieda rozširujúca triedu *Icon*. Vykreslí sa 16x16 pixlová plocha, ktorá znázorní X a tá reaguje na kliknutie myši. Vyvolá tak uzatvorenie otvorenej OOPN triedy. Pri volaní tejto metódy sa kontroluje, či sa nejedná o poslednú triedu. Ak už nie je viac tried, tak sa nastavia niektoré tlačidlá na *MyToolBar*-e a v *MainMenu* ako *disabled*.

Ďalšie metódy sú metóda *addTab*, ktorá dokáže pridať novú záložku, teda novú kresliacu plochu. Ak sa jedná o prvú záložku, tak musíme nastaviť všetky editačné tlačidlá v *MyToolBar*-e a v *MainMenu* ako *enabled.*

1.3.10 DrawPanel.java

DrawPanel je vlastne ďalšia z najdôležitejších komponentov aplikácie. Je zodpovedné za dočasné prekresľovanie a následne pre uloženie zmeny grafickej pozície objektov. Trieda v sebe implementuje metódy pre nastavenie nástrojov, a metódy pre spracovanie udalostí myši.

Ako prvá z metód je *initialize*. Táto metóda slúži nato, aby sme vkreslili ObjectContainer na obrazovku, ďalej nastavili hodnoty v paneli hierarchií. Nastaví vnútorné premenné, referencie, aby bolo možné pracovať ďalej s grafickou reprezentáciou OOPN triedy.

Potrebujeme získať ďalej základné mierky objektov. Tieto hodnoty vždy má k dispozícii ObjectContainer, takže z nej musíme získať tieto hodnoty. Pre vynútenie tejto operácii však implementujeme aj metódu *updateDefaultSizesFromObjContainer*, ktorú použijeme ak sa mení nastavenie.

Potrebujeme taktiež metódu, ktorá vynútene prekreslí všetky OOPN objekty podľa aktuálnej reprezentácie ObjectContainer-u. Túto metódu nazveme *RefreshPanel*

Ako ďalšiu metódu, potrebujeme vždy určiť, že aký je nastavený pracovný nástroj v rámci DrawPanel-u. K tomuto slúžia funkcie *setTool* a *getTool.*

Ako bolo spomenuté, hlavné funkcie prevádzajú manipulácie s myšou. Preto najdôležitejšie funkcie sú *mousePressed*, *mouseDragged* a *mouseReleased*, ktoré si rozoberieme v podkapitolách.

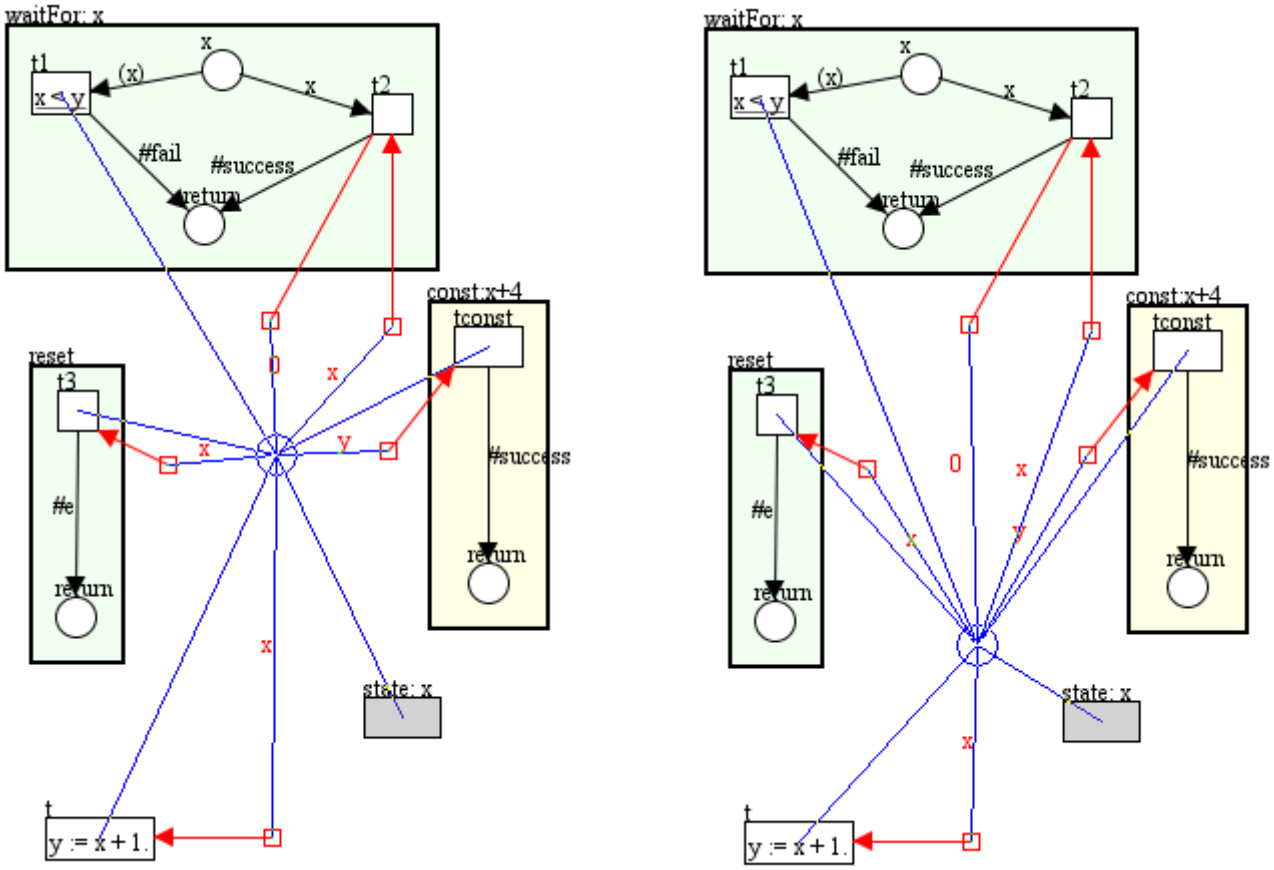
1.3.10.1 mousePressed

Táto metóda sa volá ak užívateľ klikne tlačidlom myši na kresliacu plochu. Vykoná sa rada metód. Najprv sa volá metóda odstránenia výberu v ObjectContainery s názvom *removeAllSelection*. Následne sa poznačia súradnice kliknutia a volá sa funkcia *checkForExistingObject*, ktorá vyhľadá objekt na danej súradnici ak je prítomný. Tento objekt sa poznačí. Ako ďalšie sa použije vykresľovacia funkcia. Ak je vybraný nástroj výberu, a súčasne je aj vybraný nejaký objekt, tak sa spraví výnimka objektu. To znamená, že sa odovzdá funkcii aktuálne vybraný objekt, ktorý sa nemá permanentne vykresliť. Následne sa získa zoznam bodov, s ktorými by sa mal objekt spojiť. Ako posledná sa volá metóda *painting*, ktorá využíva atribút *draggedColor* a vykreslí aktuálny element.

Samotná metóda *painting*, je komplexná metóda a využíva mnoho funkcií v závislosti na vybranom nástroji. Ak je vybraný nástroj nového synchrónneho portu, tak sa volá metóda *drawSync*, ktorá vykreslí obrys synchrónneho portu. Ak je nástroj nového prechodu, tak sa volá metóda *drawTrans*, ktorá vykreslí obrys nového prechodu. Keď máme vybrané nové miesto, tak sa vykreslí *drawPlace*, ktorá vykreslí elipsový obrys miesta. Keď je vybraný nástroj novej hrany vtedy sa vykreslí nová čiara metódou *drawLine*. Nástroj nového hranového bodu je už pomerne komplexnejší nástroj. Využíva vykreslenie čiar so zoznamom bodov, vrátených vykreslovacou funkciou ObjectContaineru. V tomto prípade sa jedná o dva segmenty hrany, spojené s aktuálnou pozíciou myši. Ako posledný nástroj je nástroj presunu objektu, ktorý jednak vykreslí obrys objektu zvoleného metódou *checkForExistingObject* a spojí tieto objekty so zoznamom bodov, ktorý vracia metóda permanentného vykreslenia *RepaintByObjectContainer*.

1.3.10.2 mouseDragged

Metódu *painting* používa aj metóda mouseDragged. Jedná sa o metódu, ktorá sa volá pri ťahaní myšou. PaintMode grafického kontextu je pritom nastavený na XORMode. To znamená, že pri každej zmene sa prepíše kresliaca plocha, čím docielime, že vykreslenie sa neuchováva permanentne.



Obrázok 10.12: Ukážka premiestnenia objektu.

1.3.10.3 mouseReleased

Pri tejto metóde sa vlastne ukončuje dočasné vykresľovanie a je čas vyhodnotiť dopady užívateľskej akcie na celkovú sieť. Zmeny sú rozdelené taktiež podľa vybraného nástroja. Ako prvé potrebujeme detekovať, či sme sa nedostali mimo kresliacu plochu. Ak sme mimo plochy, tak sa neprevedú zmeny len prekreslí plocha. Inak voláme *processing* metódu, ktorá vykoná zmeny.

Metóda *processing* je rozdelená podľa nastaveného nástroja.

- PLACE – zobrazí sa NewoPlaceDialog a následne sa podľa vrátenej hodnoty vytvorí nový objekt OOPN miesta a uloží do ObjectContainer-u
- TRANS - zobrazí sa NewoTransDialog a následne sa podľa vrátenej hodnoty vytvorí nový objekt OOPN prechodu a uloží do ObjectContainer-u
- SYNC - zobrazí sa NewoSyncDialog a následne sa podľa vrátenej hodnoty vytvorí nový objekt OOPN synchrónneho portu a uloží do ObjectContainer-u
- LINE – najprv sa skontroluje či na pozície pri kliknutí a pozícii pri upustení nachádza objekt. K tomu použijeme funkciu *isBoundsConnectedWithObjects*. Táto funkcia jednak kontroluje výskyt objektov a skontroluje aj či sa jedná o objekty ktoré môžeme spojiť. Následne sa zobrazí NewoBoundDialog ktorí podľa zadaných parametrov vytvorí nový objekt OOPN hrany a uloží do ObjectContainer-u
- LINEINTERSECT – podmienkou je, aby pri *mousePressed* metóde bola vybraná hrana alebo bod hrany. Ak bola vybraná hrana aj bod hrany, tak sa bod hrany presunie na novú pozíciu.

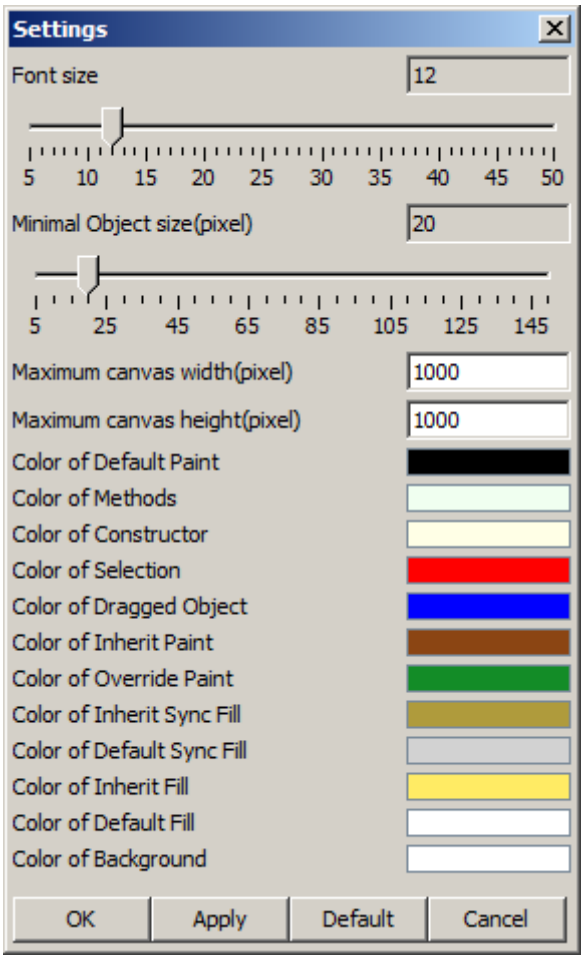
Ak bola vybraná len hrana, tak sa vytvorí nový bod hrany a zapíše do zoznamu bodov hrany.

- SELECT – podmienkou je pri tejto funkcii, aby bol vybraný OOPN objekt miesta, prechodu alebo synchrónneho portu poprípade bod hrany. Pri prevedení tejto funkcii sa presunie objekt na nové súradnice, čo sa poznačí aj v ObjectContainery.
- DELETE – podmienkou je znova vybranie OOPN objektu miesta, prechodu, synchrónneho portu alebo samotnej hrany, poprípade bodu hrany pri *mousePressed* metóde. V tomto prípade sa odstráni z ObjectContainer-u zvolený objekt.

Po prevedený jednej z týchto operácií sa samotná kresliaca plocha prekreslí permanentným kreslením volaním metódy ObjectContainer *RepaintByObjectContainer*.

1.3.11 Settings.java

Trieda nastavenia je aktívny meniaca sa trieda. Načíta vždy aktuálne nastavenia z práve aktívnej záložky. Ak nie je v aplikácii načítaná žiadna kresliaca plocha, tak sa nastavenie nezobrazí. Trieda Settings v sebe implementuje JFrame a niekoľko možností uvedené v návrhu.



Obrázok 10.13 Panel nastavenia.

Trieda pracuje s poľom premenných, ktoré uchovávajú nastavenia. Celkovo sa jedná o dve polia, jeden pre číselné hodnoty a druhé pre hodnoty farieb. Tieto polia sa vyskytujú duplicitne. Do prvého poľa sa načíta aktuálne nastavenie pri zobrazení okna. Tieto polia sa menujú

- *initialValues*
- *initialColors*

Ďalšie polia uchovajú hodnoty, ktoré si užívateľ nastavuje. Tieto polia nesú názov

- *storedValues*
- *storedColors*

Vytvoríme radu funkcií, ktoré pretransformujú údaje z užívateľského rozhrania do vnútorných polí. Ako prvá funkcia je zobrazenie okna nastavenia. To sa deje metódou *showSettingsWindow*. Táto metóda najprv naplní interné polia pomocou metódy *getSettingsFromObjC*. Ako ďalší krok sa zapíšu tieto hodnoty do komponentov pomocou metódy *writeValuesToVisualComponents*.

Interaktívna práca užívateľa ako zmena jednotlivý atribútov sa prejaví až po volaní funkcie *readValuesFromVisualComponents*. Táto funkcia naplní polia *storedValues* a *storedColor* hodnotami aby korešpondoval s aktuálne nastavenými. Ďalej volaním funkcie *setNewSettingsToObjC* sa aplikujú tieto funkcie do ObjectContainer-u.

Užívateľ dokáže ešte zvoliť funkciu Default stlačením príslušného tlačidla. Vtedy sa polia *storedValue* a *storedColors* prepíšu metódou *setDefaultValues*, na dopredu definované hodnoty uložené priamo ako konštanty triedy. Potom potrebujeme ešte zapísať tieto konštanty do komponentov pomocou metódy *writeValuesToVisualComponents*.

Ako ďalšie tlačidlo je Cancel, teda vrátenie nastavení do pôvodného formátu. K tomu nám slúžia polia *initialValues* a *initialColors*. Tieto nastavenia sa aplikujú metódou *setInitialSettingsToObjC*.

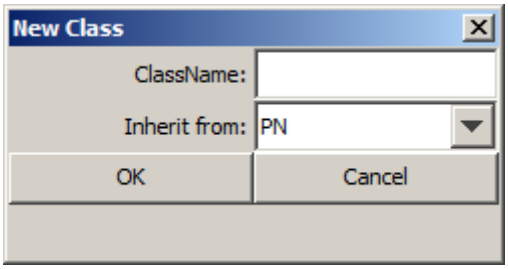
1.4 Dialógové okná

Dialógové okná slúžia pre definovanie atribútov nových objektov. Jedná sa o OOPN komponenty. Tieto triedy vytvárajú dialógové okná, ktoré sa zobrazia v popredí s tým, že kým nie sú zatvorené, tak celá aplikácia je blokovaná. To zabezpečuje atribút *ModalityType.APPLICATION_MODAL*.

Funkcionalita pozostáva z konštruktoru objektu, ktorý vytvorí okno podľa požadovaných parametrov. Všetky dialógové okná implementujú metódy *isOK*, ktorá určí či bolo zvolené OK a má sa teda previesť operácia vytvorenia nového OOPN objektu alebo Cancel, ktorá naznačuje, že sa nemá nič pridávať. Táto funkcia sa vždy kontroluje vo volanej časti.

1.4.1 NewClassDialog.java

Táto trieda vytvorí dialógové okno, ktoré sa volá ak užívateľ chce vytvoriť novú triedu. Pozostáva z niekoľkých komponentov, ako meno triedy, meno nadradenej triedy a tlačidlá OK a Cancel. JComboBox implementujúci „Inherit from“ naplní jednotlivé položky aktuálne uloženými záložkami. To znamená, že máme k dispozícii priamo ObjectContainer-y, ktoré dokážeme klonovať a poskytnúť základ novo vytvorenej triede.

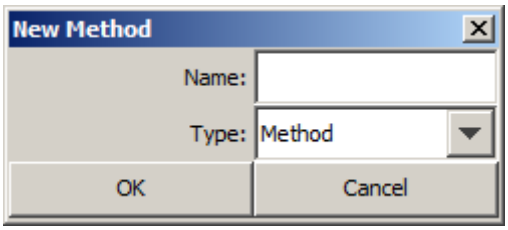


Obrázok 10.14: Okno vytvorenia novej OOPN triedy.

Ak bolo zvolené OK, tak sa najprv získa funkciou *getNameString* meno novej triedy. Metódou *getParentObjectContainer* sa zas získa OOPN trieda, ktorá má slúžiť ako nadradená trieda aktuálnej triedy.

1.4.2 NewoParentElementDialog.java

Táto trieda slúži na vytvorenie novej siete. Podporuje dve typy sietí, a to sieť OOPN konštruktoru a OOPN metódy. Tieto dva typy sú vložené do JComboBox-u Type.

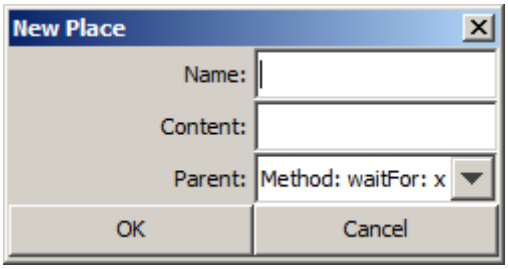


Obrázok 10.15: Okno vytvorenia novej OOPN siete.

Funkcionalita je daná tým, že ak sa zvolí OK tlačidlo, tak metódou *getNameString* sa vyžiada meno novej siete. Následne metóda *getType* vráti typ siete z položiek *MyObjectNames*.

1.4.3 NewoPlaceDialog.java

Trieda slúži pre dialógové okno vytvorenia nového OOPN miesta. Má tri atribúty, a to meno, počiatočný výraz zvaný Content a atribút Parent, ktorý je nadradená sieť do ktorej prechod má spadať. JComboBox je naplnený aktuálne definovanými sieťami.

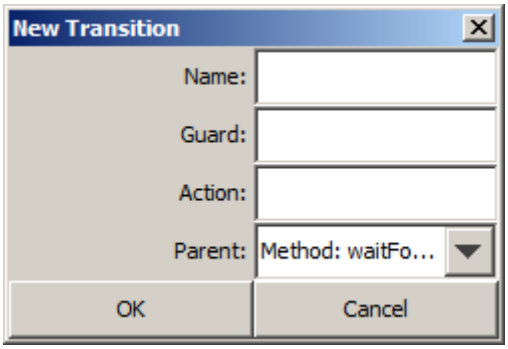


Obrázok 10.16: Okno vytvorenia nového OOPN miesta.

Po stlačení tlačidla OK môžeme využiť metódu *getNameString*, pomocou ktorej dostávame zadané meno. Metódou *getContentString* dostávame počiatočný výraz definovaný užívateľom a metóda *getParentElement* vracia sieť teda oParentElement.

1.4.4 NewoTransDialog.java

Táto trieda vytvára dialógové okno pre vytvorenie nového prechodu. Prechod má najviac atribútov a to Meno objektu, stráž, akcia a sieť. Tieto atribúty sú znázornené pomocou anglickej terminológie.



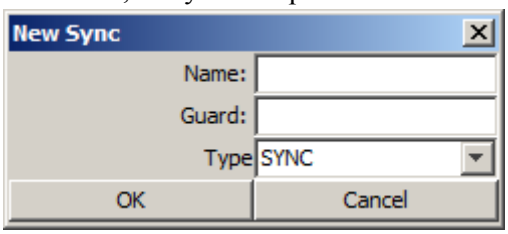
Obrázok 10.17: Okno vytvorenia nového OOPN prechodu.

Po vyplnení Dialógového boxu a stlačení OK dokážeme nasledujúcimi metódami získať jednotlivé atribúty:

- *getNameString* – získame meno prechodu
- *getGuardString* – získame stráž
- *getActionString* – získame akciu
- *getParentElement* – získame sieť do ktorej spadá prechod

1.4.5 NewoSyncDialog.java

Trieda pre vytvorenie dialógového okna nového synchrónneho portu. Objekt OOPN synchrónneho portu má atribút mena a stráže. Typ bol doplnený kvôli rozšíreniu, a to že syntax synchrónneho portu môže využívať i OOPN objekt inhibítor, ktorý však v pôvodnom návrhu OOPN neexistuje.



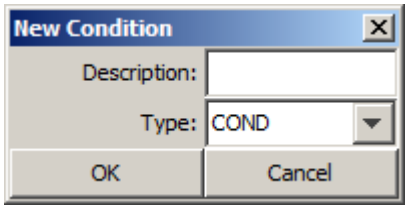
Obrázok 10.18: Okno vytvorenia nového OOPN synchrónneho portu.

Metódy pre získanie jednotlivých atribútov sú nasledujúce:

- *getNameString* – získame meno prechodu
- *getGuardString* – získame stráž
- *getType* – získame typ definovaný triedou MyObjectNames

1.4.6 NewoBoundDialog.java

Trieda slúži pre vytvorenie novej hrany. Nakoľko spojené objekty sa definujú explicitne priamo vytvorením objektu, preto musíme špecifikovať len atribúty hranového výrazu a typu objektu.



Obrázok 10.19: Okno vytvorenia novej OOPN hrany.

Po stlačení OK dokážeme pomocou nasledujúcich metód vybrať jednotlivé atribúty.
- *getDescriptionString*– získame hranový výraz.
- *getType* – získame typ definovaný triedou MyObjectNames