VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO PODPORU VÝVOJE SOFTWAROVÝCH SYSTÉMŮ

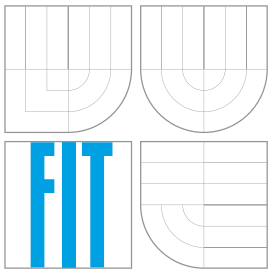
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

KAREL HALA

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO PODPORU VÝVOJE SOFTWAROVÝCH SYSTÉMŮ

TOOL FOR SOFTWARE SYSTEMS DESIGN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — KAREL HALA
AUTHOR

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

BRNO 2014

Abstrakt

Nástroj pro podporu vývoje softwarových systému je aplikace sloužící pro vizuální znázornění průběhu vývoje aplikace. Slouží k porozumění požadavků od zákazníka, tyto požadavky převádí do objektového návrhu s možností popisu chování tříd pomocí objektově orientovaných Petriho sítí. V této práci se postupně popíší nástroje věnující se tomuto tématu, vyberou se jednotlivé diagramy a popíše se způsob implementace těchto diagramů. V závěru budou představeny možná rozšíření.

Abstract

The Tool for software systems design is an application for visualization of application development. It's main goal is to achieve connection between developer and customer. It should be used to understand customer's needs, prepare workflow of project and behaviour of each class and method using objected oriented Petri nets. In this work we will look on other programs, that focus on similiar topic, we then pick some diagrams and describe how they were implemented. At the end we will disguss possible extension for this tool.

Klíčová slova

UML diagramy, Propojení UML diagramů, Případ užití, Diagram tříd, Nástroj, Softwarový vývoj, Objektově orientované Petriho sítě

Keywords

UML diagrams, Connection of UML diagrams, Use Case, Class diagram, Tool, Software development, Objected oriented Petri nets

Citace

Karel Hala: Nástroj pro podporu vývoje softwarových systémů, bakalářská práce, Brno, FIT VUT v Brně, 2014

Nástroj pro podporu vývoje softwarových systémů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Kočího, Ph.D.

.....
Karel Hala
7. května 2014

Poděkování

Předem bych rád poděkoval panu Ing. Radku Kočímu, Ph.D. za odbornou pomoc a vysvětlení propojení mezi jednotlivými částmi aplikace.

Obsah

Kapitola 1

Úvod

Při implementaci jakéholiv nástroje se často naráží na bariéru mezi zákazníkem a programátorem. Zákazník často netuší, co chce a programátor často zákazníka špatně pochopí. Tento nástroj bohužel tyto problémy nedokáže odstranit, nicméně by měl napomoci odstranit některé bariéry a zajistit snažší vizualizaci problémů.

Tento nástroj umožní větší průhlednost vývoje aplikací tím, že zaručí postupnou implementaci a vytváření jednotlivých částí. Tento nástroj dále zajistí popis chování tříd přes objektově orientované Petriho sítí, což vede k urychlení návrhu nově vytvářené aplikace a možnosti průběžného testování aplikace bez nutnosti vytvářet a spouštět zdrojový kód.

Tento editor by měl být jednoduchý na používání a snadný k pochopení. Převážně je potřeba zajistit automatické vytváření a editace při návrhu, kdy je nutné obstarat vytvoření diagramu tříd na základě diagramu případů užití a na diagram tříd také navázat objektově orientované Petriho sítě.

V kapitole 2 se nejdříve podíváme na nástroje, které se zaměřují na podobné téma jako je tomu ve specifikaci pro tento nástroj. Tyto nástroje dokážou vizualizovat vyvíjený software pomocí různých grafů a diagramů. Rozebereme jejich přednosti a jejich slabiny, poté si vybemere některé jejich vlastnosti a vysvětlíme případné řešení nedostatků těchto aplikací.

Navážeme kapitolou 3, kde vysvětlíme druhy grafů, které se hodí pro implementaci námi zvolených částí. Zaměříme se na trojici diagramů *Diagram případů užití*, *Diagram tříd* a *Objektově orientované Petriho sítě*. Provázání mezi těmito diagramy poté vysvětlíme v kapitole 3.5.

V následující kapitole přestavíme nástroj, který byl předmětem této práce. Předvedeme jeho silné stránky, jakým způsobem byly vytvořeny určité části aplikace a také mírně nastíníme ovládací prvky tohoto nástroje 4. Na toto volně navážeme představením rozšířených možností tohoto nástroje v kapitole 5.

Po vysvětlení nástroje představíme jeho použitelnost v praxi prezentací několika možným budoucím uživatelům a jejich reakcí na tento nástroj. Popíšeme jejich chování a pocity z aplikace 6.

Závěrem práce bude představení možných dalších rozšíření tohoto nástroje. Také rozebemere momentální stav této aplikace, kde se zaměříme na přínos nástroje a zhodnotíme, zda je nástroj schopný udržet se v silné konkurenci různých editorů UML diagramů 7.

Kapitola 2

Nástroje zaměřující se na podobné téma

Z hlediska snažšího porozumění vyvíjené aplikace bylo zvoleno nejdříve prozkoumámí nástrojů, které se specifikují na podobné téma. Byly vybrány nástroje, co jsou jednoduché na používání a hojně využívané v praxi. Z tohoto důvodu byly prozkoumány dva nástroje – *Umbrello* a komerční *Enterprise Architect*. Detailnější popis jednotlivých aplikací, jejich výhod a nedostatků, je popsán v sekci 2.1 a 2.2. Dále se zaměříme na využitelné součásti těchto nástrojů v sekci 2.3.

2.1 Umbrello

Umbrello, celým názvem *Umbrello UML Modeller* [1], je aplikace primárně určena pro Unix systémy, nicméně je možné ji bez problémů nainstalovat také na Windows za pomoci KDE installeru a vybrání požadovaného balíčku [2].

2.1.1 Motivace

Aplikace Umbrello je hojně využívána a chválena mezi uživateli, z toho důvodu byl tento nástroj vybrán jako první k prozkoumání. Také jednoduché používání této aplikace bylo přínosné pro analýzu požadavků a porozumění těmto nástrojům.

2.1.2 Popis aplikace

Aplikace je robustní a nabízí široké možnosti úprav. Nástroj je volně ke stažení a používání [3]. Umbrello poskytuje mnoho UML grafů a základní práce s touto aplikací je poměrně snadná, ale její jednoduchý design má několik slabin. Pro menší aplikace a pro začínající firmy je naprosto dokonalá, ale absence pokročilého editoru a ne příliš jednoduchá práce s editorem (jako například přidání nové metody, zalomení čar a také vytvoření jiného souboru) je poněkud omezující. V další kapitole se tedy podíváme na výhody této aplikace a poučíme se z případných nedostatků.

[1] Domovská stránka aplikace: http://umbrello.kde.org/

[2] Ovládání tohoto installeru je velice jednoduché a intuitivní. Lze jej nalézt na: http://windows.kde.org/download.php.

[3] Aplikace používá licenci GNU general public license, což umožňuje zapojení komunity do vývoje tohoto nástroje. Odkaz na licenci: http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

2.1.3 Výhody a nevýhody

Umbrello aplikace poskytuje několik možností, jak popsat vývoj a návrh aplikace. Nabízí převážně grafy a diagramy ze skupiny UML mezi které patří:

- *ClassDiagram* pro popis tříd a provázanosti mezi nimi.
- *Sequence Diagram* pro popis chodu aplikace, komunikace mezi vlákny a komunikace mezi aplikacemi.
- *Use Case Diagram* k popisu akcí v systému a aplikaci.
- *State Diagram* pro popis stavu systému a průběhu jednotlivých operací, převážně zachování v určitých momentech aplikace na dané podmínky.
- *Activity Diagram* popisuje akce a průběh celého programu graficky.
- *Component Diagram* pro popis spojení a komunikace mezi jednotlivými komponentami.
- *Deployment Diagram* popisuje fyzické uložiště a zařízení, na kterých program bude pracovat.
- *Entity Relationship Diagram* pro popis vztahu dat mezi sebou.

Mezi klady této aplikace patří větší výběr diagramů jak znázornit a zpřehlednit vývoj aplikace. Základní ovládání aplikace je intuitivní a design je přehledný. Jako hlavní výhodu této aplikace lze uvést možnost vytvoření zdrojových kódů na základě vytvořených diagramů a grafů. Je možné také provést vytvoření a úpravu zdrojových kódů pomocí nástroje Umbrello. Stačí pouze importovat jednu již vytvořenou třídu a Umbrello uživatele provede jednoduchým nastavením a výběrem, které třídy namodelované v aplikaci, se mají vygenerovat a nabídne také možnost mírně upravit zvolený kód. Vzhledem k tomu, že aplikace je plně zdarma, je tato vlastnost velice přínosná.

Umbrello dále poskytuje možnost částečného vytvoření dokumentace i v případě, že nejsou poskytnuty komentáře, na základě kterých by tato dokumentace mohla být vytvořena. Při generování tříd je možné také importovat zdrojový kód a nechat Umbrello vše vygenerovat samo. Tato vlastnost je podporována pouze pro jazyky `ActionScript, Ada, C++, C#, D IDL, Java, Javascript, MySQL`.

Mezi další výhody patří možnost exportování objektů jako PNG obrázky. Při psaní dokumentů je možné vytvořený diagram zkopírovat a vložit přímo do dokumentu. Pro export do formátu PNG je možné vybrat část diagramu, nebo celou síť. S touto možností také přichází možnost tisku celého diagramu. [2]

2.1.4 Zhodnocení aplikace Umbrello

Základní práce s aplikací je jednoduchá a intuitivní, nicméně dlouhodobější práce je pomalá a příliš opakující se. Nutnost vytvoření objektu na editačním plátně a poté výběru nástroje pro přesun a editaci daného objektu je zdlouhavá a špatná. Jednodušší by pro práci jistě bylo vytvoření objektu a po kliknutí na již vytvořený objekt jej pouze přesunout. Vytvoření nového atributu třídy je opět velice náročné a neintuitivní. Taktéž nemožnost zalomení spojů mezi objekty je limitující, až takřka neproveditelné.

2.2 Enterprise Architect

S ohlédnutím na aplikaci Umbrello a některé její nedostatky, bylo nutné pro pozdější vývoj nástroje prozkoumat další aplikaci. Tato aplikace se jmenuje *Enterprise Architect* a je tvořena firmou *Sparx Systems* [4]. Tato firma se soustřeďuje na vývoj nástrojů pro tvorbu UML grafů. Firmou nabízené produkty jsou placené, ale uživatel obdrží opravdu robustní systémy pro tvorbu UML grafů a možnost úpravy a editace zdrojových kódů pro několik jazyků.

2.2.1 Motivace

Tento nástroj byl vybrán z důvodu vyššího rozšíření při vytváření větších projektů a převážně díky tomu, že tato aplikace nabízí velké množství funkcí a grafů, jak popsat právě vytvářenou aplikaci.

2.2.2 Popis aplikace

Enterprise Architect je velice robustní, již nemá nedostatky jako aplikace Umbrello a nabízí několik možností k úpravě a vytvoření zdrojových kódů projektu. Aplikace rozděluje UML grafy do dvou skupin:

- *Structural Diagrams* pro diagramy na popis aplikace a její struktury.
- *Behavior Diagrams* pro popis chování aplikace a komunikace mezi jejími jednotlivými částmi.

Dále aplikace poskytuje velkou škálu grafů a diagramů – od klasických diagramů pro popis chování dat mezi sebou, přes diagramy pro návrh Win32 UI, až po návrh testování výsledné aplikace. [1]

2.2.3 Výhody a nevýhody

V aplikaci je možné vytvořit velké množství grafů a diagramů. Grafické rozhraní již není tak jednoduché jako tomu je u aplikace Umbrello, většina pokročilých funkcí (jako například přidání a editace atributů třídy) je schována a je náročné je najít.

Pro jednodušší a rychlejší možnost spojení jednotlivých objektů na pracovní ploše lze použít rychlého nástroje vedle objektu. Na stejném místě lze nalézt také funkce pro rychlou editaci a změnu stylu objektu.

Přetažení spoje mezi dvěma objekty lze vytvořit pomocí uchopení za jeden konec a tažení k dalšímu objektu. Bohužel původní spoj se neztratí do doby, než uživatel pustí myš. Toto může být poněkud matoucí.

Největší výhodou nástroje Enterprise Architect je oproti ostatním aplikacím možnost importovat zdrojové kódy a ty zobrazit v diagramu tříd. Lze také vybrat binární soubor (například `.jar` soubor) a aplikace Enterprise Architect může vygenerovat dokumentaci a diagram tříd.

Po úspěšném upravení diagramů tříd lze opět jednoduše provést export do zdrojových kódů. Na výběr je celkem jedenáct jazyků *ActionScript, C, C#, C++, Delphi, Java, PHP, Python, VBNet, Visual Basic, WorkFlow Script.* [1]

[4] http://www.sparxsystems.com

2.3 Zhodnocení nástrojů Umbrello a Enterprise Architect

Při testování nástrojů určených pro tvorbu UML grafů bylo zjištěno mnoho detailů a důležitých vlastností, které bude výsledná aplikace potřebovat. Některé výhody těchto aplikací, jako například generování kódu, nebude potřeba implementovat vzhledem k tomu, že aplikace se soustředí čistě jenom na vytváření a editaci diagramů a UML grafů.

V testovaných aplikacích nebyla nikde možnost automatické předgenerace dalších částí návrhu pomocí předešlých částí. Například provázanost mezi diagramy případů užití a diagramem tříd nebyla možná. V tomto ohledu bylo rozhodnuto, že je potřeba tyto části propojit a více zautomatizovat pro rychlejší a svižnější práci s výslednou aplikací.

V testovaných aplikacích chyběla (nebo byla nedostatečná) možnost pro zalomení hran. Tato vlastnost napomůže zpřehlednění výsledného diagramu, podobně tomu bylo s editací hran.

2.3.1 Vybrané funkce a možnosti z Umbrello

Aplikace Umbrello poskytla mnoho informací o vývoji editační aplikace, jakým chybám se vyvarovat a co je naopak nezbytné k vytvoření kvalitního nástroje zaměřené na tvorbu diagramů.

Vyplynulo, že není zapotřebí ve výsledné aplikaci mnoho náročných grafických elementů, tyto elementy by byly rušivé. Grafické rozložení a dosažitelnost většiny editačních prvků je v aplikaci Umbrello velice jednoduché a intuitvní, z tohoto důvodu bylo ve výsledném nástroji přihlíženo na tyto vlastnosti.

V nástroji, který byl později implementován, bylo zabráněno některým nedostatkům aplikace Umbrello – například zdlouhavé a repetetivní vytváření tříd, jejich přesun po pracovní ploše nebo spojení těchto tříd.

V aplikaci Umbrello je neprakticky řešeno zalomení spojů mezi objekty na pracovním plátně, kdy je nutné vybrat nástroj a až poté vytvořit zalomení, které lze přesunout. Ve vyvíjeném nástroji bylo toto implementováno poměrně jednoduše. Při vytváření spoje dát uživateli okamžitě možnost tuto hranu zalomit, což vede k celkovému zpřehlednění grafu a dává další možnosti při vytváření diagramů.

Nepraktická práce s objekty v nástroji Umbrello také není dobrým příkladem. Převážně z toho důvodu, že po přidání nového objektu na pracovní plátno, je uživatel okamžitě donucen zadat jeho jméno. Dále pro přesun objektů je nutné vybrat nástroj a až poté je možné tento objekt posunout. Toto vede ke zdlouhavým a stále se opakujícím akcím, které uživatele zbytečně zdržují od práce. Tato činost byla vyhodnocena jako nepřínosná a nutnost zadat jméno objektu při jeho vytvoření bylo nahrazeno pozdější úpravou, kdy si uživatel vytvoří objekt a až poté určí jeho jméno. Podobně tomu bylo u přesunu objektu, kdy byla zavrhnuta nutnost výběru speciálního nástroje pro tento přesun.

2.3.2 Vybrané vlastnosti z Enterprise Architect

Aplikace Enterprise Architect posloužila pro další zhodnocení požadavků a vyvarování se chyb. Především náročnost a komplexnost aplikace se nehodí pro výslednou aplikaci.

Ve výsledném nástroji bude zapotřebí možnost jednoduchého zalomení hran a jejich jednoduché editace. Tato funkce bohužel v aplikaci Enterprise Architect je opět nevyhovující a nepraktická. To vede k nepřehlednosti větších diagramů.

Umístnění otevřených souborů je v nástroji Enterprise Architect poněkud netradiční – v dolní části aplikace. Uživatel často nevidí otevřený projekt a proto se může stát, že zavře

stávající jen kvůli tomu, aby našel již otevřený soubor. Při porovnání s aplikací Umbrello, která toto řeší více standardním způsobem (umístněním záložek v horní části okna), je tento způsob poněkud nepraktický.

V Enterprise Architect jsou druhy diagramů a grafů umístněny ve stromové struktuře z důvodů rozsáhlosti aplikace. Toto je opět nepoužitelné v menších aplikacích a zbytečně náročné na hledání a přidání nového diagramu nebo grafu.

Kapitola 3

Rozbor diagramů

Po prozkoumání nástrojů specializujících se na návrh aplikace bylo zvoleno, že pro popis chování aplikace bude vhodné použít *diagramy případů užití* 3.2 a pro popis jednotlivých tříd, vztahů mezi těmito třídami a návrh operací je vhodný *diagram tříd* 3.3. Pro popis chování tříd a jejich metod bylo zvoleno Petriho sítí, přesněji jejich rozšíření s názvem *objektově orientované Petriho sítě* 3.4.

Dále byly tyto diagramy a grafy spojeny pro vzájemnou komunikaci na základě takzvaného *Use-case driven developmentu*. Tento způsob se orientuje na propojení diagramu případů užití do diagramu tříd a poté určuje napojení objektově orientovaných Petriho sítí na tyto třídy, což je blíže popsáno v kapitole 3.5.

3.1 Objektově orientované programování

Předtím, než rozebereme grafy, které slouží pro popis vývoje a vytvoření návrhu je potřeba analyzovat objěktově orientované programování. Jedná se o speciální styl programování používaný převážně při vývojí větších a náročnějčích aplikací.

3.1.1 Objekt

Základ objektově orientovaného programování je samozřejmě objekt. Je to specifický pohled na fyzikální a konceptuální jednotky reálného světa. Jednotlivé části objektu jsou:

- Stav – nese informaci, o tom v jakém stavu se objekt v daném čase nachází.
- Chování – popisuje reakce na vnější a vnitřní události.
- Identita – u každého objektu je jednoznačně identifikovatelná v rámci systému – má jméno, kterým se liší od každého jiného objektu.

Jako jeden z hlavních rysů objektově orientovaného programování je zapouzdření – skrytí veškerých dat a práce s daty před ostatními objekty. Ostatní objekty jsou schopné komunikovat mezi sebou pomocí zasílání zpráv. Tyto zprávy obsahují jméno operace, parametry (data potřebná pro zpracování) a jméno objektu obsahujícího tuto operaci. Zaslaná zpráva může sloužit jak ke zpracování a navrácení dat, tak k vnitřní změně objektu, který tuto zprávu přijal.

Veřejné rozhraní objektu je určeno množinou operací, což je výčet schopností objektu. Popis provádění operace je schován uvnitř objektu a je určen metodami. Uvnitř objektu se také nachází atributy, což je předpis pro uchování dat. [4]

3.1.2 Instanciace třídy

Ve většině programovacích jazyků se vytváření objektů provádí speciální konstrukcí, takzvanou třídou. Třída je předpis pro objekt.

Třída definuje viditelnost instančních proměnných (atributů) a metod. Dále definuje metody a jejich provádění.

Objekty mohou být instanciovány staticky a dynamicky. Staticky instanciované třídy se vytvoří při spustění programu a zanikají až při jeho ukončení. Dynamicky instanciované proměnné vznikají a zanikají v průběhu chodu aplikace v reakci na požadavky uživatele. [4]

3.1.3 Dědičnost

Další ze silných vlastností objektově orientovaného programování je dědění. Slouží ke sdílení jisté charakteristiky jednoho objektu s druhým.

Každý jazyk má tuto vlastnost implementovanou různě. Dynamické a statické dědění může být zaměřeno na třídy nebo objekty. Může být děděna reprezentace, nebo chování. Dědění může být implementováno částečně, nebo úplně. A v neposlední řadě lze podle počtu dědění rozlišit dědění násobné a jednoduché. [4]

3.2 Diagram případů užití

Diagramy případů užití jsou často modelovány a používány ke komunikaci mezi zákazníkem, který nemá příliš mnoho zkušeností s vývojem systému, ale rozumí danému problému a svému oboru, a programátorem, člověkem, který naopak má zkušenosti s vývojem aplikace, ale nechápe do hloubky zákazníkův problém a jeho obor.

Základními prvky diagramu případů užití jsou případy užití a aktéři. V požadavcích na systém rozpoznáváme takzvané „Stakeholdery“, jsou to osoby nebo věci, které mají zájem na úspěšném provedení systému. Případy užití popisují, jak se má systém zachovat na reakce stakeholderů, akterů, a dodat určité výsledky. Aktéři jsou na druhou stranu přímí uživatelé nebo věci s určitým chováním. Hlavní aktér je často stakeholderem a hlavní spouštěč tohoto systému. Občas je zapotřebí aktér, který není přímo v systému. Tomuto se říká „Podpůrný aktér“. Aktér není přímo chápán jako jedinec, nýbrž jako role v systému. [6]

3.2.1 Rozbor diagramu připadů užití

Diagramy připadů užití se používají pro grafický popis případů užití a zahrnují několik pravidel. Aktéři jsou znázorňováni ve tvaru postav a případy užití jako elipsy obsahující název prováděného úkonu. Spoje mezi případy užití a aktéry jsou znázorněny úsečkou vedoucí od aktéra k případu užití.

Mezi případy užití mohou být vedeny dále spoje pro jejich rozšíření. Jsou to *include*, *extends* a *generalizace*. Poslední jmenovaný spoj je možné také vést mezi aktéry.

Include

Include znamená to, že pro správný chod daného nadřazeného případu užití je zapotřebí použití vloženého. Pro vykonání vyššího případu užití je zapotřebí nejdříve vykonat vložený. Pokud je při vykonávání několika případů užití prováděna stejná operace, je vhodné tuto funkcionalitu oddělit z daného případu užití a spojit se všemi případy užití spojem include.

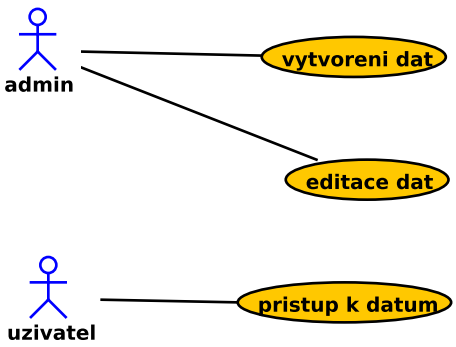
Spoj include je znázorněn přerušovanou šipkou doprovázené textem „include“. Spoj je veden z nadřazeného případu užití k vloženému.

Extends

Extends slouží pro rozšíření případů užití. To znamená, že je rozšířením hlavního případu užití, které bylo vyjmuto z původního případu. V překladu tento spoj znamená, že jeden případ užití je rozšířen dalším případem užití. Spoj je znázorňován přerušovanou šipkou vedenou do rozšiřovaného (hlavního) případu užití.

Generalizace

Generalizaci lze použít jak v případech užití, tak mezi aktéry. Slouží ke znázornění rodičů a potomků. Použití tohoto spoje znamená, že potomek přebírá veškerou funkcionalitu svého rodiče a tu může poupravit nebo přidat vlastní. Tento spoj je znázorněn spojem ukončeným prázdnou šipkou a je veden od potomka k rodiči. [6]



Obrázek 3.1: Příklad diagramu případů užití.

Na obrázku 3.1 lze vidět jednoduchý systém přidávání, editace a zobrazení dat, kdy se v systému nachází dvě role a každá má přístup k odlišným akcím.

3.3 Diagram tříd

Po znázornění funkčních požadavků na aplikaci pomocí diagramů případů užití se nabízí další nástroj z UML rodiny a tím je *Diagram tříd*. Tento diagram slouží pro grafické znázornění tříd. Bližší seznámení s objektově orientovaným programováním bylo probráno v kapitole 3.1.

3.3.1 Prvky diagramu tříd

Hlavní složkou diagramu jsou třídy, které jsou znázorněny pomocí obdélníků obsahující další informace o dané třídě. V horní části je název třídy. Poté třída obsahuje třídní proměnné, atributy. A poslední část tříd jsou operace, metody.

V diagramu tříd lze také namodelovat interface – to je speciální třída obsahující předpis pro atributy a operace. V interfacu se nenachází implementace těchto operací, ta se nachází až ve třídách, které používají tento interface.

Zápis atributů a operací tříd je stanoven tak, že za jménem následuje dvojtečka a její typ. V případě operací se napíše jméno a za ním je uvedena závorka obsahující seznam argumentů a až poté dvojtečka určující návratový typ. Pokud operace nic nevrací, je možné indikovat toto pomocí nastavení návratové hodnoty na „void“. Dále je možné omezit vyditelnost atributů a operací pomocí identifikátoru. *Private* označované „-“, *protected* jako „#“ a *public* znázorněné pomocí „+“. Tyto identifikátory se píší před jméno atributu nebo operace.

Jednotlivé hrany pro spojení se dělí na *Generalizaci*, *Asociaci*, *Agregaci*, *Kompozici* a *Implementaci*. [5]

Generalizační hrana

Hrana *Generalizace* popisuje spojení mezi třídou, která je více obecná a třídou, která je specifická. Generická třída se nazývá *superclass*, nebo také *rodič*, a více specifická třída se nazývá *subclass*, známá též jako *potomek*.

V UML je tato hrana popsána šipkou ukončenou bílým trojúhelníkem na konci. Tato hrana je vedena směrem od generické třídy k více specifické.

Je také možné několikanásobná dědičnost, ale v případě implementace této dědičnosti se musí vzít v potaz to, zda daný jazyk několikanásobnou dědičnost podporuje, protože při takovéto dědičnosti často může nastat problém. Především díky tomu, že pokud třída A dědí od tříd B a C a třída B i třída C obsahují stejnou metodu, programátor musí řešit, která metoda se má v třídě A použít. Například v jazyce Java™ několikanásobná dědičnost povolena není, ale je možné použít interface. [5]

Asociační hrana

Asociační hrana je vztah mezi třídami, který určuje, že objekty jedné třídy jsou používány v objektech druhé třídy.

Asociace je kreslena v UML pomocí hrany s jednou šipkou na konci a jmenuje se *jednosměrná*. Takto použitá asociace znamená, že třída, ze které šipka vede, je použita ve třídě, do které tato šipka směřuje. Tuto hranu lze kreslit také s šipkami na obou koncích, jmenuje se **obousměrná** a znamená, že obě třídy se používají navzájem. Takto určená asociace není příliš používaná. [5]

Agregační hrana

Použití spoje Agregace značí, že určitý objekt potřebuje ke svému chodu další objekt. Pokud ovšem přestane existovat původní objekt, další objekt lze nadále v systému použít. To znamená, že objekt třídy A potřebuje ke svému chodu objekt třídy B. V momentě odstranění objektu třídy A ze systému ale objekt třídy B nezaniká. Tomuto se říká slabá agregace. Tato hrana je kreslena souvislou čarou opatřenou bílým diamantem na konci. [5]

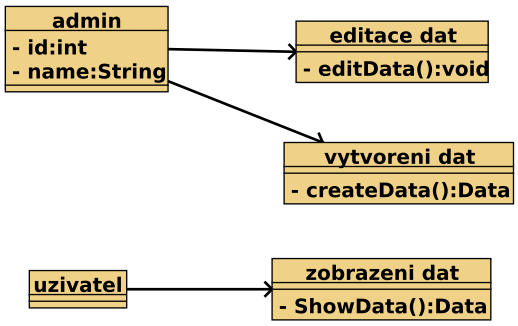
Kompoziční hrana

Tato hrana je z části stejná jako hrana Agregační s tím rozdílem, že reprezentuje silnou agregaci. To znamená, že pokud objekt třídy A potřebuje ke svému chodu objekt třídy B,

v momentě odstranění objektu třídy A je ze systému odtraněn objekt třídy B. Tato hrana je kreslena souvislou čarou opatřenou černým diamantem na konci. [5]

Implementační hrana

Tato hrana je specifická v tom, že se kreslí pouze mezi třídou a interfacem. Slouží ke znázornění, že třída A implementuje interface B, převezme všechny atributy třídy B a přetíží operace třídy B, které implementuje. Hrana je kreslena přerušovanou šipkou ve směru od implementované třídy k interfacu. [5]



Obrázek 3.2: Příklad diagramu tříd.

Na obrázku 3.2 lze vidět způsob spojení několika tříd, kdy některé třídy obsahují třídní proměnné, v horní části a ve spodní poté jejich metody.

3.4 Objektově orientované Petriho sítě

Pro popis tříd a chování objektů na zaslané zprávy a popis implementace metod (operací) bylo použito objektově orientovaných Petriho sítí.

3.4.1 Petriho sítě

Objektově orientované Petriho sítě jsou rozšířením Petriho sítí. Základní Petriho sítě jsou vhodné pro modelování a teoretické zkoumání paralerních systémů.

Chování dynamického systému je v podstatě sledování stavové proměnné a zaznamenávání její hodnoty v čase.

Petriho síť lze matematicky zapsat jako $N = (P_N, T_N, PI_N, TI_N)$, kde:

- P_N je konečná množina *míst*.
- T_N je konečná množina *přechodů*, $P_N \bigcap T_N = \emptyset$.
- $PI_N : P_N \to \mathbb{N}$ je *inicializační funkce*.
- TI_N je *popis přechodů*. Je to funkce definovaná na T_N, taková, že $\forall t \in T_N$: $TI_N = (PRECOND_t^N, POSTCOND_t^N)$, kde:

1. $RECOND_t^N : PN \to \mathbb{N}$ jsou *vstupní podmínky* přechodu.
2. $POSTCOND_t^N : P_N \to \mathbb{N}$ jsou *výstupní podmínky* přechodu t.

Petriho sítě se skládají z míst a přechodů, tyto prvky jsou spojeny orientovanými hranami. Nemohou být spojeny dvě místa mezi sebou a stejně tak nemohou být spojeny dva přechody přímou hranou.

Procesy jsou v Petriho sítích znázorněny tečkou uvnitř míst. V průběhu provádění simulace se tento bod přesouvá mezi jednotlivými místy. Tyto přesuny jsou provedeny na základě splnění pravidel zanesených v přechodech. Tato pravidla jsou atomická, což znamená že se buď toto pravidlo provede nebo neprovede.

Přechody mohou být okamžité, při splněný zanesených pravidel se okamžitě provedou. Nebo mohou být časované, což znamená, že po určitém čase se daný přechod realizuje. Toto je velice jednoduché rozšíření základních Petriho sítí.

Úspěšné provádění přechodů probíhá tak, že nejdříve dojde k odebrání značek ze vstupního místa a poté přídání značek na výstupní místo.

Dále můžeme sítě rozšířit pomocí prioritizací spojení mezi místy a přechody, nebo vložením procentuální pravděpodobnosti provedení daného spoje. [4]

Rozšíření Petriho sítí

Základní Petriho sítě představují příliš nízkoúrovňový prototyp pro modelování reálných systémů. Z toho důvodu bylo zavedeno několik rozšíření. Pro pozdější použití nás nejvíce zajímá rozšíření Petriho sítí na funkcionální, což je rozdělení přechodu na stráž a akci. Do místa akce je možné volat jednotlivé části sítí.

Toto lze poté rozšířit o objektové rozdělení s tím, že v místě akce lze volat vytvoření nového objektu jiné sítě a poté zaslání zprávy tomuto objektu.

3.4.2 Objektově orientované Petriho sítě

Model systému, ve kterém mohou dynamicky vznikat a zanikat objekty komunikující předáváním zpráv, lze vhodně znázornit pomocí objektově orientovaných Petriho sítí. Jako model je množina tříd, hierarchicky organizována podle dědičnosti. Každá z těchto tříd obsahuje tyto prvky:

- Síť objektu, která definuje reprezentaci instance třídy.
- Množinu sítí metod, které modelují reakce na zaslané zprávy.
- Množinu synchronních portů, definující reakce na synchronní zprávy.

Sítě metod mohou sdílet atributy objektu. Mají navíc parametrová místa, do kterých se při invokaci takové sítě vloží parametry ze zprávy. Dále mají výstupní místo *return*, což je, místo do kterého se uloží výstupní hodnota.

Značky v sítích nereprezentují samotné objekty, ale pouze reference na tyto objekty. V objektově orientovaných Petriho sítích jsou tři druhy objektů. Primitivní třídy to jsou čísla znaky, řetězce, symboly a seznamy. Další objekty jsou třídy, tyto objekty se v průběhu simulace nemění a rozumějí na volání *new*, což vede k vytvoření nové instance. Ostatní objekty jsou instancemi třídy.

Proveditelnost přechodu v objektově orientovaných Petriho sítích je rozděleno na stráž přechodu a akci. Stráž přechodu obsahuje sekvenci výrazů a každý z nich je zaslání zprávy. Pokud každá dílčí zpráva vrací *true*, stráž je vyhodnocena jako *true*.

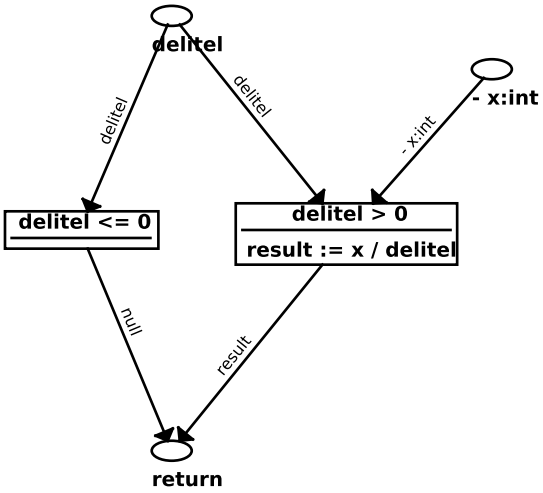
Akce přechodu specifikuje zaslání zprávy. Budeme uvažovat tvar zaslání zprávy (3.1):

$$y := x_0.msg(x_1, \ldots, x_n), \tag{3.1}$$

kde y je proměnná, x_i je term a msg je jméno zprávy. Objekt se jménem x_0 je adresátem zaslané zprávy a x_i jsou argumenty této zprávy. V těchto akcích je zapotřebí definovat také speciální zprávu *new*, která vede k vytvoření nové instance třídy x_0.

Výsledná objektově orientovaná Petriho síť má všechny atributy objektově orientovaného jazyka. Umožňuje zapouzdření, polymorfismus a dědičnost. [4]

Na obrázku 3.3 lze vidět příklad návrhu metody dělení, kdy v případě, že dělitel je 0 vrátí se null, v opačném případě se do návratového místa uloží podíl x a *delitel*.



Obrázek 3.3: Příklad popisu metody pro dělení pomocí objektově orientovaných Petriho sítí.

3.4.3 Grafické znázornění

Jednotlivá místa se znázorňují v objektově orientovaných Petriho sítích jako kolečka obsahující stav tohoto místa, například referenci na objekty. Přechody jsou znázorněny obdélníkem obsahujícím stráž a akci. Akce se nachází pod čarou a nad čarou se nachází stráž.

3.5 Koncept aplikace

Vytvořit jednotlivé diagramy a Petriho sítě je pouze část tohoto nástroje, jeho předností má být automatické a urychlené vytváření aplikací. Z tohoto důvodu bylo zvoleno zavedení *Use-Case driven developmentu*.

Tento pohled na návrh aplikace je veden tak, že na základě diagramu případů užití je možné vytvořit automaticky třídy v diagramu tříd. A to tak, že aktér je modelován jako třída se speciálním typem *Actor* a případ užití je modelován jako třída s určením *Activity*.

Pro tyto třídy nadále platí stejná omezení jako pro aktéry a případy užití. Dále je tento způsob výhodný z toho hlediska, že po navrhnutí chování aplikace máme okamžitě k dispozici základní kostru a strukturu v diagramu tříd. Tato kostra se dá rozšířit dalšími

třídami, které mohou být zobrazeny v diagramu případů užití po určení jejich role, nebo mohou být pouze jako pomocné třídy (nezobrazené v diagramu případů užití).

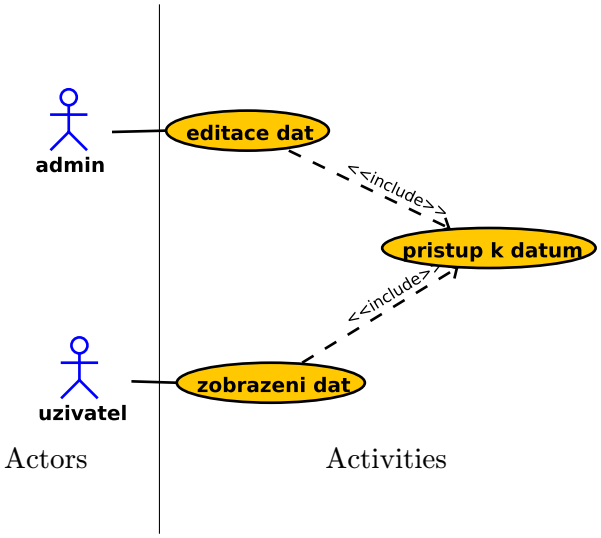
Takto vytvořené třídy a další třídy lze poté popsat pomocí objektově orientovaných Petriho sítí. To se provede tak, že každá třída má svoji zvláštní Petriho síť, ve které lze znázornit komunikaci s ostatními třídami. Každá metoda této třídy má opět vlastní Petriho síť, tato síť má ještě rozšíření v podobě speciálního místa určeného pro návratovou hodnotu.

Použitím tohoto koceptu lze dosáhnout plynulého návrhu aplikace a po vytvoření objektově orientovaných Petriho sítí je možné testování této aplikace bez nutnosti editace zdrojových kódů [7].

3.5.1 Použití ve výsledné aplikaci

Provázání mezi UML částmi probíhá tak, že při vytvoření objektu v diagramu případů užití vznikne třída se stejným názvem v diagramu tříd. Na objektovém analytikovi potom je, aby zadal jednotlivé atributy a operace. Pokud vznikne případ, že se analytická třída rozpadne na více tříd, je toto možné realizovat dvojím způsobem. A to tak, že se odstraní analytická třída a vzniknou dvě nové třídy, které se nepromítnou do diagramu případů užití. A nebo zachováním analytické třídy a navázáním nové třídy, která se nepromítne do případů užití.

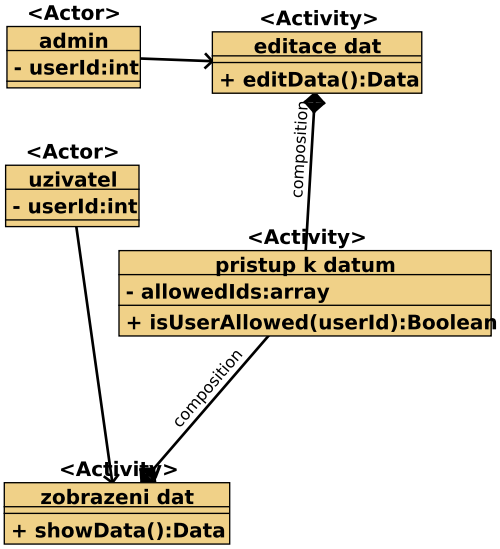
Způsob znázornění jednoduchého případu užití je vidět na obrázku 3.4, kde máme dva aktéry a každý z nich používá různou akci. Tyto akce jsou však rozšířeny pomocí include spoje o případ užití.



Obrázek 3.4: Nástin problému případů užití.

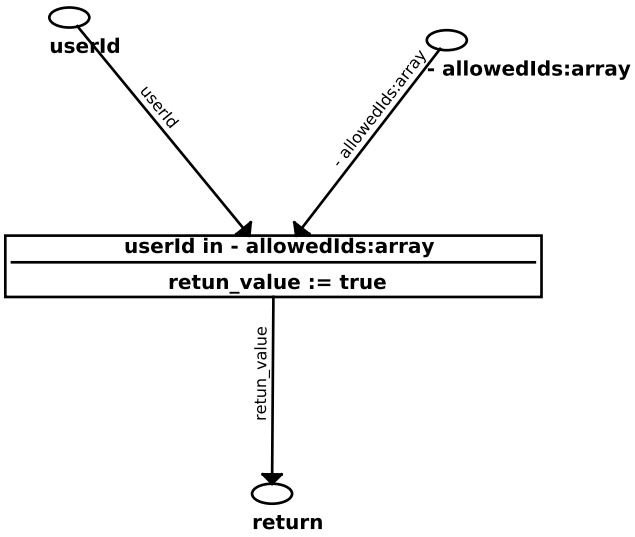
Tento případ užití se poté promítne do diagramu tříd s tím, že jednotliví aktéři budou reprezentováni třídami s příznakem *Actor* a případy užití jako *Activity*. Spoje mezi těmito třídami jsou totožné s těmi, co jsou v diagramu případů užití. S tím rozdílem, že je na objektovém analytikovi rozhodnout, jaký spoj se použije v případě include – zda agregace, nebo kompozice. Objektový analytik je před toto rozhodnutí také postaven v případě použití extends spoje. Po vytvoření těchto tříd je možné zadat třídní proměnné a metody k jednotlivým třídám. Výsledný diagram provázání můžeme vidět na obrázku 3.5, kdy aktéři odpovídají třídám, které mají příznak *Actor* a případy užití třídám s příznaky *Activity*.

K popisu jednotlivých tříd a metod dále slouží objektově orientované Petriho sítě. Každá



Obrázek 3.5: Nástin problému převodu případu užití do diagramu tříd.

třída a také každá metoda má jednu objektově orientovanou Petriho síť. Příklad navrhnutí metody *isUserAllowed* (z třídy „pristup k datum“) je vidět na obrázku 3.6, kdy se kontroluje, zda se uživatelské ID nachází v povolených, pokud tomu tak je, vrátí se hodnota *true*.



Obrázek 3.6: Metoda isUserAllowed popsána objektově orientovanou Petriho sítí.

Objektově orientované Petriho sítě mají několik speciálních míst. V případě Petriho sítě pro třídu jsou to třídní proměnné této třídy a zděděné třídní proměnné od svých předků. V případě Petriho sítě pro metodu je množina těchto zvláštních míst rozšířena o argumenty metody (které jsou vstupními místy pro tuto síť) a jedno speciální místo, kde je uložena návratová hodnota. [8]

3.6 Shrnutí

V této kapitole byly přiblíženy nástroje sloužící ke znázornění postupného vývoje aplikace, ke zjednodušení vývoje a převážně k zpřehlednění komunikace uvnitř aplikace.

Motivací pro výběr těchto nástrojů byla skutečnost, že se v moderním programování používá převážně objektově orientovaných postupů. Tyto nástroje dále podporují *Use-case driven development*, což vede k jednoduché provázanosti mezi těmito nástroji a ke zjednodušení a urychlení vývoje. S ohledem na moderní způsob agilního vývoje jsou tyto nástroje vhodné, a to převážně z hlediska jednoduché integrace zákazníka do vývoje a okamžité promítnutí změn od zákazníka na celý systém.

Kapitola 4

Návrh aplikace

Po vyzkoušení nástrojů specializujících se na návrh aplikací, bylo zjištěno, že žádný z nich neimplementuje možnost vzájemného propojení, jak bylo specifikováno v požadavcích. Také byl potvrzen očekávaný výsledek, kdy žádný nástroj neobsahoval možnost návrhu pomocí objektově orientovaných Petriho sítí.

Nejdříve si v 4.1 objasníme design aplikace, rozmístnění jednotlivých prvků a způsob ovládání. Poté zde probereme některé speciální ovládací prvky. Závěrem této kapitoly popíšeme jednotlivé editory, jejich rozšíření a používání 4.3.

4.1 Popis aplikace

Na základě poznatků z ostatních aplikací bylo rozhodnuto o jednoduchosti a intuitivním ovládání. Tohoto bylo docíleno sjednocením základního designu aplikace a použitím klávesových zkratek, které jsou známé z moderních aplikací.

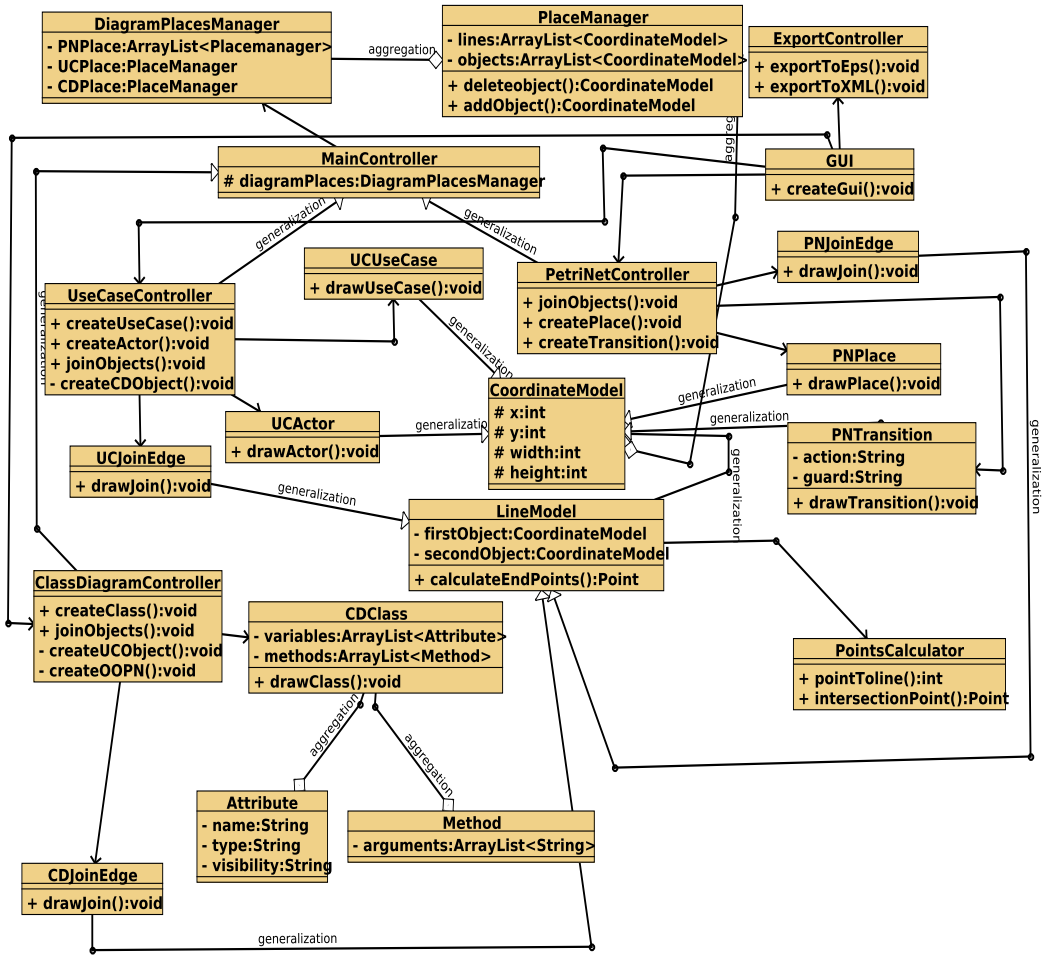
Hlavní okno se skládá z několika panelů. V horní části se nachází menu obsahující jednoduché operace nad projekty. Dále jsou zde umístněny ovládací prvky, sloužící pro práci se soubory a exporty daného souboru. Celý hlavní obsah aplikace se nachází ve zbylých dvou třetinách okna. V levé jsou tlačítka pro možnost přepínání jednotlivých vkládaných objektů a přepínání mezi druhy spojů. Dolní část obsahuje informační a editační možnosti zvoleného prvku a pro hlavní část je vyčleněn zbytek pracovní plochy – takzvané editační plátno.

Na toto plátno je možné přidávat jednotlivé prvky, spojovat je, přesouvat je a mazat je. Po vybrání některého tlačítka v levém menu je možné začít přidávat prvky. Po zakliknutí vybraného prvu a kliknutí na editační plátno se vytvoří objekt, po tažení se vybraný objekt přesune a pro spojení dvou objektů hranou je zapotřebí vybrat typ hrany z levého menu a poté kliknout na první objekt a následně na druhý. Pokud je potřeba spoj zahnout pro zpřehlednění grafu, je toto možné vyvolat kliknutím na pravé tlačítko myši během tažení spoje.

4.2 Architektura aplikace

Architekturu výsledného nástroje si lze prohlédnout na obrázku 4.1, kde je částečně znázorněno, jak jsou jednotlivé třídy navzájem provázané.

Třída *GUI* je základní třída, obsahuje *main* a také vytváří objekty z jednotlivch tříd. V třídě *PlaceManager* jsou uloženy všechny objekty a jejich všechny spoje. Tato třída je



Obrázek 4.1: Architektura aplikace.

obsahem třídy *DiagramPlacesManager*, která se stará o jednotlivé části a diagramy. Petriho sítě jsou uloženy pomocí *ArrayListu*, protože je potřeba mít více sítí pro jeden projekt.

CoordinateModel je třída starající se o každý objekt. Obsahuje X a Y souřadnice, rozměry a základní popis objektu. Od této třídy dědí dále třídy *UCActor, UCUseCase, CDClass, PNPlace a PNTransition*. Tyto třídy specifikují jednotlivé objekty na editačním plátně a také se starají o vykreslování na pracovní plátno.

Od třídy *CoordinateModel* dále dědí *Linemodel*, ta se stará o uložení jednotlivých spojů mezi objekty. Koncové body jsou spočítány pomocí *PointsCalculator* a nejdůležitější rolí třídy *Linemodel* je, že je rodičovskou třídou pro všechny spoje, takže od ní dědí třídy *UCJoinEdge, CDJoinEdge, PNJoinEdge*. Tyto třídy se starají o styl vykreslovaných spojů a nastavují různá pravidla pro spoje.

Základní třídou pro všechny editory je *MainController*. Ukládá v sobě všechny objekty z daného projektu a je rodičem tříd *UseCaseController, ClassDiagramController a PetriNetController*, které se starají o vykonání akcí – jako například vytvoření adekvátní třídy po přidání případu užití.

Každá třída má několik metod, zde uvedené jsou pouze na ukázku. Převážně třídy, které mají v názvu slovo „Model“ jsou značně rozsáhlé díky tomu, že je v nich uloženo vešekeré nastavení třídních proměnných pro potomky. Dále jednotlivé třídní proměnné mají v každé

třídě metody pro jejich získání a popřípadě jejich nastavení (takzvané „getery“ a „setery“).

Podrobnější popis tříd a metod je v dokumentaci tohoto nástroje [1].

4.3 Jednotlivé editory

Jednotlivé editory, jak již bylo řečeno v přechozích kapitolách, jsou *diagram případů užití* 3.2, *diagram tříd* 3.3 a *objektově orientované Petriho sítě* 3.4. T0yto editory byly nadále mírně pozměněny, převážně z grafického hlediska pro zlepšení přehlednosti těchto diagramů a pro jednodušší práci mezi nimi.

Základním odlišením ve všech editorech je zvýraznění objektu, který nemá žádnou spojující hranu. Tyto objekty jsou zvýrazněny pomocí čárkování. Vybraný objekt je znázorněn ohraničením a odlišnou barvou. Pokud objekt nemá odpovídající „obraz“, je zvýrazněn kurzívou a textem pod objektem. Toto platí převážně pro propojení diagramu tříd a diagramu případů užití. Pokud spoj mezi dvěma objekty nemá tento obraz, je zvýrazněn odlišnou barvou.

4.3.1 Editor případů užití

Tento editor je prvním editorem a slouží pro popis chodu aplikace, zde se nachází pouze mírné úpravy.

V případě vybrání *aktéra* je zakázáno uživateli spojit stejné objekty jinak než děděním. Dále může uživatel spojit *aktéra* a *případ užití* pouze pomocí asociace. Toto lze provést pouze ve směru od *aktéra*, tudíž pokud toto uživatel provede naopak je hrana otočena automaticky.

Pro změnu jména objektu je potřeba na něj poklepat dvakrát. Zde je také možné změnit jeho propojení s třídou, nebo reaktivovat spoj mezi třídou a objektem, pokud v Diagramu tříd není třída odpovídající editovanému obejktu. S touto reaktivací se také znovu vytvoří spoje mezi třídami a nebo mezi objekty v diagramu případů užití.

Pokud chce uživatel hromadně reaktivovat objekty případů užití a opět vytvořit třídy v diagramu tříd, lze to provést kliknutím na tlačítko *Reactivate all inactive*. Pokud na druhou stranu uživatel chce hromadně smazat všechny neaktivní objekty, může kliknout na *Delete all inactive*.

4.3.2 Editor diagramu tříd

V tomto editoru je možné již popsané chování aplikace rozdělit pomocí objektově orientovaného programování popsaného v sekci 3.1. Dále lze vytvořit jednotlivé třídy a ty poté editovat.

Pokud je vytvořen a vybrán *interface* není možné jej spojit s ostatními třídami jinak než spojem *implementace*, nebo mezi jednotlivými *interfacy* pomocí spoje *generalizace*.

Pro editaci jména je nutné dvakrát poklepat na třídu a v okně, které vyskočí je také možné nastavit její „roli“ v systému.

- *Actor* – po vybrání tohoto druhy třídy se v diagramu případů užití vyvtoří Aktér.
- *Activity* – pokud je vybrán tento typ třídy je v diagramu případů užití vytvořen případ užití.

[1] Dokumentace je součástí přiloženého CD.

- *None* – tento druh třídy slouží pro vytváření pomocných tříd.
- *Interface* – pouze pro vytvoření metod a třídních proměnných, nelze pro ně vytvořit Petriho síť.

Výběr role je možný pouze u tříd, které nemají doposud žádný spoj. Třídy *None* a *Interface* se nepromítnou do diagramu případů užití, slouží pouze ke snadnější implementaci některých funkcí.

Pokud je potřeba znovu aktivovat a vytvořit obraz v diagramu případů užití je nutné třídu nejdříve spojit s další třídou a poté po dvojkliku na tuto třídu bude vyvoláno okno s editací a tato třída bude mít možnost opětovné aktivace objektu a nebo spojení s jiným objektem.

Pro přidání metod a třídních proměnných se nachází v pravé dolní části editor tříd. Nejdříve je zapotřebí vybrat třídu, pro kterou požadujeme přídáni některého prvku a poté zadat viditelnost, jméno a typ. Příslušný řádek slouží pro třídní proměnnou nebo metodu. Pro zjednodušení je možné nastavit typ proměnné a operace, jako typ jakékoliv třídy, která se momentálně nachází na editačním plátně.

Po úspěšném vložení metod a popřípadě třídních proměnných je možné přidat argumenty metodám. To se provádí v levé spodní části. Po výběru třídy, pro kterou je potřeba editovat argumenty metody, dvojklikem vybereme metodu, kterou je potřeba editovat a poté zadat argumenty oddělené čárkami. Pokud chceme odstranit některou třídní proměnnou, nebo metodu, je toto možné provést tlačítkem vedle položky.

4.3.3 Editor objektově orientovaných Petriho sítí

Pro vstup do tohoto editoru je zapotřebí vybrat třídu, pro kterou budou editovány Petriho sítě. Automaticky je vybrána Petriho síť pro třídu, přepnutí mezi Petriho sítěmi pro třídu a metody probíhá v levém dolním rohu.

V pravém dolním rohu, pokud není vybrán žádný objekt na pracovním plátně, je možné přidat třídní proměnné nebo argumenty metody jako *místa*. Takto přidaná místa nelze editovat. Vkládat konstanty, operovat s těmito místy nebo je mazat lze, ale měnit jejich jméno není povolené. Podobně tomu je u návratových míst v Petriho sítích pro metody, kde je jedno speciální místo *return*. Toto místo je podobně needitovatelné a navíc jej není možné odstranit.

Přidaná místa je možné dále editovat pomocí vkládání konstant. Stačí vybrat místo a v pravém dolním rohu aplikace lze vidět aktuální konstantu (pokud není žádná, je input prázdný), kterou lze změnit.

Po přídáni *přechodu* (často nazývaném Transition) je možné tomuto Přechodu editovat dvě složky. Složku *stráž* (nazýván též Guard) a složku *akce* (Activity). Výběrem přechodu se zobrazí v pravé dolní části možnost tyto složky editovat. Po kliknutí na příslušné tlačítko budou vyvolány okna pro editaci jednotlivých složek.

- *Složka Stráže*: Pro základní vytváření pravidel je v horní části okna pro editaci stráže volné textové pole, kam je možné zadat jakýkoliv text. Pokud je potřeba zadat více kontrolovaný vstup, je možné si ve spodní části vybrat proměnné nebo zadat konstanty a vybrat jejich podmínku. Poté lze nastavit horní text přímo na tuto podmínku nebo spojit operátorem s přechozí podmínkou. V textové části lze poté výslednou podmínku editovat a dále upravovat.

- *Složka Akce*: Pro vytvoření nové proměnné je možné využít horního pole. Lze také vybrat proměnnou, která je již spojena s tímto přechodem. Další editace akce probíhá přes textové pole – toto pole ale není nijak kontrolované a upravované. Rozdělení takového textu a jeho kontrola je příliš náročná, navíc nebyla předmětem této práce. Ale přidání veškerých kontrol na podobné texty v objektově orientovaných Petriho sítích je jistě dobrý plán na rozšíření tohoto nástroje.

V objektově orientovaných Petriho sítí je pouze jeden spoj. Pro tento spoj platí omezení v podobě možnosti spojit pouze místo s přechodem, nebo naopak. Není však možné spojit dvě místa nebo přechody přímo mezi sebou.

Dalším rozšířením spojů oproti ostatním editorům je to, že lze přidávat proměnné, které jsou přes tento spoj „přenášeny“. Po dvjokliku na hranu si lze vybrat množinu proměnných, které jsou automaticky hlídány z míst a přechodů nebo zadat samostatný text. Tento samostatný text je zde kvůli náročnějším programům a operacím, jako je například pole nebo seznam. Tento text však není nijak kontrolovaný, převážně kvůli náročnosti na rozdělení do tokenů a provedení analýzy těchto tokenů. Pokud je potřeba dát najevo, že hrana neobsahuje žádná data, lze ji nechat prázdnou, popřípadě uvést *null* nebo *nill* do textového pole spoje.

4.3.4 Provázání jednotlivých částí

Aplikace poskytuje jednoduché provázání a spojení částí, které spolu souvisí – jako je tomu například u diagramu případů užití a diagramu tříd. Editace a vytvoření objektu v jedné části vede k vyvolání akce v druhé. Toto chování je částečně zautomatizováno a tím ulehčena práce, nicméně uživateli je ponechána volná ruka v těchto ohledech, a pokud potřebuje změnit předgenerované části, nejsou mu kladeny žádné překážky.

Aplikace pouze napomáhá vývoji a postupnému vytváření určitých částí. Nemá za úkol být naprosto zautomatizována a vytvářet vše bez zásahů od uživatele.

Kapitola 5

Rozšířené možnosti nástroje

Během vývoje aplikace bylo počítáno s některými rozšířeními a potřebnými úpravami editoru. Převážně z toho důvodu, aby aplikace byla snáze použitelná a přinášela nové prvky.

Řekneme si něco o způsobech exportu aplikace a proč byl vybrán tento druh exportů 5.1. Dále si popíšeme v kapitole 5.2 popíšeme, v čem je tento nástroj vylepšený po grafické stránce některých ovládacích prvků oproti konkurenci.

5.1 Export diagramů

Jedním z požadavků byl exportu do formátu *XML*. Tohoto bylo využito v případě ukládání stavu aplikace, kdy otevřený projekt je možné uložit do formátu XML. Tento formát slouží pro ukládání a načítání projektů. Formát XML byl zvolen částečně kvůli jeho možnému upravení, kdy lze editovat obsah objektů bez nutnosti zapnutého nástroje. Toto se ale doporučuje pouze zkušeným uživatelům, protože je možné, že se díky tomuto zásahu jakýmkoliv způsobem poruší editovaný soubor, který již nebude možné opravit v tomto nástroji. Pro export do XML byla využita speciální knihovna blíže popsána v kapitole 5.3.1.

Velice užitečným rozšířením je dále možnost vykreslit námi editované diagramy a sítě do formátu *PostScript* (formát EPS). V tomto textu byly použity převážně obrázky pocházející z výsledné aplikace. Editovaný projekt se uloží do samostatné složky pod jeho jménem a všechny výstupy z editorů jsou uloženy v této složce. Tato funkcionalita byla přidána pomocí knihovny, kterou blíže popíšeme v kapitole 5.3.2.

5.2 Grafické prvky

Pro práci s grafickými prvky na editačním plátně bylo použito různých technik. Jedna byla aplikována pro detekci objektu pod ukazatelem a druhá pro nalezení hrany pod kurzorem.

5.2.1 Detekce objektu pod kurzorem

Zjištění, zda se nachází nějaký objekt pod kurzorem, bylo velice jednoduché. Každý objekt má výpočet pro toto zjištění. Tento způsob získání objektu byl zvolen převážně kvůli jednoduchosti přidání nového tvaru.

$$\begin{aligned} isX &= |mouseX - objectX| \\ isY &= |mouseY - objectY| \end{aligned} \tag{5.1}$$

Kde (5.1):

- *ObjectX* a *objectY* jsou souřadnice X a Y pro bod objektu.
- *MouseX* a *mouseY* určují bod kurzoru.
- *IsX* a *isY* je vypočítaná vzdálenost na osách X a Y.

Výpočet je prováděn tak, že se vypočítá vzdálenost na osách mezi body a tato vzdálenost je prověřena, zda leží v daném objektu. Porovná se šířka a výška oproti vypočítaným vzdálenostem. Pokud bod kurzoru leží v těchto hranicích, vykoná se příslušná operace – například zvýraznění objektu.

Tento způsob je náročný na pamět, převážně kvůli neustálému volání při jakémkoliv pohybu myši. Pro tento nástroj však toto omezení nemá vliv na výkon nebo celkový chod (převážně díky jednoduchosti a rychlosti).

5.2.2 Výpočet vzdálenosti hrany od bodu

Na druhou stranu zjištění hrany pod kurzorem již je poněkud náročnější, převážně z toho důvodu, že hranu máme definovanou jako dvojici bodů. Výpočet pro takové zjištění je složitější a možností použití různých vzorců pro toto zjištění je mnoho.

Ve vyvíjeném nástroji jsou použity dvě metody zjištění vzdálenosti bodu od přímky. Jedna metoda je využita pro výpočet vzdálenosti kraje objektu k bodu a na základě toho je hrana zkrácena (použito převážně pro vykreslení šipky na konci hrany). Druhá metoda slouží pro výpočet bodu kurzoru a úsečky mezi dvěma body.

Výpočet kraje objektu k bodu

Před samotným výpočtem průsečíku okraje objektu a přímky z něj vycházející, je potřeba zjistit, v jakém sektoru se nachází druhý konec úsečky (střed druhého objektu). Pro zjištění sektoru jsou použity úhlopříčky objektu a to ve stylu, že do vzorců pro úhlopříčku dosadíme bod X, nebo bod Y a u výsledného bod prověříme, zda leží pod nebo nad touto úhlopříčkou. Pro výpočet bodu lze použít jednu z rovnic (5.2).

Výpočet, zda se bod nachází pod, nad a nebo na úhlopříčce:

$$resultY = \frac{YboduA - YboduB}{(XboduA - XboduB)(givenX - XboduA)} + YboduA \tag{5.2a}$$

$$resultX = \frac{givenY - YboduA}{\frac{YboduA - YboduB}{XboduA - XboduB}} + XboduA \tag{5.2b}$$

Kde pro (5.2):

- *ResultX* a *resultY* je výsledná souřednice bodu X, nebo Y.
- *XboduA* a *YboduA* znamená souřednice prvního bodu úhlopříčky.
- *XboduB* a *YboduB* určuje souřadnice druhého bodu úhlopříčky.
- *GivenX* a *givenY* jsou souřadnice X, nebo Y z daného bodu, zde je tento bod souřadnice středu druhého objektu.

Pro samotný výpočet bodu je použito shodnosti trojúhelníků s tím, že máme zadaný střed objektu, ke kterému se snažíme zjistit protínaný okraj a bod, odkud vychází tato úsečka. Proto je po vybrání sektoru zjištěn poměr stran pomocí podělení šířky, nebo výšky (záleží na sektoru) a vzdálenosti středu objektu od bodu na ose X, nebo Y (opět záleží na sektoru). Při zjištěném poměru stran se dopočítá druhá strana trojúhelníku a vynásobí se s tímto poměrem.

Výpočet bodu kurzoru a úsečky

Výpočet vzdálenosti kurzoru od úsečky je prováděn odlišným vzorcem. Při tomto výpočtu se nejdříve určí, zda se bod nachází v rozmezí dané přímky (tedy jestli bod leží pod a nebo nad úsečkou). Pro určení bodu vůči přímce byla použita rovnice (5.3).

$$h = \frac{(Px - Ax)(Bx - Ax) + (Py - Ay)(By - Ay)}{delkaUsecky} \tag{5.3}$$

V momentě kdy, výsledné h z rovnice (5.3) je v rozmezí $0 < h < 1$, se bod nachází pod a nebo nad touto úsečkou (nikoliv za nebo před úsečkou). Lze tedy vypočítat vzdálenost kurzoru od úsečky.

$$\frac{(Bx - Ax)(Py - Ay) - (Ax - Px)(By - Ay)}{delkaUsecky} \tag{5.4}$$

Pro rovnice (5.3) a (5.4) platí:

- Px a Py jsou souřadnice kurzoru.
- Ax a Ay reprezentují souřadnice bodu A úsečky AB.
- Bx a By slouží jako souřadnice bodu B na úsečce AB.

Vzorce pro výpočet polohy bodu vůči úsečce a pro finální výpočet vzdálenosti tohoto bodu od úsečky byly převzaty z [3].

5.3 Použité knihovny a ikony

Pro zjednodušení práce a možnosti zaměření se na důležitější části nástroje bylo použito několik knihoven pro urychlení vývoje. Tyto knihovny jsou volně ke stažení s přehlednou dokumentací. Jejich použití je jednoduché a rychlé. Aplikování těchto knihoven vedlo k urychlení vývoje důležitých koponent, nicméně tyto knihovny nejsou nutností pro výslednou aplikaci. Pokud by se ovšem odtranily, bylo by nutné tyto knihovny nahradit vlastní implemtnací těchto funkcí.

5.3.1 XML export

V případě expotu do XML bylo zvoleno ukládání veškerých grafů do speciální struktury, kterou lze jednoduše předat knihovně s názvem *XStream* [1]. Tato knihovna se postará o bezproblémovou serializaci a deserializaci objektu do a z XML.

Výhodou této knihovny je jednoduchost použití jak při serializaci, tak deserializaci. V případě načtení souboru je práce s touto knihovnou ještě snažší. Další výhodou knihovny

[1] Domovskou adresu pro tuto knihovnu lze nalézt na adrese: http://xstream.codehaus.org/.

je možnost nastavit aliasy pro XML tagy, takže je možné upravit XML soubor, který bude používaný v tomto nástroji pro další potřeby. Pokud například bude nutné soubory exportované tímto nástrojem upravit v další aplikaci, která dokáže číst a vytvářet XML soubory, je možné změniz vstup a výstup této knihovny tak, že nebude potřeba upravovat načítání a práci s těmito soubory v jiné aplikaci.

Použitím této knihovny bylo docíleno bezchybného, rychlého a jednoduchého vytvoření souboru pro ukládání editovaného projektu.

5.3.2 PostScript export

Export do souboru PostScript nebyl nutností pro tuto aplikaci, ale bylo zvoleno, že export do tohoto formátu bude velice užitečný a nástroj tím získá silnou funcionalitu.

Pro export do formátu EPS bylo vyzkoušeno několik nástrojů, žádný z nich ovšem nebyl tak jednoduchý na použití a nepodporoval tolik funkcí jako tomu bylo u nástroje *EPS Graphics* [2].

Tento nástroj vyčnívá nad konkurencí převážně v množství implementovaných funkcí. Byl vybrán převážně kvůli tomu, že veškerá grafika je tisknuta pomocí *Graphics2D* a tato knihovna podporuje většinu funkcí, které jsou používány právě v *Graphics2D*. [3]

V celém tomto dokumentu byly použity obrázky, které byly exportovány pomocí této knihovny, umožňující velice rychlé a užitečné rozšíření tohoto nástroje.

5.3.3 Použité ikony

V aplikaci bylo použito několika ikon pro zpřehlednění a odlehčení monotónnosti designu. Přidáním těchto ikon bylo umožněno i méně zručným uživatelům jednoduše používat a osvojit si práci s tímto nástrojem.

Ikony byly vybrány z galerie Open Icon Library[4], tyto ikony jsou volně ke stažení, a pokud tento nástroj nebude využíván pro účely zisku, neměl by být žádný problém s licencemi.

[2] Dokumentaci a knihovnu samotnou lze najít na adrese: http://www.abeel.be/epsgraphics/.

[3] Pro bližší obeznámení implementovaných funkcí je možné nahlédnout do rozsáhlé API dokumentace na stránce: http://epsgraphics.sourceforge.net/api/1.2/doc/.

[4] Adresa této galerie je: http://openiconlibrary.sourceforge.net/.

Kapitola 6

Uživatelské testování

Záměrem této kapitoly je předvést výsledky uživatelského testování. Budou zde představeny návrhy od možných zákazníků a potencionálních uživatelů tohoto nástroje.

6.1 Nezkušený uživatel

Nátroj byl představen nejdříve uživateli, který nemá žádné zkušenosti s vytvářením programů, ani nikdy nebyl zapojen do vývoje programu. Tento test má dokázat, zda je nástroj použitelný a nemá některou chybu znemožňující práci s tímto nástrojem.

6.1.1 Průběh představení

Nezkušenému uživateli bylo dáno za úkol vymyslet program, který by mu pomáhal při práci. Tento program byl poté představen a popsán programátorovi. Při tomto testu nebyla nalezena žádná chyba a nedostatek. Uživatel byl spokojený s rychlostí, s jakou se výsledný program rýsoval. Převážně kvůli tomu, že si mohl okamžitě vybavit věci, které již byly nachystány a které bude potřeba vymyslet.

V tomto případě byl nástroj byl tedy velice přínosný a bylo možné postoupit k předání nástroje více zkušenému uživateli.

6.2 Zkušenější uživatel

Tento uživatel již použil diagram případů užití a rozumí návrhu aplikace. Bylo po něm požadováno, aby předvedl, zda by zvládl v této aplikaci představit svoji vizi programátorovi a zda aplikaci neschází některý nutný prvek potřebný pro jednoduché ovládání.

6.2.1 Průběh a výsledky uživatelské práce

Při sledování práce uživatele bylo zjišťěno, že jeho učební křivka je velice strmá. To znamená, že návrh aplikace byl velice dobrý a uživatel si rychle zvykl na práci a po chvilce zkoušení nástroje byl schopný vytvořit vlastní diagram případů užití.

Během tvoření tohoto diagramu bylo ovšem zjišťěno, že aplikaci schází možnost okamžitého zrušení přidávání nových objektů na editační plátno. Tento návrh byl tedy poznamenán a později implementován.

Uživatel neměl žádnou další výtku k nástroji kromě požadavku na vytvoření nápovědy a jednoduchého popsání jednotlivých částí nástroje, což by mělo být buď v nápovědě nebo v uživatelské příručce.

6.3 Programátor

V poslední části byl nástroj představen programátorovi, kterému bylo navrhnuto, aby vytvořil jednoduchý program. Vytvořil diagram případů užití, poté přidal pár rozšiřujích metod a tyto metody popsal pomocí objektově orientovaných Petriho sítí. Převážně bylo sledováno, zda nechybí některá důležitá funkce a není příliš omezovaný a jestli není potřeba příliš představovat tento nástroj.

6.3.1 Průběh testování

Tento uživatel rychle pochopil základní ovládací prvky nástroje, ocenil různé možnosti otevírání a zavírání jednotlivých projektů. Kladně také zhodnotil provázání mezi diagramem případů užití a diagramem tříd. Po vysvětlení, jak spojit různé objekty napříč diagramy, byl s touto funkcí spokojený.

Po představení, jak fungují objektově orientované Petriho sítě, ocenil jednochusot a rychlost vytváření jednotlivých částí aplikace a také svižnost popisu metod pomocí Petriho sítí.

6.3.2 Vyhodnocení programátorovy práce

Uživatel byl s nástrojem spokojený a ocenil by další rozšíření v podobě exportu do zdrojových kódů nebo simulace Petriho sítí.

6.4 Výsledky testování

Po otestování třemi různými uživateli bylo objeveno několik chyb – například v podobě okamžitého zrušení přidávaných objektů a jejich spojování. Jinak byli uživatelé spokojeni s jednoduchostí a snadným ovládáním nástroje, což znamená splnění požadované podmínky na intuitivnost a snadnost použití i při automatickém generování – jak bylo specifikováno v úvodu této práce.

Kapitola 7

Závěr

V této kapitole se budeme postupně věnovat navrhovaným možným rozšířením, kterým by bylo vhodné tento nástroj vylepšit. Poté shrneme vytvořený nástroj, jeho přínos a aktuální stav. Také popíšeme některé známé chyby nebo nedostatky.

7.1 Budoucnost aplikace

Tato kapitola představuje navrhované rozšíření pro vytvořený nástroj. Nástroj je bez těchto rozšíření funkční, nicméně přidání dalších funkcí a možností by z tohoto nástroje udělalo pravděpodobně robustní nástroj.

Rozšíření zde navrhnutá není nutné dodržet při další implementaci a přidávání funkcí a vylepšení do tohoto nástroje. Nicméně jsou logicky vybrány a z pohledu momentálního stavu aplikace by znamenaly velmi mnoho. Do nástroje je samozřejmě možné přidat mnoho dalších funkcí.

Historie akcí

Jak již bylo zmíněno v předešlých kapitolách, jistě by se hodilo ukládat postupný chod akcí, přes které by bylo možné postupně kolovat, jak dozadu, tak dopředu, což by vedlo ke snažší editaci a pohodlnější práci s aplikací. Absence této funkce momentálně neznamená nepoužitelnost aplikace, pouze ji dělá mírně těžkopádnou. Toto rozšíření je nyní považováno za nejdůležitější a při dalším rozšíření nástroje by bylo vhodné tuto úpravu implementovat jako první.

Grafický design a editace designu

Grafický design a momentální vzhled nástroje není špatný, nebo nikterak nepoužitelný, je pouze jednoduchý a velice monotónní. Z tohoto důvodu by bylo vhodné přidat možnost změny vzhledu aplikace, ať už nastavením uživatelského rozhraní, nebo celkového vzhled objektů na editačním plátně. Aplikace by díky tomuto rozšíření byla více přívětivější k uživateli.

Možnost exportu Petriho sítě

Pro lepší komunikaci mezi tímto nástrojem a nástroji, které se soustředují například na interpretaci Petriho sítí, by bylo užitečné, kdyby nástroj mohl objektově orientované Petriho

sítě exportovat do specifického formátu, jako například *PNTalk* [1].

Simulace Petriho sítí

Pokud by se využilo předešlého rozšíření (možnost exportu Petriho sítě) a propojení s dalším nástrojem, který by se staral o simulaci Petriho sítí vytvořených v tomto nástroji, znamenalo by to značné urychlení vývoje aplikací. Kdyby se od zákazníka převzaly vstupy a výstupy pro testovací případy a ty by se porovnaly s výstupem z tohoto simulátoru, znamenalo by to velice rychlé zjištění chyb, na základě kterých by se dala upravit Petriho síť.

Vytvoření zdrojových kódů z diagramů a Petriho sítí

Po vzoru nástrojů Umbrello 2.1 a Enterprise Architektu 2.2 by bylo užitečné mít možnost vytvořit zdrojové kódy pro zvolený jazyk z vytvořených diagramů. Zapojení objektově orientovaných sítí do tohoto exportu by znamenalo vytvoření plně funkční aplikace. Díky tomuto rozšíření by se tento nástroj mohl zařadit na úroveň současných komerčních i nekomerčních nástrojů. Oproti ostatním nástrojům má vyvíjená aplikace již velikou výhodu v podobě objektově orientovaných Petriho sítí a jednoduchých editorů.

Shrnutí rozšíření

Bylo navrhnuto a objeveno mnoho možností, jak tento nástroj rozšířit a udělat z něj silný nástroj na poli UML editorů. V případě postupné implementace navrhovaných rozšíření by po dosažení *Vytvoření zdrojových kódů z diagramů a Petriho sítí* z tohoto jednoduchého editoru vytvořilo silnou aplikaci, která by se mohla srovnávat s aplikacemi jako *Umbrello*, *Enterprise Architect*, *Dia* a mnohých dalších.

7.2 Závěrečné zhodnocení aplikace

Bez ohledu na rozšíření probraná v předešlé kapitole 7.1 je aplikace dobře použitelná a vhodná pro vytvoření návrhu vývoje.

Tento nástroj má výhody oproti ostatním aplikacím převážně díky jednoduchému a snadnému ovládání. Minimum nástrojů na trhu momentálně nabízí spojení diagramu případů užití s diagramem tříd. Nástroje, které toto umožňují však nemají možnost popsat třídy pomocí objektově orientovaných Petriho sítí. Tento způsob popisu chování třídy a popisu reakcí na odchycené zprávy je unikátní převážně díky rychlosti, jakou lze takovou třídu nebo metodu popsat.

Oproti nástrojům zmiňovaným v této práci má vyvíjená aplikace výhodu díky provázanosti UML částí, kdy je možné vytvořit třídu, tu popsat a v momentě, kdy je potřeba tuto třídu použít jinak (jako například odlišný případ užití), tento nástroj změnu dokáže provést takřka okamžitě. Tento pohled na vytváření tříd a provázání s akcemi je dobrý v případě zapojení agilních metodik vývoje. Je možné vytvořit třídy a o jejich použití se postará zákazník, který nemusí být zkušený v programování, pouze je na něj kladena podmínka porozumění diagramu případů užití.

Z práce uživatelů s tímto nástrojem (viz kapitola 6) vyplývá, že nástroj je jednoduchý na používání a snadno pochopitelný. S tímto záměrem byla vyvíjená aplikace od začátku

[1] Bližší informace o tomto jazyku a o projektu PNTalk lze nalézt na stránkách: http://perchta.fit.vutbr.cz/pntalk2k.

tvořena a tento úkol také splňuje. Nástroj má některé nedostatky v podobě nemožnosti vrátit zpět provedenou akci, nebo neidentifikaci, zda byl program uložen. Tyto nevýhody ovšem nejsou limitující a neomezují uživatele v používání aplikace.

Přínosná je možnost případného rozšíření tohoto nástroje o několik dalsích funkcí. Při vytvoření navrhovaných rozšíření v kapitole 7.1 je možné z tohoto nástroje udělat užitečnou aplikaci na poli editorů, nad kterými aplikace již momentálně má výhodu díky možnosti popsání metod a tříd pomocí objektově orientovaných Petriho sítí.

Literatura

[1] *Enterprise Architect User Guide*. Accessed: 12.04.2014.

[2] *Umbrello UML Modeller Handbook*. Accessed: 12.04.2014.

[3] Andrew S. Glassner, James Arvo: Graphics Gems II. In *The Graphics Gems Series A Collection of Practical Techniques for the Computer Graphics Programmer*, Academic Press, Inc., 1991, iSBN 0-12-059756-X.

[4] janoušek, V.: *Modelování objektů petriho sítěmi*. Dizertační práce, 2008.

[5] Kettenis, J.: *Getting Started With UML Class Modeling*. Oracle, May 2007, accessed: 22.03.2014.

[6] Kettenis, J.: *Getting Started With Use Case Modeling*. Oracle, May 2007, accessed: 22.03.2014.

[7] Kočí, R.; Janoušek, V.: Object Oriented Petri Nets in Software Development and Deployment. In *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, Xpert Publishing Services, 2013, ISBN 978-1-61208-304-9, s. 485–490.
URL http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10381

[8] Radek Kočí, Vladimír Janoušek, and František Zbořil, jr. : Object Oriented Petri Nets – Modelling Techniques Case Study. In *International Journal of Simulation Systems, Science and Technology*, *3*, ročník 10, May 2009.

Příloha A

Obsah CD