VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO PODPORU VÝVOJE SOFTWAROVÝCH SYSTÉMŮ

TOOL FOR SOFTWARE SYSTEMS DESIGN

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. RADEK CRLÍK**
AUTHOR

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

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Crlík Radek, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Nástroj pro podporu vývoje softwarových systémů**
Tool for Software Systems Design
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s problematikou nástrojů pro podporu vývoje softwarových systémů.
2. Prostudujte koncept jazyka UML a formalismu Objektově orientované Petriho sítě (OOPN). Z jazyka UML se zaměřte zejména na diagramy případů užití a tříd.
3. Navrhněte nástroj umožňující návrh softwarového systému s využitím diagramů případů užití pro popis chování systému z hlediska uživatele, diagramů tříd pro popis struktury a formalismu OOPN pro popis chování tříd. Nástroj musí umožnit export a import modelů.
4. Navržený nástroj implementujte v jazyce Java.
5. Vytvořte manuál a sadu příkladů demonstrující možnosti nástroje.
6. Diskutujte dosažené výsledky a navrhněte možná rozšíření nástroje. Výsledky také prezentujte formou posteru.

Literatura:

- V. Janoušek: Modelování objektů Petriho sítěmi. Disertační práce. Brno, 1998.
- C. Larman: Applying UML and Patterns. Prentice Hall, 2005.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Pro tvorbu kvalitního softwarového systému je potřeba takový projekt dobře analyzovat, navrhnout, naprogramovat a otestovat. Celý proces se pak souhrnně označuje jako životní cyklus softwaru a zabývá se jím softwarové inženýrství. Dnes existuje celá řada nástrojů, které tyto procesy ulehčují. Pro analýzu a návrh softwaru se v praxi osvědčil jazyk UML. Dovoluje popsat různé úrovně softwaru pomocí grafických diagramů pro jejich lepší pochopení. Některé je pak možné převést na kód v požadovaném programovacím jazyce. Problémem je pak udržování diagramů, kdy se tak v pozdějších fázích projektu ztrácí jejich význam. Tento problém se snaží odstranit tzv. Model-Driven Development, kdy programátor pracuje jen s přesně definovanými modely ze kterých je možné automaticky generovat programový kód, který se ale již nemusí ručně upravovat. Bohužel tento přístup není univerzální. Tato práce se zaměřuje na tvorbu nástroje, který umí pracovat s diagramem případů užití, diagramem tříd a objektově-orientovanými Petriho sítěmi. Nástroj by měl zvládat jejich tvorbu a základní synchronizaci informací mezi diagramy a tak ulehčit návrh systémů.

Abstract

To be able to create quality software system, one need to analyse it well, design, program and test it. The whole process is called software life-cycle and is studied by software engineering. Today, there are many tools making this process easier. For analysing and designing software UML language became favourite. It enables programmers to describe different aspects of software by graphical diagrams and enables them to comprehend them better. Some of them can be translated into source code in chosen programming language. Problem is maintaining those diagrams during later phases when source code was already generated and is used exclusively. This problem is trying to be solved by Model-Driven Development, where programmer is working with well-defined models that can be automatically transformed into the source code that don't have to be edited by hand. Unfortunately this approach is not universal. This work tries to design and create tool that can work with use case diagrams, class diagrams and object-oriented Petri nets. This tool should allow designing those diagrams and be able to synchronise information between them to make the software design easier.

Klíčová slova

Software design, modelování, návrh, model-driven development

Keywords

Software design, modelling, design, model-driven development

Citace

CRLÍK, Radek. *Nástroj pro podporu vývoje softwarových systémů.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Nástroj pro podporu vývoje softwarových systémů

Prohlášení

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

.....
Radek Crlík
22. května 2017

Poděkování

Děkuji za odborné vedení Ing. Radku Kočímu, Ph.D. při psaní této práce.

Obsah

Kapitola 1

Úvod

Programování se dnes věnuje stále více lidí. Částečně je to způsobeno tím, že stále více odvětví využívá více a více elektroniky a počítačů a je tak nutné se je naučit používat. Částečně také z důvodu, že nástroje pro vývoj softwaru jsou stále jednodušší a je velké množství materiálů, které usnadňují s programováním začít.

Pro mnoho lidí je to zároveň zábava, protože programování lze přirovnat k tvůrčí činnosti. Zároveň každý problém při programování vyžaduje trochu jiný přístup nebo řešení a proto není tato činnost monotónní a také přispívá k tomu, že stále baví více lidí.

Bohužel ve vývoji větších softwarových produktů není samotné programování jediná činnost, která je potřeba provést k úspěšnému dokončení projektu. Velké projekty ve většině případů programuje větší počet lidí, aby mohl být projekt dokončen v rozumném čase. Je tak nutné analyzovat produkt, správně naplánovat činnost jednotlivých účastníků, kteří se na tvorbě podílejí, důkladně namodelovat návrh produktu a poté začít kooperativně programovat. Na konci je také důležité vytvořený systém dostatečně otestovat. Všechny tyto činnosti tak často zaberou daleko více času než samotné programování. Koordinací nebo plánováním těchto činností se zabývají teorie životního cyklu vývoje software a softwarové inženýrství.

Pro modelování softwarových částí je často používán grafický jazyk UML[1], který obsahuje standardizované diagramy sloužící pro popis různých softwarových částí od samotného kódu až po organizaci hardwaru potřebný pro provoz výsledného systému. Podobná dokumentace je potřebná i v jiných průmyslových odvětvích jako například v architektuře nebo kovoobrábění, kde se používají technické výkresy. Podobné diagramy jazyka UML mohou sloužit jako základ pro komunikaci programovacího týmu a pomoci tak překlenout pouze abstraktní myšlenky a nápady každého programátora.

I když se jazyk UML vyvinul v plnohodnotný nástroj pro modelování softwarových částí, stále je ve většině případů nutné tyto diagramy převést do kódu, který lze přeložit nebo spustit. Dnes je velké množství nástrojů, které dokáží z diagramů generovat kód v požadovaném programovacím jazyce, který programátor jen doplní nebo upraví. Doba vývoje se tak zkrátí a umožní programátorovi soustředit se jen na samotný hlavní kód a jeho vývoj. Problém je, pokud je chyba přímo ve specifikaci softwaru nebo v návrhu na kterou se přijde až v čase, kdy je software hotový a začíná se používat. Odstranit takovou vadu pak stojí nejvíce času a úsilí. Proto se lidé snaží vložit větší pozornost specifikaci návrhu a z tohoto důvodu tato část tvorby produktu trvá nejdelší dobu. Je tak vhodné mít

[1] Unified Modeling Language

nástroje, které tuto etapu tvorby softwaru ulehčují a zároveň dokáží eliminovat množství chyb, které jsou později drahé na odstranění.

Tato práce se zabývá návrhem a tvorbou nástroje, který umožňuje analyzovat požadavky pomocí grafických diagramů a pomocí diagramů také popsat chování systému bez nutnosti toto chování popisovat čistě pomocí zdrojového kódu. Využívá několik typů diagramů, kde každý z nich nějak upřesňuje požadavky nebo chování systému. Jako počáteční diagram je použit diagram případů užití, který vychází z požadavků zákazníka. Popisuje co by měl výsledný systém umět, ale neříká nic o tom, jak by to měl provádět. Tento diagram je transformován na diagram tříd. Pro popis chování těchto tříd je poté použita objektově-orientovaná Petriho síť. Ta je výhodná z toho důvodu, že je dostatečně formální a má dostatečné vyjadřovací schopnosti nutné pro přímé provádění nebo simulaci sítě, takže není třeba generovat a dále upravovat zdrojový kód. Podobně jako je to možné například u konečných automatů. Takový přístup modelování vychází z článku [16]. Lze tak pracovat na úrovni model-centrického přístupu k tvorbě softwaru. Tento přístup je poté popsán v druhé kapitole . Dále by navrhovaný nástroj měl být multiplatformní a co nejvíce zjednodušovat tvorbu těchto diagramů.

Druhá kapitola této práce popisuje běžně používané přístupy pro vývoj softwaru a jeho cykly. Je popsán klasický vodopádový model a jsou naznačeny principy iterativních modelů. Popis se zaměřuje na jejich výhody a nevýhody. Třetí kapitola obsahuje výčet některých nástrojů sloužících pro tvorbu diagramů a modelů různých částí softwaru a také jejich návaznost nebo podporu UML. Jsou zde porovnány některé jejich vlastnosti, výhody a nevýhody. Čtvrtá kapitola se věnuje popisu jazyka UML a hlavně jeho diagramům případů užití a diagramů tříd, s kterými se pracuje v navrhovaném nástroji. Pátá kapitola popisuje základ Petriho sítí a je zde naznačen formalizmus objektově-orientovaných Petriho sítí, s kterým také pracuje navrhovaný nástroj v této práci. Šestá kapitola se věnuje popisu požadavků na navrhovanou aplikaci a popisuje její možnosti. Sedmá kapitola navíc popisuje použité technologie a implementaci programu a v závěru se věnuje testovaní vytvořené aplikace, její použitelnosti a hodnotí jednoduchost použití nástroje za pomocí zpětné vazby uživatelů.

Kapitola 2

Cyklus vývoje softwaru

Cyklem vývoje softwaru se zabývá softwarové inženýrství. Protože při vývoji softwaru nemáme data potřebná pro plánování [21] podobně jako v jiných odvětvích jako například v architektuře nebo zmíněném obrábění kovů, musíme se řídit zkušenostmi z jiných podobných softwarových projektů. Co je také rozdílné v softwarovém inženýrství oproti jiným odvětvím je to, že technologie se mění velmi rychle a to i v samotné fázi realizaci projektu. Navíc konkurence na trhu je dnes velká a nutí tvořit nový software čím dál tím rychleji. Potřebný čas na vývoj softwaru je tak velice neodhadnutelný, zčásti protože většina softwaru je jiná a nelze tak odhadnout potřebný čas porovnáním s jakýmkoliv jiným již dokončeným projektem. Software je komplexní produkt a často záleží i na prostředí v kterém běží. V případě nalezení chyby tak může trvat od několika hodin až po několik dní ji správně lokalizovat a opravit. Čas pro lokalizaci a opravení chyby navíc záleží i na zkušenostech programátora a ty mohou být mezi členy vývojového týmu různé.

Složitost softwaru však není záležitost jen poslední doby [21]. Požadavek na formální proces vývoje se objevil již před desetiletími. Nastavit řád vývoji produktu se tak snaží některé modely pro cyklus vývoje softwaru. V následující části jsou některé popsány.

2.1 Vodopádový model

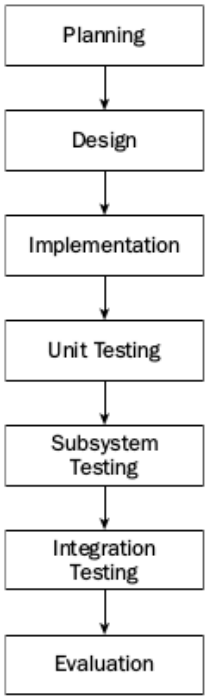
Jedná se o klasický ukázkový model, který stojí na předpokladu, že projekt nebo produkt může být vytvořen podobně jako pokrm podle receptu. To je seznam kroků, které když se postupně a správně splní, na konci dostaneme správný produkt podobně jako například výborný dort při pečení. Zobrazení kroků obecného vodopádového modelu je vidět na obrázku 2.1.

Výhoda vodopádového modelu je jeho jednoduchost. Může být vhodný pro velice malé a krátké projekty. Ale u větších než takto malých projektů to má závažný problém, protože často na začátku nejsou dostupné všechny požadavky nebo celá specifikace produktu a je potřeba ji postupně zpřesňovat. Někdy také zákazník, který si nechává produkt vytvořit sám nedokáže říct, jak by měl do detailu vypadat a různé požadavky se objevují i uprostřed jeho vývoje. Toto však může způsobit i změna konkurence na trhu, kdy je prostě nutné zavést novou funkci, aby bylo možné držet krok s konkurenčním programem.

Vodopádový model se tak prakticky nepoužívá. I když se zavede možnost, že následující fáze vodopádového modelu se mohou překrývat, stále je velké omezení nemožnost vracet se zpět o několik fází. Chyba v návrhu pak znamená velký problém, pokud se na ni přijde

těsně před vydáním produktu nebo někde v jiné pozdější fázi a může tak vést k nezdaru celého projektu a velkým finančním ztrátám.

Obrázek 2.1: Vodopádový model vývoje softwaru (zdroj: [21])



2.2 Iterativní modely

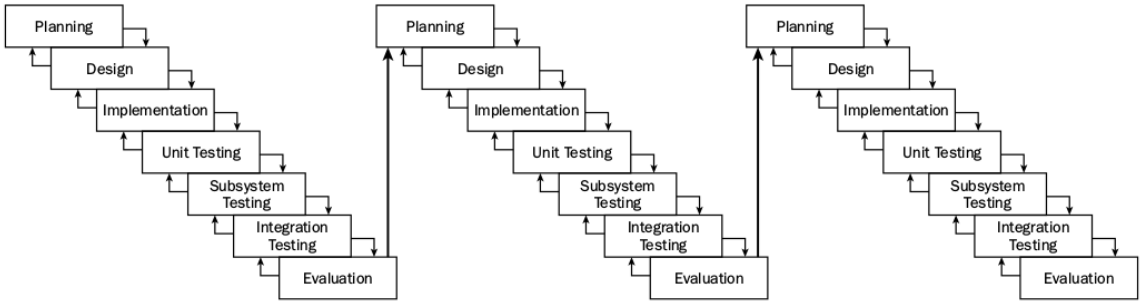
Skupina iterativních modelů se snaží odstranit nedostatky vodopádového modelu. Hlavním z nich je právě linearita vývoje. Hlavní myšlenka je rozdělit projekt na menší částí, které již mohou být řízeny samostatně pomocí vodopádového modelu. Přístup je vidět na obrázku 2.2. U každé takové části se dají samostatně odhadnout rizika, případně nedostatky návrhu. To také umožňuje se snadno věnovat nejdříve věcem, které představují největší riziko. To zabraňuje tomu, aby se co nejméně chyb dostalo až do poslední fáze celého projektu, což je nejhorší scénář pro vodopádový model. Díky rozdělení vývoje na časově omezené iterace se může samotný tým učit a postupně zlepšovat postupy v dalších iteracích a přizpůsobovat vývojový proces samotný pro daný projekt. Na konci každé iterace je pak v ideálním případě hotová nová vlastnost produktu, která je funkční a nějakým způsobem použitelná. Zákazník zde vidí produkt daleko dříve.

Tým, ale i zákazník, tak může hned zhodnotit danou vlastnost a případně vznést připomínky. To nutí řešit problémy a nedostatky okamžitě, místo až za několik měsíců před vydáním produktu. Jako další výhoda je, že pokud je taková vlastnost v pořádku, zákazník ji může hned začít využívat a může mu nějakým způsobem generovat zisk.

Ovšem ani tento přístup nemá samé výhody. Někdy není snadné naplánovat dostatečně malé nebo krátké iterace, aby se mohly využít všechny výhody této metody. Při vetším počtu lidí uvnitř týmu, který se podílí na vývoji softwarového produktu, může být velká režie s plánováním a koordinací iterací. To je způsobeno tím, že často několik týmu pracuje na různých částech projektu a pracují paralelně. Není však neobvyklé, že každý tým pracuje

jinak rychle. A zatímco jeden tým již dokončil svoji část, než může začít další činnost musí počkat, než druhý tým dokončí své úpravy.

Obrázek 2.2: Iterativní model vývoje softwaru (zdroj: [21])



2.3 Dual-track agile přístupy

Hlavní myšlenka iterativních přístupů je, že po každé iteraci je dostupná hotová část nějaké funkcionality. I když sama o sobě nemusí být dostatečná pro plné fungování systému, ale je třeba přidat další vlastnosti v dalších iteracích aby se celek dal považovat za plnohodnotný. Taková část pak již má produkční kvalitu.

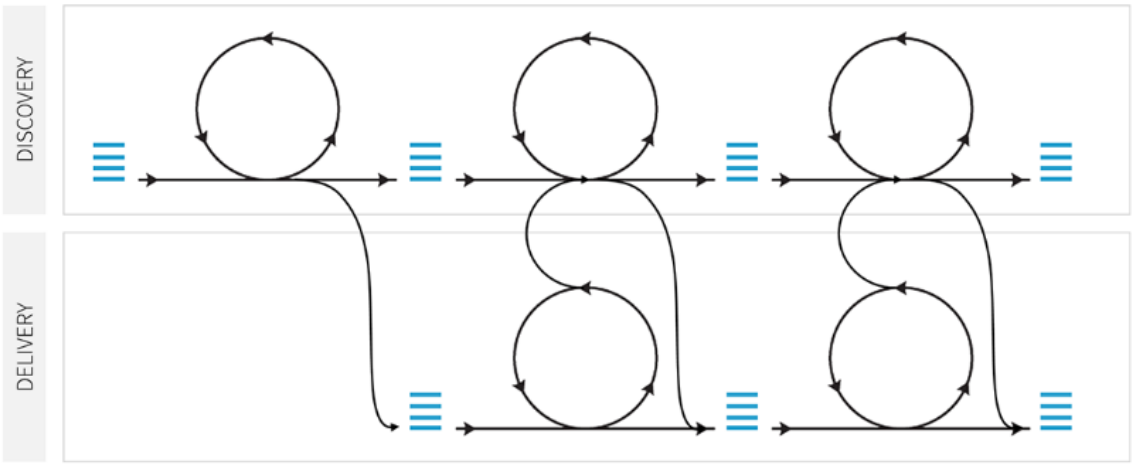
Problém však nastává u projektů, kde zákazník nedokáže přesně formulovat zadání a sám si není jistý, jak by měla nová vlastnost v systému fungovat. Pak se často stává, že bude chtít požadavky upravit poté, co si novou část vyzkouší a zjistí, co by vlastně ještě potřeboval přidat. V tomto případě vývoj produkčně kvalitního kódu stojí zbytečný čas a také peníze.

Přístupy extrémního programování pak doporučují vytvářet prototypy, které nemají potřebnou funkcionalitu a kód není psán velmi kvalitně, ale zato rychle. Takový prototyp si může zákazník prohlédnout a rychle zjistit, jestli mu to vyhovuje nebo je třeba něco upravit. Až po tomto odsouhlasení nebo případných dalších úpravách zadání se vytváří plnohodnotný kód.

Dual-track modely pak spojují iterativní vývoj s vytvářením prototypů. Myšlenka je taková, že můžeme kontinuálně objevovat požadavky zákazníka a validovat je pomocí prototypů a testování. Jakmile je požadavek přezkoumán a potvrzen, je zařazen mezi vlastnosti, které by se měli do produktu dodat (tzv. *backlog*). Funguje to tak, že se vytvoří dvě paralelní větve iterativního modelu. V prvním se nejdříve plánují iterace pro vývoj hrubých prototypů (kterých může být klidně více a představují tak více možných řešení), kde se však neztrácí čas psaním kvalitního kódu, ale je zde cílem, aby zákazník mohl v krátkém čase vidět, jak daná část bude nebo by mohla vypadat. Až poté, co si novou část zákazník prohlédne, případně doplní nebo upraví své požadavky, tak se v druhé větvi naplánuje iterace, kde by tato funkcionalita měla být naprogramována plnohodnotným způsobem vhodným pro produkční systém. Tento způsob omezuje čas, který se ztratí vývojem vlastnosti systému, který když zákazník na konci vidí často zjistí, že by ji potřeboval jinak. Ale s takovou změnou požadavku, jakmile je již část systému dokončena, často dochází k přepracování celé této části programu. Díky tomu, že se systém neupravuje tak často, také netrpí zastaráváním kódu a není tak nutné často provádět refaktorizaci. To je zvlášť problém projektů

běžících v delším časovém úseku. Kde takové zastarávání značně ztěžuje přidávání nových vlastností a zvyšuje pravděpodobnost zanášení chyb do systému.

Obrázek 2.3: Dual-track scrum model (zdroj: [22])



Přístup Dual-track agile je vidět na obrázku 2.3. Jednotlivé smyčky představují jednotlivé iterace. Tento přístup také dovoluje mít dva separátní týmy. Kde se jeden specializuje na komunikaci se zákazníkem, zjišťování jeho potřeb, vytváření prototypů a validaci jeho požadavků (v části *Discovery*). V čele tohoto týmu může stát produktový manažer, který poté vytváří seznam nových vlastností, které by se měli do produktu dodat spolu s jejich prioritami. Z obrázku je vidět, že po každé iteraci přechází tým na další iteraci, ale výstupy dokončené iterace putují dále do iterace druhého týmu. Druhý tým je specializovaný na tvorbu kvalitního kódu a nasazování kódu do produkčního systému (v části *Delivery*). Tento přístup o něco více snižuje rizika při vývoji a neztrácí se čas programováním částí, které se nakonec zákazník rozhodne odstranit. Ale zároveň zachovává disciplínu při vývoji iterativním způsobem. V praxi, při použití iterativních metod, jsou často na začátku nových iterací prováděny porady ohledně práce, která by se v následující iteraci měla provádět a lidé jsou tak seznamování s tím, co by se mělo realizovat. Jako vedlejší produkt těchto Dual-track agile přístupů je zkrácení těchto porad [9]. A to díky tomu, že do iterací v části *Delivery* vstupují již ověřené a přesně definované požadavky, které dokáže vývojový tým bez větších problémů převzít a pochopit.

Konkrétním příkladem tohoto přístupu pak může být *Dual-Track Scrum* metodika. Která takto upravuje metodiku *Scrum* [4].

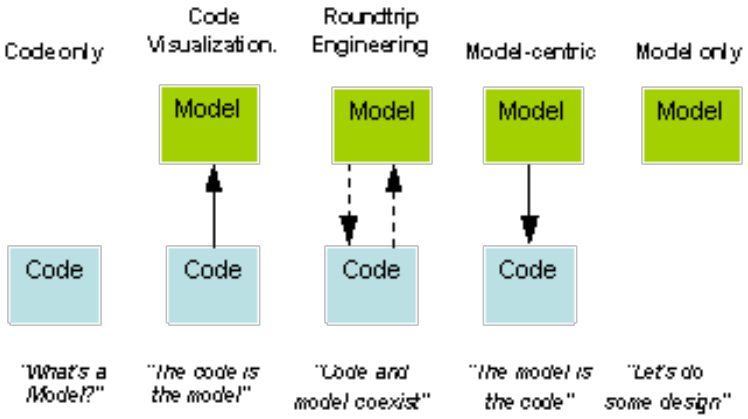
2.4 Model-driven architecture

Ve fázi návrhu se často tvoří různé modely realizovaného systému, aby všichni členové snadno pochopili zamýšlený záměr a funkci. Modely a modelování pak hrají v *Model-driven*

architecture (MDA) zásadní vliv. Proto před samotným popisem MDA se hodí zmínit používané přístupy pro modelování v tvorbě software [8].

Dá se identifikovat 5 hlavních přístupů, jak mohou být modely využity v procesu tvorby softwaru. Ty jsou zobrazeny na obrázku 2.4 a dále popsány.

Obrázek 2.4: Přístupy modelování v procesu tvorby softwaru (zdroj: [8])



První přístup nazývaný *code-only*, používaný mnoha vývojáři nebo malými týmy je orientace pouze na zdrojový kód. Nevytváří se separátní modely nebo diagramy (i když zdrojový kód se v některých publikacích také označuje za model systému). Tento přístup je vhodný u malých projektů nebo týmů, kde je snadná komunikace a vývojáři si poznámky ohledně kódu mohou vysvětlit ústně mezi sebou. Horší je to u větších projektů nebo tam, kde se vývojáři během projektu mění. Pro nově příchozí pak může být složité se zorientovat a zasvěcení od ostatních členů týmu vyžaduje čas a další režii.

Druhý přístup nazvaný *code visualization* využívá nástroje, které ze zdrojových kódů dokáží vytvořit jednoduchou vizualizaci (například hierarchii tříd nebo hypertextové stránky) pro snadnější pochopení kódu. Tyto vizualizace jsou však statické, slouží pouze pro čtení a nedovolují žádné úpravy. Oproti předchozímu přístupu tak alespoň urychlují seznámení s novým kódem. Jako příklad nástrojů, které dovolují ze zdrojového kódu vygenerovat takovou dokumentaci můžou být nástroje Doxygen, PHPDocumentor nebo JavaDoc.

Třetí přístup se nazývá *round-trip engineering* [7]. To je způsob, kdy vývojové prostředí udržuje systém ve dvou reprezentacích – v textové a grafické. To pak dovoluje upravovat modely a tyto změny automaticky promítat do výsledného zdrojového kódu. A zároveň při úpravě kódu se mohou zpětně měnit vytvořené modely a diagramy. Hlavním problémem tohoto přístupu je pak udržení informace co je vygenerovaný zdrojový kód a co je zdrojový kód doplněný programátorem. Jedna z možností je udržovat ve zdrojovém kódu speciální značky (například ve formě speciálních komentářů), které tyto části od sebe jednoznačně oddělují a je možné je strojově zpracovávat.

Čtvrtý přístup, tzv. *model-centrický* již v modelu obsahuje dostatek informací, aby z něj bylo možné vygenerovat úplný zdrojový kód, který se dá přeložit a spustit bez zásahu programátora. Takový model již může obsahovat informace například o tom jak má být uložený do databáze, business logiku a spoustu jiných věcí. Nástroje podporující tento

přístup se však často omezují na úzký profil aplikací, které je možné v nich vytvářet. V každém případě je však primárním artefaktem pouze model.

Poslední přístup *model-only* je pak nejextrémnější přístup, kdy se pracuje pouze s modelem a negeneruje se žádný zdrojový kód, který by bylo nutné překládat. To mohou dovolovat například nástroje pro tvorbu konečných automatů, které je dovolují v aplikaci okamžitě simulovat. Stejně jako objektově-orientované Petriho sítě používané v navrhovaném nástroji této práce. Takové sítě jsou dostatečně formální a mají možnost doplnit chování sítě pomocí výrazů přímo v diagramu sítě. Místa sítě poté reprezentují stav systému a přechody sítě umí vykonávat všechny akce pro změnu takového stavu. Protože tedy mají vyjadřovací schopnost jako programovací jazyky, je možné je do takového programovacího jazyka automaticky konvertovat. Případně je možné takovou síť přímo simulovat. Pak dokonce není nutné generovat žádný zdrojový kód, který by bylo nutné překládat do spustitelné aplikace. Tento poslední přístup lze také použít, kdy jsou modely použity jen pro komunikaci mezi členy týmu, ale nepředpokládá se, že by se podle nich tvořil zdrojový kód. V praxi to může být použito i v organizaci, která tvorbu kódu zadává externí společnosti, ale chce si ponechat možnost kontroly nad celkovou architekturou systému a modely tak používá jako dokumentaci a zároveň jako zadání pro onu externí organizaci, která bude zdrojový kód vytvářet.

MDA

Jedná se o další metodiku vývoje softwaru. Je založena na myšlence postupného zpřesňování modelů. Přesněji modelů na té nejvyšší úrovni abstrakce, které obsahují popis zákazníkových procesů bez jakéhokoliv vztahu k implementaci nebo technologii, až po modely na nejnižší úrovni abstrakce, které se již dají mapovat na zdrojový kód nebo jinou platformu, kde tyto modely mohou být přímo prováděny [18].

Model-driven architecture je specifikace od sdružení Object Managment Group (OMG), která se snaží formalizovat a zautomatizovat tyto transformace modelů a dále tak ulehčit vývoj systémů. Koncept MDA využívá a rozšiřuje množství stávájících specifikací skupiny OMG. Zejména UML, MOF (*Meta-Object Facility*) nebo XML (*Extensible Markup Language*). Zatímco však samotné UML je v praxi spíše používané jako nástroj pro komunikaci, v MDA se využívá již jako nástroj pro detailní popis systému, který se dá přímo implementovat. Existují však i některé alternativní nástroje k nástrojům z řady OMG.

Další klíčovou vlastností MDA je platformně nezávislý vývoj. To je možné díky oddělení business logiky v podobě modelů od technologie konkrétní platformy. Aplikace v MDA je až na konec realizována (nejlépe automaticky) podle konkrétních možností dané platformy (například pomocí CORBA, MPI, Java, .NET). Teoreticky je možné aplikace v budoucnu převést i na nové platformy. Mají tak vyšší šanci na delší životnost. Zároveň jsou tak sníženy náklady spojené s technologickými změnami [18].

V základu MDA definuje 4 úrovně modelů, které se pak mezi sebou postupně převádí a transformují. Základní definice a popis těchto modelů je uveden níže.

CIM (Computation Independent Model)

Je model nezávislý na zpracování v počítačovém systému. Modeluje hlavně business procesy důležité v prostředí zákazníka. Právě tento model se vytváří spolu se zákazníkem nebo zadavatelem softwarového projektu, abychom zjistili povahu jeho podnikání nebo slovník domény. Tímto modelem projekty v MDA začínají a vytvářejí je například business analytici nebo samotní uživatelé [18].

PIM (Platform Independent Model)

Je platformně nezávislý model, který zobrazuje koncept řešení dané oblasti na základě konkrétních požadavků. Ale stále nezohledňuje konkrétní technologie nebo implementaci. Do těchto modelů mohou patřit například diagramy tříd.

PSM (Platform Specific Model)

Je již platformně závislý model, který vyjadřuje jak je problém řešen na konkrétní platformě. Tento model nejčastěji vzniká transformací z PIM modelu. Důležité je, že pro jeden PIM model může existovat více PSM modelů. Tyto modely se pak skládají nejčastěji z diagramů, které mohou být mapovány na zdrojový kód, například diagram tříd. Ale je možné použít i jiné diagramy pro popis chování.

Code

Jedná se o výsledný zdrojový kód aplikace. Ten se dá z pohledu MDA také chápat jako model. Takový kód je pak již přímo závislý na konkrétní platformě, na kterém je systém provozován.

Transformace modelů

Jak bylo zmíněno, projekty řešené pomocí MDA začínají tvorbou modelů CIM. Jejich převod na modely PIM je čistě manuální. Obvyklým postupem (ne jen v přístupu MDA) je transformace business procesů do akcí uživatele, což je vyjadřováno diagramem případů užití. Tento převod je tedy z pohledu MDA ponechán zcela na uživateli a dál se tímto procesem nijak nezabývá.

Převod modelu PIM na PSM pak představuje hlavní přínos MDA. Po vytvoření modelu PSM a jeho prostudováním a případných úpravách, se tento model automaticky převádí na hotový model PSM nebo na kostru PSM, kterou návrhář manuálně doplní.

Převod modelu PSM na kód je již fungující koncept, který je dostupný v celé řadě jiných nástrojů. Hlavní přínos tak tedy má převod z PIM na PSM. To většinou nelze provést zcela automaticky, ale modely PIM se doplňují návrhářem o tzv. mapovací značky, které určují jak se bude element transformovat například na návrhové vzory a tak dále. Značka může mít různý význam nebo představovat jinou transformaci podle konkrétní platformy. Toto je ale pro návrháře transparentní.

V určitých případech je možné provést reverzní transformace [18], které mohou být důležité například při tvorbě dokumentace nebo při nalezení chyby. Kde se její opravení v modelech na nižší úrovně abstrakce může převést zpět do analytických modelů.

V určitém ohledu navrhovaný nástroj v této práci poté využívá podobný princip, kdy se začíná modelováním diagramu případů užití a ten se dále zpřesňuje až do podoby, který obsahuje dostatek informací ohledně chování systému. Tyto detailní modely je pak možné implementovat.

Generace programovacích jazyků

Pro programovací jazyky lze identifikovat 4 generace [13]. První generace programovacích jazyků byl zápis programu přímo v binárním kódu. Tento způsob byl velice pracný a bylo snadné v něm udělat chybu. Počítače proto programovali pouze odborníci. Do druhé generace lze zařadit jazyky typu assembler, které daným instrukcím v binárním kódu přiřazují

speciální zkratky pomocí alfanumerických symbolů. Například je jednodušší si zapamatovat CMP, ADD nebo SUB pro porovnání, sečtení nebo odečtení, než sekvenci jedniček a nul pro tyto instrukce. Program nazývaný kompilátor poté tyto zkratky převede do příslušného binárního kódu. Nevýhodou však zůstává nutnost znát konkrétní počítač a jeho architekturu, aby bylo možné tvořit funkční a efektivní kód pomocí assembleru. To je zároveň překážka pro přenositelnost programu na jiný počítač s jinou architekturou. Třetí generace jazyků, nazývaných vysoko-úrovňové jazyky, obsahují speciální funkce nebo příkazy, které provádějí desítky nebo stovky příkazů v assembleru. Sem patří procedurální nebo objektově-orientované jazyky jako například jazyk Algol, Cobol, C nebo C++ a jejich použití výrazně zvýšilo produktivitu při tvorbě softwarových systémů. Čtvrtá generace jsou pak jazyky, které neříkají počítači jak algoritmus provést, ale pouze co se má provést nebo co má být výsledkem. Sem můžeme zařadit například jazyk SQL. Jako folklór se tvrdí [1], že jazyky čtvrté generace dále zvýšili efektivitu tvorby 5 až 50krát v závislosti na řešených problémech. Jejich nevýhoda však je, že se často nehodí pro všechny programy nebo celé softwarové systémy. Jako speciální pátá generace se někdy označuje přirozený jazyk. Ten však v současné době počítače nedokáží efektivně zpracovat a nelze je tedy použít pro programování počítačů. Dalším zvýšením efektivity při tvorbě softwaru lze dosáhnout modelováním a tím urychlení pochopení nebo formalizování algoritmů. Ale tento skok již není tak vysoký jako u přechodu mezi jednotlivými generacemi programovacích jazyků. Proto jako další ulehčení a zvýšení efektivity by při tvorbě softwaru mělo být právě použití přístupů jako je MDA.

Kapitola 3

Stávající nástroje pro modelování

Tato kapitola srovnává několik nástrojů použitelných pro modelování softwaru. Následující výčet není kompletní a je spíše pro představu, které nástroje jsou k dispozici a co za prostředky nabízí. Jsou zde jak komerční nástroje, které se hodí pro různé odvětví IT, tak i nástroje zdarma ke stažení, které se snaží usnadnit a urychlit alespoň některé aspekty modelování.

3.1 Enterprise Architect

Jedná se o komerční aplikaci od firmy Sparx Systems[1], která poskytuje nástroje pro modelování částí prakticky ve všech úrovních životního cyklu tvorby softwaru. Nástroj umožňuje modelovat jak obchodní a IT systémy na vyšších úrovních abstrakce, tak software z pohledu implementace a návrhu a také tvořit diagramy pro návrh systémů pracujících v reálném čase.

Nástroj jako takový umožňuje tvorbu modelů podle standardů UML, SysML, BPMN a pár dalších podle otevřených standardů. Mimo to se také soustředí na tvorbu modelů v multiuživatelském prostředí a umožňuje tak snadno sdílet modely a vytvářet přehledné reporty a dokumentace. Kromě tvorby modelů také umožňuje simulaci modelů, které toto podporují. Lze tak snadno validovat vytvořené modely a případně rychle reagovat na změny v návrhu. Pokud se jedná o některé modely pro návrh softwarové architektury (diagramy tříd, databázové ER diagramy), lze z takto validního modelu generovat zdrojové kódy v požadovaném programovacím jazyce.

Co se UML týče, nástroj umí prakticky všechny používané modely od diagramu případů užití, diagramů tříd, diagramů interakce, stavové diagramy až po diagramy nasazení a jiné. Obecně staví na UML ve verzi 2.5. Jeho nevýhodou je, že v době psaní této práce byl na oficiálních webových stránkách dostupný jen ve verzi pro operační systém Windows.

3.2 Microsoft Visio

Nástroj Visio[2] je z větší části nástroj spíše pro tvorbu obchodních diagramů nebo obecně pro podporu dokumentace pro obchodní svět. Obsahuje například diagramy pro hierarchický popis vedení společnosti, plány kanceláří nebo diagramy pro popis obchodních procesů.

[1] http://www.sparxsystems.com/products/ea/

[2] https:\/\/www.microsoftstore.com/store/mseea/cs_CZ/pdp/Visio-Standard-2016/productID.324453500

Mimo to ale také podporuje různé diagramy pro popis počítačových sítí, hierarchie webových stránek nebo některé diagramy z rodiny UML jako diagram případů užití, diagramy aktivit, sekvenční diagramy a diagramy tříd. Visio jako takový se spíše zabývá samotnou tvorbou diagramů tak, aby byly graficky přívětivé, chybí některé možnosti jako u ostatních nástrojů specializující se na tvorbu diagramů pro popis softwaru a softwarových projektů. Omezení nástroje Visio je dostupnost pouze pro platformu Windows. Na druhou stranu se jedná o dlouze vyvíjenou a stabilní aplikaci. Jako druhá výhoda je ta, že obsahuje velké množství typů diagramů, které lze v této aplikaci vytvářet. To znamená nejen diagramy vhodné pro tvorbu softwaru, ale i pro popis obchodních činností, nákresy budov a místností a různé další.

3.3 StarUML

Aplikace pro tvorbu převážně UML diagramů[3]. Celkem jich podporuje 11 a pro každý umožňuje jejich snadnou tvorbu. Jeho příjemnou vlastností je, že například v diagramu tříd umí pracovat s hierarchií objektů v diagramu, takže je například možné přiřadit třídu do balíčku tříd a nástroj tento vztah bere v potaz. Při manipulaci s balíčkem jsou pak ovlivněny i elementy všech tříd v něm (například při přesunu balíčku na plátně nebo při smazání celého balíčku jsou smazány i jeho třídy). Mimo to také podporuje diagramy pro popis databáze nebo flowchart diagramy. Aplikace jako taková neumí sama generovat kód z vytvořených modelů, ale výsledné modely lze transformovat pomocí nástroje třetí strany.

Výhodou nástroje je vestavěná podpora pro tvorbu HTML dokumentace z vytvořených diagramů, která je pak snadno vystavitelná na web, odkud je dostupná mezi všemi členy vývojového týmu.

Aplikace je vydávána ve dvou verzích, jak komerční tak i bezplatné, jen s omezenými možnostmi. Je multiplatformní a programovaná v jazyce Javascript. Díky tomu podporuje snadnou tvorbu a instalaci doplňků, tzv. rozšíření, díky kterým je možné do aplikace dodat další nové vlastnosti. Toto je však zároveň nevýhoda. K přizpůsobení aplikace může být nutné použít velké množství rozšíření a pokud dojde k nekompatibilní změně jen v jednom rozšíření například při vydání jeho nové verze, celý program pak nemusí pracovat korektně a může dojít k omezení jeho funkčnosti.

3.4 UMLet

UMLet[4] je velice jednoduchý nástroj pro prostou tvorbu UML diagramů. Zaměřuje se na diagramy UML jako diagram případů užití, diagramy tříd, sekvenční diagramy a diagramy interakce nebo diagramy nasazení. Aplikace s diagramy pracuje jen na úrovni grafických prvků tzv. *drag-and-drop* metodou a je na uživateli jak si diagram vytvoří. Nezajišťuje tak korektnost diagramů a tedy neumožňuje jejich simulaci nebo generování kódu. Má však velice jednoduché rozhraní, kde vlastnosti jednotlivých prvků lze měnit pomocí textového popisu, místo různého nastavení v dialogových oknech, což vyžaduje více akcí uživatele a zpomaluje to tak tvorbu diagramů.

Nástroj je napsaný v jazyce Java a je tak multiplatformní. UMLet také umožňuje export jednotlivých modelů do různých obrázkových formátů nebo PDF, který je pak použitelný v dokumentaci.

[3] http://staruml.io/
[4] http://www.umlet.com/

3.5 Lucidchart

Tento nástroj[5] umožňuje tvorbu jak UML diagramů, tak různých flowchart diagramů nebo wireframe návrhů pro uživatelské rozhraní. Zaměřuje se hlavně na snadné sdílení diagramů mezi uživateli a členy týmu a nástroj jako takový je vytvořen jako webová aplikace, která běží v prohlížeči. Opět funguje na principu *drag-and-drop* přístupu. Výhoda takové webové aplikace je možnost propojení s jinými službami jako Google Docs, Confluence, Slack, Jive, Jira a jiné, do kterých je možné vytvořené diagramy vkládat.

Podle oficiálních stránek nástroje Lucidchart, se tento nástroj snaží nabídnout podobné možnosti jako má Microsoft Visio, ale všechny možnosti se snaží využít v Cloudu. O tento přístup se dnes snaží čím dál tím větší počet aplikací a nástrojů a umožňují tím kolaborativní editaci diagramu několika uživateli v reálném čase. Podobně, jako to dnes umožňují například Google dokumenty. Některé diagramy dokáže tvořit pomocí textového popisu, podobně jako UMLet. Lucidchart také funguje na iOS zařízeních, pro které vydává jednoduchou mobilní verzi aplikace.

3.6 MetaEdit+

Zatímco výše zmíněné nástroje jsou víceméně stavěné pro práci s UML (nebo podobným jazykem), který se snaží definovat diagramy a grafický jazyk pro co nejširší a nejobecnější použití, MetaEdit+[6] pracuje na bázi DSM (*Domain-Specific Modeling*). Různé odvětví nebo domény mohou mít vlastní potřeby pro zobrazení modelu. Místo toho, aby se aplikace snažila definovat všechny možné případy a tím umožnit tvorbu všech různých modelů, místo toho dovolí uživateli vytvořit si vlastní podobu specifického doménového jazyka. Uživatel pak při modelování může pracovat na vyšší úrovni abstrakce [14].

Aplikace obsahuje speciální GOPPRR metamodelovací jazyk, kterým uživatel popíše svůj model. Takový metamodel pak prakticky modeluje model. Lze zvolit podobu specifických grafických elementů, legální propojení mezi nimi a různé omezení, které je v doménovém modelu potřeba dodržet. Poté co je takový metamodel vytvořen, může uživatel tvořit konkrétní modely v tomto doménovém jazyce a MetaEdit+ poskytuje všechnu další podporu pro tvorbu a kontrolu modelů.

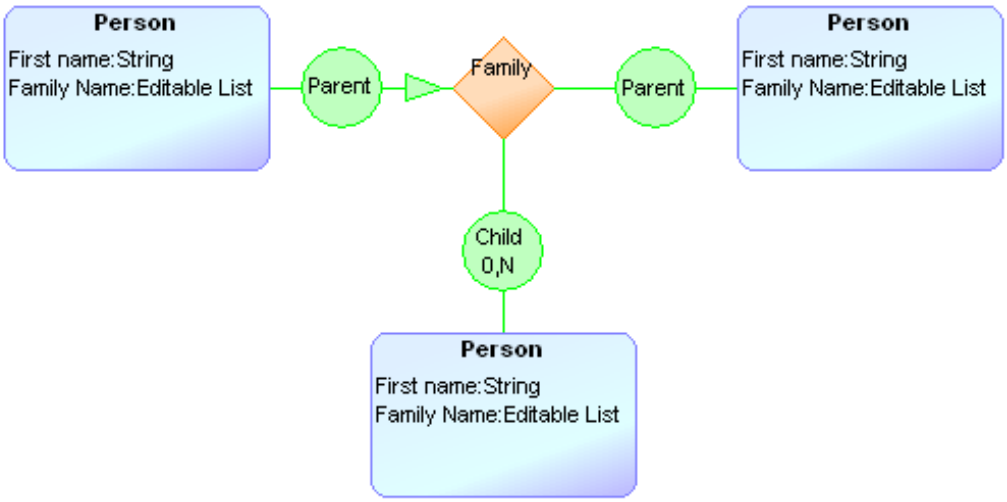
Jako příklad můžeme uvést modelování rodinného stromu. Mohli bychom k tomu zneužít některý diagram z rodiny UML, ale to v tomto případě, nemusí být správné a žádný UML diagram neposkytuje správná omezení na tvorbu takového rodinného stromu. V MetaEdit+ však můžeme snadno vytvořit například takový metamodel, který je zobrazen na obrázku 3.1.

Jako druhá věc je tvorba konkrétního modelu. To je přiřazení grafických symbolů, nejčastěji vytvořených pomocí vektorové grafiky, jednotlivým prvkům v metamodelu. Aplikace poté pomocí univerzálních nástrojů a dialogů dovoluje tvorbu konkrétního rodinného stromu pomocí drag-and-drop přístupu jako u ostatních nástrojů. Pomocí tohoto typu modelování lze vytvořit specifický modelovací jazyk podle řešeného problému nebo aplikace. Doménově specifické modelování je tak oblíbené, protože lze vytvořit a vyladit jazyk přesně pro potřeby aplikace a je poté daleko rychlejší v takovém jazyce tvořit požadované modely. Uživateli to totiž dovoluje přemýšlet přímo v konkrétním jazyce na dané úrovni abstrakce a nemusí řešit, jak model převést například na diagram v nekompatibilním jazyce jako je UML.

[5] https://www.lucidchart.com/

[6] http://www.metacase.com/products.html

Obrázek 3.1: Příklad metamodelu v aplikaci MetaEdit+



3.7 AtoMPM

Tento nástroj je výzkumný framework vyvinutý lidmi z různých univerzit sloužící pro generování doménově specifických modelovacích nástrojů běžících ve webovém prostředí. AtoMPM[7] jako takový je open-source vytvořený v jazyce Python a je tak multiplatformní.

Aplikace podporuje podobný postup práce jako MetaEdit+ a díky webovému prostředí je možné snadno sdílet modely mezi různými lidmi nebo členy týmu. Kromě možnosti tvorby samotného metamodelu také umožňuje popsat sémantiku modelu pomocí transformací stavů modelu omezené různými pravidly. Pomocí popisu těchto transformací umí nástroj sestavit i jednoduchý simulátor.

3.8 Shrnutí

Jak je vidět, dnes je možné sehnat aplikace pro tvorbu různých modelů od klasických UML diagramů, pro popis a tvorbu hardwarových systémů až po popis obchodních procesů. Velká část diagramů je standardizovaná a formalizovaná v dostatečné míře, že je možné navržené modely simulovat a validovat. Dokonce z nich lze vytvářet zdrojový kód použitelný při implementaci výsledného systému. Velké komerční nástroje pak dovolují správu modelů ve všech úrovních vývojového cyklu softwaru a s tím spojenou tvorbu dokumentace. Tyto nástroje jsou pak tvořené obecně v tom smyslu, že je lze použít u velmi rozmanitých projektů.

[7] http://www-ens.iro.umontreal.ca/~syriani/atompm/atompm.htm

Kapitola 4

UML

UML je zkratkou *Unified Modeling Language* [20], což je grafický jazyk pro vizualizaci, specifikaci a dokumentaci softwarových částí a systémů. Nabízí jak zápis softwarových částí jako popis implementačních tříd nebo chování funkcí, tak i prvky pro popis obchodních procesů nebo databázových schémat.

UML podporuje jak procedurální přístup k analýze tak i objektový. Jazyk předepisuje jak by měly jednotlivé diagramy vypadat a sémantiku jednotlivých grafických elementů. Nepředepisuje však jak se má UML používat nebo způsoby jak analyzovat nebo specifikovat softwarové systémy. Za tuto část je zodpovědný softwarový architekt nebo člověk, který vytváří specifikaci systému. UML pak slouží jen jako nástroj, který dokáže tyto úlohy zjednodušit.

Následně UML slouží pro komunikaci mezi vývojáři a modeláři nebo pro zaznamenání návrhu. Pro různé části softwaru se hodí jiné typy diagramů. Důležitá je srozumitelnost diagramů a snadnost změn. Existují specializované nástroje (CASE[1]), které dokáží kontrolovat informace mezi jednotlivými diagramy a zajistit tak konzistenci jednotlivých modelů. Také umí z modelů a diagramů vygenerovat programový kód použitelný ve vlastním kódu aplikace. Hlavním problémem tohoto základního přístupu je zajištění konzistence modelů a kódu. Jakmile se napíše kód a dále se rozšiřuje, už se však stejné informace neudržují uvnitř navržených UML modelů. Snaha o zajištění a udržení informací na dvou místech je na to příliš velká při běžném vývoji a nevyplatí se. Modely se tak zahazují a dále se pracuje jen s programovým kódem. To však může být problém pro nově příchozí vývojáře, pro které může být orientace v kódu zapsaném v programovacím jazyce obtížnější, než při čtení diagramů. Další možností je mít nástroje, které z modelů dokáží generovat kompletní programový kód, do kterého již není nutné manuálně zasahovat a jediné co musí programátor udržovat je daný model.

4.1 Diagram případů užití

Tento typ diagramu zobrazuje chování systému tak, jak ho vidí uživatel a jaké chování mu systém nabízí. Jedná se tak o diagram, který má zachytit dynamické chování systému. Hlavním účelem je zobrazit funkcionalitu systému a k čemu ho může uživatel použít. Důležitým rysem je, že zobrazuje jen funkce, které má systém umět, ale nezobrazuje detaily o tom, jak by je měl provádět. Diagram je často jeden z prvních, které se při návrhu vytvářejí a jsou často navrhovány spolu se zákazníkem, který si softwarový produkt nechává

[1] Computer Aided Software Engineering

vytvořit. Protože zákazník nemusí a ani většinou nemá hlubší základy s tvorbou softwaru nebo programování obecně, tak je podoba tohoto diagramu navržená tak, aby ho i tito lidé snadno pochopili a byli schopni ho rychle začít používat bez větších potíží.

4.1.1 Případ užití

Případ užití představuje nějakou vlastnost softwaru, kterou má plnit nebo sekvenci akcí, které plní nějakou funkci, kterou od systému požadujeme. Je to jakási diskrétní jednotka interakce mezi uživatelem (člověk nebo stroj) a navrhovaným systémem [5]. To může být akce jako vytvoření nového uživatele, zobrazení dat ve vhodné podobě, vygenerování faktury a tak dále. Podrobnosti jako validace unikátnosti uživatelského jméno nebo způsob jak se budou data vybírat pro zobrazení však v případu užití již zobrazeno není a je to záměrně skryto. Vychází tedy ze zadání systému zákazníkem, kde nás detaily implementace zatím příliš nezajímají. Každý případ užití má pak v klasickém modelování dále detail případu užití, kde je detailnější textový popis takového případu užití a jeho chování. Podmínky, které musí být před a po vykonání případu užití splněny a kroky, které má daný případ užití vykonat. Tyto detaily případů užití však jsou nejčastěji součástí druhého dokumentu a nejsou přímo součástí diagramu případů užití. Diagram případů užití pak plní stejnou roli jako tabulka aktér versus případ užití v klasickém přístupu při modelování případů užití, která zobrazuje, který aktér v systému může provádět nebo spouštět daný případ užití [10].

Grafický symbol pro případ užití nejčastěji vypadá jako elipsa uvnitř které je jeho název, tak jak je naznačeno na obrázku 4.1. Každý případ užití může stát samostatně nebo pro zjednodušení může rozšiřovat nebo obsahovat jiný případ užití (viz dále).

Obrázek 4.1: Symbol případu užití

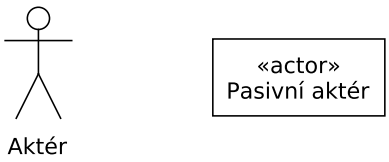
4.1.2 Aktér

Aktér zobrazuje roli, která komunikuje nebo spouští jednotlivé případy užití. Role může být cokoliv, co může interagovat s naším systémem. Například uživatel (obyčejný uživatel, administrátor) nebo externí počítačový systém. Speciálním aktérem může být čas, který může spouštět jednotlivé případy užití v určitém čase nebo periodicky v určitých intervalech.

Grafický symbol pro aktéra nejčastěji vypadá jako panáček s názvem aktéra pod ním. Ten je naznačen na obrázku 4.2. Někdy je možné se setkat se zobrazením aktéra ve tvaru obdélníku s názvem uprostřed. Zejména u pasivních aktérů.

Aktéry tedy můžeme rozdělit do dvou skupin – aktivní a pasivní. Aktivní aktér (někdy také nazýván primární aktér [10]) sám může iniciovat případy užití a vyvolávat tak akce v systému. Tito aktéři se někdy pro přehlednost kreslí na levou stranu diagramu případů užití. Druhou skupinu představují pasivní aktéři (někdy také nazýváni podpůrní aktéři [10]) jsou naopak iniciováni případem užití a umožňují mu tak dosáhnout jeho cíle. To může být například externí fakturační systém, server s informacemi o počasí nebo server s aktuálními kurzy měn. Tito se někdy pro přehlednost kreslí na pravou stranu diagramu.

Obrázek 4.2: Symbol aktéra

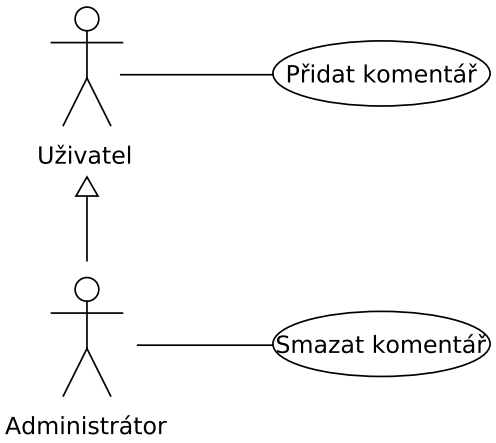


4.1.3 Vztahy mezi aktéry a případy užití

Dále musí být definovány vztahy mezi jednotlivými aktéry a případy užití, které se jich týkají. Takový vztah je naznačen jednoduchou čárou mezi nimi a znázorňuje, že daný aktér může iniciovat (nebo být iniciovaný) daným případem užití. Tak jak je vidět na obrázku 4.3.

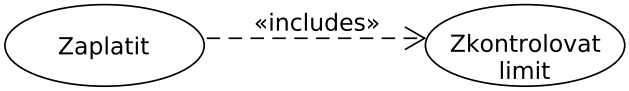
Speciální vztah generalizace používá plnou bílou šipku mezi aktéry a říká, že aktér od kterého vede šipka má stejné práva nebo možnosti jako druhý aktér, ale může přidávat některé své další. Grafický symbol tohoto vztahu je také naznačen na obrázku 4.3. Tato vazba se používá pro zjednodušení diagramu, kde dva a více aktérů mají stejné vazby na stejné případy užití.

Obrázek 4.3: Základní vztahy mezi aktéry a případy užití



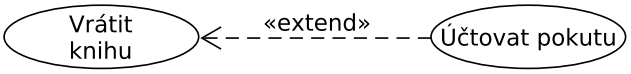
Další speciální vazby mohou vznikat mezi samotnými případy užití. To jsou vazby *include* a *extend*. Vazba *include* říká, že daný případ užití obsahuje nebo vždy spouští druhý případ užití. To umožňuje znovupoužití některých případů a tak zjednodušení scénářů detailů případů užití. Znázornění tohoto vztahu je na obrázku 4.4.

Obrázek 4.4: Vztah include



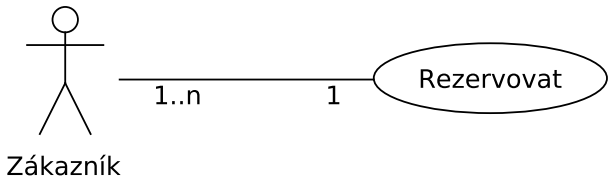
Vazba *extend* naopak říká, že případ užití může a nemusí být volán z druhého případu užití. Případ užití, který je takto rozšiřován druhým případem užití pak definuje takzvané body rozšíření. To jsou místa, odkud může být tento rozšiřující případ užití volán v detailu případu užití. Důležité je, že jakákoliv zmínka případu užití, který takto rozšiřuje primární případ užití již není nikde zmíněn v těle detailu primárního případu užití [10]. Tento typ se však objevuje méně často. Grafické znázornění tohoto vztahu je pak vidět na obrázku 4.5.

Obrázek 4.5: Násobnost vztahu mezi aktérem a případem užití



Je také možné specifikovat násobnost vztahu mezi aktérem a případem užití, podobně jak je naznačeno na obrázku 4.6. Tím lze vyjádřit kolik aktérů může vstupovat nebo se podílet na daném případu užití. Implicitní hodnota je 1:1. V takovém případě se hodnota neuvádí a je vyobrazena klasická jednoduchá čára.

Obrázek 4.6: Násobnost vztahu mezi aktérem a případem užití



4.1.4 Modelování s použitím případů užití

Hlavní úlohou modelu případů užití je pak správně zobrazit a pomoci pochopit, kdo s navrhovaným systémem může pracovat nebo komunikovat a které části systému budou využívány k dosažení daného cíle. Postup tvorby modelu případů užití pak vypadá zhruba následovně:

1. Identifikují se aktéři, kteří představují role, které se systémem budou komunikovat. Každý takový aktér se označí jménem.
2. Poté se identifikují funkce, které by měl systém poskytovat a které jednotlivé role potřebují ke své práci nebo dosažení svého cíle s použitím navrhovaného systému.
3. Vytvoří se asociace a vztahy mezi jednotlivými aktéry a případy užití.
4. Poté se vytvoří detailní popis případů užití, který může dále sloužit jako podklad pro implementaci funkce. A to buď formou textu nebo popisem pomocí jiných vhodných diagramů.

4.2 Diagram tříd

Diagram tříd je pro změnu statický diagram. Většinou se používá pro modelování objektově orientovaných programů, protože mohou být přímo převedeny na třídy konkrétního

programovacího jazyka. Obecně se jedná o jediný UML diagram, který lze jedna ku jedné transformovat na kód programovacího jazyka. Případně s možností ho dále upravovat člověkem. Díky tomu je z pohledu modelování zařazován do skupiny diagramů implementace [6]. Na rozdíl například od doménového modelu, který je spíš jakýmsi náčrtem systému. Z toho také plyne, že diagram tříd je platformně závislý na právě použitém programovacím jazyce. Díky svým výhodám a přehlednosti jsou diagramy tříd používány velmi často a to i v projektech, kde není kladen důraz na formální dokumentaci nebo návrh před fází implementace.

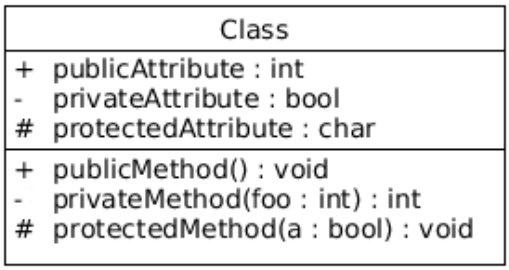
S dobře navrženým diagramem tříd by programátor již neměl řešit žádné zásadní otázky při programování a jeho práce by tak měla být co nejvíce rutinní. Stačí aby informace z diagramu jen přenesl jedna ku jedné do použitého programovacího jazyka. Většina architektonických otázek se tak řeší v čase návrhu. To při tvorbě diagramu umožňuje vidět systém jako celek. Minimalizuje se tak šance, kdy programátor stráví dlouhý čas programováním a řešením malých technických detailů u první půlky programu a u druhé zjistil, že to takto nemůže jako celek fungovat.

V následujících odstavcích si popíšeme hlavní rysy diagramu tříd, které nás budou zajímat v této práci.

4.2.1 Třída

Třída je stěžejní součástí diagramu tříd. Na následujícím obrázku 4.7 je příklad grafického znázornění třídy.

Obrázek 4.7: Symbol třídy



Protože se již jedná o dokumentaci k implementaci, nepíší se ve jménech atributů nebo metod znaky s diakritikou. V první části jsou zobrazeny atributy s datovými typy a ve druhé metody. Objektově-orientované jazyky poté k možnosti implementace zapouzdření používají různé úrovně viditelnosti atributů a metod, které lze v diagramu tříd také zachytit. To jsou:

- + (plus) – veřejný atribut nebo metoda
- - (mínus) – privátní atribut nebo metoda
- # (hash kříž) – chráněný atribut nebo metoda
- ~ (tilda) – atribut nebo metoda viditelná pouze v rámci balíčku, pokud to programovací jazyk podporuje

Je také možné zobrazit statické atributy a metody. Ty velké množství nástrojů zobrazují podtržením. Podobně můžeme vyobrazit i rozhraní. Oproti třídě se liší v tom, že má stereotyp[2] «interface» a často nemá atributy. Pouze metody.

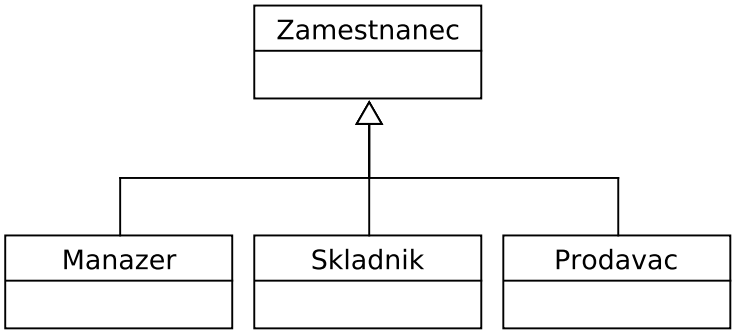
4.2.2 Vztahy

Stejně jako u případů užití tak i mezi třídami můžeme mít různé vztahy. Bez nich by diagram tříd neměl příliš smysl. Systémy realizované pomocí objektů totiž nejčastěji fungují na principu kooperace takových objektů, kdy dohromady spolupracují na nějakém výpočtu.

Generalizace/specializace

První vztah může být generalizace/specializace podobně jako u případů užití a používá se pro znázornění dědičnosti mezi třídami. Na obrázku 4.8 je pak zobrazen případ, kdy třídy *Manazer*, *Skladnik* a *Prodavac* dědí od třídy *Zamestnanec*. Dědí se atributy a metody a dědící třída může přidávat nebo upravovat tyto zděděné atributy nebo metody.

Obrázek 4.8: Znázornění vztahu generalizace v diagramu tříd



Realizace

Druhý vztah je realizace rozhraní, znázorněný na obrázku 4.9 mezi třídou *Posloupnost* a rozhraním *Iterable*. To je vztah mezi třídou a rozhraním, který dává najevo, že daná třída implementuje toto rozhraní. U této vazby se používá stejná šipka jako u generalizace/specializace, jen je čárkovaná.

Asociace

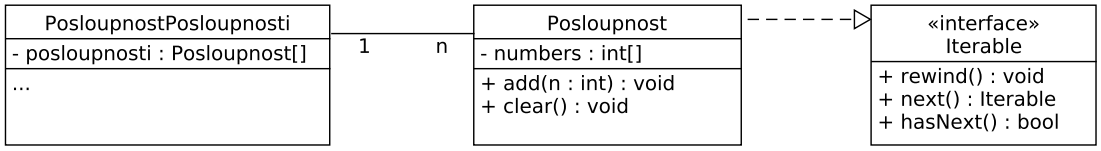
Tento vztah pak zobrazuje obecný typ asociace mezi třídami. Často se u něj udává násobnost vztahu (jestli se jedná například o vztah 1:1 nebo 1:N). Vztah je zobrazen na obrázku 4.9.

4.2.3 Tvorba diagramu tříd

Hlavním problémem při tvorbě diagramu tříd je identifikovat správné třídy [3]. Modelování lze provést dvěma způsoby. Zdola nahoru, kdy se jako první identifikují vstupy a výstupy

[2]Stereotyp se píše mezi « a » dokáže změnit význam určitého prvku v UML diagramu.

Obrázek 4.9: Příklad asociace a realizace



systému. Postupně se nalézají způsoby, jakým jsou vstupy převáděny na výstup a těmto procesům jsou přiřazeny konkrétní třídy s danou zodpovědností.

Druhý přístup je shora dolů. Třídy jsou identifikovány podle popisu požadovaného chování systému. Jako první jsou vytvořeny nejdůležitější třídy, které lze namapovat na vysokoúrovňový popis systému a postupně jsou zpřesňovány a upravovány.

Obecně se dají tyto dvě metody použít současně, kde se každá metoda hodí na jiný typ nebo na různou část modelovaného systému. V obou přístupech jsou zároveň identifikovány jednotlivé vztahy mezi třídami a jejich atributy. To jsou informace, které chceme vědět o daném objektu. Není neobvyklé, že se provádí několik iterací návrhu, než jsou nalezeny všechny třídy a jejich vztahy.

Jednotlivé třídy a jejich metody mohou být dále specifikovány dalšími typy UML diagramů. To jsou například diagramy sekvence nebo diagramy aktivit. Ty jsou pak stejně jako detaily případů užití součástí externích dokumentů a jako takové nejsou přímo součástí diagramu tříd, ale často se používají. V našem případě navrhovaný nástroj využívá objektově-orientované Petriho sítě.

Kapitola 5

Petriho sítě

Petriho síť je matematická struktura často používaná pro reprezentaci diskrétních distribuovaných systémů. Jedná se o bipartitní graf, jehož uzly představují místa a přechody. Hrany mezi nimi pak popisují, která místa a za jakých podmínek mohou způsobit přechod do místa jiného. Petriho sítě mají i grafické znázornění.

Koncept Petriho sítí má svůj původ v dizertační práci Carla Adama "Kommunikation mit Automaten" napsanou v roce 1962. Od té doby se Petriho sítě ukázaly jako účinné i pro mnohem obecnější problémy než jsou distribuované systémy a dnes se mimo jiné používají pro popis programovacích jazyků, protokolů, systémů pracujících v reálném čase, telekomunikačních systémů, v simulacích, pro popis biologických systémů nebo popisu obchodních procesů.

5.1 Základ Petriho sítí

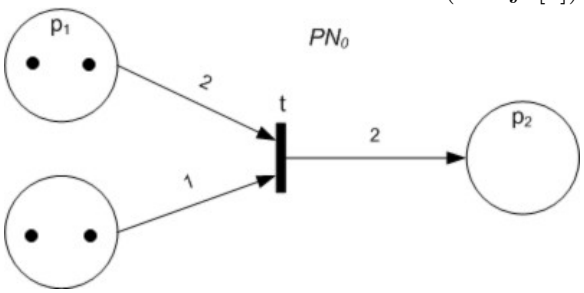
Každá Petriho síť se skládá z míst, přechodů a hran. Hrana je vždy mezi místem a přechodem nebo naopak. Nelze mít hrany mezi dvěma místy nebo dvěma přechody. Hrana má podobu šipky a určuje tak směr. Místo ze kterého vede hrana se nazývá vstupní místo a místo do kterého hrana vede se nazývá výstupní místo [19]. Příklad jednoduché Petriho sítě je zobrazen na obrázku 5.1.

Místa modelují pasivní elementy a mohou pouze ukládat nebo akumulovat značky [19]. Každé místo může mít libovolný diskrétní počet značek, tzv. tokenů (mají tedy diskrétní stav). Distribuce značek v místech a jejich počet poté představuje konfiguraci sítě. Hrana mezi vstupním místem a přechodem může mít omezení na počet značek, aby mohl být přechod proveden. Když lze provést přechod a ve vstupním místě je dostatečný počet značek, jsou tyto značky odebrány ze vstupního místa a počet specifikovaný výstupní hranou mezi přechodem a výstupním místem je vložen do výstupního místa (někdy se říká, že přechod byl odpálen).

Přechody naproti tomu modelují aktivní elementy. Mohou vytvářet, rušit nebo přemisťovat značky [19]. Přechody můžou obsahovat podmínku (stráž), která dále omezuje vykonání přechodu. Podmínka musí být pravdivá, aby mohl být přechod vykonán a nestačí tak dostatečný počet značek ve vstupních místech. Tato podmínka může být matematický výraz nebo funkce, která určuje hodnotu pravděpodobnosti jak často má být přechod odpalován. Taková funkce se často používá při různých simulacích.

Existují také speciální přechody, které nemají vstupní místo (source) a nebo výstupní místo (sink). Díky nim je možné do sítě generovat nové značky podle určité podmínky nebo

Obrázek 5.1: Příklad Petriho sítě (zdroj: [2])



naopak značky rušit. Praktické využití takových přechodů může být například v simulaci nákupní pokladny, kdy jeden přechod generuje značky představující zákazníky a druhý tyto značky zruší poté, co je zákazník na pokladně obsloužen.

Praktické omezení takovéto základní Petriho sítě v roli programovacího jazyka brání výrazné omezení, které je plošnost této struktury. Petriho sítě totiž v této základní podobě neposkytují strukturovací mechanismy, jako například procedury, funkce nebo moduly známé z klasických programovacích jazyků. To všechno umožňuje skládat složitější modely ze submodelů. Pokud totiž potřebujeme modelovat systém ve větším detailu, velikost Petriho sítě velice rychle roste a ztrácí se její pozitivní vlastnosti [12].

Formální definice

V literatuře se můžeme setkat s několika odlišnými definicemi Petriho sítí. Každá se může lišit podle toho, ke kterému účelu je Petriho síť určena. Například různé požadavky jsou pro sítě určené k popisu distribuovaných systémů a jiné pro popis obchodních procesů, kde nemusí být až tak velký důraz na matematický formalismus. Zde byla vybrána následující definice.

Petriho sít je orientovaný, ohodnocený bipartitní graf se dvěma typy uzlů, interpretované jako místa a přechody [17]. V grafické podobě se místa zobrazují jako kružnice a přechody jako čtverce nebo vyplněné tenké obdélníky. Hrany pak nikdy nespojují uzly stejného typy. Formálně je Petriho síť definována jako pětice $N = (P, T, F, M0, W)$, kde:

- P a T jsou disjunktní konečné množiny představující místa a přechody.
- $F \subset (P \times T) \cup (T \times P)$ je konečná množina hran.
- M_0: $S \to \mathbb{N}$ je počáteční značení sítě (to je počet značek v každém místě).
- W: $F \to \mathbb{N}_+$ je ohodnocení hran. To označuje kolik značek je odebráno ze vstupního místa nebo kolik značek je vloženo do výstupního místa. Poslední důležitou součástí je pravidlo pro provádění přechodů (někdy také označované jako odpálení).

Díky této matematické definici a formálnosti jsou Petriho sítě zajímavé z toho důvodu, že dovolují algoritmickou analýzu sítě. Spolu s jednoduchou grafickou syntaxí mohou být snadno navrhovány a díky této matematické teorii automaticky analyzovány a verifikovány [11].

5.2 Objektově-orientované Petriho sítě

Následující část neformálně popisuje objektově-orientované Petriho sítě. Ty se od základních Petriho sítí příliš neliší, jen zavádějí objektově-orientované koncepty. Například dědičnost a polymorfismus. Petriho sítě jsou silný nástroj pro modelování chování obyčejných a paralelních systému. Na druhé straně se objektově-orientované paradigma stalo velice oblíbené pro implementaci systémů díky tomu, že dovoluje snadnou dekompozici systému na menší části, u kterých je složitost natolik jednoduchá, že je lze snadno pochopit. Údržba se tak stává daleko jednodušší než při použití procedurálního programování. Objektově-orientované Petriho sítě jsou tak příhodným kompromisem mezi oběma přístupy a berou si z nich jejich výhody.

5.2.1 Třídy

V třídních programovacích jazycích třídy představují jakési šablony, které dovolují vytvoření konkrétních instancí objektů. Každý objekt je pak instancí nějaké třídy a dále je charakterizován identifikací a stavem v daném okamžiku [12]. Mají společné vlastnosti a chování, ale jednotlivé instance se od sebe mohou lišit. Třídy mohou být definovány i tak, že rozšiřují existující třídy. Hovoříme o dědičnosti. Dědí se jak atributy, tak operace. Nová synovská třída pak může přidávat nové atributy a operace a takto rozšiřovat rodičovskou třídu. Je to tedy šikovný nástroj jak dosáhnout sdílení kódu. Dá se říci, že třídy existují staticky a až jejich instance se podílejí na dynamickém běhu systému [12].

5.2.2 Dědičnost

Jedná se o inkrementální modifikaci existujících tříd. Každá třída může specifikovat svoji rodičovskou třídu, tedy tu od které dědí. Taková synovská třída dědí reprezentaci a chování. Dědičnost reprezentace spočívá v možnosti přidat další atributy ke stávájícím atributům rodičovské třídy. Dědičnost chování zase obsahuje možnost přidávat nové další metody nebo nahrazovat metody rodičovské třídy [12].

5.2.3 Objekty

Objekty v objektově-orientované Petriho síti mohou mít svoji vlastní síť, která je vykonávána samostatně a nezávisle na ostatních objektech. Pak se mluví o aktivních objektech. Pasivní objekty jsou pak případy objektů, které vlastní síť nemají a obsahují pouze metody. Mimo tuto sít, pak může mít každý objekt svoji sadu metod, které mají další své vlastní sítě. Tyto metody jsou pak vykonávány synchronně a lze jimi simulovat například volání procedur nebo zasílání zpráv. Tyto metody mají vstup a výstup stejně jako klasické metody v programovacích jazycích.

Objekty mohou během existence systému dynamicky vznikat a zanikat [12]. Každý objekt je jedinečně identifikován svým identifikátorem a má svůj vlastní stav. Jiné objekty nemohou přímo ovlivnit jeho stav. Ten má pod kontrolou jen daný objekt sám podle principu zapouzdření. Každý objekt poskytuje sadu operací nebo služeb, které mohou používat jiné objekty a vytváří tak veřejné rozhraní objektu. Stav objektu tedy může objekt změnit po provedení takovéto veřejné operace nebo autonomně, z vlastní příčiny na základě vykonávání své vnitřní sítě. Aby objekt mohl volat operace jiného objektu, musí navíc znát jeho jedinečný identifikátor.

5.2.4 Paralelismus

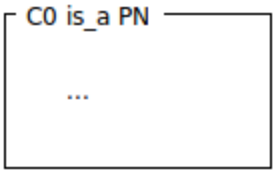
Objekty existují v nějakém čase [12]. V každém okamžiku tak lze nalézt množinu objektů, které existují. V každém takovém okamžiku se pak objekt nachází v určitém stavu. Sekvence změn stavu daného objektu v čase se pak označuje za proces objektu [12]. Stejně tak může být prováděna aktivita na straně metod a ty se mohou překrývat. Zpracování takových metod neproběhne okamžitě, ale trvá určitý čas. V tomto okamžiku může probíhat jiná aktivita uvnitř objektu. Lze tak hovořit o paralelismu.

5.2.5 Grafická notace objektově-orientované Petriho sítě

Objektově-orientovaná Petriho síť je trojice (Σ, c_0, oid_0), kde Σ je systém tříd, c_0 je počáteční objekt z tříd Σ a oid_0 je název počátečního objektu c_0.

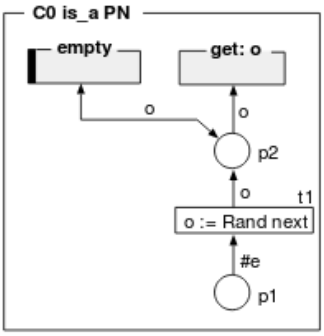
Základem je třída, která je popsána objektem. Graficky pak taková třída může vypadat jako na obrázku 5.2

Obrázek 5.2: Symbol objektu v objektově-orientované Petriho síti



Každý objekt může mít vlastní objektovou síť, která má za úkol popsat autonomní činnost objektu. Může být kreslena uvnitř symbolu objektu. Příklad je zobrazen na obrázku 5.3. Každá taková síť je pak prováděna paralelně a nezávisle na ostatních objektech. Mimo to má také počáteční značení, stejně jako u obyčejné Petriho sítě [15].

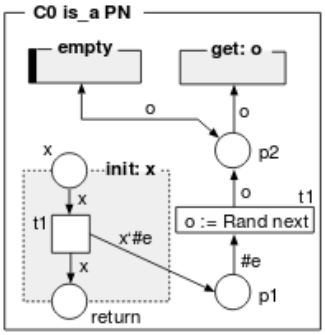
Obrázek 5.3: Objektová síť (zdroj: [12])



Každý objekt pak může mít sadu metod, které jsou popsány vlastními sítěmi. Od vnitřní sítě objektu se liší tím, že mají místa představující parametry metody a návratové místo. Taková síť může přistupovat k místům ve vnitřní objektové síti. Tím lze v metodě měnit stav objektu. Ukázková síť metody je zobrazena na obrázku 5.4 a je na něm vidět i možnost zasahovat ze sítě metody do sítě objektu (hrana z přechodu t1 do místa p1).

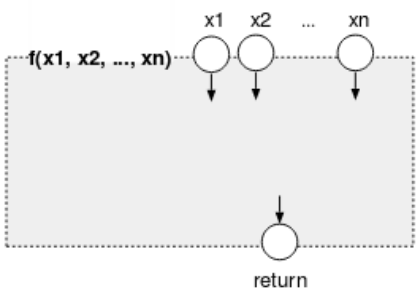
Obecně je třeba namapovat principy z programovacího jazyka do Petriho sítě. Metoda nebo funkce v nich může mít libovolný počet parametrů a jednu výstupní hodnotu. I toto

Obrázek 5.4: Příklad sítě metody objektu (zdroj: [12])



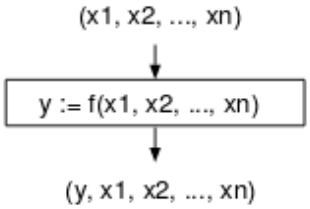
musíme převést do objektově-orientované Petriho sítě. Takovou metodu pak lze graficky vyjádřit stejně jako na obrázku 5.5.

Obrázek 5.5: Grafická definice metody (zdroj: [12])



Takto definovanou metodu také budeme chtít spustit. Z toho důvodu se zavádí speciální invokační přechod, jehož symbol je uveden na obrázku 5.6. Přechod je analogií k volání funkce v programovacím jazyce [12].

Obrázek 5.6: Invokační přechod (zdroj: [12])



Kapitola 6

Nástroj pro tvorbu diagramů

V této kapitole je popsán navrhovaný nástroj pro tvorbu diagramů pro usnadnění vývoje systémů a jeho možnosti a požadavky na něj. Hlavním cílem programu je umožnit tvorbu diagramu případů užití, z kterého je pak možné automaticky vygenerovat základní strukturu diagramu tříd. Každá třída pak může být popsána objektově-orientovanou Petriho sítí. Tento přístup modelování systému vychází z článku [16]. Nástroj pro tvorbu těchto diagramů pak umožňuje jak jejich tvorbu, tak synchronizaci informací jednotlivých diagramů mezi sebou. To znamená, že pokud se změní diagram případu užití, zároveň se tato změna automaticky promítne do diagramu tříd. Tím se zjednodušuje práce ve fázi návrhu a umožňuje modeláři soustředit se na jeho hlavní činnost a to návrh a analýzu systému, místo na samotnou grafickou podobu modelů a udržení konzistence informací.

Požadavkem na vytvořený nástroj byla i jeho přenositelnost mezi různými operačními systémy. Proto byl zvolen jazyk Java, který je dnes dostupný na každé významné platformě.

6.1 Požadavky na navrhovaný nástroj

Navrhovaný nástroj by měl mít možnost vytvářet 3 typy diagramů a to jsou diagramy případů užití, diagramy tříd a objektově-orientované Petriho sítě.

Pro diagram případů užití musí jít vytvářet elementy aktérů a případů užití a také vztahy mezi nimi. Vztahy mezi aktéry by se měli správně zobrazit jako vztah generalizace a jako vztahy mezi případy užití může nástroj podporovat pouze jako jednoduché vztahy. Vztahy *include* a *exclude* není potřeba podporovat. Viz dále.

Nástroj musí umět automaticky transformovat diagram případů užití na diagram tříd podle následujícího způsobu. Každý aktér a případ užití je převeden na jednu třídu se stejným jménem. Každé třídě musí jít klasicky zadat seznam atributů a metod včetně parametrů. V tomto případě nemusí být nutné uvádět viditelnost atributů nebo metod. Vztahy z diagramu případů užití jsou stejným způsobem převedeny na vztahy mezi korespondujícími třídami v diagramu tříd. S tím, že je zachován vztah generalizace.

Pro každou třídu z diagramu tříd musí nástroj umět vytvořit objektově-orientovanou Petriho síť. Hlavním požadavkem je synchronizace metod z diagramu tříd do sítě daného objektu. Včetně všech parametrů metody.

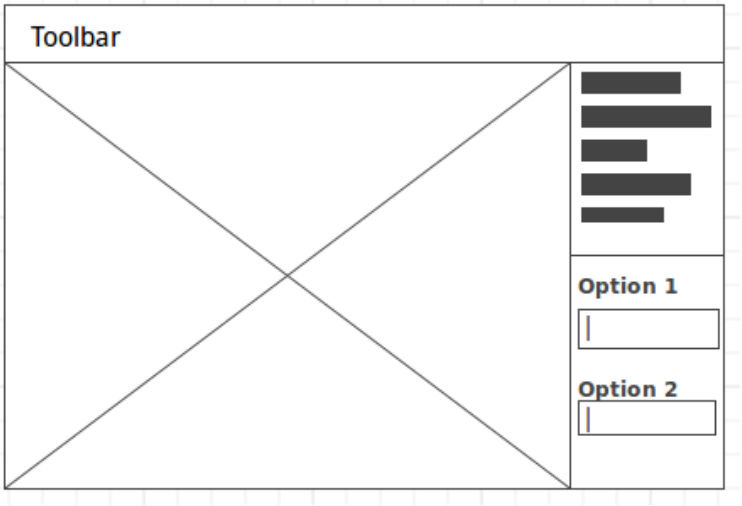
Nástroj musí umět vytvořené diagramy ukládat do souboru a následně je i umět nahrát. Formát není nijak definovaný a může jít o proprietární formát navrhovaného nástroje.

Posledním požadavkem byla přenositelnost programu na hlavní operační systémy. Uvažovali se operační systémy Linux, Windows a MacOS.

6.2 Grafické uživatelské rozhraní

Velké nástroje pro tvorbu různých diagramů většinou mají celou řadu možností a nastavení, které někdy však nejsou pro samotné kreslení třeba. Jejich uživatelské rozhraní je pak plné různých tlačítek a dialogů. Někdy pak trvá nějaký čas, než se v takovém programu nový uživatel zorientuje. Navrhovaný nástroj by měl mít uživatelské rozhraní co nejjednodušší a většinu funkcí potřebných pro vytvoření diagramu by měla být dostupná z hlavního okna aplikace. A tím tedy minimalizovat počet dialogových oken a akcí uživatele. Celé okno aplikace je rozděleno na tři hlavní části. Navrhnutý wireframe okna aplikace je ukázán na obrázku 6.1.

Obrázek 6.1: Wireframe hlavního okna aplikace



Toolbar

Toolbar neboli panel nástrojů tvoří spolu s hlavním menu druhý hlavní ovládací prvek aplikace. Obsahuje často používané tlačítka, které jsou až na některé výjimky společné pro všechny diagramy. Tyto tlačítka jsou tak globální a nezávislé na stavu aplikace nebo právě editovaného diagramu. Podobné panely nástrojů jsou dostupné v celé řadě aplikací a zkracují počet uživatelských akcí, které je nutné použít k vyvolání určité funkce (na rozdíl například od procházení několika úrovňového menu nebo navigací přes několik dialogových oken).

Kreslící plátno

V levé části okna je pak plátno na které se kreslí samotný diagram. Protože aplikace pracuje s několika typy diagramů, které je potřeba upravovat nezávisle, ale stále je potřeba mezi nimi rychle přepínat, jednotlivá kreslící plátna se zobrazují ve formě záložek. Hlavní jsou záložky pro diagram případů užití, z kterého se generuje diagram tříd. Tyto záložky jsou viditelné vždy. Jednotlivé diagramy pro objektově-orientované Petriho sítě daných tříd se také zobrazuji ve formě záložek. Tento styl zobrazení je použit kvůli zmenšení počtu dialogových oken, které jsou použity u celé řady jiných nástrojů pro modelování nebo popis požadavků. Kde se například pro editaci detailu určitého prvku diagramu otevře nové

dialogové okno s možností úpravy nebo se stávající diagram nahradí jiným diagramem. Nepříjemnou vlastností dialogových oken je, že se uživatel musí neustále mezi jednotlivými okny na obrazovce přepínat a ztrácí tak celkový přehled o diagramu. Záložky přesně určují, kde je na obrazovce dostupný jaký diagram a lépe se v modelu orientuje se zachováním jednoho okna aplikace. Jako další výhoda by měla být jednodušší navigace mezi záložkami pomocí klávesnice.

Postranní panel

Postranní panel, zobrazovaný na pravé straně aplikace je hlavní ovládací prvek programu. Skládá se ze dvou částí. První část obsahuje seznam pro výběr prvku, který má být umístěn na kreslící plátno. Tato část panelu je důležitá hlavně pro editaci diagramu případů užití a diagramu objektově-orientované Petriho sítě. Zde si uživatel může zvolit, který grafický prvek chce do diagramu vkládat a poté pomocí dvojkliku může v kreslícím plátně tento prvek vložit. Tento princip eliminuje přidávání prvků pomocí přetahování myší. Kdy uživatel ze seznamu dostupných prvků tyto prvky myší přetahuje na kreslící plátno. Ve větších diagramech to vyžaduje delší dráhu, kterou je nutné myší ujet a celkově se tvorba diagramu zpomaluje. Tento způsob je také vhodnější při použití jiného periferního zařízení jako jsou různé grafické tablety nebo touchpady, které někdy nemusejí být pro práci ve stylu *drag-and-drop* příliš vhodné. Druhá spodní část panelu se zobrazuje po označení grafického prvku na kreslícím plátně. Tato část slouží pro nastavení vlastností tohoto označeného prvku. Obsah panelu se tak může měnit podle označeného prvku v diagramu. Návrh tohoto panelu opět eliminuje použití dialogových oken, které jsou v některých aplikacích použity pro nastavení vlastností daného prvku. Díky tomu, že je panel stále viditelný, uživatel může rychle procházet prvky na kreslícím plátně a hned vidět nebo nastavovat jejich vlastnosti. Neztrácí se tak přehled o vytvářeném modelu.

6.2.1 Modely případů užití

První záložka kreslícího plátna je vždy viditelná záložka pro kreslení modelu případů užití. To je stěžejní diagram, od kterého se odvíjejí všechny ostatní zpřesňující diagramy. Funguje tak jako vstupní model. Momentálně jsou v tomto diagramu navrhovanou aplikací podporovány tyto prvky:

- Aktéři
- Případy užití
- Asociace mezi aktéry a případy užití
- Generalizace mezi aktéry

Momentálně tedy chybí vztahy mezi případy užití jako *include* a *extend*. Tímto omezením mezi všemi podporovanými vlastnosti diagramu případů užití ze standardu UML může být definována transformace mezi případy užití a třídami v diagramu tříd (viz dále).

Editace diagramu funguje na bázi *drag-and-drop* jako u většiny podobných nástrojů. Jednotlivé elementy na plátně lze označit kliknutím myší, kde se poté zobrazí v pravé části okna informace a vlastnosti elementu, které poté lze měnit. Aplikace nyní podporuje změnu jména aktéra nebo jména případu užití.

Pro vytvoření vztahu mezi prvky je nutné podržet klávesu CTRL, kliknout na zdrojový prvek a myš přetáhnout na druhý cílový prvek vztahu. Po označení prvku nebo čáry

představující vztah ji lze klávesou DELETE smazat. Protože však různé prvky v diagramu případů užití mohou mít již obrazy v dopředných modelech, nejsou nové prvky nebo mazané prvky přidány nebo odstraněny okamžitě. Nejdříve je nutné takové změny v diagramu případů užití tzv. odsouhlasit (*commit*). To by mělo zabránit nechtěnému smazání aktéra nebo případu užití a jeho již definované třídě a objektově-orientované Petriho síti a smazat tak část práce. Aplikace tak uživatele upozorní, že smazání prvku má vliv i v dopředných modelech. Na druhou stranu, pokud si uživatel změny rozmyslí, může provést krok zpět (*rollback*). Kdy jsou tyto změny anulovány a uživatel se může vrátit zpět k funkčnímu diagramu případů užití.

6.2.2 Diagramy tříd

Po vytvoření a potvrzení diagramu případů užití se lze přepnout do druhé záložky. Zde je zobrazen diagram tříd. Diagram tříd je založen na transformacích z diagramu případů užití, které mají následující pravidla:

- Aktér je převeden na jednu třídu se stejným jménem
- Případ užití je převeden na jednu třídu se stejným jménem

Jako druhá věc je potřeba převést asociace do diagramu tříd. To lze provést následujícím způsobem:

- Pokud existuje vztah generalizace mezi aktéry v diagramu případů užití, je tato generalizace i mezi korespondujícími třídami v diagramu tříd
- Pokud existuje asociace mezi aktérem a případem užití, existuje tato asociace i mezi korespondujícími třídami v diagramu tříd.

Příklad takového mapování je zobrazen na obrázku 6.2. Kde vlevo je počáteční diagram případů užití a vpravo výsledný diagram tříd.

Mimo identifikaci samotných tříd a vztahů mezi nimi je také třeba definovat atributy a metody těchto tříd. Po označení třídy v tomto diagramu je pak na pravé straně okna aplikace zobrazen panel pro zadání těchto vlastností tříd. Nástroj obsahuje jednoduché prvky pro co nejsnadnější definice těchto informací. Důležité je, že některé vlastnosti jsou automaticky synchronizovány zpět do diagramu případů užití. Momentálně se jedná o jméno třídy.

6.2.3 Objektově-orientovaná Petriho síť

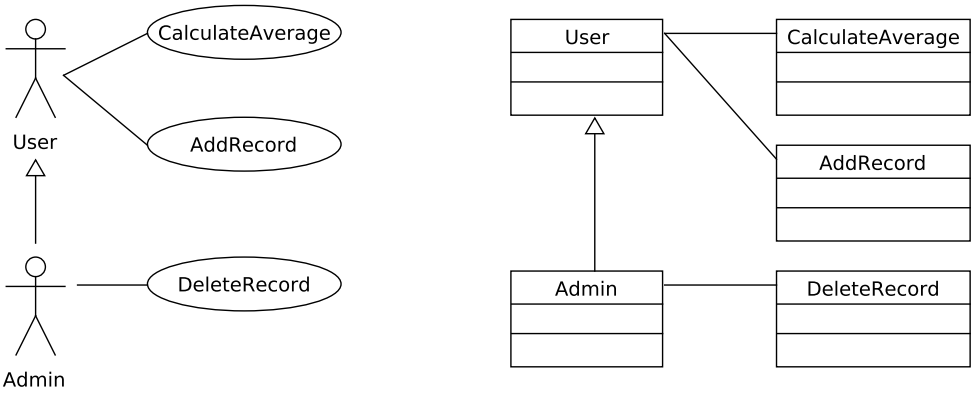
Jako poslední krok po dotvoření diagramu tříd je možnost specifikovat chování každé třídy. Chování každé třídy z diagramu tříd nakonec může být popsáno pomocí objektově-orientované Petriho sítě, která tak zakončuje celou hierarchii modelů. Z pohledu diagramu tříd lze otevřít editor objektově-orientované Petriho sítě poklepáním na zvolenou třídu.

Počáteční Petriho síť je předpřipravena podle definice třídy jako jsou její metody. Zbytek Petriho sítě je pak možné doplnit podle požadovaného chování třídy.

6.3 Import a export diagramů

Program umí ukládat diagramy v jednoduchém formátu XML. Jako textový formát se tak spíše hodí pro správu verzovacími systémy jako je GIT. Takový formát také dovoluje

Obrázek 6.2: Příklad mapování diagramu případů užití na diagram tříd



jednodušší zpracování dalšími nástroji než binární formát souboru. Níže je zobrazena ukázka formátu XML souboru.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<diagram version="1">
  <entities>
    <entity id="0" name="User">
      <usecase type="actor" x="334.0" y="309.0"/>
      <class x="98.0" y="372.0">
        <attributes>
          <attribute>id</attribute>
          <attribute>name</attribute>
        </attributes>
        <methods/>
        <oopn-diagram>
          <oopn-entities>
            <oopn-entity
              id="2"
              marking="C'5"
              name="P_23"
              type="place"
              x="692.0"
              y="133.0"
            />
            <oopn-entity
              id="3"
              name="t1"
              action="guard1"
              guard="guard"
              type="transition"
              x="793.0"
              y="211.0"
            />
```

```
            <oopn-entity
              id="4"
              action="guard4"
              guard="guard"
              type="port"
              x="722.0"
              y="329.0"
            />
          </oopn-entities>
          <oopn-connections>
            <oopn-connection
              to="3"
              from="2"
              type="normal"
              value=""
            />
            <oopn-connection
              to="4"
              from="2"
              type="both"
              value=""
            />
          </oopn-connections>
        </oopn-diagram>
      </class>
    </entity>
  </entities>
  <connections>
    <connection from="0" to="2"/>
    <connection from="0" to="3"/>
    <connection from="0" to="1"/>
  </connections>
</diagram>
```

Celá sada modelů je uložena hierarchicky. Každá značka *entity* představuje jeden případ užití nebo aktéra a jeho asociovanou třídu z diagramu tříd. Značky *usecase* a *class* představující informace potřebné pro zobrazení v diagramu případů užití nebo v diagramu tříd. Využívá se toho, že se tyto prvky (tedy případ užití, případně aktér a třída) mapují jedna ku jedné. Každá taková entita má poté vlastní objektově-orientovanou Petriho síť, která je uložena pod značkou *oopn-diagram*.

Vztahy mezi entitami nejsou z pohledu diagramu samostatné objekty, ale vždy jsou asociované se dvěma entitami. Proto jsou vztahy uloženy separátně pod značkou *connections* (případně *oopn-connections* v případe modelu pro objektově-orientovanou Petriho síť). Každý vztah pak obsahuje indexy entit v seznamu všech entit v diagramu, které tento vztah spojuje.

XML soubor je generován pomocí vestavěných tříd z balíku *javax.xml* a není tedy využívána žádná speciální knihovna. Celý proces tvorby XML stromu má pak na starosti jediná

třída *DiagramXMLExporter*. Pro čtení XML souboru a tvorby diagramu z exportovaného souboru má zase na starosti třída *DiagramXMLParser*.

Kapitola 7

Implementace a testování

V této kapitole je popsán způsob implementace navrhovaného nástroje, použité technologie a postupy. Jako poslední je popsán průběh testování a vyhodnocení tohoto testování.

7.1 Implementace

Technologie

Celý navrhovaný nástroj byl vytvořen v programovacím jazyce Java 8 a frameworkem JavaFX. A to hlavně z důvodu přenositelnosti. Java je totiž dnes dostupná na většině platforem. Java ve verzi 8 byla zvolena hlavně z důvodu nových vlastností, které usnadňují vývoj aplikací využívající grafické uživatelské rozhraní. Využity jsou zejména třídy *Binding*, které fungují na principu návrhového vzoru *Observer*, díky kterému je snadné zajistit udržování konzistentních hodnot mezi různými prvky uživatelského rozhraní. Jako další vlastnost jazyka Java verze 8 jsou lambda funkce. Není tak nutné vytvářet velký počet tříd (případně i anonymních) pro zápis obsluhy prvků uživatelského rozhraní. Takové anonymní třídy jsou navíc dlouhé a hůře se udržují. Jako příklad uveďme následující použití třídy *Runnable*.

```
public class RunnableTest {
    public static void main(String[] args) {
        // Anonymous Runnable
        Runnable r1 = new Runnable() {
            @Override
            public void run(){
                System.out.println("Hello world one!");
            }
        };

        r1.run();
    }
}
```

Celý tento kód se dá pomocí lambda funkce zkrátit na následující tvar.

```
public class RunnableTest {
    public static void main(String[] args) {
        // Lambda Runnable
        Runnable r1 = () -> System.out.println("Hello world two!");
```

```
            r1.run();
        }
}
```

Framework JavaFX pak dovoluje snadnou tvorbu rozhraní pomocí grafického editoru nebo speciálního jazyka FXML založeného na XML. Tím se snadno oddělí prezentační vrstva aplikace od samotné logiky aplikace a je to daleko přehlednější, než vytvářet uživatelské rozhraní programově. Velice podobný postup funguje například na mobilní platformě Android a není tak nic složitého s frameworkem JavaFX začít. Jinak kód navrhovaného nástroje nepoužívá žádné další speciální knihovny. Navíc je možné jednotlivé prvky graficky upravit pomocí speciálního jazyka podobného CSS, který je však upravený speciálně pro FXML. Toto se dobře hodí pro definování grafického vzhledu jednotlivých prvků, které tvoří kreslené diagramy. Jako poslední věcí byla nativní podpora FXML pro lokalizaci textů do různých jazyků. Díky tomu je aplikace přeložena do češtiny a angličtiny.

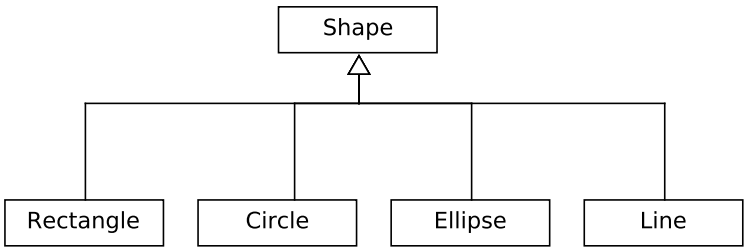
Kreslící plátna

Framework JavaFX má pro umisťování prvků uživatelského rozhraní do okna aplikace různé třídy, které tyto prvky dokáží sdružovat do skupin a prvky ve skupině rozmísťovat určitým způsobem automaticky. Jako příklad mohou být zmíněný třídy *VBox* a *HBox*. Ty zobrazují prvky buď vertikálně pod sebou nebo horizontálně vedle sebe. Obecnou třídou je pak třída *Pane*, která prvky umisťuje na základě souřadnic x a y a pokud se prvky překrývají, tak poslední přidaný do skupiny se zobrazí nahoře.

Jednotlivé grafické prvky, které tvoří jednotlivé typy diagramů jsou složeny ze standardních tříd frameworku JavaFX. Jsou to zejména třídy dědící od třídy *Shape*, která popisuje základní geometrické obrazce. Příklad hierarchie je na obrázku 7.1.

Každá tato třída reprezentuje grafický prvek, který je možné vložit do scény uživatelského rozhraní. A právě tyto jednoduché prvky jsou složeny a umístěny do objektu třídy *Pane*, který tvoří výsledné plátno. Například grafický prvek pro případ užití je složen z jedné elipsy a klasického textového prvku.

Obrázek 7.1: Ukázka hierarchie třídy Shape

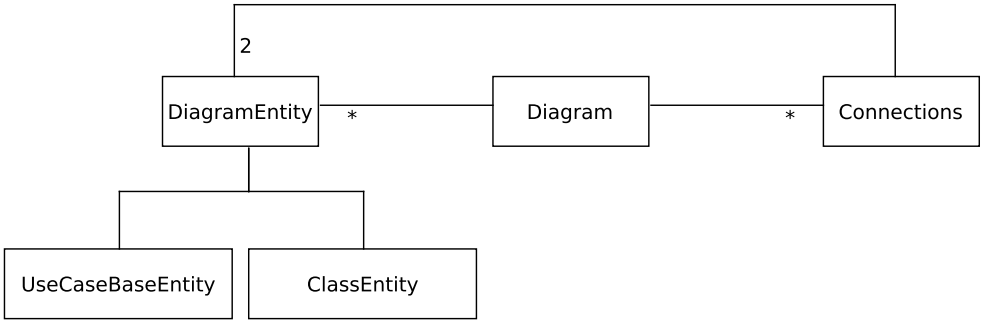


Datový model

Jako první bylo nutné vytvořit třídu pro reprezentaci diagramu případů užití. Ten je popsán třídou *Diagram*. A protože se diagram případů užití mapuje identicky na diagram tříd, obsahuje tato třída informace i pro tento diagram tříd. Základem informací je seznam entit

obsažených v diagramu a seznam vztahů mezi nimi. Aplikace vždy obstarává jeden *Diagram* objekt a to ten, který je momentálně editovaný v aplikaci. Na obrázku 7.2 je pak zobrazen model domény všech tříd, které mají za úkol reprezentovat informace vytvářeného modelu.

Obrázek 7.2: Datový model diagramu

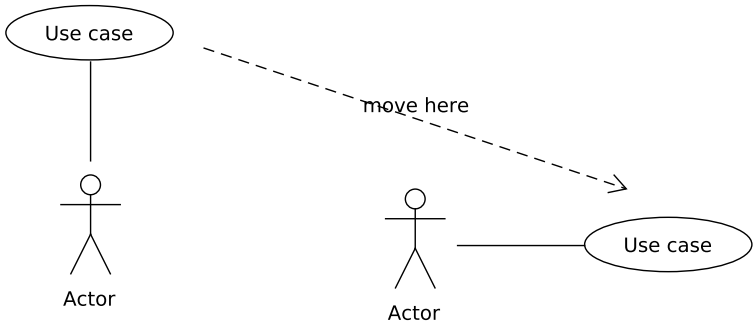


Třída *DiagramEntity* zaštiťuje jednotlivou entitu v diagramu (aktéra nebo případ užití a jejich třídu). Konkrétní objekt pak obsahuje 2 objekty upřesňující informace pro diagram případů užití (objekt třídy *UseCaseBaseEntity*) a informace pro diagram tříd (v podobě objektu třídy *ClassEntity*) potřebné pro korektní vykreslení na kreslícím plátně těchto dvou typů diagramů. Třída *UseCaseBaseEntity* má poté 2 potomky a to jsou třídy *Actor* a *UseCase*. Ty popisují konkrétní typ entity v diagramu případů užití. Všechny tyto třídy, tj. *UseCaseBaseEntity*, jejich potomci a *ClassEntity* obsahují informace jako název entity, informace o grafických prvcích, které představují entity na kreslícím plátně a tak dále. Každý prvek navíc obsahuje čtveřici takzvaných kotev. Což jsou 4 neviditelné body – nahoře, dole, vlevo a vpravo kolem grafického prvku entity. Jejich pozice si každá entita při změně velikosti nebo pozice přepočítává. Z pohledu implementace se jedná o dvojici *DoubleProperty* objektů, představujících x a y pozici kotvy v kreslícím plátně.

Třída *Connection* představuje vztah mezi entitami. Objekt této třídy je vždy asociován se dvěma *DiagramEntity* objekty a rozlišuje se počáteční a koncový vztah relace. Díky tomu je možné správně určit směr vztahu a vykreslit případnou šipku generalizace v diagramu. Každý vztah mezi aktéry (třída *Actor*) navíc obsahuje údaj, že se jedná o vztah generalizace a je tak vykreslena plná šipka. Třída *Connection* kromě této asociace obsahuje pomocné metody, které dokáží zjistit, které entity spojuje a tak dále. Další potřebné údaje pro takový vztah je opět grafický prvek, který na kreslícím plátně zobrazuje tento vztah. V tomto případě je to prostá přímka, reprezentovaná třídou Line opět z balíčku JavaFX frameworku. Protože se však jedná o tenkou přímku, na kterou je těžké kliknout myší, nebylo tak jednoduché odchytit událost zmáčknutí myší nad přímkou a v editoru ji označit. Proto je nad touto přímkou vykreslována daleko tlustší přímka, která má však maximální průhlednost, ale dokáže pohodlně odchytávat takové události myši. Každý konec přímky je poté svázán s pozicí některé kotvy asociovaných entit. Pokud se tedy změní pozice nebo velikost entity v diagramu, vykreslovaná čára vztahu se také automaticky změní. Problém při implementaci třídy *Connection* bylo s umístěním prvku *Line* do kreslícího plátna. Pozice kotev totiž nesmí být v souřadném systému jednotlivé diagramové entity, ale musí být přepočítány do globálního souřadnicového systému kreslícího plátna. V opačném případě by došlo k nesprávnému vykreslování čáry. O tento přepočet se musí starat každá entita po každé změně pozice nebo velikosti. Entita tak musí vědět kde je umístěna, aby tento pře-

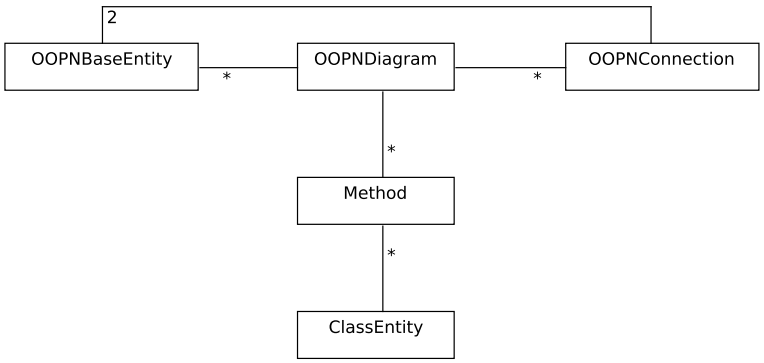
počet mohla vykonat. Vytváří se tak obousměrný vztah mezi entitou a kreslícím plátnem, který je nutné udržovat. Na změnu pozice každého konce čáry je poté zavěšen háček, který se stará o správné přichycení čáry ke spojovaným objektům. Podle relativní pozice spojovaných objektů je zjištěna nejvhodnější kotva objektu, ke které by měla být čára přichycena. Jak to funguje v grafickém rozhraní je potom ukázáno na obrázku 7.3.

Obrázek 7.3: Změna pozice případu užití způsobí automatické přichycení čáry k vhodnější kotvě



Každý objekt třídy *ClassEntity* navíc obsahuje speciální objekt popisující objektově-orientovanou Petriho síť, která této třídě z diagramu tříd patří. V případě této objektově-orientované Petriho sítě je situace s modelem diagramu analogická jako výše. Ukázka hierarchie tříd pro popis této sítě je pak na obrázku 7.4. Avšak zde je situace o něco složitější, protože objektově-orientovaná Petriho síť může obsahovat metody, jejíchž informace jsou primárně uložené v objektu třídy *ClassEntity*. Ty jsou popsány třídou *Method* a je u ní potřeba asociace na *ClassEntity* třídu a *OOPNDiagram* třídu. Každá metoda navíc může obsahovat parametry. Třída *OOPNBaseEntity* je analogií k třídě *DiagramEntity* a opět obsahuje základní informace pro prvek diagramu. Od této třídy poté dědí třídy představující konkrétní prvky objektově-orientované Petriho sítě jako je prvek pro místo, přechod, port a tak dále. *OOPNConnection* je poté analogie k třídě *Connection*, která však obstarává vztahy platné v objektově-orientované Petriho síti. Tato třída také dokáže uložit ohodnocení vztahu a zobrazit ho v diagramu na kreslícím plátně.

Obrázek 7.4: Datový model pro diagram objektově-orientované Petriho sítě



Editace diagramu

Nyní když máme systém tříd pro reprezentaci diagramů, ještě je potřeba mít způsob jak těmito diagramy manipulovat. Pro tento účel je zde trojice tříd nazvaných *UseCaseEditor*, *ClassEditor* a *OOPNEditor*. Všechny tyto třídy mají na starosti zpracovávat hlavní uživatelské akce z uživatelského rozhraní a manipulovat s modelem diagramu. Každý editor je asociován s jednou záložkou v grafickém uživatelském rozhraní. Objekty prvních dvou tříd (tj. UseCaseEditor a ClassEditor) existují stále, stejně jako i záložky pro diagram případů užití a diagram tříd, které jsou viditelné vždy. Jednotlivé objekty třídy OOPNEditor se poté vytváří dynamicky podle toho, které diagramy objektově-orientované Petriho sítě si uživatel zobrazí.

Tyto editory mají za úkol přidávat nové entity, evidovat které entity nebo vztahy jsou označené, mazat označené entity nebo vztahy a tak dále. S tím souvisí i taková logika, kdy je potřeba například při smazání třídy v diagramu tříd odstranit i všechny s ní asociované vztahy a tím i jejich grafické elementy z kreslícího plátna a tak dále. Výjimkou je změna pozice prvků. Ty jsou implementovány standardními událostmi myši frameworku JavaFX. Ty jsou obsluhovány samotnými entitami.

Klávesové zkratky

Framework JavaFX dovoluje relativně jednoduchou definici klávesových zkratek pokud je použito horní menu aplikace. U každé položky menu lze definovat klávesovou zkratku, která je automaticky odchytávána a která poté spouští definovanou akci. To je například případ u klávesy DELETE, která má za úkol smazat objekt z diagramu. Ale některé události klávesnice takovou položku menu neobsahují a je proto nutné programově navěsit akci pro zpracování události. Takto je například implementována funkce pro vytváření vztahů mezi entitami, kdy je nutné podržet klávesu CTRL. Při obsluze události myši je proto nutné programově testovat, jestli je daná klávesa stisknuta nebo ne.

7.2 Další vývoj

Pro další vývoj by se vyplatilo zaměřit na grafický design aplikace. Protože hlavním cílem nebylo za úkol vytvořit hezky vypadající diagramy s různými grafickými efekty, výsledek tak může oproti profesionálním nástrojům vypadat relativně strohý.

Dále pro účely dokumentace by bylo vhodné exportovat vytvořené modely v nějakém vhodném obrázkovém formátu nebo ve formátu PDF. Ty se pak dají vložit například na webovou stránku nebo do jiné dokumentace. Užitečné by to bylo i v případě, když uživatel pracuje jen s diagramem případů užití nebo diagramem tříd, které se používají relativně často, ale nepotřebuje objektově-orientované Petriho sítě. Avšak je třeba diagramy umístit do dokumentace.

Pokud pomineme grafickou stránku, tak jako další funkční vylepšení by se mohlo jednat o napojení vygenerovaných modelů objektově-orientovaných Petriho sítě na některý simulátor. Bylo by tak možné vytvořený model okamžitě validovat a případně podle potřeby upravit návrh. Takový simulátor je již dostupný, bylo by třeba jen navrhnout rozhraní mezi aplikací a simulátorem. Případně nějakým způsobem vhodně prezentovat výsledek simulace přímo v aplikaci.

Jako poslední věc je, že navržený nástroj momentálně neumí pracovat s dědičností v objektově-orientovaných Petriho sítích. V diagramech případů užití, které jsou dále zpřes-

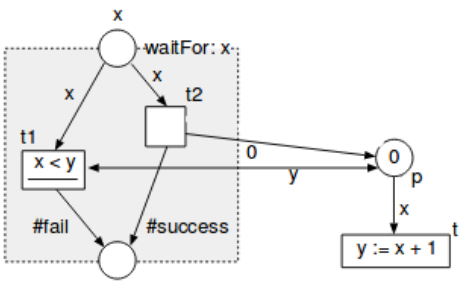
ňovány, se dědičnost využívá jen mezi aktéry. Možnost nástroje je proto momentálně limitován na tvorbu diagramů bez těchto vazeb generalizace mezi aktéry.

7.3 Testování

Testování aplikace bylo provedeno formou testování na uživatelích. Způsob testu byl následující. Všichni uživatelé nejdříve dostali jednoduchý manuál pro aplikaci, který měli za úkol přečíst a seznámit se s ovládáním. Poté uživatelé dostali zadání jednoduchého modelu, který měli za úkol v nástroji vytvořit. Zadání vypadalo následovně:

1. Vytvořte diagram případů užití, který obsahuje jednoho aktéra s názvem „Stroj“. Pro tohoto aktéra vytvořte dva případy užití s názvy „Start“ a „Stop“.
2. Z vytvořeného diagramu případů užití vygenerujte diagram tříd.
3. Pro třídu „Start“ vytvořte dva atributy. Například „time“ a „user“.
4. Pro třídu „Stroj“ vytvořte dvě libovolné metody. První bude bez parametrů a druhá metoda bude mít 3 parametry.
5. Pro třetí třídu „Stop“ vytvořte objektově-orientovanou Petriho síť, podle následujícího zjednodušeného vzoru z obrázku 7.5.

Obrázek 7.5: Zadání objektově-orientované Petriho sítě



Poté co uživatelé úkol dokončili, dostali k vyplnění dotazník. Otázky dotazníku jsou uvedeny níže v tabulce 7.1. Odpovědi uživatelů jsou poté dále v tabulkách 7.2, 7.3 a 7.4.

Z odpovědí v dotaznících bylo vidět, že aplikace je jednoduchá a uživatelé neměli větší problém ji začít používat. Bez větších potíží dokázali vytvořit diagram případů užití a diagram tříd. Chyběli však některé vlastnosti jako možnost udělat krok zpět nebo vpřed při editaci diagramů, které některým uživatelům znepříjemňovalo práci. Jako poslední věc podle reakcí uživatelů byl problém s neznalostí objektově-orientovaných Petriho sítí. Nejspíše by bylo vhodné napsat samotný manuál pro popis tohoto konceptu a vysvětlit podrobjněji principy a možnosti objektově-orientovaných Petriho sítí.

1	Jak se vám zdál složitý manuál k aplikace na pochopení?
	Snadný / Střední / Složitý
2	Jak dlouho se vám trvalo zorientovat v aplikaci po prvním spuštění?
	do 5 minut / do 15 minut / do 30 minut / více jak 30 minut
3	Slovní hodnocení:
4	Jak bylo pro vás složité tvořit diagramy případů užití a diagramy tříd?
	Snadné / Středně těžké / Složité
5	Slovní hodnocení:
6	Jak se vám zdálo složité celkové ovládání aplikace?
	Snadné / Středně těžké / Složité
7	Co se vám na aplikaci líbilo?
8	Co se vám na aplikace nelíbilo nebo vám chybělo?

Tabulka 7.1: Otázky dotazníku

	Běžný uživatel (ví co jsou diagramy případů užití a dokáže vytvořit základní diagram tříd)
1	Snadný
2	Do 15 minut
3	Program má celkem jednoduché ovládání.
4	Snadné
5	-
6	Snadné
7	Jednoduchost
8	Chyběla mi možnost vrátit krok zpět.

Tabulka 7.2: Odpovědi uživatele č. 1

	Pokročilý uživatel (již má povědomí o Petriho sítích)
1	Středně složitý
2	Do 15 minut
3	-
4	Snadné
5	Ze začátku nezvyk z jiné aplikace, kdy se prvky vytváří přetažením z palety prvků.
6	Snadné
7	Automatické přichycení spojů mezi prvky
8	Přichycení spojů někdy není ideální. Chyběla mi možnost undo (pozn. Vrátit krok zpět).

Tabulka 7.3: Odpovědi uživatele č. 2

	Pokročilý uživatel (již má povědomí o Petriho sítích)
1	Snadný
2	Do 15 minut
3	-
4	Snadné
5	-
6	-
7	-
8	Krok pro potvrzení diagramu mi přišel jako zbytečný. Není mi moc jasný princip objektově-orientovaných Petriho sítí.

Tabulka 7.4: Odpovědi uživatele č. 3

Kapitola 8

Závěr

Na začátku této práce byly popsány různé přístupy modelování při tvorbě softwaru. Byl popsán přístup za použití pouze zdrojového kódu, vytváření modelů pro dokumentaci systému ze zdrojových kódů až po model-centrický způsob vývoje, který při tvorbě pracuje pouze s modely, ze kterých se generuje přeložitelný kód. Pro tvorbu modelů byla představena řada nástrojů, které tuto činnost umožňují nebo nějakým způsobem ulehčují. Ve většině případů se jedná o nástroje pracující na bázi jazyka UML, který se snaží být dostatečně univerzální tak, aby jím bylo možné popsat co největší počet situací nebo softwarových částí. To mohou být nástroje jako Microsoft Visio, Enterprise architect nebo jednodušší nástroje jako StarUML nebo UMLet. Mimo to však byly popsány i nástroje pracující s jinými typy diagramů než jen UML. Zmíněný byl nástroj MetaEdit+, který využívá doménově-specifické modelování. V takovém případě si uživatel může sám navrhnout podobu diagramů a různé omezení, které jsou na ně kladeny.

Práce se poté zabývala tvorbou vlastního nástroje pro podporu vývoje softwarových systémů. Tento nástroj je založen na metodě, kdy dokáže popsat uživatelské požadavky pomocí modelu případů užití. Nástroj nepodporuje všechny možnosti tohoto diagramu, ale díky tomu pak umožňuje automatickou transformaci na diagram tříd. Tento diagram pak může být dále upravován a zpřesňován. Chování každé takové třídy je poté popsáno pomocí objektově-orientované Petriho sítě. Přístup tak připomíná princip používaný v technice model-driven-architecture. Přínosem aplikace je, že je multiplatformní a také dostatečně jednoduchá. Podle testování aplikace na uživatelích vyplynulo, že nebyl větší problém se aplikaci naučit používat a vytvářet základní modely. Druhý přínos je použití objektově-orientovaných Petriho sítí, které jako formální model obsahují všechny potřebné výrazové prostředky pro popis chování systému a přitom je možné je formálně analyzovat.

Pro další vývoj aplikace by bylo zajímavé rozšíření o možnost simulace vytvořených objektově-orientovaných Petriho sítí. To může být za pomoci externího simulátoru. Spolu s ním by pak mohl návrhář okamžitě modely validovat. Tím může být dosašeno ještě vyšší míry korektnosti vytvářených modelů. Druhá věc je zaměření na možnost vracet krok zpět nebo vpřed při editaci diagramů. Tato možnost nebyla původně implementována z důvodu způsobu práce pomocí potvrzení (*commit*) změn diagramu. Po delším používání se ale ukázalo, že tato možnost není dostatečně přesná a hodí se vracet kroky zpět i po menších částech.

Aplikace je jako taková funkční a umožňuje tvorbu modelů, jejich ukládání a nahrávání. Jako nejsložitější se pak zdál pouze koncept objektově-orientovaných Petriho sítí, které nejsou mezi běžnými uživateli příliš známé.

Literatura

[1] *Computer programming*. The university of Rhode Island. Department of computer science and statistics, [Online, navštíveno 2017-05-10].
URL http://homepage.cs.uri.edu/faculty/wolfe/book/Readings/Reading13.htm

[2] *Petri net*. Wikipedia, [Online, navštíveno 2017-05-15].
URL https://en.wikipedia.org/wiki/Petri_net

[3] *Use Case Diagrams*. [Online; navštíveno 2017-01-07].
URL https://sourcemaking.com/uml/modeling-business-systems/external-view/use-case-diagrams

[4] *What is Scrum?* Scrum.org, [Online, navštíveno 2017-05-05].
URL https://www.scrum.org/resources/what-is-scrum

[5] *UML Tutorials: The Use Case Model*. Sparx Systems, 2004, [Online, navštíveno. 2017-01-07].
URL http://www.sparxsystems.com.au/downloads/whitepapers/The_Use_Case_Model.pdf

[6] *UML - Class diagram*. 2013, [Online, navštíveno 2017-05-02].
URL https://www.itnetwork.cz/navrhove-vzory/uml/uml-class-diagram-tridni-model

[7] AßMANN, U.: *Automatic Roundtrip Engineering. Electronic Notes in Theoretical Computer Science*, ročník 83, č. 5, 2003, [Online, navštíveno 2017-05-11].
URL http://ac.els-cdn.com/S1571066104807321/1-s2.0-S1571066104807321-main.pdf?_tid=52d879de-364e-11e7-ae79-00000aab0f27&acdnat=1494509748_463882c10bd64e3848e2320290e5e2ed

[8] Brown, A.: *An introduction to Model Driven Architecture*. [Online, navštíveno 2017-03-14].
URL https://www.ibm.com/developerworks/rational/library/3100.html

[9] Cagan, M.: *Dual-Track Agile*. 2012, [Online, navštíveno 2017-05-05].
URL http://svpg.com/dual-track-agile/

[10] Cockburn, A.: *Writing Effective Use Cases*. Boston: Addison-Wesley, 2001, ISBN 978-0201702255.

[11] Esparza, J.; Nielsen, M.: *Decidability Issues for Petri Nets. Basic Research in Computer Science*, 1994, ISSN 0909-0878, [Online; navštíveno 2017-01-05].
URL http://www.brics.dk/RS/94/8/BRICS-RS-94-8.pdf

[12] Janoušek, V.: *Modelování objektů Petriho sítěmi*. Disertační práce, Ústav informatiky a výpočetních techniky FEI VUT v Brně, 1998.

[13] Jawad, K.: *Generation of Programming languages*. [Online, navštíveno 2017-05-10].
URL http://www.byte-notes.com/generation-programming-languages

[14] Kelly, S.; Tolvanen, J. P.: *A guide to domain specific modeling*. MetaCase Software, 2014, [Online, navštíveno 2017-01-07].
URL http://www.embedded.com/design/programming-languages-and-tools/4434608/A-guide-to-domain-specific-modeling---Part-1--Code--vs--model-driven-design

[15] Kočí, R.; Janoušek, V.: *Modeling and Simulation-Based Design Using Object-Oriented Petri Nets: A Case Study. Proceedings of the International Workshop on Petri Nets and Software Engineering*, ročník 851, 2012: s. 253–266, ISSN 1613-0074, [Online, navštíveno 2017-05-15].
URL http://ceur-ws.org/Vol-851/paper19.pdf

[16] Kočí, R.; Janoušek, V.: *Modeling System Requirements Using Use Cases and Petri Nets*. In *ThinkMind ICSEA 2016, The Eleventh International Conference on Software Engineering Advances*, Řím, IT: Xpert Publishing Services, 2016, ISBN 978-1-61208-498-5, s. 160–165.
URL http://www.thinkmind.org/index.php?view=article&articleid=icsea_2016_6_40_10186

[17] Murata, T.: *Petri nets: Properties, Analysis and Applications. Proceedings of the IEEE*, ročník 77, č. 4, 1989: s. 541–580, [Online, navštíveno 2017-05-15].
URL https://inst.eecs.berkeley.edu/~ee249/fa07/discussions/PetriNets-Murata.pdf

[18] Patočka, M.: *Řešení IS pomocí Model Driven Architecture*. Diplomová práce, Masarykova univerzita v Brně, 2008, vedoucí práce RNDr. Radek Ošlejška, Ph.D.

[19] REISIG, W.: *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013, ISBN 978-3-642-33278-4.

[20] Rumbaugh, J.; Jacobson, I.; Booch, G.: *The Unified Modeling Language Reference Manual*. Addison-Wesley, druhé vydání, 2004, ISBN 0-321-24562-8.

[21] Solter, N. A.; Kleper, S. J.: *Professional C++*. Wiley Publishing, 2005, ISBN 0-7645-7484-1.

[22] Zaman, K.: *Dual-Track Scrum*. 2014, [Online, navštíveno 2017-05-15].
URL https://www.scrumalliance.org/community/articles/2014/december/dual-track-scrum