České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jan Lantora**

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: **PCTGen - modul pro podporu analýzy software**

Pokyny pro vypracování:

Pro aplikaci PCTGen [1, 2, 3] implementujte modul, který rozšíří stávající funkcionalitu o podporu analýzy software [4]. Nový modul umožní správu požadavků, případů užití a analytických tříd. Při realizaci se zaměřte na propojení jednotlivých výstupů analýzy mezi sebou tak, aby byl modul schopný ověřit vzájemnou konzistenci záznamů. Pro potřeby zachycení analytických modelů implementujte jednak grafické prostředí pro potřeby vizualizace a jednak textový editor, který využije některý z textových modelovacích jazyků [5].
Očekávané výstupy:
Analýza a návrh rozšiřujícího modulu pro PCTGen.
Implementace modulu včetně vhodných testů implementace.
Vytvoření nástrojů pro kontrolu vzájemné konzistence modelů včetně vhodných úprav textových jazyků.

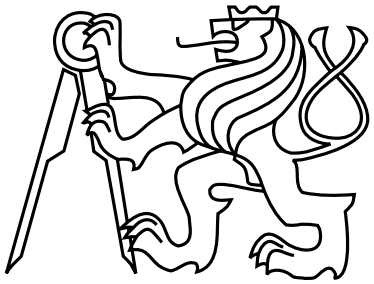
Seznam odborné literatury:

1. Löwinger, L. Aplikace pro generování testovacích situací pro techniku Process Cycle Test. ČVUT. 2014. bakalářská práce
2. Bureš, M.: PCTgen: Automated Generation of Test Cases for Application Workflows. In New Contributions in Information Systems and Technologies, Advances in Intelligent Systems and Computing, vol.353, Springer, 2015, p. 789-794.
3. Bureš, M. et. al.: PCTgen - Process Cycle Test cases generator. http://webing.felk.cvut.cz/bures/pctgen/, 2015
4. Arlow, J. Neustadt, I. 2005. UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition). Addison-Wesley Professional.
5. Mazanec, M., Macek, O. On general-purpose textual modeling languages. In Databases, Texts, Specifications, Objects 2012, MATFYZPRESS, 2012, pp. 1-12.

Vedoucí: Ing. Ondřej Macek, Ph.D.

Platnost zadání: do konce zimního semestru 2016/2017

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

PCTGen - modul pro podporu analýzy software

Bc. Jan Lantora

Vedoucí práce: Ondřej Macek, Ph.D.

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

11. ledna 2016

Poděkování

Chtěl bych tímto poděkovat především celé své rodině za jejich podporu během mého studia, všem přátelům, kteří mi byli nablízku. A v neposlední řádě také vedoucímu práce, panu Ondřeji Mackovi, Ph.D. za odbornou pomoc a rady při tvorbě práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 11. 1. 2016 .....

Abstract

This work deals with design and imlementation of a module into PCTGen tool, which will extend existing functionality to support software analysis. The module will provide management of requirements, use cases and analytic classes. For the purpose of capture the analytical models will be used textual modeling language. The part of the thesis will be analysis of existing textual modeling languages. Based on the evaluation, will be choosen a language, which will be used. The Language will be appropriately extended according to identified deficiencies.

Abstrakt

Tato práce se zabývá návrhem a implementací modulu do nástroje PCTGen, který rozšíří stávající funkcionalitu o podporu analýzy software. Modul bude umožňovat správu požadavků, případů užití a analytických tříd. Pro zachycení analytických modelů bude použit textově modelovací jazyk. Součástí práce je rešerše existujících jazyků, podle které bude zvolen použitý jazyk, který bude na základě nalezených nedostatků patřičně rozšířen.

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Kapitola 1

Úvod

Cílem této práce je navrhnout a implementovat modul do již existující desktopové aplikace PCTGen [10]. Nástroj uživateli umožňuje nakreslení grafu reprezentujícím diagram aktivit v interaktivním grafického editoru a následně z tohoto grafu vygenerovat testovací scénáře dle metody *Process cycle test*[17].

Modul aplikaci rozšíří o možnost zaznamenávání softwarových požadavků, tvorbu modelu tříd a případů užití. Pro zaznamenávání těchto diagramů bude využit textový modelovací jazyk, který bude možné převést do grafické podoby standardu UML.

Mezi další cíle tak patří analýza již existujících textových jazyků, z kterých bude zvolen ten, který nejvíce vyhovuje naším požadavkům. Lze předpokládat, že žádný z textových jazyků plně nesplní naše požadavky a bude tak zapotřebí vybraný jazyk upravit.

1.1 Motivace

Hlavní motivací projektu je větší přehlednost a zjednodušení pracovních procesů pro uživatele, kteří dříve museli využívat rozličné aplikace, pokud chtěli například získat informace o tom, jak se jednotlivé softwarové požadavky váží na případy užití, respektive testovací scénáře.

Další velkou motivací je navržení nového modulu tak, aby respektoval původní aplikaci, tedy jednoduché, přehledné a intuitivní prostředí bez zbytečných pokročilých funkcí.

1.2 Pracovní postup

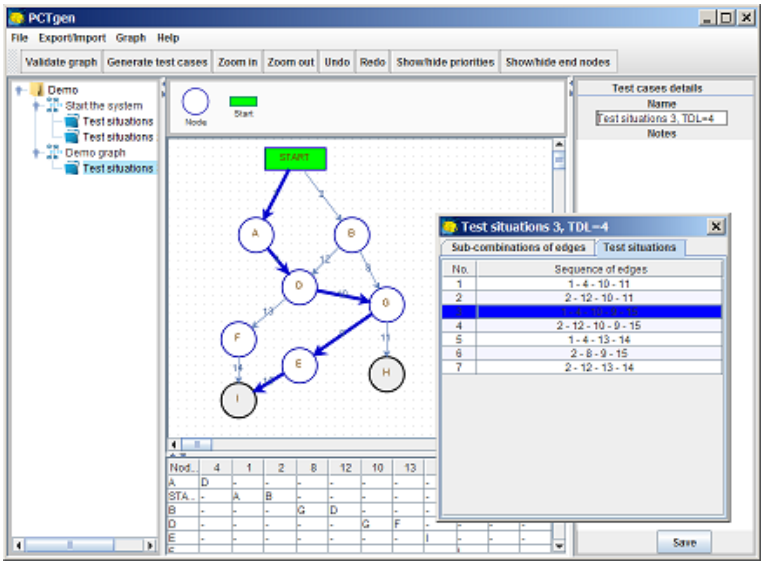
Vývoj projektu bude probíhat iteračním způsobem. Práce bude rozdělena na bloky, kde obsahem bloku budou postupně analýza, návrh aplikace, rešerše a návrh jazyka, implementace a nakonec testování. Každý blok je pak rozdělen na týdenní úseky, po kterých vždy bude následovat představení a konzultování provedené práce. Výsledkem každého bloku by měly být artefakty potřebné pro blok následující.

Kapitola 2

Projekt PCTGen

Funkční testovaní aplikace podle diagramu aktivit je jeden z nejrozšířenějších způsobů testováni. Za účelem snížit náročnost vytváření těchto testů a snížení rizika lidské chyby je vhodné tento způsob testování automatizovat. PCTGen se snaží tento problém řešit. Projekt pokrývá oblasti jako jsou automatizované generování testovacích případů, konzistence testovacích dat. Projekt PCTGen se zaměřuje na zvýšení efektivity a možné vylepšení testovacích metod, procesů a zajištění konzistence dat při testování softwarových aplikací. Jedním z výstupů projektu je i aplikace PCTGen, která slouží k automatickému generování testovacích případů podle definovaných pracovních postupů [9].

Výzkumný tým projektu PCTGen je tvořen M. Burešem, M. Filipským a K. Frajtakem. Aplikace PCTgen byla navržena a implementována v bakalářské práci L. Löwingerem [21] a na vývoji aplikace se dále podílí M. Bureš.



Obrázek 2.1: Ukázka aplikace. Převzato z [10].

2.1 Současné použití

Aplikace je postavena okolo algoritmu na generování testovacích případů. Nástroj nabízí jednoduché, intuitivní řešení, za které se navíc nemusí platit. Aplikace neklade velké nároky na testovaný projekt, například nepředpokládá existenci dokumentace ve formě UML. Navíc je zaručena přenositelnost dat mezi dalšími aplikacemi. Hlavním účelem aplikace je významné snížení času, který je potřebný při vyvářeení testovacích případů manuálně.

2.2 Struktura aplikace PCTGen

S Aplikací se pracuje ve formě samostatných projektů. Obsahem projektu jsou orientované grafy, které v nástroji reprezentují pracovní postupy. V grafu představují uzly rozhodovací body a hrany zastupují akce nebo varianty rozhodování.

Každý projekt muže obsahovat více grafů a každý graf může být zdrojem pro více testovacích scénářů. Aplikace tyto projekty ukládá ve formátu XML s koncovkou „.prj“. Která byla zvolena pro snadnější odlišení od klasických souborů ve formátu XML. Aplikace umí tyto soubory opět načíst a není tak problém na rozdělanou práci později opět navázat. Při načtení projektu se projekt bere jako uložený. Po provedení libovolné změny se projekt opět označí jako neuložený. Zda je projekt aktuálně uležený, je možné vidět na první pohled, neuložené projekty jsou totiž podbarveny červeně a uložené mají barvu zelenou [21].

Každý projekt, graf a jeho části nebo testovací scénář obsahují navíc další meta informace. Konkrétně u projektu je to aktuálně název, popis, odkaz na dokumentaci a verze projektu. Testovací scénář je rozšířen o název a poznámky. Graf má název, popis, vazbu na dokumentaci a prioritu, která je viditelná jak v grafu, tak v meta informacích.

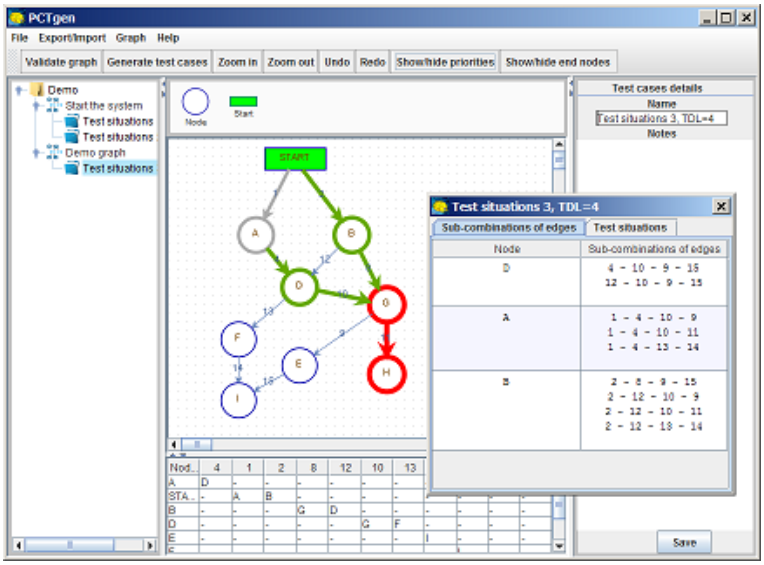
2.2.1 Práce s grafy

Pro tvorbu grafů slouží v aplikaci interaktivní grafický editor nebo je graf možné zadat pomocí tabulky. Z grafu je možné vygenerovat testovací případy. Po vytvoření grafu nabízí aplikace možnost automatického rozložení jednotlivých uzlů pro větší přehlednost.

Před samotným generováním testovacích situací je potřeba, aby graf prošel validací. Validace je tvořena sadou validačních pravidel [21]. V kterých se kontroluje například zda graf není prázdný. Zda v grafu existuje počáteční uzel, nebo že každý uzel je dosažitelný z počátečního uzlu.

2.2.2 Testovací případy

Pro generování testovacích případů ze zadaného grafu je použita technika Process Cycle Test z metodiky TMap Next [18]. Která je navíc rozšířena o algoritmus pro redukci cyklů. Při redukci však může docházet ke ztrátě pokrytí, které definuje test depth level [21]. Po vygenerování testovacích případů nástroj umožňuje pro aktuálně zvolený testovací případ zvýraznit jeho průchod v grafu, z kterého byl vygenerován.



Obrázek 2.2: Ukázka aplikace. Převzato z [10].

2.2.3 Možnosti aplikace

Aplikace obsahuje některé základní prvky snazší ovladatelnosti. Mezi tyto prvky patří schopnost přiblížení a oddálení při vytváření grafu. Nebo možnost jakoukoli změnu vrátit zpět, případně vpřed. Tímto způsobem je možné vrátit i více provedených kroků. Aplikace podporuje multijazyčnost, přičemž v aktuální verzi se jedná o češtinu a angličtinu.

Nástroj kromě ukládání samotného projektu také podporuje export a import pro grafy a testovací případy. Podporované formáty jsou XML, JSON a CSV. Kde pro CSV je nutné dodržet základní pravidla, aby následný import proběhl v pořádku. V případě CSV totiž vzniká problém, jak identifikovat počáteční uzel a je tak nutné, aby u počátečního vrcholu zůstalo jako název klíčové slovo *start*. A takto označený uzel se může v grafu vyskytovat pouze jednou, pokud by toto pravidlo nebylo dodržené, mohou nastat problémy během zpětného importu. Možnostmi import a následného exportu dává nástroj možnost propojení s dalšími aplikacem [21].

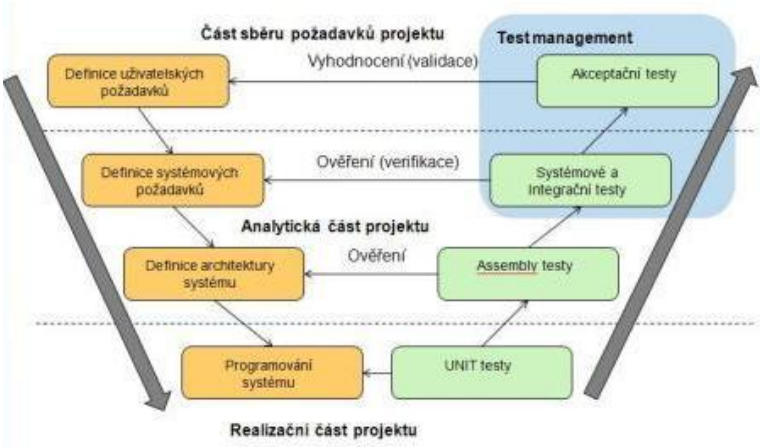
2.3 Nevýhody

Aplikace PCTGen byla navržen pouze za účelem automatizovaného generování testovacích scénářů, čímž se stává jedním z mnoha nástrojů, které tester ke své práci využívá. V mnoha případech u testera nastává okamžik, kdy nestačí pouhé vygenerování testovacích scénářů bez jakékoliv znalostí testovaného systému jako celku. Je proto nucen zjišťovat tyto informace, buď za pomoci dokumentace projektu nebo kontaktování zodpovědných osob. Například v případě určování priority jednotlivých testovacích scénářů, musí byt tester obeznámen s celým systémem, aby mohl tuto prioritu určit. Pro nejsnazší orientaci v testované aplikaci slouží dokumentace v podobě UML diagramů. Pokud ovšem tester chce nahlédnout do takové dokumentace, za podmínky, že vůbec existuje, musí k tomu využít jiný nástroj než

PCTGen. To testera vede k nutnosti umět ovládat více nástrojů a schopnosti mezi nimi plynule přecházet a neztratit se.

2.4 Navrhované použití

Nástroj byl původně navržen za jediným účelem, generování testovacích scénářů podle pracovních postupů. Nikde však není řešeno, kde se postupy vezmou. Tato problematika je popsána pomocí V-modelu. V-model podává přehled o základních implementačních artefaktech a určuje typy testů a jejich vazby na projektové artefakty [41]. Pro nás podstatnou informací, která je z V-modelu patrná je, že testovací scénáře jsou vytvořeny podle případů užití. Příprava případů užití vychází z funkčních a nefunkčních požadavků. Díky takto navrženému procesu je pak možné ověření, že implementovaná aplikace odpovídá business požadavkům nebo možnost nastavování priority testovacím scénářům a generování testovacích dat. Aby bylo možné tyto prvky propojit, je zapotřebí doménový model, který nabízí určitý přehled o budovaném systému.



Obrázek 2.3: V-Model převzat z [1]

V levé části modelu jsou uvedeny fáze vývoje aplikace. V pravé pak jednotlivé druhy testování. Každý druh testů slouží k ověření jiné fáze vývoje. Základem tohoto přístupu je postup testování od malých částí aplikace, přes větší funkční celky, přes integraci komplexních částí aplikace případně integraci více aplikací až po kompletní otestování celé aplikace. Přechod na další druh testování předpokládá úspěšné dokončení předcházející fáze [6].

Z těchto informací jsme dospěli k názoru přidat do nástroje PCTGen analytický modul, který bude pokrývat potřebné body z analýzy aplikace. Jedná se o požadavky na software, které nám pomohou lépe prioritizovat testovací scénáře. Dále se jedná o případy užití a z nich vycházející grafy pro generování testovacích scénářů. A nakonec doménový model, který bude sloužit jako propojení požadavků a případů užití.

Dalším vylepšením je využití databázové struktury pro ukládání projektů. Celý projekt bude uložen v lokálním databázovém souboru, který je bez problémů možné v korporátním prostředí nahradit standardní databází (např. Oracle DB, MSSQL).

Kapitola 3

Úvodní studie

V této kapitole se zabýváme podrobným popisem požadavků na analytický modul pro aplikaci PCTGen. Představíme si seznam funkčních a nefunkčních požadavků, z kterých bude vycházet analýza a návrh. Funkční požadavky jsou rozděleny na čtyři skupiny, kde každá skupina reprezentuje samostatnou část modulu.

3.1 Požadavky software

3.1.1 Funkční požadavky

V této části si představíme seznam funkčních požadavků, z kterých se bude během vývoje aplikace vycházet. Funkční požadavky jsou rozděleny do čtyř skupin, první skupina popisuje společnou funkcionalitu napříč modulem a jeho propojení s původní aplikací. Druhá skupina představuje požadavky, které se týkají operací s analytickými požadavky. Třetí skupina popisuje požadavky pro práci s analytickým modelem. A poslední skupina reprezentuje požadavky pro část jednající o případech užití. K jednotlivým požadavkům je vždy na začátku, v hranaté závorce, přidělena jejich priorita, která je reprezentována pomocí MoSCoW metody [36].

Metoda MoSCoW využívá rozdělení požadavků do čtyř skupin podle jejich výsledné důležitosti(viz níže).

- M - Must have - Požadavek je nutný a bez něj není možné produkt dodat
- S - Should have - Požadavek by měl být zapracován, ale v případě nesplnění je stále možné produkt dodat
- C - Could have - Požadavek by bylo pěkné splnit, ale není pro aktuální dodání podstatný
- W - Won't have - Požadavek není momentálně možné dodat, ale bude připraven v dalším balíku funkcionality.

3.1.1.1 Společná funkcionalita modulu

- **FRQ-1 [M] Editor pro textovou reprezentaci**
  Systém bude obsahovat jednoduchý textový editor pro zápis strukturovaného textu. Jednotlivé elementy(požadavek, entita, případ užití) bude tak možné vytvářet pomocí textového zápisu. Stejně tak bude možné tímto způsobem vytvářet všechny vztahy mezi elementy.

- **FRQ-2 [M] Vztahy mezi diagramy**
  Systém bude umožňovat vytvářet závislosti napříč požadavky, entitami a případy užití. Tato možnost bude přístupná pomocí textové reprezentace. Pro přidání nového vztahu je nutné přidat příslušnou část kódu do textové reprezentace diagramu a ten pak přegenerovat.

- **FRQ-3 [M] Integrace s PCTGen**
  Systém bude integrován do projektu PCTGen jako modul, který bude možné zpětně oddělit.

- **FRQ-4 [S] Export a Import**
  Systém bude umožňovat export a nasledný import jednotlivých diagramů požadavků, Entit a případů užití do formátů XML a JSON.

- **FRQ-5 [C] Tagy**
  Systém bude umožňovat vytvářet a mazat štítky k jednotlivých diagramům požadavků, Entit a případů užití, stejně tak i k jejich vnitřním elementům.

- **FRQ-6 [C] Komentáře**
  Systém bude umožňovat přidávat a mazat komentáře k jednotlivých diagramům požadavků, Entit a případů užití, stejně tak i k jejich vnitřním elementům.

3.1.1.2 Požadavky na požadavky software

- **FRQ-7 [M] Vytváření požadavků**
  Systém bude umožňovat vytvářet nové požadavky. Tato možnost bude přístupná pomocí textové reprezentace požadavku, kde pro přidání nového požadavku, uživatel napíše příslušný příkaz a nechá si vygenerovat takto upravený dokument.

- **FRQ-8 [M] Úprava požadavků**
  Systém bude umožňovat upravovat požadavky. Tato možnost bude přístupná pomocí textové reprezentace požadavku, kde pro úpravu stávajícího požadavku, uživatel najde odpovídající řádek kódu, upraví ho a nechá si vygenerovat takto upravený dokument. Další možností je využití přehledného formuláře, poskytující doplňující údaje, které se v textové reprezentaci nevyskytují.

- **FRQ-9 [M] Mazání požadavků**
  Systém bude umožňovat odstranit již existující požadavky. Tato možnost bude snadno přístupná pomocí textové reprezentace požadavku, kde pro odebrání existujícího požadavku, uživatel najde příslušný řádek kódu a smaže ho, poté přegeneruje takto upravený dokument.

- **FRQ-10 [M] Grafická reprezentace požadavků**
  Systém bude umožňovat graficky zobrazit závislosti mezi požadavky. Tato grafická reprezentace závislostí mezi požadavky se bude generovat z textové reprezentace.

- **FRQ-11 [S] Validace požadavků**
  Systém bude umožňovat validaci požadavků tak, že bude upozorňovat, že k danému požadavku se neváže žádná entita nebo případ užití.

- **FRQ-12 [S] Verzování požadavků**
  Systém bude umožňovat uživateli ke každému požadavku přidělit verzi, která bude odpovídat verzi celého projektu.

- **FRQ-13 [C] Přílohy k požadavkům**
  Systém bude umožňovat přidávat a odebírat přílohy k požadavku. Přílohy je možné reprezentovat odkazem nebo cestou k cílovému adresáři a jménu souboru.

- **FRQ-14 [C] Filtrování nad požadavky**
  Systém bude pro lepší orientaci v seznamu požadavků umožňovat jednoduché filtrování nad požadavky.

- **FRQ-15 [W] Diff verzí požadavků** Systém bude umožňovat zobrazovat rozdíly mezi jednotlivými verzemi požadavků.

3.1.1.3 Požadavky na anylytický model

- **FRQ-16 [M] Vytváření entit**
  Systém bude umožňovat vytvářet nové entity. Tato možnost bude přístupná pomocí textové reprezentace entity, kde pro přidání nové entity, uživatel napíše příslušný příkaz a nechá si vygenerovat takto upravený dokument.

- **FRQ-17 [M] Úprava entit**
  Systém bude umožňovat upravovat entity. Tato možnost bude přístupná pomocí textové reprezentace entity, kde pro úpravu stávající entity, uživatel najde odpovídající řádek kódu, upraví ho a nechá si vygenerovat takto upravený dokument. Další možností je využití přehledného formuláře, poskytující doplňující údaje, které se v textové reprezentaci nevyskytují.

- **FRQ-18 [M] Mazání entit**
  Systém bude umožňovat odstranit již existující entity. Tato možnost bude snadno přístupná pomocí textové reprezentace entity, kde pro odebrání existující entity, uživatel najde příslušný řádek kódu a smaže ho, poté přegeneruje takto upravený dokument.

- **FRQ-19 [M] Vztahy mezi entitami**
  Systém bude umožňovat graficky zobrazit závislosti mezi entitami, podle standardu UML. Pro přidání takového vztahu je nutné přidat příslušnou část kódu do textové reprezentace daného analytického modelu a ten pak přegenerovat.

- **FRQ-20 [M] Atributy u entity**
  Systém bude umožňovat přidávat a upravovat atributy k entitě. Pro přidání nového

atributu je nutné přidat příslušnou část kódu do textové reprezentace daného analytického modelu a ten pak přegenerovat. Pro odebrání je nutné nalézt v kódu příslušný řádek reprezentující daný atribut a ten pak smazat. Pro úpravu atributu je pak možné využít k tomu sloužící přehledný formulář.

- **FRQ-21 [S] Validace analytického modelu**
  Systém bude umožňovat validaci entit, tak že bude upozorňovat, že k dané entitě se neváže žádný požadavek nebo případ užití.

- **FRQ-22 [S] Verzování analytického modelu**
  Systém bude umožňovat verzování entit, atributů i celého analytického modelu.

- **FRQ-23 [C] Přílohy k analytickému modelu**
  Systém bude umožňovat přidávat a odebírat přílohy k entitě, atributu i k celému analytickému modelu. Přílohy je možné reprezentovat odkazem nebo cestou k cílovému adresáři a jménu souboru.

3.1.1.4 Požadavky na případy užití

- **FRQ-24 [M] Vytváření případů užití**
  Systém bude umožňovat vytvářet nové případy užití. Tato možnost bude přístupná pomocí textové reprezentace případu užití, kde pro přidání nového případu užití, uživatel napíše příslušný příkaz a nechá si vygenerovat takto upravený dokument.

- **FRQ-25 [M] Úprava případů užití**
  Systém bude umožňovat upravovat případy užití. Tato možnost bude přístupná pomocí textové reprezentace případu užití, kde pro úpravu stávajícího případu užití, uživatel najde odpovídající řádek kódu, upraví ho a nechá si vygenerovat takto upravený dokument. Další možností je využití přehledného formuláře, poskytující doplňující údaje, které se v textové reprezentaci nevyskytují.

- **FRQ-26 [M] Mazání případů užití**
  Systém bude umožňovat odstranit již existující případ užití. Tato možnost bude snadno přístupná pomocí textové reprezentace případu užití, kde pro odebrání existujícího případu užití, uživatel najde příslušný řádek kódu a smaže ho, poté přegeneruje takto upravený dokument.

- **FRQ-27 [M] Závislosti mezi případy užití**
  Systém bude umožňovat graficky zobrazit závislosti mezi případy užití , podle standardu UML. Pro přidání závislosti je nutné přidat příslušnou část kódu do textové reprezentace daného diagramu a ten pak přegenerovat.

- **FRQ-28 [M] Aktéři u případu užití**
  Systém bude umožňovat přidávat a upravovat aktéry v diagramu případu užití. Pro přidání nového aktéra je nutné přidat příslušnou část kódu do textové reprezentace daného diagramu a ten pak přegenerovat. Pro odebrání je nutné nalézt v kódu příslušný řádek reprezentující daného aktéra a ten pak smazat. Pro úpravu aktéra je pak možné využít k tomu sloužící přehledný formulář.

- **FRQ-29 [S] Scénáře k případu užití**
  Systém bude umožňovat vytvářet,upravovat scénáře k případu užití.

- **FRQ-30 [S] Validace případů užití**
  Systém bude umožňovat validaci případů užití tak, že bude upozorňovat, že k danému případu užití se neváže žádný požadavek nebo entita.

- **FRQ-31 [S] Verzování případů užití**
  Systém bude umožňovat verzování jednotlivých případů užití, jejich aktérů nebo celého diagramu.

- **FRQ-32 [C] Přílohy k případům užití**
  Systém bude umožňovat přidávat a odebírat přílohy k jednotlivým případům užití, aktérům nebo k celému diagramu. Přílohy je možné reprezentovat odkazem nebo cestou k cílovému adresáři a jménu souboru.

3.1.2 Nefunkční požadavky

- **NFR-1 Nezávislost na operačním systému**
  Aplikace bude vyvíjena v jazyku Java. Zvolené vývojové prostředí je multiplatformní, což znamená, že vytvořená aplikace běží na libovolném operačním systému nebo libovolné architektuře. Ke spuštění programu je potřeba pouze to, aby byl na dané platformě instalován správný virtuální stroj. Podle konkrétní platformy se může přizpůsobit vzhled a chování aplikace. ?citace wiki?

- **NFR-2 Databáze**
  Veškerá data se budou ukládat v libovolné JDBC kompatibilní databázi. V základním nastavení bude připravena databáze H2, ale bude podporován snadný přechod na jiné standardní databáze (například Oracle DB, SQL Server, MySQL aj.)

- **NFR-3 Rozšiřitelnost**
  Modul bude navrhnut a implementován tak, aby byl snadno rozšiřitelný, modifikovatelný a spravovatelný.

- **NFR-4 Dokumentace**
  K systému bude dodána jednoduchá uživatelská dokumentace.

Kapitola 4

Rešerše existujících textových jazyků

Tato kapitola se zaměřuje na výběr textového jazyka pro zápis diagramů. Na jazyk je kladena řada nároků. Ať už se jedná o všeobecná kriteria nebo kriteria, která blíže specifikují naší problematiku.

Pro výběr je zvolena nejužívanější množina textových jazyků a podle níže specifikovaných kriterií je zvolen pro nás nejvhodnější. S takto zvoleným jazykem se bude dále pracovat. Nelze předpokládat, že zvolený jazyk bude splňovat všechny naše požadavky. Hledáme pouze takový, který bude nutné rozšířit co nejméně, ale zároveň bude splňovat co nejvíce ze standardních kriterií.

4.1 Kriteria vhodnosti

V této části jsou blíže specifikována kriteria, která poslouží ke zhodnocovaní vybraných textových jazyků. Kriteria jsou rozdělena na obecná. Tedy ty, které by měl splňovat každý textově modelovací jazyk (TML).

Obecná kriteria slouží zejména k ohodnocení použitelnosti při modelování software a jsou založena na článku [23]. Z kriterií je však odebrán důraz na schopnost poslat kompletně software, protože to není v našem případě relevantní požadavek. Místo něj jsou navržena kriteria která blíže specifikují naše nároky.

4.1.1 Schopnost popsat software

Pro standardní popis software je většinou využíván model složený z několika pohledů [19]. Pro naše potřeby však není nutné, aby jazyk byl schopný popsat všechny tyto pohledy. Pravděpodobně ani neexistuje jazyk, který by byl schopný popsat všechny pohledy najednou. Pro každý pohled se tak využívá odlišný jazyk, podobně jako v grafické reprezentaci UML jsou použity odlišné diagramy pro různé pohledy. Podstatným kriteriem je tedy i možnost propojení těchto pohledů.

- **Požadavky**
  Požadavky popisují nároky na software a slouží jako výchozí bod pro všechny pohledy.

- **Logický pohled**
  Logický pohled se zabývá zejména pojmy z problémové domény zadavatele a jejich vzájemnými statickými vztahy.

- **Případy užití**
  V případech užití jsou vyjádřeny základní požadavky kladené na systém. Veškeré další pohledy se pohybují v mantinelech vymezených pohledem případů užití.

- **Propojení pohledů**
  Je podstatné, aby existovaly vazby mezi jednotlivými pohledy a bylo tak možné konstrukt použitý v jednom pohledu rozeznat v pohledu jiném a tvořit tak závislosti mezi jednotlivými pohledy.

4.1.2 Čitelnost a jednoduchost jazyka

Čitelnost jazyka je jedním z nejdůležitějších aspektů při jeho výběru. Je tedy žádoucí, aby TML nabízel srozumitelnou a jednoduchou syntaxi. Pokud je TML dobře čitelný a jednoduchý, nejsou na jeho porozumění kladeny takové technické nároky [23]. Stává se pak dobře přístupný pro větší škálu lidí. Jednoduchost jazyka pak znamená, že i pro vývojáře je snazší se v jazyce orientovat. Navíc, pokud je jazyk dostatečně jednoduchý, je čas strávený učením syntaxe a sémantiky výrazů mnohem kratší.

4.1.3 Jednoznačnost TML výrazů

Nedostatečná jednoznačnost je zásadním problémem grafických modelovacích jazyků. Hledaný textový modelovací jazyk by se tak měl snažit co nejvíce vyhnout mnohoznačnosti jednotlivých výrazů. Nejednoznačné výrazy pak mohou způsobovat problémy při model-to-model transformacích či jiné operace při nichž je vyžadována přesná interpretace [22]. Nejčastější nejednoznačnost způsobují vztahy mezi konstrukty uvnitř modelu. Pokud není dostatečně dodržována jednoznačnost výrazů může docházet k ztrátám informací a v nejhorším případě pak může výsledný model obsahovat zcela jiné informace než-li model původní.

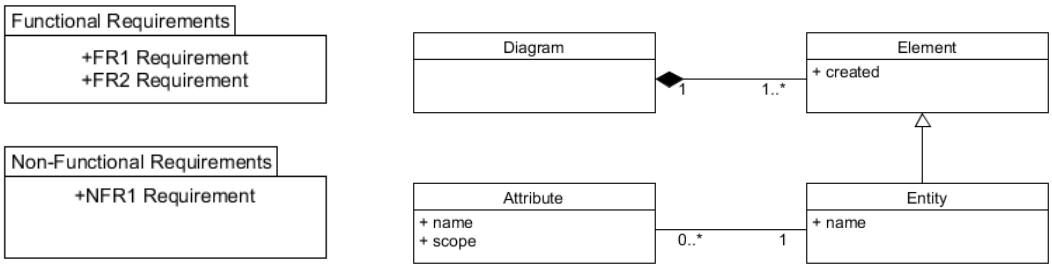
4.1.4 Podpora, Integrovatelnost a Rozšiřitelnost

Požadavek na podporu a integrovatelnost vychází z článku [16]. Kde pojem *podpora* znamená, že pro TML existují nástroje umožňující základní práce s modelem (vytváření, mazání, editace a transformace). Pojem integrovatelnost pak znamená, že jazyk a jeho nástroje je možné jednoduše propojit s dalšími jazyky a nástroji s vynaložením minimálního úsilí. Na integrovatelnost úzce navazuje pojem rozšiřitelnosti. Rozšiřitelnost nám zaručuje, že daný jazyk je možné s minimální pracností rozšířit o další výrazy a konstrukty [23].

4.2 Existující TML

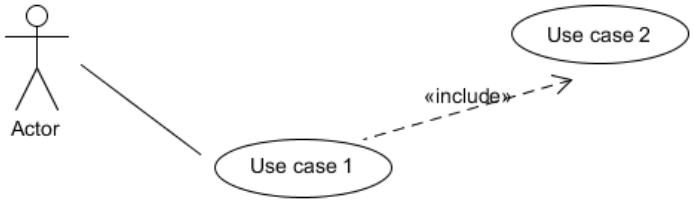
Tato sekce obsahuje přehled používaných textově modelovacích jazyků a jejich porovnání podle výše zmíněných kriterií v sekci 4.1. Podle nichž se rozhodne, který jazyk bude nakonec použit při implementaci analytického modulu do nástroje PCTGen.

Pro snadné porovnání čitelnosti a jednoduchosti zvolených TML budou využity jednoduché vzorové UML diagramy, které budou reprezentovat jednotlivé pohledy na software. Každý diagram bude vždy přepsán do textové formy daného jazyka a na konci kapitoly bude provedeno porovnání s ostatními jazyky. Na obrázku 4.1 se nachází tři modely, seznam požadavků, model tříd a případ užití. Obrázek zachycuje diagramy, které budou používány v analytickém modulu aplikace PCTGen, nesnažíme se tak o kompletní popis všech pohledů na software.



(a) Diagram požadavků

(b) Diagram tříd



(c) Diagram případů užití

Obrázek 4.1: UML diagramy použité pro porovnání textových modelovacích jazyků

4.2.1 PlantUML

PlantUML [40] je sada jazyků, která je schopná popsat nejvíce pohledů na softwarový systém ze všech testovaných jazyků. Dokáže popsat software na různých úrovních abstrakce. Mezi diagramy, které je schopný pomocí textové reprezentace popsat, je sekvenční a stavový diagram, diagram aktivit a tříd, případy užití, a mnoho dalších. Dokonce je schopný i vytvářet prototypy obrazovek ve formě wireframe [38]. Bohužel nepodporuje propojení mezi jednotlivými pohledy a případy užití jsou reprezentovány pouze pomocí diagramu a neexistuje tak možnost jak zapsat scénáře.

Nevýhodou PlantUML může být obtížnější srozumitelnost, zejména v případě zápisu vztahů mezi elementy diagramu. Jazyk totiž z velké části kopíruje grafickou podobu UML značek. Pokud je uživatel znalý UML nebude pro něj zápis problém, ale v opačném případě může být forma zápisu nepřehledná.

PlantUML nesplňuje vlastnost jednoznačnosti výrazů jazyka. Každý jazyk ze sady má definovanou svoji gramatiku, která obsahuje informace o strukturálních pravidlech. Avšak neexistuje matematická formální definice, jež by přesně definovala sémantiku výrazu jazyku.[22]

Listing 4.1: Diagram tříd PlantUML

```
class Diagram {}

class Element {
        +created
}

class Entity {
       +name
}

class Attribute {
       +name
       +scope
}

Diagram "1" * -- "1..* " Element
Element <| -- Entity
Attribute "0..*" -- "1" Entity
```

PlantUML má široký rozsah možností integrace. Jako plugin lze využít přímo ve vývojovém prostředí (Eclipse) nebo v prostředí Confluence. Zvládá i podporu pro textové editory jako Word, Open Office nebo VIM. Podporuje i rozsáhlou sadu programovacích jazyků, mezi které patří Java, Javascript, Python, PHP nebo Groovy. Umožňuje i tvorbu grafických UML diagramů přímo ve zdrojovém kódu pomocí speciálních komentářů.

Listing 4.2: Případ užití PlantUML

```
actor Actor

Actor -- (Use case 1)

(Use case 1) .> (Use case 2) : include
```

PlantUML je open-source projekt a je možné ho upravovat pod různými licencemi. Mezi použitelné patří licence MIT, GPL, Eclipse Public License(EPL) a jiné. Celý seznam a mnoho dalších informací je možné získat přímo na stránkách PlantUML [40].

4.2.2 yUML

Jazyk yUML [43] funguje ve formě webové služby a vznikl za účelem rychlého a jednoduchého zápisu diagramu tříd a aktivit. Jazyk je však v aktuální verzi schopný i popisu případů užití, ale stejně jako PlantUML pouze ve formě diagramu případu užití bez zápisu scénáře.

Listing 4.3: Diagram tříd yUML

```
[ Diagram ] ++1 -1..*[ Element | created ]
[ Element ]^[ Entity | name ]
[ Entity ]<1 -0..*[ Attribute | name ; scope ]
```

Sada jazyků není vhodná pro zápis větších modelů a s přibývající složitostí se stává vysoce nečitelnou. To je způsobeno nutností řetězení všech konstruktů. yUML je především webovou službou pro rychlý převod textového modelu do grafické podoby. Z tohoto důvodu je tomu uzpůsobena i forma zápisu, která má být, co nejvíce jednoduchá.

Listing 4.4: Případ užití yUML

```
[Actor]-(Use case 1)>(Use case 2)
```

Absence formální definice sémantiky výrazu sady jazyků yUML způsobuje nejednoznačnost. Neexistuje veřejně dostupná verze gramatik yUML.

yUML je webovou službou, která je dostupná pouze online a není k dispozici jiným způsobem. Na rozdíl od ostatních jazyků se zde jedná o komerční projekt a možnost rozšiřitelnosti je tak značně omezená.

4.2.3 TextUML

TextUML [12] je jazyk, který je určený pouze pro zápis logického pohledu a v aktuální verzi tak není možné zapsat jiné pohledy na software. Jeho rozsah je tak značně limitován a znamenalo by to v našem případě značná rozšíření jazyka. Do budoucna se očekává, že bude jazyk rozšířen o další pohledy, ale není určen termín, kdy nová verze vyjde.

Na druhou stranu jazyk disponuje dobrou čitelností a nevyžaduje nutně znalost grafické podoby UML. Ale některé konstrukty jsou v zápise nadbytečné a i když není nutné je použít je nutné, aby v zápise byly obsaženy.

Vytvořené konstrukce v jazyce TextUML nejsou jednoznačné a pro jazyk neexistuje veřejně dostupná verze matematické formální definice sémantiky výrazu.

Listing 4.5: Diagram tříd TextUML

```
package testpackage;

        class Diagram
        end;

        class Element
                attribute created;
        end;

        class Entity specializes Element
                attribute name;
        end;

        class Attribute
                attribute name;
                attribute scope;
        end;

        association
                navigable role name : Entity [1];
                role type : Attribute [*];
        end;

        composition
                navigable role parent : Diagram [1];
                navigable role child : Element [*];
        end;
end.
```

TextUML je možné používat jako plugin ve vývojovém prostředí Eclipse. A je dostupný ve formě open-source pod licencí EPL z úložiště GitHub [11].

4.2.4 Umple

Jazyk Umple [13] je použit především k model-oriented programming [8], které je založené na diagramu tříd a stavovém diagramu a slouží ke generování kódu do jazyků jak Java, Ruby a další. Zmíněné diagramy jsou jediné, které lze pomocí jazyku Umple popsat. Některé

pohledy na software tak chybí a pro tvorbu případů užití a diagramu požadavků by bylo nutné rozšíření jazyka.

Stejně jako jazyk TextUML i Umple disponuje dobrou čitelností, ale opět jako v případě PlantUML vychází některé konstrukty z grafické podoby UML. Výhodou oproti TextUML však je, že konstrukty, které nejsou v zápise nutné, nemusí být zapsány.

Jednoznačnost výrazů není sadou jazyků Umple splněna. Jednotlivé jazyky mají jako jediné veřejně dostupné gramatiky, ale postrádají existenci matematické formální definice sémantiky výrazu [22].

Listing 4.6: Diagram tříd Umple

```
class Diagram {
}

class Element {
        created;
}

class Entity {
        isA Element;
        name;
}

class Attribute {
        name;
        scope;
}

association {
        0..* Attribute -> 1 Entity;
}

association {
        1 Diagram -- 1..* Element;
}
```

Jazyk Umple je dostupný jako plugin do vývojového prostředí Eclipse, ale navíc je dostupný i online pomocí webové služby UmpleOnline [20]. Obě formy jsou zpětně kompatibilní. Jazyk Umple je dostupný pod licencí MIT.

4.2.5 UML Graph

Jazyk UML Graph [42] má značně podobnou formu zápisu jako jazyk Java. Jednotlivé elementy diagramu jsou popsány ve formě kódu a vztahy mezi těmito elementy jsou popsány pomocí speciální formy dokumentace připomínající Javadoc. Jazyk je schopný popsat pouze logický pohled na software. Pro ostatní pohledy by byla nutná další rozšíření jazyka.

Nevýhodou jazyka může mít obtížnější srozumitelnost v případě neznalosti programovacího jazyka Java.

Listing 4.7: Diagram tříd UML Graph

```
/** @has 1 features * Element */
class Diagram {
}

class Element {
        public created;
}

class Entity extends Element {
        public name;
}

/** @navassoc * - 1 Entity */
class Attribute{
        public name;
        public scope;
}
```

Jednoznačnost výrazů není sadou jazyků UMLGraph splněna. Jednotlivé jazyky mají veřejně dostupné gramatiky, avšak neexistuje jejich formální definice [22].

Sada jazyků je dostupná jako knihovna napsána v jazyce Java. Jsou však volně k dispozici i zdrojové kódy a je tak možnost rozšířit jazyk dle vlastní potřeby. UML Graph je vázán licencí CC BY-SA 3.0 [27].

4.2.6 MetaUML

MetaUML [25] je sada jazyků, která je schopná popsat řadu pohledů na softwarový systém. Mezi diagramy, které je schopný pomocí textové reprezentace popsat, je stavový diagram, diagram aktivit a tříd, případy užití, a jiné. Bohužel nepodporuje propojení mezi jednotlivými pohledy a případy užití jsou reprezentovány pouze pomocí diagramu a neexistuje tak možnost jak zapsat scénáře.

Listing 4.8: Diagram tříd MetaUML

```
Class.A("Diagram")
        ()
        ();

Class.B("Element")
        ("+created")
        ();

Class.C("Entity")
        ("+name")
        ();

Class.D("Attribute")
        ("+name",
          +scope)
        ();

clink(inheritance)(C, B);

clink(associationUni)(C, D);
        item(iAssoc)("1")(obj.nw = C.e);
        item(iAssoc)("0..*")(obj.ne = D.w);

clink(composition)(B, A);
        item(iAssoc)("1..*")(obj.nw = B.e);
        item(iAssoc)("1")(obj.ne = A.w);

drawObjects(A, B, C, D);
```

Nevýhodou jazyka je obtížnější čitelnost, která je převážně způsobena nutností zápisu informací, které nejsou využité. Také zápis vztahů je v některých případech komplikovaný. Pokud chceme textovou reprezentaci graficky vykreslit, je nutné zadat pozicování jednotlivých elementů ručně.

Listing 4.9: Případ užití MetaUML

```
Actor.A("Actor");

Usecase.B("Use case 1");
Usecase.C("Use case 2");

clink(association)(A.human, B);

clink(association)(B, C);
        item(iAssoc)("<<include>>")(obj.s =.5[B.e,C.w]);

drawObjects(A, B, C);
```

Jazyk MetaUML je veřejně dostupný na uložišti GitHub[24], ale navíc je dostupný i online pomocí webové služby MetaUMLOnline [26]. Obě formy jsou zpětně kompatibilní. Jednoznačnost výrazů není sadou jazyků MetaUML splněna. Jednotlivé jazyky mají veřejně dostupné gramatiky, avšak neexistuje jejich formální definice. Jazyk MetaUML je k dispozici pod licencí GNU GPL.

4.3 Vyhodnocení

Žádný z testovaných jazyků nesplnil všechna kriteria, která byla definována v sekci 4.1. Testované jazyky pro textové modelování mají řadu nedostatků, zásadním problémem je omezený počet pohledů na software, které jsou schopné popsat. Další problém, který mají všechny TML společný je, že se nesnaží popsat softwarový systém jako takový, pouze se snaží přepsat grafickou reprezentaci UML. To způsobuje častou ztrátu použitelnosti a čitelnosti.

Ani jeden z jazyků není schopný popsat požadavky na software. Tří jazyky jsou schopné popsat diagram případů užití, ale žádný z nich není schopný popsat scénáře k těmto diagramům. Dalším nedostatkem u všech testovaných jazyků je nemožnost vytvoření vazeb mezi jednotlivými diagramy.

	Požadavky	**Logický pohled**	**Případy užití**	**Propojení pohledů**
PlantUML	✗	✓	✓	✗
TextUML	✗	✓	✗	✗
Umple	✗	✓	✗	✗
yUML	✗	✓	✓	✗
UML Graph	✗	✓	✗	✗
MetaUML	✗	✓	✓	✗
UML	✓	✓	✓	✓

Tabulka 4.1: Schopnost popsat software

Kritérium jednoznačnosti není ani jedním z jazyků splněno. Většina jazyků disponuje veřejně dostupnou verzí gramatik, ale postrádají jakoukoliv matematickou formální definici sémantiky výrazů jazyka. Naopak kriterium podpory a integrovatelnosti je splněno u všech jazyků. Ale v případě jazyka yUML je vyžadována dostupnost webové služby. Většina jazyků také splňuje i kriterium rozšiřitelnosti. Kriteria čitelnosti je hodnoceno z pohledu využití námi hledaného jazyka. Očekává se tak od uživatele znalost UML.

	Čitelnost a jednoduchost	**Jednoznačnost**	**Integrovatelnost a podpora**	**Rozšiřitelnost**
PlantUML	✓	✗	✓	✓
TextUML	✓	✗	✓	✓
Umple	✓	✗	✓	✗
yUML	✗	✗	✓	✗
UML Graph	✓	✗	✓	✓
MetaUML	✗	✗	✓	✓

Tabulka 4.2: Přehled splněných kriterií

Cílem této rešerše bylo najít nejvhodnější jazyk, jehož syntaxe by byla snadno srozumitelná, intuitivní a podporovala základní prvky daného modelu. Po otestování vybraných jazyků byl jako nejvhodnější kandidát vybrána sada jazyků PlantUML, která jako jedna z mála podporuje model pro případy užití, ale především nabízí poměrně snadnou syntaxi, která je intuitivní a do určité míry i přehledná.

Kapitola 5

Analýza a návrh řešení

V následující kapitole si rozebereme model analytických tříd. Podrobně popíšeme jednotlivé entity vyskytující se v analytickém modulu. Představíme grafické rozhraní pro nový analytický modul. Navrhneme změny do sady jazyků PlantUML tak, aby splňoval naše požadavky. A na závěr si probereme technologie, které budou využity pro implementaci modulu.

5.1 Model analytických tříd

V této části si představíme a podrobněji popíšeme entity vyskytující se v systému. Entita vyobrazuje objekt z reálného světa a obsahuje vztahy s jinými entitami.

5.1.1 Model pro Společnou funkcionalitu modulu

5.1.1.1 AnalysisElement

Entita AnalysisElement je základem celého analytického modulu. Jedná se o abstraktní entitu, z které vychází všechny elementy, které je možné vidět v diagramech. Zároveň může představovat i celý diagram.Jedná se o stěžejní entitu, na kterou navazují všechny společné atributy. Dále také uchovává informace o čase vytvoření a čase poslední změny, která byla na elementu provedena.

5.1.1.2 ElementType

Entita ElementType reprezentuje rozhraní, z kterého dědí všechny výčtové type Enum, které slouží pro určení jakého typu je element (AnalysisElement 5.1.1.1). U diagramu(Diagram 5.1.1.3) je implementován pomocí výčtu DiagramType 5.1.1.4, u entit reprezentující požadavky(Requirement 5.1.2.1) jde o entitu RequirementType 5.1.2.2. AnalysisEntity 5.1.3.1 reprezentující entity v doménovém modelu, využívá enum AnalysisEntityType 5.1.3.2. Pro entitu Relation 5.1.3.4, která slouží pro uchování vztahů mezi entitami AnalysisEntity 5.1.3.1, je použita implementace RelationType 5.1.3.5. V diagramech případů užití je k dispozici výčet UseCaseType 5.1.4.2. Pro určení typu relace u aktérů (ActorRelation 5.1.4.4), je použita entita ActorRelationType 5.1.4.5. Jednotlivé scénáře případu užití (Scenario 5.1.4.6) jsou

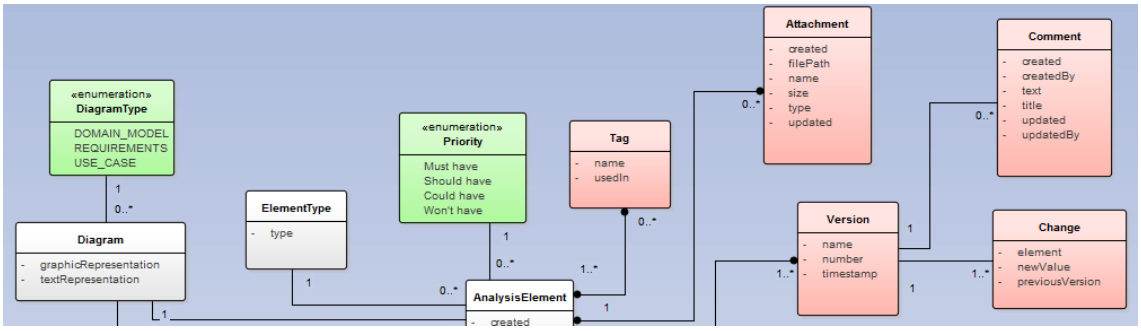
rozděleny na typy podle ScenarioType 5.1.4.7. Pro zvolení vhodného vztahu mezi případy užití uvnitř diagramu je využit výčet UseCaseRelationType 5.1.4.11.

5.1.1.3 Diagram

Entita Diagram slouží k reprezentaci UML diagramu. Diagram se dělí podle entity DiagramType 5.1.1.4 na diagram požadavků, diagram tříd nebo na diagram případů užití. Diagram je potomkem entity AnalysisElement 5.1.1.1. Zároveň obsahuje seznam entit AnalysisElement 5.1.1.1, které představují elementy uvnitř diagramu. Diagram obsahuje dva podstatné atributy. První atribut obsahuje odkaz na soubor, ve kterém je uložena textová reprezentace celého diagramu. Druhý atribut ukazuje na cestu k souboru, kde je uložena grafická reprezentace diagramu.

5.1.1.4 DiagramType

Entita DiagramType slouží k určení typu diagramu (Diagram 5.1.1.3). V analytickém modulu nástroje PCTGen jsou použity tři typy diagramů. První typ je diagram požadavků, ve kterém jsou podrobně popsány funkční a nefunkční požadavky testovaného systému. Druhý typ je diagram tříd, který popisuje doménový model testovaného systému. Poslední typ diagramu je diagram případů užití, který slouží pro pochopení funkcionality systému a jako předloha pro návrh grafu podle, kterého jsou generovány testovací scénáře.



Obrázek 5.1: Návrhový model společné funkcionality modulu

5.1.1.5 Priority

Entita Priority představuje způsob, jak v systému určit, jaký element má vetší důležitost a měl by být zpracováván přednostně. K určení priority se používá metoda MoSCoW. Prioritu je možné určovat u libovolné entity, která vychází z AnalysisElement (5.1.1.1).

5.1.1.6 Tag

Entita Tag je použita pro systém štítkování. Libovolné entity typu AnalysisElement 5.1.1.1, je možné označit jedním nebo více štítky (tagy). Podle těchto štítků je pak možné vyhledávat napříč aplikací a najít tak všechny elementy, které mají podle štítku něco společného.

5.1.1.7 Attachment

Entita Attachment se váže na entitu AnalysisEntity 5.1.3.1 a slouží k uchovávání informací o přílohách, které patří k danému elementu. Udržují se zde informace o velikosti přílohy. Dále je zde uvedena adresářová cesta k příloze a o jaký typ přílohy se jedná.

5.1.1.8 Version

Entita version je implementací verzování v analytickém modulu. Verze se váže na každý element (AnalysisElement 5.1.1.1). Verzování je provedeno inkrementálně. Při přechodu na novou verzi jsou zaznamenány všechny změny oproti původní verzi a je tak možné sledovat historii projektu.

5.1.1.9 Change

Entita Change představuje jednu změnu, která byla provedena při přechodu na novou verzi (Version 5.1.1.8). U změny se detekuje element (AnalysisElement 5.1.1.1), u kterého změna proběhla, atribut, který byl změněn, jeho aktuální a původní hodnota.

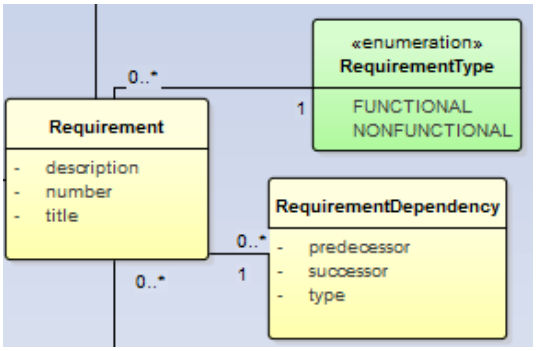
5.1.1.10 Comment

Entita Comment reprezentuje způsob jakým je možné přidělit komentář k libovolnému elementu (AnalysisElement 5.1.1.1). U komentáře je vždy uveden titulek a pak text se samotným komentářem.

5.1.2 Model pro Požadavky

5.1.2.1 Requirement

Entita Requirement představuje požadavek na testovaný systém. Požadavky se dělí na funkční a nefunkční. Požadavky jsou součástí diagramu požadavků. Požadavky jsou očíslované pro snazší možnost odkazování se na ně.



Obrázek 5.2: Návrhový model pro část zabývající se požadavky

5.1.2.2 RequirementType

Entita RequirementType má za úkol rozdělit jednotlivé požadavky (Requirement 5.1.2.1) podle jejich typu. Požadavek může být dvou typů, funkční požadavek, který popisuje chování systému. Nebo nefunkční požadavek, který popisuje vlastnosti a omezení služeb, jenž jsou systémem nabízeny.

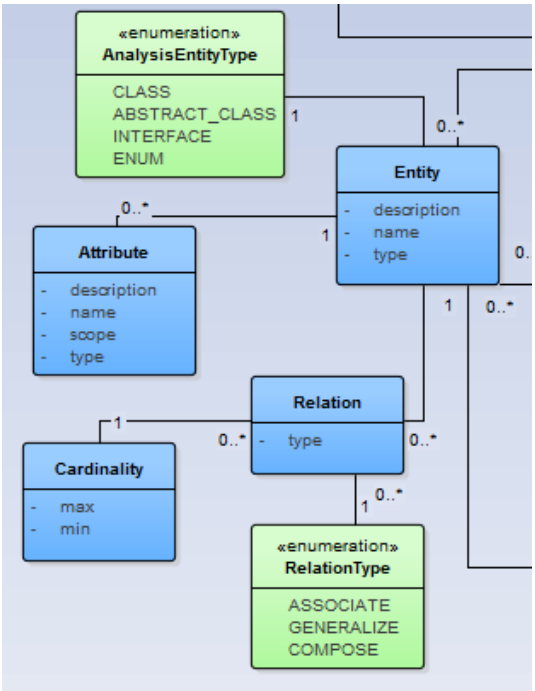
5.1.2.3 RequirementDependency

Entita RequirementDependency popisuje vztahy mezi požadavky (Requirement 5.1.2.1). Jednotlivé požadavky mohou navazovat na sebe, případně jeden může rozšiřovat druhý. Tyto vztahy jsou popsány právě touto entitou.

5.1.3 Model pro Analytický model

5.1.3.1 AnalysisEntity

Entita AnalysisEntity je klíčovým prvkem pro tvorbu analytického modelu. V modelu reprezentuje prvek třída, na který se váží relace (Relation 5.1.3.4) a obsahuje jednotlivé atributy (Attribute 5.1.3.3). Entita je rozdělena na více typů, typ je určen pomocí entity AnalysisEntityType 5.1.3.2.



Obrázek 5.3: Návrhový model pro část zabývající se analytickým modelem

5.1.3.2 AnalysisEntityType

Entita AnalysisEntityType je použita k určení typu AnalysisEntity 5.1.3.1. Využití nabývá zejména v případech, kdy se jedná o návrhový model. Slouží k rozlišení zda se jedná o entitu typu abstraktní třída, třída, rozhraní nebo výčet.

5.1.3.3 Attribute

Entita Attribute představuje atributy entity (AnalysisEntity 5.1.3.1). Podle typu modelu pak rozlišujeme jejich specifičnost. Atribut může obsahovat informace jakého je datového typu nebo jakou má přístupnost (public, protected, private, atd.).

5.1.3.4 Relation

Entita Relation slouží k reprezentaci vztahů mezi entitami (AnalysisEntity 5.1.3.1). Vazby mezi entitami mohou být specifikovány pomocí entity RelationType 5.1.3.5, která určí o jaký typ vazby se jedná.Násobnost vztahu je vytvořena entitou Cardinality 5.1.3.6, která slouží pro definování kolik výskytů jedné entity a kolik výskytů druhé entity může vstoupit do vztahu.

5.1.3.5 RelationType

Entita RelationType je použita k určení typu vztahu (Relation 5.1.3.4) mezi entitami (AnalysisEntity 5.1.3.1). Vztah může být typu dědičnost nebo asociace, agregace, kompozice a další.

5.1.3.6 Cardinality

Entita Cardinality určuje mocnost vztahu mezi entitami (AnalysisEntity 5.1.3.1). Vyjadřuje skutečnost kolik výskytů jedné entity může vstoupit do vztahu s kolika výskyty druhé entity.

5.1.4 Model pro Případy užití

5.1.4.1 UseCase

Entita UseCase představuje jeden případ užití v diagramu případů užití. Zaznamenáváme u něj kromě obecných parametrů z entity AnalysisElement 5.1.1.1 také číslo, podle kterého je jednotně identifikován. UseCase v sobě nese informace o aktérech (Actor 5.1.4.3), kteří v případu užití figurují. Dále obsahuje scénáře případu užití. Entita UseCase může být také vázána na jiné případy užití pomocí entity UseCaseRelation 5.1.4.10. Pro každý případ užití také mohou být definovány vstupní a výstupní podmínky a to pomocí entity Condition 5.1.4.9.

5.1.4.2 UseCaseType

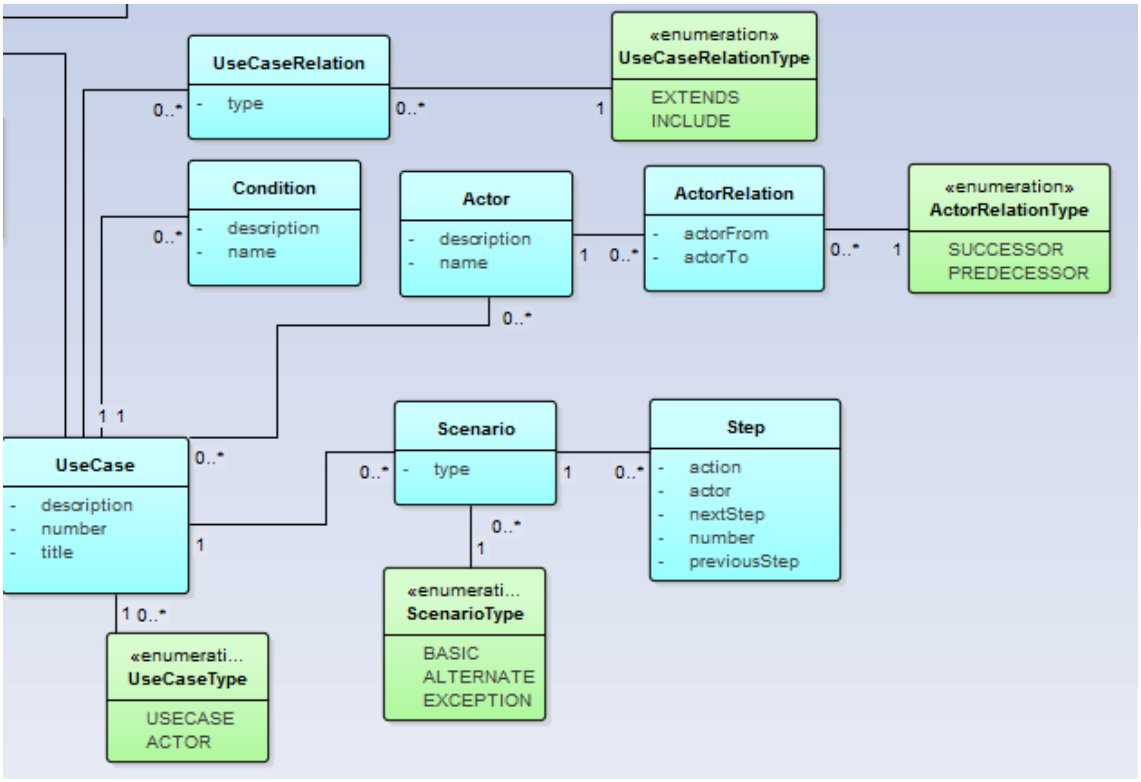
Entita UseCaseType je používána v diagramu případů užití pro zjištění o jaký typ elementu se v diagramu jedná. Zda se jedná o případ užití (UseCase 5.1.4.1), čí aktéra (Actor 5.1.4.3).

5.1.4.3 Actor

Entita Actor reprezentuje aktéra v případech užití. Mezi jednotlivými aktéry mohou existovat vazby dědičnosti pomocí entity ActorRelation 5.1.4.4. Dále pak entita Actor nese informace o tom, v kterých případech užití figuruje.

5.1.4.4 ActorRelation

Entita ActorRelation slouží k identifikaci vztahu mezi aktéry (Actor (5.1.4.3). Vztah je pak dále specifikován pomocí entity ActorRelationType 5.1.4.5, kde je určeno, kdo je ve vztahu předek a kdo následník.



Obrázek 5.4: Návrhový model pro část zabývající se případy užití

5.1.4.5 ActorRelationType

Entita ActorRelationType má za úkol určit, kdo je ve vztahu (ActorRelation 5.1.4.4) předek a kdo následník.

5.1.4.6 Scenario

Entita Scenario představuje scénář případu užití (UseCase 5.1.4.1). Scénář slouží pro podrobný popis případu užití. Scénáře jsou rozděleny na typy podle entity ScenarioType 5.1.4.7. Samotný scénář se pak skládá ze samostatných kroků (Step 5.1.4.8).

5.1.4.7 ScenarioType

Entita ScenarioType je použita k určení typu scénáře pro případ užití (UseCase 5.1.4.1). Rozlišujeme tři typy scénářů. Prvním typem je základní, podle kterého se postupuje, pokud není určené jinak. Dalším typem je alternativní, ten je použit, pokud je v některém z kroků splněna jeho podmínka. Posledním typem je scénář, který nastává, pokud se během případu užití naskytla nějaká chyba.

5.1.4.8 Step

Entita Step reprezentuje jeden krok ve scénáři (Scenario 5.1.4.6) případu užití. Každý krok ve scénáři má své číslo, pomocí kterého je ve scénáři identifikován. Dále pak obsahuje informaci o samotné akci, která má být v průběhu kroku provedena. A obsahuje informace o předchozím a následujícím kroku.

5.1.4.9 Condition

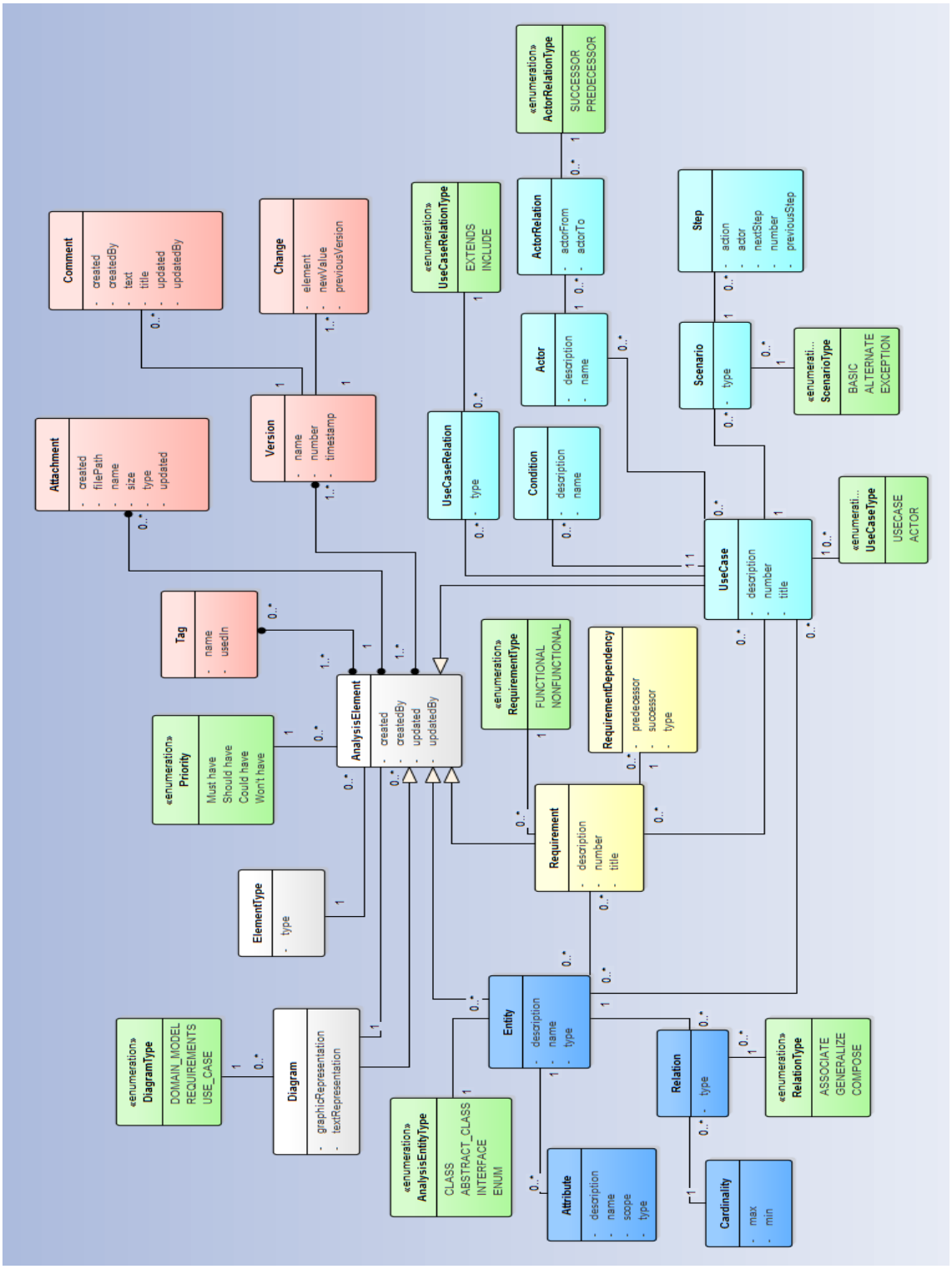
Entita Condition představuje podmínku, která je kladena na případ užití (UseCase (5.1.4.1). U podmínky rozlišujeme, zda se jedná o vstupní čí výstupní podmínku.

5.1.4.10 UseCaseRelation

Entita UseCaseRelation slouží k reprezentaci v vztahů mezi jednotlivými případy užití (UseCase 5.1.4.1). Vztah je pak dále specifikován pomocí entity UseCaseRelationType 5.1.4.11, která určuje o jaký typ vztahu se jedná.

5.1.4.11 UseCaseRelationType

Entita UseCaseRelationType identifikuje typ vztahu mezi případy užití (UseCase 5.1.4.1). Rozlišujeme dva typy vztahů a to *include* a *extends*.



Obrázek 5.5: Návrhový model pro analytický modul

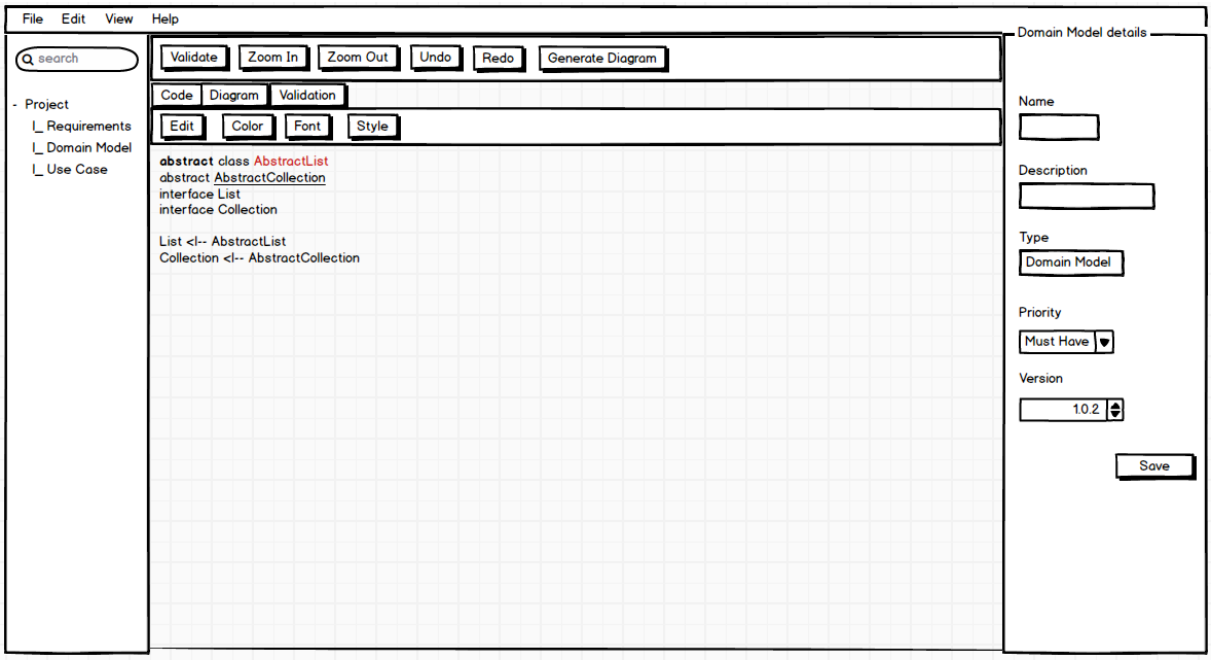
5.2 Grafické rozhraní

Grafické rozhraní bylo navrhováno v souladu s původní aplikací. Nová funkcionalita, tak není odstíněna od původní, ve většině případů právě naopak. Tam, kde dříve bylo možné vytvořit nový graf k projektu, přibyly možnosti vytvoření diagramu požadavků, případů užití a modelu. V hlavní nabídce byla rozšířena možnost Export/Import o další sekce. Stejně tak zůstal zachován i detailní přehled aktuálního výběru v pravé straně obrazovky.

5.2.1 Hlavní plocha

Hlavní plocha doznala v případě zvolení diagramu několik změn. V první řadě je plocha rozdělena na tři taby, mezi kterými je možné libovolně přepínat.

První tab slouží pro psání textové reprezentace diagramu. Pro tyto účely je navíc připraven pod taby panel, pomocí kterého lze formátovat text. V druhém tabu se nachází grafická reprezentace diagramu, za podmínky, že byl diagram úspěšně vygenerován. V posledním tabu se nachází tabulka, která slouží jako přehled závislostí diagramu a jeho částí na ostatních elementech v projektu. Pod taby je pak možné nalézt tabulku obsahující aktuální seznam elementů v diagramu.



Obrázek 5.6: Mockup hlavní obrazovky

5.2.2 Detailní přehled

Do detailního přehledu v případě elementu v diagramu nebo celého diagramu jsou zvoleny atributy *Name*, který představuje název zvoleného prvku. *Description*, kam je možné psát

strukturovaný popis prvku. Atribut Type pouze informuje uživatele u typu zvoleného prvku. Dále pak atribut *Prority*, kde je možné určit prioritu zvoleného prvku. A posledními atributy jsou *Last update*, který představuje datum poslední změny a *Version*, který uvádí aktuální verzi zvoleného prvku.

5.2.3 Ovládací panel

Ovládací panel, který se nachází pod hlavní nabídkou, bude rozšířen o funkce pro generování diagramu a pod původní funkcí validace grafu, bude v případě zvolení diagramu možnost validovat diagram oproti zbývajícím diagramům a určit tak závislosti mezi nimi.

Dále je ovládací panel přepracován, aby nabízel pouze relevantní funkcionalitu, tedy pokud máme aktuálně zvolený graf, nebude viditelná možnost generování diagramu a jiné. Naopak, pokud bude vybrán diagram, nebude možné vidět například tlačítko pro generování testovacích případů.

5.3 Modelovací jazyk

Na základě provedené rešerše existujících jazyků byl zvolen jazyk PlantUML. Bohužel jazyk nepokrývá kompletně funkcionalitu, která je potřebná pro analytický modul. Je tedy nutné jazyk rozšířit o vytváření diagramu požadavků a možnost propojení jednotlivých diagramů.

V návrhu se tak zaměříme na tyto nedostatky a zároveň popíšeme již existující funkcionalitu, která je pro vytvářený analytický modul potřebná. Popis stávající struktury vychází z přehledné dokumentace [39].

5.3.1 Stávající funkcionalita

5.3.1.1 Logická struktura

Pro reprezentaci logické struktury bude použit diagram tříd, který je v PlantUML kompletně implementován, není tak nutné žádné rozšíření. Jazyk umožňuje popis tříd, jejich atributů, operací a vztahů mezi třídami.

Základním prvkem pro popis logické struktury jsou entity. Každá entita má v diagramu své unikátní jméno a je podle něj identifikována. Entity rozlišujeme podle typu, který je zapsán před jménem entity.

Třída	class
Abstraktní třída	abstract abstract class
Rozhraní	interface
Výčet	enum

Tabulka 5.1: Typy entit v diagramu tříd

Obsah entity se uvádí do složených závorek. Obsahem mohou být atributy entity nebo její metody. U atributů jsme schopní definovat datové typy. Datové typy lze zapsat dvěma

způsoby, buď před název atributu nebo za názvem atributu uvedeme dvojtečku a datový typ. Metody a atributy jsou jazykem rozlišovány pomocí vyhledávání znaku závorek. U atributů a metod jsme schopni definovat viditelnost.

private	-
protected	#
package private	~
public	+

Tabulka 5.2: Typy viditelnosti atributů a metod v diagramu tříd

Pro popis vztahů mezi entitami je použit zápis, který vychází z UML značení. Základní verze vztahu má podobu dvou entit, mezi kterými jsou znaky pro vztah. Mezi nepovinné parametry vztahu pak patří kardinalita, kde na každou stranu vztahu se do uvozovek napíše požadovaná hodnota. A pokud by jsme chtěli vztahu přiřadit popisek, je nutné na konci zápisu za cílovou entitu uvést dvojtečku a požadovaný popisek.

Extension	<|--
Composition	*--
Aggregation	o--
a jiné	.. --

Tabulka 5.3: Základní sada typů vztahů v diagramu tříd, další je možné odvozovat

Listing 5.1: Příklad zápisu diagramu tříd

```
abstract class Foo
interface Demo

class Dummy {
        #field1
        +method1()
}

enum TimeUnit {
        DAYS
        HOURS
}

Foo <|-- Demo

Foo "1" *-- "many" Dummy : contains
```

5.3.1.2 Případy užití

Jazyk PlantUML podporuje zápis diagramů případů užití, bohužel však nijak neřeší zápis scénářů. Jazyk umožňuje vytváření případů užití a k těm pak vázat jednotlivé aktéry.

Pro zápis případu užití se využívají kulaté závorky tak, aby byla opět dodržena podoba s UML značením. Druhou možností zápisu je místo závorek napsat před název případu užití výraz *usecase*. Případ užití lze identifikovat přímo jeho názvem a nebo je možné mu přiřadit identifikátor a to pomocí výrazu *as* a samotný identifikátor.

Pro zápis aktéra je nutné jeho název vložit mezi dvě dvojtečky nebo použitím speciálního výrazu *actor*. I zde je možné vytvoření identifikátoru stejně jako v předchozím případě.

Aktér	Aktér	<|--
Případ užití	Aktér	<-- --
Případ užití	Případ užití	<.

Tabulka 5.4: Základní sada typů vztahů pro diagram případů užití, další je možné odvozovat

Pro popis vztahů mezi aktéry a případy užití je použit zápis, který vychází stejně jako v diagramu tříd z UML značení. Základní podoba vztahu má tak podobu dvou entit, mezi kterými jsou znaky pro vztah.

Listing 5.2: Příklad zápisu případu užití

```
actor customer

actor clerk

usecase help as uc1

customer -- (checkout)

(checkout) .> (payment) : include

uc1 .> (checkout) : extends

(checkout) -- clerk
```

5.3.2 Přidaná funkcionalita

5.3.2.1 Diagram požadavků

Jazyk PlantUML a ani jiný z testovaných jazyků nenabízí možnost zápisu požadavků na software. Je proto nutné rozšířit existující sadu jazyků PlantUML o jazyk, který bude popisovat právě požadavky.

Při snaze vytvořit co nejjednodušší zápis požadavku, jsme dospěli k názoru, že jsou potřebné tři informace. První je typ požadavku, u každého budeme rozlišovat, zda se jedná o požadavek funkční či nefunkční. Druhým nutným atributem je unikátní označení požadavku, například číslo nebo řetězec znaků. A posledním atributem je samotný název požadavku, který popisuje o čem požadavek pojednává, jedná se však pouze o stručný popis. Podrobnější popis požadavku necháme dostupný mimo strukturu jazyka. Každý z popsaných atributů požadavku je oddělený znakem dvojtečky.

Funkční požadavek	functional
Nefunkční požadavek	nonfunctional

Tabulka 5.5: Typy požadavků

Listing 5.3: Příklad zápisu diagramu požadavků

```
functional : FRQ1 : requirement1
functional : FRQ2 : requirement2

nonfunctional : NFQ1 : requirement3
nonfunctional : NFQ2 : requirement4
```

5.3.2.2 Propojení pohledů

Propojení pohledů je velmi podstatnou vlastností UML, bohužel žádný z testovaných jazyků tuto vlastnost neobsahuje. Je proto nutné rozšířit každý ze sady použitých jazyků PlantUML o možnost vztahu k jiným diagramům. Tento požadavek je rozšířen do takové míry, že je možné z libovolného diagramu v projektu odkazovat na jiné diagramy nebo dokonce jejich vnitřní elementy.

Pro zápis vztahu mezi dvěma entitami je nutné uvést zdrojovou a cílovou entitu a mezi ně vložit výraz pro vztah reprezentující propojení pohledů. Výraz pro tento vztah musí být unikátní napříč všemi využitými diagramy, aby nedocházelo ke konfliktům.

Listing 5.4: Diagram požadavků

```
functional : FRQ1 : requirement1
functional : FRQ2 : requirement2
```

Listing 5.5: Případ užití se vztahem k požadavku ve vedlejším diagramu

```
actor customer
customer -- (checkout)

(checkout) >>> requirement1
```

5.4 Použité technologie

Výběr technologií byl proveden na základě stávajícího stavu aplikace. Celá aplikace je vytvořena v jazyce Java. Pro grafickou podobu aplikace byl zachována knihovna Swing. Všechny nově zvolené technologie byly vybrány na základě předchozích zkušeností.

5.4.1 Maven

Apache Maven [4] je nástroj pro správu, řízení a automatizaci buildů aplikací. Základním principem fungování Mavenu je popsání projektu pomocí Project Object Model. Tento model popisuje softwarový projekt nejen z pohledu jeho zdrojového kódu, ale včetně závislostí na externích knihovnách [35]. V aplikaci je použit k jednoduchému připojení analytického modulu k původnímu systému.

5.4.2 H2

H2 [2] je relační databáze napsaná v jazyce Java. Nabízí možnost pracovat v ruzných režimech, jedním z nich je *embedded*. Hlavní výhodou toho režimu je, že běží v adresovém prostoru aplikace, která je využívá a není tak nutná existence připojení typu klient-server. Databáze byla použita díky své jednoduchosti, ale v případě nutnosti není problém ji nahradit jinou implementací. V aplikací je použit k uchování dat, která využívá analytický modul. Do budoucna existuje možnost propojení i s původní aplikací.

5.4.3 Hibernate

Hibernate [3] je framework napsaný v jazyce Java, který umožňuje tzv. objektově-relační mapování (ORM) [37]. Usnadňuje řešení otázky zachování dat objektů i po ukončení běhu aplikace. Je jednou z implementací Java Persistence API (JPA)[33]. V aplikací slouží k připojení na relační databázi.

5.4.4 Spring

Spring [5] je komplexní technologie pro vývoj aplikací bez rozdílu zaměření. A slouží jako zprostředkovatel mezi aplikací a výše zmíněnými technologiemi. Jeho hlavní předností je odstranění těsných programových vazeb jednotlivých objektů a vrstev pomocí návrhového vzoru Inversion of Control [34]. V projektu je použit modul *CORE* na propojení jednotlivých vrstev aplikace.

5.4.5 GraphViz

Graphviz [7] je Software pro znázorňování grafů. Jako balík svobodného software je použit pro kreslení grafů zadaných ve formátu DOT. Graf se zapíše pomocí speciálního jazyka a zvolený nástroj vhodně rozmístí jeho uzly, hrany, a vykreslí jej. GraphViz jako takový poskytuje pouze program pro příkazovou řádku, pomocí kterého lze grafy generovat do různých grafických formátů nebo do textového formátu reprezentujícího informace o layoutu [15]. V aplikaci je použit pro transformaci textové reprezentace diagramu na reprezentaci grafickou.

Kapitola 6

Implementace

Implementace byla rozdělena do dvou hlavních částí. První část se zabývá vypracováním samotného analytického modulu a druhá část řeší integraci se stávajícím systémem.

První část se pak skládá z vytvoření perzistentní vrstvy a vrstvy servisní, přes kterou probíhá veškerá interakce se stávajícím systémem. Důležitou součástí servisní vrstvy jsou konvertory, které slouží k překladu námi vytvořených diagramů na diagramy PlantUML a zpět. Posledním bodem jsou úpravy jazyka PlantUML o podporu námi navržených změn.

Druhá část je složena ze samotného propojení modulu k původní aplikaci. Vytvoření grafického rozhraní pro novou funkcionalitu a úpravu presenterů. Realizování funkcí import a export pro diagramy a úprava stávající struktury pro podporu diagramů v projektu.

6.1 Perzistentní vrstva

Perzistentní vrstva je jediné místo aplikace, které má možností komunikace s databází. Je tvořena třídami reprezentujícími entity v rámci návrhového modelu a třídami DAO [29]. Entity jsou pomocí objektově relačního mapování propojeny s tabulkami v databázi. Logika pro přístup do relační databáze je implementována pomocí návrhového vzoru Data Access Object(DAO objekty). Jedná se převážně o operace typu CRUD (Create, Read, Update a Delete).

Objektově relační mapování a práce s databází je v režii frameworku Hibernate, který je konfigurován pomocí aplikačního kontextu frameworku Spring. Výhodou této technologie je nezávislost na použité databázi.

6.2 Servisní vrstva

Balík service obsahuje rozhraní pro komunikaci s původní aplikací a slouží jako odstínění od práce s databází. Veškerý obsah balíku je registrovaný ve frameworku Spring a je tak možné k němu přistupovat pomocí dependency injection [30]. Součástí servisní vrstvy jsou konvertory použité k mapování diagramů z datového modelu na diagramy jazyka PlantUML a zpět.

6.3 Modelovací jazyk

Jazyk PlantUML není možné v projektu používat pouze jako externí knihovnu, z důvodu nutnosti implementace změn, které jsou navrženy v předchozí kapitole. Zdrojové kódy jsou umístěny v samostatném balíku, kam jsou implementovány navržené změny.

PlantUML nabízí možnost generování grafické podoby diagramu ve standardu UML přímo z textové reprezentace. Je k tomu využita externí aplikace GraphViz.

Jazyk PlantUML je definován příkazy za použití návrhového vzoru Command [28]. Příkazy je možné dělit na jednořádkové a víceřádkové. První, co se při zpracování zdrojového kódu provede, je jeho naparsování na tyto příkazy, s kterými se pak pracuje.

Pro nás podstatná struktura PlantUML se skládá převážně z návrhového vzoru Factory [31]. Každý typ diagramu má vlastní továrnu, která rozhoduje, zda je z obdrženého kódu schopna sestavit příslušný diagram. Každá továrna má přiřazenou sadu příkazů, které je možné v kódu použít. Typ diagramu je určen, pokud je továrna schopna zpracovat všechny příkazy ze zdrojového kódu, taková továrna je vždy maximálně jedna.

Vnitřní struktura příkazů je definována pomocí regulárních výrazů. Pro vytvoření regulárních výrazu je použita knihovna Regex, která je v PlantUML rozšířena o několik výrazů(např. Leaf, Or, Concat nebo Optional).

6.3.1 Diagram požadavků

Pro zápis požadavků je vytvořen nový jazyk, který je vložen do sady jazyků PlantUML. Nový jazyk reprezentuje textovou reprezentaci diagramu požadavků. V diagramu požadavků jsou popsány business požadavky na software. Každý požadavek je rozdělen podle typu(funkční, nefunkční) a je možné ho jednotně identifikovat pomocí alfanumerického klíče. Každý požadavek také obsahuje popisek, který vyjadřuje podmínku nebo funkci, kterou musí systém splňovat.

Diagram požadavků byl vytvořen na základě ostatních diagramů jazyka PlantUML. Diagram obsahuje pouze jeden příkaz, který reprezentuje zápis požadavku. Příkaz se skládá ze tří povinných parametrů. První z nich slouží pro určení typu požadavku a je nutné mu přiřadit jednu z hodnot *functional* nebo *nonfunctional*. Druhý parametr obsahuje alfanumerický identifikátor, pomocí kterého je možné odkazovat na jiné diagramy a elementy. Tato hodnota musí být unikátní napříč všemi diagramy. Poslední atribut slouží ke stručnému popisu požadavku, který musí systém splňovat.

Listing 6.1: Regární výraz pro příkaz Vytvoření požadavku

```
new RegexConcat(
// Prikaz musi zacinat na novem radku
new RegexLeaf("^"),

// Nasleduje libovolny pocet mezer
new RegexLeaf("[%s]*"),

// definice typu pozadavku
new RegexLeaf("TYPE", "(functional|nonfunctional)+"),

// Nasleduje libovolny pocet mezer
new RegexLeaf("[%s]*"),

// Oddeleni pomoci znaku dvojtecky
new RegexLeaf([:]+"),

// Nasleduje libovolny pocet mezer
new RegexLeaf("[%s]*"),

// identifikator pozadavku
new RegexLeaf("ID", "([\\p{L}0-9_.]+)"),

// Nasleduje libovolny pocet mezer
new RegexLeaf("[%s]*"),

// Oddeleni pomoci znaku dvojtecky
new RegexLeaf([:]+"),

// definice ciloveho elementu
new RegexLeaf("CODE", "([\\p{L}0-9_.]+)"),

// Oddeleni pomoci znaku dvojtecky
new RegexLeaf([:]+"),

// Nasleduje libovolny pocet mezer
new RegexLeaf("[%s]*"),

// Prikaz je ukoncen koncem radku
new RegexLeaf("$")
);
```

6.3.2 Příkaz Dependency

Pro definování vztahů napříč diagramy byl navržen příkaz Dependency, který byl vložen do všech využívaných typů diagramů. Příkaz má strukturu, která byla navržena v předchozí kapitole. Skládá se ze dvou elementů(zdrojového a cílového) a výrazu pro vztah. Příkaz nemá žádné grafické vyjádření v diagramech UML.

Listing 6.2: Regární výraz pro příkaz Dependency

```
new RegexConcat(
        // Prikaz musi zacinat na novem radku
        new RegexLeaf("^"),

        // Nasleduje libovolny pocet mezer
        new RegexLeaf("[%s]*"),

        // definice zdrojoveho elementu
        new RegexLeaf("CODE1", "([\\p{L}0-9_.]+)"),

        // Nasleduje libovolny pocet mezer
        new RegexLeaf("[%s]*"),

        // Vyraz >>> pro vztah mezi elementy
        new RegexLeaf("ARROW", "(\\>\\>\\>)+"),

        // Nasleduje libovolny pocet mezer
        new RegexLeaf("[%s]*"),

        // definice ciloveho elementu
        new RegexLeaf("CODE2", "([\\p{L}0-9_.]+)"),

        // Nasleduje libovolny pocet mezer
        new RegexLeaf("[%s]*"),

        // Prikaz je ukoncen koncem radku
        new RegexLeaf("$")
);
```

6.3.3 Použití PlantUML

PlantUML nabízí snadný způsob integrace, veškerá komunikace probíhá pomocí instance třídy SourceStringReader nebo SourceFileReader, podle toho v jaké podobě vlastníme zdrojový kód(řetězec znaků nebo soubor). V našem případě je využita třída SourceStringReader, která jako zdroj dat používá řetězec znaků. K vytvoření instance slouží řada konstruktorů, které se liší mírou konfigurovatelnosti. Jediným povinným parametrem je zdrojový kód reprezentovaný datovým typem String. Zpracování kódu je vázané přímo na konstruktor.

Listing 6.3: Vytvoření jednoduchého diagramu v PlantUML

```
String source = "@startuml\n";

source += "class Foo\n";
source += "class Dummy\n";
source += "Foo <|-- Dummy\n";

source += "@enduml\n";
```

Pro vygenerování grafické reprezentace diagramu slouží metoda *generateImage* s parametrem typu OutputStream, do kterého je uložen obrázek ve formátu PNG. Pro zvolení jiného výstupního formátu je možné využit nepovinný parametr typu FileFormatOption.

Listing 6.4: Vygenerování grafické reprezentace UML

```
File file = ...;
OutputStream pngImage = new FileOutputStream(file);

SourceStringReader reader = new SourceStringReader(source);
reader.generateImage(pngImage);
```

Jazyk PlantUML umožňuje vytvoření více diagramů v jednom zdrojovém kódu. Tyto diagramy jsou ve vnitřní struktuře označeny jako bloky. V našem případě však nikdy nenastane možnost, že by vzniklo více diagramů. Pro získání zdrojového PlantUML diagramu je nutné nejdříve získat list vytvořených bloků, kde víme, že existuje maximálně jeden.

Zdrojový diagram je poté vložen do připraveného konvertoru a za pomocí metody *generateDiagramStructure* s parametrem cílového diagramu, je provedena konverze.

Listing 6.5: Získání PlantUML diagramu a konverze na diagram z datového modelu

```
SourceStringReader reader = new SourceStringReader(source);
List<BlockUml> blocks = reader.getBlocks();

if(!CollectionUtils.isEmpty(blocks)){
        BlockUml blockUml = blocks.get(0);
        Diagram diagram = blockUml.getDiagram();

        converter.setSourceDiagram(diagram);
        converter.generateDiagramStructure(new Diagram());
}
```

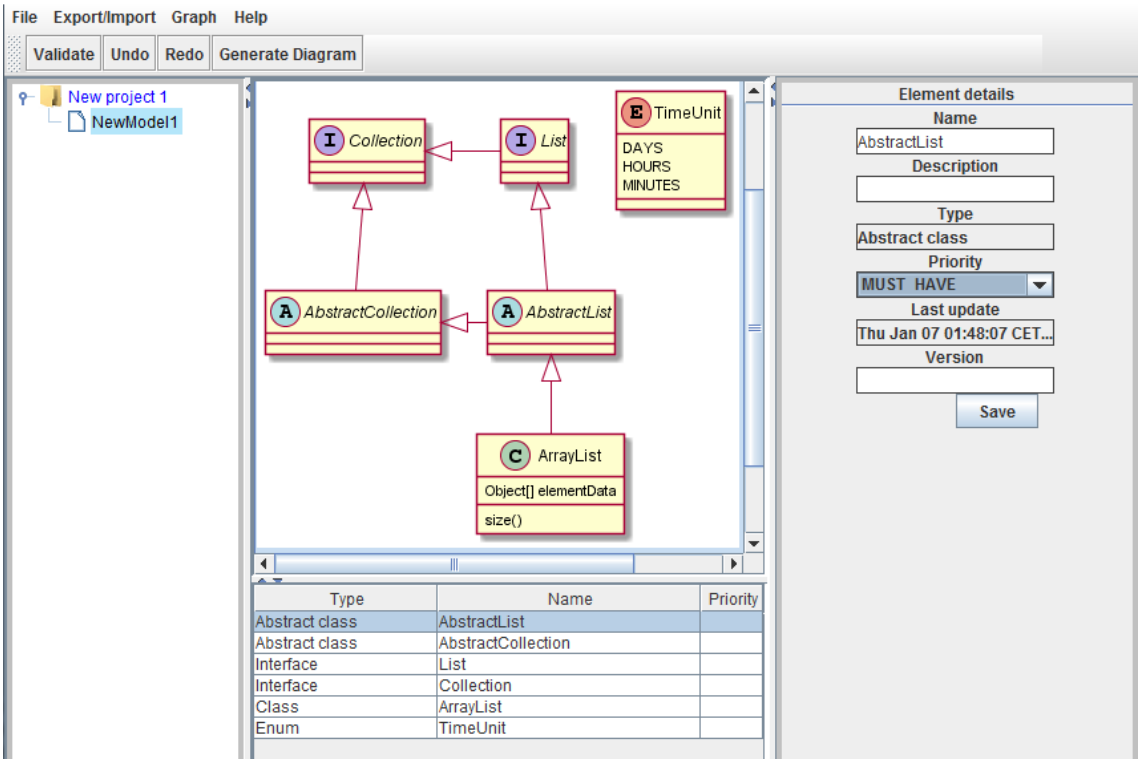
6.4 Integrace s PCTGen

Propojení původní aplikace a analytického modulu bylo provedeno pomocí nástroje Apache Maven. Byl vytvořen nadřazený projekt s názvem PCTGen-parent, který neobsahuje žádnou logiku a slouží pouze k propojení původní aplikace a modulu. Viditelnost obsahu analytického modulu je zajištěna závislostí v konfiguračním souboru pom.xml v původním projektu. Komunikace mezi moduly pak probíhá pomocí servisní vrstvy analytického modulu. Původní aplikace byla upravena tak, aby bylo možné použití dependency injection pomocí Spring frameworku.

6.5 Úpravy původní aplikace

Původní aplikace je navržena podle architektury Model-View-Presenter [14]. Systém obsahuje několik view (např. MainGui). Každé view si udržuje vazbu na patřičný presenter (např. MainGuiPresenter),který obsluhuje uživatelské vstupy.

V řadě případů aplikace nepočítala s možností rozšíření a bylo nutné ji refaktorovat. Zejména komunikace mezi komponentami probíhala pomocí konkrétní implementace a ne rozhraní. Rozhraní však již byla připravena a nebylo tak nutné je vytvářet.



Obrázek 6.1: Obrazovka s funkcionalitou analytického modulu

Dále bylo nutné rozšířit rozhraní zodpovědné za import a export do formátu XML. Bylo nutné navrhnout a implementovat strukturu pro možnost zápisu a opětovného načtení projektu, který obsahuje diagramy z analytického modulu. Diagramy jsou ukládány pod tagy, které přímo reprezentují typ diagramu, například diagram požadavků používá tag *requirements*. Pro zachování atributů diagramu jsou použity atributy u tohoto tagu. Pro uchování struktury textové reprezentace je nutné zachovat rozdělení řádků textu, protože jazyk PlantUML je na znacích signalizujících ukončení řádku silně závislý. Každý řádek textové reprezentace je uložen v samostatném elementu, který má tag *line*.

Kapitola 7

Testování

Testování je podstatnou součástí vývoje software. Každý správný softwarový projekt by měl být podroben důkladnému testování. Testování slouží k odhalení chyb, které mohly vzniknout v průběhu projektu a ověření, zda funkcionalita odpovídá finální specifikaci. Testování by mělo probíhat po celou dobu vývoje a udržovat tak dostatečnou kvalitu produktu.

Testování je možné dělit do různých kategorií, typů a fází, jak je možné vidět například u obrázku V-modelu 2.3. Pro otestování aplikace byly v průběhu vývoje použity jednotkové testy a testy integrační. Testy byly tvořeny a upravovány společně s implementací. Pro ověření, že aplikace jako celek funguje korektně a že správně plní úlohu, pro kterou byla vyvinuta, že vrací správné výstupy a v neposlední řadě, že byly pokryty všechny požadavky ze strany zákazníka byly použity manuální systémové testy [6].

7.1 Jazyk Groovy

Pro tvorbu jednotkových a integračních testů byl použit jazyk Groovy [32]. Jazyk byl použit pro svoji jednoduchost a stručnost oproti jazyku Java. Groovy využívá výhody objektového programování, ale zároveň poskytuje zjednodušenou syntaxi, která umí zabalit a rozbalit často opakované části kódu. Zbavíme se tak zbytečné duplikace kódu, který by nám přinesla Java. Navíc je možné jej použít všude tam, kde funguje Java, nepotřebuje žádnou další konfiguraci. Jediné nastavení, které je nutné provést, je podpora Groovy v nástroji Maven, aby se testy spouštěly automaticky při startu buildovacího procesu. Toho je dosaženo přidáním pluginu v konfiguračním souboru *pom.xml* . Nastavení tohoto pluginu je možné vidět na obrázku 7.1.

Listing 7.1: Nastavení Apache Maven pro podporu jazyka Groovy v testech

```
<plugin>
  <groupId>org.codehaus.gmaven</groupId>
  <artifactId>gmaven-plugin</artifactId>
  <version>1.4</version>
  <configuration>
    <providerSelection>2.0</providerSelection>
    <source/>
  </configuration>
    <executions>
      <execution>
        <id>testCompile</id>
        <goals>
          <goal>testCompile</goal>
        </goals>
        <configuration>
          <sources>
            <fileset>
              <directory>${pom.basedir}/src/test/java</directory>
              <includes>
                <include>**/*.groovy</include>
              </includes>
            </fileset>
          </sources>
        </configuration>
      </execution>
  </executions>
</plugin>
```

7.2 Jednotkové testy

Jednotkové neboli unit testy se zaměřují na nejmenší testovatelné části aplikace. Obvykle se jedná o komponenty na úrovni objektů a tříd. Slouží pro ověření, že vytvořená část kódu funguje a její funkce odpovídá očekávání. Při jednotkovém testování jsou použity tzv. *mock* objekty pro náhradu funkčních vrstev, na kterých je testovaná část závislá. Dosáhne se tak izolace od částí aplikace, které nejsou součástí testované jednotky.

Pro tento typ testování byl použit framework JUnit ve verzu 4.1 a samotné testy jsou napsány v jazyce Groovy. Pro schopnost vytváření mock objektů je použit framework EasyMock ve verzi 3.0. Při vytváření testů byla dodržena standardní jmenná konvence, tedy všechny testovací třídy mají název složeny z testované třídy a postfixu *Test*.

Jednotkové testy byly požity převážně pro ověření správné funkčnosti konvertorů, které slouží pro překlad diagramů z jazyka PlantUML na diagramy z našeho datového modelu.

7.3 Integrační testy

Integrační testy se zaměřují na korektní komunikaci jednotlivých vrstev. Testuje se vzájemná interakce mezi operacemi napříč moduly. Právě integrace je z pohledu vývoje aplikací poměrně kritickou oblastí. Z toho důvodu mají integrační testy svoje nezastupitelné místo.

Stejně jako v případě jednotkových testů i zde byl použit framework JUnit, navíc je zda však použit modul *spring-test* frameworku Spring, který slouží pro nastavení testovacího prostředí spolu s testovací databází.

V našem případě bylo integračními testy otestována komunikace servisní a perzistentní vrstvy spolu s databází. Integrační testy v tomto případě z části nahrazují testy jednotkové. Od servisní vrstvy níže je aplikace pokryta více než 50 integračními testy.

7.4 Pokrytí testy

Nový analytický modul, který je integrován v původní aplikaci, je pokryt sadou jednotkových a integračních testů. Celkový počet testů se pohybuje kolem 60. Celá aplikace pak prošla řadou systémových testů, které měly za úkol odhalit nesoulad aplikace se softwarovými požadavky. V průběhu testování byla odhalena řada chyb, které se podařilo díky testům opravit.

Cílem testování je nalézt maximální možný počet softwarových chyb, ne však dosáhnout bezchybnosti testovaného systému. Testováním totiž není možno zajistit naprostou bezchybnost, protože nejsme schopni vyzkoušet všechny možné kombinace vstupních hodnot.

	Požadavek	**Priorita**	**Splněno**
FRQ-1	Editor pro textovou reprezentaci	M	✓
FRQ-2	Vztahy mezi diagramy	M	✓
FRQ-3	Integrace s PCTGen	M	✓
FRQ-4	Export a Import	S	✓
FRQ-5	Tagy	C	✗
FRQ-6	Komentáře	C	✗

Tabulka 7.1: Výstup testování společné funkcionality modulu

	Požadavek	**Priorita**	**Splněno**
FRQ-7	Vytváření požadavků	M	✓
FRQ-8	Úprava požadavků	M	✓
FRQ-9	Mazání požadavků	M	✓
FRQ-10	Grafická reprezentace požadavků	M	✓
FRQ-11	Validace požadavků	S	✓
FRQ-12	Verzování požadavků	S	✓
FRQ-13	Přílohy k požadavkům	C	✗
FRQ-14	Filtrování nad požadavky	C	✗
FRQ-15	Diff verzí požadavků	W	✗

Tabulka 7.2: Výstup testování požadavků na požadavky software

	Požadavek	**Priorita**	**Splněno**
FRQ-16	Vytváření entit	M	✓
FRQ-17	Úprava entit	M	✓
FRQ-18	Mazání entit	M	✓
FRQ-19	Vztahy mezi entitami	M	✓
FRQ-20	Atributy u entity	M	✓
FRQ-21	Validace analytického modelu	S	✓
FRQ-22	Verzování analytického modelu	S	✓
FRQ-23	Přílohy k analytickému modelu	S	✗

Tabulka 7.3: Výstup testování požadavků na anylytický model

	Požadavek	**Priorita**	**Splněno**
FRQ-24	Vytváření případů užití	M	✓
FRQ-25	Úprava případů užití	M	✓
FRQ-26	Mazání případů užití	M	✓
FRQ-27	Závislosti mezi případy užití	M	✓
FRQ-28	Aktéři u případu užití	M	✓
FRQ-29	Scénáře k případu užití	S	✗
FRQ-30	Validace případů užití	S	✓
FRQ-31	Verzování případů užití	S	✓
FRQ-32	Přílohy k případům užití	S	✗

Tabulka 7.4: Výstup testování požadavků na případy užití

Kapitola 8

Závěr

Cílem práce bylo vytvořit modul do nástroje PCTGen, který rozšíří stávající funkcionalitu o podporu analýzy software. Součástí modulu je práce s textovým modelovacím jazykem, proto byla provedena rešerše existujících jazyků určených k modelování softwarových systémů. Jazyky byly vyhodnocovány dle námi navrhnutých kriterií. Na základě výsledků získaných z rešerše, jsme dospěli k závěru, že žádný z testovaných jazyků plně nesplňuje daná kriteria. Zvolen byl jazyk PlantUML, který vyžaduje nejmenší počet zásahů a zároveň splňuje ostatní kriteria. Jazyk byl rozšířen o diagram požadavků a možnost vytvoření vazeb mezi odlišnými diagramy.

Rozšiřující modul byl zanalyzován, navrhnut, implementován a otestován. Veškeré požadavky s prioritou *MUST HAVE* se podařilo implementovat. Požadavky s nižší prioritou se však bohužel nepovedlo kompletně splnit a jsou tak přesunuty do další etapy vývoje. Modul se podařilo úspěšně integrovat do původní aplikace PCTGen, za použití nástroje Maven. Komunikace mezi modulem a aplikací probíhá přes rozhraní servisní vrstvy. Systém je postaven na technologiích fungujících na principu vícevrstvé architektury a díky vytvoření rozhraní na každé vrstvě, je možné libovolnou vrstvu nahradit.

Původní aplikace byla rozšířena o možnosti analytického modulu, za tímto účelem byly vytvořeny nové části obrazovek a rozšířena původní rozhraní. Jedná se zejména o části obrazovek umožňující vytvoření textové a grafické reprezentace diagramů nebo tabulky zobrazující vztahy mezi diagramy. Aplikace byla také rozšířena o řadu funkcí podporujících funkcionalitu analytického modulu, například import a export diagramů. Původní aplikace byla refaktorována a je tak dosaženo snadnější možnosti rozšiřitelnosti.

Hlavním přínosem pro mě bylo seznámení s textovými modelovacími jazyky, které je možné použít jako náhradu grafického standardu UML. A také rozšíření jazyka PlantUML a komunikace s open-source komunitou, který stojí za vývojem jazyka, byla velmi přínosnou zkušeností.

Literatura

[1] . *V-model* [online]. 2015. [cit. 23. 12. 2015]. Dostupné z: <http://www.bugtracker.cz>.

[2] *H2 Database Engine* [online]. [cit. 8. 1. 2016]. Dostupné z: <http://www.h2database.com/html/main.html>.

[3] *Hibernate. Everything data.* [online]. [cit. 8. 1. 2016]. Dostupné z: <http://hibernate.org/>.

[4] *Apache Maven Project* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://maven.apache.org/>.

[5] *Spring* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://spring.io/>.

[6] . *Druhy testování v procesu vývoje SW* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://www.swtestovani.cz/index.php/uvod-do-testovani/18-druhy-testovani-v-procesu-vyvoje-sw>.

[7] AT&T Labs Research. *Graphviz* [online]. 2012. [cit. 8. 1. 2016].

[8] BADREDDIN, O. Umple: A Model-oriented Programming Language. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, s. 337–338, New York, NY, USA, 2010. ACM. doi: 10.1145/1810295.1810381. Dostupné z: <http://doi.acm.org/10.1145/1810295.1810381>. ISBN 978-1-60558-719-6.

[9] BURES, M. PCTgen: Automated Generation of Test Cases for Application Workflows. In ROCHA, A. et al. (Ed.) *New Contributions in Information Systems and Technologies*, 353 / *Advances in Intelligent Systems and Computing*. Springer International Publishing, 2015. s. 789–794. doi: 10.1007/978-3-319-16486-1_78. Dostupné z: <http://dx.doi.org/10.1007/978-3-319-16486-1_78>. ISBN 978-3-319-16485-4.

[10] BURES, M. *PCTgen - Process Cycle Test cases generator* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://webing.felk.cvut.cz/bures/pctgen/>.

[11] CHAVES, R. *TextUML repository* [online]. 2011. [cit. 8. 1. 2016]. Dostupné z: <https://github.com/abstratt/textuml>.

[12] CHAVES, R. *TextUML Toolkit* [online]. 2011. [cit. 8. 1. 2016]. Dostupné z: <http://abstratt.github.io/textuml/>.

[13] FORWARD, A. et al. Model-driven rapid prototyping with Umple. *Software: Practice and Experience*. 2011.

[14] FOWLER, M. *GUI Architectures* [online]. [cit. 8. 1. 2016]. Dostupné z: <http://martinfowler.com/eaaDev/uiArchs.html>.

[15] HORDEJCUK, V. *Graphviz* [online]. [cit. 8. 1. 2016].

[16] KOLOVOS, D. et al. Requirements for Domain-specific Languages. *Proc. of ECOOP Workshop on Domain-Specific Program Development*. 2006.

[17] KOOMEN, T. a kol. *Process Cycle Test* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://www.tmap.net/wiki/process-cycle-test-pct>.

[18] KOOMEN, T. a kol. *TMap Next* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://www.tmap.net/en/tmap-next>.

[19] KRUNTCHEN, P. *The 4+1 View Model of Architecture*. IEEE Software., 1995. ISBN 0740-7459.

[20] LETHBRIDGE, T. C. *UmpleOnline: Generate Java, PHP, or Ruby code from Umple* [online]. 2012. [cit. 8. 1. 2016].

[21] LOWINGER, L. Aplikace pro generování testovacích situací pro techniku Process Cycle Test. *Bakalářská práce ČVUT FEL*. 2014.

[22] MAZANEC, M. Jazyky pro textové modelování. *Bakalářská práce ČVUT FEL*. 2012.

[23] MAZANEC, M. – MACEK, O.

[24] Ovidiu Gheorghies. *MetaUML repository* [online]. 2015. [cit. 8. 1. 2016]. Dostupné z: <https://github.com/ogheorghies/MetaUML>.

[25] Ovidiu Gheorghies. *An Introduction to MetaUML* [online]. 2005. [cit. 8. 1. 2016]. Dostupné z: <http://metauml.sourceforge.net/>.

[26] Ovidiu Gheorghies. *MetaUML Live!* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://metauml.denksoft.com/>.

[27] Přispěvatelé Wikipedie. *CC-BY-SA* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://cs.wikipedia.org/wiki/CC-BY-SA>.

[28] Přispěvatelé Wikipedie. *Command pattern* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Command_pattern>.

[29] Přispěvatelé Wikipedie. *Data access object* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Data_access_object>.

[30] Přispěvatelé Wikipedie. *Dependency injection* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Dependency_injection>.

[31] Přispěvatelé Wikipedie. *Factory (object-oriented programming)* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)>.

[32] Přispěvatelé Wikipedie. *Groovy (programming language)* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Groovy_(programming_language)>.

[33] Přispěvatelé Wikipedie. *Hibernate* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://cs.wikipedia.org/wiki/Hibernate>.

[34] Přispěvatelé Wikipedie. *Inversion of control* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Inversion_of_control>.

[35] Přispěvatelé Wikipedie. *Apache Maven* [online]. [cit. 8. 1. 2016]. Dostupné z: <https://cs.wikipedia.org/wiki/Apache_Maven>.

[36] Přispěvatelé Wikipedie. *MoSCoW method* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/MoSCoW_method>.

[37] Přispěvatelé Wikipedie. *Object-relational mapping* [online]. 2013. [cit. 10. 5. 2013]. Dostupné z: <http://en.wikipedia.org/wiki/Object-relational_mapping>.

[38] Přispěvatelé Wikipedie. *Website wireframe* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <https://en.wikipedia.org/wiki/Website_wireframe>.

[39] ROQUES, A. Drawing UML with PlantUML. *PlantUML Language Reference Guide*. 2016.

[40] ROQUES, A. *PlantUML* [online]. 2011. [cit. 8. 1. 2016]. Dostupné z: <http://plantuml.com/>.

[41] SLUNECKO, Z. *Testování podnikových aplikací* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://www.systemonline.cz/sprava-it/testovani-podnikovych-aplikaci.htm>.

[42] SPINELLIS, D. *UMLGraph: Declarative Drawing of UML Diagrams* [online]. 2012. [cit. 8. 1. 2016].

[43] TOBBIN, H. *Create UML diagrams online in seconds, no special tools needed* [online]. 2016. [cit. 8. 1. 2016]. Dostupné z: <http://yuml.me/>.

Příloha A

Seznam použitých zkratek

TML Textual Modeling Language

UML Unified Modeling Language

JSON JavaScript Object Notation

CSV Comma-separated values

MIT Massachusetts Institute of Technology

GPL General Public License

EPL Eclipse Public License

ORM Object Relational Mapping

PNG Portable Network Graphics

SE Java Standard Edition

JPA Java Persistance API

JDBC Java Database Connectivity

POJO Plain Old Java Object

XML Extensible Markup Language

DAO Data Access Object

CRUD Create, Read, Update, Delete

SQL Structured Query Language

⋮

Příloha B

Instalační a uživatelská příručka

Zde je popsán postup, jak správně aplikaci nainstalovat a spustit.

B.1 Potřebné programy

- Java JRE a JDK (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>)
- IntelliJ Idea 14.0 a vyšší (<https://www.jetbrains.com/idea/>)
- GraphViz Release 2.38 a vyšší (<http://www.graphviz.org/>)

B.2 Instalace a nastavení

B.2.1 GraphViz

1. Stáhnout a instalovat podle pokynů instalátoru.
2. Pokud není nainstalován do default adresáře
   (a) Windows
      i. Přejit na: My Computer/Properties/Advanced/Environment Variables
      ii. Vytvořit proměnou GRAPHVIZ_DOT a hodnota je cesta k adresáři (např. `d:\example\dot.exe`)
   (b) Linux/Mac OS-X
      i. setenv GRAPHVIZ_DOT `/usr/local/bin/graphviz/dot`
      ii. export GRAPHVIZ_DOT

B.3 Spuštění

1. Otevřít projekt PCTGen-parent ve vývojovém prostředí jako Maven projekt přes `pom.xml`
2. Na projektu `PCTGen-parent` provést Maven příkaz `mvn clean install`
3. Spustit projekt PCTGen pomocí přes třídu `ProcessCycleTestEditor`

Příloha C

Obsah přiloženého CD

```
|---AbstraktCZ
|       abstrCZ.txt
|
+---AbstraktEN
|       abstrEN.txt
|
+---EA
|   |
|      +---PCTGen-Analysis Modul.eap
|
|      +---reverse-engineering class_Diagram.eap
|
+---k336_thesis
|
+---src
|   \---PCTGen-parent
|    |
|            +---analysisModul
|
|            +---PCTGen
|
\---text
        Lantora_DP.pdf
```