VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PŘEVOD UML DIAGRAMŮ MEZI VISUAL PARADIGM A TEXTOVÝMI FORMÁTY

UML DIAGRAMS CONVERSION BETWEEN VISUAL PARADIGM AND TEXT-BASED FORMATS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. LUKÁŠ ONDRÁK**
AUTHOR

VEDOUCÍ PRÁCE — **MAREK RYCHLÝ, RNDr., Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Ondrák Lukáš, Bc.**
Program: Informační technologie
Obor: Informační systémy a databáze
Název: **Převod UML diagramů mezi Visual Paradigm a textovými formáty**
UML Diagrams Conversion between Visual Paradigm and Text-Based Formats
Kategorie: Softwarové inženýrství

Zadání:

1. Seznamte se s modelovacím nástrojem Visual Paradigm a s jeho systémem rozšíření pomocí zásuvných modulů. Seznamte se s nástroji pro textový zápis UML diagramů a jejich vizualizaci, jako je PlantUML a další.
2. Navrhněte zásuvný modul pro Visual Paradigm, který umožní převod (import/export) mezi UML diagramy ve Visual Paradigm a různými textovými formáty.
3. Po konzultaci s vedoucím modul implementujte, alespoň s podporou PlantUML a možností pozdějšího rozšíření o další formáty.
4. Řešení otestujte na různých UML diagramech, vyhodnoťte a diskutujte výsledky. Výsledný software publikujte jako open-source.

Literatura:

- PETRAUSCH, Vanessa, Stephan SEIFERMANN a Karin MÜLLER. *Guidelines for Accessible Textual UML Modeling Notations*. In: Computers Helping People with Special Needs. ICCHP 2016. Lecture Notes in Computer Science, vol 9758. Cham: Springer International Publishing, 2016, s. 67-74. ISBN 978-3-319-41264-1. Dostupné z: [https://doi.org/10.1007/978-3-319-41264-1_9]
- CABOT, Jordi. Text to UML tools - Fastest way to create your models. *Modeling Languages* [online]. 2020 [cit. 2020-10-26]. Dostupné z: [https://modeling-languages.com/text-uml-tools-complete-list/]
- FOWLER, Martin. UML Mode. *ThoughtWorks* [online]. 2003 [cit. 2020-10-26]. Dostupné z: [https://www.martinfowler.com/bliki/UmlMode.html]
- How to Develop Visual Paradigm Plug-in? *Visual Paradigm* [online]. 2011 [cit. 2020-10-26]. Dostupné z: [https://www.visual-paradigm.com/tutorials/plugin.jsp]

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a rozpracovaný bod 3.

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

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 27. října 2020

Abstrakt

Tato diplomová práce se zabývá možnostmi úprav a rozšíření funkcionality programu Visual Paradigm pomocí zásuvných modulů a také studiem nástrojů pro tvorbu UML diagramů. Prvotním cílem práce je popis grafických nástrojů a zároveň méně známé varianty tvorby UML diagramů, kterými jsou nástroje zpracovávající textové UML formáty. Speciální pozornost je přitom kladena na textový nástroj PlantUML a na grafický nástroj Visual Paradigm. Dále se práce věnuje využití otevřeného rozhraní programu Visual Paradigm k tvorbě zásuvných modulů. Hlavním výstupem je pak implementovaný zásuvný modul, který umožní převádět UML diagramy mezi Visual Paradigm a textovým formátem PlantUML, který byl zveřejněn jako open-source. Pro zásuvný modul byla také vytvořena gramatika pro jazyk nástroje PlantUML.

Abstract

This master's thesis deals with possibilities of modifying and extending functionality of Visual Paradigm with plug-ins as well as the study of tools for creating UML diagrams. The primary goal of this thesis is to describe graphical tools and simultaneously, the less known variants of creating UML diagrams. Those are tools that process text UML formats. Special attention is given to text tool PlantUML and to the graphical tool Visual Paradigm. Furthermore, the thesis deals with use of the Visual Paradigm open interface for programmers to create plug-ins. The main output is an implemented plug-in that allows you to convert UML diagrams between Visual Paradigm and text format PlantUML, which was published as open source software. Grammar for the PlantUML language have also been created for this plug-in.

Klíčová slova

Visual Paradigm, PlantUML, Visual Paradigm OpenAPI, UML, TextUML, UML Graph, yUML, CASE, zásuvné moduly do Visual Paradigm, PlantUML gramatika

Keywords

Visual Paradigm, PlantUML, Visual Paradigm OpenAPI, UML, TextUML, UML Graph, yUML, CASE, Visual Paradigm plug-ins, PlantUML grammar

Citace

ONDRÁK, Lukáš. *Převod UML diagramů mezi Visual Paradigm a textovými formáty.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Marek Rychlý, RNDr., Ph.D.

Převod UML diagramů mezi Visual Paradigm a textovými formáty

Prohlášení

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

.....
Lukáš Ondrák
17. května 2021

Poděkování

Chtěl bych poděkovat vedoucímu mé diplomové práce panu Marku Rychlému, RNDr., Ph.D. za profesionální vedení, milý přístup a odborné rady při řešení práce. Také bych chtěl poděkovat své přítelkyni za trpělivost.

Obsah

Kapitola 1

Úvod

Tato diplomová práce se zabývá především dvěma hlavními způsoby modelování UML diagramů. V kapitole 2 jsou nejprve stručně vysvětleny základní informace o modelovacím jazyce UML a jeho dosavadních verzích. Dále se pak kapitola věnuje způsobům modelování UML diagramů, přičemž prvně je popsán nejvyužívanější způsob modelování, kterým se staly grafické nástroje pro tvorbu diagramů. Jedním z nejčastěji využívaných je nástroj Visual Paradigm, který je používán v různých odvětvích, převážně pak při návrhu, implementaci a údržbě softwaru. Poslední část této kapitoly se poté věnuje popisu méně častého způsobu modelování diagramů, kterým jsou nástroje, jež dokáží zpracovat a vykreslit UML diagram vytvořený v textové podobě. Kapitola se také zaměřuje na rozdíly jednotlivých způsobů modelování.

Kapitola 3 přímo navazuje na předchozí kapitolu, neboť se věnuje jednomu z nejpoužívanějších nástrojů pro textové formáty UML, kterým je PlantUML. Konkrétně jsou zmíněny podporované diagramy, obecná specifikace stejnojmenného jazyka a poté podrobnější specifikace z pohledu několika nejvyužívanějších typů UML diagramů.

Funkcionalitu grafického nástroje Visual Paradigm lze snadno modifikovat pomocí zásuvných modulů použitím otevřeného rozhraní v jazyce Java, které program nabízí. O tvorbě a přístupu k vytvořeným diagramům a jejich částem pomocí zásuvných modulů a také obecně o Visual Paradigm se čtenář dozví v kapitole 4.

Hlavním cílem této práce je navrhnout a implementovat zásuvný modul, který dokáže převádět mezi UML diagramy ve Visual Paradigm a textovými formáty UML. Proto se v kapitole 5 zaměříme na požadavky a návrh tohoto modulu.

Náplní šesté kapitoly pak bude podrobný popis implementace navrženého zásuvného modulu, který bude rozdělen dle jednotlivých operací (export a import). Konec této kapitoly se věnuje vytvořené gramatice pro jazyk PlantUML.

Kapitola 7 se zaměřuje na způsob ověřování správnosti implementace pomocí testování a také na zhodnocení dosažené práce. Kapitola také vymezuje, co již implementováno nebylo.

V poslední kapitole jsou nakonec zrekapitulovány cíle práce a shrnuty způsoby, jakými jich bylo dosaženo. V této kapitole je rovněž nastíněno, jakým způsobem lze na tuto práci navázat.

Kapitola 2

UML a způsoby jeho modelování

V této kapitole budou nejprve čtenáři nastíněny základní informace o standardizovaném modelovacím jazyce UML (Unified Modeling Language), jelikož se tato práce tohoto rozsáhlého tématu v mnoha směrech dotýká. Dále se tato kapitola zabývá dvěma hlavními přístupy k datovému modelování a vizualizaci diagramů, tvořených právě v jazyce UML.

První část této kapitoly tedy bude věnována základům UML a datovému modelování obecně. V dalších částech kapitoly budou poté projednány dvě hlavní techniky modelování UML diagramů. Nejprve se v podkapitole 2.2 zaměříme na častěji používanou techniku datového modelování, kterou jsou grafické nástroje pro podporu analýzy a návrhu aplikací (CASE, Computer-Aided Systems Engineering). Někdy jsou za CASE považovány i nástroje pro ostatní fáze, poté jsou CASE rozdělovány na Upper CASE a Lower CASE, přičemž tato práce se bude zabývat pouze prvním jmenovaným, tedy nástroji pro fáze analýzy a návrhu. Nejvyužívanější nástroj z této kategorie je bezpochyby Visual Paradigm (viz kapitola 4.2). V podkapitole 2.3 se čtenář dozví o novějším a méně častém způsobu modelování diagramů, kterým je využití nástrojů užívajících textové formáty UML. Zástupci těchto nástrojů jsou například UML Graph nebo PlantUML. Specifikace druhého zmíněného nástroje bude dopodrobna probrána v následující kapitole (viz kapitola 3), PlantUML tedy není součástí této kapitoly.

2.1 UML

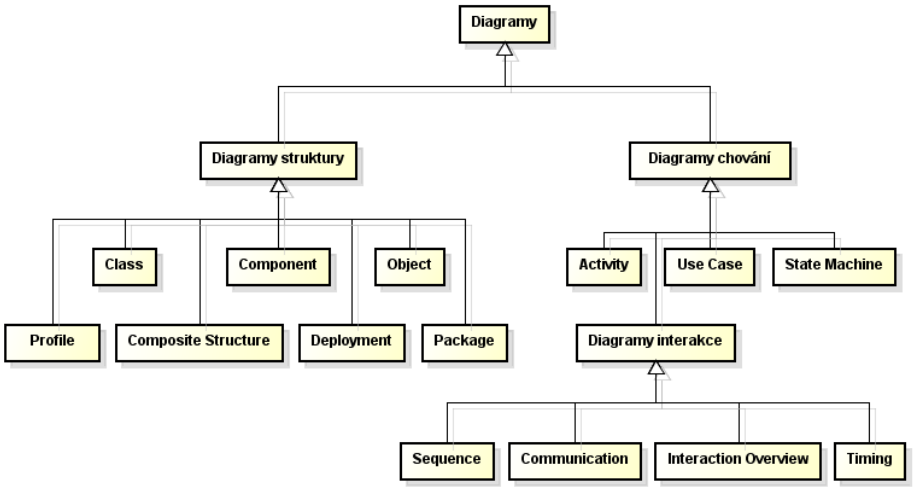
UML (Unified Modeling Language) je standardizovaný modelovací jazyk sloužící pro popis objektově orientovaných systémů. Jedná se o soubor převážně grafických notací, který je využíván hlavně v prvotních vývojových fázích – při analýze požadavků a návrhu systému. Datové modelování se provádí pro snadnější zvládnutí vývoje a údržby především komplexních systémů, ale najde uplatnění i v menších projektech. Modelem je obecně myšlena abstrakce reálného objektu. Je tomu tak i při modelování systému, kdy se pomocí abstrahování odstraňují nerelevantní detaily, které by mohly být zároveň i matoucí. Probíhá tedy snaha o zjednodušení celého reálného systému. [8]

Modelovací jazyk UML má formálně definovanou syntaxi. Různé UML diagramy modelují systém z různých pohledů, žádný se nezaměřuje na navrhovaný systém jako na celek. Díky tomu mohou být i rozsáhlé systémy při správném využití vhodných diagramů relativně snadno pochopitelné, což výrazně usnadňuje sdílení podoby systému (případně výsledků práce) s ostatními návrháři nebo programátory. Navržené modely vytvářeného systému je vhodné předvádět i zákazníkům, jelikož s jejich pomocí půjde snáze vyjasnit uživatel-

ské požadavky. [7] Některé společnosti stále staví celý návrh systému zcela dle původního plánu tvůrců UML, tedy modelují systém pomocí všech dostupných diagramů. Většina společností, která využívá modelování pomocí UML, však dává v dnešní době přednost pragmatičtějšímu přístupu a používají při modelování pouze určitou část UML diagramů, tedy nemodelují systém jako celek ale pouze jeho méně jasné části. Nejčastěji využívanými diagramy jsou diagram případů užití, diagram tříd a sekvenční diagram.

Motivací k vytvoření UML byla převážně snaha o sjednocení a standardizaci souhrnu metod a objektově orientovaných modelovacích jazyků, které se dříve využívaly. První verze jazyka UML byla publikovaná roku 1997. Největší změny přinesla verze 2.0 z roku 2005, ve které přibylo pět nových diagramů a velká část původních byla rozšířena či zjednodušena. Od té doby je jazyk spíše ve fázi údržby než dalšího rozšiřovaní. UML se od verze 2.0 skládá ze 14 typů diagramů. Jejich struktura lze vidět na obrázku 2.1, přičemž typy UML diagramů jsou listové uzly uvedeného stromového diagramu. Momentálně je UML již několik let ve verzi 2.5.1 a není jisté, zda se pracuje na nové verzi. Jeho vývoj je řízen neziskovým mezinárodním konsorciem OMG (Object Management Group), které stojí za mnoha standardy objektově orientovaných systémů. [6]

UML lze považovat také za jazyk pro vizualizaci, specifikaci a dokumentaci softwarových systémů.



Obrázek 2.1: Struktura zobrazující různé skupiny UML diagramů, převzato z [10]

2.2 Grafické nástroje CASE

CASE je zkratka pro Computer-Aided Systems Engineering, tedy vývoj systému s využitím počítačové podpory. Jedná se o souhrnný název pro širokou škálu modelovacích nástrojů. V některých článcích se o CASE mluví i v širším kontextu, tato kapitola se ovšem bude zabývat pouze CASE nástroji pro analýzu a návrh diagramů. Tyto nástroje slouží ke zjednodušení a často i k automatizaci zdlouhavé fáze návrhu systému, a tím pomáhají ke snížení času a nákladů na vývoj softwaru. Jsou tedy určeny primárně pro návrháře, manažery a programátory, kteří na vytvářeném systému spolupracují, aby mohli schématicky a graficky znázornit data a systémové procesy. Vytvářejí rámec pro jeho správu, čímž uživatelům pomáhají v organizaci a zlepšují produktivitu. Většina z nich se přímo nezaměřuje jen

na UML diagramy, ale podporují i modelování databází, cloudových architektur, business modelů aj. Tyto nástroje povětšinou obsahují datový slovník. Tím je myšlen organizovaný centralizovaný seznam datových částí objektů ve vytvořených diagramech (obsahuje např. popisky entit, názvy datových položek či formáty zpráv). Datový slovník obvykle také automaticky spravuje vztahy mezi komponentami a objekty různých diagramů. Tento přístup se kladně projevuje při jejich změnách, jelikož nástroj sám je schopný je distribuovat do všech ostatních diagramů, tedy například změnou názvu metody v diagramu tříd se automaticky změní název i na místě volání této metody v sekvenčním diagramu. [3]

Původním záměrem při vytváření CASE nástrojů bylo téměř zcela nahradit modelováním diagramů programování jako takové, a proto se vyvíjely tak, aby podporovaly objektově orientované programování a agilní procesy vývoje softwaru. Tato odvážná myšlenka automatického generování plně funkčního zdrojového kódu z navržených diagramů se však ukázala jako nevhodná, neboť by touto metodou byl kód velmi špatně udržovatelný a i malá změna by mohla vést k rozsáhlé změně návrhu či dokonce architektury. Přesto však tyto nástroje povětšinou nabízejí částečné generování kódu (přinejmenším šablony) a uživatelské dokumentace, což může být často nápomocné. [3]

Mezi výhody použití těchto typů nástrojů pro tvorbu návrhových diagramů je zejména již zmiňované snížení nákladů a času při tvorbě složitějších systémů. Díky organizovanému přístupu k vývoji získáme vyšší kvalitu výsledného produktu, a tedy konkurenční výhodu. [15]

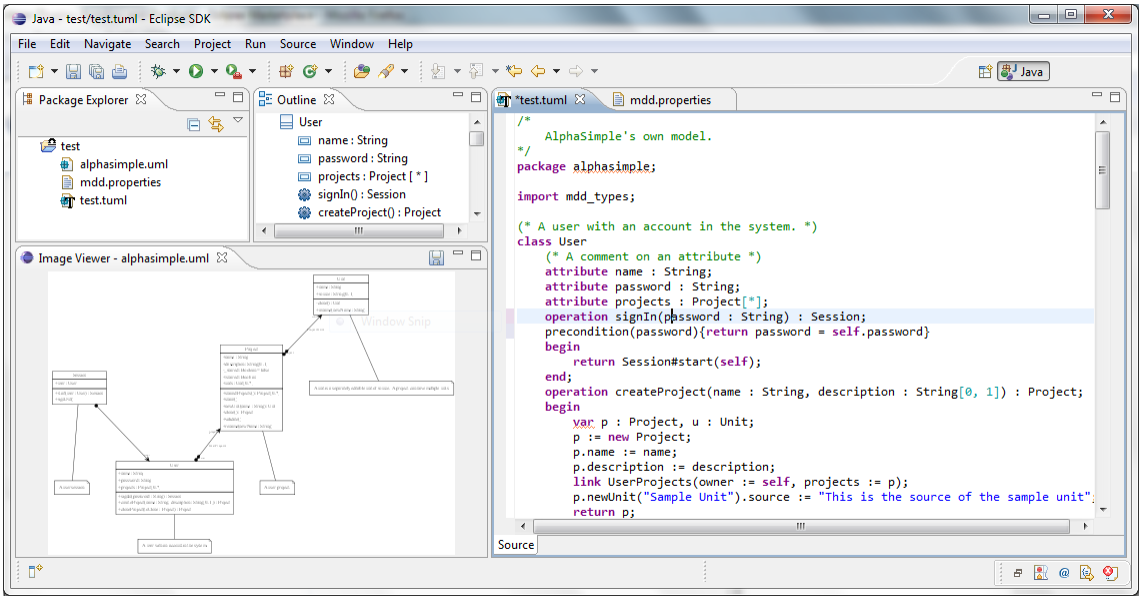
2.3 Nástroje pro textové formáty UML

V souvislostech s modelováním UML diagramů se v posledních letech začíná čím dál častěji mluvit o modelování pomocí textových notací/jazyků, sloužících k textovému popisu UML diagramů. Z těchto textových popisů poté dokáží specializované nástroje vykreslit odpovídající grafický diagram. Tyto nástroje jsou povětšinou přímo vyvíjeny pro specifický jazyk, jako je tomu například u jazyka PlantUML. Stoupající popularitu tomuto přístupu k modelování přináší především integrace s velkou řadou nástrojů (např. verzovací systémy) a zároveň větší blízkost textových popisů k programovacím jazykům (oproti grafickému popisu), což vyhovuje hlavně programátorům. Syntaxe některých textových formátů tedy také proto přímo vychází z programovacích jazyků. Deklarativní reprezentace diagramů je rovněž velmi tvárná, jelikož nebrání velkým změnám, které lze provést snadněji a většinou rychleji než při grafické reprezentaci. [2]

Nástroje pro textové formáty UML jsou však na druhou stranu stále ještě nedostatečně rozvinuté a to především kvůli zaměření většiny jazyků pouze na určité UML diagramy. Další část z nich podporuje jenom ty nejpoužívanější typy. Další velkou nevýhodou v porovnání s grafickými nástroji je absence datových slovníků, nelze tedy vytvářet asociace mezi komponentami z různých diagramů. Kvůli nedostatečné robustnosti a v neposlední řadě také omezené funkcionalitě nejsou (alespoň prozatím) využívány ve velkých projektech, ač mají některé nesporné výhody oproti použití grafických nástrojů. [2] Nejvyužívanějším nástrojem tohoto typu je již zmiňovaný PlantUML, kterému je věnována samostatná kapitola (viz 3). V následujících částech budou popsány některé jeho alternativy.

2.3.1 TextUML Toolkit

TextUML Toolkit je open-source IDE[1], vytvořené v jazyce Java, které slouží pro zrychlení modelovacího procesu. Využívá jazyk TextUML se syntaxí podobnou programovacím jazykům. Podporuje velkou část UML diagramů, přičemž definuje dvě sady notací – pro diagramy struktury a diagramy chování (viz obrázek 2.1). TextUML Toolkit lze využít buď jako samostatný nástroj, nebo jako zásuvný modul do IDE Eclipse. Další variantou je použití online IDE (např. Cloudfier[2]) jako platformu pro vytváření TextUML modelů. [1] Tento nástroj poskytuje většinu funkcionalit, které nabízejí dnešní IDE, jako je zvýrazňování syntaxe nebo verifikace kódu. Dokonce poskytuje grafickou vizualizaci v reálném čase přímo v průběhu modelování. Na obrázku 2.2 lze vidět příklad kódu jazyka TextUML, přičemž nástroj je zde použit v prostředí Eclipse.



Obrázek 2.2: Zásuvný modul TextUML Toolkit v Eclipse IDE, převzato z [5]

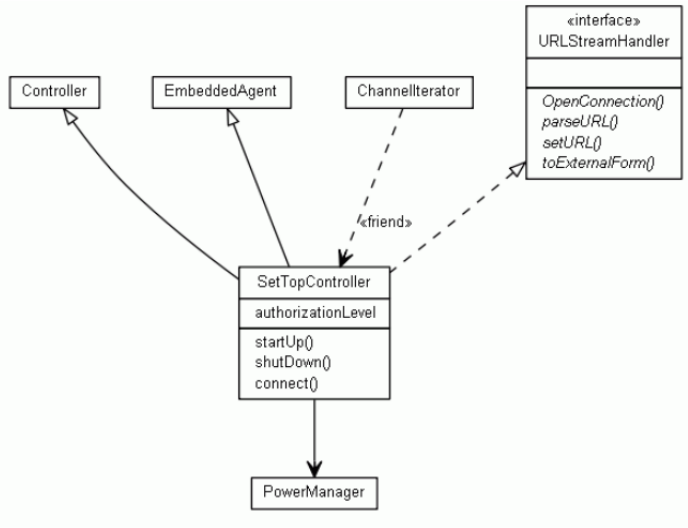
2.3.2 UMLGraph

UMLGraph se v současné době zaměřuje pouze na sekvenční diagramy a omezeně také na diagramy tříd. Syntaxe pro oba diagramy je velmi odlišná. Pro diagramy tříd se využívá syntaxe založená na jazyce Java, doplněná o speciální značení na styl Javadocu[3]. Její ukázka se nachází na levé části obrázku 2.3, přičemž vpravo je grafická podoba modelovaného diagramu. Syntaxe sekvenčního diagramu v jazyce UML Graph používá tzv. pic makra, která definují objekty a volání metod. Balíček `GNU plotutils`[4] obsahuje software `pic2plot`, který dokáže zpracovat tato makra a převést je do grafické podoby (viz obrázek 2.4). Využití tohoto open-source nástroje přináší některé výhody oproti konkurenčním nástrojům, především přehlednou a snadno zapamatovatelnou syntaxi. Nástroj UMLGraph[5]

[1] IDE – Integrované vývojové prostředí
[2] https://github.com/abstratt/cloudfier/
[3] https://www.oracle.com/cz/technical-resources/articles/java/javadoc-tool.html
[4] http://www.gnu.org/software/plotutils/plotutils.html
[5] stažitelný z https://www.spinellis.gr/umlgraph/download.html

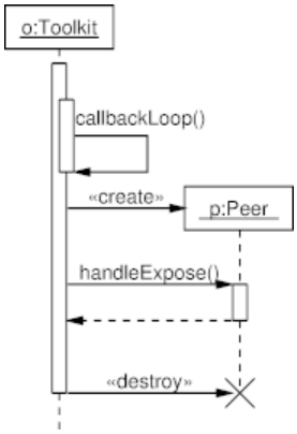
lze také integrovat do IDE Eclipse. Nevýhodou však je využití rozdílných přístupů pro oba typy diagramů, což může působit jako užívání dvou různých nástrojů. Nejspíš proto není tento nástroj v oblasti datového modelování příliš využíván. [2]

```
/**
 * @opt attributes
 * @opt operations
 * @hidden
 */
class UMLOptions {}

class Controller {}
class EmbeddedAgent {}
class PowerManager {}

/**
 * @extends Controller
 * @extends EmbeddedAgent
 * @navassoc - - - PowerManager
 */
class  SetTopController implements URLStreamHandler {
        int authorizationLevel;
        void startUp() {}
        void shutDown() {}
        void connect() {}
}

/** @depend - <friend> - SetTopController */
class ChannelIterator {}

interface URLStreamHandler {
        void OpenConnection();
        void parseURL();
        void setURL();
        void toExternalForm();
}
```



Obrázek 2.3: Syntaxe UML Graph popisující diagram tříd, převzato z [18], upraveno

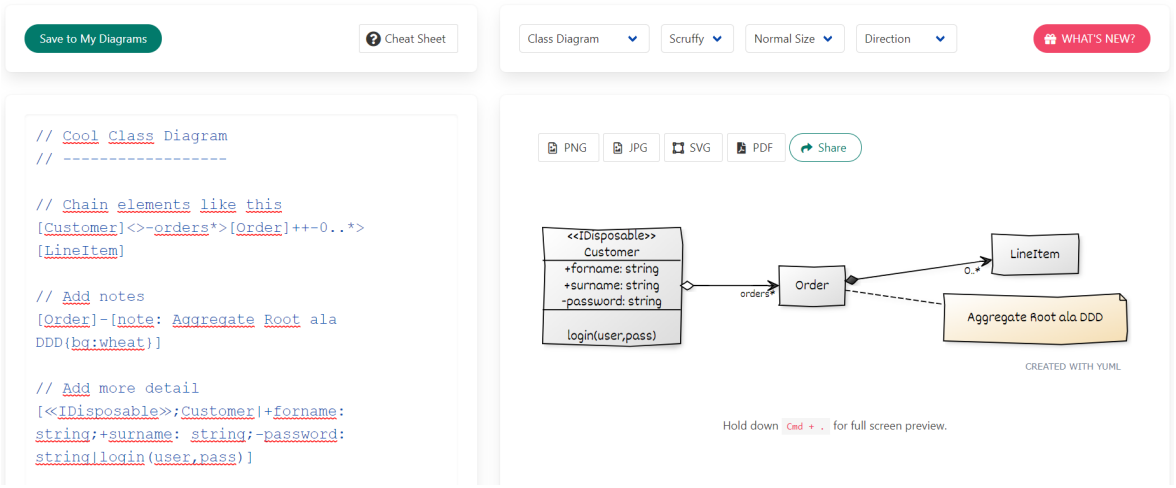
```
# Define the objects
object(O,"o:Toolkit");
placeholder_object(P);
step();

# Activation and messages
active(O);
message(O,O,"callbackLoop()");
create_message(O,P,"p:Peer");
message(O,P,"handleExpose()");
active(P);
return_message(P,O,"");
inactive(P);
destroy_message(O,P);
inactive(O);

# Complete the lifeline of O
step();
complete(O);
```



Obrázek 2.4: Použití pic maker pro modelování sekvenčního diagramu, převzato z [17]

2.3.3 Služba yUML

Online služba yUML poskytuje editor pro tvorbu některých typů UML diagramů, konkrétně jsou jimi diagram tříd, diagram případů užití a diagram aktivit. Jako ostatní nástroje používá vlastní deklarativní jazyk, jehož syntaxe více připomíná grafickou podobu diagramu než u předchozích nástrojů. Tento přístup však bohužel často způsobuje větší nepřehlednost textové podoby diagramu. Naopak výhodou (například oproti dříve zmiňovanému UMLGraph) je jednotnost syntaxe a přístupu modelování u různých typů diagramů. Co se týče integrace s nástroji třetích stran, je služba yUML poněkud omezená, přesný výpis těchto nástrojů je k nalezení na oficiálním webu yUML[6].



Obrázek 2.5: Využití online editoru služby yUML pro vytvoření diagramu tříd

[6] https://yuml.me/integrations

Kapitola 3

PlantUML

PlantUML je nástroj, který na základě textového popisu diagramu ve stejnojmenném jazyce dokáže vytvořit jeho grafickou reprezentaci ve výstupním formátu SVG nebo PNG. Obecné informace o těchto nástrojích se nachází v sekci 2.3, kde jsou také stručně popsány alternativy k tomuto nástroji. PlantUML je napsaný v jazyce Java s důrazem na snadné použití pro koncového uživatele. V současné době je podporováno devět z celkových čtrnácti typů UML diagramů[1], přičemž jsou zastoupeny všechny tři kategorie UML diagramů (viz 2.1). V tomto nástroji lze vytvářet ale i jiné typy diagramů, jako jsou Ganttovy nebo WBS diagramy. Uživatelé mohou také částečně ovlivnit vizuální stránku výsledných diagramů, a to nastavováním barev a stylů pro jednotlivé komponenty, což u konkurenčních nástrojů většinou v celé míře chybí. [21]

Nástroj je možné použít buď pomocí spustitelného Java archivu[2], případně využít online editoru na oficiálních stránkách[3]. Díky velké komunitě kolem PlantUML, je nástroj integrovatelný do velkého množství aplikací[4] (například jako zásuvný modul do IntelliJ IDEA).

Diagramy jsou popisovány pomocí intuitivního jazyka PlantUML, jehož specifikace bude hlavní náplní po zbytek této kapitoly. V následujících podkapitolách bude nejprve rozebrána obecná specifikace jazyka PlantUML, následovaná popisem tvorby konkrétních UML diagramů v tomto textovém formátu.

3.1 Obecná specifikace jazyka PlantUML

Diagramy modelované pomocí jazyka PlantUML jsou obsaženy v jednoduché konstrukci, která označuje obsah PlantUML dokumentu (formát souboru PUML). Ta začíná klíčovým slovem `@startuml` a končí `@enduml`[5], přičemž se tato slova musí nacházet na samostatných řádcích. Uvnitř této konstrukce se postupně nachází jednotlivé elementy tvořící výsledný diagram (rovněž odděleny odřádkováním). V případě modelování elementu, který je tvořen jinými elementy (nebo třeba atributy jako třída), se jedná o blokový element, který má svůj obsah definovaný nejčastěji ve složených závorkách. Typ modelovaného diagramu se určí implicitně dle deklarovaných elementů, míchání elementů specifických pro různé diagramy

[1] jmenovitě diagram tříd, sekvenční diagram, diagram aktivit, stavový diagram, diagram případů užití, objektový diagram, časový diagram, diagram nasazení a diagram komponent

[2] ke stažení na https://sourceforge.net/projects/plantuml/files/plantuml.jar/download

[3] Online Server na https://plantuml.com/

[4] viz https://plantuml.com/running

[5] u jiných typů diagramů obdodně, např. `@startgantt` a `@endgantt`

tedy samozřejmě končí syntaktickou chybou. Většinu elementů lze zapsat více způsoby a s různými volitelnými dovětky (jak bude možno vidět v pozdějších částech), proto není snadné provádět nad jazykem syntaktickou analýzu.

Ke všem diagramům lze přidávat poznámky, přičemž ty se dají přímo svázat s elementy. Velké spoustě elementů, jako jsou například třídy a aktéři, je možné přiřadit alias, který plně zastupuje jejich název. Alias může být nastaven pomocí klíčového slova `as` (například (Usecase) as UC1). Pro použití víceslovného názvu elementu (popř. aliasu) je třeba využít uvozovky.

Jednořádkové komentáře se provádí uvozením komentovaného textu znakem `'`. Víceřádkový komentář začíná sekvencí znaků `\'` a končí sekvencí opačnou, tedy `'\`. [12]

Elementy diagramů i samotné diagramy lze také různě vizuálně přizpůsobovat (např. měnit barvy elementů a textů, velikosti fontů, pozadí diagramu) a to buď přímo při jejich deklaraci, nebo specifičtěji pomocí atributu `skinparam`, který nabízí širokou škálu barev, tvarů a podobně.

Grafické rozložení elementů provádí nástroj PlantUML automaticky zleva doprava a shora dolů dle pořadí jejich deklarací. To je možné změnit, například u diagramu případu užití konstrukcí `left to right direction`, která posune aktéry vlevo od případů užití a většinou přinese přehlednější vizuální výsledek. Také je možné specifikovat svázání některých elementů pomocí bloku `together` pro jejich umístění poblíž sebe. Poznámky je možné umisťovat různě kolem ostatních elementů klíčovými slovy `note left of`, `note right of` a `note over`, za které se zadá daný element. Další grafické výjimky specifické pro dané diagramy budou uvedeny dále v podkapitolách.

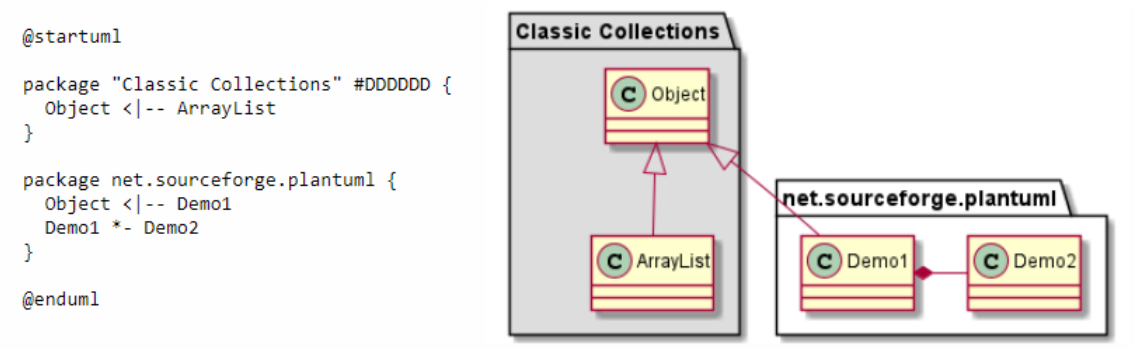
Většinu elementů je možné definovat samostatně nebo vnořené v určitém bloku kódu (např. případ užití v elementu systém).

```
@startuml
note as N1
        testnote
end note

package TestPackage {
        enum TestEnum {
                MONDAY
                TUESDAY
        }

        abstract class TestClass {
                -testAttribute
                +getTestAttribute()
                +setTestAttribute(testAttribute) : void
        }
}

TestEnum .. TestClass : testAssociation
@enduml
```

Výpis 3.1: Příklad diagramu tříd v jazyce PlantUML

3.2 Diagram tříd

Deklarovat třídu v diagramu tříd (Class diagram) [11] lze jednoduše pomocí klíčového slova `class` následovaného názvem třídy. Obdobně je možné nadefinovat[6] i abstraktní třídu (`abstract` nebo `abstract class`), výčtový typ (`enum`), entitu (`entity`), anotaci (`annotation`) nebo rozhraní (`interface`). Podobně jako třídy je možné vytvářet balíky (packages)[7], do kterých lze zanořovat třídy nebo i jiné balíky. Uvnitř nich je také možné definovat vztahy mezi třídami nebo přidávat třídám atributy a metody. Pozadí vykreslených tříd a balíků je možné změnit pomocí hexadecimální hodnoty za jménem (viz obrázek 3.1).



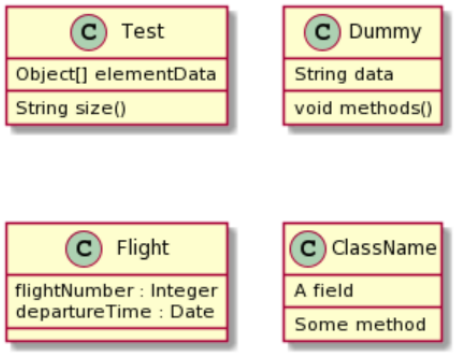
Obrázek 3.1: Vytvoření a obarvení balíků (packages), převzato z [11]

Atributy a metody tříd

Pro deklarování atributů a metod (dále také jako položky třídy) lze využit sekvenci název třídy, symbol : a jméno položky třídy. Tato třída nemusí být dříve deklarována, neboť bude vytvořena automaticky. V případě, že budeme chtít položky specifikovat přímo s deklarací třídy, je nutné vytvořit třídu jako blok (pomocí složených závorek), uvnitř kterého se tyto položky definují. Syntaxe jejich popisu je tedy velmi flexibilní, jak také ukazuje obrázek 3.2. Atributům a metodám lze také volitelně definovat modifikátor přístupu pomocí symbolů uvedených na obrázku 3.3, přičemž symbol se uvádí před deklaraci a ve vykresleném diagramu bude u položky ikona v závislosti na jejím typu a modifikátoru přístupu. Pro definování statické položky třídy stačí před definici zadat klíčové slovo `{clasifier}` nebo `{static}`. Stejně lze definovat abstrakci položky třídy pomocí `{abstract}`. Vyhodnocení, zda se bude jednat o atribut nebo metodu, se provádí buď pomocí toho, zda položka obsahuje kulaté závorky pro parametry metody, nebo také pomocí klíčových slov `{method}` a `{field}`. V případě zadání jednoho z těchto slov bude vyhodnocení upřednostněno oproti předchozímu stylu. Pro úplnost nutno dodat, že při definování obou klíčových slov položce se upřednostní vytvoření metody.

[6] termy v závorce udávají klíčová slova pro deklaraci
[7] případně i jmenné prostory (namespaces)

```
@startuml
class Test
Test : Object[] elementData
Test : String size()

class Dummy {
  String data
  void methods()
}

class Flight {
  flightNumber : Integer
  departureTime : Date
}

class ClassName {
  {field} A field
  {method} Some method
}

@enduml
```



Obrázek 3.2: Čtyři různé možnosti definování atributů a metod v diagramu tříd

Symbol	Ikona pro atribut	Ikona pro metodu	Modifikátor přístupu
-	□	■	private
#	◇	◆	protected
~	△	▲	package private
+	○	●	public

Obrázek 3.3: Tabulka modifikátorů přístupu pro položky třídy, převzato z [11], přeloženo

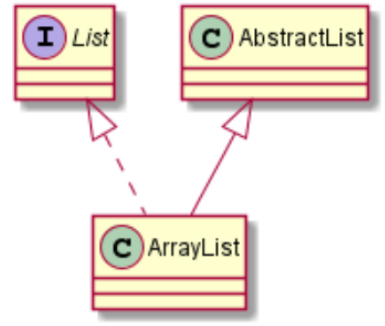
Vztahy v diagramu tříd

V diagramu tříd existuje několik typů vztahů. Nejjednodušší na popis v PlantUML jsou dědění ze třídy (generalizace, specializace) a implementace rozhraní (realizace) jednoduše při tvorbě třídy pomocí klíčových slov `extends` a `implements` (viz obr. 3.4).

Obecnějším přístupem k modelování relací mezi elementy, který dovolí jejich různé typy včetně agregací a kompozicí, je definovat vztahy pomocí sekvence znaků, představující šipky. Různé relace tvořené touto metodou přehledně ukazuje obrázek 3.5. Těmto vztahům lze přidat kardinalita a popis vztahu (např. `Class01 "1" *-- "many" Class02 : contains`). V PlantUML je možné snadno definovat i asociační třídy a to pomocí spojení dvojice tříd s jednou (např. `(Student, Course) .. Enrollment`).

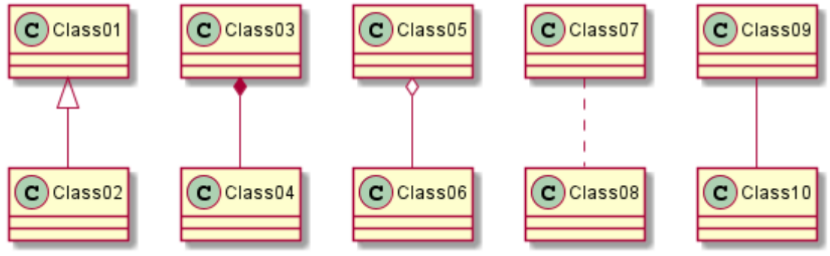
Všechny třídy, které se účastní vztahů, dokonce nemusejí být ani předtím deklarovány, neboť definování vztahu je automaticky vytvoří (zcela základní, tedy bez atributů a metod), jak lze také vidět na následujících obrázcích. Vztahy lze definovat i mezi různými elementy než jen mezi třídami (například vztah mezi balíkem a třídou).

```
@startuml
class ArrayList implements List
class ArrayList extends AbstractList
@enduml
```

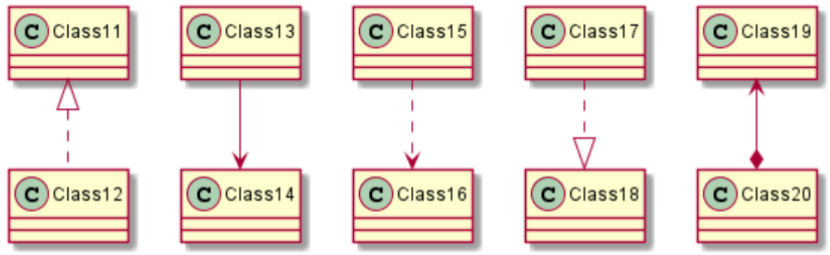


Obrázek 3.4: Ukázka vztahů `extends` a `implements`, převzato z [11]

```
@startuml
Class01 <|-- Class02
Class03 *-- Class04
Class05 o-- Class06
Class07 .. Class08
Class09 -- Class10
@enduml
```




```
@startuml
Class11 <|.. Class12
Class13 --> Class14
Class15 ..> Class16
Class17 ..|> Class18
Class19 <--* Class20
@enduml
```



Obrázek 3.5: Obecnější způsob definování vztahů mezi třídami, převzato z [11]

Stereotypy a generické typy

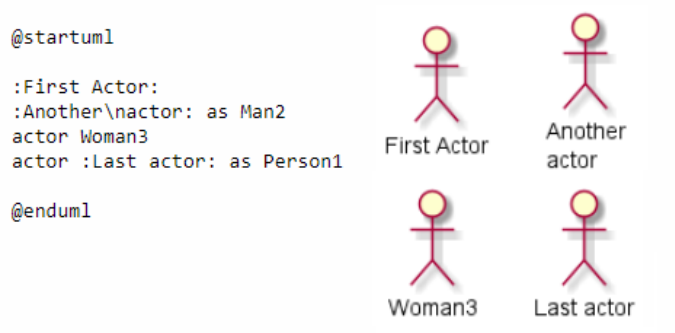
Stereotypy lze třídám přiřadit pouze přímo při deklaraci třídy a to pomocí dvojitých špičatých závorek. Stereotypy jsou další možností vytvoření abstraktní třídy, rozhraní apod. V PlantUML lze elementům jednoduše přiřadit i více stereotypů a to jejich řazením za sebe. Generické typy se vytvářejí totožně, pouze jsou označeny v jednoduchých špičatých závorkách a může být třídě přidán pouze jeden, což vyplývá již z jejich definice. Při nutnosti definování obou vlastností je nutné nejprve napsat generický typ a až poté stereotypy. Příkladem přidání stereotypu i generického typu ke třídě Řidič může být následující část kódu: `class Řidič<? extends Vehicle> <<abstract>>`.

3.3 Diagram případů užití

V diagramu případů užití (Use case diagram) [14] se modeluje vnější pohled na systém, tedy interakce jeho uživatelů. Hlavními elementy jsou tedy aktéři (actor) a obecně definované funkce, které mohou aktéři se systémem provádět (případy užití, use cases). Mezi aktéry a případy užití se popisují relace, kterými jsou buď asociace, generalizace, nebo vztahy include a extend.

Aktéři

Aktéry je možné v PlantUML vytvořit větším množstvím způsobů. Jedním z nich je definováním jména aktéra uvnitř uvozovek nebo dvojteček, případně pomocí klíčového slova `actor` a zadáním jejich jména. Při použití druhé varianty je nutné jméno obalit uvozovkami nebo dvojtečkami, pokud má být víceslovné. Aktérům je opět možné klasickým způsobem přiřadit alias, barvu a stereotypy (viz diagram tříd, 3.2). Ukázka možných syntaxí pro vytvoření aktérů systému se nachází na obrázku 3.6. Při tvorbě diagramu případů užití je vhodné změnit orientaci diagramu pomocí příkazu `left to right direction`, který posune aktéry doleva od případů užití, za účelem vyšší přehlednosti modelovaného diagramu. Pro vztah generalizace u aktérů se používá šipka, vytvořená jako sekvence znaků `<|--` (např. `User <|-- Admin`).

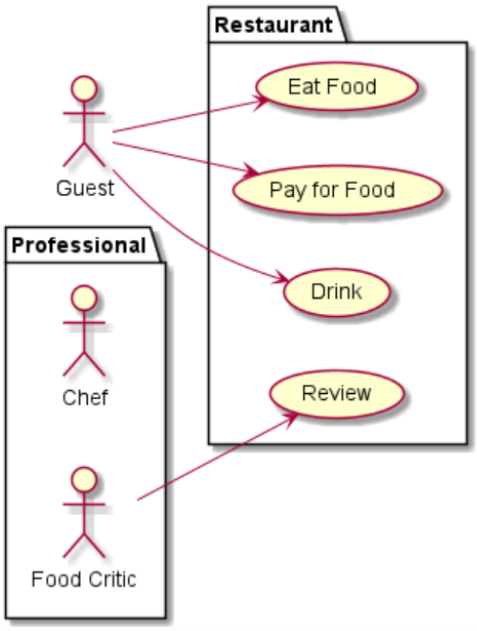


Obrázek 3.6: Několik možností, jak lze vytvořit aktér v PlantUML

Případy užití a vztahy v diagramu případů užití

Případy užití se deklarují buď využitím klíčového slova `usecase` a přidáním názvu, nebo jednoduše obalením názvu závorkami, jelikož připomínají oválný tvar případu užití. Rovněž k němu lze přiřadit alias, barvu a stereotypy (viz 3.2). Asociace a ostatní relace se vytvářejí stejně jako v diagramu tříd a lze jimi rovněž vytvářet elementy (např. aktéry nebo případy užití), které dříve nebyly deklarovány. Aktéry i případy užití mohou být také součástí balíků (packages) stejným způsobem jako u diagramu tříd. Balíky mohou mít navíc zjednodušený grafický model ve tvaru obdelníku, pokud se použije klíčové slovo `rectangle` místo `package`. Příklad jednoduchého diagramu případů užití modelovaný v PlantUML ukazuje obrázek 3.7. Vztahy mezi aktéry a případy užití se definují podobně jako v diagramu tříd, přičemž syntaxe umožňuje i typy vztahů, které jsou nevhodné pro tento diagram (např. kompozice). Vztahům lze samozřejmě přiřadit popisek, což je využitelné například pro definování vztahů `extend` a `include`.

```
@startuml
left to right direction
actor Guest as g
package Professional {
  actor Chef as c
  actor "Food Critic" as fc
}
package Restaurant {
  usecase "Eat Food" as UC1
  usecase "Pay for Food" as UC2
  usecase "Drink" as UC3
  usecase "Review" as UC4
}
fc --> UC4
g --> UC1
g --> UC2
g --> UC3
@enduml
```



Obrázek 3.7: Diagram případů užití v PlantUML, převzato z [14]

3.4 Sekvenční diagram

Sekvenční diagram [13] modeluje interakci několika objektů většinou v rámci jednoho případu užití. Interakce probíhá zasíláním zpráv, případně vytvářením nových objektů. Objekty mají své čáry života (lifelines), které znázorňují jejich aktivitu v průběhu času. Jazyk PlantUML poskytuje obrovskou škálu klíčových slov a konstrukcí, kterými lze tyto diagramy upravovat a vylepšovat, některé z nich budou popsány v následujících podkapitolách.

Deklarace objektů

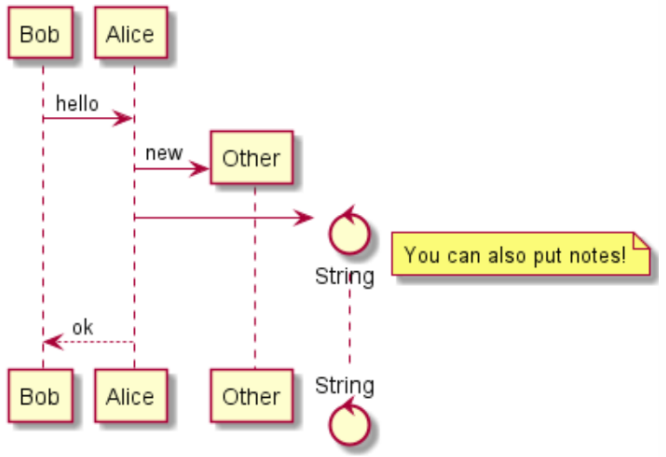
K deklarování objektů se využije klíčové slovo `participant`. Pokud však chceme, je možné definovat i aktéry, entity, databáze apod., kteří se od objektů liší pouze grafickým modelem. Objektům je opět možné přiřadit alias, který je poté nutné využívat místo původního jména (např. při zasílání zpráv). Pomocí klíčového slova `order` a celého čísla se dá objektům určit pořadí, ve kterém se budou zobrazovat na výstupu, přičemž objekty se poté řadí zleva doprava vzestupně dle čísla. Bez definování budou řazeny klasicky zleva doprava dle pořadí deklarace. Pokud chceme uvést jméno objektu, které neobsahuje pouze písmena, je nutné ho opatřit uvozovkami. Objektům lze stejným způsobem jako třídám v diagramu tříd přidávat stereotypy nebo měnit barvy.

Zasílání zpráv

Zprávy lze zasílat podobně jako vznikaly vztahy u diagramu tříd a to pomocí definování odesílatele, sekvence symbolů pro typ zprávy (`->`, `->>` pro asynchronní zprávu, nebo `-->` pro tečkovanou čáru, značící odpověď[8]) a jméno příjemce zprávy. Za tento zápis lze ještě přidat dvojtečka a název/obsah zprávy, kterým je většinou název metody volané na daném

[8] další typy zpráv jsou na https://plantuml.com/sequence-diagram

```
@startuml
Bob -> Alice : hello

create Other
Alice -> Other : new

create control String
Alice -> String
note right : You can also put notes!

Alice --> Bob : ok

@enduml
```



Obrázek 3.8: Příklad jednoduchého sekvenčního diagramu s vytvářením objektů, převzato z [14]

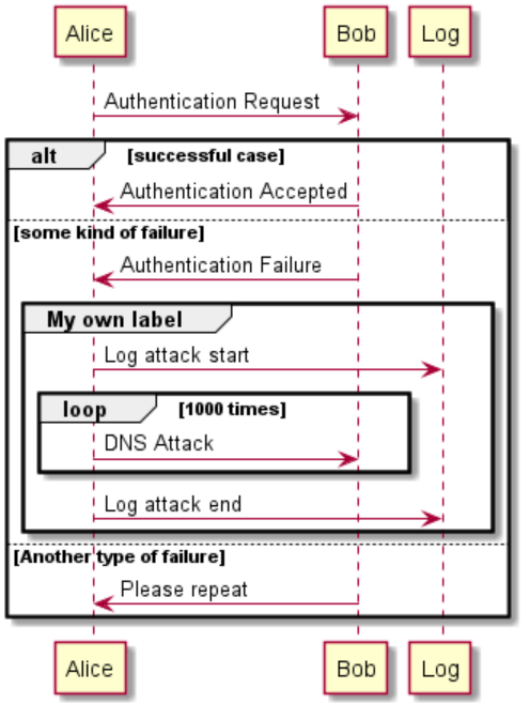
objektu. Pokud chce objekt zaslat zprávu sám sobě, může to provést jednoduše definováním stejného jména pro odesílatele i příjemce. Objekty, které vysílají (nebo přijímají) zprávu nemusí být opět ani předem deklarovány, interakcí ve zprávě budou implicitně vytvořeny. Zprávám lze jednoduše přidat automatické číslování pomocí klíčového slova `autonumber`, případně číslování korigovat. Také lze provést obarvení šipky zprávy a různé další grafické úpravy. Pro vytvoření objektu zprávou se využije konstrukce `create [typ objektu]` a definováním názvu objektu, jak znázorňuje obrázek 3.8. Při využití této konstrukce musí do nově vytvořeného objektu vést následující zpráva, která bude objekt v grafickém výstupu vytvářet.

Tečkovaná zpráva, která většinou značí odpověď, lze také napsat zjednodušeně pomocí klíčového slova `return` a názvu zprávy, přičemž bude mít odesílatele a příjemce určené dle předchozí zprávy (která nebyla v rámci jednoho objektu). Pro vytvoření tzv. nalezené (příchozí) zprávy (v případě zaměření na část komunikace) odesílatel chybí a před sekvenci symbolů pro šipku je vměstnán symbol `[` (např. [->), podobně také u ztracené (odchozí) zprávy, kde není definován příjemce (viz obr. 3.10).

Rámce

PlantUML poskytuje několik typů rámců, jako jsou klasické podmínkové – `alt/else` nebo rámce pro cykly – `loop`, je ale možné si definovat i uživatelské rámce. Vestavěné se vytváří pomocí typu rámce (např. alt), následovaný textem v hlavičce rámce (např. podmínkou). Rámec může, ale nemusí být ukončen klíčovým slovem `end`, přičemž je samozřejmě možné vnořovat rámce do sebe. Jejich velikost je automaticky přizpůsobována dle komunikujících objektů. Vlastní rámce se definují pomocí slova `group` a titulku v hlavičce. Rámce jsou primárně určeny pro sdružování zasílaných zpráv, ale lze je využít i pro definici objektů, fungují tedy podobně jako balíky v předešlých diagramech. Sekvenční diagram obsahující různé typy rámců je na obrázku 3.9.

```
@startuml
Alice -> Bob: Authentication Request

alt successful case

    Bob -> Alice: Authentication Accepted

else some kind of failure

    Bob -> Alice: Authentication Failure
    group My own label
    Alice -> Log : Log attack start
        loop 1000 times
            Alice -> Bob: DNS Attack
        end
    Alice -> Log : Log attack end
    end

else Another type of failure

   Bob -> Alice: Please repeat

end
@enduml
```

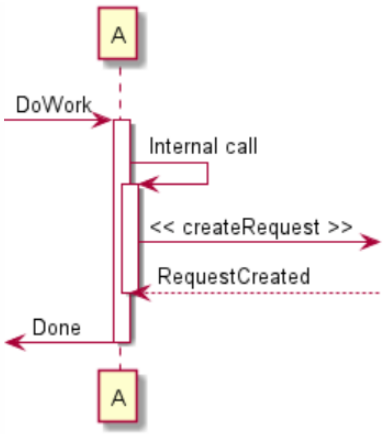


Obrázek 3.9: Příklad sekvenčního diagramu s různými typy rámců, převzato z [14]

Čáry života

Pro vytvoření čáry života (lifeline) se využívá klíčové slovo `activate`, za kterým se definuje, pro který objekt je čára vytvářena. Tato konstrukce se nachází za první zprávou daného objektu, se kterou se pojí, jelikož čára života může vzniknou pouze při zaslání zprávy objektem. Zánik této čáry života lze provést podobně pomocí klíčového slova `deactivate`, případně `destroy`. Další možností aktivace je přidání sekvence symbolů `++` před šipku v zaslané zprávě, podobně pomocí znaků `**` za šipkou zprávy jde vytvořit objekt apod. Čáry života je možné i skládat vedle sebe (viz obr. 3.10).

```
@startuml
[-> A: DoWork

activate A

A -> A: Internal call
activate A

A ->] : << createRequest >>

A<--] : RequestCreated
deactivate A
[<- A: Done
deactivate A
@enduml
```



Obrázek 3.10: Příklad sekvenčního diagramu s více čárami života na jednom objektu společně s nalezenými a ztracenými zprávami, převzato z [14]

Kapitola 4

Tvorba zásuvných modulů do Visual Paradigm

Ve čtvrté kapitole bude čtenáři poskytnut náhled do možnosti rozšíření funkcionalit programu Visual Paradigm pomocí implementace zásuvných modulů. Znalosti, nabyté touto kapitolou, bude moci uplatnit v kapitole 5, která se věnuje návrhu řešení, kde je využíváno termínů z této kapitoly.

V první části bude nejprve vysvětlen pojem zásuvný modul, poté v podkapitole 4.2 získá čtenář stručný přehled o modelovacím nástroji Visual Paradigm. Následně je vysvětlen pojem otevřené rozhraní (OpenAPI), které se využívá právě pro tvorbu zásuvných modulů do programu Visual Paradigm. Následují podkapitoly o základních komponentech a konfiguračním souboru zásuvných modulů. V poslední části je popsáno možné použití OpenAPI a jeho hlavních tříd.

4.1 Zásuvný modul

Zásuvným modulem (také známý jako plug-in nebo addon) je nazýván software, který rozšiřuje funkcionalitu hostitelského programu přidáním nových funkcí nebo úpravou funkcí stávajících. Zásuvný modul tedy nepracuje samostatně, ale je možné ho přidat do hostitelské aplikace jako doplněk. Velké množství aplikací poskytuje vývojářům vlastní aplikační rozhraní (API) pro přístup k datovým položkám a funkcionalitám aplikace. Dříve nejčastěji poskytovali otevřené API pro tvorbu zásuvných modulů grafické aplikace pro zpracování videa a zvuku, v dnešní době se jedná převážně o primární internetové prohlížeče. Jedny z prvních zásuvných modulů vznikaly pro program Adobe Photoshop. [19]

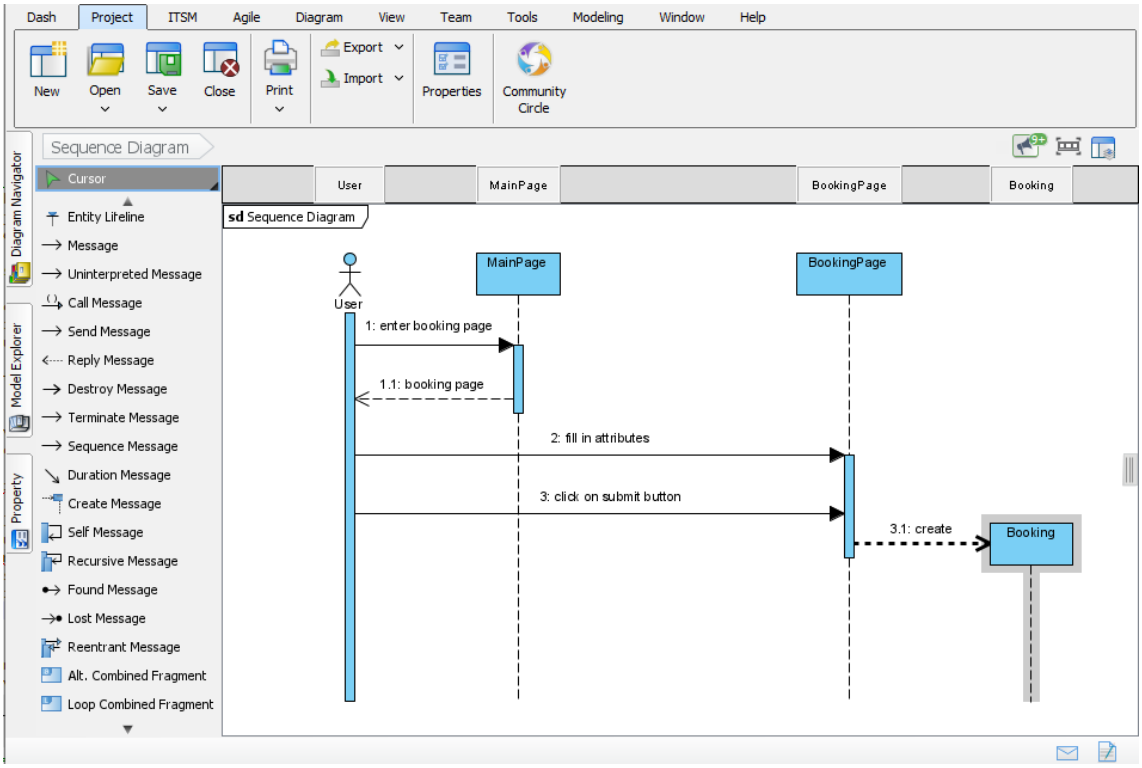
4.2 Visual Paradigm

Visual Paradigm je multiplatformní nástroj pro nákladově efektivní modelování a vizualizaci různých typů diagramů, který byl navržen s ohledem na agilní týmovou spolupráci. Využívá se převážně pro datové modelování UML diagramů a mezi jejich závislostmi. Jedná se o jeden z nejvyužívanějších CASE nástrojů (viz kapitola 2.2), hlavně pro jeho snadné použití, velkou spoustu funkcionalit a velkou komunitu lidí. V nástroji je možné modelovat všechny typy diagramů UML 2 (viz 2.1), ale také různé obchodní, síťové či databázové modely. Má tedy využití v různých odvětvích a zároveň pro různé velikosti projektu. Snadnost použití spočívá v přehlednosti katalogu komponent, provázanosti komponent mezi diagramy a cel-

kové pomoci při modelování. Pro snadnější práci v týmu nástroj nabízí centrální repositář s názvem VPository, verzování diagramů nebo třeba nástroje pro synchronizaci týmu.

Visual Paradigm nabízí jednoduchou a účinnou integraci do vývojových prostředí jako jsou Visual Studio nebo IntelliJ IDEA. V modelovacím nástroji poté lze z vytvořených diagramů přímo vygenerovat zdrojové kódy v jazycích Java či C++ do těchto (i jiných) programů, případně různé typy databází či REST API. [16] [26]

Na následujícím obrázku 4.1 lze vidět ukázka uživatelského rozhraní programu Visual Paradigm.



Obrázek 4.1: Uživatelské rozhraní Visual Paradigm

Verze a licence Visual Paradigm

Tvůrci nástroje poskytují bezplatnou třicetidenní demo verzi pro vyzkoušení. Následně jsou k dispozici čtyři různé úrovně předplatného dle šíře sady poskytnutých nástrojů[1]. Při dlouhodobém intenzivním využití programu je však výhodnější trvalá licence. Pro nekomerční použití většinou postačí komunitní verze, která však obsahuje pouze omezené množství modelovatelných diagramů. Existuje také online verze, která nabízí přímou kolaboraci členů týmu a disponuje širokou škálou dostupných diagramů. Visual Paradigm také nabízí akademické licence pro školy. [20]

[1] přehled funkcionalit jednotlivých verzí je dostupný na https://www.visual-paradigm.com/editions/

4.3 OpenAPI

Visual Paradigm dává vývojářům možnost modifikovat nebo rozšířit funkcionalitu nástroje poskytnutím otevřeného rozhraní (OpenAPI) pro tvorbu zásuvných modulů implementovaných v jazyce Java. Pomocí OpenAPI lze jednoduše přistupovat k datovému modelu a upravovat ho buď jako celek, nebo jen jeho jednotlivé části. Rovněž je možné přímo pomocí implementace vytvářet nové diagramy se stejným smyslem pro detail jako v nástroji Visual Paradigm.

Otevřené rozhraní také dovoluje přizpůsobit chování vestavěných funkcí ve Visual Paradigm. Díky OpenAPI je rovněž možné rozšířit uživatelské rozhraní nástroje jednoduše tím, že vývojář přidá nové položky menu a naprogramuje jejich funkcionalitu po stisku příslušné položky, případně změní chování položky, která se již v menu nachází. [4]

Pro OpenAPI je k dispozici Javadoc dokumentace, která na rozdíl od OpenAPI není dostupná přímo po nainstalování programu, nachází se však na oficiálních webových stránkách Visual Paradigm[2]. Povětšinou však bohužel obsahuje pouze názvy tříd společně s atributy a metodami, přičemž chybí jejich podrobnější popis.

K vytváření zásuvných modulů do Visual Paradigm je k dispozici knihovna s názvem `Openapi.jar`, která se po instalaci programu nachází ve složce `bin` v hlavním adresáři, kde je nástroj Visual Paradigm nainstalován. Tato knihovna je určena, aby se (případně společně s dalšími ve stejné složce) importovala do projektů do vývojových prostředí (např. IntelliJ IDEA nebo Eclipse). Knihovna `Openapi.jar` poskytuje rozhraní právě mezi Visual Paradigm a zásuvným modulem. [4]

4.4 Popis základních komponent

Pro tvorbu zásuvných modulů do Visual Paradigm je nutné pochopit čtyři hlavní komponenty. Těmi jsou Model element, Diagram element, Diagram a Action. V následujících podsekcích budou postupně vysvětleny.

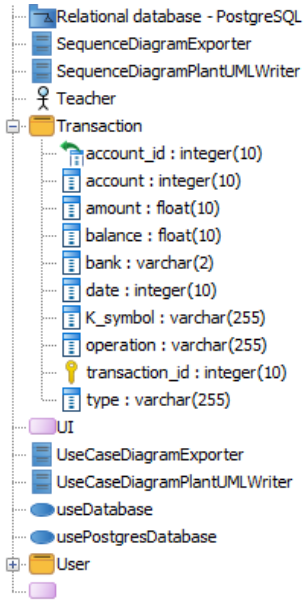
Model element

`Model element` je základní stavební jednotkou diagramů. Tou je například myšlena třída v diagramu tříd a současně i její atributy a metody. Dalšími příklady jsou aktér a případ užití z diagramu případů užití. Nejedná se však o elementy, které se objevují na uživatelském rozhraní při vizualizaci diagramu, jedná se spíše o objekt, popisující jejich vlastnosti. Model elementy jsou hierarchicky uspořádány ve stromové datové struktuře zvané `Model Explorer` (průzkumník model elementů), ve kterém lze přistupovat k jednotlivým elementům z vytvořených diagramů. Příklad této struktury je k vidění na obrázku 4.2. Vývojáři mohou snadno přidávat, upravovat a mazat Model elementy, a přistupovat k jejich atributům přímo v `Model Exploreru` nebo za pomocí zásuvných modulů. [25]

Diagram element a Diagram

`Diagram element` je striktně svázán s Model elementem a představuje jeho model zobrazený na uživatelském rozhraní. Uchovává tedy atributy o pozici elementu v diagramu (vzdálenost objektu od levého rohu, výška a šířka objektu atd.). Jeden Model element může být součástí

[2] https://www.visual-paradigm.com/support/documents/pluginjavadoc/



Obrázek 4.2: Ukázka části stromové struktury Model Explorer

několika diagramů (například dva diagramy tříd mohou sdílet jednu třídu), ovšem v každém diagramu se jedná o jiný Diagram element. [25]

Ve Visual Paradigm existují dva typy Diagram elementů. Prvním typem je `connector` (konektor, spojovací část), který symbolizuje vztah mezi několika (nejčastěji dvěma) Model elementy. Příkladem může být vztah One-to-Many v E-R modelu nebo abstrakce v diagramu tříd. Druhým typem je `Shape` (tvar), reprezentující nevztahové Model elementy jako jsou třídy, aktéři a podobně. [25]

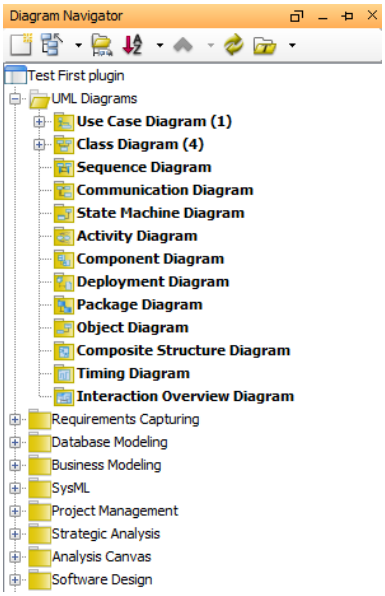
Diagram se skládá z množiny Diagram elementů specifických pro daný typ diagramu. Jeho grafická reprezentace je zobrazena v `Diagram editoru`, kde je možné jeho součásti editovat. `Diagram Navigator` je podobně jako Model Explorer stromová datová struktura, ve které je možné si prohlížet a upravovat vytvořené diagramy, případně vytvářet nové. Obrázek 4.3 poskytuje jeho názornou ukázku.

Action

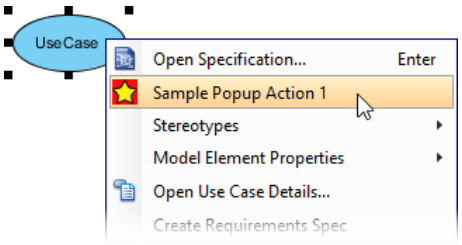
Komponenta `Action` (akce) označuje různá menu nebo jejich části, konkrétně se jedná o tlačítka, vyskakovací menu, případně panel nástrojů diagramu. Jejich stiskem se vyvolá zpracování funkce `performAction` v příslušném `Action Controlleru`, který určuje jejich funkcionalitu (viz 4.6). Příklad akce ve vyskakovacím menu lze vidět na obrázku 4.4.

4.5 Plugin.xml

`Plugin.xml` je název konfiguračního souboru ve formátu XML, který obsahuje základní informace o vyvíjeném zásuvném modulu. Tento soubor slouží primárně k definování relativní cesty k Java třídě, která bude představovat hlavní třídu, implementující VPPlugin z OpenAPI, případně cesty k ovládacím třídám položek menu (tzv. akce, actions – viz 4.4), tedy nových dodefinovaných funkcionalit. Dále slouží k nastavení méně podstatných (ovšem nezbytných) metadat jako jsou unikátní identifikátor, jméno, autor a popis modulu.



Obrázek 4.3: Ukázka části stromové struktury Diagram Navigator obsahující vytvořené UML diagramy



Obrázek 4.4: Vyskakovací menu s akcí definovanou uživatelem, převzato z [25]

V neposlední řadě je v tomto souboru možné formulovat požadované knihovny pro nasazení modulu nebo třeba vlastní tvary komponent diagramů.

Konfigurace akcí

Vzpomínané akce jsou seskupeny v `Action Setech`, které definují množinu podobných akcí. Existují dva typy Action Setů – `actionSet` a `contextSensitiveActionSet`. `ActionSet` je množina akcí, která bude zobrazena v klasickém nebo diagramovém panelu nástrojů. `ContextSensitiveActionSet` definuje vyskakovací okno s množinou akcí. Vytváření akcí je velmi jednoduché. Jednotlivým akcím je třeba v konfiguračním souboru určit typ akce, popisek, jejich umístění a podobné atributy. Přidání akce již vyžaduje pouze určení ovládací třídy (kontroleru). O tom, zda bude akce součástí menu nebo panelu nástrojů, rozhoduje atribut `ribbonPath` (v dřívějších verzích `menuPath`) nebo `toolbarPath` uvnitř elementu action. Atribut `actionType` určuje druh akce, přičemž možné hodnoty jsou `generalAction`, `shapeAction` a `connectorAction`. Hodnota `generalAction` je určena pro běžnou akci, zbylé dvě hodnoty se používají pro definování vlastního tvaru nebo konektoru. [23] Příklad souboru plugin.xml, ve kterém se provádí konfigurace akcí se nachází ve výpisu 4.1.

```
<plugin
  id="sample.genflowofevents"
  name="Generate Flow of Events Report"
  description="Generate an HTML report of use case flow of events."
  provider="Visual Paradigm"
  class="sample.genflowofevents.GenFlowOfEvents">
<actionSets>
<actionSet id="sample.genflowofevents.actionset">
<action
  id="sample.genflowofevents.actions.EventFlowActionController"
  actionType="generalAction"
  label="Generate Flow of Events Report"
  tooltip="Generate Flow of Events Report"
  ribbonPath="Tools/Report">
<actionController
  class="sample.genflowofevents.actions.EventFlowActionController"
/>
</action>
</actionSet>
</actionSets>
</plugin>
```

Výpis 4.1: Konfigurační soubor plugin.xml s vytvořenou uživatelskou akcí v menu, převzato z [23], upraveno

4.6 Tvorba zásuvného modulu

Pro implementaci zásuvného modulu [24] [22] do Visual Paradigm slouží poskytnutá knihovna `Openapi.jar`, kterou je nejprve nutné importovat do projektu. Tuto knihovnu není potřeba stahovat, neboť je součástí instalace nástroje Visual Paradigm. Základem knihovny OpenAPI jsou dvě hlavní rozhraní, které definují hlavní vstupní body do modulu – `VPPlugin`[3] a `VPActionController`[4].

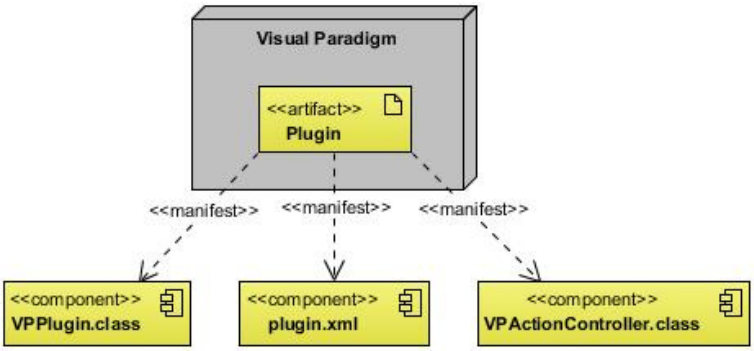
Třída VPPlugin

Rozhraní `VPPlugin` musí být v každém zásuvném modulu implementováno jednou třídou. Relativní cesta k této třídě musí být uvedena v souboru Plugin.xml (viz 4.5) v atributu `class` hlavního elementu `plugin`, jinak nedojde ke správnému nahrání zásuvného modulu. Rozhraní definuje metody `loaded` a `unloaded`, které se vykonají při načtení modulu, respektive ukončení programu Visual Paradigm, dají se tedy použít pro inicializaci a deinicializaci modulu.

Třída VPActionController

Rozhraní `VPActionController` slouží k definování uživatelských akcí (viz 4.4). Každá třída, rozšiřující toto rozhraní, určuje chování jedné akce, a to pomocí metod `performAction`

[3] com.vp.plugin.VPPlugin
[4] com.vp.plugin.action.VPActionController



Obrázek 4.5: Diagram popisující základní strukturu zásuvného modulu Visual Paradigm, převzato z [22]

a `update`. První zmíněná metoda je zavolána při stisku konkrétního tlačítka akce v příslušném menu. Metoda `update` je využitelná pouze pro akce, vyskytující se v hlavním menu (například v kategorii Tools), přičemž je volána při otevření rodičovské kategorie. Tato metoda nemá většinou reálné využití, lze však v ní kupříkladu ověřovat, zda je v daném stavu možné na tlačítko akce kliknout. Atributy uživatelských akcí a cesty ke třídám Controller je nutné zapsat do konfiguračního souboru, jak bylo podrobněji popsáno v kapitole 4.5.

Další užitečné třídy z OpenAPI

OpenAPI dále poskytuje vývojářům třídy a rozhraní, které určují způsob práce s veškerými komponentami vytvořených projektů i s uživatelským rozhraním Visual Paradigm. `ApplicationManager`[5] je hlavní singleton třída, která slouží k získání specializovaných služeb, které OpenAPI poskytuje. Tyto služby spravují třídy `Manager`. Jednou z těchto tříd je `ViewManager`, starající se o funkce grafického uživatelského rozhraní, pomocí kterého je možné například zobrazovat dialogy se zprávami. Správu projektů a přístup k aktuálně otevřenému projektu zajišťuje třída `ProjectManager`.

Rozhraní `IProject` zapouzdřuje veškerá metadata týkající se projektů. Také umožňuje získat strukturu diagramů projektu v podobě kolekce `Iterator` nebo pole objektů `IDiagramUIModel`. Z této třídy lze přistoupit k Diagram elementům konkrétního diagramu a podobně jako `IProject` se stará o metadata. Rohraní `IDiagramModel`, reprezentující Diagram element, má kvůli různému způsobu manipulace s elementy dvě dceřiná rozhraní, `IShapeUIModel` pro nevztahové Diagram elementy a `IConnectorUIModel` pro implementaci konektorů mezi nevztahovými Diagram elementy. Třída, mající na starosti Model elementy, také dodržuje konzistentní názvosloví – `IModelElement`. Množinu Model elementů lze získat buď ze třídy `IProject`, nebo pro konkrétní diagram přímo ze souvisejícího Diagram elementu pomocí metody `getModelElement()`.

Další užitečnou třídou je `DiagramManager`, který lze opět získat pomocí ApplicationManageru. Tato třída poskytuje metodu `createDiagram()`, která bere parametr typu String, k určení typu vytvářeného diagramu. Také obsahuje metody `openAndLayoutDiagram()` a `openDiagram()`.

[5] com.vp.plugin.ApplicationManager

Na vytváření základních komponentů, které jsou popsány v podkapitole 4.4, slouží singleton třídy, využívající návrhový vzor Továrna (Factory). Příkladem takové třídy je `IModelElementFactory`[6]. [24]

Pro další elementy (nejen) UML diagramů, jako jsou třídy, aktéři či zprávy existují specifické třídy pro Model elementy a k ním korespondující Diagram elementy (např. `IMessage` a `IMessageUIModel`).

Nasazení zásuvného modulu

Pro nasazení zásuvného modulu stačí překopírovat adresář se sestrojeným projektem do složky `plugins` v kořenovém adresáři nástroje Visual Paradigm nebo do jeho datového adresáře, který se nachází v adresáři `~/VisualParadigm/plugins` na operačních systémech na bázi Unixu, ve Windows pak v adresáři `%appdata%/VisualParadigm/plugins`. Pro správné nasazení musí mít projekt správnou strukturu, což zahrnuje validní soubor plugin.xml, který má správně nastavené cesty k hlavním třídám (viz 4.5). Zásuvné moduly lze také instalovat přímo v programu Visual Paradigm v záložce Help a to nejsnadněji z archivů ve formátu ZIP. Tento způsob instalace ale pouze umístí obsah zásuvného modulu do datového adresáře, jak kdyby to bylo provedeno ručně.

[6] com.vp.plugin.model.IModelElementFactory

Kapitola 5

Požadavky a návrh zásuvného modulu

Hlavním cílem této práce je vytvořit zásuvný modul do nástroje Visual Paradigm pomocí otevřeného rozhraní (viz 4.3), který dokáže převádět diagramy mezi Visual Paradigm a textovými formáty. Tato kapitola se tedy bude nejprve zabývat popisem požadavků na vyvíjený zásuvný modul a poté jeho návrhem. Následující kapitola bude poté věnována implementací a testováním tohoto navrženého modulu.

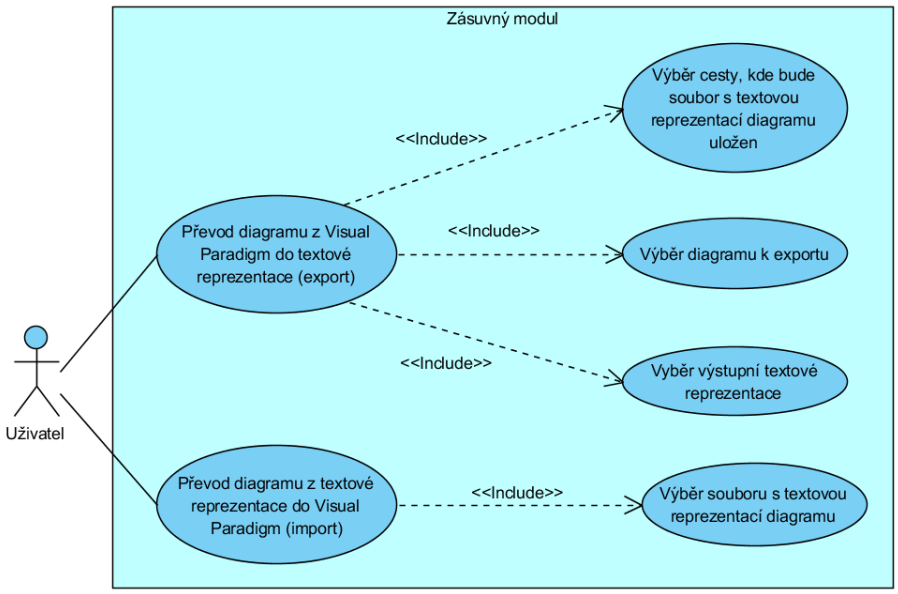
5.1 Požadavky na zásuvný modul

Zásuvný modul do nástroje Visual Paradigm bude sloužit ke dvěma účelům. V prvním bude moci uživatel načíst textovou reprezentaci UML diagramu ve formátu PlantUML do programu Visual Paradigm (importování). Druhým případem použití bude pak opačný postup, tedy vytvoření souboru s textovou reprezentací PlantUML z diagramu, který byl namodelován ve Visual Paradigm. Dodatečným požadavkem je snadná rozšiřitelnost o další textové formáty. Implementace zásuvného modulu, dostatečně otestovaná na různých UML diagramech, se má poté publikovat jako open-source. Na následujícím obrázku 5.1 je diagram případů užití, kde jsou jednotlivé požadavky rozvedeny z pohledu uživatelských interakcí.

5.2 Návrh zásuvného modulu

Popis návrhu vyvíjeného zásuvného modulu se tedy bude přirozeně dělit do dvou velkých úkolů – import a export, jak byly definovány v předchozí části, dle kterých bude rozdělena i tato podkapitola. Tyto dva úkoly budou v rámci svých částí ještě dekomponovány na sérii menších úkolů/fází, což pomůže jak čtenáři pro snadnější pochopení návrhu, tak autorovi v pozdější implementaci. Zásuvný model jsem pojmenoval jednoduše `Text Format Diagrams`.

Pro tvorbu zásuvných modulů je třeba vytvořit konfigurační soubor `plugin.xml` (viz 4.5), ve kterém budou kromě informací o zásuvném modulu vytvořeny dvě akce (viz 4.4). Ty budou umístěny v hlavním menu uvnitř záložky `Project` jako součást položek `Export` a `Import`. Tato akční tlačítka spustí provádění vybrané operace, které budou implementovány ve třídách s názvy `ExportToTextFormatController` a `ImportFromTextFormatController`, které budou implementovat rozhraní VPActionController (viz 4.6), přičemž vstupním bodem budou metody `performAction()`.



Obrázek 5.1: Diagram případů užití vyvíjeného zásuvného modulu

5.2.1 Export

Tato podkapitola se bude zaměřovat na exportování diagramu z Visual Paradigm do textové reprezentace. V první podkapitole se nachází stručný popis návrhu operace export. V dalších podkapitolách následuje podrobnější popis navržených fází, na které byla operace dekomponována.

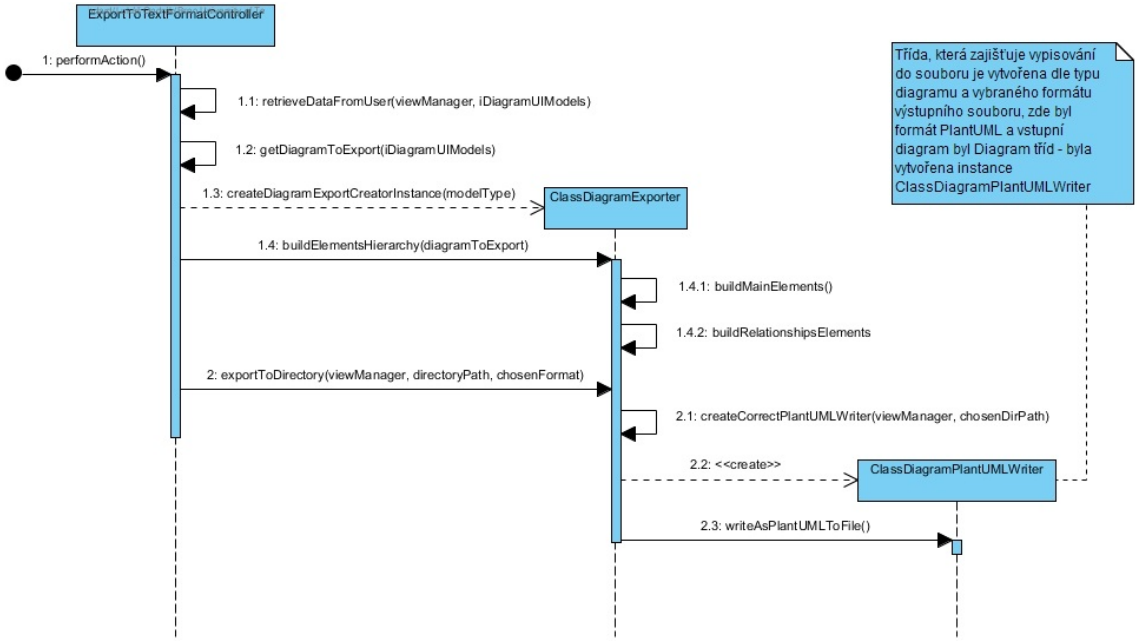
Základní popis operace export

Po vyvolání akce export bude nejprve uživatel vyzván pro zadání informací, které jsou potřebné pro další části algoritmu (viz 5.1). Na základě typu vstupního diagramu bude vytvořena instance abstraktní třídy `DiagramExporter`. V té se následně převede diagram do vnitřní reprezentace elementů. V této třídě se poté vytvoří instance abstraktní třídy `PlantUMLWriter`, jež bude použita pro vytvoření textové reprezentace, a to na základě výstupního textového formátu a typu diagramu. Procházením vnitřní reprezentace se v této třídě bude postupně tvořit diagram v textové podobě.

Na sekvenčním diagramu 5.2 je názorná ukázka zjednodušeného návrhu, přičemž v tomto případě byl výstupním formátem PlantUML a exportován byl diagram tříd. V případě rozšíření o další textový formát bude nutné pouze vytvořit třídu obdobnou třídě `PlantUMLWriter` a zajistit v ní správné vypisování z vnitřní reprezentace elementů.

Interakce s uživatelem

V první fázi operace export bude potřeba od uživatele získat potřebné informace. Uživatel bude proto vyzván k vybrání výstupního adresáře k uložení souboru, výběru exportovaného diagramu a výstupního textového formátu (jelikož zásuvný modul bude implementován rozšiřitelný o další formáty). Uživatel bude mít samozřejmě možnost přerušení exportu v jakémkoliv z těchto kroků. Všechny uživatelovy vstupy (i pro část importu) bude zpracovávat třída `UserInputHandler`. Přímo v metodě `performAction()` budou odchytávány výjimky,



Obrázek 5.2: Zjednodušený návrh fáze export (pro PlantUML a diagram tříd)

vzniklé jak během interakce s uživatelem (např. pokud uživatel nemá v otevřeném projektu žádný vytvořený diagram), tak při následném zpracování diagramu nebo vstupního souboru (u operace import, např. v případě jeho chybné syntaxe), které budou uživatele informovat pomocí message dialogu o vzniklé chybě. K vytvoření dialogu se využije Java Swing společně se třídou `ViewManager` z OpenAPI. Tento přístup bude stejně využit i pro operaci import.

Převod diagramu do vnitřní struktury

Dle názvu zadaného diagramu z předchozí fáze se poté pomocí třídy `ProjectManager` (viz 4.6) získá jeho instance (třída `IDiagramUIModel`), ze které lze přistupovat k jeho Diagram elementům. Z Diagram elementů je poté přímý přístup ke korespondujícím Model elementům (Diagram a Model elementy byly popsány v kapitole 4.4).

Vzhledem k vlastnostem syntaxe jazyka PlantUML je dalším logickým krokem převod struktury Diagram elementů do vlastní interní reprezentace, díky které bude dosaženo snadnějšího přístupu ke všem atributům daného diagramu a celkově pomohou ke snadnějšímu generování textové reprezentace. Tato struktura bude využívat určité hierarchie, jelikož některé atributy, jako jsou jméno nebo barva lze abstrahovat do abstraktních tříd. Podobně mohou také některé elementy do sebe vnořovat jiné. Těmi jsou například elementy package, které mohou obsahovat různé jiné elementy dle typu exportovaného diagramu, proto budou mít abstraktní rodičovskou třídu pro obecný popis elementu package. Tyto typy elementů tím pádem budou mít atributy se seznamy referencí na instance svých vnořených elementů (např. třída z diagramu tříd bude mít takto uložené své atributy a metody, pro které budou také vlastní třídy definující tyto elementy).

Třídy znázorňující elementy budou z hlediska struktury projektu v odděleném balíku `elements`. Ten bude dál dělen na balíky pro elementy, které jsou společné pro některé

diagramy (např. vztahy), a balíky se specifickými elementy pro daný typ diagramu (např. případ užití).

O převod do vnitřní struktury se bude starat jedna ze tříd, rozšiřující abstraktní třídu `DiagramExporter`, dle typu exportovaného diagramu (např. `SequenceDiagramExporter`), a to konkrétně pomocí metody `buildElementsHierarchy(diagramUIModel)`, ve které se bude iterovat nad seznamem Diagram elementů, které exportovaný diagram obsahuje. Ve třídě `DiagramExporter` bude poté celá vytvořená hierarchie elementů dostupná v atributu `elements`.

Generování textové reprezentace

Po vytvoření vnitřní struktury započne generování textové reprezentace. V hlavní třídě, která řídí export, se dle typu diagramu a zadaného výstupního formátu vytvoří instance třídy, implementující abstraktní třídu `DiagramWriter` (např. UseCasePlantUMLWriter). Tato třída bude procházet vytvořenou strukturu a pomocí aplikace pravidel daného textového formátu vytvoří textový popis uložené struktury elementů diagramu. V prvotních verzích návrhu tyto třídy nebyly děleny dle typu diagramu, třídu `DiagramWriter` tedy rozšiřovaly třídy s názvy jako `PlantUMLWriter`, to se ovšem ukázalo jako nevhodné a nesplňující princip jedné odpovědnosti (single responsibility principle). Tato třída však zůstane abstraktní a bude poskytovat některé metody, které budou společné pro její následovníky, přičemž z ní dědí konkrétní třídy pro generování textové reprezentace ve formátu PlantUML. Podobně to je vhodné provést i pro jiné formáty v případě rozšiřování tohoto modulu.

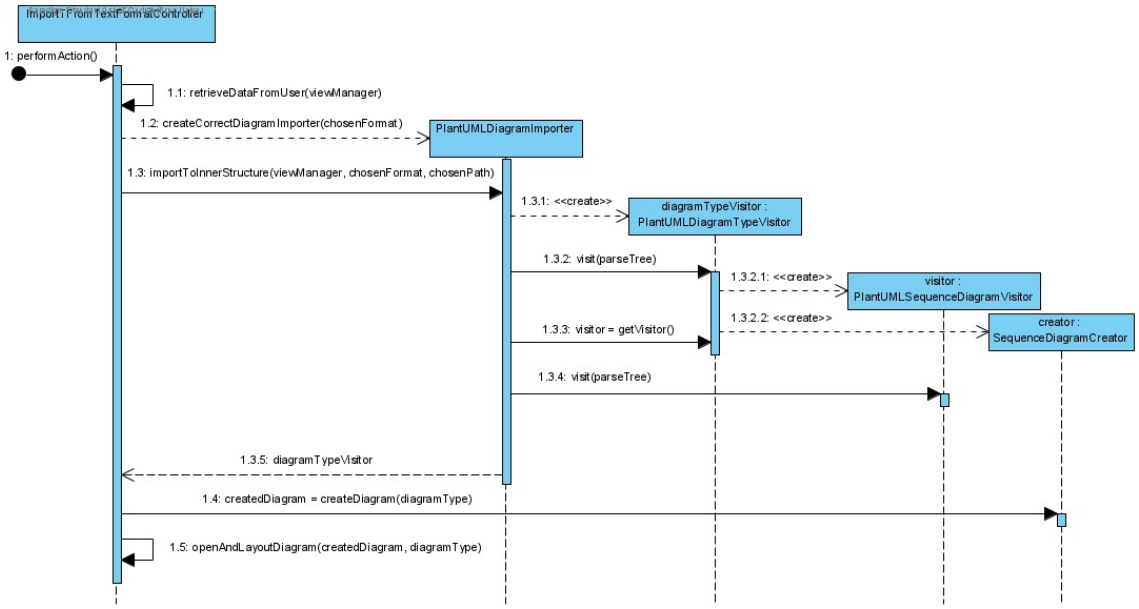
5.2.2 Import

Zbytek této kapitoly bude věnován návrhu operace import, tedy importování UML diagramu z textové reprezentace do nástroje Visual Paradigm. Tato operace je značně složitější oproti exportu, neboť není tak snadné převést textový popis diagramu do interní struktury, jako bylo procházení elementů pomocí otevřeného rozhraní Visual Paradigm. V první podkapitole se nachází stručný popis návrhu této operace. V dalších podkapitolách následuje podrobnější popis navržených fází, na které byla operace dekomponována.

Základní popis operace import

Fáze interakce s uživatelem je v této části velmi přímočará, neboť jediné co uživatel dělá je výběr textového souboru s UML diagramem ve formátu, který zásuvný modul podporuje. K této části tedy bude implementačně přistupováno velmi podobně jako u operace export. Na základě textového formátu vstupního souboru se vytvoří instance abstraktní třídy `DiagramImporter`, ve které je následně prováděna analýza souboru pomocí nástroje ANTLR, přičemž nejprve je pouze zjištěno, o jaký diagram se jedná (dle elementů diagramu), což bude zajištěno ve třídě `PlantUMLDiagramTypeVisitor`. V průběhu tohoto zjišťování se vytvoří korektní instance abstraktní třídy `PlantUMLBaseDiagramVisitor`, pomocí které bude provedena skutečná analýza textového souboru, při které se bude postupně pomocí `visit` metod (návrhový vzor visitor) tvořit vnitřní struktura elementů, když se bude procházet vstupní textový soubor. Při zjišťování, o jaký typ diagramu se jedná, se také vytvoří instance abstraktní třídy `DiagramCreator`, na které bude poté zavolána metoda, která z vnitřní struktury elementů dokáže vytvořit diagram ve Visual Paradigm. Ten je již pouze otevřen v tomto programu a jeho elementy správně rozloženy.

Na sekvenčním diagramu 5.3 je názorná ukázka zjednodušeného návrhu, přičemž v tomto případě byl vstupním formátem PlantUML a importován byl sekvenční diagram. V případě rozšíření o další textový formát bude nutné vytvořit gramatiku, třídu, která bude obdobná třídě `PlantUMLDiagramImporter` a k ní přidružené diagram visitor třídy, které dokáží analyzovat potřebný jazyk a vytvářet vnitřní reprezentaci elementů. Jinou možností by bylo použít již vytvořenou gramatiku pro tento textový formát a převádět text do vnitřní reprezentace pomocí ní.



Obrázek 5.3: Zjednodušený návrh fáze import (pro PlantUML a sekvenční diagram)

Lexikální a syntaktická analýza obsahu textového souboru

Vybraný soubor bude otevřen a provede se jeho lexikální a syntaktická analýza, přičemž případně bude uživatel obeznámen o chybách, které se v souboru nachází. K těmto úkolům bude použit nástroj `ANTLR`[1]. Ten dokáže na základě gramatiky vytvářet syntaktický strom (parse tree). Gramatika pro jazyk PlantUML však není nikde dostupná v dostatečném rozsahu a kvalitě pro tento projekt a oficiální nástroj pro parsování tohoto jazyka používá regulární výrazy. Součástí implementace tedy bude i tvorba gramatiky pro každý podporovaný typ diagramu. Pomocí ANTLR je možné z gramatiky vygenerovat tři druhy syntaktických stromů, přičemž v tomto projektu bude využita varianta s návrhovým vzorem Visitor. ANTLR z poskytnuté gramatiky vygeneruje značné množství souborů (Parser, Lexer, Visitor atd.), které se budou pro přehlednost nacházet v samostatném balíku. Vygenerovaná třída `BaseVisitor` poskytuje `visit` metody pro každé pravidlo gramatiky, které budou volány, pokud bude dané pravidlo použito při analýze obsahu textového souboru.

Vytvoření gramatik pro jednotlivé diagramy je do budoucna stabilní řešení, které bude i dobře udržovatelné. Méně vhodnou variantou by bylo integrování knihovny plantuml.jar, která dokáže z PlantUML reprezentace generovat elementy diagramu. Problémem tohoto řešení je hlavně použití dvou rozdílných struktur pro podobné účely a také závislost implementace na této knihovně, která by se mohla časem výrazně měnit.

[1] ANother Tool for Language Recognition – https://www.antlr.org/

Převod diagramu do vnitřní struktury

Ve třídách, které vzniknou zděděním třídy `BaseVisitor` bude probíhat převod diagramu do vnitřní struktury a současně kontrola sémantické analýzy. Každý podporovaný diagram bude mít pro tento účel vytvořenou vlastní třídu (tedy např. `ClassDiagramVisitor`) kvůli přehlednosti a oddělení tvorby elementů jednotlivých diagramů. Struktura elementů, popisovaná v operaci export (viz 5.2.1) bude využita i u této operace. Pomocí volání metod `visitRulename()`[2] bude postupně tvořena hierarchie elementů, která bude využita v další části při vytváření diagramu ve Visual Paradigm. V rámci budování interní struktury bude také zjištěn a uložen typ diagramu dle popsaných elementů ve vstupním souboru.

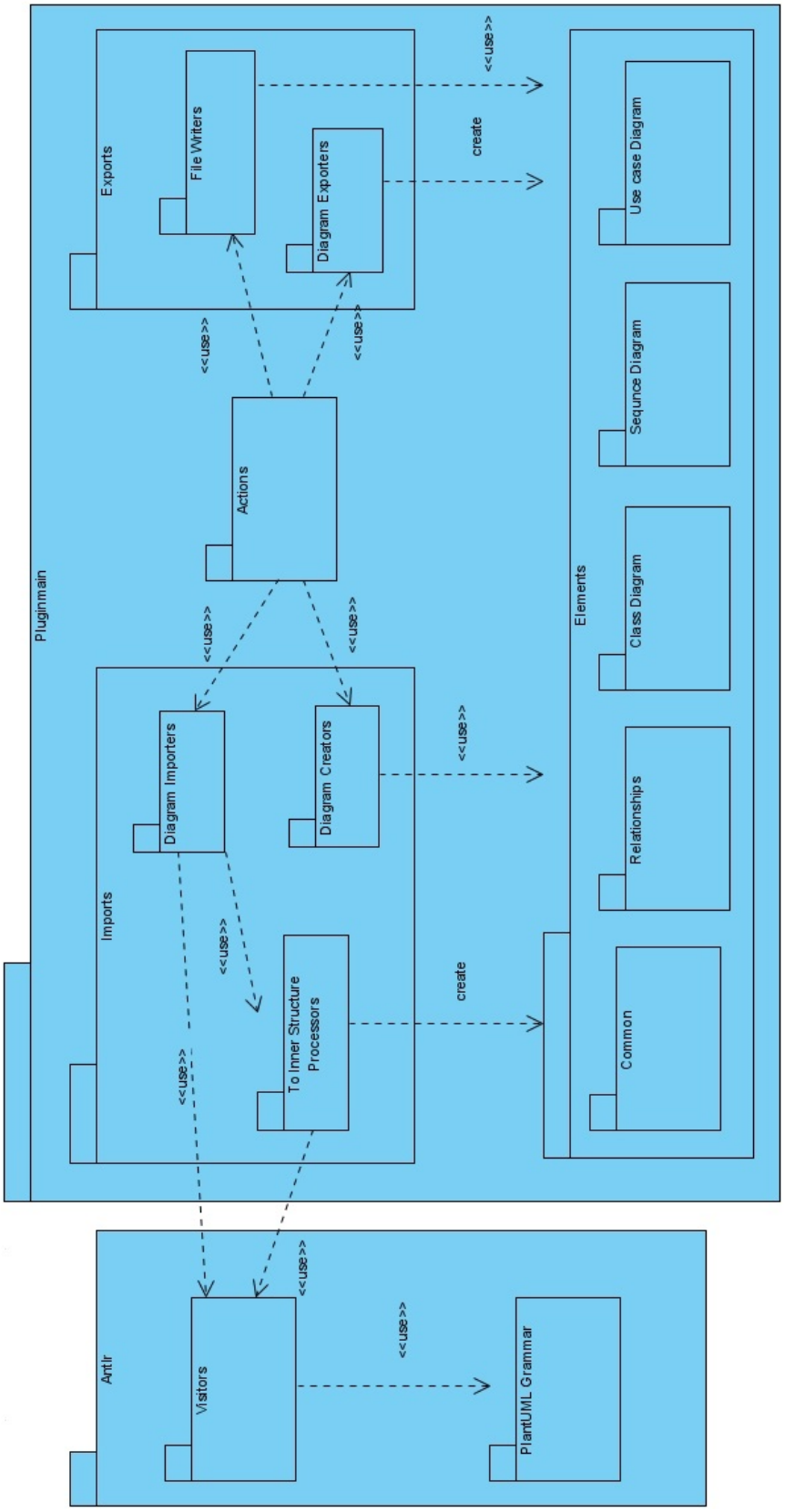
Vytvoření diagramu ve Visual Paradigm

K vytváření diagramů bude sloužit abstraktní třída `DiagramCreator`, kterou podědí třídy pro jednotlivé diagramy (např. `SequenceDiagramCreator`). V těchto třídách se nejprve provede vytvoření modelu diagramu pomocí třídy z OpenAPI (např. `IClassDiagramUIModel`). Tyto třídy lze vyrobit pomocí metody `createDiagram()` ve třídě `DiagramManager` (viz 4.6). Následně bude ve zmiňovaných třídách použita vytvořená interní struktura. Ta bude iterativně a hierarchicky procházena, neboť bude sloužit k vytvoření Diagram a Model elementů jednotlivým elementům ze struktury. Pro tvorbu Diagram elementů bude opět použita třída `DiagramManager`, Model elementy lze vytvořit pomocí `IModelElementFactory`.

5.2.3 Návrh struktury balíků

Package diagram, který je k vidění na obrázku 5.4, přehledně popisuje jednotlivé navržené části, jež budou implementovány. Struktura se dělí na dva hlavní balíky – `Compiler`, který bude řešit veškerou činnost ohledně lexikální a syntaktické analýzy, a `Pluginmain`. V tom se bude nacházet balík `Actions`, ve kterém budou pouze třídy, sloužící jako kontrolery pro vstupní akce (viz 4.4). Tyto třídy budou řídit celou fázi importu nebo exportu. Ve fázi importu budou nejprve využívat balík `DiagramImporters`, který sám používá balíku `Visitors`, určeného k analýze vstupního textového souboru a následně balík, ve kterém se bude vytvářet vnitřní struktura elementů. Z balíku `Actions` se poté využije balík `DiagramCreators`, ve kterém se bude postupně vytvářet správný diagram s elementy dle vnitřní struktury elementů. Fázi export bude řídit třída v balíku `Actions` a to nejprve pomocí tříd z balíku `DiagramExporters`, ve kterých bude probíhat procházení diagram elementů vstupního diagramu a vytváření vnitřní struktury. Ta poté bude využita v třídách pro tvorbu výstupních textových souborů v balíku `FileWriters`.

[2] Rulename bude pro každé pravidlo nahrazeno jeho názvem v gramatice, např. `visitClass()`



Obrázek 5.4: Zjednodušený package diagram navrženého zásuvného modulu

Kapitola 6

Implementace

Následující podkapitoly se budou zabývat implementačními detaily zásuvného modulu do programu Visual Paradigm. Implementace byla prováděna dle návrhu, který byl popsán v kapitole 5. Tato kapitola bude podobně jako zmiňovaná kapitola Návrh rozdělena na hlavní části dle operací – export a import, které jsou popisovány v podkapitolách 6.1 a 6.2. Zásuvný modul byl navrhnut a implementován tak, aby mohl být v budoucnu snadno rozšířitelný o další textové formáty, což byl i jeden z požadavků. Poslední část této kapitoly se věnuje vytvořené gramatice pro jazyk PlantUML. Před samotnou implementací byl vytvořen konfigurační soubor Plugin.xml (jeho účel je popsán v kapitole 4.5), který obsahuje dvě akce. Výpis výsledného souboru je k vidění v příloze (viz B.1).

6.1 Export

Vstupní bod operace export je akce v kontextovém menu `Projekt/Export/Text format`, která je obsloužena metodou `performAction` ve třídě `ExportToTextFormatController`. Tato metoda řídí celý proces exportování a deleguje hlavní úkoly navrženého algoritmu mezi ostatní třídy. V této třídě se nejprve získá instance tříd `ViewManager` a `ProjectManager`, které jsou použity pro přístup k pohledu a projektům (více v kapitole 4.6). Na základě uživatelské interakce je následně získán diagram k exportu a dle jeho typu je vytvořena instance abstraktní třídy `DiagramExporter` (např. SequenceDiagramExporter). Následují fáze převodu elementů diagramu do vnitřní struktury a generování textové reprezentace v PlantUML, které jsou podrobněji rozepsány v následujících podkapitolách. Operaci export u aktivního diagramu lze také vyvolat ve stejném kontextovém menu pomocí speciální akce.

6.1.1 Převod diagramu do vnitřní struktury

Převod do vnitřní struktury, která je tvořena vnořenými seznamy (především `ArrayList`) elementů, je zajištěn ve třídě `DiagramExporter`. Pro procházení diagram elementů (viz 4.4) se pomocí třídy `IDiagramUIModel` získá kolekce Iterator. Pro snažší průběh vytváření struktury se nejprve při průchodu kolekce v metodě `buildMainElements` zpracují všechny elementy kromě vztahových elementů, které budou následovat v druhém průchodu kolekce v metodě `buildRelationshipElements`. To se provádí zejména pro jednodušší znovupoužití, případně overriding těchto metod v případě rozšiřování funkcionality o jiné diagramy nebo formáty.

Převod nevztahových elementů do vnitřní struktury

Při převodu nevztahových elementů do vnitřní struktury elementů se poprvé využije pomocná singleton třída `ExportDiagramInfo`, ve které jsou udržovány informace o exportovaném diagramu, přičemž si uloží všechny jeho diagram elementy. Následně se již prochází kolekce Iterator, která postupně poskytuje instance třídy `IDiagramElement`. U každého diagram elementu se zjistí, zda žádný jiný diagram element není jeho otcem, tedy zda se jedná o kořenový diagram element. V případě, že by nebyl kořenovým, tak by byl přeskočen a prováděn až při procházení synovských elementů svého otce. V dřívějších verzích implementace se elementy diagramu procházely pomocí jejich model elementů (viz 4.4), ovšem aby bylo zajištěno exportování i tzv. auxiliary views (pomocných pohledů), které v exportovaném diagramu nemají model element ale jen pohled (diagram element), musela být implementace změněna právě na ně. Model elementy jsou z nich následně získány pomocí metody `getModelElement` (u auxiliary views se získají z diagramu, kde je hlavní pohled, tzv. Master view).

Získaný model element je poté zpracován buď metodou `processChildrenBaseElement`, pokud se jedná o typické elementy, které slouží k vnořování elementů (např. Package či System), nebo některou z implementací abstraktní metody `processNotMainPackageElement` dle typu diagramu. První jmenovaná je vytvořena genericky, je tedy společná pro všechny typy podporovaných diagramů, neboť umožňuje vytvářet balíky, modely, systemy apod. (dále jako ChildrenBase element), což zamezí určitému množství duplikovaného kódu. V této metodě se získá barva elementu, jméno, pozice atd. Kvůli zvláštnímu návrhu OpenAPI není možné získat stereotypy přímo z rodičovské třídy `IHasChildrenBaseModelElement`, ale je nutné ji přetypovat přímo na konkrétní dle typu elementu (např. ISystem), což přivodilo jisté neodstranitelné duplikace kódu při této činnosti. Po nastavení parametrů a stereotypů instanci jedné ze synovských tříd abstraktní třídy `ChildrenBasedElement` je takto vytvořený element přiřazen do struktury `elements`, což je atribut třídy `DiagramExporter`, ve kterém se ostatní elementy nacházejí. Dále jsou již pouze procházeny synovské diagram elementy, na které je zavolána metoda `processNotMainPackageElement`, která zpracovává ostatní elementy, které buď v diagramu mají otcovský diagram element, nebo nemají ale nejsou ChildrenBase element. Pokud byla metoda `processChildrenBaseElement` volána na synovský diagram element jiného elementu, je princip zpracování stejný, jen bude vytvořený element přiřažen do vnořené struktury daného otcovského elementu, který má za atribut seznam ChildrenBase elementů.

Uvnitř implementace abstraktní metody `processNotMainPackageElement` se dle typu exportovaného diagramu zpracovávají různé typy elementů. Pro příklad uveďme diagram případů užití, který podporuje následující elementy: případ užití, aktéra, ChildrenBase element a poznámku (note). Pro jejich zpracování se dle typu elementu zavolají specifické metody, například `processActor` nebo `processNote`. Uvnitř nich jsou z daného elementu (např. IUseCase) získány požadované informace, které jsou nastaveny do vytvořeného elementu vnitřní reprezentace (např. UseCaseElement) a ten je následně uložen stejným způsobem jako dříve popsané ChildrenBase elementy.

U většiny elementů byly tyto process metody velmi přímočaré, naopak například u elementu třída (class) musely být zpracovány i generické parametry (template parameters), atributy a operace třídy aj., následně u atributů a operací třídy bylo potřeba zjistit a uložit typ, viditelnost, zda se jedná o abstraktní a statickou položku apod. Třída `ClassElement` nakonec má pět klasických atributů a navíc čtyři seznamy jiných elementů nebo řetězců[1].

[1] jmenovitě jsou jimi: atributy, operace, template parameters, enumeration literals

U všech typů elementů, kromě klasických relací, se nastavuje také atribut úroveň zanoření, který bude vysvětlen v další podkapitole, jelikož je využíván až při generování textové reprezentace diagramu.

Dalším zajímavějším elementem na zpracování byl rámec/kombinovaný fragment (combined fragment) ze sekvenčního diagramu, který představuje skupiny zpráv (např. podmínkový blok if). Ten je v OpenAPI reprezentován třídou `ICombinedFragment` a jeho jednotlivé části jsou uloženy jako synovské elementy, které jsou instancí třídy `IInteractionOperand`. Tyto operandy však mohou mít znovu vnořené skupiny zpráv (např. loop v podmínce v else větvi). Při procházení těchto operandů se do vnitřní reprezentace prvnímu operandu nastaví název operátoru (např. loop) a ostatním je přidělen název else, pro následné jednodušší vypisování. Zpracování operandů zahrnuje mimo jiné projití a uložení všech zpráv, jejichž struktura je dostupná pomocí metody `toMessageArray()`, a také ostatních vnořených elementů. Do vnitřní reprezentace se třída pro skupiny zpráv nazývá `GroupElement` a jedná se o specializaci dříve zmíněné třídy `ChildrenBasedElement`. Její součástí je poté seznam instancí třídy `GroupCaseElement`, která reprezentuje operandy.

Do process metod (např. `processActor`) většiny elementů byl parametrem předáván ChildrenBasedElement, z kterého byla tato metoda zavolána, nebo hodnota null, pokud byla volána přímo. Díky tomu nebylo nutné ve vnitřní struktuře vyhledávat, když se do jejich seznamu měl tento nově vytvářený element přidat. Pokud byla hodnota null, element je uložen přímo do „kořene“ struktury `elements`.

Převod vztahových elementů do vnitřní struktury

Vztahové elementy jsou převáděny pomocí metody `processRelationshipElements` ve třídě `DiagramExporter`. Tato metoda převádí buď různé typy relací, které lze modelovat v diagramech tříd a případů užití, jako jsou asociace, realizace či třeba asociační třída, nebo zprávy sekvenčního diagramu, jelikož je ve třídě `SequenceDiagramExporter` její implementace přepsána. U všech relací se ukládají minimálně jméno elementu ze kterého relace vychází a jméno elementu do kterého směřuje (dále jako aktéři relace). Proto byla vytvořena abstraktní třída `RelationshipElement` s těmito atributy, ze které všechny vztahové třídy vnitřní reprezentace dědí. Model elementy aktérů relace se získají pomocí metod otevřeného rozhraní, konkrétně se jedná o metody `getFrom` a `getTo` třídy `IRelation`, ze kterého dědí všechny typy relací v OpenAPI.

Při ukládání vytvořených vztahových elementů vnitřní struktury se tyto ukládájí přímo do „kořene“ struktury `elements`. U asociací jsou získávány také začátky a konce této relace (třída IAssociationEnd), neboť z nich lze následně získat multiplicity vztahu, druh agregace a zda je na konci šipka, křížek apod. Asociační třída je ve Visual Paradigm v podstatě tvořena dvěma relacemi, proto získání všech aktérů není obtížné. Pro ostatní typy relací diagramů tříd a případů užití byla vytvořena společná třída `SimpleRelationshipElement`, které je přiřazena jako atribut celá třída s informacemi o vztahu (`ISimpleRelationship`), přičemž její zpracování je řešeno až v rámci vypisování. Tento přístup je u těchto typů relací vhodný, neboť se pouze při vypisování zjistí daný typ relace a dle něj je určena výsledná šipka[2].

Metoda `processMessage`, volaná buď z metody `processRelationshipElements`, nebo při zpracovávání operandu skupiny (viz předchozí podkapitola), vytváří instance třídy `MessageElement`. Popisek zprávy, který je povětšinou jméno volané metody, může být získán ze dvou zdrojů. Prvním je klasické zavolání metody `getName()` na zpracovávaném

[2] v jazyce PlantUML například sekvence znaků <|- pro generalizaci

elementu a druhým tzv. tranzitní operace. Jméno z tranzitní operace je získáno tak, že je využito metody `getModelPropertyByName`[3], díky které je možné zjistit určitou vlastnost dané zprávy (obecně elementu). Tato metoda je využita pro zjištění model elementů, které jsou transitní k této zprávě. Z těchto model elementů je poté získaná správná operace, jejíž název je nastaven popiskem zprávy. Zprávě jsou poté klasicky zjištěni aktéři, kterými mohou být elementy typu `InteractionLifeLine` nebo `Actor`. U prvního jmenovaného typu je také nutné nastavit správné výstupní jméno, neboť to může být tvořeno jak jménem instance, tak objektu, případně pouze libovolným z nich. Tyto typy elementů jsou ve vnitřní reprezentaci uloženy jako `ParticipantElement`, neboť se takto nazývají v PlantUML. V dalším kroku zpracování zprávy je řešen tvar šipky – a to konkrétně synchronnost, typ akce (destroy, return, ...) nebo také, zda se nejedná o zprávu, která vytváří aktéra (create message).

Zpracování aliasu u elementů

Jak bylo popsáno v kapitole 3.1, PlantUML ve své syntaxi podporuje u spousty elementů tzv. alias. Ten se dá považovat za jakési druhé jméno, které je poté nutné využívat místo jména, tedy například pokud je element aktérem relací. Alias je v tomto zásuvném modulu podporován pro následující elementy: třída, balík, poznámka, aktér zprávy, případ užití a aktér v diagramu případu užití. Jelikož elementy otevřeného rozhraní Visual Paradigm nic jako alias neznají, musela v nich být tato informace uložena jiným způsobem, neboť alias může být importován z textové reprezentace a je nutné ho uchovat pro zpětný export. Toho bylo docíleno jeho nastavením jako jeden ze stereotypů, který má následující tvar: `$alias: název`. Tento stereotyp je v diagram elementu (tedy ve svém pohledu) nastaven tak, aby se nezobrazoval.

6.1.2 Generování textové reprezentace v jazyce PlantUML

Po dokončení vytváření vnitřní struktury elementů je vše připraveno pro převod do textové reprezentace. Dle vybraného výstupního formátu a typu diagramu je vytvořena instance třídy, ve které bude generování probíhat (např. `ClassDiagramPlantUMLWriter`). Elementy ve vnitřní struktuře `elements` jsou v případě diagramu tříd a případů užití seřazeny dle x-ové souřadnice, u sekvenčního diagramu jsou takto seřazeni pouze aktéři komunikace a poté taky zprávy dle y-ové souřadnice. Následně je již procházena vnitřní struktura a jednotlivé elementy, společně se svými vlastnostmi, jsou dle specifikace PlantUML vypisovány do vytvořeného a připraveného souboru klasicky pomocí třídy `FileWriter`. U zanořených elementů je vypisováno odsazení (velikost tabulatoru), které je řízeno zmiňovaným atributem úroveň zanoření, který je při procházení při tvoření elementů inkrementován o jedna vždy při dalším zanoření.

Při vypisování jmen elementů (tím pádem i aktérů relací) je nutné si dát pozor na tvar, neboť víceslovné názvy musí být ohraničeny dvojtečkami (u aktérů v diagramu případů užití), závorkami (u případů užití), nebo obecně uvozovkami.

U některých elementů může některý stereotyp částečně změnit jeho výslednou syntaxi, například pokud participant element (aktér zprávy v sekvenčním diagramu) obsahuje stereotyp actor[4], tak se místo klíčového slova `participant` využije právě `actor`. Speciální částí syntaxe jazyka PlantUML je také vytvoření aktéra zprávy (create message), kdy je před

[3] např. iMessage.getModelPropertyByName(IModel.PROP_TRANSIT_FROM).getValue()

[4] podobně pro entity, boundary a control

touto zprávou vypsána sekvence `create [typ aktéra] [jméno aktéra]`, jak je ostatně možné vidět i na obrázku 3.8.

Pro vypisování nalezených a ztracených zpráv v PlantUML se místo aktéra využije symbolu `]` respektive `[`, jak je ukázáno na sekvenčním diagramu na obrázku 3.10.

Alias je získáván z pole stereotypů, jelikož je mu nastaven speciální tvar (jak bylo popsáno v podkapitole Zpracování aliasu u elementů) a pokud je nastaven, je využíván (po upravení na správný tvar) ve všech relacích, kterých je daný element aktér.

6.2 Import

Vstupním bodem operace import je, podobně jako v operaci export, akce v kontextovém menu `Projekt/Import/Text format`, která je obsloužena metodou `performAction` v kontroleru `ImportToTextFormatController`. Nejprve je uživatel opět vyzván k vybrání cesty ke vstupnímu souboru a jeho formátu (pro případ rozvoje zásuvného modulu o další formáty). Na základě formátu je poté vytvořena instance třídy `DiagramImporter`, kterou je zatím pouze `PlantUMLDiagramImporter`.

6.2.1 Vytvoření syntaktického stromu ze vstupního souboru

Jelikož nástroj PlantUML neposkytuje gramatiku pro svůj jazyk (dokonce ji ani nevyužívá, analýza probíhá pouze pomocí regulárních výrazů) a zároveň není tato gramatika veřejně dostupná na internetu od jiných tvůrců (v potřebné kvalitě a především rozsahu), bylo rozhodnuto, že bude vytvořena vlastní od základu. Pro tento účel byl využit nástroj ANTLR4[5]. Pro tento nástroj je potřeba vytvořit gramatiku dle speciální syntaxe do souboru ve formátu `g4`, z kterého se poté právě pomocí nástroje vygenerují potřebné soubory pro dynamickou analýzu (Lexer, Parser, BaseVisitor apod.), přičemž některé budou dále používány, jelikož se jedná o javovské třídy. Ty jsou pojmenovány dle názvu g4 souboru (např. PlantUMLLexer), který udává jméno gramatiky. [9] ANTLR má dobrou podporu v různých vývojových prostředí, autor využil zásuvný modul ANTLR v4 do IntelliJ IDEA[6]. Vytvořená gramatika bude více popsána na konci této kapitoly.

V rámci dalšího pokračování zpracování vstupního textového souboru je zavolána metoda `importToInnerStructure` třídy `DiagramImporter`, ve které jsou využity javovské třídy, zmiňované v předchozím odstavci. Nejprve jsou vytvořeny instance potřebných tříd, které jsou nutné k vytvoření instance třídy `PlantUMLParser`. Té je ještě nastaven vlastní listener na syntaktické chyby a poté je využita pro vytvoření jednoduchého syntaktického stromu (třída `ParseTree`). To je provedeno zavoláním metody, jejíž název je shodný s počátečním neterminálem vytvořené gramatiky. V případě lexikálních nebo syntaktických chyb je o nich uživatel obeznámen a import je ukončen.

6.2.2 Tvorba vnitřní struktury pomocí procházení syntaktického stromu

Další třídy, které ANTLR vygeneruje z gramatiky jsou `BaseVisitor` a `BaseListener`, které každá slouží pro jeden typ procházení syntaktického stromu. V tomto projektu byl využit přístup pomocí třídy `BaseVisitor`, která ve visit metodách (návrhový vzor Visitor) definuje chování při navštívení daných neterminálů gramatiky na levé straně pravidel. Základní chování je zavolání metody `visitChildren`. Tato třída byla poděděna

[5] ANTLR – Another Tool for Language Recognition, https://www.antlr.org/

[6] zásuvný modul je dostupný z https://plugins.jetbrains.com/plugin/7358-antlr-v4

třídou `PlantUMLDiagramTypeVisitor`, ve které bylo přepsáno chování při navštívení tří hlavních kořenových (hned po počátečním) neterminálů, které určují, o který diagram půjde. Změna spočívá v tom, že při návštěvě proběhne vytvoření správných `DiagramVisitor` a `DiagramCreator` tříd (např. PlantUMLUseCaseDiagramVisitor a UseCaseDiagramCreator), které budou použity následně, a dále již nebude strom procházen, jak by to bylo ve výchozí implementaci způsobené zmiňovanou metodou `visitChildren`.

Na nově vytvořené `Visitor` instanci je poté opět provedeno navštívení syntaktického stromu. V této třídě jsou již přepsány visit metody pro zpracování jednotlivých elementů (např. visitClassRule nebo visitAddResourceToClassRule). Nutno však říci, že autor nevyužil plný potenciál těchto visit metod, neboť si implementaci mohl zjednodušit přepisováním i jiných pravidel, jejichž návratovou hodnotu by nemusel upravovat vícekrát v kódu. Visit metody obecně mají v parametru předaný kontext pravidla, který udává informace o aktuálním uzlu stromu a byly využity pouze pro zavolání `process` metod na správné `Processor` třídě (např. ClassProcessor.processClass) anebo případné ošetření nastalých chyb v průběhu zpracování. Kontext pravidla je nastaven ve třídě, která je poděděná z abstraktní třídy `ParserRuleContext`. Obsah kontextu se odvíjí od gramatiky, respektive přímo od zpracovávaného pravidla.

Postup implementace použitý v process metodách, které zpracovávají různé elementy zapsané v textové podobě, se samozřejmě odvíjí dle specifikace jazyka PlantUML (viz 3.1), tedy jejich implementace jsou od sebe často dost odlišné. Co je však vždy společné je, že jsou v těchto metodách pomocí kontextu uzlu procházeny a zpracovávány i podstromy tohoto uzlu. K těmto podstromům lze přistoupit pomocí metody `getChild(index)`. To jsou opět instance třídy `ParseTree`, které jde buď dále procházet, nebo z nich získat textovou hodnotu pomocí metody `getText`. Takto jsou postupně procházeny všichni potomci (podstromy) z výchozího kontextu uzlu, ze kterých jsou získávány potřebné vlastnosti pro vytvoření elementu, který bude přidán do vnitřní struktury. Ukázka kódu process metody pro element poznámka (note) se nachází na následujícím výpisu 6.1. PlantUML definuje několik typů poznámek, které jsou poté zpracovávány ve volaných metodách ve výpisu.

```
public void processNote(ParserRuleContext ctx, List<Element> elements,
        String currentPackageName, String diagramType) {

  NoteElement newNote = new NoteElement();
  int i = 1;
  String firstChild = ctx.getChild(i++).getText();

  if (firstChild.startsWith("\"") && firstChild.endsWith("\"")) { //note "description" as Alias
    processBasicNote(ctx, elements, currentPackageName, newNote, i, firstChild);
  } else {
    if (ctx.getChild(2).getText().equals("of")) {
      // note direction of ClassNameOrAlias color? : (: description) | multiple line
      processNoteWithDirectionAndObject(ctx, elements, currentPackageName, newNote, diagramType);
    } else if (firstChild.startsWith("as")) {
      // note as alias ... end note
      processNoteWithEnd(ctx, elements, currentPackageName, newNote, i, firstChild);
    } else {
      // note direction color? (: description) | multiple line
      processNoteWithDirectionOnly(ctx, elements, currentPackageName, newNote, diagramType);
    }
  }
}
```

Výpis 6.1: Ukázka metody, která vytváří element pro poznámku

Postup zpracování a vytvoření elementu pro třídu

Jelikož není tento přístup snadno uchopitelný, uveďme pro příklad zpracování třídy z diagramu tříd, jejíž process metoda je jedna z komplikovanějších ma zpracování. Jak již bylo naznačeno, postupným procházením potomků kontextu `ClassParserRule` vytvoříme `ClassElement`. Prvním potomkem bude u tohoto typu elementu typ třídy, tedy zda se jedná o klasickou třídu, rozhraní nebo třeba abstraktní třídu. Druhým a třetím potomkem může být jméno třídy nebo alias v libovolném pořadí. Co je třída a co alias, pokud jsou nastaveny obě vlastnosti, se určí dle toho, který z nich je uvozen uvozovkami. Následně se využitím pomocné třídy `ClassDiagramElementFinder` projde již vytvořená struktura (rekurzivně volaná na balíky), aby se zjistilo, zda se v ní tento element již nenachází. Pokud ano, jsou vlastnosti tohoto elementu následně přepisovány nově zjištěnými. Takto je to ve specifikaci uděláno u většiny elementů, a dokonce i třeba atributy a metody třídy jsou takto přepisovány, naopak u sekvenčního diagramu k žádnému podobnému přepisování vlastností nedochází. Zároveň je povolené mít dvě třídy se stejným jménem, pokud mají rozdílné aliasy (nebo pokud jedna alias má a druhá ne). Všechny tyto funkcionality (a další podobná přepisování) byly implementovány i do zásuvného modulu.

Po vyřešení jména a aliasu třídy se další potomci aktuálního uzlu procházejí v metodě `processClassHeader`. V první zmiňované jsou všechny ostatní vlastnosti volitelné, tedy není jisté, které z nich se budou v definici dané třídy nacházet. Jsou jimi v tomto pořadí: [generický parametr], [stereotypy]*, [barva elementu], [extends ClassName], [implements InterfaceName]. Tyto vlastnosti jsou opět získávány procházením dalších potomků kontextu pomocí indexu a je kontrolovaná vstupní syntaxe, tedy například, pokud text dalšího potomka po aliasu začíná na znak '⟨' a nezačíná na '⟨⟨', jedná se o generický parametr, který bude zpracován a index zvýšen o jedna. Pokud tato podmínka platit nebude, index zvýšen nebude a postupně bude podobným způsobem zjišťováno, zda se jedná o stereotyp, barvu atd. V případě zpracování konstrukcí extends ClassName nebo implements InterfaceName jsou do struktury uloženy i tyto třídy (bez jiných vlastností, případně pouze se stereotypem `interface`), pokud se v ní již nevyskytují, a také je vytvořen a přidán do struktury patřičný vztah, který reprezentuje generalizaci nebo realizaci.

Po zpracování celé hlavičky je ještě třeba vyřešit tělo třídy, které může obsahovat atributy a metody (položky) třídy. V metodě `processClassBody` se postupně zpracovávají jednotlivé řádky těla, neboť každá položka je v PlantUML na samostatném řádku. Správný typ položky je vytvořen na základě definovaných vlastností u položky (static, abstract, field apod.), které jsou ohraničeny ve složených závorkách, jak je vidět např. na obrázku 3.2. U položek mohou být také na různých pozicích modifikátor přístupu (na začátku, mezi vlastnostmi ve složených závorkách, ...), který je získán a nastaven do položky. Položkami jsou instance tříd `OperationElement` a `AttributeElement`, které mají společného předka `ResourceElement`. Pokud nebyly definovány vlastnosti field a method, tak se typ položky určí dle přítomnosti kulatých závorek pro parametry. V případě definování obou vlastností se dle PlantUML upřednostňuje vytvoření metody (operace). Takto vytvořené elementy jsou poté přiřazeny třídě do jejího seznamu operací, případně atributů.

6.2.3 Vytvoření diagramu ve Visual Paradigm z vnitřní strukutry

Pro převod vnitřní struktury elementů na grafický diagram se využije třída `DiagramCreator`, která byla již dříve vytvořená (viz 6.2.2). Na této třídě je zavolána abstraktní metoda `createDiagram`, v jejíž implementacích probíhá celé vytváření diagramu dle typu odvozené třídy. Základ pro model diagramu (např. `IClassDiagramUIModel`) je vytvořen pomocí

třídy `DiagramManager` z OpenAPI. Diagramu je nastaveno jméno a následně jsou procházeny jednotlivé elementy vnitřní struktury. Nejprve jsou vytvářeny modely (diagram a model element, viz 4.4) pro nevztahové elementy a poté pro vztahové. Nevztahové elementy se ukládají do hashovací tabulky, přičemž klíčem je jméno (nebo alias) a hodnotou třída `ElementInfo`, ze které jsou přístupné instance diagram a model elementu. Tato struktura je poté využívána především pro snadnější vytváření vztahových elementů.

Model element vznikne pomocí třídy `IModelElementFactory` a následně jsou mu nastaveny všechny potřebné vlastnosti, které jsou uloženy v právě zpracovávaném elementu vnitřní struktury. Těmi jsou například jméno, popisek, stereotypy apod. V případě nastaveného aliasu, se přidá stereotyp $alias, aby byl při následném exportu zachován. Ten je následně ve vytvořeném diagram elementu schován, aby nebyl zobrazován. U elementu třída jsou také případně vyrobeny instance tříd `ITemplateParameter` (generický parametr), `IAttribute` a `IOperation`, které jsou přiřazeny k model elementu třídy. Těmto třídám jsou opět nastaveny potřebné vlastnosti a u operací ještě vytvářeny parametry. Podobně u jakéhokoliv elementu s potomky (balík, skupina zpráv apod.) jsou samozřejmě volány metody pro vytvoření potomků a otcovskému elementu jsou správně přiřazeny.

Nakonec se vytvoří diagram element opět za pomoci třídy `DiagramManager`, kterému je nastaven za parametr diagram, do kterého má být přidán a vytvořený model element[7]. Tomu jsou již pouze upraveny některé grafické vlastnosti, například barva.

V následujícím odstavci bude popsáno vytvoření asociace, což je jeden z komplikovanějších vztahových elementů. Pokud se při procházení vnitřní struktury zjistí, že se jedná o vztahový element, je nejprve rozhodnuto, zda byla jeho šipka v PlantUML souboru tečkovaná nebo čárkovaná, což pomůže přiblížit typ vztahu. Následně se dle tvaru obou konců šipky určí přesný typ, o který se jedná a zavolá se metoda, zodpovědná za vytvoření tohoto typu vztahu, v našem případě `createAssociationRelation`. Uvnitř ní jsou získány dříve zmiňované instance `ElementInfo` dle jmen aktérů vztahu. Poté je vytvořena třída `IAssociation`, které je nastaveno jméno a model elementy obou účastníků vztahu (získané ze třídy ElementInfo). V dalším kroku jsou vytvořeny a přiřazeny konce vztahu (třída `IAssociationEnd`), ve kterých je nastavena multiplicita aktéra, typ agregace apod. Nakonec je vytvořen diagram element (označovaný také jako UIModel) pro vztah asociace, kterému je předán model element vztahu a diagram elementy aktérů. Následně je `ElementInfo` vztahu přidáno do kolekce vytvořených asociací pro snadnější vytváření asociačních tříd.

Obecně u vztahových elementů v PlantUML je možné definovat jakýkoliv tvar šipek (například i *--+), ovšem ve Visual Paradigm to možné není, proto v těchto kolizních případech relace nevzniknou a uživatel je o tom zpraven. Podobně je v PlantUML možné definovat všechny typy relací mezi všemi typy elementů, což dovolí například vytvořit relaci kompozice mezi aktérem a případem užití, což samozřejmě ve Visual Paradigm není možné. Toto chování je ošetřeno stejným způsobem jako předešlý problém.

6.2.4 Dokončení a grafické rozložení diagramu

Po vytvoření všech diagram elementů se zkontroluje, že definovaný diagram neobsahoval žádné chyby a na konec procesu je diagram dle jeho typu graficky rozložen. U sekvenčního diagramu se využije automatického rozložení diagram elementů a následně jejich popisků. U aktérů komunikace v tomto diagramu je ještě mírně upravena y-ová souřadnice, neboť automatické rozložení je v tomto případě nevhodné. U diagramů tříd a případů užití je pro lepší výsledek využito specifického typu rozložení, kterým bylo zvoleno tzv. hierarchické

[7] např. (IClassUIModel) diagramManager.createDiagramElement(classDiagramName, iClass))

rozložení, kterému jsou upraveny některé atributy vykreslování. U diagramu případů užití je navíc nastaven směr (orientace), který lze v PlantUML přizpůsobit. Nutno dodat, že použitím těchto postupů se výsledné rozložení diagramů více či méně odklání od toho, které vytváří nástroj PlantUML. Nakonec je vybrané rozložení aplikováno a hotový diagram je zobrazen uživateli.

6.3 Gramatika jazyka PlantUML

V této části bude ještě stručně probrána vytvořená gramatika pro jazyk PlantUML, která se také nachází mezi zdrojovými soubory této práce[8]. Tato gramatika byla tvořena pomocí nástroje ANTLR4, bylo tedy potřeba nastudovat, jakým způsobem se pomocí tohoto nástroje gramatika vytváří a také jak ji následně zpracovávat, až bude využita pro analýzu definovaných diagramů. Počáteční neterminál gramatiky se nazývá `uml`. Jeho přepisovací pravidlo zkontroluje korektní strukturu vstupního souboru dle specifikace PlantUML, uvnitř které se nachází syntaxe jednoho z diagramů (pokud je daný diagram podporován). Počáteční pravidlo lze vidět ve výpisu 6.2.

```
uml:
  NEWLINE*
  '@startuml' NEWLINE+
  (
      sequenceDiagram
    | classDiagram
    | useCaseDiagram
  )
  '@enduml' NEWLINE*
  EOF;
```

Výpis 6.2: Počáteční pravidlo gramatiky PlantUML

Pravidla pro neterminály znázorňující diagramy se skládají z N řádků diagramu, které jsou odděleny odřádkováním (nebo i více řádky). Každý jednotlivý řádek (nepočítaje prázdné řádky) obsahuje pravidlo, které definuje jeden element (pozn. nebo více například v případě vztahu, jehož aktéři nebyli definováni). Výčet pravidel jednotlivých diagramů je vidět na výpisu 6.4 a konkrétní pravidla pro některé elementy sekvenčího diagramu na výpisu 6.3.

```
messageRule:
  (identificator basicAlias?)?
  (ARROW_SEQUENCE_DIAGRAM | ARROW_DASHED)
  (identificator basicAlias?)?
  (COLON (identificator_with_space_and_special_chars | QUOTED_IDENTIFICATOR))? ;

groupRule:
  ((GROUP_TYPE_KEYWORD) identificator_with_space_and_special_chars? groupCase)
  elseGroupBranch* END_KEYWORD identificator_with_space? ;

createRule :
  CREATE_KEYWORD
  (PARTICIPANT_TYPE_KEYWORD | ENTITY_KEYWORD | ACTOR_KEYWORD)?
  identificator basicAlias? stereotypes* order? HEX_COLOR? NEWLINE messageRule ;
```

Výpis 6.3: Definice přepisovacích pravidel pro elementy sekvenčního diagramu: Běžná zpráva, Skupina a Zpráva, která vytváří aktéra komunikace

[8] cesta k souboru: /src/antlr/plantumlgrammar/PlantUML.g4

```
classDiagramLine:
  classRule |
  addResourceToClassRule |
  classPackageRule |
  classDiagramRelationRule |
  noteRule
;

useCaseDiagramLine :
  actorRule |
  useCaseRule |
  useCasePackageRule |
  useCaseDiagramRelationRule |
  noteRule
;

sequenceDiagramLine :
  participantRule |
  messageRule |
  groupRule |
  refRule |
  createRule |
  returnRule |
  activateRule |
  deactivateRule |
  sequenceNoteRule
;
```

Výpis 6.4: Pravidla jednotlivých diagramů v PlantUML gramatice

Kapitola 7

Testování a zhodnocení

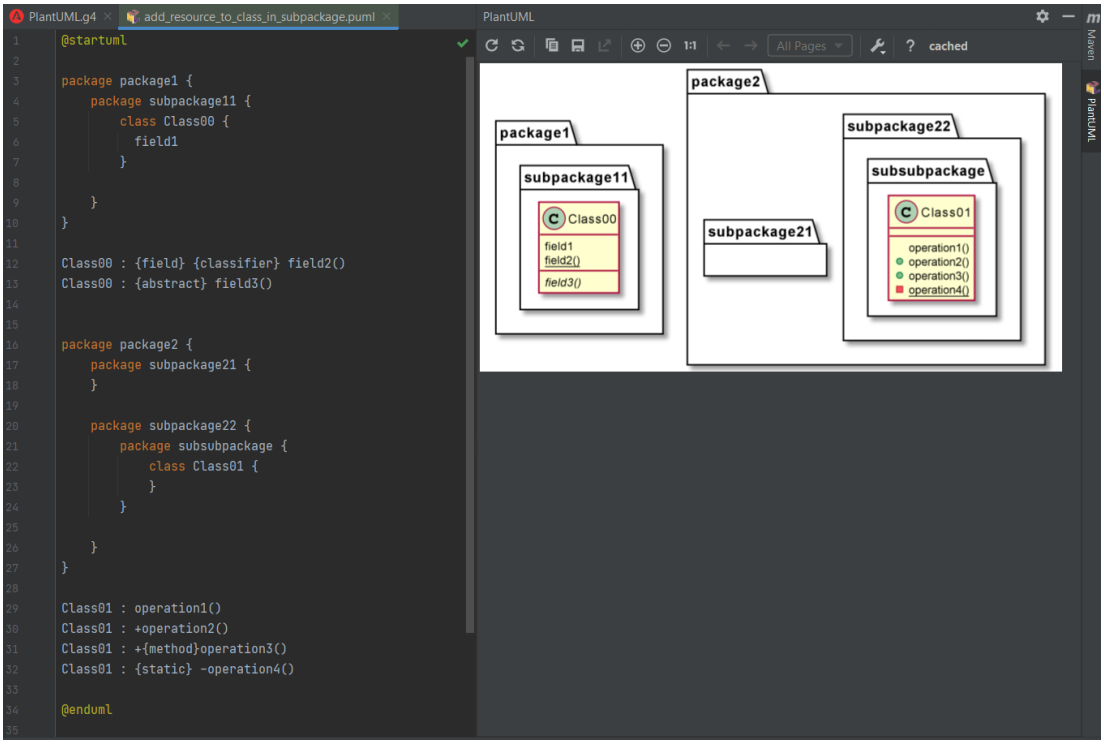
V sedmé kapitole bude nastíněn způsob testování zásuvného modulu a následně provedeno zhodnocení dosažené práce, v rámci kterého jsou také popsány nedostatky zásuvného modulu a vytvořené gramatiky.

7.1 Testování zásuvného modulu

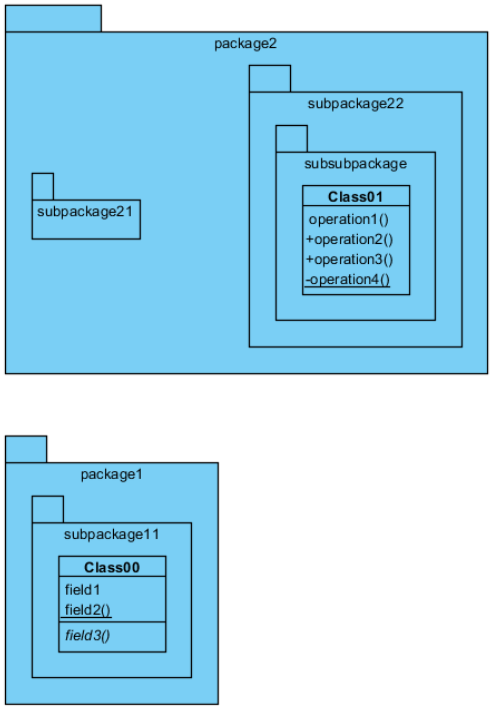
Testování implementovaného zásuvného modul probíhalo po celou dobu fáze implementace a následně také po jejím ukončení. Pro testování jednotlivých podporovaných diagramů si autor vytvořil kolem osmdesáti souborů ve formátu PlantUML s různě definovanými diagramy, které pokrývaly veškeré podporované elementy a konstrukce. Tyto soubory byly využity pro testování obou operací (export a import). Operace import byla částečně testována automaticky, celkově pomocí 69 automatických testů, které většinou testovaly větší množství vlastností většiny elementů v definovaných diagramech (jednalo se o testy s mnoha asserty). Všechny tyto automatické testy sloužily k otestování gramatiky, tedy převodu textové PlantUML reprezentace do vnitřní reprezentace elementů.

Program Visual Paradigm bohužel neposkytuje prostředí, které by bylo možné použít pro automatické testování, proto nebylo možné napsat automatické testy i na pokrytí vytváření diagramů v tomto programu (z textové i vnitřní reprezentace). Současně nebylo ze stejného důvodu možné automaticky testovat export. Tyto nedostatky se autor snažil odstranit dostatečným testováním těchto funkcionalit „ručně“, tedy exportováním velkého množství různých diagramů a porovnáváním s výsledkem nástroje PlantUML, který byl integrován v IntelliJ IDEA. Stejným způsobem byl testován i převod z textové reprezentace přímo do Visual Paradigm (kompletní operace import). K „ručnímu“ testování obou operací byly částečně použity i vytvořené soubory PlantUML diagramů, které byly zmíněny v předchozím odstavci.

Využití integrovaného nástroje PlantUML pro testování se nachází na obrázku 7.1. Výsledný grafický diagram, získaný pomocí operace import, jež byla aplikována na soubor z tohoto obrázku je k vidění na 7.2. Díky testování bylo dle očekávání nalezeno a opraveno velké množství chyb různého charakteru a různé závažnosti.



Obrázek 7.1: Využití integrovaného nástroje PlantUML v prostředí IntelliJ IDEA na jednom z vytvořených testovacích souborů



Obrázek 7.2: Výsledný diagram získaný operací import (diagram z obrázku 7.1)

7.2 Vyhodnocení vývoje

Zásuvný modul byl navrhnut a implementován tak, aby byl snadno rozšířitelný jak o další textové formáty, tak i o podporu dalších typů UML diagramů. To bylo splněno a při případném implementačním navázání na tuto práci bude spousta tříd, metod a již vytvořených algoritmů znovupoužitelných. Projekt je logicky řazen do balíků, jak bylo možno vidět na diagramu 5.4. Zásuvným modulem jsou podporovány tři nejhlavnější typy UML diagramů (pro export i import operace), pro které byla vytvořena ANTLR4 gramatika. U diagramu tříd jsou podporovány následující elementy: třída, balík, model, asociační třída, asociace, různé typy jiných relací (generalizace, realizace, ...), poznámka a také přidávání dalších atributů a metod k elementu třída (u exportu, jelikož se jedná o část specifikace PlantUML). Ze sekvenčního diagramu jsou podporovány různí aktéři zasílání zpráv, automaticky číslované zprávy (různé typy, včetně zpráv vytvářejících aktéry) a skupiny zpráv s různými typy operandů. U diagramu případů užití to jsou následující typy elementů: případ užití, aktér, balík, systém, různé typy relací a poznámka. Z dalších vlastností je podporováno například pořadí aktérů (klíčové slovo `order` v PlantUML) v sekvenčním diagramu. Obecně u všech elementů, kde to specifikace PlantUML dovoluje, je zaznamenávána barva (v základní hexadecimální podobě). U gramatiky jsou podporovány komentáře a také další části PlantUML specifikace, které však nejsou zpracovávány. To proto aby analýza pokud možno neselhala kvůli chybějící implementaci funkcionality. Jedná se například o pravidla pro reference, (de)aktivaci čar života, skinparams apod.

Co nebylo implementováno

V rámci implementace zásuvného modulu se autor snažil podporovat co nejširší část elementů a vlastností, které jsou v programu Visual Paradigm pro vytváření diagramů využívány a zároveň také co nejširší část PlantUML specifikace u podporovaných typů diagramů. Z této specifikace bylo vynecháno několik méně důležitý funkcionalit a to z různých důvodů. Konkrétně se jedná většinou o grafické úpravy elementů (např. rozdělení řádků čárou v případu užití, šipky se směry – například -left->, veškeré skinparams parametry aj.). Vynechána byla také podpora (de)aktivací životních čar, poznámek a referencí v sekvenčním diagramu. V diagramu tříd není možné poznámky směřovat na atributy a metody, což je dle specifikace PlantUML možné. V rámci tohoto typu diagramu také není podporováno skrývání (hide) a odstraňování (remove) různých tříd. Také elementy namespace, což jsou pokročilejší typy balíků, nejsou vhodně podporované. Ze vztahových elementů není podporována relace s názvem Containment, jelikož nemá model element, což zapříčinilo nemožnost získání aktérů vztahu u operace export, a rovněž také rekurzivní zpráva. Naopak zpráva v rámci jednoho aktéra, stejně jako nalezené a ztracené zprávy, podporované jsou. Méně používané elementy ve Visual Paradigm, jako je třeba Collaboration nebo Rest Service také nejsou implementovány. Známým problémem u operace import je také neschopnost správně vykreslit vnořenou skupinu uvnitř jiné. Dalším problémem by mohlo být již zmiňované jiné rozložení importovaných elementů, než je v nástroji PlantUML. Také vytvořená PlantUML gramatika rozhodně nepodporuje všechny možné (především grafické) prvky, které specifikace tohoto jazyka nabízí. Byla však tvořena tak, aby pokrývala většinu možné (negrafické) syntaxe.

Další možnosti rozšíření zásuvného modulu

Implementovaný zásuvný modul by bylo vhodné vylepšit a zdokonalit v několika směrech. Tím hlavním by mohla být podpora dalších nejpoužívanějších typů UML diagramů pro jazyk PlantUML. To by zahrnovalo mimo jiné rozšíření vytvořené ANTLR gramatiky. V případě potřeby je implementace kvalitně připravena na tuto možnost, stejně tak na rozšíření o další textové formáty (např. ty, které byly popsané v kapitole 2.3).

Další možnost rozšíření by spočívala ve vylepšení podpory pro již implementované diagramy, neboť některé méně významné elementy a jejich vlastnosti, které se mohou v diagramech vyskytovat, nebyly brány v potaz.

Také by bylo vhodné provést zkvalitnění vytvořené PlantUML gramatiky, neboť například u některých speciálních názvů identifikátorů může špatně rozpoznávat analyzovanou větu a tím pádem ji nesprávně vyhodnotit, případně zahlásit syntaktickou chybu.

Kapitola 8

Závěr

Cílem této práce bylo seznámit se s různými způsoby modelování UML diagramů a tvorbou zásuvných modulů do nástroje Visual Paradigm a jeden takový zásuvný modul vyvinout, jehož hlavním cílem bude schopnost převádět (import/export) UML diagramy mezi programem Visual Paradigm a textovými formáty.

Nejprve byl pečlivě prostudován způsob, jak je možné rozšířit funkcionalitu Visual Paradigm pomocí tvorby zásuvných modulů. Toho lze dosáhnout za pomoci otevřeného rozhraní, které vývojáři nástroje Visual Paradigm poskytují v rámci instalace. Následně se autor zaměřil na způsoby modelování UML diagramů, kterými jsou nástroje podporující textové formáty a také grafické nástroje. Speciální pozornost je věnována nástroji PlantUML, který je nejvyužívanější z nástrojů pro textové formáty UML, přičemž diagramy v něm jsou vytvářeny pomocí stejnojmenného jazyka.

Následně byl navržen zmiňovaný zásuvný modul pro převod UML diagramů mezi Visual Paradigm a různými textovými formáty, který byl implementován s podporou tří hlavních UML diagramů (diagram tříd, případů užití a sekvenční), přičemž jediný podporovaný textový formát je PlantUML. Zásuvným modulem nejsou podporovány všechny UML diagramy také z toho důvodu, že práce byla ztížena tím, že k jazyku PlantUML neexistuje dostupná gramatika, musela tedy být tvořena od základu. Návrh si však kladl za cíl snadnou rozšířitelnost o další diagramy i další textové formáty.

Implementovaný zásuvný modul s názvem Text Format Diagrams byl dostatečně otestován „ručně“ a částečně také pomocí automatických testů. V rámci hodnocení výsledků bylo mimo jiné nastíněno, jaké funkcionality nebyly implementovány. Výsledný projekt, zveřejněný jako open-source se nachází na gitlabu[1]. Příspěvek o implementovaném zásuvném modulu byl uveřejněn na oficiálním Visual Paradigm fóru[2], aby byla zjištěna zpětná vazba.

V návaznosti na tuto práci by bylo vhodné rozšířit podporu implementace o další využívané typy UML diagramů (např. diagram aktivit a diagram komunikace), případně také o další textové formáty, které byly v rámci práce popsány.

[1] https://gitlab.com/ondry96/TextFormatDiagrams
[2] https://forums.visual-paradigm.com/c/plug-in-open-api

Literatura

[1] ABSTRATT. *TextUML Toolkit*. [Online; navštíveno 19.12.2020]. Dostupné z: https://abstratt.github.io/textuml/index.html.

[2] CABOT, J. *Text to UML tools – Fastest way to create your models*. [Online; navštíveno 18.12.2020]. Dostupné z: https://modeling-languages.com/text-uml-tools-complete-list/.

[3] CORONEL, C. a MORRIS, S. *Database Systems: Design, Implementation, and Management*. Cengage Learning, 2014. ISBN 978-1-337-62790-0.

[4] DLOUHÝ, I. I. *Nástroj pro kontrolu správnosti návrhových diagramů v UML*. Brno, 2014. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

[5] ECLIPSEMARKETPLACE. *TextUML Toolkit*. [Online; navštíveno 19.12.2020]. Dostupné z: https://marketplace.eclipse.org/content/textuml-toolkit.

[6] FOWLER, M. *UML Distilled*. Addison-Wesley, third edition, 2004. ISBN 978-0-321-19368-1.

[7] KANISOVÁ, H. a MÜLLER, M. *UML srozumitelně*. Computer Press, a.s, druhé aktualizované vydání, 2007. ISBN 80-251-1083-4.

[8] MILES, R. a HAMILTON, K. *Learning UML 2.0*. O'Reily Media, Inc., 2006. ISBN 978-0-596-00982-3.

[9] PARR, T., HARWELL, S. a FISHER, K. *Adaptive LL(*) Parsing: The Power of Dynamic Analysis*. [Online; navštíveno 10.5.2021]. Dostupné z: https://www.antlr.org/papers/allstar-techreport.pdf.

[10] ČÁPKA, D. *Úvod do UML*. [Online; navštíveno 30.12.2020]. Dostupné z: https://www.itnetwork.cz/navrh/uml/uml-uvod-historie-vyznam-a-diagramy.

[11] PLANTUML. *Class Diagram*. [Online; navštíveno 3.1.2021]. Dostupné z: https://plantuml.com/class-diagram.

[12] PLANTUML. *PlantUML Language Reference Guide*. [Online; navštíveno 3.1.2021]. Dostupné z: http://plantuml.com/guide.

[13] PLANTUML. *Sequence Diagram*. [Online; navštíveno 9.1.2021]. Dostupné z: https://plantuml.com/sequence-diagram.

[14] PlantUML. *Use Case Diagram.* [Online; navštíveno 9.1.2021]. Dostupné z: https://plantuml.com/use-case-diagram.

[15] pp_pankaj. *Computer Aided Software Engineering (CASE).* [Online; navštíveno 16.12.2020]. Dostupné z: https://www.geeksforgeeks.org/computer-aided-software-engineering-case/.

[16] PredictiveAnalyticsToday. *Visual Paradigm Review.* [Online; navštíveno 21.12.2020]. Dostupné z: https://www.predictiveanalyticstoday.com/visual-paradigm/.

[17] Spinellis, D. *Automated Drawing of UML Diagrams.* [Online; navštíveno 19.12.2020]. Dostupné z: https://www.spinellis.gr/umlgraph/.

[18] Spinellis, D. *Class Diagram Example: Advanced Relationships.* [Online; navštíveno 19.12.2020]. Dostupné z: https://www.spinellis.gr/umlgraph/doc/ceg-adv.html.

[19] Sterne, J. *Plug-in software.* [Online; navštíveno 30.12.2020]. Dostupné z: https://www.britannica.com/technology/plug-in.

[20] TechnologyEvaluation. *Visual Paradigm.* [Online; navštíveno 21.12.2020]. Dostupné z: https://www3.technologyevaluation.com/sd/solutions/visual-paradigm-46207.

[21] Tišňovský, P. *Nástroje pro tvorbu UML diagramů z příkazové řádky.* [Online; navštíveno 1.1.2021]. Dostupné z: https://www.root.cz/clanky/nastroje-pro-tvorbu-uml-diagramu-z-prikazove-radky/.

[22] VisualParadigm. *[Guide] Getting started with plugin development.* [Online; navštíveno 23.4.2021]. Dostupné z: https://forums.visual-paradigm.com/t/guide-getting-started-with-plugin-development/13915/1.

[23] VisualParadigm. *How to Develop Visual Paradigm Plug-in?* [Online; navštíveno 20.12.2020]. Dostupné z: https://www.visual-paradigm.com/tutorials/plugin.jsp.

[24] VisualParadigm. *Implementing plugin.* [Online; navštíveno 25.12.2020]. Dostupné z: https://www.visual-paradigm.com/support/documents/vpuserguide/124/254/7040_implementing.html.

[25] VisualParadigm. *Introduction to plugin support.* [Online; navštíveno 22.12.2020]. Dostupné z: https://www.visual-paradigm.com/support/documents/vpuserguide/124/254/7039_introduction.html.

[26] VisualParadigm. *Visual Paradigm User's Guide.* [Online; navštíveno 20.12.2020]. Dostupné z: https://www.visual-paradigm.com/support/documents/vpuserguide.jsp.

Příloha A

Obsah paměťového média

Na přiloženém CD se nacházejí následující soubory a adresáře:

- `src` – zdrojové soubory implementovaného zásuvného modulu
- `TextFormatDiagrams.zip` – archiv se zásuvným modul k instalaci do Visual Paradigm
- `xondra49-UMLDiagramsConversion.pdf` – technická zpráva ve formátu PDF
- `README_en` – readme soubor v angličtině
- `README_cz` – readme soubor v češtině
- `own-tests` – adresář s vlastními testy
- `doc-codes` – adresář se zdrojovými kódy technické zprávy
- `examples` – adresář s ukázkami importu a exportu na různých diagramech

Příloha B

Výpis konfiguračního souboru

```
<plugin
  id="TextFormatDiagrams"
  name="TextFormatDiagrams"
  description="TextFormatDiagrams"
  provider="TextFormatDiagrams"
  class="pluginmain.TextFormatDiagrams">
<actionSets>
<actionSet id="actionset">
<action
  id="TextFormatImport"
  actionType="generalAction"
  label="Text format..."
  tooltip="Text format..."
  style="normal"
  icon="icons/icon_bigger.png"
  ribbonPath="Project/Import/XML">
<actionController
  class="pluginmain.actions.ImportFromTextFormatController"/>
</action>
<action
  id="TextFormatExportActiveDiagram"
  actionType="generalAction"
  label="Active Diagram as Text format..."
  tooltip="Active Diagram as Text format..."
  style="normal"
  icon="icons/icon_bigger.png"
  ribbonPath="Project/Export/XML">
<actionController
  class="pluginmain.actions.ExportActiveDiagramToTextFormatController"/>
</action>
<action
  id="TextFormatExport"
  actionType="generalAction"
  label="Text format..."
```

```
  tooltip="Text format..."
  style="normal"
  icon="icons/icon_bigger.png"
  ribbonPath="Project/Export/XML">
<actionController
  class="pluginmain.actions.ExportToTextFormatController"/>
</action>
</actionSet>
<contextSensitiveActionSet id="actionset">
<contextTypes all="false">
  <include type="ClassDiagram"/>
  <include type="InteractionDiagram"/>
  <include type="UseCaseDiagram"/>
</contextTypes>
<action
  id="TextFormatExportContext"
  label="Export as Text format"
  icon="icons/icon_smaller.png"
  style="normal"
  menuPath="Export">
<actionController
  class="pluginmain.actions.ExportDiagramFromToolbarController"/>
</action>
</contextSensitiveActionSet>
</actionSets>
</plugin>
```

Výpis B.1: Konfigurační soubor implementovaného zásuvného modulu – plugin.xml