FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Vyhledávání v konceptuálních modelech -- UML diagramy Package a Activity
Student:	František Štěpánek
Vedoucí:	doc. Ing. Robert Pergl, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Zadání je součástí projektu Repocribro skupiny CCMi, ve kterém vzniká systém pro vyhledávání v konceptuálních modelech. V této práci je cílem vytvořit moduly parseru pro UML Package Diagram a Activity Diagram ve formátu Enterprise Architect XMI.

1. Seznamte se projektem indexování a vyhledáváni v konceptuálních modelech Repocribro, formátem OMG XMI a jeho implementací v Enterprise Architect (Package Diagram a Activity Diagram) a databází Neo4j.
2. Implementujte moduly parseru.
3. Ve spolupráci s dalšími členy týmu integrujte vytvořené moduly do systému Repocribro.
4. Řešení zdokumentujte, otestujte a demonstrujte na ukázkové studii.

–

https://github.com/MarekSuchanek/repocribro
https://en.wikipedia.org/wiki/XML_Metadata_Interchange
https://neo4j.com/Arlow, J., Neustadt, I., 2005. UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition). Addison-Wesley Professional.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 23. ledna 2021 v Praze.

Bakalářská práce

VYHLEDÁVÁNÍ V KONCEPTUÁLNÍCH MODELECH – UML DIAGRAMY PACKAGE A ACTIVITY

František Štěpánek

Fakulta informačních technologií ČVUT v Praze
Katedra softwarového inženýrství
Vedoucí: doc. Ing. Robert Pergl, Ph.D.
12. května 2021

Odkaz na tuto práci: František Štěpánek. *Vyhledávání v konceptuálních modelech – UML diagramy Package a Activity*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat především vedoucímu mé práce panu doc. Ing. Robertovi Perglovi, Ph.D., který mi pomohl s volbou tématu a v průběhu práce mi vždy dobře poradil a nasměroval mě správným směrem. Dále bych rád poděkoval panu Bc. Richardovi Husárovi, který mi asistoval při integraci kódu do stávajícího projektu.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 12. května 2021 .....

Abstrakt

Hlavním cílem práce je implementovat moduly parseru pro UML diagramy balíčků a aktivit. Moduly musí být implementovány tak, aby byly integrovatelné do projektu na indexování a vyhledávání v konceptuálních modelech, který je součástí aplikace Repocribro. Součástí práce je také seznámení s UML diagramy a s elementy, které se v nich vyskytují. Výsledkem práce je rozšíření stávajícího projektu o moduly parseru vybraných diagramů. Funkčnost implementovaných modulů parseru je demonstrována na případové studii.

Klíčová slova UML, konceptuální modelování, parsování XMI souborů, diagram balíčků, diagram aktivit, vyhledávání v konceptuálních modelech, Repocribro

Abstract

The main aim of this work is to implement parser modules for the UML Activity diagram and the UML Package diagram. Modules must be integrable into the project for indexing and searching in conceptual models, which is part of the application Repocribro. Part of this work is to introduce UML Activity diagram, UML Package diagram and chosen elements which are presented therein diagrams. The result of this work is an extension of the current project via adding parser modules for chosen diagrams. The functionality of the parser modules is demonstrated in the case study.

Keywords UML, conceptual modeling, parsing XMI files, package diagram, activity diagram, searching in conceptual models, Repocribro

Seznam zkratek

CASE	Computer Aided Software Engineering
CCMi	Centre for Conceptual Modelling and Implementation
FIFO	First in, First out
ISO	International Organization for Standardization
LIFO	Last in, First out
MOF	Meta-Object Facility
Neo4j	Network Exploration and Optimization 4 Java
OMG	Object Management Group
SQL	Structured Query Language
UML	Unified Modeling Language
UUID	Universally Unique Identifier
XMI	XML Metadata Interchange
XML	Extensible Markup Langugage

Kapitola 1
Úvod

Konceptuální modelování je důležitou disciplínou software inženýrství. Umožňuje nám zachytit strukturu modelovaného systému z požadované perspektivy, která nám zjednodušuje pochopení daného systému. Ve skupině CCMi (Centrum pro konceptuální modelovaní a implementaci) v aplikaci Repocribro vzniká projekt, který se zabývá indexováním a vyhledáváním v konceptuálních modelech. Repocribro seskupuje konceptuální modely, které jsou od různých autorů, ale zabývají se podobnou problematikou. Vyhledávání v konceptuálních modelech je užitečné ve dvou rovinách užití. První je možnost hledání podobností nebo naopak odlišností mezi konceptuálními modely, které se zabývají podobnou problematikou. Druhou možností je vyhledávání zajímavých částí jednoho konkrétního modelu. Konceptuální modely jsou graficky reprezentovány pomocí diagramů. Stávající projekt na indexování a vyhledávání v konceptuálních modelech dokáže pracovat pouze s UML diagramy tříd (Class diagrams).

Hlavním cílem práce je rozšířit projekt o moduly parseru, které dokáží zpracovat UML diagramy balíčků (Package diagrams) a aktivit (Activity diagrams) tak, aby v nich bylo možné vyhledávat. Moduly následně budeme integrovat do stávajícího projektu. Vstupem modulů parseru jsou soubory ve formátu XMI, které vzniknou vyexportováním diagramů z modelovacích nástrojů. Jedním z dílčích cílů práce je podpora více modelovacích nástrojů. Jmenovitě se jedná o modelovací nástroje Enterprise Architect, Visual Paradigm a OpenPonk.

Struktura práce odpovídá úkolům, které je nutné vykonat pro splnění cílů práce. Začíná kapitolou Teorie a základní pojmy, kde se nachází shrnutí a vysvětlení pojmů, se kterými pracujeme ve zbytku práce. Následuje kapitola Analýza a návrh, která se zaměřuje na prozkoumání problému a navržení řešení za pomoci zvolených metod a technologií. Tato kapitola obsahuje také odůvodnění, proč byla učiněna daná rozhodnutí. Práce pokračuje kapitolou Implementace, kde je podrobněji popsáno, jak jsou jednotlivé části implementovány. Na implementaci navazuje kapitola Testování, která dokumentuje formu a počet testů, které bylo potřeba implementovat pro ověření požadované funkčnosti. Práci uzavírá kapitola Případová studie, ve které jsou demonstrovány výsledky práce na konkrétních příkladech diagramu balíčků a diagramu aktivit.

Teoretické pojmy, které vychází ze standardu UML, jsou v souladu s verzí UML 2.5.1., části, které se týkají XMI souborů, vychází z verze 2.5.1. standardu XMI. Verze modelovacích nástrojů, které jsou podporovány moduly pareseru jsou následující: Enterprise Architect verze 14, Visual Paradigm verze 16.2 a OpenPonk verze 2.1. Implementační část je napsána v programovacím jazyce Python verze 3.6. Není zaručena kompatibilita s jinými verzemi, než je zde uvedeno.

Kapitola 2

Teorie a základní pojmy

2.1 UML

Unified Modeling Language (unifikovaný modelovací jazyk) je v software inženýrství grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci software systémů. UML nabízí standardní způsob zápisu jak návrhů systému včetně konceptuálních prvků, jako jsou business procesy, tak konkrétních prvků, jako jsou příkazy programovacího jazyka, databázová schémata a znovupoužitelné programové komponenty. UML podporuje objektově orientovaný přístup k analýze, návrhu a popisu programových systémů. Neobsahuje způsob, jak se má používat, ani neobsahuje metodiku, jak analyzovat, specifikovat či navrhovat programové systémy. UML původně vzniklo za účelem standardizování notací a přístupů k software designů. Bylo vyvinuto v Rational Software v letech 1994–1996. V roce 1997 bylo UML přijato jako standard skupinou OMG (Object Management Group) a od té doby je spravované touto skupinou. V roce 2005 ISO (mezinárodní organizace pro standardizace) vydala UML jako ověřený ISO standard. UML se během let velmi měnilo, nejnovější verze 2.5.1 je z roku 2017. V této verzi byla specifikace přepsána ze své dřívější verze, aby byla lépe čitelná odebráním nadbytečných částí a zlepšení srozumitelnosti textu. [1]

2.2 XMI

XMI (XML Metadata Interchange) je standard definovaný OMG pro výměnu metadat informací pomocí jazyka XML. XMI může být použito pro libovolná metadata, jejichž metamodel je možné vyjádřit v MOF (Meta-Object Facility). Nejčastější použití XMI je výměna formatů UML modelů. OMG nahlíží na modelování ze dvou pohledů – abstraktní model a konkrétní model. Abstraktní model reprezentuje sémantické informace. Konkrétní model reprezentuje vizuální stránku diagramu. Abstraktní modely jsou instancemi libovolného modelovacího jazyka založeného na MOF, například UML nebo SysML. Pro diagramy je použit Diagram Interchange (DI, XMI[DI]) standard. V současné době není dostatečná kompatibilita mezi implementacemi vytváření a čtení XMI souborů jednotlivých modelovacích nástrojů dokonce i v takovém případě, kdy se jedná o výměnu abstraktních modelů. Diagram Interchange se z toho důvodu téměř nepoužívá. To znamená, že výměna souborů mezi UML modelovacími nástroji je jen výjimečně možná. Jedním z účelů XMI je poskytnout jednoduchou výměnu metadat mezi modelovacími nástroji založenými na UML. XMI je také často využíváno jako prostředek, který je poslán z modelovacího nástroje do nástroje na generování software. [2]

Každý XMI soubor obsahuje mechanismus pro rozšiřování. Každý element v modelu může mít definovaný libovolný počet rozšiřujících elementů, ale také nemusí mít definovaný žádný. Tyto

rozšiřující elementy mohou obsahovat úplně cokoliv, což poskytuje velkou volnost ve způsobu rozšiřování. U nástrojů, které pracují se standardem XMI, se očekává, že uchovají rozšiřující informace a vyexportují je, i když je nepravděpodobné, že tyto nástroje budou později schopné zpracovat tyto rozšiřující informace. [2]

2.3 Package diagram

Package diagram (diagram balíčků) patří mezi UML strukturální diagramy. Strukturální diagramy jsou diagramy popisující elementy, které nejsou závislé na čase. Package diagram ukazuje strukturu systému na úrovni Packages (balíčků). [3]

V Package diagramu se nejčastěji vyskytují elementy Package, Model a Profile. Pro účely této práce je rozsah omezen pouze na Package a Model. Dále Package diagram obsahuje relace mezi jednotlivými elementy.

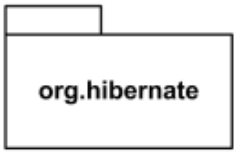
2.3.1 Elementy

Tato sekce se věnuje definováním elementů, které se vyskytují v Package diagramech a jsou obsahem této práce. Konkrétně se jedná o elementy Package a Model.

2.3.1.1 Package

Package (balíček) sdružuje své členy do namespace (jmenného prostoru). Package tedy představuje skupinu, ve které se mohou nacházet jednotlivé elementy. Tyto elementy mohou být přímo vlastněné daným Package, nebo mohou být do tohoto Package importovány. Dále je možné jednotlivé balíčky slučovat pomocí Package merge (slučování balíčků). Pomocí Packages je definována hlavní generická a organizační způsobilost UML. Existují specializace pro Models (modely) a Profiles (profily), které organizují rozšíření do UML. [3] Obsahem této práce je pouze podtřída Model.

Package je zobrazen jako obdélník s malým štítkem připnutým k levému hornímu rohu tohoto obdélníku. Jméno Package je napsáno uvnitř obdélníku. [4] Příklad Package je zobrazen na obrázku 2.1.



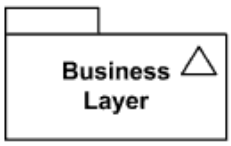
Obrázek 2.1 Package (balíček) [3]

2.3.1.2 Model

Model (model) je popis systému, kde systém je chápán v nejširším možném smyslu a může obsahovat nejen software a hardware, ale také organizaci a procesy systému. Model popisuje systém z požadovaného úhlu pohledu pro určitou skupinu zainteresovaných osob na požadované úrovni abstrakce. Model pokrývá celý systém, což znamená, že obsahuje všechny prvky. Jen některé prvky jsou však prezentovány v daném modelu. Prezentovány jsou takové prvky, které jsou relevantní pro daný účel – jsou relevantní na požadované úrovni abstrakce a zároveň relevantní

z daného úhlu pohledu. Stejně jako Package, tak i Model obsahuje skupinu členů, které dohromady popisují modelovaný systém. Systém může mít více Models. Složením všech Models vznikne komplexní pohled na systém, který jednotlivé Models popisují. Model je rozšířením Package. [3]

Model je zobrazen stejně jako Package, tedy jako obdélník s malým štítkem připnutým k levému hornímu rohu tohoto obdélníku. Rozdílem je, že uvnitř obdélníku je vedle jména Model malý trojúhelník, který se nachází v pravém horním rohu. [4] Příklad Model je zobrazen na obrázku 2.2.



Obrázek 2.2 Model (model) [3]

2.3.2 Relace

Následující sekce se zabývají relacemi, které se nejčastěji vyskytují v Package diagramech. Pro účel této práce jsme vybrali nejpoužívanější a v kontextu Package diagramu nejrelevantnější relace. Konkrétně se jedná o relace Package merge, Package import, Dependency a její podtřídu Usage.

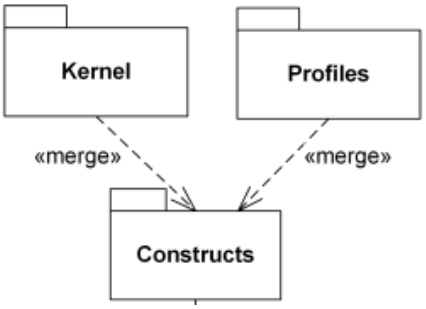
2.3.2.1 Package Merge

Package merge (sloučení balíčku) je přímá relace mezi dvěma Packages, která indikuje, že obsah cílového Package je přidán do obsahu zdrojového Package podle definovaných pravidel. Zdrojový Package konceptuálně přidává charakteristiky cílového Package do svých vlastních charakteristik, z čehož vznikne Package, který kombinuje charakteristiky obou Packages. Přijímající Package není obvykle vykreslen se sloučenými elementy ze slučovaných Packages. Z pohledu sémantiky modelu není žádný rozdíl mezi modelem s explicitním použitím relace Package merge a modelem, ve kterém všechna Package merge již byla vykonána, tedy že obsahy cílových Packages byly přidány do obsahů zdrojových Packages. Stejně je tomu u XMI souborů. XMI soubory obsahující Package merge jsou sémanticky ekvivalentní XMI souborům již rozšířeným o Package merge. Package nemůže sloučit sám sebe, a to přímo ani nepřímo. Tato schopnost je navržena pro použití, když elementy definované v jiném Package mají stejné jméno a mají reprezentovat stejný koncept v cílovém Package. Základní koncept může být sloučen pro rozdílné účely, kde každý účel je definován v separátním přijímajícím Package. Vybráním různých přijímajících Packages je možné získat různé definice konceptů pro požadované výsledky. [3]

Relace Package merge je zobrazena jako přerušovaná čára s otevřenou šipkou směřující od zdrojového Package do cílového Package. Klíčové slovo «merge» je blízko přerušované čáry. [4] Příklad Package merge je zobrazen na obrázku 2.3.

2.3.2.2 Package Import

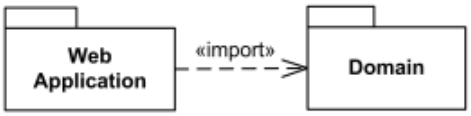
Package import (import balíčku) je přímá relace mezi dvěma Packages, která umožňuje využití nekvalifikovaných jmen, aby referovala na členy Package z jiných Packages. Importující Package přidá jména členů importovaného Package do svého obsahu. Konceptuálně je Package import ekvivalentní Element import pro každého člena importovaného Package s výhradou, kdy je separátně definován daný Element import. Package import je zobrazen pomocí otevřené šipky



Obrázek 2.3 Package merge (sloučení balíčku), upraveno z [4]

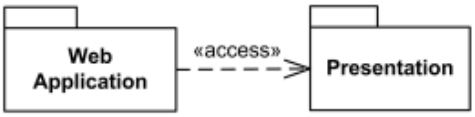
s přerušovanou čarou směřující od importujícího Package k importovanému Package. Vypadá stejně jako Dependency a Usage, ale jsou to různé přímé relace. Visibility (viditelnost) Package import může být public (veřejná) nebo private (soukromá). Pokud je Package import public, importované elementy budou přidány do Package a budou viditelné mimo tento Package (mohou být například znovu importovány jiným Package). Pokud je Package import private, importované elementy budou také přidány do Package, ale nebudou viditelné mimo tento Package. Parafrázováno z [5], podobně popsáno také v [3].

Pokud se jedná o Packege import s visibility public, tak je nad přerušovanou čarou zobrazeno klíčové slovo «import». [5] Příklad public Package import je zobrazen na obrázku 2.4.



Obrázek 2.4 Public Package import (veřejný import balíčku) [3]

Pokud se jedná o Package import s visibility private, tak je na stejném místě zobrazeno klíčové slovo «access». Výchozí hodnota visibility je public. [5] Příklad private Package import je zobrazen na obrázku 2.5.

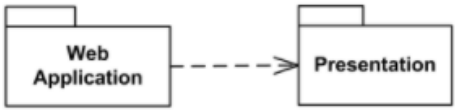


Obrázek 2.5 Private Package import (soukromý import balíčku) [3]

2.3.2.3 Dependency

Dependency (závislost) je relace, která indikuje, že client (klient) není úplný bez svého supplier (poskytovatele). Relace se nemusí vyskytovat pouze mezi dvěma Packages. Dependency může být dále specifikována konkrétními podtřídami. Jedná se o podtřídy Usage (použití), Abstraction (abstrakce) a Deployment (nasazení). [3] Obsahem této práce je pouze podtřída Usage.

Relace Dependency je zobrazena jako přerušovaná čára s otevřenou šipkou směřující od client (klienta) do supplier (poskytovatele). [4] Příklad Dependency je zobrazen na obrázku 2.6.



Obrázek 2.6 Dependency (závislost) [4]

2.3.2.4 Usage

Usage (použití) je Dependency, ve které jeden element (nebo více elementů) vyžaduje jiný element (nebo více elementů) pro svou plnou implementaci. Usage nespecifikuje, jak je daný element použit, jedná se jen o fakt, že je nezbytný pro úplnou definici nebo implementaci daného elementu. [3]

Relace Usage je zobrazena jako přerušovaná čára s otevřenou šipkou směřující od elementu, který vyžaduje element pro svou implementaci, do elementu, který je vyžadován. Nad přerušovanou čarou je zobrazeno klíčové slovo «use». [4] Příklad Usage je zobrazen na obrázku 2.7.

Obrázek 2.7 Usage (použití) [4]

2.4 Activity diagram

Activity diagram (diagram aktivit) patří mezi UML behaviorální diagramy. Behaviorální diagramy popisují dynamické chování objektů v systému včetně jejich metod, spolupráce, aktivit a stavové historie. Activity diagram popisuje průběh aktivity v čase. Aktivitu lze popsat jako soubor Actions, kdy Action je atomická část aktivity. Také lze aktivitu vyjádřit jako popsání posloupností změn v daném systému. Tato posloupnost je popsána za pomocí tokenů. Tokeny nejsou explicitně modelovány, jsou ale použity pro popis vykonání dané aktivity. Tokeny mohou být control token nebo object token. Control token ovlivňuje vykonání dané aktivity, ale neobsahuje žádná data a může se pohybovat pouze po hranách typu Control flow. Object token slouží primárně k uchování a předávání dat mezi Object nodes. Může se pohybovat po hranách Object flow, ale také v jistých případech po hranách Control flow. Tokeny jsou od sebe rozlišitelné a to i v případě, že mají stejnou hodnotu. [3]

Activity diagram zobrazuje tok s důrazem na sekvenci a podmínky tohoto toku. Nejpodstatnější částí aktivit jsou Actions (akce). Actions koordinovaných aktivit mohou být spuštěny z důvodu, že jiné Actions byly dokončeny a jsou tak dané objekty a data k dispozici, nebo z důvodu, že nastala externí událost. V Activity diagramech se vyskytují Control nodes, Object nodes a Executable nodes. Tyto uzly jsou propojeny pomocí hran Control flow a Object flow. V Activity diagramech je možné definovat ještě Activity partition (rozdělení aktivity), které slouží k uskupení elementů do skupin. [3]

2.4.1 Control node

Control node (řídící uzel) je uzel, který slouží k řízení a koordinaci tokenů mezi ostatními uzly v dané aktivitě. Control node nemůže zadržovat tokeny, slouží pouze k jejich koordinaci. [3]

Jedná se o abstraktní třídu, což znamená, že Control node se sám konkrétně nikde v diagramu nemůže vyskytnout. Slouží pouze pro definici toho, co mají jeho podtřídy společné. Konkrétní podtřídy Control node, které se přímo vyskytují v Activity diagramech, jsou následující:

- Initial node (počáteční uzel)
- Final node (koncový uzel)
- Fork node (rozdělovací uzel)
- Join node (spojovací uzel)
- Merge node (slučovací uzel)
- Decision node (rozhodovací uzel) [3]

2.4.1.1 Initial node

Initial node (počáteční uzel) je Control node, který se chová jako startovací bod pro vykonání dané aktivity. Aktivita může mít více než jeden Initial node. V takovém případě je při spuštění aktivity spuštěno více toků najednou – jeden tok z každého Initial node. Initial node by neměl mít žádnou příchozí hranu, což znamená, že Initial node vlastněný aktivitou bude vždy dostupný ve chvíli, kdy je aktivita spuštěna. Naopak může mít více odchozích hran. Při spuštění aktivity jsou umístěny control tokeny na každý takový Initial node. Odchozí hrany Initial node musí být Control flow. Control token umístěný na Initial node je nabídnut souběžně všem odchozím hranám. Initial node je výjimkou z pravidla, že Control nodes nemohou zadržovat tokeny, ale mohou pouze řídit jejich tok. Pokud není nabídnutý token ihned přijat nebo je blokován jeho pohyb ve směru toku, zůstane v počátečním uzlu. [3]

Initial node je zobrazen jako malé plné kolečko. [6] Příklad Initial node je zobrazen na obrázku 2.8.

Obrázek 2.8 Initial node (počáteční uzel) [6]

2.4.1.2 Final node

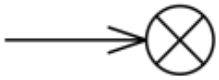
Final node (koncový uzel) je Control node, ve kterém tok aktivity končí. Final node by neměl mít žádnou odchozí hranu, ale může mít více hran příchozích. Final node přijímá všechny tokeny nabídnuté po jeho příchozích hranách. Existují dvě podtřídy Final node:

- Flow final node (uzel ukončující tok)
- Activity final node (uzel ukončující aktivitu) [3]

2.4.1.3 Flow final node

Flow final node (uzel ukončující tok) je Final node, který ukončuje tok aktivity zkonzumováním tokenu. Všechny tokeny, které do něj dorazí, jsou zničeny. Tato událost nemá žádný efekt na ostatní toky v aktivitě. [3]

Flow final node je zobrazen jako malé kolečko se symbolem × uvnitř. [6] Příklad Flow final node je zobrazen na obrázku 2.9.



Obrázek 2.9 Flow final node (uzel ukončující tok) [6]

2.4.1.4 Activity final node

Activity final node (uzel ukončující aktivitu) je Final node, který zastavuje všechny toky v dané aktivitě. K tomu dojde ve chvíli, když do Activity final node dorazí token, který je vlastněn aktivitou. V tu chvíli je přerušeno vykonávání dané aktivity. Aktivita může obsahovat více Activity final nodes. Pokud tomu tak je, tak je vykonávání aktivity ukončeno ve chvíli, kdy je libovolný token přijat libovolným Activity final node. Během ukončení vykonávání aktivity by měly být zničeny všechny tokeny, které jsou drženy v Object nodes. Také by měla být ukončena všechna vykonávání, která byla vyvolána synchronně. Naopak vykonávání vyvolaná asynchronně nebudou touto událostí ovlivněna. [3]

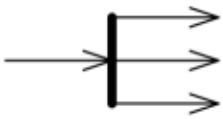
Activity final node je zobrazen jako malé kolečko s menším plným kolečkem uvnitř. [6] Příklad Activity final node je zobrazen na obrázku 2.10.

Obrázek 2.10 Activity final node (uzel ukončující aktivitu) [6]

2.4.1.5 Fork node

Fork node (rozdělovací uzel) je Control node, který rozděluje jeden tok do více souběžných toků. Měl by mít přesně jednu příchozí hranu, ale může mít více hran odchozích. Pokud je příchozí hrana typu Control flow, potom všechny odchozí hrany by měly být typu Control flow. Stejně by to mělo být i u hran typu Object flow, tudíž pokud je příchozí hrana typu Object flow, odchozí hrany by měly být typu Object flow. Token nabídnutý Fork node je nabídnut všem odchozím hranám Fork node. Pokud je alespoň jedna z nabídek přijata, nabízený token je odstraněn z původního zdroje a příjemci obdrží kopie daného tokenu. Jakákoliv jiná nabídka, která nebyla přijata z důvodu, že cíl ji nepřijal, zůstane na hraně ve stavu nevyřízená s tím, že může být přijata později. Tyto hrany přijmou separátní kopii nabízeného tokenu. Nabídky jim učiněné by měly být v pořadí, ve kterém byly přijaty danou hranou – FIFO. Toto je výjimka z pravidla, že hrany nemohou zadržovat tokeny, pokud jsou zablokovány v dalším pohybu v toku. Odchozí hrany z Fork node pokračují v zadržování tokenů, které přijaly, do doby, než všechny dosud nepřijaté nabídky nebudou přijaty jejich cíli. [3]

Fork node je zobrazen jako úzký obdélník s jednou příchozí hranou a dvěma nebo více odchozími hranami. [6] Příklad Fork node je zobrazen na obrázku 2.11.



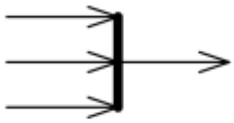
Obrázek 2.11 Fork node (rozdělovací uzel) [6]

2.4.1.6 Join node

Join node (spojovací uzel) je Control node, který synchronizuje více toků. Měl by mít přesně jednu odchozí hranu, ale může mít více příchozích hran. Pokud jsou všechny příchozí hrany typu Control flow, odchozí hrana by měla být typu Control flow. Pokud však je alespoň jedna příchozí hrana typu Object flow, odchozí hrana by měla být typu Object flow. Join node může mít definovaný atribut joinSpec, který vyjadřuje podmínku, kdy Join node odešle token dál po své odchozí hraně. Pokud atribut joinSpec není definovaný, tak je to ekvivalentní, jako kdyby definovaný byl a měl hodnotu operátoru `and`. Tento operátor je vyhodnocen na pravdu právě tehdy, když je z každé jeho příchozí hrany nabídnut alespoň jeden token. Obsah atributu joinSpec je vyhodnocen vždy, když Join node přijme po libovolné příchozí hraně nový token. Výsledkem vyhodnocení je pravda, nebo lež. Pokud je joinSpec vyhodnocen na pravdu, Join node nabídne token odchozí hraně podle následujících pravidel:

1. Pokud jsou všechny tokeny nabídnuté příchozími hranami Control tokens, tak jeden Control token je nabídnut odchozí hraně.
2. Pokud jakýkoliv nabídnutý token příchozími hranami je Object token, pouze Object tokens jsou nabídnuty na odchozí hraně. Tokeny jsou nabídnuty na odchozí hraně ve stejném pořadí, v jakým byly nabídnuty Join node. [3]

Join node je zobrazen jako úzký obdélník s jednou odchozí hranou a dvěma nebo více příchozími hranami. [6] Příklad Join node je zobrazen na obrázku 2.12.



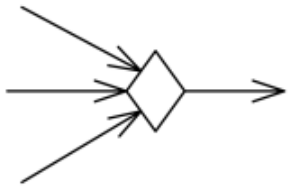
Obrázek 2.12 Join node (spojovací uzel) [6]

2.4.1.7 Merge node

Merge node (slučovací uzel) je Control node, který spojuje více toků bez toho, aby je synchronizoval. Merge node by měl mít přesně jednu odchozí hranu, ale může mít více příchozích hran. Pokud je odchozí hrana Merge node typu Control flow, poté všechny příchozí hrany by měly být typu Control flow. Pokud je odchozí hrana typu Object flow, všechny příchozí hrany by měly

být typu Object flow. Všechny tokeny nabídnuté příchozími hranami Merge node jsou nabídnuty odchozí hraně. Neprobíhá zde žádná synchronizace toků ani žádné spojování tokenů. [3]

Merge node je zobrazen jako prázdný diamant s právě jednou odchozí hranou a jednou nebo více příchozími hranami. [6] Příklad Merge node je zobrazen na obrázku 2.13.

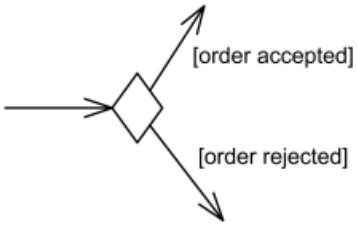


Obrázek 2.13 Merge node (slučovací uzel) [6]

2.4.1.8 Decision node

Decision node (rozhodovací uzel) je Control node, který provádí výběr mezi svými odchozími hranami. Měl by mít alespoň jednu a nejvýše dvě příchozí hrany. Počet odchozích hran by měl být větší nebo roven jedné. Pokud má Decision node jednu příchozí hranu a tato hrana je Control flow, všechny odchozí hrany by měly být typu Control flow. Pokud má jednu příchozí hranu a tato hrana je Object flow, všechny odchozí hrany by měly být typu Object flow. Decision node přijme token na své příchozí hraně a nabídne ho všem svým odchozím hranám. Každý token přijatý Decision node by měl být přijat právě jednou odchozí hranou. Nedochází k duplikování tokenů. Pokud mají odchozí hrany atribut guard (strážce), tak je tento atribut vyhodnocen pro každý příchozí token. Pořadí vyhodnocení jednotlivých guard není definováno, ty mohou být vyhodnocovány současně. [3]

Decision node je zobrazen jako prázdný diamant s jednou nebo dvěma příchozími hranami a s jednou nebo více odchozími hranami. [6] Příklad Decision node je zobrazen na obrázku 2.14.



Obrázek 2.14 Decision node (rozhodovací uzel) [6]

2.4.2 Executable node

Executable node (spustitelný uzel) je základní jednotkou aktivit. Samotné vykonání aktivity je prováděno po krocích v jednotlivých Executable nodes. Obecně platí, že Object nodes a Control nodes slouží ke koordinaci toku a dat mezi Executable nodes. Všechny konkrétní podtřídy Executable node jsou Action. Všechny příchozí a odchozí hrany by měly být Control flow. Executable node umí také produkovat a konzumovat data. V takovém případě je to ale prováděno za pomocí Pins. Executable node není vykonán, dokud všechny příchozí hrany nenabídnou token.

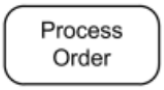
Pokud příchozí hrana nabídne více tokenů, jsou všechny zkonzumovány. Když Executable node vykonává svou práci, tak drží jeden control token, který značí, že probíhá vykonávání. Když je vykonávání u konce, tak je tento control token odebrán z Executable node a je nabídnut všem odchozím hranám, čímž proběhne implicitní rozdělení toku na více toků podle počtu odchozích hran Executable node. Jeho podtřídou je třída Action. [3]

Jedná se o abstraktní třídu, což znamená, že Executable node se sám konkrétně nikde v diagramu nemůže vyskytnout.

2.4.2.1 Action

Action (akce) je Executable node a je základní jednotkou aktivit. Action reprezentuje atomický krok v aktivitě, zároveň představuje transformaci nebo proces v daném systému. Aktivita reprezentuje soubor Actions. Action zprostředkovává Executable node v aktivitách. Actions koordinovaných aktivit mohou být spuštěny z důvodu, že jiné Actions byly dokončeny a jsou k dispozici potřebné control tokeny, nebo že jsou k dispozici potřebná data, nebo z důvodu, že nastala externí událost. Výsledkem Action mohou být mimo control tokenu také data, o která se starají Output pins dané Action. [3]

Action je zobrazena jako obdélník se zaoblenými rohy. Jméno Action nebo popis Action je umístěn uvnitř obdélníku. [4] Příklad Action je zobrazen na obrázku 2.15.



Obrázek 2.15 Action (akce) [4]

2.4.3 Object node

Object node (objektový uzel) je uzel, který je použit pro uchovávání object tokenů během vykonání aktivity. Tokeny jsou přijaty po příchozích hranách a mohou být předány dál po hranách odchozích. Token může být předán pouze jedné odchozí hraně. Object node může uchovávat v jednu chvíli více object tokenů se stejnou hodnotou, zároveň může mít definován typ object tokenu, který je schopný přijmout. Pokud takový typ definovaný nemá, může přijmout libovolný typ. Dále může mít Object node definovanou horní hranici počtu tokenů, které je schopný v jednu chvíli uchovávat. Pokud je počet uchovávaných tokenů větší nebo roven této horní hranici, Object node nepřijme další token. Jedním z atributů Object node je atribut ordering (řazení), který definuje, v jakém pořadí jsou tokeny nabízeny odchozím hranám. Atribut ordering může nabývat následujících hodnot:

- unordered – Pořadí, ve kterém jsou tokeny nabízeny, není definované.
- FIFO (“First In First Out”) – Tokeny jsou nabízeny odchozím hranám ve stejném pořadí, ve kterém byly přijaty Object node.
- LIFO (“Last In First Out”) – Tokeny jsou nabízeny odchozím hranám v opačném pořadí, než ve kterém byly přijaty Object node.
- ordered – Pořadí je konkrétně definované a podle toho jsou tokeny nabízeny na odchozích hranách. [3]

Dalším z atributů Object node je atribut isControlType. Tento atribut indikuje, zda se Object node může podílet na řízení toku, nebo ne. Příchozí a odchozí hrany Object node jsou typu Object flow. Pokud je atribut isControlType roven pravdě, tak Object node může mít příchozí a odchozí hrany také typu Control flow. Potom po těchto hranách typu Control flow mohou být předány vedle control tokenů také object tokeny. [3]

Object node je abstraktní třída, což znamená, že se sám konkrétně nikde v diagramu nevyskytuje, pouze definuje, co mají jeho podtřídy společné. Konkrétní podtřídy Object node, které se přímo vyskytují v Activity diagramech, jsou následující:

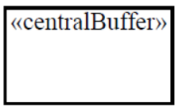
- Activity parameter node
- Central buffer node
- Expansion node
- Pin [3]

Obsahem této práce jsou pouze podtřídy Central buffer node a Pin.

2.4.3.1 Central buffer node

Central buffer node (uzel centrálního bufferu) se chová jako buffer (vyrovnávací paměť) mezi příchozí hranou a odchozí hranou. Obě tyto hrany by měly být typu Object flow. Přijímá všechny tokeny, které jsou mu nabídnuty na příchozích hranách, a následně je zde zadržuje. Držené tokeny jsou nabídnuty odchozím hranám podle pravidel pro řazení definované u Object node. Když je nabídka tokenu přijata, token je odebrán z Central buffer node a směřuje po odchozí hraně do přijímajícího Object node. [3]

Central buffer node je zobrazen jako obdélník s klíčovým slovem «centralBuffer» napsaným uvnitř. [6] Příklad Central buffer node je zobrazen na obrázku 2.16.



Obrázek 2.16 Central buffer node (uzel centrálního bufferu) [3]

2.4.3.2 Data store node

Data store node (uzel datového úložiště) je Central buffer node, který drží své tokeny po celou dobu exekuce dané aktivity. Když je nabídka tokenu přijata, token je stejně jako u Central buffer node odebrán z Data store node a poslán po odchozí hraně do přijímajícího Object node. Zároveň je však vyrobena kopie tohoto tokenu a ta je přidána do Data store node. To znamená, že tokeny poté přetrávají v Data store node po zbytek vykonávání dané aktivity. Pokud Data store node přijme token, který již má u sebe uložený, tak by při vytváření kopie neměla být tato kopie opakovaně přidána do Data store node. Tím, že obsahuje pouze unikátní tokeny, se liší od běžného Central buffer node. [3]

Data store node je zobrazen podobně jako Central buffer node, tedy jako obdélník. Rozdílem je, že uvnitř je napsáno klíčové slovo «datastore». [6] Příklad Data store node je zobrazen na obrázku 2.17.

Obrázek 2.17 Data store node (uzel datového úložiště) [6]

2.4.3.3 Pin

Pin (pin) reprezentuje vstup do Action nebo výstup z Action. Input pin (vstupní pin) představuje vstup, Output pin (výstupní pin) představuje výstup. Soubory vstupů i výstupů vlastněných Action jsou řazené. Pin je podtřída Object node, takže umí zadržovat object tokeny. Od Object node zdědil také atribut ordering (řazení), takže pořadí tokenů v Input pin a Output pin je definované stejně, jako je tomu u jiných Object node. Hodnoty, které jsou v tokenech Input pins, slouží jako vstupní data pro vykonání Action. Výstupní data Action jsou zabalena do object tokenů a umístěna do Output pins. Pin má definováno rozmezí, kolik celkově hodnot může být vstupem nebo výstupem dané Action. Soubor vstupních i výstupních tokenů není unikátní, takže se zde může nacházet více tokenů se stejnou hodnotou. [3]

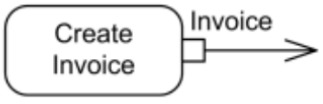
Input pin je Pin, který zadržuje vstupní hodnoty, které jsou následně zkonzumovány Action. Action nemůže začít své vykonání, pokud libovolný Input pin obsahuje méně object tokenů, než definuje jeho rozmezí. Horní hranice rozmezí určuje maximální počet hodnot, které mohou být zkonzumovány danou Action během jednoho vykonání. Zkonzumované tokeny jsou odstraněny z Input pin ve chvíli, kdy Action začne své vykonání. [3]

Output pin je Pin, který zadržuje výstupní hodnoty, které vznikly jako výsledek vykonání Action. Při každém vykonání Action nemůže skončit, dokud nevyprodukuje alespoň takový počet hodnot, jako je spodní hranice rozmezí definované v Output pin. Action by neměla vyprodukovat více hodnot, než je horní hranice rozmezí Output pin. [3]

Input pin je zobrazen jako malý obdélník připojený k Action. Jméno Pin je zobrazeno blízko malého obdélníku. Do Input pin vede Object flow. [6] Příklad Input pin je zobrazen na obrázku 2.18.

Obrázek 2.18 Input pin (vstupní pin) [6]

Output pin je zobrazen jako malý obdélník připojený k Action. Jméno Pin je zobrazeno blízko malého obdélníku. Z Output pin vede Object flow. [6] Příklad Output pin je zobrazen na obrázku 2.19.



Obrázek 2.19 Output pin (výstupní pin) [6]

2.4.4 Relace

Relace v Activity diagramech jsou realizovány pomocí Activity edges (hrany aktivity). Activity edge je přímé spojení mezi dvěma uzly, mezi kterými může proudit tok tokenů od zdrojového uzlu do cílového uzlu. Tokeny jsou nabídnuty Activity edge zdrojovým uzlem této hrany. Nabídka je poslána dál do cílového uzlu. Cílovým uzlem může být buď Object node, Executable node nebo Control node. Mají definované, kdy je token přijat a kdy není. Pokud uzel token přijme, tak je token odebrán ze zdrojového uzlu a putuje dál do cílového uzlu. Pokud ho nepřijme, tak je na cílovém uzlu, co s tokenem udělá, protože Activity edges nemohou zadržovat tokeny. [3]

Activity edge je abstraktní třída, což znamená, že se sama konkrétně nikde v diagramu nevyskytuje, pouze definuje, co mají její podtřídy společné. Konkrétní podtřídy Activity edge, které se přímo vyskytují v Activity diagramech, jsou následující:

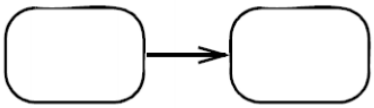
- Control flow (řídící tok)
- Object flow (objektový tok) [3]

2.4.4.1 Control flow

Control flow (řídící tok) je Activity edge, která posílá pouze control tokeny. Control flows jsou použity pro explicitní posloupnost vykonání uzlů. Cílový uzel nemůže přijmout control token a začít své vykonání, dokud zdrojový uzel nedokončí své vykonání a neposkytne token. [3]

Control flow je zobrazena jako plná čára s otevřenou šipkou na konci ve směru toku. [6] Příklad Control flow je zobrazen na obrázku 2.20. Obrázek 2.21 zobrazuje Control flow spojující dvě Actions.

Obrázek 2.20 Control flow (řízení toku) [3]



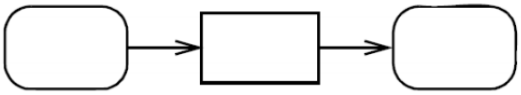
Obrázek 2.21 Control flow spojující dvě Actions [3]

2.4.4.2 Object flow

Object flow (objektový tok) je Activity edge, přes kterou je možné posílat object tokeny. Object flow modeluje tok hodnot mezi Object nodes. Tokeny jsou nabízeny cílovému uzlu ve stejném pořadí, jako byly nabídnuty ze zdroje. Pokud je více tokenů nabídnuto ve stejný čas, jsou tokeny nabízeny ve stejném pořadí, jako kdyby bývaly přišly ze zdroje jeden po druhém. [6]

Object flow je zobrazena stejně jako Control flow, tedy jako plná čára s otevřenou šipkou na konci ve směru toku. [6] Příklad Object flow je zobrazen na obrázku 2.22. Obrázek 2.23 zobrazuje Object flow spojující dvě Actions přes jeden Object node.

Obrázek 2.22 Object flow (objektový tok) [3]



Obrázek 2.23 Object flow spojující dvě Actions přes jeden Object node [3]

2.4.5 Activity partition

Activity partition (rozdělení aktivity) slouží k identifikaci uzlů, které mají společné vlastnosti. Activity partition mohou navzájem sdílet svůj obsah. Nejčastěji vyjadřují rozdělení podle organizačních jednotek v modelu. [3] Příkladem může být rozdělení aktivity na Activity partition ”Zákazník” a Activity partition ”Prodejce”. V takové aktivitě jsou Actions, které vykonává ”Zákazník”, například Action ”Platba”. Action ”Předání produktu” bude naopak vykonávat ”Prodejce”. Právě díky Activity partition je možné skupiny uzlů přiřadit dané organizační jednotce a tedy to, co mají společné, je, že jsou vykonávány právě touto organizační jednotkou.

Activity partition může být zobrazeno několika způsoby. Jedním z nich je pomocí swimlane (dráha), která je zobrazena jako dvě souběžné čáry (horizontální či vertikální) se jménem, které je v obdélníku na jednom z konců. [3] Příklad Activity partition, která je vyjádřena pomocí horizontální swimlane, je zobrazen na obrázku 2.24.

Obrázek 2.24 Activity partition swimline (rozdělení aktivity) [3]

2.5 Modelovací nástroje

2.5.1 Enterprise Architect

Enterprise Architect od společnosti Sparx Systems je kompletní CASE nástroj pro systémovou analýzu a návrh, který pokrývá celý životní cyklus vývoje systému. Tím je myšleno od zadání požadavků přes analýzu stavů, návrh modelů, testování a údržbu, vše s využitím UML diagramů. Enterprise Architect poskytuje podporu pro týmový vývoj pro jednotlivé role. [7]

2.5.2 Visual Paradigm

Visual Paradigm je UML CASE nástroj podporující notaci UML 2, SysML a Business Process Modeling Notation (BPMN) ze skupiny OMG. Kromě podpory modelování poskytuje také možnosti generování sestav a inženýrství kódu včetně generování kódu. Umí reverzně převést kód do diagramů a poskytuje round-trip inženýrství pro různé programovací jazyky. [8]

2.5.3 OpenPonk

OpenPonk je metamodeling platforma implementovaná v dynamickém prostředí Pharo zaměřená na podporu aktivit obklopující software a podnikového inženýrství, jako je například modelování, vykonávání, simulování a generování zdrojového kódu. [9]

2.6 Python

Python je vysokoúrovňový skriptovací programovací jazyk, který v roce 1991 navrhl Guido van Rossum. Nabízí dynamickou kontrolu datových typů a podporuje různá programovací paradigmata včetně objektově orientovaného, imperativního, procedurálního a funkcionálního. Python je vyvíjen jako open source projekt, který zdarma nabízí instalační balíčky pro většinu běžných platforem (Unix, MS Windows, macOS, Android). Ve většině distribucí systému Linux je Python součástí základní instalace. [10]

2.7 Neo4j

Neo4j je grafová databáze, která je od základu postavená nejen pro uchování dat, ale i relací mezi daty. V grafové databázi jsou data uchována pomocí uzlů (nodes) a relací (relationships). Uzly představují samotná data. Relace, kromě informace které uzly spojují, mohou také obsahovat vlastní data. [11]

Neo4j je implementována v jazyce Java a je dostupná pro softwares napsané v jiných programovací jazycích za použití dotazovacího jazyka Cypher pomocí HTTP endpointu nebo pomocí binárního ”bolt” protokolu. [11]

2.7.1 Cypher

Cypher je deklarativní dotazovací jazyk, který umožňuje efektivní a expresivní dotazování nad daty v grafech. Cypher byl původně určen pouze pro grafovou databázi Neo4j, ale nakonec byl zpřístupněn jako openCypher projekt v srpnu 2015. Jazyk Cypher byl navrhnut tak, aby dokázal vykonat stejné operace jako SQL (standardní jazyk pro dotazování v relačních databází), ale také aby dokázal uspokojit potřeby databází postavených na základě konceptu grafové teorie. Aby bylo možné zaznamenat, jak jsou jednotlivé uzly navzájem provázané, jsou data v grafovém modelu reprezentovány jako uzly a relace. [12]

2.8 UUID

Univerzální unikátní identifikátor (UUID) je 128bitové číslo používané k identifikaci informací v počítačových systémech. Když jsou UUID generovány podle standardních metod, jsou pro praktické účely jedinečné. Jejich jedinečnost nezávisí na ústředním registračním orgánu ani na koordinaci mezi stranami, které je vytvářejí. I když není nulová pravděpodobnost že UUID může existovat duplicitně, je taková možnost dostatečně blízká nule, aby byla zanedbatelná. Kdokoli tedy může vygenerovat UUID pro svoji potřebu a použít jej k identifikaci něčeho s jistotou, že

identifikátor se nebude shodovat s UUID pro identifikaci něčeho jiného. Informace označené UUID nezávislými stranami mohou být proto později sloučeny do společné databáze se zanedbatelnou pravděpodobností duplikátů. [13]

2.9 Repocribro

Repocribro je webová aplikace napsaná v jazyce Python, která umožňuje uživatelům registrovat GitHub repositáře. Slouží například pro vyhledávání, prohlížení a testování repositářů uvnitř aplikace. Hlavní myšlenkou je poskytnout jednoduchý, ale mocný modulární nástroj pro vytváření skupin GitHub repositářů, které jsou vyvinuty různými uživateli a organizacemi, ale sdílejí podobný cíl. [14]

Kapitola 3
Analýza a návrh

Exquiro je projekt pro indexování a vyhledávání v konceptuálních modelech. Tento projekt je součástí webové aplikace Repocribro. Exquiro je implementováno v jazyce Python. Primárně slouží k parsování XMI souborů, které vznikly vyexportováním UML diagramů, a k následnému nahrání do grafové databáze Neo4j, kde je možné rychlé a snadné dotazování nad jednotlivými grafy nebo soubory grafů. Jelikož hlavním cílem této práce je implementovat moduly parseru, které se následně budou integrovat do projektu Exquiro, je zapotřebí zanalyzovat stávající projekt a zvolit vhodné prostředky a technologie, aby následná integrace byla co nejjednodušší.

3.1 Programovací jazyk Python

Pro implementaci jednotlivých modulů parseru je zapotřebí vybrat vhodný programovací jazyk, ve kterém budou parsery implementovány. Stávající projekt indexování a vyhledávání v konceptuálních modelech Exquiro je implementován v jazyce **Python**. Z důvodu maximální kompatibility se stávajícím projektem jsme se rozhodli moduly implementovat také v jazyce Python.

3.1.1 Python balíčky

Jazyk Python je open source projekt, tudíž umožňuje, aby kdokoliv implementoval rozšíření tohoto jazyka pomocí balíčků. Pro části, které nejsou primárním cílem této práce, je vhodné využít již implementované balíčky. Zvoleným balíčkům a odůvodnění jejich volby se věnuje tato sekce.

3.1.1.1 Čtení XMI souborů

Implementace čtení souborů, které jsou ve formátu XML, není cílem této práce, proto jsme se rozhodli využít již implementovaný Python balíček, který poskytuje efektivní práci s XML soubory. Pro tento účel jsme našli dva vyhovující balíčky. Prvním z nich je xml.etree.ElementTree [15], který je doporučen přímo v dokumentaci jazyka Python a je součástí základní instalace jazyka. Druhým vhodným balíčkem je lxml [16], který není součástí Pythonu a pro jeho použití je nutná instalace.

Balíček lxml je rozšířením vestavěného balíčku xml.etree.ElementTree. Poskytuje o mnoho více možností, například podporuje XSLT, XPath a další. Z toho důvodu jsme se rozhodli pro použití balíčku **lxml**. Obsahuje například metodu `findall(self, path, namespaces)`, která umožňuje najít všechny elementy v konkrétním XML elementu například podle požadovaného atributu. Příklad použití metody je zobrazen ve výpisu kódu 3.1, kde výsledkem je

seznam XML elementů, které mají tag roven `packagedElement` a zároveň atribut `xmi:type` roven `uml:Package`.

```
packages = element.findall('.//packagedElement[@xmi:type="uml:Package"]',
                           namespaces)
```

Výpis kódu 3.1 Příklad použití metody findall z balíčku lxml

3.1.1.2 Neo4j v Pythonu

Výstupem parserů musí být třídy, které lze nahrát do grafové databáze Neo4j. Pro tento účel jsme zvolili balíček **neomodel** [17], který je doporučen přímo společností Neo4j a poskytuje všechny metody a třídy potřebné k propojení s grafovou databází Neo4j.

3.1.1.3 Testování v Pythonu

Každý správný software by měl být testován, aby se dalo do jisté míry zaručit, že funguje tak, jak by měl fungovat. Testy také slouží pro případ, kdy se provádí refactoring kód. Cílem refactoring je změnit to, jak je daná věc prováděna, ale zároveň zachovat to, co je prováděno. Testy proto odhalí, zda tomu tak opravdu je a zda testovaná jednotka má stále stejný výstup.

Existují různé druhy testů. Jedním z nich jsou unit testy. Ty se zaměřují na nejmenší testovatelné části software. V tomto případě se jedná o konkrétní metody modulů parseru. Pro testování unit testů jsme se rozhodli využít testovací framework **unittest**. Unittest je součástí základní instalace jazyka Python. Tento framework obsahuje všechno potřebné pro testování unit testů. V nich jde primárně o porovnávání hodnot. Pro tyto účely unittest obsahuje mnoho metod, například metoda `assertEqual(first, second)`[18] porovná, zda parametr `first` je totožný s parametrem `second`. Také podporuje testování odchytávání výjimek, což je pro naši práci velmi užitečné, protože pomocí výjimek jsou řešeny případné problémy a chyby.

Unittest poskytuje možnost Test Discovery. Ta mimo jiné umožňuje najít všechny testovací soubory, které mají stejný název souboru jako zadaný vzor. Tento vzor může obsahovat znak `*`, který značí, že se na jeho místě může nacházet libovolný řetězec znaků. Díky tomu je možné spouštět jen vybrané skupiny testů. Pro naši práci je to užitečné, protože máme dvě oddělené sady testů – jednu pro otestování parserů zpracovávajících Package diagramy, druhou pro parsery zpracovávající Activity diagramy.

Dalším druhem testů jsou integrační testy. Ty se zaměřují na správné fungování celku, který je tvořen jednotlivými částmi systému. Integrační testy bude potřeba provést, protože jedním z dílčích cílů práce je integrovat implementované moduly parseru do projektu Exquiro. Integrační testy není vhodné provádět pomocí napsání automatických testů, ale pomocí toho, že jsou otestovány přímo uživatelem. Z toho důvodu jsme se rozhodli testy neimplementovat. Integrační testy jsme se rozhodli provést pomocí uživatelských testů.

3.1.1.4 Generování náhodných identifikátorů

V grafové databázi je zapotřebí u každého prvku registrovat mimo jiné unikátní identifikátor. To znamená, že každý uzel a každá relace, která se v grafové databázi vyskytne, musí mít svůj vlastní unikátní identifikátor. Uzly a relace, které jsou explicitně definované ve zpracovávaném modelu, mají identifikátor definovaný přímo jako jeden ze svých atributů. Unikátnost identifikátoru zajišťuje samotný modelovací nástroj, který je zodpovědný za vytvoření XMI souboru. Některé výsledné relace v grafové databázi nejsou součástí modelu, ale vznikly při jeho zpracování. Jedná se například o relace **Member of** nebo **Partition member**, které nejsou explicitně vyjádřeny v původním modelu, ale pro zachování vztahů je nutné je definovat.

První možností, kterou jsme zvažovali, byla implementace vlastního systému na generování unikátních identifikátorů pro celý model. Po analýze XMI souborů ze všech modelovacích nástrojů

jsme došli k závěru, že všechny uzly, které se v modelech vyskytují, mají unikátní identifikátor jako jeden ze svých atributů. V dalším kroku analýzy jsme zjistili, že zdroje a cíle relací v modelu jsou definované pomocí identifikátorů uzlů. Proto nemá smysl nahrazovat identifikátory uzlů nově vytvořenými identifikátory, protože by bylo zapotřebí využít ty stávající pro správné zachování relací.

Relacím, které jsou v modelu definované explicitně, jsme po podobné úvaze jako u uzlů ponechali jejich identifikátor stejný, jako je identifikátor definovaný v jejich atributech. Relacím, které vznikly zpracováním modelu a nebyly součástí původního modelu, je zapotřebí přiřadit unikátní identifikátor. Jelikož se jedná o relace vytvořené pro zachování vazeb mezi uzly a v původním modelu se nevyskytují, není na ně v modelu žádná reference ani odkaz. Proto může být vytvoření identifikátoru čistě náhodné a v databázi se objeví pouze jednou, a to jako atribut vytvořené relace. Z důvodu, že tato situace nenastává moc často, jsme se rozhodli využít Python balíček **uuid** pro generování náhodných identifikátorů. Tento balíček poskytuje možnost vytvoření UUID podle standardních metod, což pro praktické účely zajišťuje jejich jedinečnost. Konkrétně jsme se rozhodli pro implementaci využít metodu `uuid4()`, která bez vstupu vygeneruje náhodný unikátní identifikátor. Příklad použití metody je zobrazen ve výpisu kódu 3.2.

```
unique_member_id = str(uuid.uuid4())
```

Výpis kódu 3.2 Příklad použití metody uuid4 z balíčku uuid

3.2 XMI

Soubory XMI slouží k výměně UML metadat modelů mezi různými modelovacími nástroji. Jedná se o standard definovaný OMG. Hlavním cílem práce je implementovat parsery, které budou zpracovávat soubory právě ve formátu XMI, jelikož to je nejlepší způsob, jak z modelovacích nástrojů získat informace o UML metadat modelů. Analýzou XMI souborů se věnují následující sekce.

3.2.1 Struktura XMI

Základní struktura XMI souborů je definována standardem XMI [2]. V něm je ale také uvedeno, že každý XML element v souboru může mít libovolný počet elementů, které ho rozšiřují. Tyto elementy jsou konkrétně nazvány `xmi:Extensions`. Počet a obsah rozšiřujících elementů je čistě na modelovacím nástroji a může se v nich objevit téměř cokoliv. Například obsahují informace o tom, kde jsou umístěny jednotlivé elementy v diagramu (pozice na obrazovce). Modelovací nástroje mají možnost vygenerovat XMI soubory bez rozšiřujících elementů. Z toho důvodu se na jejich přítomnost nemůžeme spoléhat a při zpracování souboru nejsou zohledňovány. Rozšiřující elementy mají přímý vliv na strukturu XMI souborů, která je závislá na modelovacím nástroji, což je patrné z příkladů 3.3 a 3.4, které vznikly vyexportováním stejného modelu za pomocí různých modelovacích nástrojů.

```
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
                 xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
                 xmlns:EAUML="http://www.sparxsystems.com/profiles/EAUML/1.0">
    <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
    <uml:Model xmi:type="uml:Model" name="EA_Model" visibility=""...>
        ...
    </uml:Model>
    <xmi:Extension extender="Enterprise Architect" extenderID="6.5"...>
</xmi:XMI>
```

Výpis kódu 3.3 Příklad struktury XMI souboru (Enterprise Architect)

```
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.0"
                           xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
    <xmi:Documentation exporter="Visual Paradigm" exporterVersion="7.0.2"...>
    <xmi:Extension extender="Visual Paradigm"...>
    <uml:Model name="model" xmi:id="x3r5uB6GAqAUdQkK">
        ...
    </uml:Model>
    <uml:Diagram diagramType="ActivityDiagram" name="Swimlane..."...>
</xmi:XMI>
```

Výpis kódu 3.4 Příklad struktury XMI souboru (Visual Paradigm)

Je patrné, že ani přímí potomci kořenového elementu `xmi:XMI` nejsou totožní. Kořenový element z příkladu 3.4 obsahuje element `uml:Diagram`, přičemž kořenový element z příkladu 3.3 takový element neobsahuje, jelikož je zahrnut v elementu `xmi:Extensions`.

Pro práci je nejdůležitější element `uml:Model`, který obsahuje informace o vyexportovaném UML modelu. Kořenový element souboru `xmi:XMI` obsahuje informace o namespaces, které jsou v souboru použity. Dané informace je nutné uchovat pro budoucí zpracování jednotlivých elementů. Všechny ostatní části XMI souborů nejsou pro naši práci důležité. Je zajímavé, že užití rozšiřujících elementů se velmi liší s každým modelovacím nástrojem, což se odráží například i na velikosti jednotlivých souborů. Soubory, které vznikly vyexportováním z modelovacího nástroje Visual Paradigm jsou až desetkrát větší než soubory, které vznikly vyexportováním z modelovacích nástrojů Enterprise Architect a OpenPonk.

3.2.2 Modely v XMI

UML model se v XMI souboru nachází v XML elementu `uml:Model`. Standard XMI definuje, že model musí být v souladu se specifikací UML. Modelovací nástroje se staví k popisu modelu různými způsoby. Enterprise Architect umístí obsah modelu do XML elementu `packagedElement` s atributem `xmi:type="uml:Package"`, zatímco Visual Paradigm obsah modelu umístí do XML elementu `ownedMember` s atributem `xmi:type="uml:Model"`. Tato skutečnost je demonstrována na příkladech 3.5 a 3.6. Modelovací nástroj OpenPonk volí stejnou variantu jako Enterprise Architect, tedy XML element `packagedElement`.

```
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    <packagedElement xmi:type="uml:Package" ... name="Shopping system">
        ...
    </packagedElement>
</uml:Model>
```

Výpis kódu 3.5 Příklad struktury XML elementu uml:Model (Enterprise Architect)

```
<uml:Model name="model" xmi:id="P2gshR6GAqAUIAYR">
    <ownedMember xmi:type="uml:Model" ... name="Shopping System">
        ...
    </ownedMember>
</uml:Model>
```

Výpis kódu 3.6 Příklad struktury XML elementu uml:Model (Visual Paradigm)

Podle UML specifikace jsou obě varianty přípustné, protože ani jedna není v rozporu s uvedenou specifikací. Výsledkem je, že bude nutné implementovat moduly parseru v závislosti na modelovacím nástroji. Tomu, které konkrétní parsery je potřeba implementovat, se věnuje sekce Parsery.

3.3 Package diagram

Package diagram popisuje strukturu z pohledu Packages (balíčků). V takovém diagramu by se měly vyskytovat pouze elementy druhu Package. Podtřídou samotného Package jsou Profile (profil) a Model (model). Dále se v Package diagramu vyskytují relace mezi jednotlivými elementy. Tomu, které elementy a relace jsme se rozhodli začlenit do rozsahu práce, se věnuje tato sekce.

3.3.1 Elementy

Hlavním elementem v Package diagramu je samotný element Package, tudíž jsme uvážili, že ho nemůžeme opomenout v obsahu práce. Dalšími elementy, které bychom mohli uvažovat jsou elementy Profile, Model a Component.

Profile je podtřída Package, která se používá pro rozšíření UML. Jedním z nejčastějších rozšíření je rozšíření o stereotypy. Takové rozšíření není z hlediska této práce zajímavé. Proto jsme se rozhodli elementy typu Profile nezpracovávat.

Model je také podtřída Package. Používá se pro zachycení pohledu fyzického systém za účelem rozlišit to, co je podstatné, a to, co není. Stejně jako Package Model v jistém smyslu seskupuje své členy do skupiny. Rozdíl mezi nimi je takový, že Package vlastní elementy, zatímco Model popisuje pohled na elementy.[3] Jelikož třída Model neobsahuje žádné důležité vlastnosti navíc (oproti Package), tak jsme se rozhodli, že pokud se ve zpracovávaném modelu vyskytne element Model, tak ho budeme brát jako prvek Package. V grafové databázi takový prvek bude reprezentován stejně jako element Package. Modelovací nástroje Enterprise Architect a OpenPonk při exportu dokonce nerozlišují mezi elementy Model a Package a oba elementy vyexportují jako element Package. Příklad reprezentace elementu Package s názvem `Customer` v XMI souboru je zobrazen ve výpisu kódu 3.7. Informace o tom, že se jedná o Model a ne o Package, je uchována v rozšiřujících elementech, které nejsou v této práci zpracovávány.

```
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    <packagedElement xmi:type="uml:Package" ... name="Shopping system">
        <packagedElement xmi:type="uml:Package" ... name="Customer" .../>
    </packagedElement>
</uml:Model>
```

Výpis kódu 3.7 Package v XMI souboru (Enterprise Architect)

Dalším elementem, který se někdy objevuje v Package diagramech, je element Component. Component reprezentuje modulární část systému, která zapouzdřuje svůj obsah a která je nahraditelná ve svém prostředí. Component může být chápán jako autonomní část v daném systému. [3] Component není podtřída Package a k zaznamenání Component slouží Component diagram. Z toho důvodu jsme se rozhodli takové elementy při zpracování ignorovat a nezačlenit do výsledné reprezentace v grafové databázi.

3.3.2 Relace

Relací, které se mohou objevit v Package diagramu, je velké množství. Rozhodli jsme se vybrat ty nejdůležitější a nejrelevantnější z pohledu Package. Mezi takové relace jsme vybrali **Package merge** (slučování balíčku), **Package import** (import balíčku), **Dependency** (závislost) a **Usage** (použití). Relace, které se vyskytují v Package diagram, jsou v XMI souborech reprezentovány následovně:

- Jsou reprezentovány jako XML elementy.
- Pokud je Package zdrojem libovolné relace, potom obsahuje odpovídající XML element dané relace.

- Mají definované atributy supplier a client, které vyjadřují cíl a zdroj relace, nebo mají definovaný pouze cíl relace, přičemž zdrojem je Package, ve kterém se daný XML element nachází.

Package merge a Package import jsou přímo spjaté s Packages a jejich obsahem. Package merge slučuje Packages do jednoho nového výsledného Package, Package import definuje, že obsah jednoho Package je přidán do obsahu druhého Package. Jedná se o základní relace mezi Packages, a proto jsme se je rozhodli začlenit do této práce.

Příklad relace Package merge v XMI je uveden ve výpisu kódu 3.8. XML element se jménem `packageMerge` představuje Package merge relaci a obsahuje atribut `mergedPackage`, který určuje, který Package je cílem relace, přičemž zdrojem je Package, ve kterém se XML element `packageMerge` nachází. Package import volí stejnou variantu s jiným pojmenováním XML elementu a atributu.

```
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    <packagedElement xmi:type="uml:Package" ... name="Shopping system">
        <packagedElement xmi:type="uml:Package" xmi:id="EAP...F3" name="A"\>
        <packagedElement xmi:type="uml:Package" xmi:id="EAP...99" name="B">
            <packageMerge xmi:type="uml:PackageMerge" mergedPackage="EAP...F3"/>
        </packagedElement>
    </packagedElement>
</uml:Model>
```

Výpis kódu 3.8 Package merge v XMI souboru (Enterprise Architect)

Dependency a její podtřída Usage jsou relace, které nemusí být jen mezi Packages. Obě jsou client-supplier relace (klient-poskytovatel), kde klient je do jisté míry závislí na poskytovateli. Rozdíl mezi Dependency a Usage je takový, že Dependency vyjadřuje libovolnou závislost mezi danými Packges, přičemž Usage definuje, že client není bez supplier plně definován a nutně vyžaduje alespoň jeden z jeho elementů pro svou plnou implementaci. V Package diagramech se mohou objevit obě. Jejich použití v Package diagramech dává dobrý smysl, protože umožňuje vyjádření různých závislostí mezi jednotlivými Packages. Z těchto důvodů jsme se rozhodli rozlišit mezi Dependency a Usage a obě relace zapracovat do této práce.

Příklad relace Dependency v XMI je uveden ve výpisu kódu 3.9. XML element se jménem `packagedElement` a s atributem `xmi:type="uml:Dependency"` představuje Dependency relaci. Obsahuje atributy `client` a `supplier`, které určují zdroj a cíl relace Dependency. Relace Usage se liší pouze atributem `xmi:type`, který je roven `"uml:Usage"`.

```
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    <packagedElement xmi:type="uml:Package" ... name="Shopping system">
        <packagedElement xmi:type="uml:Package" xmi:id="EAP...F3" name="A"\>
        <packagedElement xmi:type="uml:Package" xmi:id="EAP...99" name="B">
            <packagedElement xmi:type="uml:Dependency" supplier="EAP...F3"
                                                      client="EAP...99"/>
        </packagedElement>
    </packagedElement>
</uml:Model>
```

Výpis kódu 3.9 Dependency v XMI souboru (Enterprise Architect)

Mezi relace, které jsme uvažovali zahrnout do této práce, patří relace Profile application (aplikování profilu). Tato relace se často používá v Package diagramech a vyjadřuje, jaké Profiles (profily) jsou aplikovány na daný Package. Aby bylo možné relaci Profile application zahrnout, bylo by potřeba do obsahu práce zahrnout také element Profile. Ten však není z hlediska této práce zajímavý, takže ani relaci Profile application nemá smysl začleňovat do obsahu práce.

Dalšími relacemi, o kterých jsme uvažovali, jsou další podtřídy relace Dependency, konkrétně Abstraction (abstrakce) a Deployment (nasazení), ale jejich použití není moc časté a z toho důvodu se jimi práce nezabývá.

Mimo relace, které jsou definované přímo specifikací UML, stojí relace **Member of** (být členem). Relace Member of je vytvořena jen pro účely této práce a to z důvodu zachování vztahů mezi Packages, kdy jeden je obsahem druhého. Relace je v UMl definována jako relace nestedPackage. Relace nestedPackage se týká pouze elementů Packages, což limituje budoucí rozšíření například z pohledu tříd (Classes). Z toho důvodu jsme se rozhodli vytvořit novou relaci Member of, která umožní lepší budoucí rozšíření a také zjednoduší samotné výsledné vyhledávání v modelech. Konkrétně bude možné jednoduše vyhledat obsah Package podle jedné jediné relace. Příklad situace, kdy je potřeba vytvořit relaci Member of, je zobrazen ve výpisu kódu 3.10, kde Packages s atributy `name="Customer"` a `name="Shopping Cart"` jsou obsahem Package s atributem `name="Shopping system"`.

```
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    <packagedElement xmi:type="uml:Package" ... name="Shopping system">
        <packagedElement xmi:type="uml:Package" name="Customer"\>
        <packagedElement xmi:type="uml:Package" name="Shopping Cart"\>
    </packagedElement>
</uml:Model>
```

Výpis kódu 3.10 Situace použití relace Member of v XMI souboru (Enterprise Architect)

3.4 Activity diagram

Activity diagram popisuje aktivity v systému. Obsahuje převážně Control nodes (řídící uzly), Executable nodes (spustitelné uzly), Object nodes (objektové uzly) a Activity partitions (rozdělení aktivity). Dále obsahuje hrany mezi jednotlivými uzly, které mohou být buď Control flow (řídící tok), nebo Object flow (objektový tok).

3.4.1 Elementy

Elementy, které se vyskytují v Activity diagramech, lze rozdělit do tří skupin. Rozdělení jsme provedli podle toho, jaké funkce jednotlivé typy elementů zastávají v Activity diagramu. Výsledkem je rozdělení na Control nodes, Executable nodes a Object nodes. Následně jsme se zaměřili na jednotlivé skupiny a vybrali z nich nejdůležitější typy elementů, které jsou v Activity diagramech používány. Mimo zmíněné tři skupiny stojí element Activity partition, který není v diagramech reprezentován jako element účastnící se aktivity ale spíš jako element, který seskupuje jiné elementy do skupiny.

Všechny elementy, které jsou obsaženy v aktivitě, se v XMI souborech nachází v XML elementu `packagedElement` s atributem `xmi:type="uml:Activity"`. Elementy jsou následně reprezentovány jako XML elementy `node`. Příklad je zobrazen ve výpisu kódu 3.11. Výjimku tvoří XMI soubory, které vznikly z modelovacího nástroje Visual Paradigm, kde může dojít k situaci, že elementy nejsou umístěny do XML elementu dané aktivity, ale jsou v XMI souboru na stejné úrovni. Tato situace je vyřešena v implementaci samotného parseru.

```
<uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
    <packagedElement xmi:type="uml:Activity" ... name="EA_Activity1">
        <node xmi:type="uml:Action" name="Close order" visibility="public">
    </packagedElement>
</uml:Model>
```

Výpis kódu 3.11 Aktivita v XMI souboru (Enterprise Architect)

3.4.1.1 Control nodes

Control nodes (řídící uzly) slouží ke koordinaci aktivity. Tyto uzly jsou nezbytné pro základní procesy v aktivitách. Všechny jsou rozdílné a mají jinou úlohu v aktivitě. Proto jsme ani neuvažovali některý Control node vynechat a nezačlenit ho do této práce. Mezi Control nodes jmenovitě patří **Initial node**, **Fork node**, **Join node**, **Merge node**, **Decision node**, Final node a jeho podtřídy – **Flow final node** a **Activity final node**.

V Activity diagramech jsou Decision node a Merge node reprezentovány úplně stejně, liší se pouze počtem příchozích a odchozích hran. Stejně tomu je i u Fork node a Join node. Modelovací nástroje většinou nerozlišují mezi jednotlivými elementy, což se odráží i na výsledném obsahu XMI souboru, kde jsou Fork node a Join node zaznamenány totožně. Stejně tomu je i u dvojice Decision node a Merge node. Došli jsme k závěru, že bude nejlepší tyto elementy v jednotlivých dvojicích sloučit do jednoho typu elementu, který se objeví ve výsledné grafové databázi. Rozhodli jsme se je pojmenovat DecisionMergeNode a ForkJoinNode, aby bylo patrné, že daný typ reprezentuje oba typy elementů.

Control nodes jsou reprezentovány v XMI souborech jako XML elementy `node` s příslušným atributem `xmi:type`, který blíže specifikuje, o jaký konkrétní typ se jedná. Výčet Control nodes je zobrazen ve výpisu kódu 3.12.

```
<node xmi:type="uml:ActivityFinalNode" ...>
<node xmi:type="uml:DecisionNode" ...>
<node xmi:type="uml:FlowFinalNode" ...>
<node xmi:type="uml:ForkNode" ...>
<node xmi:type="uml:InitialNode" ...>
```

Výpis kódu 3.12 Control nodes v XMI souboru (Enterprise Architect)

3.4.1.2 Executable nodes

Executable nodes (spustitelné uzly) jsou základní jednotkou dané aktivity. Podtřídou Executable node je třída Action. Tato třída má velmi mnoho podtříd. Zdaleka ne všechny podtřídy jsou v praxi používané. To se odráží i na modelovacích nástrojích, které umožňují namodelovat pouze vybrané druhy Action. Modelovací nástroje nejčastěji nabízí Call Behavioral Action (akce zavolání chování), Send Signal Action (akce poslání signálu) a Accept Action (přijímající akce). Z nich jsou nejčastěji používané Call Behavioral Actions, které jsou často prezentovány v modelovacích nástrojích jako Actions. Do práce jsme se rozhodli začlenit pouze **Call Behavioral Actions** s tím, že je budeme reprezentovat jako **Actions**. Příklad reprezentace Action v XMI souboru je zobrazen ve výpisu kódu 3.11.

3.4.1.3 Object nodes

Object nodes (objektové uzly) slouží k zacházení s daty a object tokeny během vykonání aktivity. Dělí se na Activity parameter node, Central buffer node, Expansion node a Pin. Central buffer node a podtřída Data store node slouží k uchování object tokenů během vykonání aktivity. To v software systémech může sloužit například jako databáze údajů. Z daného důvodu jsme se rozhodli začlenit elementy **Central buffer node** a **Data store node** do obsahu práce. Jelikož se jedná o uzly, tak jsou v XMI souborech reprezentovány jako XML elementy `node`. Rozlišení obstará atribut `xmi:type`. Příklad Central buffer node a Data store node je zobrazen ve výpisu kódu 3.13.

```
<node xmi:type="uml:CentralBufferNode" ...>
<node xmi:type="uml:DataStoreNode" ...>
```

Výpis kódu 3.13 Central buffer node a Data store node v XMI souboru (Enterprise Architect)

Object nodes mají příchozí i odchozí hrany typu Object flow. Tato relace musí být pouze mezi Object nodes. Aby bylo možné propojit jiné elementy s Object nodes, jsou zapotřebí Pins. Ty slouží právě k tomu, aby bylo možné zpracovat výstupní či vstupní data z Action. Proto jsme usoudili, že element **Pin** je nezbytný a je nutné ho začlenit do této práce, pokud chceme zahrnout i jiné Object nodes. Příklad reprezentace Pin v XMI souborech je zobrazen ve výpisu kódu 3.17.

Elementy Activity parameter node a Expansion node nepatří mezi nejčastěji používané elementy v Activity diagramech, a proto jsme se rozhodli je nezačlenit do rozsahu práce. Dalším důvodem, který přispěl k rozhodnutí nezačlenit tyto elementy do rozsahu práce, je fakt, že modelovací nástroje nepodporují namodelování některých zmíněných elementů.

3.4.1.4 Activity partition

Activity partition určuje rozdělení aktivity na skupiny, které mají něco společného. Nejčastěji jsou Activity partition použity pro rozdělení Actions do skupin podle toho, kým jsou vykonávány. Jejich značení v Activity diagramech může být různé. Vždy ale obsahují informaci, které elementy patří do dané Activity partition. Jedná se o často používaný koncept, a proto jsme se rozhodli **Activity partition** zapracovat do této práce. V XMI souborech je Activity partition reprezentováno pomocí XML elementu `group`, který obsahuje atribut `xmi:type` roven `uml:ActivityPartition`. V tomto elementu se vyskytují identifikátory uzlů, které přísluší k dané Activity partition. Příklad v XMI souboru je zobrazen ve výpisu kódu 3.14.

```
<group xmi:type="uml:ActivityPartition" xmi:id="EAI...21" name="Customer" >
    <node xmi:idref="EAI...7A"/>
    <node xmi:idref="EAI...65"/>
</group>
```

Výpis kódu 3.14 Activity partition v XMI souboru (Enterprise Architect)

3.4.2 Hierarchie elementů v grafové databázi

Pro možnost kvalitního vyhledávání je zapotřebí u jednotlivých elementů zachovat informaci o tom, které třídy je daný element instancí. Tím je myšleno i to, jakých všech tříd je daná třída podtřídou. To jsme se rozhodli vyřešit pomocí navržení hierarchického stromu tříd, které bude potřeba implementovat. Hierarchie tříd je zobrazena na obrázku A.1, který se z důvodu velikosti nachází v příloze.

Pomocí dědění je možné dosáhnou požadovaného výsledku, tedy například instance třídy InputPin je zároveň instancí třídy Pin, ObjectNode, ActivityNode a ActivityDiagramNode. Zároveň zdědí všechny atributy, které jsou obsaženy v jeho nadtřídách. Atributy, které jsou typu `relation`, vyjadřují to, že daná třída může být cílem nebo zdrojem daného typu relace. To slouží také jako částečné ověření pravidel, které se vztahují na možnost použití konkrétních typů relací, tedy které elementy mohou být propojeny danou relací a které nikoliv. Nejedná se však o validátor diagramů, takže odhalí pouze některé zjevné chyby, jako je například pokus o propojení Activity partition s jiným elementem pomocí relace Control flow.

3.4.3 Relace

Relace, které se vyskytují v Activity diagramech a jsou definované v UML specifikaci, jsou relace Control flow a Object flow. Control flow slouží k umožnění toku aktivity. Bez relací Control flow

by téměř nemělo smysl modelovat Activity diagramy. Začlenit **Control flow** do rozsahu práce nám tedy přišlo samozřejmé. Příklad reprezentace Control flow s atributem guard (strážce) je zobrazen ve výpisu kódu 3.15.

```
<edge xmi:type="uml:ControlFlow" source="EAI...89" target="EAI...45">
    <guard xmi:type="uml:OpaqueExpression" body="Yes"/>
</edge>
```

Výpis kódu 3.15 Control flow s atributem guard v XMI souboru (Enterprise Architect)

Object flow umožňuje tok dat a object tokenů. Jelikož jsme se rozhodli zapracovat vybrané Object nodes, tak by vynechání Object flow nedávalo smysl. Proto jsme **Object flow** zahrnuli do této práce. Příklad Object flow v XMI je zobrazen ve výpisu kódu 3.16.

```
<edge xmi:type="uml:ObjectFlow" source="EAI...17" target="EAI...39"\>
```

Výpis kódu 3.16 Object flow v XMI souboru (Enterprise Architect)

Obsahem práce je také element Activity partition. Abychom v grafové databázi zachovali vztah mezi Activity partition a elementy, které se nachází v Activitity partition, rozhodli jsme se zavést novou relaci, která bude právě mezi Activity partition a jednotlivými elementy. Relaci jsme se rozhodli pojmenovat **Partition member**. Relace vede od elementu, který je obsažen v Activity partition, do odpovídající Activity partition. Příklad situace, kdy je potřeba vytvořit relaci Partition member, je zobrazen ve výpisu kódu 3.14, kde Activity partition s atributem `name="Customer"` obsahuje dva uzly, které jsou definované podle atributu `xmi:idref`.

Jedním z Object nodes, které jsou obsahem práce, je element Pin. Pin se nejčastěji váže k elementu Action. Pro zachování toho, že Action má Pin, jsme se rozhodli zavést novou relaci, která bude mezi elementy Action a Pin. Relaci jsme pojmenovali **Has pin**. Relace vede od elementu Action do elementu Pin, který je buď vstupním, nebo výstupním Pin dané Action. V XMI souborech je Pin reprezentován buď jako XML element `input`, nebo jako XML element `output`, podle toho, zda se jedné o vstupní (input), nebo výstupní (output) Pin. Takový XML element je obsažen v XML elementu, který reprezentuje Action, ke které patří daný Pin. Příklad situace, kde je potřeba vytvořit relaci Has pin, je zobrazen ve výpisu kódu 3.17.

```
<node xmi:type="uml:Action" xmi:id="EAI...8E" name="Accept Payment">
    <input xmi:type="uml:InputPin" xmi:id="EAI...0B" ordering="FIFO"\>
</node>
```

Výpis kódu 3.17 Input pin v XMI souboru (Enterprise Architect)

3.5 Parsery

Cílem této práce je zpracovat Package diagramy a Activity diagramy, proto jsme se rozhodli rozdělit parsery na dvě skupiny. První skupinu tvoří parsery, které zpracovávají Package diagramy, ve druhé se nachází parsery, které zpracovávají Activity diagramy.

3.5.1 Package parsery

Vstupem Package parseru je soubor ve formátu XMI, který obsahuje model popisující systém z pohledu Packages. Všechny podporované modelovací nástroje obsahují možnost namodelovat systém z pohledu Packages a také tento model vyexportovat do formátu XMI. Z důvodu nekonzistence XMI souborů je potřeba pro každý modelovací nástroj implementovat vlastní Package parser. Z toho vyplývá, že je potřeba implementovat tři Package parsery. Konkrétně se jedná o **Enterprise Architect Package parser**, **Visual Paradigm Package parser** a **OpenPonk Package parser**.

3.5.2 Activity parsery

Vstupem Activity parseru je soubor ve formátu XMI, který obsahuje model popisující systém z pohledu aktivit. Modelovací nástroje Enterprise Architect a Visual Paradigm poskytují možnost modelování aktivit, naopak nástroj OpenPonk možnost modelování aktivit nenabízí. Z toho důvodu postačí implementovat pouze dva parsery. Konkrétně se jedná o **Enterprise Architect Activity parser** a **Visual Paradigm Activity parser**.

Kapitola 4
Implementace

Hlavním cílem práce je samotná implementace modulů parseru. Moduly parseru je potřeba integrovat do stávajícího projektu. V tomto projektu se již vyskytují Class diagram parsery pro nástroje Enterprise Architect a OpenPonk. Pro implementaci Package diagram parseru a Activity diagram parseru je nezbytné použít rozhraní stávajícího projektu. Všechny implementované parsery musí být podtřídou `DiagramParser`. Třída `DiagramParser` má deklarovanou jen jednu abstraktní metodu, a to metodu `parse_file`. Metoda `parse_file` přijímá jeden parametr `file_path`, který reprezentuje cestu ke XMI souboru, ve kterém se nachází diagram ke zpracování. Výstupem této metody je třída `Model` nebo její libovolná podtřída. Třída `Model` reprezentuje zpracovaný diagram. `Model` má deklarované metody `get_neo4j_model`, `get_nodes` a `get_relations`. Metoda `get_neo4j_model` je i definovaná a vrátí nodes (uzly) a relations (relace) v takové formě, že jsou připravené k nahrání do grafové databáze Neo4j. Samotné nahrání do databáze je implementováno ve stávajícím projektu, takže není cílem této práce. Metody `get_nodes` a `get_relations` jsou pouze deklarované, takže samotná implementace je na podtřídách třídy `Model`.

Podtřídy třídy `DiagramParser` jsou `PackageDiagramParser`, která se zaměřuje na zpracování Package diagramů, a `ActivityDiagramParser`, která slouží ke zpracování Activity diagramů.

4.1 PackageDiagramParser

Třída `PackageDiagramParser` slouží k definování rozhraní, co všechno je možné z Package diagramu zpracovat. Obsahuje metody, které nejsou pouze deklarované, ale jsou i definované, takže jejich implementace je pro všechny podtřídy společná, protože je nezávislá na jednotlivých modelovacích nástrojích. Jedná se o metody `parse_file`, `parse_model` a `get_namespaces`. Zmíněné metody jsou společné pro všechny podtřídy `PackageDiagramParser`.

Metoda `get_namespaces` slouží k získání namespaces ze XMI souboru. Vstupem je cesta ke XMI souboru a výstupem je seznam namespaces (jmenných prostorů). Metoda `parse_model` slouží ke zpracování modelu. Vstupem jsou namespaces a XML element, ve které se nachází model. Výstupem je instance třídy `PackageDiagramModel`. Třída `PackageDiagramModel` je podtřída třídy `Model`. Má definované atributy `id`, `nodes` (uzly) a `relations` (relace). Tato třída pomocí metody `get_neo4j_model` vrátí `nodes` a `relations` ve formě, která je vhodná pro nahrání do grafové databáze.

`PackageDiagramParser` obsahuje také metody, které jsou pouze deklarované, nikoliv definované. Jejich implementace je na konkrétních podtřídách, jelikož je různá v závislosti na konkrétním modelovacím nástroji. Jedná se tyto metody:

- `parse_nodes` – zpracuje a vrátí všechny nodes (uzly), které se nachází v modelu

- `parse_relations` – zpracuje a vrátí všechny relations (relace), které se nachází v modelu
- `parse_id` – vrátí id modelu
- `parse_packages` – zpracuje a vrátí všechny elementy, které jsou Package
- `parse_dependencies` – zpracuje a vrátí všechny Dependency relace
- `parse_merges` – zpracuje a vrátí všechny Package merge relace
- `parse_imports` – zpracuje a vrátí všechny Package import relace
- `parse_member_packages` – zpracuje a vrátí všechny Member of relace
- `parse_usages` – zpracuje a vrátí všechny Usage relace
- `get_model` – vrátí část XMI souboru, kde se nachází model

Modelovací nástroje se liší v tom, jaké elementy a relace umožňují namodelovat. Výsledkem je, že implementace konkrétních parserů nedefinuje všechny deklarované metody. Tomu, co je možné namodelovat a tedy má cenu implementovat, se věnují následující sekce.

4.1.1 EAPackageDiagramParser

`EAPackageDiagramParser` je podtřídou třídy `PackageDiagramParser`. Představuje parser, který zpracovává XMI soubory, které vznikly vyexportováním Package diagramů z modelovacího nástroje Enterprise Architect. Enterprise Architect umožňuje namodelovat téměř všechny elementy a relace, které jsou obsahem této práce a týkají se Package diagramů. Výjimkou je relace private Package import, která se značí klíčovým slovem «access». Public Package import je možné namodelovat pomocí Enterprise Architect. Z toho důvodu třída `EAPackageDiagramParser` definuje všechny metody, které jsou třídou `PackageDiagramParser` pouze deklarované ale nejsou definované, jen výstupem metody `parse_imports` nejsou všechny druhy relace Package import, ale pouze relace public Package import.

Třída `EAPackageDiagramParser` obsahuje také další pomocné metody, které slouží k jednoduššímu zpracování jednotlivých elementů a relací. Ke každé metodě, která zpracovává určitou skupinu elementů nebo relací, je definována metoda, která obstarává zpracování jednoho elementu nebo jedné relace. To znamená, že například k metodě `parse_packages` existuje metoda `parse_package`, která je zavolána pro každý element Package. Příklad metody `parse_package` je zobrazen ve výpisu kódu 4.1, kde vstupem je XML element reprezentující element Package a namespaces a výstupem je instance třídy `PackageNode`, která slouží pro následné nahrání do grafové databáze.

```
@staticmethod
def parse_package(package, namespaces):
    try:
        node_id = package.attrib["{" + namespaces['xmi'] + "}" + "id"]
        node_name = package.attrib["name"]
        node_class = "Package"
        node_visibility = package.attrib["visibility"]
        return PackageNode(node_name, node_id, node_class, node_visibility)
    except Exception as exc:
        raise AttributeError("Corrupted package node in source file")
                             from exc
```

Výpis kódu 4.1 Metoda parse_package (EAPackageDiagramParser)

4.1.2 VPPackageDiagramParser

`VPPackageDiagramParser` je podtřídou třídy `PackageDiagramParser`. Představuje parser, který zpracovává XMI soubory, které vznikly vyexportováním Package diagramů z modelovacího nástroje Visual Paradigm. Visual Paradigm umožňuje namodelovat téměř všechny elementy a relace, které jsou obsahem této práce a týkají se Package diagramů. Výjimkou je relace **Usage**, kterou není možné pomocí Visual Paradigm namodelovat. Z toho důvodu není definována metoda `parse_usage`. Všechny ostatní metody, které jsou deklarované ve třídě `PackageDiagramParser` jsou definované ve třídě `VPPackageDiagramParser`.

Třída `VPPackageDiagramParser` obsahuje také další pomocné metody, které slouží k jednoduššímu zpracování jednotlivých elementů a relací. Ke každé metodě, která zpracovává určitou skupinu elementů nebo relací, je definována metoda, která obstarává zpracování jednoho elementu nebo jedné relace. To znamená, že například k metodě `parse_imports` existuje metoda `parse_import`, která je zavolána pro každou relaci Package import. Příklad `parse_imports` je zobrazen ve výpisu kódu 4.2, kde vstupními parametry jsou `model`, který reprezentuje XML element, ve kterém se nachází model, a `namespaces`.

```
def parse_imports(self, model, namespaces):
    m_imports = set()
    imports = model.findall('.//packageImport', namespaces)
    for import_e in imports:
        m_imports.add(self.parse_import(import_e, namespaces))
    return m_imports
```

Výpis kódu 4.2 Metoda parse_imports (VPPackageDiagramParser)

4.1.3 OpenPonkPackageDiagramParser

`OpenPonkPackageDiagramParser` je podtřídou třídy `PackageDiagramParser`. Představuje parser, který zpracovává XMI soubory, které vznikly vyexportováním Package diagramů z modelovacího nástroje OpenPonk. OpenPonk oproti jiným modelovacím nástrojům nedisponuje rozsáhlou paletou elementů a relací, které se vyskytují v Package diagramech. OpenPonk umožňuje namodelovat elementy Package a Model a mezi nimi jedinou relaci, a to relaci public Package import. Z toho důvodu `OpenPonkPackageDiagramParser` nedefinuje metody `parse_dependencies`, `parse_merges` a `parse_usages`.

Třída `OpenPonkPackageDiagramParser` obsahuje také další pomocné metody, které slouží k jednoduššímu zpracování jednotlivých elementů a relací. Ke každé metodě, která zpracovává určitou skupinu elementů nebo relací, je definována metoda, která obstarává zpracování jednoho elementu nebo jedné relace. Jelikož OpenPonk exportuje XMI soubory téměř stejně jako modelovací nástroj Enterprise Architect, je implementace metod téměř stejná jako u třídy `EAPackageDiagramParser`. Kód metody `parse_relations` je zobrazen ve výpisu kódu 4.3. Tato metoda zpracovává všechny relace, které se v modelu nachází. Metodu uvádíme z důvodu, že se implementačně liší od metody se stejným jménem ze třídy `EAPackageDiagramParser`.

```
def parse_relations(self, model, namespaces):
    try:
        packages = self.get_packages(model, namespaces)
        m_relations = self.parse_imports(packages, namespaces)
        m_relations.update(self.parse_member_packages(packages, namespaces))
    except Exception as exc:
        raise Exception("Corrupted model in source file") from exc
    return m_relations
```

Výpis kódu 4.3 Metoda parse_relations (OpenPonkPackageDiagramParser)

4.1.4 PackageDiagramModel

`PackageDiagramModel` je podtřídou třídy `Model`. Instance třídy `PackageDiagramModel` je výstupem metod `parse_file` tříd `PackageDiagramParser`. Třída `PackageDiagramModel` obsahuje metodu `get_neo4j_model`, jejímž výstupem jsou `nodes` (uzly) a `relations` (relace), které je možné nahrát do grafové databáze Neo4j. Toho je docíleno tak, že třída v konstruktoru dostane set `nodes`, které jsou instancemi třídy `PackageNode`, a set `relations`, které jsou instancemi třídy `PackageRelation`. Třídy `PackageNode` a `PackageRelation` slouží jako pomocné třídy pro dosažení jednotné struktury daných prvků. Každý z těchto prvků má definovaný atribut `node_type` nebo `relation_type`. Podle tohoto atributu je následně vytvořen odpovídající objekt, který je následně možné nahrát do Neo4j. Obsahem práce jsou pouze elementy typu Package, z toho důvodu je implementována pouze třída `Package`, která je podtřídou třídy `StructuredNode`. `StructuredNode` je třída definována v balíčku neomodel a umožňuje nahrání dat do grafové databáze. Propojení jednotlivých elementů obstarává třída `Relationship`, která je také definována balíčkem neomodel.

4.2 ActivityDiagramParser

Třída `ActivityDiagramParser` slouží k definování rozhraní, co všechno je možné z Activity diagramu zpracovat. Obsahuje metody, které nejsou pouze deklarované, ale i definované, takže jejich implementace je pro všechny podtřídy stejná. Je tomu tak z důvodu, že jejich implementace je nezávislá na jednotlivých modelovacích nástrojích. Konkrétně se jedná o metody `parse_file`, `parse_model` a `get_namespaces`. Tyto metody jsou společné pro všechny podtřídy `ActivityDiagramParser`.

Metoda `get_namespaces` slouží k získání namespaces (jmenných prostorů) ze XMI souboru. Vstupem je cesta ke XMI souboru a výstupem je seznam `namespaces`. Metoda `parse_model` slouží ke zpracování modelu. Vstupem jsou namespaces a XML element, ve kterém je definovaný model. Výstupem je instance třídy `ActivityDiagramModel`. Třída `ActivityDiagramModel` je podtřída třídy `Model`. Má definované atributy `id`, `nodes` (uzly) a `relations` (relace). Tato třída pomocí metody `get_neo4j_model` vrátí `nodes` a `relations` ve formě, která je vhodná pro nahrání do grafové databáze.

`ActivityDiagramParser` obsahuje také metody, které jsou pouze deklarované. Jejich implementace je na konkrétních podtřídách, jelikož je různá v závislosti na konkrétním modelovacím nástroji. Jedná se tyto metody:

- `parse_nodes` – zpracuje a vrátí všechny nodes (uzly), které se nachází v modelu
- `parse_relations` – zpracuje a vrátí všechny relations (relace), které se nachází v modelu
- `parse_id` – vrátí id modelu
- `get_model` – vrátí část XMI souboru, kde se nachází model
- `parse_actions` – zpracuje a vrátí všechny Action elementy
- `parse_initial_nodes` – zpracuje a vrátí všechny Initial node elementy
- `parse_activity_finals` – zpracuje a vrátí všechny Activity final node elementy
- `parse_flow_finals` – zpracuje a vrátí všechny Flow final node elementy
- `parse_fork_joins` – zpracuje a vrátí všechny Fork node a Join node elementy
- `parse_decisions_merges` – zpracuje a vrátí všechny Decision node a Merge node elementy
- `parse_control_flows` – zpracuje a vrátí všechny Control flow relace

- `parse_object_flows` – zpracuje a vrátí všechny Object flow relace
- `parse_partitions` – zpracuje a vrátí všechny Partition elementy
- `parse_partition_relations` – zpracuje a vrátí všechny Partition member relace
- `parse_central_buffers` – zpracuje a vrátí všechny Central buffer node elementy
- `parse_pins` – zpracuje a vrátí všechny Pin elementy
- `parse_data_stores` – zpracuje a vrátí všechny Data store elementy

Modelovací nástroje Enterprise Architect a Visual Paradigm umožňují namodelovat všechny elementy a relace, které jsou obsahem této práce. Přesto je potřeba implementovat pro každý modelovací nástroj vlastní parser, jelikož modelovací nástroje popisují model rozdílnými způsoby.

4.2.1 EAActivityDiagramParser

`EAActivityeDiagramParser` je podtřídou třídy `ActivityDiagramParser`. Představuje parser, který zpracovává XMI soubory, které vznikly vyexportováním Activity diagramů z modelovacího nástroje Enterprise Architect. Enterprise Architect umožňuje namodelovat všechny elementy a relace, které jsou obsahem této práce a týkají se Activity diagramů. Z toho důvodu třída `EAActivityDiagramParser` deklaruje všechny metody, které jsou v `ActivityDiagramParser` pouze definované.

Třída `EAActivityDiagramParser` obsahuje také další pomocné metody. Důležitou pomocnou metodou je metoda `parse_activity_node`, která obstarává zpracování jednoho konkrétního elementu. Na vstupu dostane daný element ve formátu XML elementu a jeho typ. Výstupem je instance třídy `ActivityNode`, která slouží pro umožnění následného nahrání do grafové databáze. Při nenalezení atributů, které jsou povinné pro každý uzel, vyhodí metoda výjimku. Kód metody je zobrazen ve výpisu kódu 4.4.

```
@staticmethod
def parse_activity_node(node, namespaces, n_type):
    try:
        node_id = node.attrib["{" + namespaces['xmi'] + "}" + "id"]
        if 'name' in node.attrib and node.attrib['name'] is not "":
            node_name = node.attrib["name"]
        else:
            node_name = None
        if 'ordering' in node.attrib:
            node_ordering = node.attrib["ordering"]
        else:
            node_ordering = None
        node_type = n_type
        node_visibility = node.attrib["visibility"]
        return ActivityNode(name=node_name,
                            node_id=node_id,
                            node_type=node_type,
                            visibility=node_visibility,
                            ordering=node_ordering)
    except Exception as exc:
        raise AttributeError("Corrupted activity node in source file:
                             type " + n_type) from exc
```

Výpis kódu 4.4 Metoda parse_activity_node (EAActivityDiagramParser)

4.2.2 VPActivityDiagramParser

`VPActivityDiagramParser` je podtřídou třídy `ActivityDiagramParser`. Představuje parser, který zpracovává XMI soubory, které vznikly vyexportováním Activity diagramů z modelovacího nástroje Visual Paradigm. Visual Paradigm umožňuje namodelovat všechny elementy a relace, které jsou obsahem této práce a týkají se Activity diagramů. Z toho důvodu třída `VPActivityDiagramParser` deklaruje všechny metody, které jsou v `ActivityDiagramParser` pouze definované.

Třída `VPActivityDiagramParser` obsahuje také další pomocné metody, které jsou často implementačně podobné nebo dokonce stejné jako jako u `EAActivityDiagramParser`. Například obsahuje metodu `parse_activity_node`, která je totožná z implementačního hlediska jako metoda u `EAActivityDiagramParser`. Kód metody je zobrazen ve výpisu kódu 4.4. Jednou z metod, která používá uvedenou metodu `parse_activity_node`, je metoda `parse_actions`. Úkolem této metody je najít a zpracovat všechny elementy, které jsou typu Action. Kód metody je zobrazen ve výpisu kódu 4.5.

```
def parse_actions(self, model, namespaces):
    m_actions = set()
    actions = set(model.findall(
                  './/ownedMember[@xmi:type="uml:CallBehaviorAction"]',
                  namespaces))
    actions.update(model.findall(
                  './/node[@xmi:type="uml:CallBehaviorAction"]',
                  namespaces))
    for action in actions:
        m_actions.add(self.parse_activity_node(action,
                                               namespaces,
                                               "Action"))
    return m_actions
```

Výpis kódu 4.5 Metoda parse_actions (VPActivityDiagramParser)

4.2.3 ActivityDiagramModel

`ActivityDiagramModel` je podtřídou třídy `Model`. Instance třídy `ActivityDiagramModel` je výstupem metod `parse_file` tříd `ActivityDiagramParser`. Třída `ActivityDiagramModel` obsahuje metodu `get_neo4j_model`, jejímž výstupem jsou `nodes` (uzly) a `relations` (relace), které je možné nahrát do grafové databáze Neo4j. Toho je docíleno tak, že třída v konstruktoru dostane set `nodes`, které jsou instancemi třídy `ActivityNode`, a set `relations`, které jsou instancemi třídy `ActivityRelation`. Třídy `ActivityNode` a `ActivityRelation` slouží jako pomocné třídy pro dosažené jednotné struktury daných prvků. Každý z `nodes` má definovaný atribut `node_type`. Podle tohoto atributu je následně vytvořena instance dané třídy z obrázku A.1, kterou je následně možné nahrát do Neo4j. Podobný princip je u `relations`, které mají atribut `relation_type`, který určuje, o jaký druh relace se jedná.

4.3 Řešení chyb

Všechny implementované parsery reagují při nalezení chyby stejným způsobem a to vyhozením výjimky (Exception). Jelikož se jedná o moduly, které slouží k rozšíření stávajícího projektu, tak řešení výjimek není obsahem této práce a je čistě na projektu, který bude moduly používat. Jak je patrné například u kódu 4.4, tak metoda je zabalena do `try`, `except` bloku, což je standardní postup při odchytávání výjimek v jazyce Python.

Kapitola 5
Testování

Testování je nedílnou součástí vývoje software. Díky testování je možné kód udržovat, rozšiřovat a do jisté míry je díky němu možné zaručit požadované a správné fungování software. Existuje velké množství testů. Jedním z nejčastějších druhů testů jsou **unit testy**. Ty se zaměřují na nejmenší testovatelné části software. V tomto případě se jedná o metody jednotlivých tříd. Pro unit testy je použit Python framework pro testování unit testů – unittest. S jeho pomocí jsou implementovány všechny unit testy.

Dalším druhem testů jsou **integrační testy**. Integrační testy se zaměřují na správné fungování celku, který je tvořen jednotlivými částmi systému. Testuje se například, jestli části sytému mezi sebou správně komunikují. Jedním z cílů této práce je integrovat implementované moduly do projektu Exquiro. Z toho důvodu jsou integrační testy nezbytné, protože je potřeba otestovat, že moduly parseru správně spolupracují s projektem Exquiro. Tyto testy jsou provedeny pomocí uživatelských testů.

5.1 Unit testy

5.1.1 Package diagram parsers

Vstupem Package diagram parser je soubor ve formátu XMI. Je důležité, jakým modelovacím nástrojem soubor vznikl. Pokud vznikl z nástroje Enterprise Architect, tak je nutné pro zpravování souboru použít `EAPackageDiagramParser`. Stejně tomu je i u modelovacích nástrojů Visual Paradigm a OpenPonk. Z toho důvodu jsme si připravili testovací soubory, které vznikly vyexportováním validních Package diagramů z jednotlivých modelovacích nástrojů.

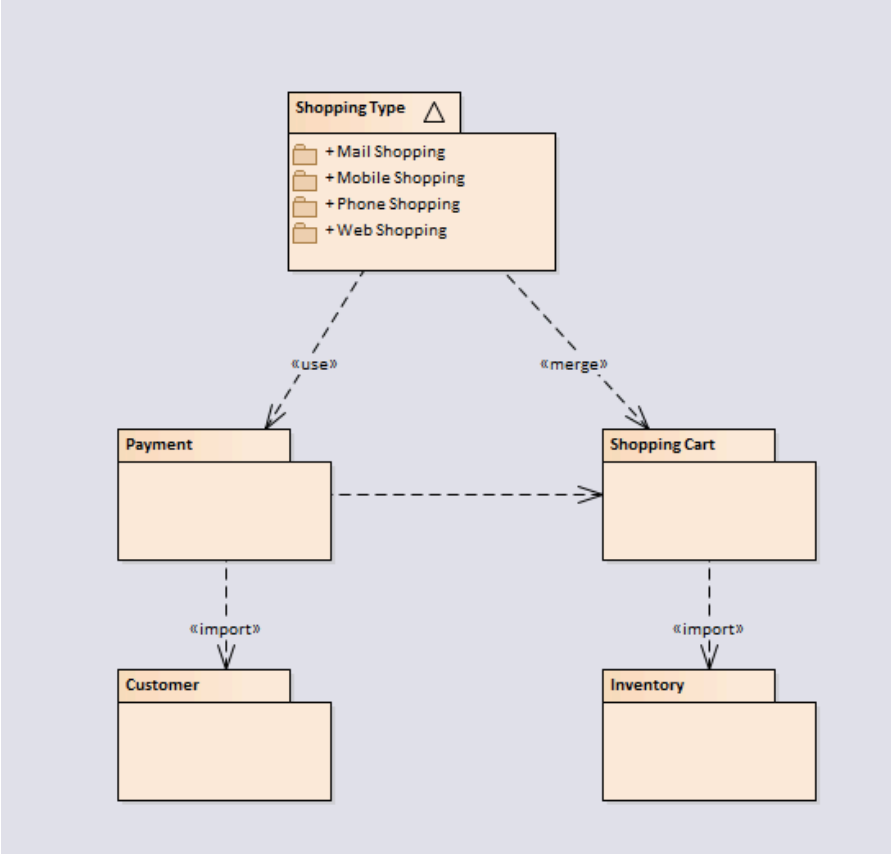
Unit testy jsme obecně rozdělili na dvě skupiny testů. Do první skupiny testů jsme zařadili ty testy, kdy vstupem jednotlivých metod jsou validní data. V takovém případě potom porovnáváme výstupy metod s očekávanou hodnotou. Druhou skupinu testů tvoří naopak testy, které mají na vstupu data, která nejsou validní. U této skupiny zkoumáme, zda metoda zareaguje na chybu očekávaným způsobem. Nejčastější reakcí při chybě je vyhození výjimky. Konkrétně jsme například testovali, zda výsledný zpracovaný model obsahuje správný počet jednotlivých elementů. Dále jsme testovali, zda se zachovaly všechny relace a jestli relace jsou stále mezi stejnými elementy, jako je tomu v diagramu. Dalším důležitým testem, který jsme provedli, je test, zda všechny elementy a relace mají různé ID. Kdyby tomu tak nebylo, nešlo by je v grafové databázi od sebe rozeznat. Kód testu unikátních ID je zobrazen na 5.1. Tento test je prováděn všemi moduly parseru. Pro otestování modulů parseru, které zpracovávají Package diagramy, jsme implementovali celkem 128 unit testů.

```
def test_unique_ids(self):
    relations = self.parser.parse_relations(self.model, self.namespaces)
    nodes = self.parser.parse_nodes(self.model, self.namespaces)
    ids = set()
    for relation in relations:
        ids.add(relation.id)
    for node in nodes:
        ids.add(node.id)
    self.assertEqual(len(ids), len(relations) + len(nodes))
```

Výpis kódu 5.1 Test unikátních ID

5.1.1.1 Enterprise Architect package parser

K testování `EAPackageDiagramParser` na úrovni unit testů je zapotřebí diagram, který obsahuje všechny elementy, které `EAPackageDiagramParser` dokáže zpracovat. Pro tento účel jsme v modelovacím nástroji Enterprise Architect namodelovali Package diagram zobrazený na obrázku 5.1.

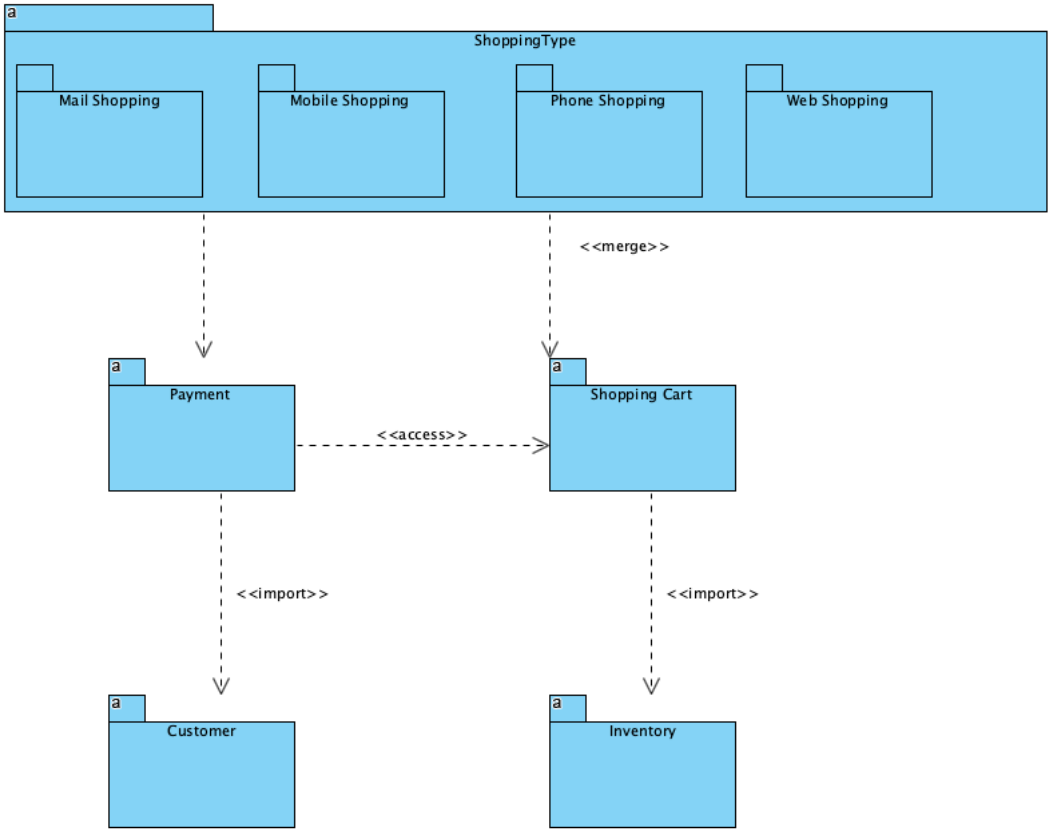


Obrázek 5.1 Obchodní systém (Enterprise Architect)

Modelovací nástroj Enterprise Architect umožňuje namodelovat všechny elementy a téměř všechny relace. Z toho důvodu je počet unit testů relativně vysoký. K otestování jsme implementovali 51 unit testů.

5.1.1.2 Visual Paradigm package parser

K testování `VPPackageDiagramParser` na úrovni unit testů je zapotřebí diagram, který obsahuje všechny elementy, které `VPPackageDiagramParser` dokáže zpracovat. Pro tento účel jsme namodelovali v modelovacím nástroji Visual Paradigm Package diagram zobrazený na obrázku 5.2.



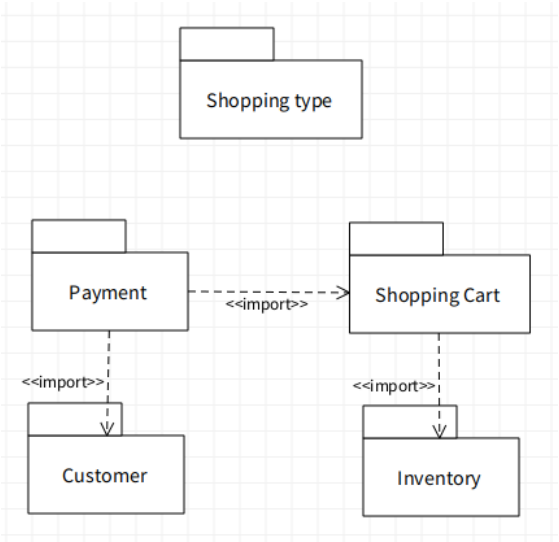
Obrázek 5.2 Obchodní systém (Visual Paradigm)

Modelovací nástroj Visual Paradigm umožňuje namodelovat téměř všechny relace a všechny elementy. Z toho důvodu je počet unit testů podobný jako u Enterprise Architect. K otestování jsme implementovali 46 unit testů.

5.1.1.3 OpenPonk package parser

K testování `OpenPonkPackageDiagramParser` na úrovni unit testů je zapotřebí diagram, který obsahuje všechny elementy, které `OpenPonkPackageDiagramParser` dokáže zpracovat. Pro tento účel jsme namodelovali v modelovacím nástroji OpenPonk Package diagram zobrazený na obrázku 5.3.

Modelovací nástroj OpenPonk umožňuje namodelovat všechny elementy. Na druhou stranu neumožňuje namodelovat většinu relací. Z toho důvodu je počet unit testů relativně nízký. K otestování jsme implementovali 31 unit testů.



Obrázek 5.3 Obchodní systém (OpenPonk)

5.1.2 Activity diagram parsers

Vstupem Activity diagram parser je soubor ve formátu XMI. Je důležité, jakým modelovacím nástrojem soubor vznikl. Pokud vznikl z nástroje Enterprise Architect, tak je nutné pro zpravování souboru použít `EAActivityDiagramParser`, pokud vznikl z modelovacího nástroje Visual Paradigm, je nutné použít `VPActivityDiagramParser`. Z toho důvodu jsme si připravili testovací soubory, které vznikly vyexportováním validních Activity diagramů z jednotlivých modelovacích nástrojů.

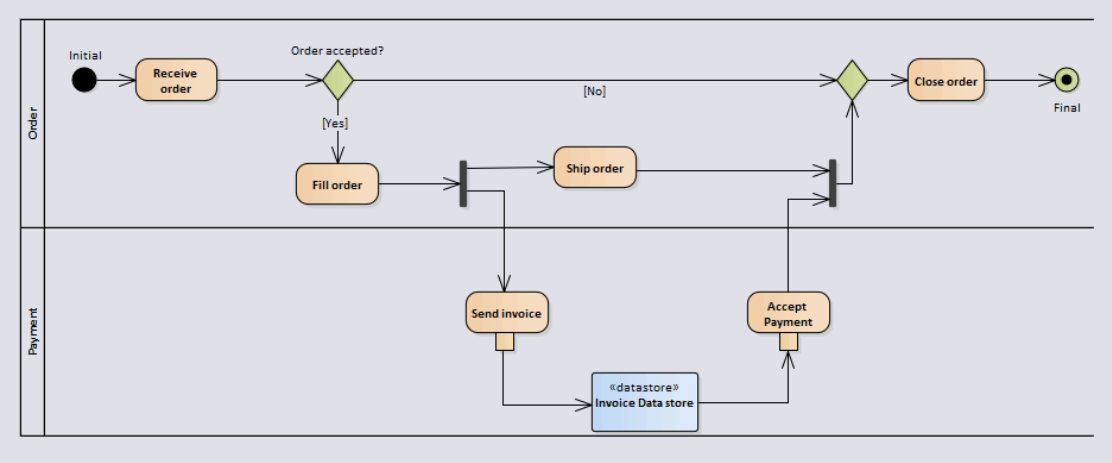
Unit testy jsme obecně rozdělili stejně jako u Package diagramů na dvě skupiny testů. Do první skupiny testů jsme zařadili ty testy, ve kterých jsou vstupem metod validní data. V takovém případě potom porovnáváme výstupy metod s očekávanou hodnotou. Druhou skupinu testů tvoří naopak testy, které mají na vstupu data, která validní nejsou. U této skupiny zkoumáme, zda metoda zareaguje na chybu očekávaným způsobem. Nejčastější reakcí při chybě je vyhození výjimky. Konkrétně jsme například testovali, zda výsledný zpracovaný model obsahuje správný počet jednotlivých elementů. Dále jsme testovali, zda se zachovaly všechny relace a jestli relace jsou stále mezi stejnými elementy, jako je tomu v diagramu. Dalším důležitým testem, který jsme provedli, je test, zda všechny elementy a relace mají různé ID. Kdyby tomu tak nebylo, nešlo by je v grafové databázi od sebe rozeznat. Kód testu unikátních ID je totožný jako u Package diagram parserů a je zobrazen na 5.1.

Pro otestování modulu parseru, které zpracovávají Activity diagramy, jsme implementovali 126 unit testů.

5.1.2.1 Enterprise Architect activity parser

K testování `EAActivityDiagramParser` na úrovni unit testů je zapotřebí diagram, který obsahuje všechny elementy, které `EAActivityDiagramParser` dokáže zpracovat. Rozhodli jsme se pro srozumitelnost diagramů namodelovat místo jednoho diagramu diagramů více. Primární diagram, ze kterého vychází většina testů, je zobrazen na obrázku 5.4.

Diagram na obrázku 5.4 neobsahuje element Central buffer node a element Flow final node. Pro otestování práce s těmito elementy jsou použity jiné diagramy, které nejsou obsahově nijak zajímavé a jsou namodelované pouze pro účely testování.

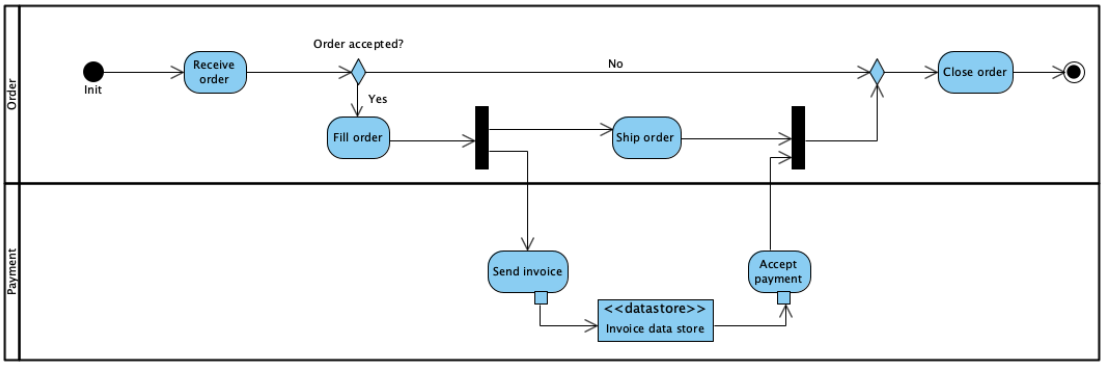


Obrázek 5.4 Aktivita vyřízení objednávky (Enterprise Architect)

Modelovací nástroj Enterprise Architect umožňuje namodelovat všechny elementy a relace, které jsou obsahem této práce. K otestování jsme implementovali 64 unit testů.

5.1.2.2 Visual Paradigm activity parser

K testování `VPActivityDiagramParser` na úrovni unit testů je zapotřebí diagram, který obsahuje všechny elementy, které `VPActivityDiagramParser` dokáže zpracovat. Rozhodli jsme se pro srozumitelnost diagramů namodelovat místo jednoho diagramu diagramů více. Primární diagram, ze kterého vychází většina testů, je zobrazen na obrázku 5.5.



Obrázek 5.5 Aktivita vyřízení objednávky (Visual Paradigm)

Diagram na obrázku 5.5 neobsahuje element Central buffer node a element Flow final node. Pro otestování práce s těmito elementy je použit jiný diagram, který není obsahově nijak zajímavý a je namodelován pouze pro účely testování.

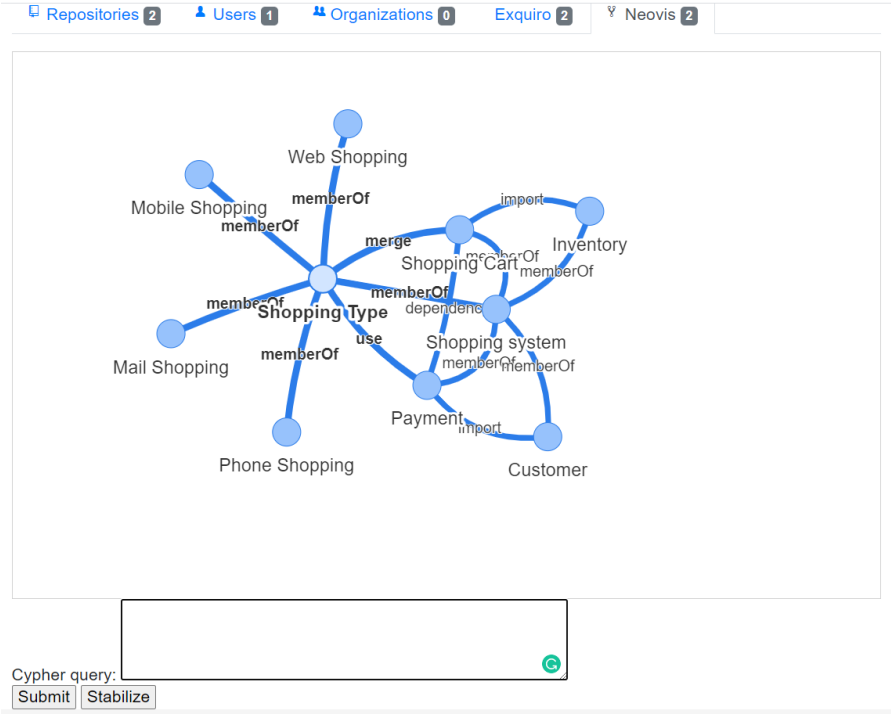
Modelovací nástroj Visual Paradigm umožňuje namodelovat všechny elementy a relace, které jsou obsahem této práce. K otestování jsme implementovali 62 unit testů.

5.2 Integrační testy

Jedním z cílů této práce je integrovat moduly parseru do projektu Exquiro, který je součástí aplikace Repocribro. Integrační testy, které ověřují úspěšnost samotné integrace, jsme se rozhodli provést pomocí uživatelských testů. V aplikaci Repocribro je možné nahrát soubor ve formátu XMI, který reprezentuje vyexportovaný diagram. Jelikož implementované moduly parseru zpracovávají dva druhy diagramů, je potřeba rozdělit integrační testy na ty, které testují zpracování Package diagramů, a na testy, které testují zpracování Activity diagramů.

5.2.1 Package diagram

Pro otestování Package diagramů jsme použili soubor XMI, který vznikl vyexportováním diagramu, který je zobrazen na obrázku 5.1. Výsledná reprezentace diagramu v aplikaci Repocribro je zobrazena na obrázku 5.6.



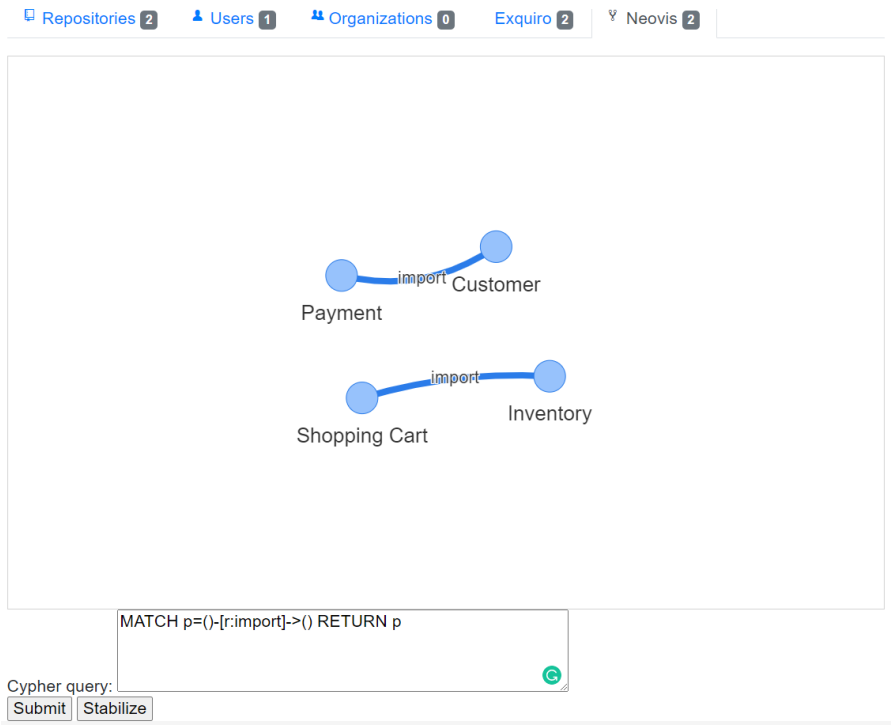
Obrázek 5.6 Obchodní systém (Repocribro)

Struktura grafu v aplikaci Repocribro z obrázku 5.6 odpovídá očekávané struktuře. Všechny elementy a relace z původního diagramu zůstaly zachovány. K nim se přidaly relace, které bylo nutné přidat pro zachování vztahů mezi jednotlivými elementy.

Hlavní důvod samotného zpracování diagramů je umožnit vyhledávání v těchto diagramech. Pro otestování vyhledávání jsme zvolili dotaz 5.2, který nalezne všechny uzly, které jsou spojené relací import. Výsledek dotazu v aplikaci Repocribro je zobrazen na obrázku 5.7. Grafová reprezentace dotazu se shoduje s očekávaným výsledkem a můžeme tedy prohlásit, že integrace modulu parseru pro Package diagram byla úspěšná.

```
MATCH p=()-[r:import]->() RETURN p
```

Výpis kódu 5.2 Dotaz pro vyhledání uzlů spojené relací import



Obrázek 5.7 Obchodní systém – relace import (Repocribro)

5.2.2 Activity diagram

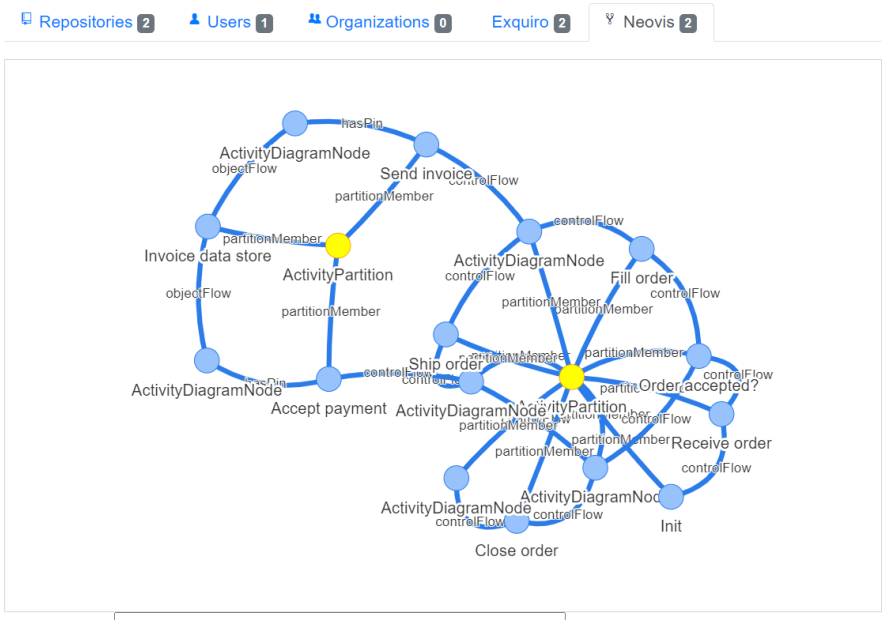
Pro otestování Activity diagramů jsme použili soubor XMI, který vznikl vyexportováním diagramu, který je zobrazen na obrázku 5.5. Výsledná reprezentace diagramu v aplikaci Repocribro je zobrazena na obrázku 5.6.

Zpracovávaný diagram je rozsáhlejší než diagram z příkladu 5.2.1. Po podrobném prozkoumání jsme ale zjistili, že i v tomto případě obsahuje všechny elementy a relace, které obsahoval zpracovávaný diagram. Zároveň mezi uzly, mezi kterými měla být vytvořena nová relace pro zachování vztahů, byla zavedena potřebná nová relace.

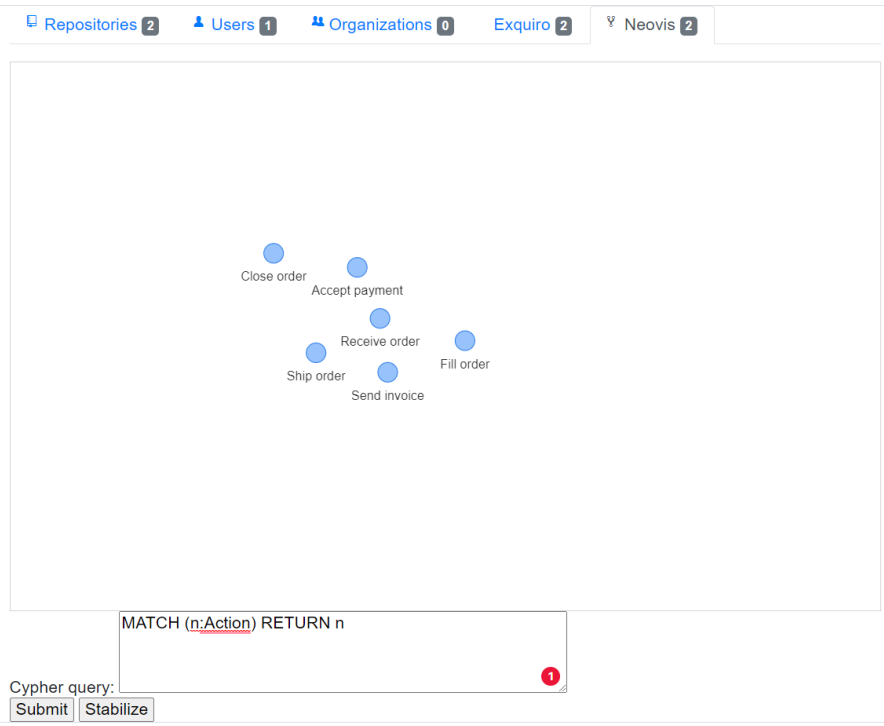
Hlavní důvod samotného zpracování diagramů je umožnit vyhledávání v těchto diagramech. Pro otestování vyhledávání jsme zvolili dotaz 5.3, který nalezne všechny uzly, které jsou typu Action. Výsledek dotazu je zobrazen na obrázku 5.9. Aktivita, která je zobrazena na obrázku 5.5, obsahuje přesně šest Actions, což je přesně stejný počet, jako je zobrazen na obrázku 5.9. Můžeme tedy prohlásit, že vyhledávání v grafové databázi funguje a integrace modulu parseru, který zpracovává Activity diagramy, byla úspěšná.

```
MATCH (n:Action) RETURN n
```

Výpis kódu 5.3 Dotaz pro vyhledání Actions



Obrázek 5.8 Vyřízení objednávky (Repocribro)



Obrázek 5.9 Vyřízení objednávky – Actions (Repocribro)

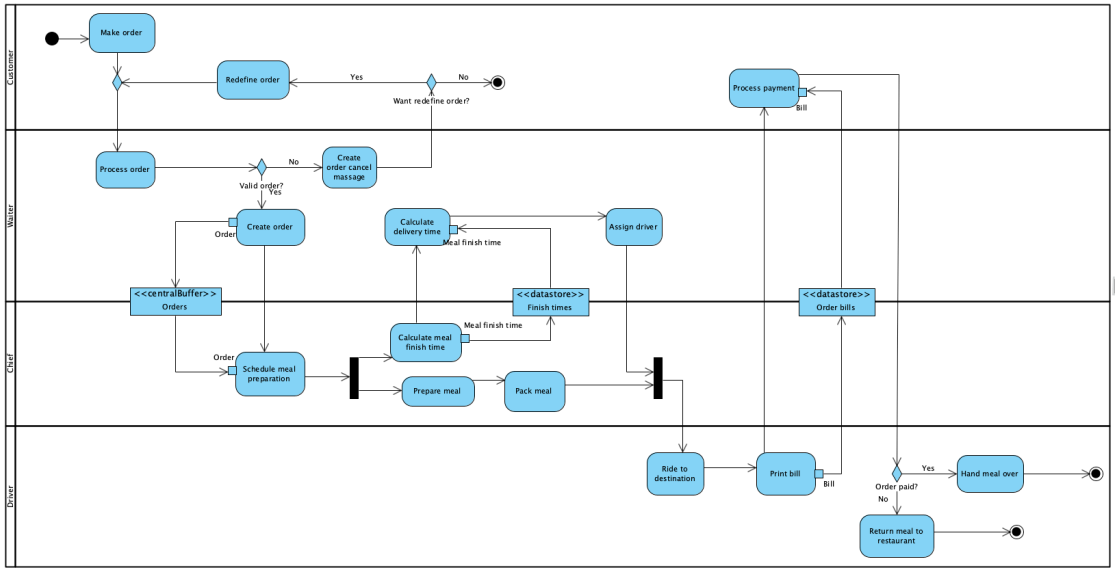
Kapitola 6

Případová studie

Výsledkem této práce je rozšíření stávajícího projektu Exquiro o možnost zpracování Package diagramů a Activity diagramů. V této kapitole je demonstrována funkčnost implementovaných modulů parseru na konkrétních příkladech. Funkčností parseru je myšleno, že dokáže zpracovat vyexportované UML diagramy tak, že je v nich možné vyhledávat. Vyhledávání může sloužit buď k nalezení podobností mezi jednotlivými modely, nebo k hledání zajímavých částí v konkrétním modelu. Zde je demonstrováno vyhledávání zajímavých částí v konkrétním modelu. Výsledky dotazů v grafové databázi jsou zobrazeny v aplikaci Neo4j Desktop [19].

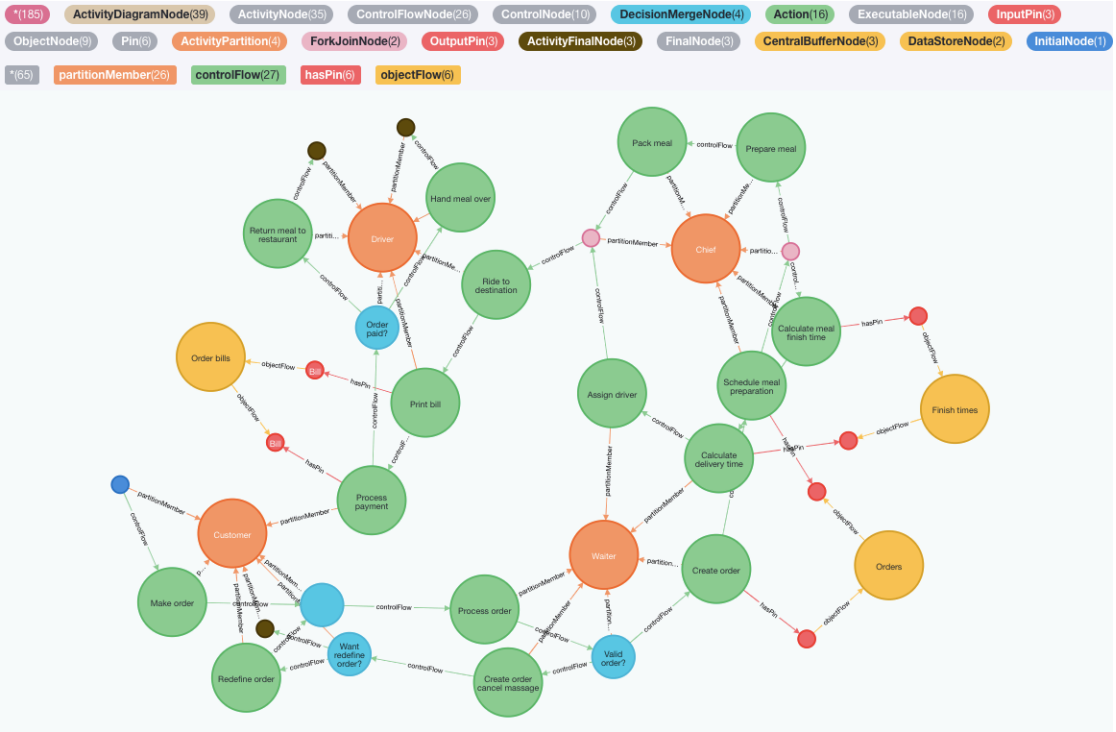
6.1 Activity diagram

Pro demonstraci vyhledávání v Activity diagramu jsme v modelovacím nástroji Visual Paradigm namodelovali diagram, který je zobrazen na obrázku 6.1. Tento diagram dokumentuje průběh vyřízení telefonní objednávky v restauraci.



Obrázek 6.1 Activity diagram – telefonní objednávka v restauraci (Visual Paradigm), ve větší velikosti v příloze na obrázku A.2

Po zpracování XMI souboru, který vznikl vyexportováním diagramu na obrázku 6.1, a nahrání do grafové databáze Neo4j jsme dostali reprezentaci diagramu v grafové databázi, která je uvedena na obrázku 6.2.



Obrázek 6.2 Telefonní objednávka v restauraci (grafová databáze), ve větší velikosti v příloze na obrázku A.3

Je patrné, že samotné převedení diagramu do grafové databáze neznamená zpřehlednění daného diagramu. Pro přehlednost slouží samotný diagram. Grafová databáze však umožňuje vyhledávání, čímž je možné zobrazit pouze žádoucí uzly a hrany. Vyhledávání je prováděno pomocí dotazovacího jazyka Cypher.

6.1.1 Vyhledání Actions podle Activity partition

V tomto příkladě je demonstrováno vyhledání všech Actions, které patří do konkrétní Activity partition. Výsledek dotazu 6.1 je zobrazen na obrázku A.4, který se nachází v příloze.

```
MATCH (n:Action)-[rel:partitionMember]->(:ActivityPartition{name:"Waiter"})
RETURN n
```

Výpis kódu 6.1 Dotaz pro vyhledání Action, které patří do Activity partition se jménem Waiter

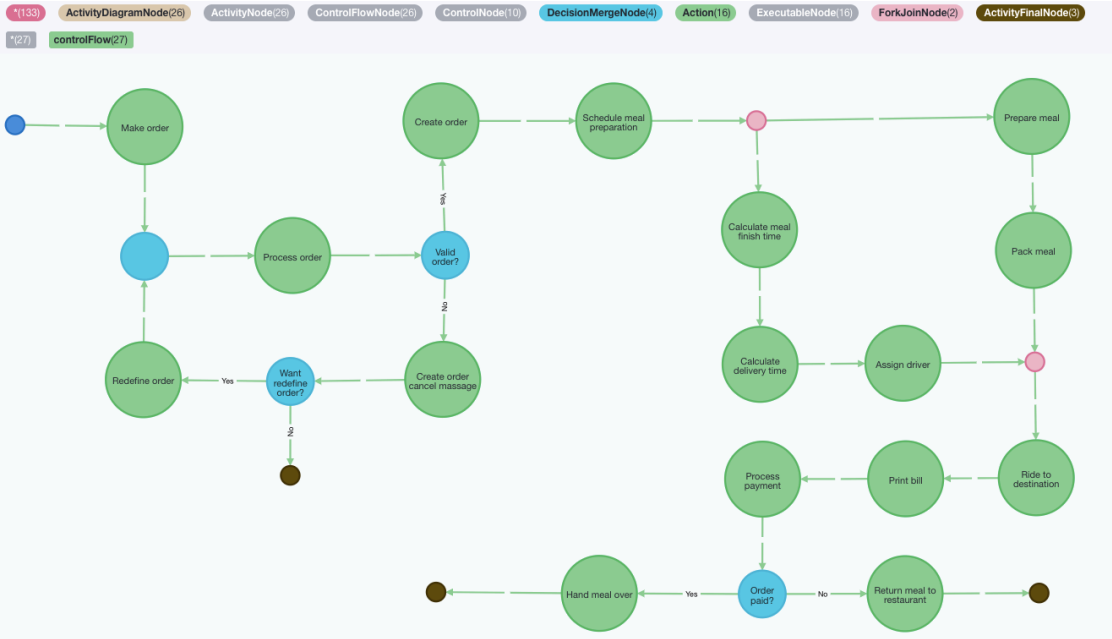
Výsledek nám ukazuje, které všechny Actions jsou přiřazeny účastníkovi aktivity s rolí `Waiter` (číšník). Případ použití tohoto dotazu může být, pokud si daná organizační jednotka (Activity partition) chce vyhledat všechny Actions, které má na starosti. Také si můžeme všimnout, že pokud mezi vyhledanými Actions byla libovolná relace, tak je tato relace zachována. Zde je patrné, že v aktivitě po Action se jménem `Calculate delivery time` následuje Action se jménem `Assign driver`. Ostatní Actions nejsou spojené žádnou relací, tudíž v toku aktivity nenásledují přímo za sebou.

6.1.2 Vyhledání Control flow

Jedním z příkladů, které jsme připravili, je vyhledání uzlů, které spojuje relace Control flow (řídící tok). Dotaz, pomocí kterého je provedeno vyhledání, je zobrazen ve výpisu kódu 6.2. Výsledek dotazu je zobrazen na obrázku 6.3.

```
MATCH (n:ActivityDiagramNode)<-[rel:controlFlow]->(:ActivityDiagramNode)
RETURN n
```

Výpis kódu 6.2 Dotaz pro vyhledání Control flow



Obrázek 6.3 Telefonní objednávka v restauraci – Control flow (grafová databáze), ve větší velikosti v příloze na obrázku A.5

V levém horním rohu obrázku 6.3 se nachází malé modré kolečko, které představuje Initial node. Zde začíná tok aktivity a pokračuje ve směru šipek (hran), dokud nedorazí do libovolného hnědého kolečka, která představují Activity final nodes. V nich vykonání aktivity končí. Vyhledáním Control flow odstraníme z grafu všechny Object nodes a Activity partition. Výsledkem dotazu je zobrazení posloupnosti Actions podle toho, jak jsou v čase vykonány.

6.1.3 Vyhledání uzlů podle podmínky Decision node

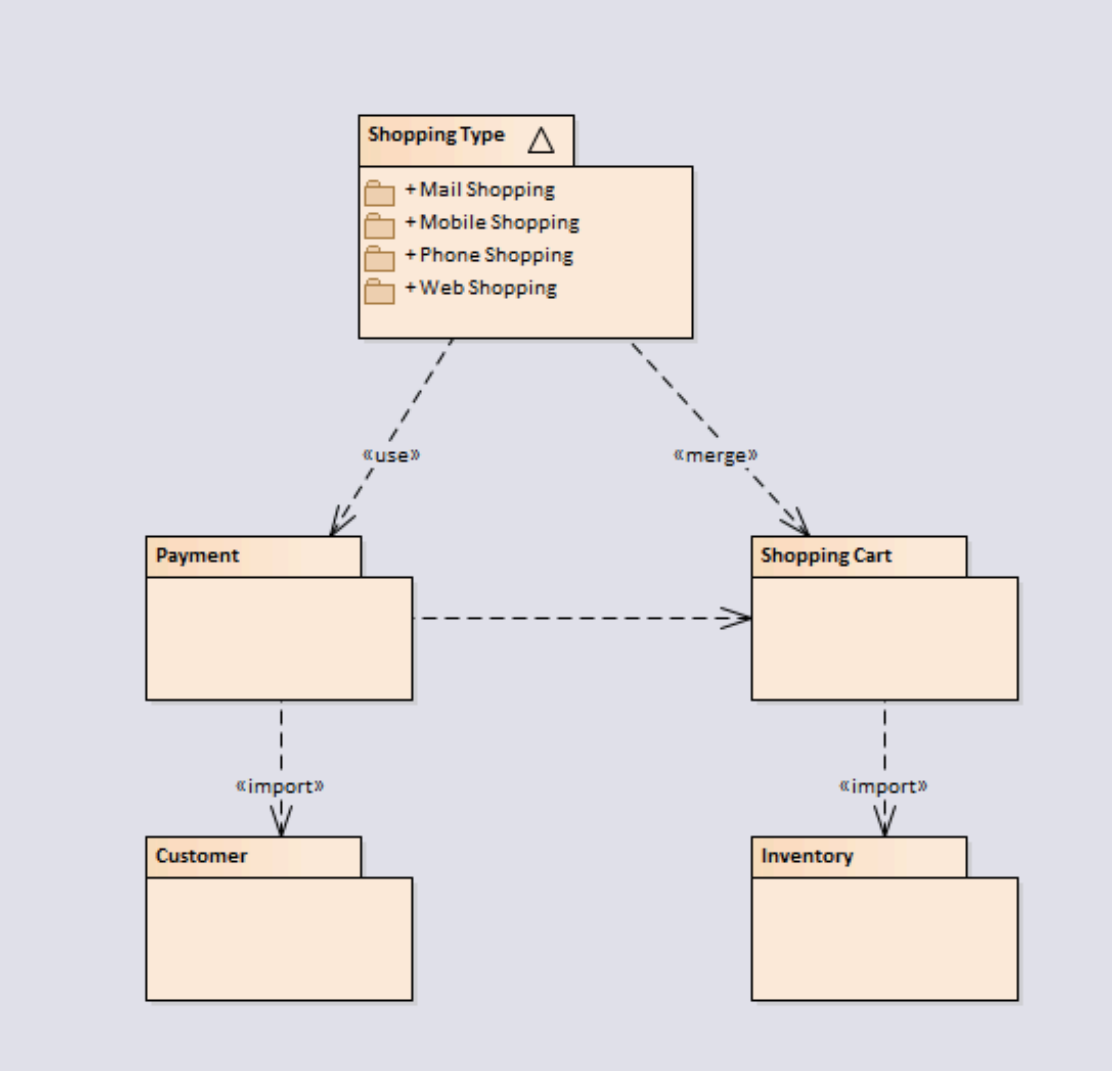
Dotazovací jazyk Cypher umožňuje i složitější dotazování. Dotaz 6.3 vyhledá všechny Decision node, které jsou zdrojem relace controlFlow s atributem `guard`, který je roven hodnotě `Yes`. Dotaz vrátí jak Decision nodes, tak i uzly, které jsou cílem relace controlFlow. Výsledek dotazu je zobrazen na obrázku A.6, který se nachází v příloze.

```
MATCH (n:DecisionMergeNode)-
[rel:controlFlow{guard:"Yes"}]->(r:ActivityDiagramNode)
RETURN n,r
```

Výpis kódu 6.3 Dotaz pro vyhledání uzlů podle podmínky Decision node

6.2 Package diagram

Pro demonstraci vyhledávání v Package diagramu jsme v modelovacím nástroji Enterprise Architect namodelovali diagram, který je zobrazen na obrázku 6.4. Nejedná se o složitý Package diagram, na druhou stranu v něm jsou reprezentovány všechny elementy a relace, které je možné namodelovat pomocí nástroje Enterprise Architect a jsou obsahem této práce.



Obrázek 6.4 Package diagram – obchodní systém (Enterprise Architect)

Po zpracování XMI souboru, který vznikl vyexportováním diagramu na obrázku 6.4, a nahrání do grafové databáze Neo4j jsme dostali reprezentaci diagramu v grafové databázi, která je uvedena na obrázku 6.5.

Obrázek 6.5 Obchodní systém (grafová databáze)

Jelikož se jedná o poměrně jednoduchý diagram, reprezentace v grafové databázi je celkem přehledná a do jisté míry čitelná. U složitějších diagramů by však reprezentace v grafové databázi přestala být přehledná, jelikož jejím smyslem není diagram zpřehlednit, ale umožnit v něm vyhledávat. Přehlednou vizualizaci obstarává samotný diagram.

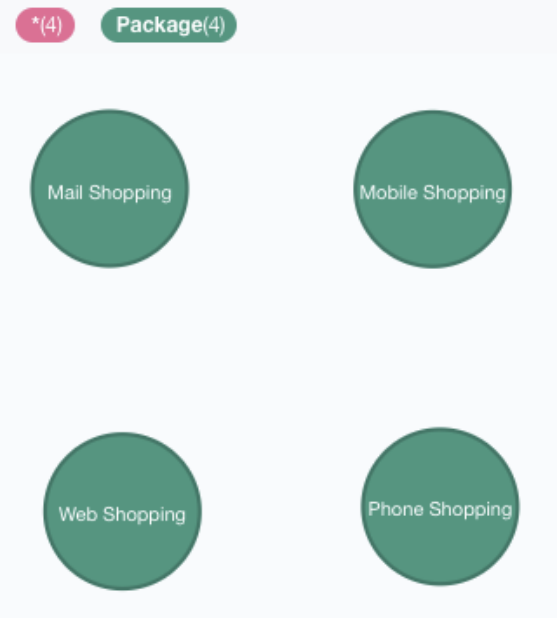
Stojí za zmínku, že v grafové databázi jsou všechny prvky na stejné úrovni. Znamená to, že pokud je Package obsahem jiného Package, tak je tento vztah nahrazen za relaci memberOf, která je na obrázku 6.5 reprezentována žlutou barvou. Tato vlastnost se může hodit v případě, pokud z důvodu přehlednosti není v diagramu možné vizuálně zachytit všechny takto zanořené Packages. Ve výsledné reprezentaci v grafové databázi se však objeví všechny elementy, které jsou v modelu obsaženy.

6.2.1 Vyhledání obsahu Package

Jedním z příkladů, na kterých demonstrujeme výsledky práce, je dotaz 6.4. Výsledkem tohoto dotazu jsou všechny Packages, které jsou obsahem Package se jménem `Shopping Type`. Obrázek 6.6 reprezentuje výsledek dotazu v grafové databázi.

```
MATCH (n:Package)-[rel:memberOf]->(:Package{name:"Shopping Type"})
RETURN n
```

Výpis kódu 6.4 Dotaz pro vyhledání obsahu Package podle jména



Obrázek 6.6 Obchodní systém – obsah Package podle jména (grafová databáze)

Z diagramu uvedeném na obrázku 6.4 není patrné, zda nejsou mezi Packages, které jsou obsahem Package se jménem `Shopping Type`, nějaké relace. Jak je patrné z výsledku dotazu, tak mezi nimi žádné relace opravdu nejsou. Pokud by mezi nimi nějaké relace existovaly, objevily by se ve výsledné reprezentaci v grafové databázi.

6.2.2 Vyhledání zdroje Package import

Tento příklad se zaměřuje na relaci public Package import, která je v grafové databázi reprezentována jako import. Dotaz 6.5 vyhledá všechny Package, které jsou zdrojem relace import. Výsledek dotazu v grafové databázi je zobrazen na obrázku A.7, který se nachází v příloze.

```
MATCH (n:Package)-[rel:import]->(:Package)
RETURN n
```

Výpis kódu 6.5 Dotaz pro vyhledání zdrojů relace import

Vyhledání uzlů, které odpovídají dotazu 6.5, může být užitečné v situaci, kdy je potřeba nalézt všechny Packages, které nejsou kompletní, protože jejich obsah je rozšířen o obsah jiných Packages.

6.2.3 Vyhledání cíle Package import

Na tomto příkladu je demonstrována relace public Package import. Od příkladu 6.2.2 se liší v tom, že vyhledá cíle relace public Package import. Dotaz v jazyce Cypher je zobrazen ve výpisu kódu 6.6 a od dotazu 6.5 se odlišuje pouze jedním znakem a to znakem <, který vyjadřuje směr hledané relace. Výsledek dotazu je zobrazen na obrázku A.8, který se nachází v příloze.

```
MATCH (n:Package)<-[rel:import]-(:Package)
RETURN n
```

Výpis kódu 6.6 Dotaz pro vyhledání cílů relace import

Vyhledání uzlů, které odpovídají dotazu A.8, může být užitečné v situaci, kdy je potřeba nalézt všechny Packages, které jsou vyžadovány jinými Packages, protože jejich obsah je importován do obsahu dalších Packages.

Kapitola 7
Závěr

Zadáním práce bylo implementovat moduly parseru pro UML diagramy balíčků a aktivit. Součástí zadání byla integrace implementovaných modulů parseru do aplikace Repocribro, konkrétně do projektu Exquiro. Pro splnění zadání bylo zapotřebí seznámit se s grafovou databází Neo4j, formátem OMG XMI a jeho implementací v Enterprise Architect pro UML diagramy balíčků (Package diagrams) a aktivit (Activity diagrams). Dále bylo zapotřebí seznámit se s projektem Exquiro, který se zabývá indexováním a vyhledáváním v konceptuálních modelech a je součástí aplikace Repocribro. Implementované moduly parseru bylo nutné navrhnout tak, aby je bylo možné jednoduše integrovat do stávajícího projektu. Dalším bodem zadání bylo řešení zdokumentovat, otestovat a demonstrovat na případové studii.

V práci jsme představili UML diagramy balíčků a aktivit včetně elementů, které se mohou v daných diagramech objevit a jsou součástí naší práce. To, které elementy jsme se rozhodli zařadit do rozsahu práce, vyplynulo z provedené analýzy. Implementovali jsme moduly parseru pro UML diagramy, které dokážou zpracovat soubory XMI, které vznikly vyexportováním z modelovacího nástroje Enterprise Architect. Navíc jsme implementovali moduly parseru tak, aby dokázaly zpracovat i soubory z modelovacích nástrojů Visual Paradigm a OpenPonk. Implementaci jsme zdokumentovali a provedli testování na úrovni unit testů. Implementované moduly parseru se nám podařilo úspěšně integrovat do projektu Exquiro, což jsme ověřili pomocí integračních testů. Nakonec jsme demonstrovali funkčnost vytvořených modulů parseru na případové studii. V ní jsou výstupy modulů parseru zobrazeny v grafové databázi Neo4j, za pomocí které je možné ve zpracovávaných diagramech vyhledávat.

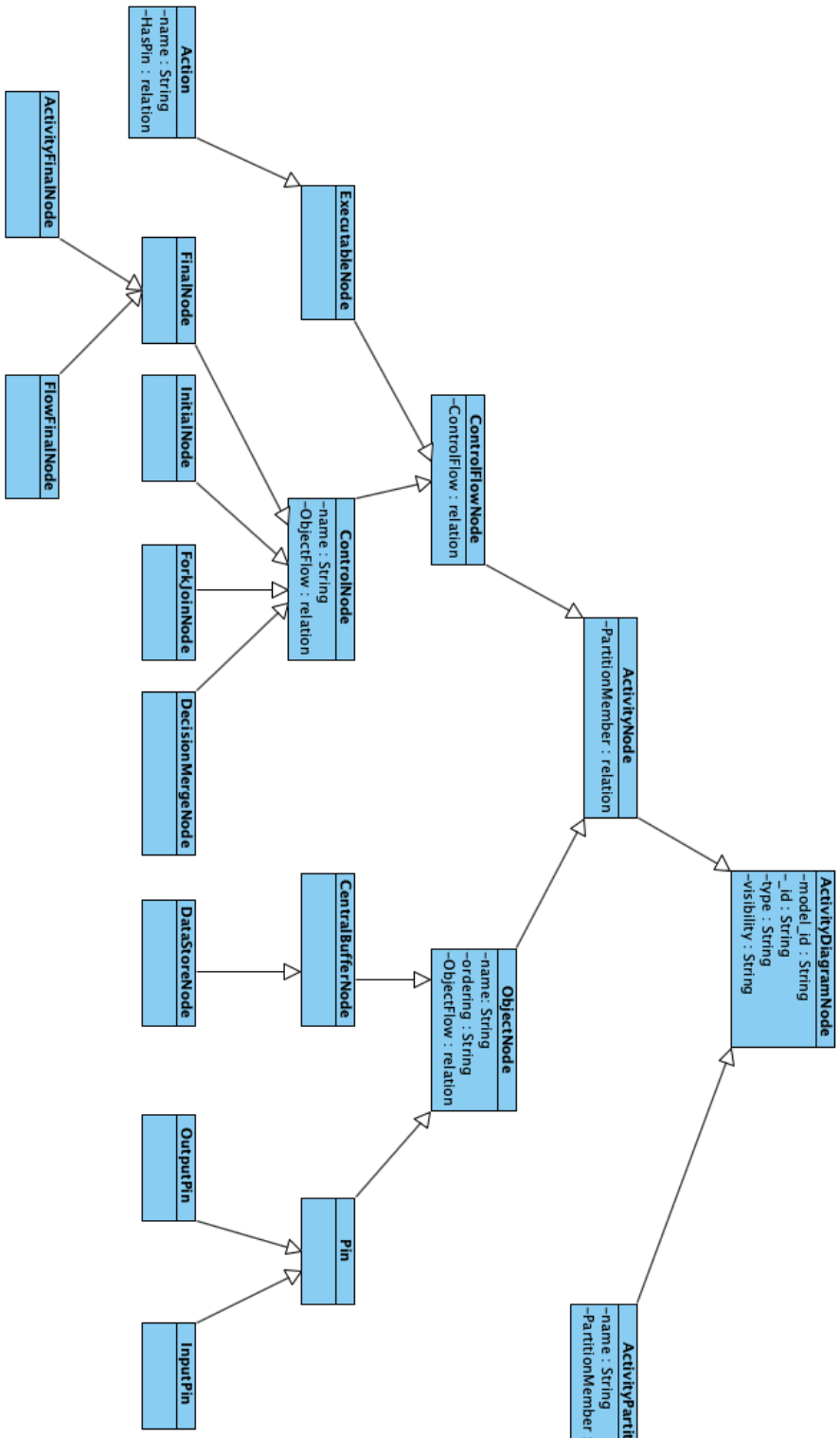
Ve zpracovávaných UML diagramech se může podle UML specifikace vyskytnout velké množství elementů a relací. Ve spolupráci s vedoucím práce jsme se rozhodli zpracovat pouze vybranou podmnožinu elementů a relací, která je v konkrétních diagramech nejpoužívanější. Zde bychom viděli možnost pro rozšíření modulů parseru takovým směrem, aby dokázaly zpracovat další elementy a relace, které nejsou obsahem naší práce. Další prostor pro pokračování této práce by mohlo být propojení jednotlivých modulů parseru tak, aby dokázaly zpracovat i takové diagramy, které kombinují elementy z různých UML diagramů.

Příloha A

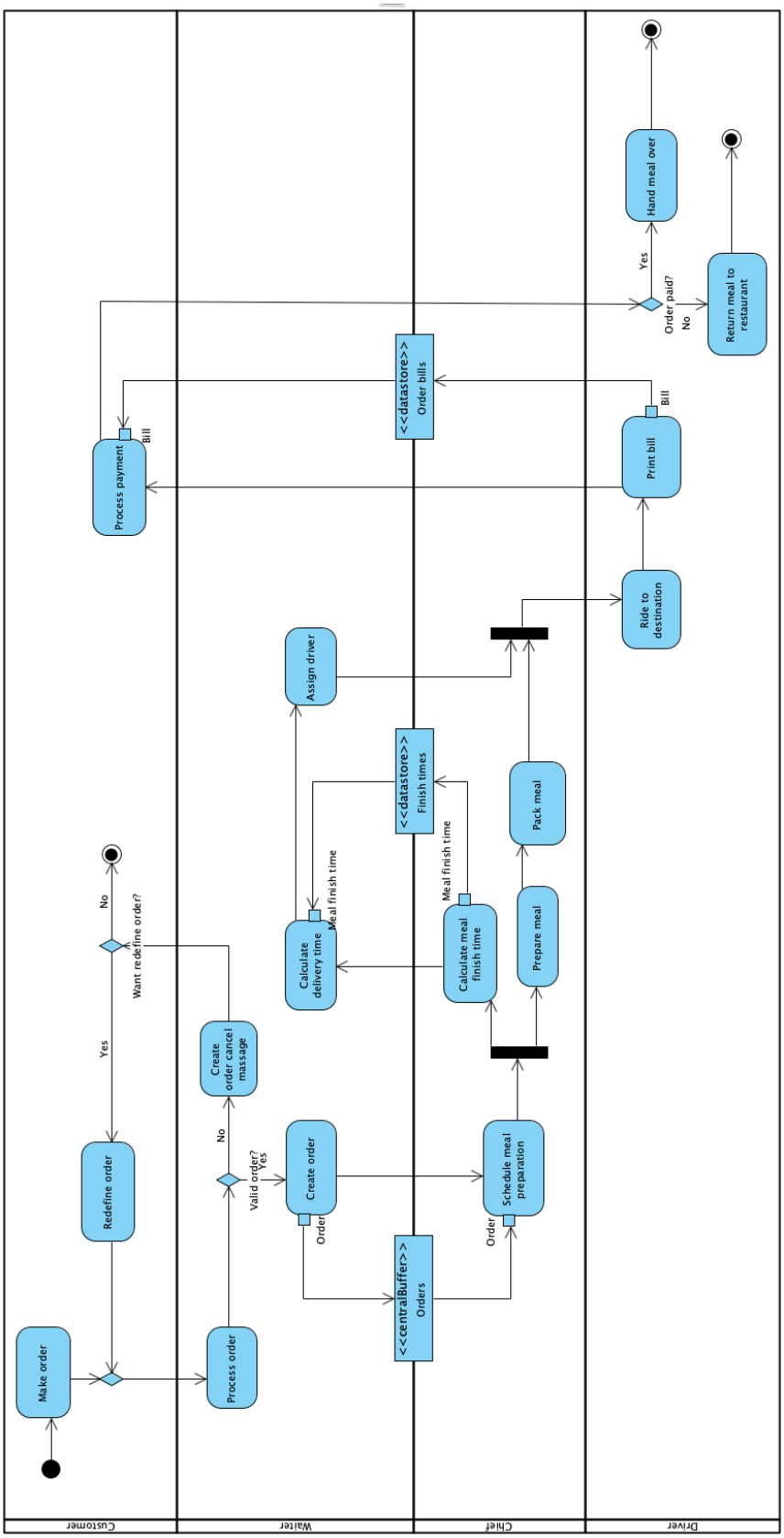
Příloha

Následující tabulka uvádí české překlady anglických termínů používaných v práci.

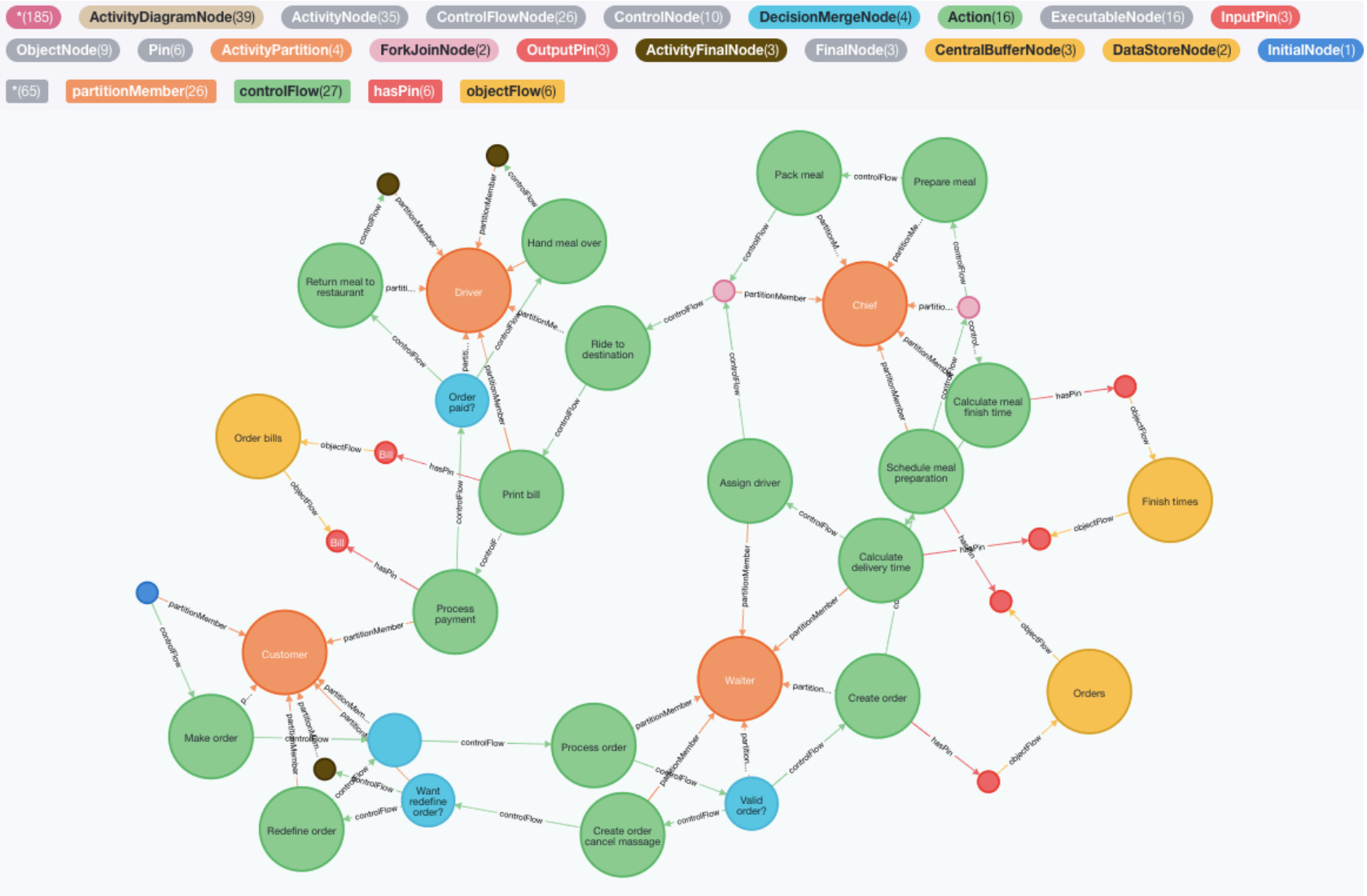
Anglicky	Česky
Action	Akce
Activity Diagram	Diagram aktivit
Activity Partition	Rozdělení aktivity
Activity Final Node	Uzel ukončující aktivitu
Component	Komponenta
Data Store Node	Uzel datového úložiště
Central Buffer Node	Uzel centrálního bufferu
Control Flow	Řídící tok
Control Node	Řídící uzel
Decision Node	Rozhodovací uzel
Dependency	Závislost
Edge	Hrana
Executable node	Spustitelný uzel
Final Node	Koncový uzel
Flow	Tok
Flow Final Node	Uzel ukončující tok
Fork Node	Rozdělovací uzel
Initial Node	Počáteční uzel
Input Pin	Vstupní pin
Join Node	Spojovací uzel
Merge Node	Slučovací uzel
Model	Model
Node	Uzel
Object Flow	Objektový tok
Object Node	Objektový uzel
Output Pin	Výstupní pin
Package	Balíček
Package Diagram	Diagram balíčků
Package Import	Import balíčku
Package Merge	Sloučení balíčku
Relation	Relace
Usage	Použití



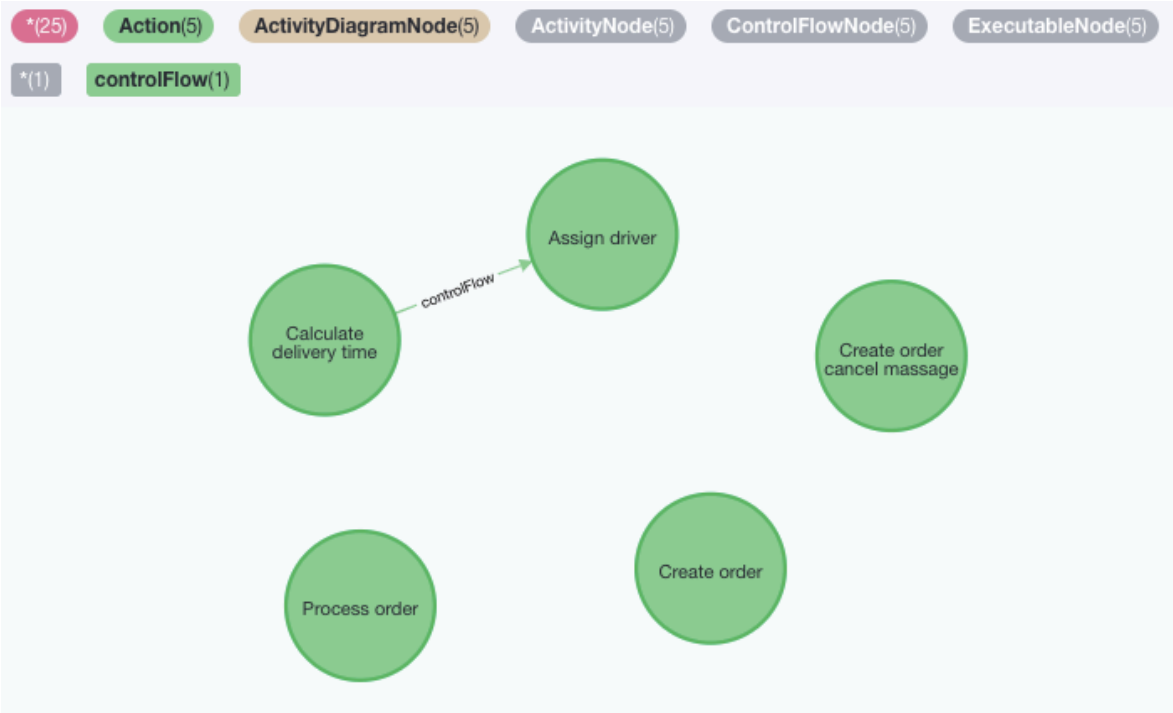
Obrázek A.1 Hierarchie tříd v diagramu aktivit



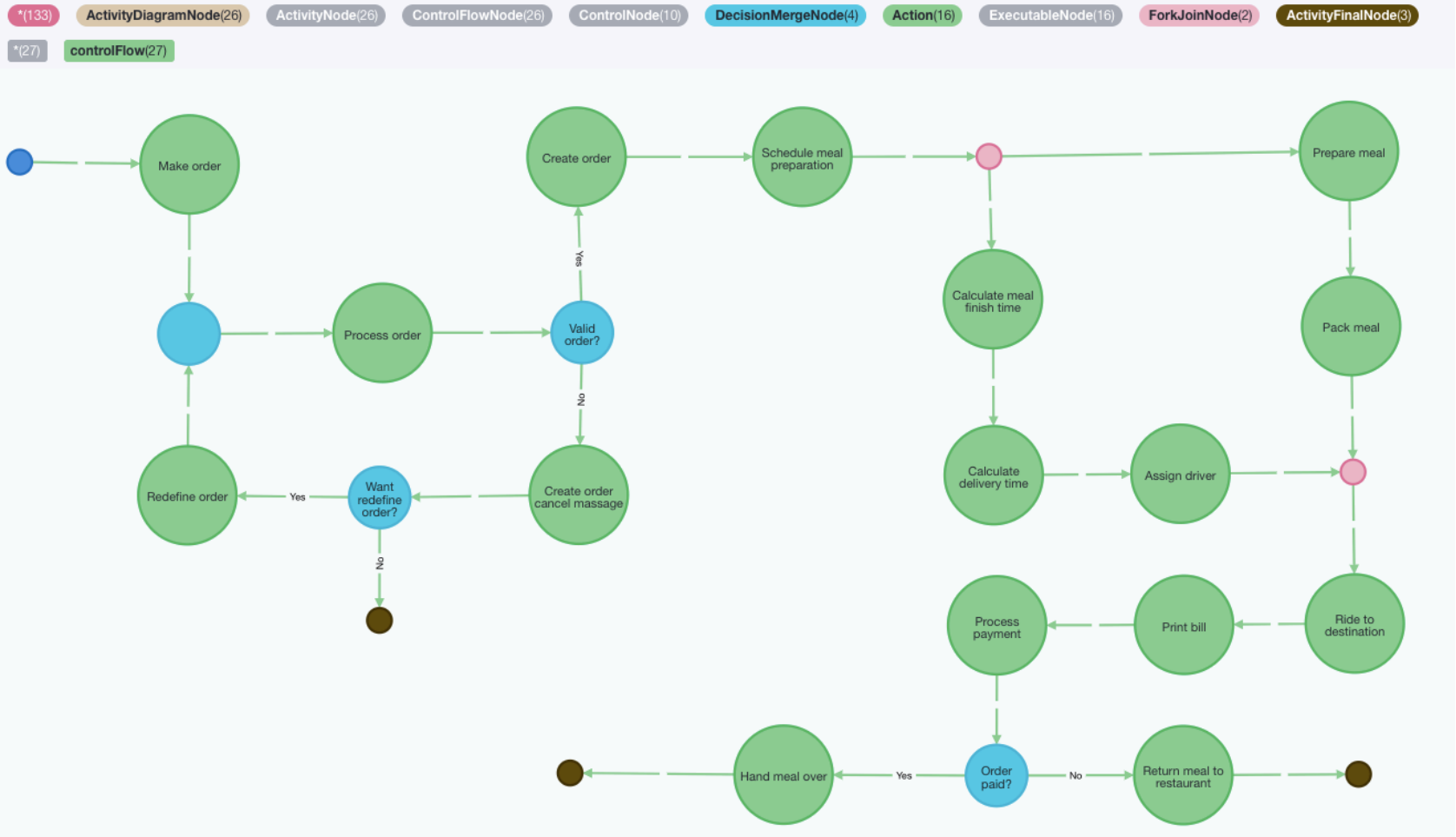
Obrázek A.2 Activity diagram – telefonní objednávka v restauraci (Visual Paradigm)



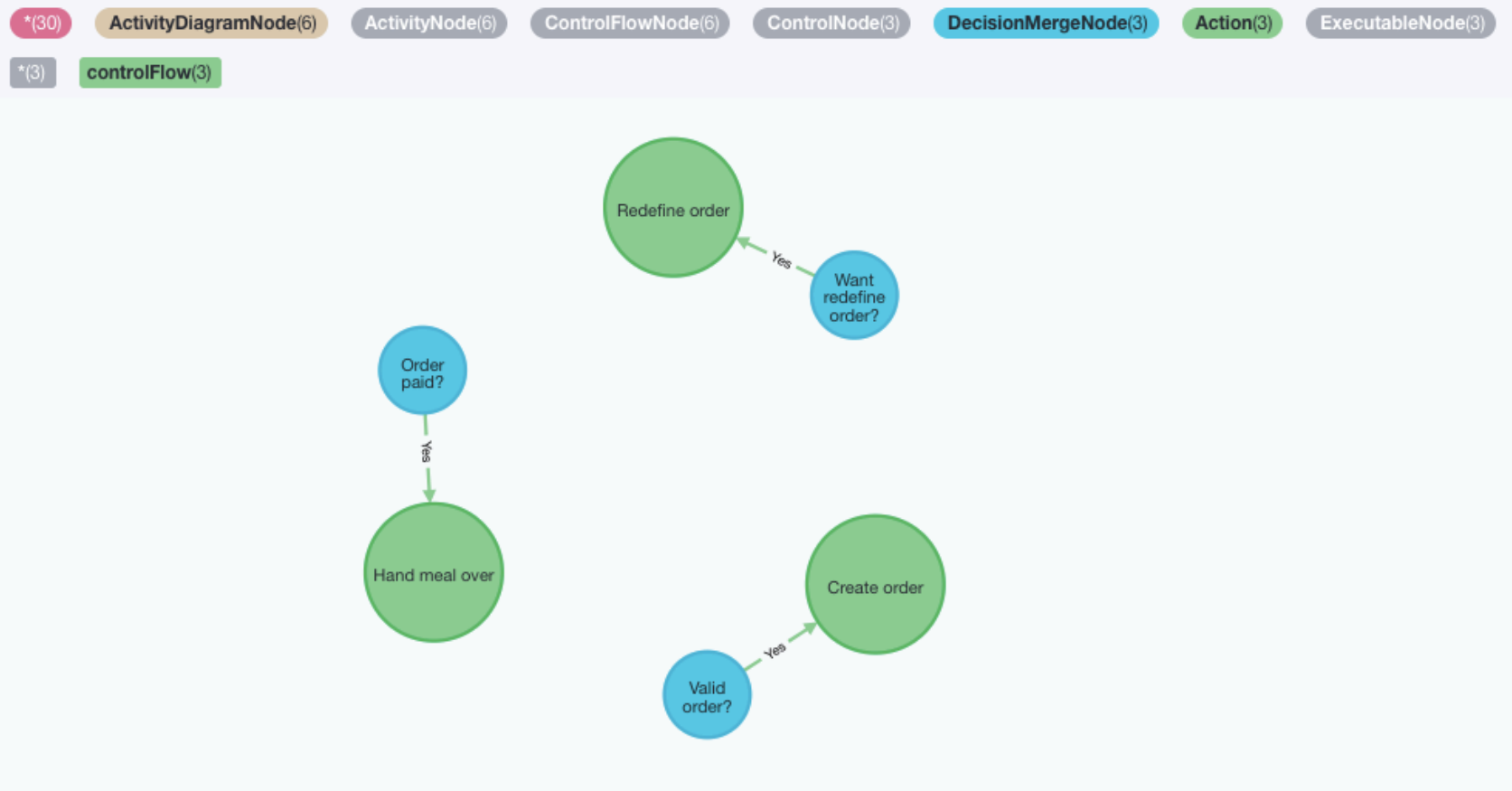
Obrázek A.3 Telefonní objednávka v restauraci (grafová databáze)



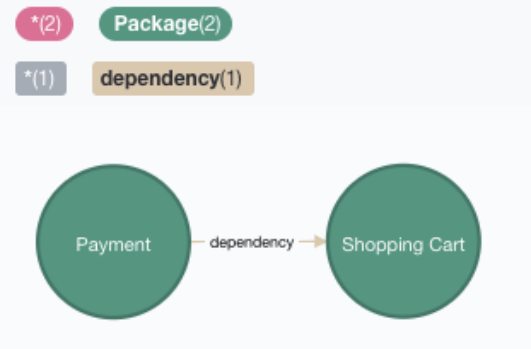
Obrázek A.4 Telefonní objednávka v restauraci – Activity partition (grafová databáze)



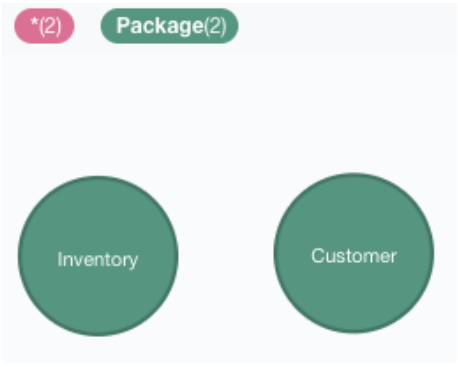
Obrázek A.5 Telefonní objednávka v restauraci – Control flow (grafová databáze)



Obrázek A.6 Telefonní objednávka v restauraci – Decision node s atributem guard (grafová databáze)



Obrázek A.7 Obchodní systém – zdrojové uzly relace import (grafová databáze)



Obrázek A.8 Obchodní systém – cílové uzly relace import (grafová databáze)

Bibliografie

1. ARLOW, Jim; NEUSTADT, Ila. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. Addison-Wesley, 2006. ISBN 9780321112309.
2. OMG. *XML Metadata Interchange (XMI) Specification, version 2.5.1* [online]. 2015 [cit. 2021-04-24]. Dostupné z: `https://www.omg.org/spec/XMI/2.5.1/PDF`.
3. OMG. *Unified Modeling Language, version 2.5.1* [online]. 2017 [cit. 2021-04-24]. Dostupné z: `https://www.omg.org/spec/UML/2.5.1`.
4. FAKHROUTDINOV, Kirill. *UML Package Diagrams Reference* [online]. 2016. Dostupné také z: `https://www.uml-diagrams.org/package-diagrams-reference.html`.
5. FAKHROUTDINOV, Kirill. *UML Package Import* [online]. 2016. Dostupné také z: `https://www.uml-diagrams.org/package-import.html`.
6. FAKHROUTDINOV, Kirill. *UML Activity Diagrams Reference* [online]. 2016. Dostupné také z: `https://www.uml-diagrams.org/activity-diagrams-reference.html`.
7. WIKIPEDIE. *Enterprise Architect — Wikipedie: Otevřená encyklopedie* [online]. 2021 [cit. 2021-04-24]. Dostupné z: `https://cs.wikipedia.org/wiki/Enterprise_Architect`.
8. VISUAL PARADIGM INTERNATIONAL LTD. *Visual paradigm – about us* [online]. 2020 [cit. 2021-04-24]. Dostupné z: `https://www.visual-paradigm.com/aboutus/`.
9. UHNAK, Peter; BLIZNIČENKO, Jan. *OpenPonk* [online]. 2020 [cit. 2021-04-24]. Dostupné z: `https://openponk.org`.
10. VAN ROSSUM, GUIDO. *Python - zdrojové kódy. Python Foundation* [online]. 2006 [cit. 2021-04-24]. Dostupné z: `https://web.archive.org/web/20160217132249/http://svn.python.org/view/*checkout*/python/trunk/Misc/HISTORY`.
11. NEO4J, INC. *Neo4j* [online]. 2021 [cit. 2021-04-24]. Dostupné z: `https://neo4j.com`.
12. PANZARINO, Onofrio. *Learning Cypher*. Packt Publishing, 2014. ISBN 1783287756.
13. LEACH, P. *A Universally Unique IDentifier (UUID) URN Namespace* [online]. 2005 [cit. 2021-04-24]. Dostupné z: `https://tools.ietf.org/html/rfc4122`.
14. SUCHÁNEK, Marek. *Repocribro Introduction* [online]. 2017 [cit. 2021-04-24]. Dostupné z: `https://repocribro.readthedocs.io/en/latest/intro.html`.
15. PYTHON SOFTWARE FOUNDATION. *xml.etree.ElementTree – The ElementTree XML* [online]. 2021 [cit. 2021-04-24]. Dostupné z: `https://docs.python.org/3/library/xml.etree.elementtree.html`.
16. BEHNEL, Stefan. *lxml - XML and HTML with Python* [online]. 2021 [cit. 2021-04-24]. Dostupné z: `https://lxml.de`.

17. NEO4J, INC. *Using Neo4j from Python* [online]. 2021 [cit. 2021-04-24]. Dostupné z: `https://neo4j.com/developer/python/`.

18. PYTHON SOFTWARE FOUNDATION. *Python unittest* [online]. 2021 [cit. 2021-04-24]. Dostupné z: `https://docs.python.org/3/library/unittest.html`.

19. NEO4J, INC. *Neo4j Desktop* [soft.]. 2021 [cit. 2021-04-24]. Dostupné z: `https://neo4j.com/developer/neo4j-desktop/`.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `BP_Stepanek_Frantisek.zip` ..... archiv zdrojové formy práce ve formátu LaTeX
  - `implementace.zip` ..... archiv zdrojových kódů implementace
  - `readme_implementace.txt` ..... popis použití zdrojových kódů
- `text` ..... text práce
  - `BP_Stepanek_Frantisek.pdf` ..... text práce ve formátu PDF