Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Analýza závislostí softwarových artefaktů

Plzeň 2022 Bc. Milan Hotovec

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Milan HOTOVEC**
Osobní číslo:	**A20N0081P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Analýza závislostí softwarových artefaktů**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s principy komponentově orientovaných softwarových systémů, způsoby reprezentace a analýzy grafových dat.
2. Na základě studia literatury a podnětů z praxe vyberte a popište množinu problémů, které vyžadují analýzu informací o softwarových komponentách a/nebo grafu jejich závislostí.
3. Analyzujte reprezentace komponent a jejich závislostí používané v různých technologiích (např. Java/Maven, .NET/NuGet apod.), najděte nebo navrhněte vhodný obecný model a úložný formát pro reprezentaci výsledného grafu komponent.
4. Navrhněte a implementujte sadu nástrojů pro získání reprezentace grafu komponent (pro alespoň dvě technologie) a jeho analýzu, použijte je v návaznosti na bod 2 zadání.
5. Ověřte funkčnost a kvalitu vytvořených nástrojů, kriticky zhodnoťte jejich použití a výsledky provedených analýz.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Doc. Ing. Přemysl Brada, MSc., Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **10. září 2021**
Termín odevzdání diplomové práce: **19. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2021

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 19. května 2022

Bc. Milan Hotovec

Abstract

The aim of this thesis is to familiarize with the component oriented software development, furthermore with possibilities of graph data representation and analysis. Within the thesis, a general storage format for storing the dependency graph was proposed. A set of the problems that requires additional dependency graph analysis, was also defined within this thesis. Two dependency graph creation tools were implemented, each tool specializes on one repository type and its result is the dependency graph for the given artefacts in the given technology. One tool is for Nuget, the .NET platform package repository and the second is for PyPI, Python package registry. Also, specialized analyzers were designed and implemented, performing specific analysis on the given dependency data. The tools were designed with emphasis on general use and easy extensibility. The tools can provide the necessary data for analysis and the results of the analyzers are beneficial in improving the quality of the software.

Abstrakt

Cílem diplomové práce je seznámit se s komponentově orientovaným vývojem softwaru, následně se způsoby reprezentace a analýzy grafových dat. V rámci práce byl navrhnut a popsán obecný úložný formát pro uložení grafu závislostí. Byla vydefinována množina problémů, které vyžadují dodatečnou analýzu grafu závislostí. Došlo k vytvoření dvou nástrojů, vytvářejících graf závislostí ze specifických repozitářů. Nástroje jsou pro repozitář Nuget, pro platformu .NET a pro repozitář PyPI, pro jazyk Python. V neposlední řadě byly také navrhnuty a vytvořeny specializované analyzátory, které dokáží provádět analýzu nad získanými daty. Nástroje byly vytvořeny s důrazem na co největší obecné použití a snadnou rozšířitelnost. Nástroje dokáží poskytnout potřebná data pro analýzu a výstupy analýz jsou přínosné při zkvalitňování softwaru.

Poděkování

Chtěl bych poděkovat vedoucímu diplomové práce, kterým byl doc. Ing. Přemysl Brada, MSc., Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce. Mé poděkování patří též kolegům z firmy Leuze Engineering Czech s.r.o. za spolupráci při získávání údajů pro výzkumnou část práce. Na závěr bych také rád poděkoval své rodině a všem přátelům, kteří mě při vytváření této práce podpořili a bez jejichž pomoci by nebylo možné práci dokončit.

Obsah

1 Úvod

Vývoj softwaru se neustále mění a jedním z oblíbených a rozšířených způsobů vývoje je využívat již existující části kódu. Tím se velmi urychlí vývoj a zároveň je to rozumná cesta k tomu mít kód stabilnější. Využíváním takových kusů kódu, typicky komponent či obecněji řečeno balíčků, ovšem vzniká závislost na použitém kódu. Tato závislost znamená, že vyvíjený kód nemůže bez takových závislostí fungovat. Minimálně ne tak, jak bylo zamýšleno. Dalším problémem je, že závislost může mít i jiné závislosti, a tím vzniká graf závislostí. I z jediné závislosti může vzniknout velmi obsáhlý graf závislostí.

Samotný graf vzniká z principu věci. Bohužel se může stát, že není možné jej správně sestavit. Může to být způsobeno vinou samotného programátora, protože špatně definoval závislosti aplikace. Horší případ je, když programátor udělal vše správně, ale vlivem nepřímých závislostí, získaných z přímých závislostí, dojde k problému, který způsobí, že graf závislostí není možné správně sestavit. Tyto problémy jsou obtížně odhalitelné, protože nejsou na první pohled vidět.

Cílem této práce je získat graf závislostí pro zvolený programovací jazyk a jeho příslušný balíčkovací systém, detailněji popsáno v kapitole 8. Získaný graf následně uložit do navrženého obecného formátu, který je popsán v kapitole 6. Nad získanými daty následně provést různé druhy analýz vycházejících z podnětů v kapitole 5. Analyzátory, které jsou popsány v kapitole 9 jsou specializovány pro jednotlivé problémy a dokážou poskytnout potřebné informace, které povedou ke zkvalitnění a spolehlivosti vyvíjeného softwaru. Výsledky práce jsou podrobně popsány v kapitole 10.

2 Komponentový vývoj

Smyslem komponentově orientovaného přístupu k vývoji softwaru je rozdělit aplikaci na menší části [18]. Tyto části jsou samostatné, zaobalují ucelenou funkcionalitu a nazývají se komponenty. Přínosem tohoto přístupu je hlavně urychlení vývoje softwaru. Díky znovupoužití již existujících částí kódu dojde také ke zlevnění ceny za vývoj. Další výhodou je zjednodušit vývoj programátorům, není třeba pochopit problematiku do hloubky, pokud existuje komponenta, která již problém řeší. Aplikace může být postavena pouze pomocí propojení několika komponent. Logika aplikace se může starat pouze o správné řízení volání funkcionalit z dostupných komponent bez nutnosti řešení specializovaného problému, o který se postará dodaná komponenta [17].

Komponenty je možné snadno zaměňovat za jiné. Stačí pouze upravit logiku, která s nimi pracuje. Pokud se jedná o podobné komponenty, záměna nebývá příliš nákladná. Důvodem pro takovéto záměny může být zlepšení efektivity aplikace jako celku, nebo v případě již neudržované komponenty, přechod na modernější verzi komponenty [17].

Velmi markantní význam má použití komponent v korelaci s otestováním výsledného software. Použití komponent, které jsou již delší dobu k dispozici a jsou aktivně používány, evokuje, že komponenty svou činnost plní dobře. Zároveň díky aktivnímu používání se komponenty podrobily širokému spektru otestování a pokud se nějaké chyby našly, byly zjevně opraveny. Komponenty obsahující chyby by širší veřejnost aktivně nepoužívala. Je důležité zmínit, že tak rozsáhlému testování, jakému byly vystaveny aktivně užívané komponenty se námi napsaný kód většinou vyrovnat nemůže. Nedostatečné testování je často způsobeno vlivem časového a finančního omezení při vývoji. V případě nalezení chyby v komponentě je běžné mít cestu, jak chybu vývojáři příslušné komponenty nahlásit, aby byla co nejdříve opravena.

2.1 Komponenta

Jedná se o ucelenou část funkcionality, která je snadno využitelná třetí stranou [18]. Samotná implementace komponenty není vidět, pro interakci s komponentou stačí mít k dispozici její rozhraní [2]. Pomocí tohoto rozhraní je možné pracovat s komponentou a využít její funkcionalitu. Například volání metod z rozhraní. Znalost rozhraní sama o sobě ještě není zcela dokonalá. Je ještě potřeba kompletně pochopit chování komponenty pro příslušné vstupy

a jaké stavy má volající očekávat, případně jaké chybové stavy ošetřit. Je tedy třeba znát specifikaci, kontrakt, aby volající věděl jak s komponentou správně pracovat. Takovému principu práce s komponentou se říká black-box model [17].

Komponenta je vytvořena programátorem, který ji následně distribuuje ostatním programátorům pod příslušnou licencí. Komponenta může pro svou činnost potřebovat funkcionalitu z jiných komponent [17].

2.2 Jak získat komponenty

Komponenty je možné sdílet mezi programátory soukromě, nebo případně v rámci firmy. Ovšem mnohem snadnějším a více rozšířeným způsobem je komponenty získávat z veřejných datových úložišť zvaných softwarové repozitáře [3]. Jedná se o úložiště, které uchovává balíčky, včetně jejich starších verzí. Balíček obsahuje určité dodatečné informace, ale hlavně obsahuje komponentu, která je požadována.

Existují různé druhů repozitářů, které se liší určitými vlastnostmi. Například vyhledávacími schopnostmi, případně zaměřením. Některé jazyky mají i své příslušné repozitáře, ze kterých se dají získat balíčky obsahující komponenty kompatibilní s příslušným jazykem. Repozitáře běžně poskytují cestu, jak do nich balíčky nahrát a tím je v repozitáři uložit. Je obvyklé mít možnost mezi balíčky vyhledávat, ať už za pomoci dostupného nástroje pro specifický repozitář, či pomocí webového rozhraní. Schopnosti těchto vyhledávačů nebývají příliš rozsáhlé, je běžné vyhledávat alespoň pomocí jména balíčku. Lepší vyhledávače poskytují i hledání pomocí klíčových slov. Základní vlastností repozitáře je možnost získat balíček o specifické verzi, případně získat informace o verzích požadovaného balíčku. Repozitáře u balíčků uchovávají dodatečné informace, pro lepší pochopení významu a obsahu balíčku. Mezi základní informace patří popisek balíčku, či licence. Repozitář ale uchovává u balíčků i velmi kritickou informaci a tou je seznam přímých závislostí [3]. Jedná se o další balíčky z repozitáře, které jsou nezbytné ke správné funkcionalitě balíčku.

Pro platformu .NET je možné balíčky obstarávat z repozitáře zvaného Nuget. Pomocí dostupného nástroje, nesoucího stejný název, je možné balíčky získávat. V době psaní této práce je v repozitáři Nuget dostupných přes 4 miliony balíčků a z toho 307 tisíc unikátních balíčků.

Pro jazyk Java je k dispozici repozitář zvaný Maven. Pomocí dostupného nástroje, nesoucího opět stejný název, je možné balíčky získávat. V době psaní této práce je v repozitáři Maven dostupných necelých 9 milionů balíčků

a z toho 474 tisíc unikátních balíčků.

Pro jazyk Python je k dispozici repozitář zvaný PyPI, jedná se o zkratku z Python Package Index. Pomocí dostupného nástroje, nesoucího název pip, je možné balíčky získávat. V době psaní této práce je v repozitáři PyPI dostupných necelých 3,5 milionu balíčků a z toho 375 tisíc unikátních balíčků.

2.3 Rizika

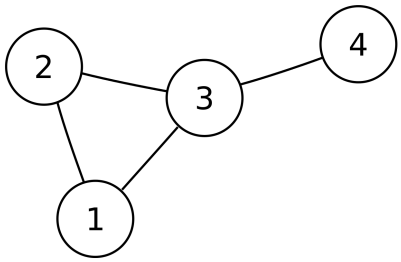
Komponentový vývoj má mnoho pozitivních vlastností, které zkvalitní vývoj software. Ovšem jsou zde i úskalí, kterými může být i samotný přechod na komponentově orientovaný vývoj. Například pro starší software bude obtížné do něj komponenty vkládat a s největší pravděpodobností budou zastaralé.

Dalším problémem je, že komponent řešících problém existuje většinou více a je třeba si vybrat tu nejvhodnější, mohou se lišit například rozsahem funkcionality. Komponenty jsou většinou v aktivním vývoji, a proto je třeba sledovat jestli komponenta nebyla aktualizována. Tyto aktualizace mohou přidávat pouze dodatečné funkcionality, ale mohou být také mnohem důležitější. Aktualizace mohou opravovat chyby, které byly odhaleny až při širším použití, v horším případě by se mohlo jednat o bezpečnostní chyby. Bezpečnostní chyby mohou ve vyvíjeném softwaru způsobit závažný problém, který by se měl urychleně řešit aktualizací použité komponenty. Při větším počtu komponent může být obtížné hlídat jejich aktuální verze.

3 Grafové struktury a jejich zpracování

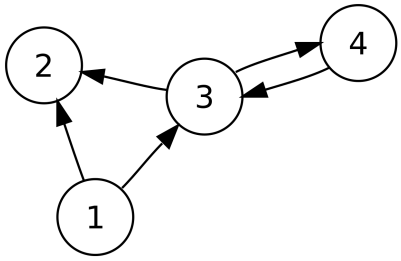
Při analýze závislostí balíčků na jiných balíčcích vzniká graf závislostí. Jedná se tedy o grafová data, které se dají reprezentovat různými způsoby a dají se na ně aplikovat známé grafové algoritmy, například pro prohledávání. Je tedy potřeba nejdříve pochopit co je graf a jak se s ním pracuje.

Graf lze definovat jako uspořádanou dvojici uzlů a hran [11]. Uzlem označujeme vrchol. Hrana spojuje právě dva vrcholy. Pokud jsou dva vrcholy spojeny hranou, říká se že spolu sousedí. Stupeň vrcholu je vyjádřen jako počet vrcholů, které s ním sousedí. V případě neorientovaného grafu, viz obrázek 3.1, je možné hranou procházet oběma směry. V orientovaném grafu ovšem záleží na směru hrany a je možné procházet hranou pouze v jednom směru, dle orientace příslušné hrany. Ukázka orientovaného grafu je vidět na obrázku 3.2.



Obrázek 3.1: Příklad neorientovaného grafu

Zdroj: wikimedia.org - Undirected graph



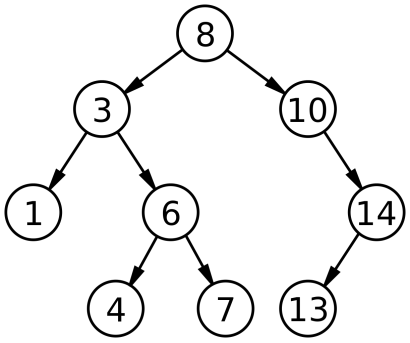
Obrázek 3.2: Příklad orientovaného grafu

Zdroj: wikimedia.org - Directed graph

Posloupnost vrcholů taková, že mezi každými dvěma po sobě jdoucími vrcholy existuje hrana se nazývá sled [11]. V případě orientovaného grafu je potřeba zohlednit, že hrana je správně orientovaná. Za pomoci sledu je možné definovat souvislost grafu. Souvislým grafem je takový graf, mezi jehož libovolnými uzly existuje sled. Komponenta grafu je maximální souvislý podgraf. Analogicky pro orientovaný graf hovoříme o silně souvislém grafu a silné komponentě.

V případě, že procházíme orientovaný graf z určitého vrcholu 'v' a v rámci procházení orientovaných hran dojdeme opět do vrcholu 'v', ze kterého započalo prohledávání, říkáme že, graf obsahuje cyklus. Orientovaný graf, který neobsahuje cyklus označujeme za acyklický orientovaný graf [10].

Speciální případem souvislého orientované acyklického grafu je strom. Ukázka binárního stromu je vidět na obrázku 3.3. Jedná se o hierarchickou strukturu. Strom je definován tak, že každý vrchol má maximálně jednoho předka. Uzel, který je praotcem všech uzlů nazýváme kořenem. Uzel, který nemá žádné potomky nazýváme listem. V případě, že ve stromu má každý uzel maximálně dva potomky, hovoříme o binárním stromu. Při odebrání jedné hrany ve stromu dojde k porušení souvislosti. Přidáním hrany vzniká kružnice [8].



Obrázek 3.3: Příklad binárního stromu

Zdroj: stanford.edu - Binary tree

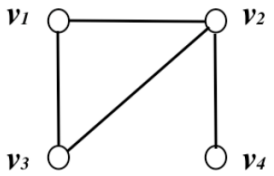
3.1 Reprezentace grafových dat

Graf lze reprezentovat několika různými způsoby. Výběr způsobu reprezentace je většinou podle typu úlohy, ke které je graf použit. Velmi často bývá graf reprezentován pomocí matice sousednosti, matice incidence (zohledňuje směr hran) nebo spojovou reprezentací.

3.1.1 Matice sousednosti

Matice sousednosti je jedním ze základních způsobů reprezentace grafových dat. Jedná se o matici velikosti U×U, ve které je souřadnice daná řádkem m a sloupcem n. Hodnota na souřadnicích je jednotková právě tehdy, když z uzlu m vede hrana do uzlu n, v opačném případě je nulová. Matice sousednosti je pro neorientované grafy symetrická podle diagonály. Tento způsob reprezentace informuje hlavně o vztahu mezi vrcholy, ovšem samotné vrcholy je v případě potřeby nutné uložit navíc, například do pole. Příklad matice sousednosti, spolu s odpovídajícím grafem, je vidět na obrázku 3.4.

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$



Obrázek 3.4: Matice sousednosti

Zdroj: medium.com - Adjacency matrix

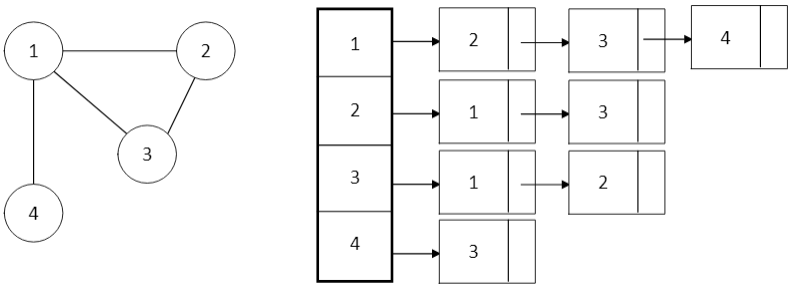
3.1.2 Seznam uzlů a hran

Jedním z dalších základních způsobů reprezentace grafu je uložení seznamu uzlů a seznamu hran. Reprezentace jedné hrany je možné brát jako orientovanou i jako neorientovanou, záleží na implementaci. Ve většině případů se uvažuje orientovaná hrana. Pak je tedy potřeba pro neorientovaný graf vytvořit pro jednu jeho existující neorientovanou hranu, kvůli korektní reprezentaci, dvě hrany orientované, opačnými směry. Tento přístup není příliš optimální, velice ztěžuje procházení, protože při hledání vazby mezi dvěma vrcholy je v nejhorším případě potřeba projít všechny hrany [1].

3.1.3 Spojová struktura

Tento způsob reprezentace je jedním z nejpoužívanějších, z hlediska implementace v programech [11]. Pro každý vrchol existuje spojový seznam. Každý z těchto spojových seznamů obsahuje ukazatele na všechny uzly, do kterých vede hrana. Tato reprezentace je tedy vhodná jak pro orientovaný, tak neorientovaný graf.

Reprezentaci spojovou strukturou lze ještě trochu upravit, při zohlednění vlastností konkrétního grafu. Pokud lze o grafu prohlásit, že se jedná o strom,



Obrázek 3.5: Reprezentace grafu spojovou strukturou

Zdroj: debug.to - Adjacency list

každý vrchol grafu u sebe stále drží seznam vrcholů, do kterých od něho vede hrana. Není ovšem nutné uchovávat si reference na všechny vrcholy v grafu, ale stačí pouze uchovávat referenci na kořen stromu. Všechny ostatní vrcholy jsou dosažitelné z vrcholu, který je označen jako kořen. Například na obrázku 3.3 stačí uložit pouze referenci na vrchol značený jako 8. Ostatní vrcholy je možné zjistit při procházení grafu.

3.2 Způsoby uložení

Při výběru úložného média je potřeba silně zohlednit jak je graf reprezentován. Dle reprezentace je poté možné zvolit nejvhodnější formát. Pro uložení grafových dat lze použít známé datové formáty, kterými jsou například JSON, případně XML. Pro grafová data existují i specializované databáze, které svými vlastnosti poskytují určitý komfort při práci s daty.

3.2.1 JSON

Název JSON je zkratkou pro JavaScript Object Notation, neboli JavaScriptový zápis objektů. Jedná se o formát pro výměnu dat, který je hojně využíván ve webových technologiích. Je často využíván hlavně pro zápis krátkých strukturovaných dat. JSON formát dovoluje uložit několik základních typů dat [9]. Velkou výhodou tohoto formátu je, že je čitelný člověkem a je možné do něj ručně provádět úpravy, při dodržení podmínek validního JSON formátu. Dominantou při použití JSONu je využití vazby klíč-hodnota [20].

- JSONString - jedná se o uložení textového řetězce. Řetězec je potřeba uložit do uvozovek, apostrofy povoleny nejsou.
- JSONNumber - jedná se o uložení číselné hodnoty. Číslo může být celočíselné nebo reálné. Je povolen i zápis s exponentem.

- JSONBoolean - jedná se o uložení logické hodnoty. Povolené jsou hodnoty *true* a *false*.
- JSONNull - jedná se o uložení speciální hodnoty null. Tato hodnota vyjadřuje, že zde opravdu nic není.
- JSONArray - jedná se o uložení seřazeného pole hodnot. V rámci JSONu se jedná o kontejner, který obsahuje hodnoty. Hodnoty v poli nemusí být stejného typu. Pole je ohraničeno hranatými závorkami. Pole v sobě může obsahovat další pole, je dovoleno je do sebe vnořovat.
- JSONObject - jedná se o uložení objektu. Jedná se o kontejner, který obsahuje data. Každá datová položka je dvojicí klíč-hodnota, klíčem je typ JSONString. Do objektů lze vkládat další objekty a vytvářet tak složitější struktury.

Níže je ukázka pole v JSONu, pole je o třech prvcích, prvním prvkem je další pole o dvou prvcích. Dalším prvkem je null a posledním prvkem je řetězec.

```
[[1, 2], null, "apple"]
```

Níže je ukázka objektu v JSONu, jedná se o objekt reprezentující osobu. Objekt má několik atributů, prvním je vnořený objekt, ve kterém se nachází atributy pro jméno a věk. Dalším atributem je příznak, vyjádřený typem JSONBoolean, který vyjadřuje jestli se osoba již setkala. Atribut „hobbies“ je nastaven na hodnotu null, poslední atribut „data“ obsahuje pole dvou hodnot.

```
{"person": {"name": "Robin", "age": 35},
           "met": false,
           "hobbies": null,
           "data": [1,2]}
```

3.2.2 XML

Název XML je zkratkou pro eXtensible Markup Language, což znamená rozšiřitelný značkovací jazyk. Jedná se o formát pro výměnu serializovaných dat, stejně jako JSON. V XML jsou informace členěny pomocí tagů do skupin podle logického uspořádání dokumentu. Jedná se o standardizovaný formát uložení dat, který je lidsky čitelný a upravitelný. Implementace parserů je vcelku jednoduchá a již existuje velké množství implementací, které je možné využít [12]. Základní pravidla pro psaní XML jsou vcelku jednoduchá.

- Obsah značky (elementu) musí začínat tvarem: <jmenoZnacky>
- Konec elementu, který je povinný, je vyznačen příslušnou koncovou značkou: </jmenoZnacky>
- Před ukončením elementu musí být ukončeny všechny jeho vnitřní elementy
- Prázdný element se označí speciálně, aby bylo vidět, že je sám sebou uzavřený: <jmenoZnacky/>
- Počáteční značka může obsahovat volitelné atributy v následujícím formátu: <jmenoZnacky atr1=„aaa“ atr2=„abc“>
- Znaky, které jsou součástí textu a mají speciální význam musí být escapnuty
- Nejvyšší element v hierarchii elementů se nazývá kořenový element a smí být pouze jeden

Mimo základní pravidla by měl XML dokument také obsahovat hlavičku, ve které dojde k definování verze XML standardu, kterou dokument používá. Níže je ukázka XML dokumentu včetně jeho hlavičky.

```
<?xml version="1.0" encoding="iso-8859-2" ?>

<dokument>
  <nadpis level="1">Muj zkusebni dokument</nadpis>
  <kapitola number="1">
    <nadpis level="2">Prvni kapitola</nadpis>
    <odstavec>Tak tohle je muj prvni XML dokument
    (tedy neni, ale jako by byl).</odstavec>
    <odstavec>A tohle je dalsi odstavec. Trocha
    matematiky: 2&lt;5.</odstavec>
    <obrazek soubor="/home/beda/muj_obrazek.png"/>
  </kapitola>
</dokument>
```

3.2.3 Grafová databáze

Jedná se o specializovanou databázi určenou pro ukládání grafových dat. Z obecné definice lze označit za grafovou databázi jakýkoliv systém, kde vrchol obsahuje přímé odkazy na své sousedy a tím nám odpadá nutnost používání

indexů. Díky tomu, že jsou grafové databáze nativně určeny pro tento typ dat, práce nad daty je poměrně rychlá.

Specializovaná grafová databáze proto eliminuje problémy, které vznikají při uložení grafových dat do relační databáze. Relační databáze totiž pro taková data není vhodná, kvůli nutnosti použití výpočetně drahých operací typu JOIN. Dále jsou ideální pro mapování na objektovou strukturu aplikací bez rigidního modelu, což zaručuje volnost schéma libovolně rozvíjet. Naopak v porovnání relační databáze velmi dobře pracují s tabulárními daty, nad kterými jsou prováděny agregační a často se opakující úlohy.

Většinou jsou vhodným řešením pro grafové úlohy, jako například hledání nejkratších cest v grafu [15].

Mezi významné zástupce grafových databází patří například Neo4j [16]. Jedná se o grafovou databázi napsanou v jazyce Java. První verze byla vydána v roce 2007. Ukládání grafu probíhá uložením vrcholů, hran a vlastností. Vlastnosti jsou dodatečnou přidanou hodnotou a mohou být svázány s vrcholem, případně s hranou. Jedná se o data typu klíč-hodnota.

3.3 Prohledávání do šířky

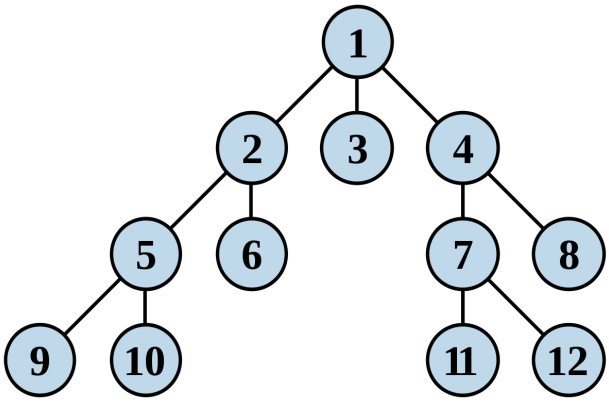
Algoritmus prohledávání do šířky, zkráceně známý jako BFS (breadth-first search), je jedním ze základních algoritmů, které se používají k procházení grafu [4]. Tento algoritmus je základem jiných složitějších grafových algoritmů. Například Dijkstrův algoritmus.

Princip tohoto algoritmu je vcelku jednoznačný. V grafu se vybere právě jeden uzel a z tohoto uzlu započne prohledávání. Uzel se vloží do fronty uzlů ke zpracování. Dokud není fronta prázdná, vybere se z fronty jeden uzel a ten se zpracuje. Zpracováním uzlu se rozumí zjištění jeho přímých potomků. Tyto potomky vložíme do fronty uzlů ke zpracování. Tím se dokončí zpracování aktuálního uzlu a je potřeba opět z fronty vybrat uzel ke zpracování a celý postup opakujeme dokud není fronta prázdná. V rámci vyřešení problémů s cykly je potřeba udržovat v paměti seznam již zpracovaných vrcholů. Vrcholy které byly již zpracovány, se znovu nezpracovávají, a pokud se na ně narazí, jednoduše se přeskočí.

Významnou vlastností tohoto algoritmu je zpracovávání grafu po patrech, neboli vlnách. Za první vlnu lze označit kořen, další vlnou jsou jeho přímí potomci, a obdobně až do konce pohledávání grafu. Uzly jsou zpracovány v pořadí daném jejich vzdáleností od kořene ve smyslu počtu hran. Díky této vlastnosti lze prohlásit, že pokud narazíme na vrchol, který hledáme, cesta která byla použita k nalezení je nejkratší možná. Nejkratší možná znamená

ve smyslu počtu hran, tento algoritmus nezohledňuje ohodnocení hran.

Na obrázku 3.6 je vizualizace prohledávání do šířky nad grafem. Uzel označený jako "1" je kořenem a z něj se zahájí prohledávání. Označení vrcholů odpovídá pořadí, ve kterém jsou vrcholy zpracovávány.



Obrázek 3.6: Prohledávání do šířky

Zdroj: wikimedia.org - Breadth-first tree

Za předpokladu rozumné implementace, kdy je přístup k potomkům daného uzlu konstantní. Kdy rovněž uložení a výběr prvku z fronty je též konstantní. Asymptotickou složitost celého algoritmu lze vyjádřit jako $O(|U| + |H|)$, kde U je množina uzlů a H je množina hran. Tato složitost odpovídá myšlence algoritmu, protože projde každý uzel a hranu právě jednou.

3.4 Prohledávání do hloubky

Algoritmus prohledávání do hloubky, zkráceně známý jako DFS (depth-first search), je algoritmus určený k procházení grafu. Jeho přístup je jiný než u BFS, algoritmus preferuje co největší zanoření před zpracováním ostatních uzlů v daném patře grafu [19]. Tímto způsobem dojde k průchodu všech větví grafu do maximální hloubky. Algoritmus je velmi vhodný při hledání počtu komponent v grafu, případně při hledání cyklu.

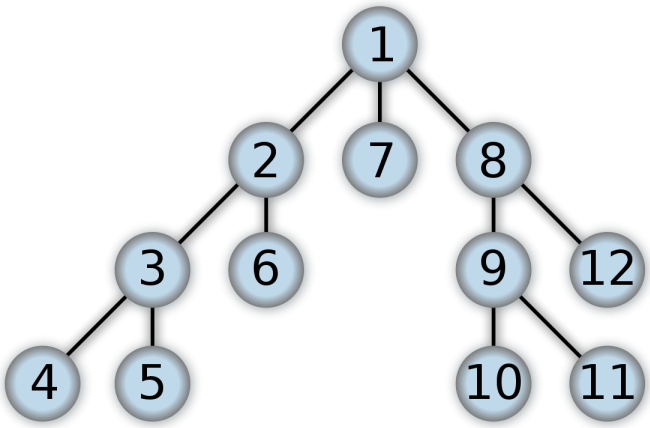
Princip tohoto algoritmu lze odvodit už z jeho názvu. Pro nerekurzivní přístup se v grafu vybere právě jeden uzel a z tohoto uzlu započne prohledávání. Uzel se vloží na zásobník uzlů, které se mají zpracovat. Dokud není zásobník prázdný, vybere se z něj jeden prvek a ten se zpracuje. V rámci zpracování se do zásobníku vloží reference na všechny přímé potomky aktuálně zpracovávaného uzlu. Tím se dokončí zpracování aktuálního uzlu a označí se za již zpracovaný. Celý postup se opakuje, dokud v zásobníku nedojdou uzly ke zpracování.

V případě potřeby zpracovat graf rekurzivně je realizace algoritmu velmi obdobná. V místě, kde by se uzly ukládaly do zásobníku, se pouze zavolá rekurzivně zpracovaní uzlu.

Pro zjištění cyklu je třeba hlídat, že vrchol, který má být zpracován, není již označen jako zpracovaný. V případě, že tento stav nastane, lze prohlásit, že graf obsahuje cyklus. Pro zjištění cesty, která vede na cyklus, je potřeba navíc ukládat seznam vrcholů reprezentující aktuální zanoření v grafu. V případě nalezení cyklu je cestou posloupnost vrcholů od vrcholu, u kterého algoritmus odhalil cyklus až do konce seznamu.

Za předpokladu rozumné implementace, kdy je přístup k potomkům daného uzlu v konstantním čase. Asymptotickou složitost celého algoritmu lze vyjádřit jako $O(|U|+|H|)$, kde U je množina uzlů a H je množina hran. Tato složitost odpovídá myšlence algoritmu, protože projde každý uzel a hranu právě jednou.

Na obrázku 3.7 je vizualizace prohledávání do hloubky. Uzel označený jako "1" je kořenem a z něj se zahájí prohledávání. Označení vrcholů odpovídá pořadí, ve kterém jsou vrcholy zpracovávány.



Obrázek 3.7: Prohledávání do hloubky

Zdroj: wikimedia.org - Depth-first tree

4 Reprezentace komponent

Komponenta společně s metadaty tvoří takzvaný balíček. Metadata v balíčku jsou vázána nejen ke komponentě v balíčku, ale i k balíčku samotnému. Metadata mohou být různá, každý repozitář, přesněji řečeno balíčkovací systém, má jiné požadavky. Jako absolutní minimum se dá označit následující dvojice informací.

- Id - jednoznačný identifikátor balíčku v rámci repozitáře
- Verze - verze balíčku

Balíček je třeba jednoznačně identifikovat, takže je potřeba, aby existovalo určité *Id*. Tento identifikátor může být v rámci repozitářů rozdílný, může to být samotné jméno balíčku, ale i jméno spolu s celým jmenným prostorem. Důležité je vědět, že se může jednat o znakový řetězec. Druhou podstatnou informací je verze. Balíčky podléhají verzování a v rámci repozitáře by nikdy neměl existovat balíček se stejným identifikátorem a stejnou verzí. V případě, že by balíčků existovalo více, nebylo by možné snadno rozhodnout, který balíček použít, a v repozitáři by vznikl konflikt.

Balíčky mohou mít i mnoho dodatečných informací, které lépe specifikují balíček jako takový. Případně informují omezení balíčku a jeho potřebné prerekvizity. Může se také jednat o informace, které jsou podstatné pouze pro repozitář a uživatel o nich není vůbec informován. V seznamu níže je několik vybraných informací, které balíčky běžně poskytují v rámci metadat, ovšem nejsou povinné.

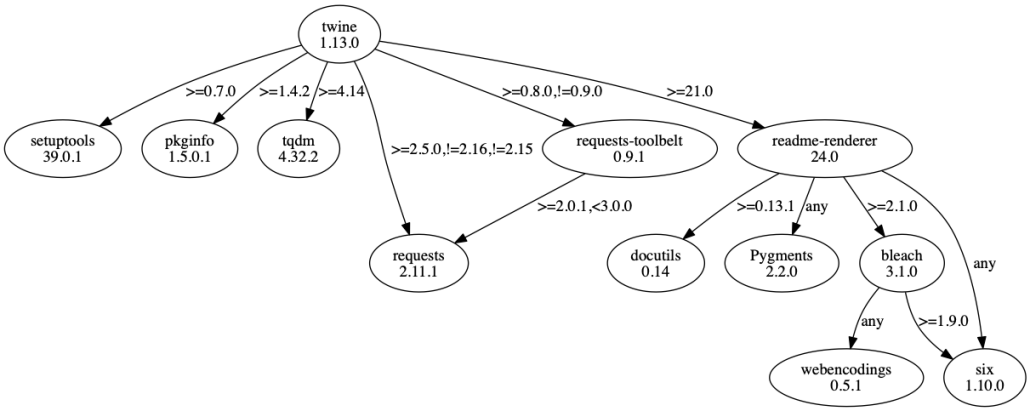
- Autor - jméno autora balíčku
- Závislosti - dodatečné závislosti, které balíček potřebuje
- Popis - popis balíčku, co obsahuje, jaký problém řeší
- Licence - pod jakou licencí je balíček poskytován
- Domovská stránka - poskytuje dodatečné informace o komponentě

4.1 Graf závislostí

Balíček může mít potřebu požadovat balíčky na kterých závisí, své závislosti. Pokud má balíček požadavky na závislosti a tyto závislostí mohou požadovat

další závislosti, vzniká graf závislostí, někdy také známý jako strom závislostí. Označení strom závislostí je ovšem trochu zavádějící, protože strom říká, že uzel je přístupný nanejvýše z jednoho jiného uzlu. V grafu závislostí se může stát, že je uzel referencován více uzly a tím se z něj stává opravdu graf, nikoliv strom.

Za kořen grafu závislostí se dá označit zkoumaný objekt, což může být právě jeden specifický balíček, který definuje závislosti na jiných balíčcích. V případě více zkoumaných balíčků najednou v jednom grafu závislostí je možné považovat za kořen každý z těchto zkoumaných balíčků, v tomto případě existuje kořenů více, bylo by ovšem rozumnější najít pro ně společný nadřazený prvek jakožto jeden kořen. Uzlem grafu je specifický balíček o určité verzi. Hrana je orientovaná, vyjadřuje že zdrojový balíček, odkud hrana vede, závísí na cílovém balíčku, do kterého hrana vede. Hrana primárně obsahuje informaci o tom, jaký balíček je požadován, případně další omezující pravidla pro splnění závislosti. Typicky se jedná o omezení na verzi, například větší nebo rovnu specifikované verzi. Případně se může jednat o závislost, která je pouze pro určitou část sestavení projektu. Balíček, do kterého hrana vede, musí podmínky splnit. Pokud balíček podmínku nesplní, graf závislostí je chybný a spuštění projektu skončí zcela jistě chybou, protože nejsou správně vyřešeny závislosti.



Obrázek 4.1: Příklad grafu závislostí

Zdroj: psfhosted.org - Python dependency tree

Na obrázku 4.1 se nachází vizualizace grafu závislostí pro specifikovaný balíček Twine o příslušné verzi 1.13.0, který slouží jako kořen grafu.

4.2 Vyřešení závislostí

Požadování balíčku, vytvoření závislosti na balíčku, znamená balíček před spuštěním sestavení projektu nejen získat, ale také vyřešit všechny jeho závislosti. Tento proces může být problém, protože nemusí skončit úspěšně. Každý balíčkovací systém si navíc řeší problém podle svých definovaných pravidel.

Za pojmem vyřešení závislostí se vlastně schovává pouze zjištění všech, tedy nejen přímých, závislostí zkoumaného balíčku nebo projektu a následného sestavení grafu závislostí. Graf se sestaví pomocí rekurzivního procházení všech závislostí, dokud není kompletní. Podle požadavků závislosti se vybere dostupný balíček, který splňuje podmínku a ten se stává uzlem v grafu. Ve chvíli, kdy byly vhodně vybrány všechny balíčky a splněny požadavky všech existujících hran, je graf správně sestaven.

Při sestavování grafu a vybírání vhodných balíčků ovšem začíná být problém, protože výběrem specifické verze balíčku z dostupných verzí splňujících podmínku vzniká určitá volnost, která může způsobit v následném řešení grafu závislostí kritickou chybu. Je tedy na balíčkovacím systému jak a jestli si s chybou dokáže poradit. Chyby, které mohou nastat, jsou podrobněji rozepsány v kapitole 5. Nejproblematičtější z pohledu sestavení grafu závislostí jsou konflikt verzí v sekci 5.2 a cyklická závislost v sekci 5.4.

4.3 Nuget

Pro platformu .NET existuje Nuget jakožto celý komponentový systém, tento systém má i svoje úložiště balíčků. Balíčky je možné vyhledávat a stahovat přímo z webové stránky. Tento přístup není doporučený, protože takto nebudou dodány případně další závislosti balíčku a projekt tak bude nefunkční. Vyhledáváč je ovšem vcelku schopný a poskytuje relevantní výsledky.

Doporučeným přístupem je používat nástroj *nuget*, pojmenovaný stejně jako repozitář. Tento nástroj se umí postarat o stažení balíčku z příslušného repozitáře. Zároveň je přímo integrován do vývojového prostředí Visual Studio, určeného pro vývoj pro platformu .NET, a tím velmi zjednodušuje práci se závislostmi.

Nástroj nuget řeší problém se závislostmi závislostí zhruba následujícím způsobem. Závislost se snaží splnit nejnižší možnou verzí balíčku, která splňuje omezující podmínky [6]. V případě, že graf obsahuje balíček v jiné verzi, je preferována verze blíže ke kořeni grafu. Tento přístup může být nebezpečný, a proto je uživatel varován o tomto rozhodnutí. V případě, že na

stejném patře grafu je definován balíček vícekrát, nuget zvolí nejnižší možnou verzi, která splňuje všechny podmínky. V případě, že podmínky nelze s dostupnými balíčky splnit, uživatel je informován o chybě a tím i nemožnosti vyřešit graf závislostí. V tomto případě je možné se pokusit chybu obejít tím, že sama aplikace bude uměle požadovat problematický balíček o specifické verzi. Jednoduše bude mít přímou závislost na problematickém balíčku. Tím, že je balíček požadován přímo kořenem stromu, pravidlo výběru verze podle nejbližšího patra zvolí nejbližší verzi balíčku. Tím bude problém pro nuget vyřešen.

Metadata balíčku v systému Nuget mají dle dokumentace povinné pouze 2 atributy [7]. Jedná se o název a verzi. V následujícím seznamu je výběr zajímavých atributů z pohledu analýzy balíčku.

- id - Povinný atribut, jméno balíčku a unikátní identifikátor v rámci repozitáře, reprezentován řetězcem
- version - Povinný atribut, verze balíčku, reprezentován řetězcem
- summary - Jednořádkový slovní popis balíčku, reprezentován řetězcem
- description - Delší slovní popis balíčku, reprezentován řetězcem
- projectUrl - Odkaz na domovskou stránku projektu, reprezentován řetězcem
- authors - Pole autorů balíčku, pouze jména, pro jedno jméno reprezentován řetězcem, pro více jmen polem řetězců
- licenseExpression - Pod jakou licencí je balíček poskytován, reprezentováno řetězcem
- licenseUrl - Odkaz na text licence, reprezentován řetězcem
- dependencyGroups - Pole dodatečných závislosti, které balíček potřebuje, reprezentováno polem objektů typu DependencyGroup

Skupina závislostí, neboli „DependencyGroup“ z atributů balíčku je definována následovně.

- TargetFramework - Cílová platforma, reprezentováno řetězcem
- dependencies - Pole závislostí, reprezentováno polem objektů typu PackageDependency

Jedna závislost z atributu závislosti ve skupině závislostí, neboli „DependencyGroup“ je definována následovně.

- id - Povinný atribut, jméno balíčku, reprezentováno řetězcem
- range - Rozsah verzí, reprezentován vlastním objektem
- registration - Odkaz na registrační index pro tuto závislost, reprezentován řetězcem

Rozsah verzí je v dokumentaci definován jako interval, není tedy možné skládat složitější omezení než intervalově vyjádřitelné. Například omezení na verzi větší než a zároveň nerovno tímto systémem nejde vyjádřit.

4.3.1 API

Pro Nuget je dostupné vcelku rozsáhlé API (Application Programming Interface). Práce s tímto API není příliš jednoduchá, protože jsou informace o balíčku distribuovány po několika koncových bodech [5]. Vývojový tým, který se stará o repozitář Nuget proto vytvořil v rámci několika balíčků implementaci klienta, která pomáhá s připojením k repozitáři typu Nuget. Klient dovoluje poskládat dotazy velmi snadno tak, aby bylo získáno co nejvíce informací o balíčku. Oproti tomu, vlastní implementace volání API metod by byla velmi pracná a bylo by třeba se dotázat více koncových bodů, než by byl uspokojivý výsledek získán. Balíčky s klientem je dostupný na následující adrese.

`https://www.nuget.org/packages/NuGet.Protocol`

`https://www.nuget.org/packages/NuGet.Packaging`

V rámci balíčku je obsažen celý komunikační protokol, který je potřeba pro práci s balíčky pomocí API. Rovněž jsou zde implementovány veškeré potřebné modely, které jsou potřeba pro komunikaci s API koncovými body.. Ve vývojovém repozitáři je několik ukázkových kousků kódu, které velmi pomohou s pochopením práce s klientem. V ukázkách je demonstrováno, jak získat metadata o balíčku a jak získat jeho závislosti.

4.4 PyPI

Pro jazyk Python existuje úložiště balíčků známé jako Python Package Index, zkráceně jako PyPI. Balíčky je možné vyhledávat a stahovat přímo z webové stránky. Ovšem pouhé stažení balíčku, bez instalace potřebných

prerekvizit, závislostí, způsobí, že nebude funkční. Je potřeba ještě dořešit získání potřebných závislostí.

Preferovanější přístup pro získávání balíčku je pomocí nastroje zvaného *pip*. Jméno nástroje je anglická rekurzivní zkratka z Pip installs packages, v překladu Pip instaluje balíčky. Jedná se o nástroj napsaný v Pythonu, který komunikuje s PyPI a stará se o získání balíčku a vyřešení jeho případných závislostí.

Nástroj pip řeší závislosti zhruba následujícím způsobem. Snaží se vybrat co nejaktuálnější verzi balíčku, jaký dle případných omezení lze vybrat [14]. Z těchto vybraných balíčků se opět snaží v rámci jejich závislostí vybírat co nejaktuálnější verzi balíčků, které splňují podmínku. Z případě výběru balíčku se nástroj podívá, pomocí zpětného prohledávání (backtracking) jestli zvolený balíček nezpůsobí problém s jinými, již vyřešenými závislostmi. Pokud ano, pokusí se vyměnit zvolenou verzi za nejbližší nižší, dokud mu to omezující pravidla dovolují. V případě, že se mu nepodaří najít vhodnou verzi, skončí s chybou. Pokud uspěje, použitá verze stále nemusí být finální, je možné, že bude v rámci jiného zpětného prohledávání změněna na ještě nižší. Graf závislostí se v rámci řešení může dost dramaticky měnit. Zvolením špatné verze se může celý podstrom příslušné závislosti, který z ní vzniká, předělat.

Metadata balíčku v systému PyPI mají dle dokumentace povinné pouze 3 atributy [13]. Jedná se o Název, Verzi a Verzi metadat. V následujícím seznamu je výběr zajímavých atributů z pohledu analýzy balíčku. Bohužel dokumentace neuvádí přesné datové typy, proto je jako výchozí považován řetězec.

- Metadata-Version - Povinný atribut, interní informace, udává verzi metadat, kterou balíček poskytuje
- Name - Povinný atribut, jméno balíčku a unikátní identifikátor v rámci repozitáře
- Version - Povinný atribut, verze balíčku
- Summary - Jednořádkový slovní popis balíčku
- Description - Delší slovní popis balíčku
- Home-page - Odkaz na domovskou stránku projektu
- Download-URL - Odkaz ke stažení balíčku
- Author - Autorovo jméno, případně dodatečné kontaktní informace

- License - Pod jakou licencí je balíček poskytován
- Requires-Dist - Dodatečné závislosti, které balíček potřebuje, pole objektů

Pro objekt vyjadřující jednu dodatečnou závislost je k dispozici následující seznam atributů.

- Name - Povinný atribut, jméno balíčku
- Version - Omezující pravidla na verzi, může jich být na řádce více, odděleny čárkou

4.4.1 API

Veřejné API (Application Programming Interface) poskytuje jeden hlavní koncový bod, který slouží k dotazování se na informace o balíčcích [21]. Je možné se dotázat na balíček a všechny jeho verze, případně pouze na specifickou verzi balíčku. Toto rozhraní neposkytuje možnost vyhledávat balíčky. Pokud není znám název balíčku, je potřeba jej zjistit jinak.

Pro získání informací o balíčku, včetně všech jeho verzí je možné použít následující koncový bod. Vrátí informace o nejvyšší verzi balíčku, včetně seznam ostatních verzí. Data jsou ve formátu JSON. Odkaz vypadá následovně:

```
https://pypi.python.org/pypi/<package_name>/json
```

Je potřeba doplnit jméno příslušného balíčku, například pro balíček se jménem „Flask“ vypadá odkaz následovně:

```
https://pypi.python.org/pypi/Flask/json
```

Pro získání informací o specifické verzi balíčku, bez nutnosti zjiš'tovat ostatní verze je možné použít následující koncový bod. Data jsou ve formátu JSON. Odkaz vypadá následovně:

```
https://pypi.python.org/pypi/<package_name>/<version>/json
```

Je potřeba doplnit jméno a verzi příslušného balíčku, například pro balíček se jménem „Flask“ o verzi „2.1.2“ vypadá odkaz následovně:

```
https://pypi.python.org/pypi/Flask/2.1.2/json
```

Na obrázku 4.2 je k dispozici fragment dat, které je poskytován koncovým bodem. Data jsou velmi obsáhlá co se počtu řádek týče a nevešla by se na stránku celá tak, aby nebyla narušena čitelnost.

```
{
    "info": {
        "author": "Armin Ronacher",
        "author_email": "armin.ronacher@active-4.com",
        "bugtrack_url": null,
        "classifiers": [ ···
        ],
        "description": "Flask\n=====\n\nFlask is a lightweight `WSGI`_ web appli
        "description_content_type": "text/x-rst",
        "docs_url": null,
        "download_url": "",
        "downloads": {
            "last_day": -1,
            "last_month": -1,
            "last_week": -1
        },
        "home_page": "https://palletsprojects.com/p/flask",
        "keywords": "",
        "license": "BSD-3-Clause",
        "maintainer": "Pallets",
        "maintainer_email": "contact@palletsprojects.com",
        "name": "Flask",
        "package_url": "https://pypi.org/project/Flask/",
        "platform": null,
        "project_url": "https://pypi.org/project/Flask/",
        "project_urls": {
            "Changes": "https://flask.palletsprojects.com/changes/",
            "Chat": "https://discord.gg/pallets",
            "Documentation": "https://flask.palletsprojects.com/",
            "Donate": "https://palletsprojects.com/donate",
            "Homepage": "https://palletsprojects.com/p/flask",
            "Issue Tracker": "https://github.com/pallets/flask/issues/",
            "Source Code": "https://github.com/pallets/flask/",
            "Twitter": "https://twitter.com/PalletsTeam"
        },
        "release_url": "https://pypi.org/project/Flask/2.1.2/",
        "requires_dist": [
            "Werkzeug (>=2.0)",
            "Jinja2 (>=3.0)",
            "itsdangerous (>=2.0)",
            "click (>=8.0)",
            "importlib-metadata (>=3.6.0) ; python_version < \"3.10\"",
            "asgiref (>=3.2) ; extra == 'async'",
            "python-dotenv ; extra == 'dotenv'"
        ],
        "requires_python": ">=3.7",
        "summary": "A simple framework for building complex web applications.",
        "version": "2.1.2",
        "yanked": false,
        "yanked_reason": null
```


Obrázek 4.2: Fragment výsledku dotazu na PyPI API

5 Problémy se závislostmi

Zavádění závislostí do aplikací je určitě přínosné, díky úspoře času, který by se ztratil vývojem příslušné komponenty. Problém je, že existence závislostí znamená automaticky potřebu závislost někde obstarat. Bez přítomnosti příslušné závislosti je aplikace nekompletní, mnohdy i nespustitelná.

V rámci této kapitoly jsou více vysvětleny problémy, které byly získány při studiu literatury zabývající se příslušnou problematikou. Rovněž jsou zde zohledněny podněty, které byly získány v rámci dotazování vývojářů zabývajících se komponentovým vývojem software.

5.1 Způsoby řešení závislostí balíčkovacími manažery

Pro nalezení vhodných komponent je rozumné použít vyhledáváč ze zvoleného repozitáře. Získání balíčku může být provedeno manuálně, ale pokud má balíček další závislosti je potřeba je vyřešit taktéž ručně. Je proto vhodné použít balíčkovací manažer příslušného systému, je-li k dispozici. Balíčkovací manažer se o veškeré operace spojené se získáním příslušného balíčku postará. Včetně vyřešení příslušných závislostí, pomocí stavby grafu závislostí. Následně obstará všechny potřebné balíčky, které jsou v grafu reprezentovány jako uzly. A tím by mělo být zajištěno, že aplikace nebude mít žádné chybějící přímé, či nepřímé komponenty z příslušných závislostí.

Je zde bohužel i určitá vlastnost celé myšlenky týkající se závislostí, která je vcelku nepříjemná. Jedná se o problémy, které vznikají v pozadí a na první pohled nejsou odhalitelné. V rámci vyřešení závislostí může balíčkovací manažer rozhodnout o získání balíčku, který je pro projekt z určitých důvodů nežádoucí. Už jen zjistit, že něco takového nastalo není triviální. Znamená to ručně zkontrolovat celý graf závislostí manuálně a hledat případný problematický balíček. Tento proces se vyloženě nabízí k automatizaci, pro uživatele by mohl být časově velmi náročný.

Nejhorší z problémů si uživatel může způsobit nevhodným výběrem přímých závislostí. Tento problém se také dá zjistit až při řešení grafu závislostí. Určitá kombinace závislostí může způsobit, že graf vůbec nelze správně sestavit. I s vhodným balíčkovacím manažerem dojde řešení grafu závislostí do stavu, že zahlásí chybu a ukončí řešení grafu závislostí se závěrem, že není možné požadavky na závislosti splnit. Tento problém také nemusí být okamžitě

vidět, minimálně ne při výběru samotného balíčku.

V případě, že se při vývoji objeví problém, který je způsobem závislostí, je to překážka k vývoji kvalitního software. Jak již bylo naznačeno, problém se závislostmi většinou vede na analýzu grafu závislostí. Už jen získat kompletní graf závislostí není triviální úkol.

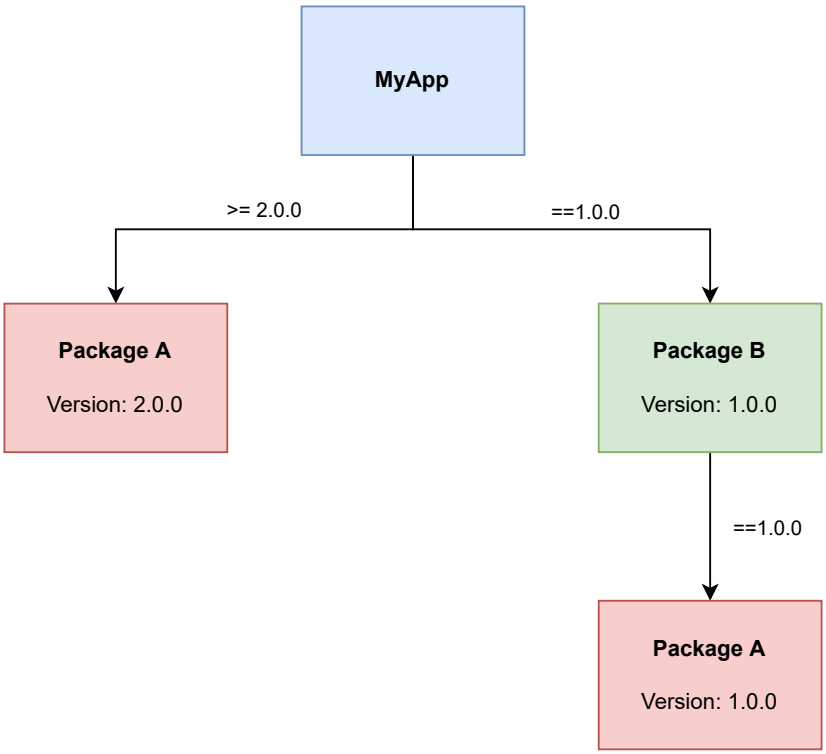
5.2 Konflikt verzí

Jedním z problémů, které většinou nejsou přímo vinou uživatele, je konflikt verzí. Jedná se o stav, kdy se v grafu závislostí vyskytne více hran s požadavky na obstarání určitého balíčku. Balíček, který bude vybrán musí plnit všechny požadavky, omezení na verzi balíčku. Pokud tomu tak není, graf závislostí není možné vyřešit.

Je důležité zohlednit volnost, která v grafu vznikne možností výběru vhodné verze balíčku podle omezujících podmínek. Pokud některý z předků dovoluje rozsah verzí balíčku, je možné vyzkoušet různé kombinace verzí, právě díky možnosti zkusit zaměnit předka za jinou verzi. V rámci verzí mohlo dojít ke změně, která konflikt verzí eliminuje. Tento přístup ovšem nemusí dojít ke správnému řešení, jedná se spíše o heuristiku, protože zaměňování verzí předků nemusí problém vyřešit. Navíc je to časově velmi náročné, pokud je rozsah příliš velký. Díky tomuto přístupu se sice může problém vyřešit, ale za cenu vyzkoušení všech kombinací balíčků, které je možné zaměňovat.

Na obrázku 5.1 je ukázka konfliktu verzí. Je zde balíček s označením „Package A“, který je požadován přímo aplikací, omezení je specifikováno na verze větší než 2.0.0. Další balíček, který aplikace požaduje, „Package B“ požaduje také „Package A“. Bohužel jej požaduje ve specifické verzi 1.0.0. Tím vzniká konflikt verzí, který není možné vyřešit ani zaměňováním rodičovských závislostí. Graf závislostí požaduje komponentu ve více různých verzích. V případě že komponenta se připojuje jako knihovna, mohou s tím mít některé nástroje problém. Například připojit dvě stejně pojmenované knihovny, kdy nástroj neví kterou z nich vybrat, tím při připojení knihovny nastal konflikt. Tento problém by se dal odhalit uživatelem, za předpokladu že si zkontroluje závislosti balíčku před tím, než se jej pokusí přidat. Díky tomu, že se jedná o přímou závislost příslušného balíčku, tak je to přímo v jeho metadatech. Za předpokladu, že by konflikt verzí nastal hlouběji v grafu, je již velmi obtížné jej uživatelsky odhalit bez použití specializovaného analytického nástroje. Použitím komponenty, která je od jiného poskytovatele je programátor nucen přijmout do svého projektu i všechny závislosti pří-

slušné komponenty, nemá kontrolu nad hlubšími závislostmi. Jediné co může programátor ovlivnit je výběr použitých přímých komponent, co je hlouběji, než tyto přímé komponenty, není v možnostech programátora ovlivnit.



Obrázek 5.1: Konflikt verzí v závislostech

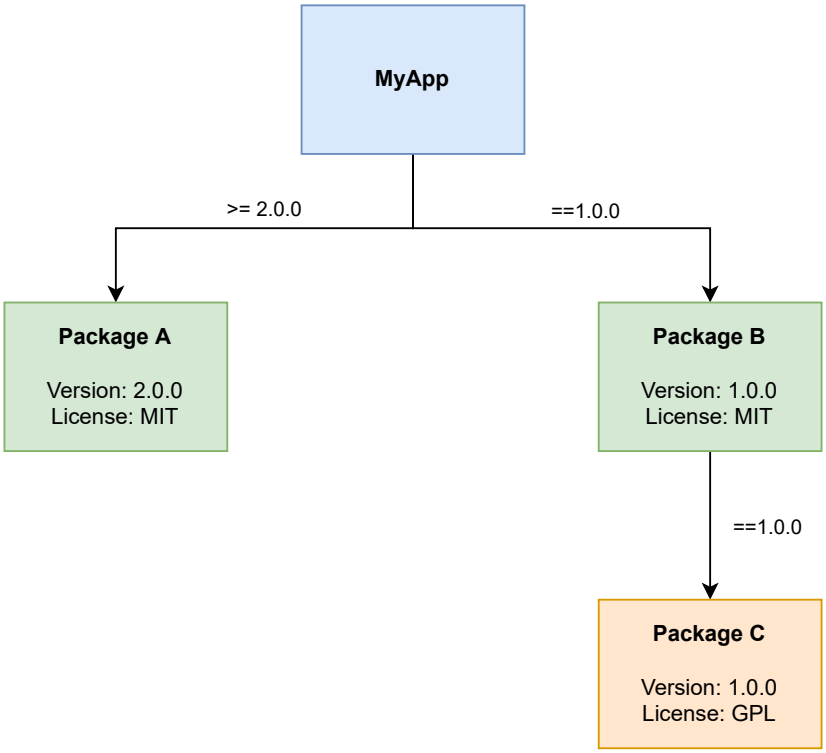
5.3 Licence

V případě použití balíčku je třeba zohlednit i jeho licenci, pod kterou je poskytován. Jsou různé druhy licencí. Některé pouze explicitně říkají, že je kód možné použít pro téměř libovolný účel. Existují i takové, které vyžadují aby licence, pod kterou bude distribuována aplikace byla stejná jako licence použitého balíčku. Pokud je balíček distribuován s licencí, která zakazuje komerční využití jedná se o vcelku markantní omezení při vývoji firemního software. Na následujícím odkazu je k dispozici porovnání různých licencí, včetně jejich vlastností.

https://choosealicense.com/licenses/

V případě přímých závislostí je tato kontrola vcelku snadná. Stačí pouze zkontrolovat licence požadovaných balíčků. V případě hlouběji zakořeněných balíčků se opět jedná o netriviální úkol, spojený se znalostí kompletního grafu závislostí. Je zapotřebí zkontrolovat celý strom a hledat v balíčcích problematické licence. Bez použití specializovaného nástroje je to velmi pracné, je třeba vědět jaký balíček o jaké verzi byl použit. Byť nástroj pouze porovná atribut o licenci vůči vyhledávacím parametrům.

Na obrázku 5.2 je vidět ukázka přítomnosti nevhodné licence. Aplikace nechce mít ve svých závislostech licenci typu GPL. Tato licence se nenachází v přímých závislostech, ale je hlouběji v grafu závislostí. Požadováním balíčku „Package B“ vznikla také závislost na „Package C“, který obsahuje licenci GPL.



Obrázek 5.2: Nevhodná licence v závislostech

5.4 Cyklická závislost

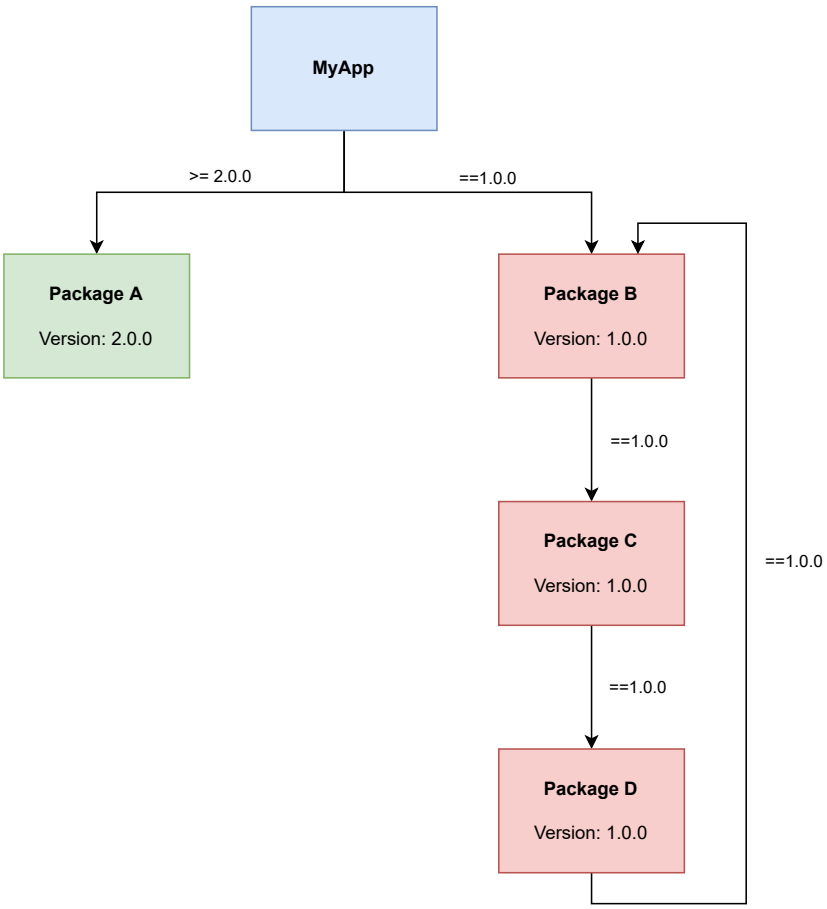
Tento specifický problém v závislostech je velmi závažný. V rámci řešení grafu závislostí se zjistí, že je určitou závislostí požadován balíček, který je v grafu závislostí předkem aktuálního balíčku. Jednodušeji řečeno, graf obsahuje cyklus. Balíčkovací nástroj, který narazí na tuto chybu se jí může snažit vyřešit záměnou verzí balíčků. Je možné, že se závislosti v rámci verzí změnily. Tento přístup ale nemusí vést k řešení problému. Balíčkovací nástroj tedy oznámí chybu a zdůrazní, proč neuspěl. Informuje o cyklické závislosti.

Doba, kterou potřebuje balíčkovací nástroj k tomu, aby oznámil chybu je vcelku dlouhá. Potřebuje projít různé kombinace balíčků, aby se pokusil cyklus eliminovat. Proto by bylo lepší mít k dispozici nástroj, který se pouze pokusí o zjištění, jestli se v grafu závislostí nenachází cyklická závislost. Tento nástroj by byl specializován na tento problém a poskytl informaci v rozumnějším čase, bez potřeby se snažit cyklus z grafu odstranit, jako to dělá balíčkovací manažer. Nalezení cyklické závislosti v grafu se dá realizovat pomocí algoritmu DFS, který se spustí nad kořenem grafu závislostí. Pokud najde balíček, který již zpracoval dokáže oznámit přítomnost cyklu. Díky informaci navíc, která říká jaké prvky v grafu se zpracovávaly a v jakém pořadí, je možné získat i cestu která cyklus způsobí. Opakováním postupu

nad všemi prvky lze získat seznam všech unikátních cyklů v grafu.

Odhalení tohoto problému před přidáním balíčku do aplikace je velmi náročné, uživatel bez spuštění specializovaného software nemá téměř šanci problém odhalit. Je třeba si postavit graf závislostí a ten postupně analyzovat. I pro specializovaný software to není časově triviální úloha.

Na obrázku 5.3 je ukázka cyklické závislosti. Závislost aplikace „Package B“ je požadován i jedním z jeho potomků. Požaduje jej „Package D“, který je v grafu závislostí pod ním. Tento cyklus způsobuje problém. V této ukázce je cyklus pouze o třech balíčcích, jedná se o ukázku. V praxi se může jednat o desítky balíčků.



Obrázek 5.3: Cyklická závislost v závislostech

5.5 Rozdíly závislostí mezi verzemi

V rámci vývoje aplikací se mohou měnit i jejich závislosti. Případně alespoň verze jejích závislostí. Děje se tak hlavně za účelem udržování aplikace a jejích závislostí aktuálních. Pokud je aplikace rozšiřována, může to být i kvůli rozšíření funkcionality aplikace. Potom je vhodné mít k dispozici nástroj, který porovná změny mezi grafy závislostí předchozích verzí vůči grafu závislostí aktuální verze aplikace.

Tato informace je přínosná hlavně z pohledu změny samotných závislostí. Pokud byla změněna verze specifické závislosti, díky porovnání se to dá snadno zjistit. Může se jednat o změnu způsobenou jiným přístupem k vyřešení stromu závislostí a tak by se ovlivnily i verze závislostí u kterých nebyla změna očekávána. V případě změny verze závislosti úmyslně, může být její podgraf dramaticky odlišný. Tím je možné získat například starší verze závislostí, než které byly v předchozí verze a tím do aplikace vnést i neúmyslně chyby. Tyto chyby mohou být v kódu komponenty, která je v balíčku, jež prošel snížením vlastní verze.

Analýza přítomnosti tohoto problému je netriviální. Manuální operace je jako ve většině případů velmi pomalá. Bylo by vhodnější použít specializovaný nástroj. Tento nástroj by provedl kontrolu a ohlásil případné změny, které nalezl.

V rámci podnětů z praxe bylo zjištěno, že tento problém je vhodné řešit. Není třeba ale nezbytně hledat změny v grafu jako takovém. Stačí pouze informovat o změnách v seznamu balíčků, který vznikne jako množina všech unikátních uzlů grafu závislostí. Dokonce je někdy požadováno pouze porovnat přímé závislosti a informovat o případných změnách. Tento podnět vznikl na základě požadavku dodávat k software i seznam jeho závislostí, které nejsou produktem firmy.

5.6 Počet závislostí

V rámci práce se závislostmi je i zajímavou informací počet použitých závislostí. Těchto metrik může být více. Je rozumné vědět přesný počet závislostí, to znamená počet uzlů v grafu závislostí. Ovšem přínosnější je znát počet unikátních závislostí, protože v rámci grafu závislosti se může mnoho z nich opakovat, i několikrát. Typicky se jedná o balíčky pro logování, nebo balíčky, které usnadňují práci s daty. Díky znalosti grafu závislostí, je i rozumné prezentovat počet přímých závislostí. Pokud se bude v rámci vývoje počet rapidně zvedat, je rozumné přehodnotit, zda-li jsou všechny balíčky opravdu nutné.

Tato analýza je bez přítomnosti grafu závislostí vcelku obtížná. Opět je potřeba projít všechny závislosti a graf závislostí vytvořit. S existujícím grafem závislostí se jedná pouze o problém procházení grafu. V rámci procházení je třeba vhodně ukládat již navštívené nové vrcholy a udržovat si počet analyzovaných vrcholů. Po skončení procházení stačí pouze informovat, kolik vrcholů bylo analyzováno. Pro počet unikátních balíčků stačí získat počet uložených unikátních vrcholů.

5.7 Shrnutí

Představené problémy nejsou pouze teoretické, ale jedná se o podněty z praxe. Potřeba analyzovat graf závislostí a poskytnout výstupy, které pomohou s řešením představených problémů není triviální úkol. Existující problémy jsou blokující ve vývoji softwaru a je potřeba je řešit. Například firemní vývoj potřebuje u svých produktů informaci o tom, zda li všechny závislosti, které byly v rámci projektu použity splňují podmínky pro komerční použití.

Grafy závislostí navíc mohou dosahovat i pro malé projekty, mající jednotky přímých závislostí, několika stovek uzlů. Pro rozsáhlé firemní projekty se může snadno graf závislostí rozrůst i nad tyto hodnoty. Manuální analýza takovýchto grafů je tedy velmi pracná a časově náročná. Velikost grafů samotných je již závažnou překážkou. Z těchto podnětů vzniká potřeba automatizovaného přístupu k řešení problému a nutnost mít k dispozici nástroje, které provedou potřebnou analýzu.

6 Obecný úložný formát

V rámci analýzy reprezentace komponent v kapitole 4 bylo zjištěno, jaká data poskytují konkrétní repozitáře. Jedná se hlavně o repozitáře Nuget a PyPI. Z těchto repozitářů silně vychází návrh datového modelu obecného formátu, jehož popis je předmětem této kapitoly.

Na základě výstupu z analýzy reprezentace komponent v kapitole 4 byla sestavena množina informací, kterou je třeba do úložného formátu ukládat. S přihlédnutím k potřebám následné analýzy byly výstupy analýz také zahrnuty do množiny informací, které je třeba ukládat. Do úložného formátu je třeba uložit informace tak, aby uložením neztrácely svou informační hodnotu. Mimo samotné závislosti je přínosné ukládat také informace o projektech, ve kterých se závislosti nachází.

Pro zajištění maximálního pokrytí všech potřeb by model měl obsahovat u závislosti i slovník atributů. Tento slovník by měl jako klíč typ řetězec (String). Hodnotou slovníku by byla nejobecnější volba, která lze aplikovat. Jedná se o typ objekt (Object), praotec všech existujících objektů. Zde bylo potřeba rozhodnout, jestli existuje specifičtější volba. Například zvolit také řetězec, seznam řetězců, nebo vlastní datovou strukturu. Ve všech případech by došlo k omezení ukládání hodnot do slovníku. Tím by došlo i k zvýšení pravděpodobnosti výskytu případů, které povedou k závažné změně datového modelu. Tyto změny by navíc mohly velmi snadno zamezit zpětné kompatibilitě již existujících datových souborů s tímto univerzálním formátem.

Práce s daty ve slovníku typu objekt je trochu náročnější. Základní vlastnost, možnost převést každý objekt na řetězec bude ovšem k dispozici. Pokud bude objekty možné převádět na srozumitelné textové reprezentace, je možné dodávat objekty a dosáhnout zpětné kompatibility s nástroji, které takovéto objekty neočekávaly.

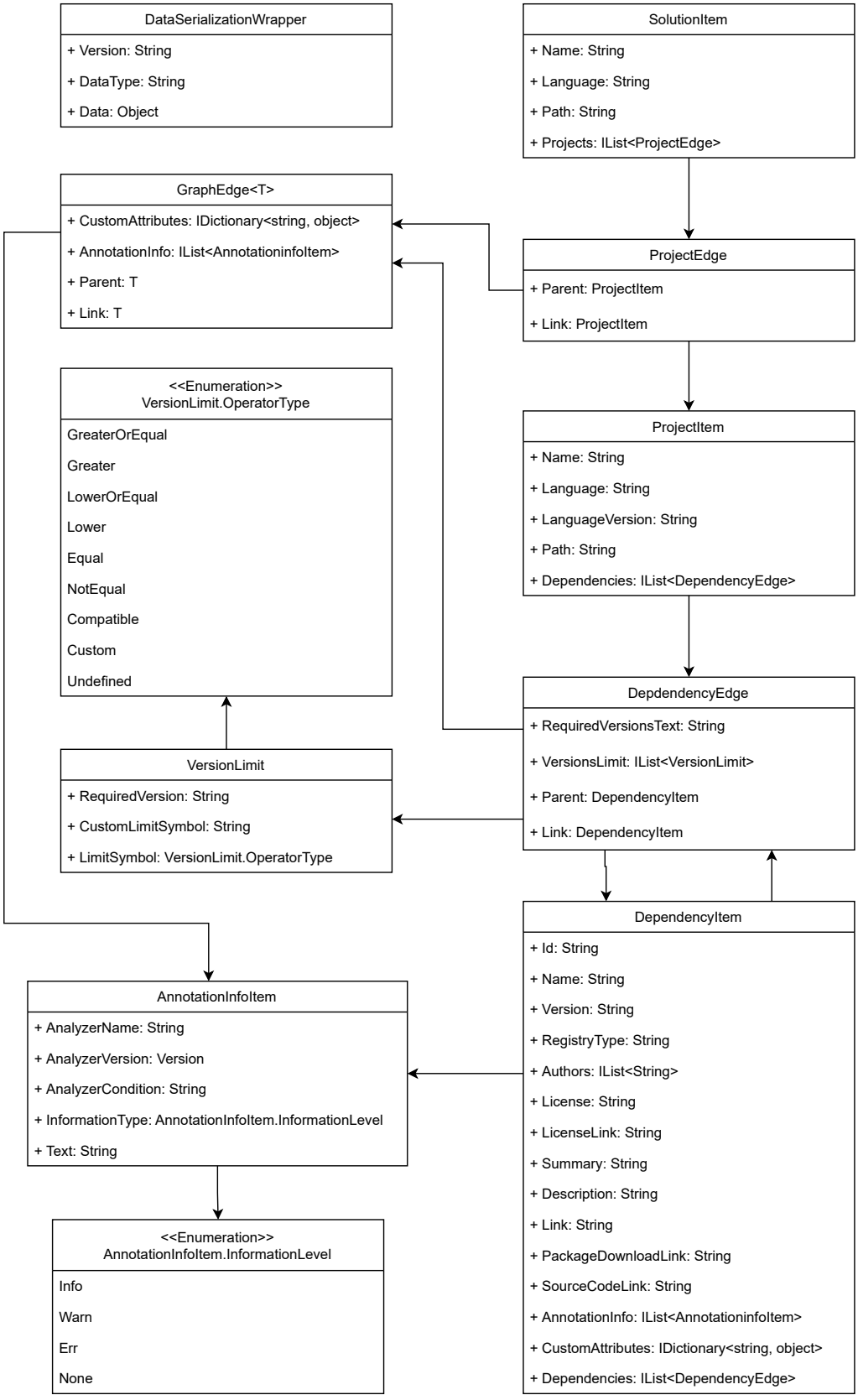
Pokud byla potřeba uložit do slovníku vlastních atributů speciální objekt, lze předpokládat, že je třeba jej i analyzovat. Analyzátor, který se postará o příslušné řešení by tedy měl umět s takovýmto objektem pracovat. S vysokou pravděpodobností se bude jednat o výsledek práce téhož programátora, nebo týmu, který měl potřebu uložit tento specifický objekt.

Je žádoucí aby výsledky analyzátorů bylo možné do modelu taktéž snadno ukládat a tím vytvářet anotovaná data. Takto anotovaná data by mělo být možné za pomoci analyzátorů tvořit tak, aby se neovlivnil existující obsah. Přesněji řečeno, aby se neovlivnil jinak, než přidáním anotace k specifické

struktuře. Tato anotace nesmí způsobit žádné problémy v případě opětovného spuštění se stejným analyzátorem. Je tedy možné opakovat analýzu s konzistentním výsledkem. Jediný problém je, že v případě opětovného spuštění budou anotace duplicitní. Tento fakt je ovšem z myšlenky použití analyzátorů zcela v pořádku. Analyzátor by neměl v datové struktuře ovlivnit výsledky předchozích analýz.

Návrh datové reprezentace anotace z analyzátorů je zobrazen v diagramu datového modelu na obrázku 6.1, pod jménem „AnnotationInfoItem“. Tato datová položka slouží k reprezentaci jedná anotace, vytvořené analyzátorem. Anotace uchovává jméno a verzi analyzátoru, který anotaci poskytl. Je zde i podmínka, jeli k dispozici, která analyzátor přiměla anotaci vytvořit. Typicky se jedná o podmínky, které vyhledávají určité vlastnosti v grafu závislostí. Například vyhledávání všech závislostí, kde hodnota atributu odpovídá vyhledávacím parametrům. V rámci položky je k dispozici i možnost specifikovat typ anotace, tento atribut může způsobit zvýraznění ve vizualizačních nástrojích, případně v textové reprezentaci. Je k dispozici několik základních úrovní typů anotace, informační (Info), upozornění (Warn), chyba (Err) a nespecifikovaný typ (None).

Úložný formát je navrhnut tak, aby se dal snadno do budoucna rozšiřovat. V případě, že by chyběl určitý atribut je možné model rozšířit. Je možné i model obohatit o další struktury, které se v rámci používání projeví jako užitečné, či nutné. Navrhovaný model obsahuje atributy, které pokrývají potřeby zmíněných repozitářů a jsou vhodné pro následnou analýzu.



Obrázek 6.1: Datový model univerzálního úložného formátu

6.1 Ukládání

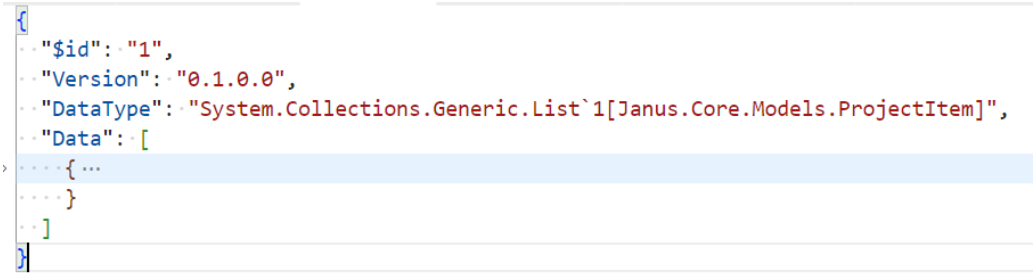
Způsob uložení obecného formátu byl zvolen s přihlédnutím k výstupu analýzy v kapitole 3. Byl zvolen formát JSON, který je představen v sekci 3.2.1. Formát vyhovuje k použití s vytvořeným modelem a typu předávaných dat. Jeho serializace a deserializace je rovněž velmi přímočará. Původně byl zvažován formát XML, ale ten se projevil jako nevhodný. Serializace formátu XML se setkala s několika problémy, souvisejících hlavně se slovníkem. Dostupné serializátory nedokážou provést bezchybnou serializaci tohoto abstraktního datového typu. Rovněž se potýkaly s problémy v podobě serializace atributů, definovaných jako rozhraní. Práce s formátem JSON byla mnohem jednodušší a nevykazuje chyby, minimálně ne ty, se kterými se potýkal formát XML. Grafová databáze, i když je na tento druh dat perfektně určena, nebyla vybrána protože práce s grafovou databází by znamenala mít databázi nasazenou a dostupnou pro všechny případy použití, což může být nevhodným omezením. Oproti tomu sdílení datových souborů je v tomto ohledu univerzálnější.

Formát JSON je textový a je velmi snadno přenositelný. V případě nutnosti je možné jej i ručně editovat a tím upravit uložená data. Editaci je možné provést z běžného textového editoru, není vyloženě potřeba specializovaný nástroj. Díky textové reprezentaci lze rovněž data předávat mezi nástroji za pomoci přesměrování výstupu nástroje ze standardního výstupu do vstupu dalšího nástroje. Případně je možné jej uložit přímo do souboru, nebo si jej nechat pouze vypsat na standardní výstup.

Velmi příjemnou vlastností formátu JSON je možnost referencovat duplicitní objekty. V případě rozsáhlého stromu závislostí je vysoká pravděpodobnost opakujících se závislostí. Tyto duplicitní položky stačí uložit jako jednu, s kompletními daty, a ostatní pouze nareferencovat na tuto specifickou položku. Vyžaduje to ovšem přítomnost dodatečného identifikačního atributu.

6.2 Datový obal

Aby bylo možné datový soubor volně předávat mezi nástroji, je potřeba v rámci předávaných dat informovat i o datovém typu, který se předává. Rovněž je třeba hlídat, jestli nástroj, který se chystá data přijmout je vůbec schopen je načíst. V případě, že byl datový formát pozměněn, je možné že se ztratila kompatibilita mezi použitým modelem a modelem obsaženým ve zmíněném nástroji. Proto je v rámci dat předávána i informace o verzi navrženého univerzálního datového formátu. Kořenovým objektem JSONu je ob-



Obrázek 6.2: Ukázka reprezentace datového obalu v JSONu

jekt „DataSerializationWrapper“, který obsahuje pouze zmíněné informace. Jedná se o určitý obal, který obaluje předávaná data. Verzi univerzálního datového formátu, datový typ předávaných dat a data jako taková. Datový obal je obsažen v diagramu datového modelu na obrázku 6.1, pod jménem „DataSerializationWrapper“. Ukázka datového obalu v JSONu je vidět na obrázku 6.2.

Předávaná data mohou být různá, dokonce se mohou přidat i další, mimo ta již vytvořená v rámci této práce. Je vhodné preferovat předávání dat za pomoci seznamů, i kdyby se mělo jednat o jeden prvek. Datový formát sice dovoluje předat pouze jeden prvek samostatně, ale analyzátory s tím nemusejí počítat. Primárně jsou stavěny na to očekávat větší množství dat, takže zcela jistě musí mít implementaci která dokáže načíst seznamy položek. Mezi základní datové typy, vytvořená v rámci této práce, se řadí následující položky. Více o těchto položkách je popsáno v příslušných sekcích v této kapitole.

- Seznam závislostí
- Seznam projektů
- Seznam řešení

6.3 Závislost

Nejpodstatnější a nejzákladnější ze všech položek je závislost. Jedná se o datovou reprezentaci jednoho balíčku, závislosti. Položka je zobrazena v diagramu datového modelu na obrázku 6.1, pod jménem „DependencyItem“. Její atributy jsou navrženy tak, aby pokryly co nejobecněji potřeby pro uložení informací o závislosti. Pokrývá běžně dostupná metadata, které je možné do něj uložit. V případě, že bude třeba uložit dodatečná metadata,

která v modelu nejsou, je možné je uložit do slovníku označovaného jako „CustomAttributes“.

Závislost má jednoznačný identifikátor, který je označen jako „Id“, jedná se o řetězec. Tento identifikátor by měl jednoznačně definovat komponentu. Oproti tomu, atribut „Name“ udává jméno komponenty, opět se jedná o řetězec. Tyto dva atributy mohou být v mnoha případech stejné. Pokud repozitář rozlišuje mezi názvem komponenty a plnohodnotným identifikátorem, tato vlastnost neplatí a proto je doporučeno nespoléhat se na unikátnost atributu „Name“. Jednoznačný identifikátor v kombinaci s atributem verze, v modelu jako „Version“, společně jednoznačně identifikují balíček. Atribut verze je uložen jako řetězec, protože verze může obsahovat mimo čísel i písmena. Například verze může být definována jako 1.1.0-alpha. V rámci univerzálního datového formátu je potřeba ještě pro naprosto jednoznačnou identifikaci doplnit informaci o repozitáři, ve kterém je balíček dostupný. Tato informace je dostupná v atributu „RegistryType“, jedná se o řetězec. Výčtový typ není v tomto případě rozumný, protože je nepraktické přidáním repozitáře požadovat rozšiřování datového formátu.

Model obsahuje i dodatečné atributy, sloužící hlavně po specializované analyzátory. Jedním z dodatečných atributů je informace o jménech autorů komponenty, uloženo v atributu „Authors“, jedná se o seznam řetězců. Zajímavější informací, z hlediska analýz, je atribut pro uložení licence komponenty. Jedná se o atribut „License“, je reprezentován jako řetězec. V tomto atributu by měla být uloženo jméno použité licence, nikoliv licence celá. Pro uložení licence slouží atribut „LicenseLink“, je to řetězec uchovávající odkaz k licenci příslušného balíčku.

Pro popisné atributy balíčku jsou k dispozici následující atributy. Atribut „Summary“, jakožto řetězec, ukládá stručný popis balíčku. Většinou by se mělo jednat o jednořádkové popisky. Pro delší popis slouží atribut „Description“, jedná se o delší řetězec popisující balíček. Je možné je využít při analýze pro vyhledávání podobných balíčků, podle určitých klíčových slov.

Atributy uchovávající odkazy jsou také k dispozici. Jedná se o atribut „Link“, který ukládá odkaz na balíček v repozitáři. Oproti tomu atribut „PackageDownloadLink“ uchovává odkaz na balíček jako takový. Pomocí tohoto odkazu by mělo být možné stáhnout si samotný balíček. Posledním z odkazových atributů je „SourceCodeLink“, jedná se o odkaz na zdrojový kód komponenty. Typicky se jedná o odkazy na vývojářské repozitáře, typu Github, Gitlab a jiné.

V rámci analýz může být potřeba uložit anotaci k samotné závislosti. K tomuto účelu složí atribut „AnnotationInfo“, jedná se o seznam anotačních

Obrázek 6.3: Ukázka reprezentace závislosti v JSONu

položek, protože anotací může být potřeba uložit více.

Závislost může definovat další své závislosti, proto je zde přítomen atribut „Dependencies“. Tento atribut je seznamem hran k závislostem aktuální závislosti. Hrana k závislosti je podrobněji vysvětlena v následující sekci 6.4.

Formát je navržen tak aby neobsahoval prakticky žádné povinné atributy. Jako rozumné minimum, kdy má vůbec smysl závislost ukládat je dvojice jednoznačného identifikátoru a verze. Ostatní atributy jsou sice určené pro dodatečnou analýzu, ale povinné nejsou. Doplnění hodnot je v režii nástrojů, množina vyplněných atributů je podle jejich schopností a schopností repozitářů, jaká metadata poskytnou. Ukázka reprezentace závislosti v JSONu je vidět na obrázku 6.3.

6.4 Hrana k závislosti

Jedná se o reprezentaci hrany vedoucí od objektu, ve kterém je hrana uložena, do cílové závislosti. Hrana k závislosti je zobrazena v diagramu datového modelu na obrázku 6.1, pod jménem „DependencyEdge“.

Tato položka datově reprezentuje hranu mezi závislostmi, případně mezi projektem a závislostí. Uchovává informaci o cílové závislosti v rámci atributu „Link“. V atributu „Parent“ je uchováván zdroj, odkud hrana vede. V atributu je uložena reference na objekt reprezentující závislost. Hrana udává jaká závislost má být obstarána. Obsahuje omezující podmínky, které musí obstaraná závislost, referencována hranou, plnit. Atribut „VersionsLimit“ je

seznamem objektů reprezentujících omezení na verzi. Dalším atributem je „RequiredVersionsText“, definován jako řetězec, jedná se o textovou reprezentací všech omezujících podmínek, vyjádřených jako text a zřetězených dohromady.

V případě, že projekt definuje své závislosti, opět se dá použít tento model, jako hrana mezi projektem a jeho závislostí. V tomto případě je zamýšleno ponechat rodičovský atribut prázdný. Tento stav popisuje že se jedná o přímou závislost nad projektem.

Pokud je třeba, dovoluje hrana uložit i vlastní atributy, stejným mechanismem jako to dovoluje závislost. Pomocí slovníku v příslušném atributu „CustomAttributes“. Zároveň je zde i možnost uložení anotací z analyzátorů do atributu „AnnotationInfo“, který je seznamem anotačních položek. Opět se jedná o obdobný přístup, jako při anotování závislostí.

6.4.1 Limit verzí

Limit verzí je definován položkou „VersionLimit“, je zobrazena v diagramu datového modelu na obrázku 6.1. V atributu „RequiredVersion“ uchovává verzi, která je spjatá s tímto omezením, ve formě řetězce. Další atribut, „LimitSymbol“ je určen pro symbol, kterým je omezení definováno. Jedná se například o symbol „>=“, který říká větší nebo rovno. Dostupné symboly jsou v příslušném výčtovém typu „VersionLimit.OperatorType“, výčtový typ je zobrazen v diagramu datového modelu na obrázku 6.1. Výčtový typ definuje nejběžnější operátory.

- Větší nebo rovno (>=)
- Větší (>)
- Menší nebo rovno (<=)
- Menší (<)
- Rovno (==)
- Nerovno (!=)
- Kompatibilní (∼=)

V případě, že operátor není k dispozici ve výčtovém typu, je možné v něm zvolit typ „Custom“, který říká že bude definován vlastní operátor. Vlastní operátor je možné uložit do k tomu určenému atributu. Jedná se o atribut „CustomLimitSymbol“, reprezentován jako řetězec, v rámci objektu reprezentujícího limit verzí.

6.5 Seznam závislostí

Preferovanou volbou, při předávání závislostí, bez přítomného projektu, je seznam závislostí. Nejedná se tedy, jako u projektu o seznam hran k závislostem. V případě potřeby předávat seznam závislostí se předpokládá, že závislosti byly vydefinovány správně a není k dispozici rodičovský prvek, na který závislosti navázat. Tento přístup může být výhodný za předpokladu požadavku analýzy výčtu závislostí, jak jejich přítomnost ovlivní výsledný graf závislostí.

6.6 Projekt

Jedná se o datovou reprezentaci projektu. Jedná se o jeden konkrétní vývojový projekt, který může definovat závislosti. Projekt je zobrazen v diagramu datového modelu na obrázku 6.1, pod jménem „ProjectItem“. Atributy byly zvoleny s přihlédnutí ke zkušenostem s různými projekty různých programovacích jazyků.

Nejvýznamnější atribut, označení jako „Dependencies“ uchovává seznam hran k závislostem. Do tohoto atributu je potřeba uložit všechny potřebné závislosti analyzovaného projektu. Bez naplnění tohoto atributu téměř nemá smysl projekt vůbec ukládat, i tak není tato položka povinná.

Dodatečné atributy slouží k uložení podpůrných informací. Kvůli získání vhodných závislostí může být nutné zprostředkovat verzi použitého jazyka, k tomu složí atribut „LanguageVersion“, který uloží verzi jazyka jako řetězec. V analogii k verzi, může být vhodné uložit i typ jazyka jako takový, k tomu slouží atribut „Language“, který uchovává typ jazyka reprezentovaný jako řetězec. Další z dodatečných atributů, označený jako „Name“, slouží k uložení jména projektu v textové podobě. Je to užitečné při zkoumání a rozpoznávání, který projekt vyžadoval problematické závislosti. K použití je i poslední atribut, „Path“, který uchovává cestu k projektu v rámci adresářové struktury. Tento atribut je zde hlavně pro snadné nalezení problematického projektu. Je třeba zdůraznit, že cesta je platná pouze z počítače, na kterém se nachází projekt, podle kterého byla tato položka naplněna. V jiných počítačích se mohou cesty lišit, tato informace není tedy spolehlivě přenositelná mezi počítači a měla by se používat pouze orientačně, hlavně v rámci stejného počítače. Ukázka reprezentace projektu v JSONu je k dispozici na obrázku 6.4.

```
"$id": "13",
"Name": "Janus.Core",
"Language": "C#",
"LanguageVersion": "netstandard2.1",
"Path": "C:\\Users\\milan\\git\\janus\\src\\Core\\Janus.Core\\Janus.Core.csproj",
"Dependencies": [ ···
]
```


Obrázek 6.4: Ukázka reprezentace projektu v JSONu

6.6.1 Hrana k projektu

Jedná se o reprezentaci hrany vedoucí od objektu, ve kterém je hrana uložena, do cílového projektu. Hrana k projektu je zobrazena v diagramu datového modelu na obrázku 6.1, pod jménem „ProjectEdge“.

Tato položka datově reprezentuje hranu mezi projektem a řešením. Uchovává informaci o cílovém projektu v rámci atributu „Link“. V atributu „Parent“ lze uchovávat referenci na rodičovský projekt. V atributu je uložena reference na objekt reprezentující projekt. Potřeba řetězit projekty by mohla nastat a proto je model na toto připraven. V případě, že se váže projekt na řešení, lze také použít tuto hranu. Atribut s referencí na rodičovského předka lze ponechat prázdný.

V případě potřeby dovoluje hrana uložit i vlastní atributy, stejným mechanismem jako to dovoluje závislost. Za pomoci slovníku v příslušném atributu „CustomAttributes“. Zároveň je zde i možnost uložení anotací z analyzátorů do atributu „AnnotationInfo“, který je seznamem anotačních položek. Opět se jedná o obdobný přístup, jako při anotování závislostí.

Hrana neobsahuje žádné další atributy. Vlastně se jedná jen o specializovanou položku typu obecné hrany v grafu, s tím že typ odkazu je specifikován na položku projektu. Rozhodnutí použít oddědený objekt bylo provedeno proto, že takto se dá do budoucna snadno přidat atribut do hrany k projektu. Bez nutnosti upravovat příliš rozsáhle model a již implementované nástroje.

6.7 Seznam projektů

Preferovanou volbou, při předávání projektů, bez přítomné závislosti, je seznam projektů. Nejedná se tedy, jako u řešení o seznam hran k projektům. V případě potřeby předávat seznam projektů se předpokládá, že projekty byly nemají k dispozici rodičovský prvek, na který je lze navázat. Tento přístup může být výhodný za předpokladu požadavku analýzy výčtu projektů, jak jejich spojení ovlivní výsledný graf závislostí.

```
{
  "$id": "2",
  "Name": "Solution-Janus",
  "Language": "C#",
  "Path": "C:\\Users\\milan\\git\\janus\\src\\Janus.sln",
  "Projects": [ ···
  ]
}
```


Obrázek 6.5: Ukázka reprezentace řešení v JSONu

6.8 Řešení

Jedná se o datovou reprezentaci tvz. „Solution“, přeloženo jako řešení. Obsahuje seznam společně vyvíjených projektů. Řešení je zobrazeno v diagramu datového modelu na obrázku 6.1, pod jménem „SolutionItem“. Atributy byly zvoleny s přihlédnutí ke zkušenostem s prostředím Visual Studio, určením pro vývoj v rámci .NET platformy. V rámci analýzy byl jediný, který něco takového poskytoval.

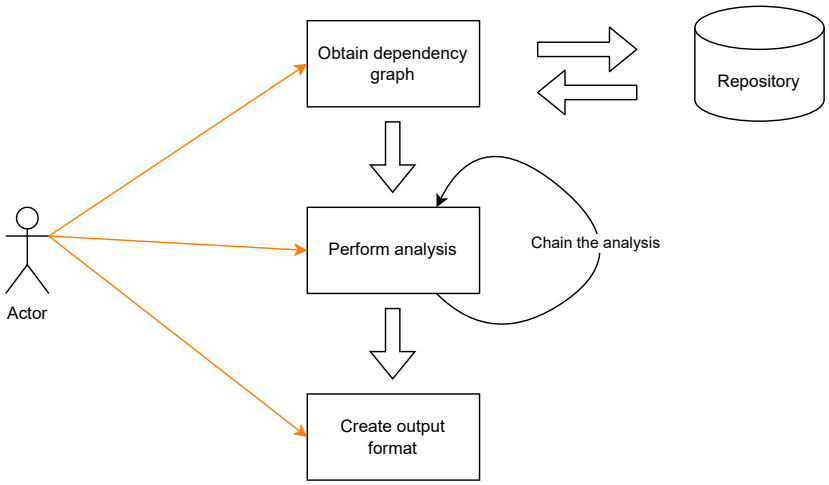
Objekt je vlastně jen kontejner pro uložení seznamu projektů, přesněji seznamu hran k projektům. K tomuto účelu slouží atribut „Projects“. Jsou zde i dodatečné atributy, které pomáhají k lepší specifikaci řešení. Jedním z nich je „Name“, udává textovou reprezentaci jména řešení. Dalším je „Language“, jedná se specifikaci jazyka použitou v rámci řešení. Posledním je „Path“, který uchovává cestu k řešení v rámci adresářové struktury. Tento atribut je zde hlavně pro snadné nalezení zpracovávaného řešení. Je třeba zdůraznit, že cesta je platná pouze z počítače, na kterém se nachází řešení, podle kterého byla tato položka naplněna. V jiných počítačích se mohou cesty lišit, tato informace není tedy spolehlivě přenositelná mezi počítači a měla by se používat pouze orientačně, hlavně v rámci stejného počítače. Tento problém je analogický jako u projektu. Ukázka reprezentace řešení v JSONu je k dispozici na obrázku 6.5.

6.9 Seznam řešení

Preferovanou volbou, při předávání řešení, je seznam řešení. V případě potřeby předávat seznam projektů se předpokládá, že projekty byly nemají k dispozici rodičovský prvek, na který je lze navázat. Tento přístup může být výhodný za předpokladu požadavku analýzy výčtu řešení, spojení všech grafů závislostí z každého řešení do jednoho jediného souboru univerzálního datového formátu.

7 Architektura navrženého frameworku

Návrh architektury vychází z workflow k analýze grafu závislostí, zobrazena na obrázku 7.1. Hlavním problémem je získání samotného grafu závislostí, o řešení tohoto problému se starají nástroje zvané „Parsery“, více v sekci 7.4. Nad získaným grafem závislostí je potřeba provádět analýzy, tento problém řeší specializované analyzátory, více o analyzátorech v sekci 7.5. Pro výměnu dat mezi nástroji byl použit univerzální datový formát k uložení grafu závislostí, popsaný v kapitole 6.

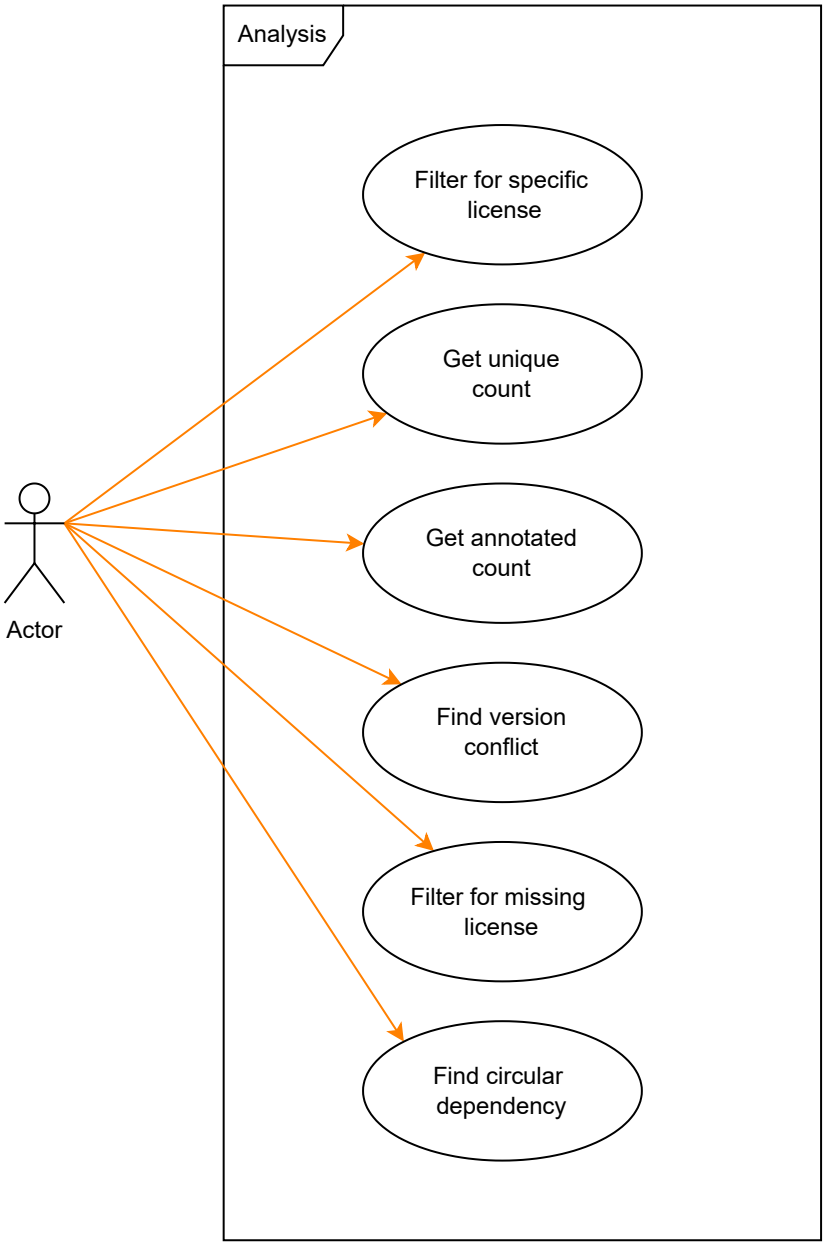


Obrázek 7.1: Workflow k analýze grafu závislostí

Požadavky na navrhovanou architekturu jsou popsány na obrázku 7.2. Je důležité vyzdvihnout potřeby různých druhů analýz dat a případných filtrací. V rámci návrhu architektury je třeba splnit všechna očekávání plynoucí z příslušných případů užití. Navíc by bylo ještě vhodné docílit stavu, kdy výsledky analýzy je možné pomocí tvz. roury přesměrovat na vstup dalšího analyzátoru a tím řetězit analyzátory. Tento případ užití vyplývá z workflow na obrázku 7.1.

Datové struktury navrženého formátu pro předávání grafu závislostí jsou uloženy ve společném jádře, mimo to je v jádře přítomna i společná logika pro vstup a výstup dat, více v sekcích 7.2 a 7.3. Celý navržený framework tedy má tři hlavní bloky, jádro, parsery a analyzátory.

Architektura je navržena s důrazem na snadnou rozšiřitelnost, předávání dat mezi nástroji je umožněno díky navrženému univerzálnímu datovému



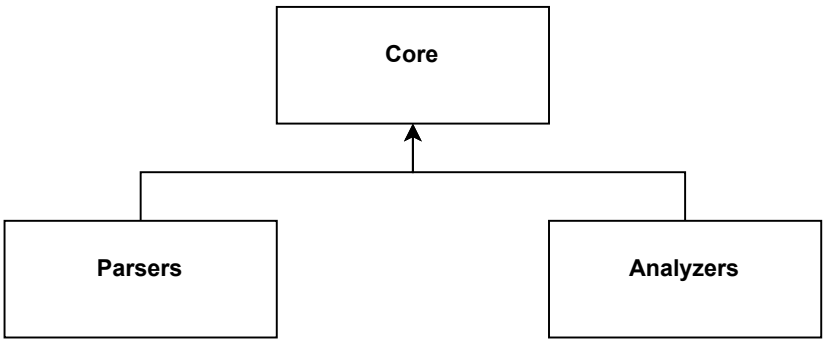
Obrázek 7.2: Případy užití architektury

formátu pro uložení grafu závislostí. Je vhodné, ale ne nezbytně nutné, využít logiku pro importování a exportování zmíněného formátu dat dostupnou z jádra. Díky jedinému omezení, v podobě předávaného datového formátu, je velmi snadné vytvářet další nástroje, které mohou rozšiřovat existující dostupnou sadu.

Rozdělením na jednotlivé nástroje, které se specializují na určitou funkcionalitu, a definováním jednoznačného formátu předávaných dat je možné snadno aktualizovat případné nedostatky nástrojů. Kompatibilita s ostat-

ními nástroji je limitována pouze verzí formátu předávaných dat, není třeba zohledňovat ostatní existující nástroje. Například aktualizace parseru, která povede na zkvalitnění získávaných dat, nijak neovlivní případné analyzátory, kterou mohou být použity k analýze získaných dat tímto konkrétním parserem.

Implementace architektury je vytvořena v rámci této práce v jazyce C#. Tento jazyk a jeho dostupné pomocné knihovny plně postačují potřebám projektu. Jazyk, včetně dostupných knihoven, je aktivně udržován a proto je vhodnou volbou z hlediska udržitelnosti vyvíjeného softwaru.



Obrázek 7.3: Bloky architektury

7.1 Jádro

Společné jádro bylo navrženo tak, aby co nejvíce vypomohlo při vytváření nástrojů pro řešení problémů s analýzou softwarových závislostí. Obsahuje společné datové struktury, mezi které se řadí hlavně implementace obecného úložného datového formátu grafu závislostí, dle navrženého datového modelu. V rámci konstant v jádře je uložena verze obecného úložné datového formátu, dle kterého je implementace vytvořena. Samotné jádro je možné distribuovat jako balíček, do příslušného repozitáře.

V jádře jsou pomocné třídy, které operují nad daty, poskytují práci s vstupem a výstupem. Jsou zde i rozšiřující metody pro práci nad grafovými daty uloženými v obecném formátu, konkrétně pro jednu závislost. Nad závislostí je tak možné zavolat metodu, která provede procházení všech přímých závislostí, základní potřeba nad závislostí. Užitečnější je implementace procházení do šířky, která je taktéž dostupná pro jednu závislost jako rozšiřující metoda. Předchozí dvě zmíněné rozšiřující metody benefitují díky možnosti předat jako argument akci, které se má vykonat nad určitým prvkem. Při volání akce je předána informace o závislosti, rodičovské závislosti

a případně i o konkrétní hraně. Poslední z rozšiřujících metod nad závislostí je pro získání všech unikátních uzlů a hran, které se v grafu závislostí vyskytují.

Kritickou součástí jsou i rozhraní pro tvorbu importerů a exporterů pro data. Tato rozhraní definují, jaké vlastnosti mají mít příslušné implementace. Následně jádro obsahuje i implementace těchto rozhraní, pro základní vstupní a výstupní formáty, které jsou obecně použitelné. Jsou zde i rozhraní pro analyzátory a parsery.

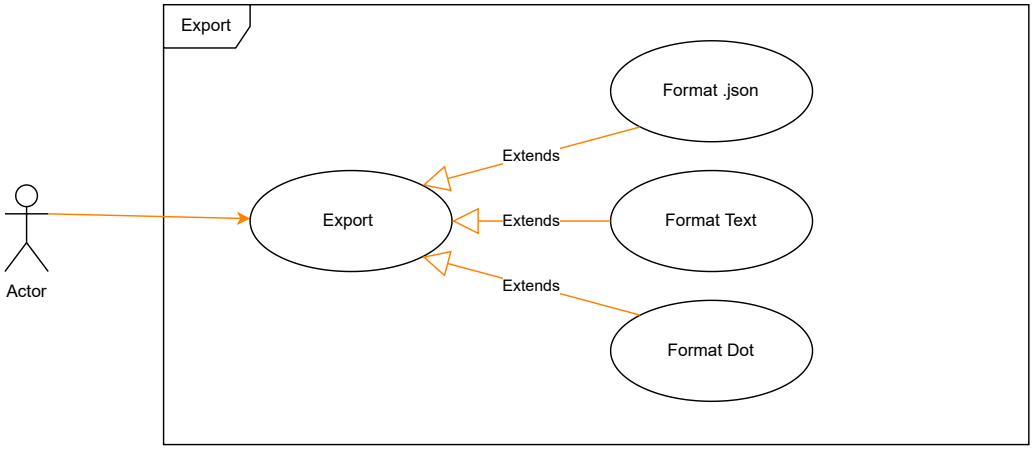
Verzování jádra je navrženo systémem sémantického verzování, toto verzování je spojeno s verzí podporovaného univerzálního formátu. V případě aktualizace verze univerzálního datového formátu a ztráty kompatibility s předchozí verzí je potřeba aktualizovat i verzi jádra tak, aby vyjadřovala, že došlo ke ztrátě kompatibility. V případě dodávání další funkcionality, například další exportery, stačí navyšovat pouze minoritní verzi, nedojde ke ztrátě kompatibility.

V případě většího zásahu do modelu je třeba zohlednit, jestli došlo k porušení kompatibility s předchozími verzemi univerzálního datového formátu. Podle závažnosti zásahu do modelu je potřeba inkrementovat jeho verzi, která je uložena v jádře v rámci konstant.

7.2 Dostupné výstupní formáty

Při analýze softwarových artefaktů je potřeba poskytnout rozumný výstup, který bude smysluplný, ať už pro potřeby dalšího zpracování, nebo pro prezentaci dosažených výsledků. Je tedy potřeba mít k dispozici jeden dokonale vyhovující, nebo více různých výstupních formátů, které bude možné využívat k ukládání výstupů příslušných nástrojů.Problém jednoho dokonalého výstupního formátu je splnění podmínky kompletnosti a přehlednosti prezentovaných výstupů, data mohou být velmi obsáhlá a tím hůře čitelná. Oproti tomu, přehledná data mají za efekt ztrátovost informací, za přínosu lepší přehlednosti. V rámci práce byly vytvořeny základní exportery, po jejichž vzoru je možné vytvářet další implementace, případně existující implementace volně upravovat a vylepšovat. Na obrázku 7.4 jsou vidět různé dostupné formáty výstupních dat.

Pro tvorbu exporterů je přístup velmi obecný, dle rozhraní musí umět přijmout libovolný datový typ a volitelně cestu k uložení do souboru. Je tedy v režii příslušné implementace exporteru, aby rozhodla, zda zvládne provést export s předaným datovým typem. Tato volnost by na první pohled mohla způsobovat problémy, ale opak je pravdou. Není třeba definovat v rozhraní



Obrázek 7.4: Dostupné výstupní formáty

exportní metody pro každý datový typ speciálně, stačí jedna generická, zodpovědnost za zpracování je až v implementaci exporteru. Tím je možné, při zachování stejného rozhraní, volně rozšiřovat schopnosti exporteru. Nutí to programátora zamyslet se nad předávaným datovým typem a realizací co nejobecnější impelmentace, která je lépe do budoucna rozšiřitelná.

7.2.1 JSON formát

Představený formát JSON se pro ukládání programových dat hodí velmi dobře, je možné do něj uložit beze ztráty informace celý graf závislostí, uchovaný v navrženém univerzálním formátu. Pro bezproblémové ukládání dat tak, aby data bylo možné co nejsnadněji znovu načíst jsou data reprezentující graf závislostí, v univerzálním datovém formátu, zabalena v datovém obalu. Formát JSON ukládá data do obecných objektů, proto není problém do něj uložit libovolný, serializovatelný objekt. V případě, že se v datech vyskytují opakující se objekty, jsou uloženy formou reference na první výskyt tohoto objektu. Referencování má za přínos bezproblémové uložení dat obsahujících v datech cyklické vazby, zároveň dochází k výrazné úspoře potřebné paměti k uložení dat. Toto jsou hlavní přínosy JSONu, oproti formátu XML, kde je třeba velmi pracně hlídat cyklické závislost a řešit vlastní referencování, je to velmi náročné. Serializace do formátu JSON je možné provádět za asistence knihoven k tomu určených, například pro jazyk C# existuje knihovna „Newtonsoft.Json“

7.2.2 Textový formát

Reprezentace dat pomocí textového formátu je v jádře k dispozici pro objekty z univerzálního datového formátu. Dochází k vypsání stromové struktury za pomoci ASCII znaků. Jedná se vlastně o výpis od kořene, až do listů, za pomoci ASCII znaků a mezer je naznačena struktura. V tomto zobrazení je problém předat větší množství informací o uzlu, protože jeho informace jsou na jedné řádce. Je tedy potřeba zobrazit pouze nezbytné informace. Dalším problém jsou informace z hrany, ta je zde hůře reprezentovatelná spolu s daty, proto jsou hranové informace umístěny za textovou reprezentaci uzlu. Na ilustraci 7.5 je ukázka reprezentace grafových dat, konkrétně jde o balíčky a jejich závislosti.

```
\-Id: pipgrip, Name: pipgrip, Version: 0.1.0, Direct dependencies: 6
  |-Id: anytree, Name: anytree, Version: 2.8.0, Direct dependencies: 1
  | \-Id: six, Name: six, Version: 1.16.0, Direct dependencies: 0 (>=1.9.0)
  |-Id: click, Name: click, Version: 8.1.3, Direct dependencies: 1
  | \-Id: colorama, Name: colorama, Version: 0.4.4, Direct dependencies: 0
  |-Id: packaging, Name: packaging, Version: 21.3, Direct dependencies: 1 (>=17)
  | \-Id: pyparsing, Name: pyparsing, Version: 3.0.9, Direct dependencies: 0 (!=3.0.5)
  |-Id: pkginfo, Name: pkginfo, Version: 1.8.2, Direct dependencies: 0 (>=1.4.2)
  |-Id: setuptools, Name: setuptools, Version: 62.3.1, Direct dependencies: 0 (>=38.3)
  \-Id: wheel, Name: wheel, Version: 0.37.1, Direct dependencies: 0
\-Id: Flask, Name: Flask, Version: 2.1.2, Direct dependencies: 5
  |-Id: click, Name: click, Version: 8.1.3, Direct dependencies: 1 (>=8.0)
  | \-Id: colorama, Name: colorama, Version: 0.4.4, Direct dependencies: 0
  |-Id: itsdangerous, Name: itsdangerous, Version: 2.1.2, Direct dependencies: 0 (>=2.0)
  |-Id: Jinja2, Name: Jinja2, Version: 3.1.2, Direct dependencies: 1 (>=3.0)
  | \-Id: MarkupSafe, Name: MarkupSafe, Version: 2.1.1, Direct dependencies: 0 (>=2.0)
  \-Id: Werkzeug, Name: Werkzeug, Version: 2.1.2, Direct dependencies: 0 (>=2.0)
```

Obrázek 7.5: Reprezentace grafu pomocí textového formátu

7.2.3 Dot formát

Jedná se o grafový formát, pomocí textové reprezentace uzlů a hran lze definovat graf. Vytvoření grafu není příliš problematické, jde pouze o to vydefinovat unikátní uzly v grafu a ty do souboru s definicí grafu přidat. Uzel je definován identifikátorem a případnými atributy. Hrany se definují pomocí referencování identifikátorů uzlu, mezi identifikátory je potřeba vložit styl vazby. Na obrázku 7.6 je vidět ukázka výstupu pro menší graf, nejprve jsou definovány uzly, následně jsou definovány vazby mezi uzly, tedy hrany.

Formát DOT lze velmi dobře vizualizovat, z vizualizace lze rychleji a přehledněji analyzovat případné problémy. Zároveň dokáže zvýraznit anotovaná data, čímž usnadní jejich nalezení v nepřehledném grafu závislostí. Aby byl graf co nejúspornější, uzly obsahují pouze nutné informace, těmi jsou identifikátor balíčku a jeho verze. Veškeré dodatečné informace o balíčku, včetně anotací, jsou dostupné z tooltipu. Vizualizace grafu popsaného na obrázku 7.6 je vidět na obrázku 7.7. Pro vizualizaci lze využít DOT kompatibilní

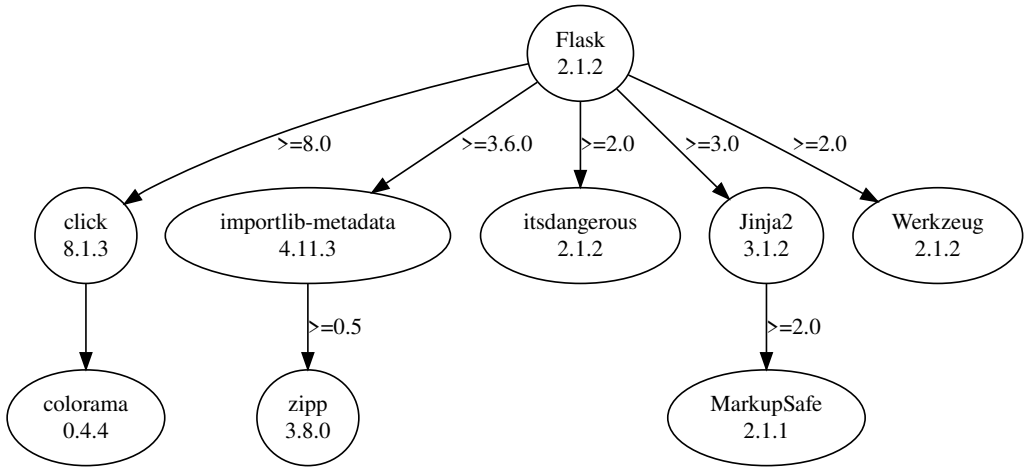
vizualizátor, například nástroj Graphviz. Jeho online verze je dostupná z následujícího odkazu.

https://dreampuf.github.io/GraphvizOnline/

```
digraph
{
  "click8.1.3" [ label = "click\n8.1.3", shape = oval]
  "colorama0.4.4" [ label = "colorama\n0.4.4", shape = oval]
  "Flask2.1.2" [ label = "Flask\n2.1.2", shape = oval]
  "importlib-metadata4.11.3" [ label = "importlib-metadata\n4.11.3", shape = oval]
  "itsdangerous2.1.2" [ label = "itsdangerous\n2.1.2", shape = oval]
  "Jinja23.1.2" [ label = "Jinja2\n3.1.2", shape = oval]
  "Werkzeug2.1.2" [ label = "Werkzeug\n2.1.2", shape = oval]
  "zipp3.8.0" [ label = "zipp\n3.8.0", shape = oval]
  "MarkupSafe2.1.1" [ label = "MarkupSafe\n2.1.1", shape = oval]

  "click8.1.3" -> "colorama0.4.4" [ label = ""]
  "Flask2.1.2" -> "click8.1.3" [ label = ">=8.0"]
  "Flask2.1.2" -> "importlib-metadata4.11.3" [ label = ">=3.6.0"]
  "Flask2.1.2" -> "itsdangerous2.1.2" [ label = ">=2.0"]
  "Flask2.1.2" -> "Jinja23.1.2" [ label = ">=3.0" ]
  "Flask2.1.2" -> "Werkzeug2.1.2" [ label = ">=2.0"]
  "importlib-metadata4.11.3" -> "zipp3.8.0" [ label = ">=0.5"]
  "Jinja23.1.2" -> "MarkupSafe2.1.1" [ label = ">=2.0"]
}
```

Obrázek 7.6: Graf popsaný DOT formátem



Obrázek 7.7: Vizualizace grafu z DOT formátu

7.3 Import dat

Pro import dat je potřeba mít data uložena v univerzálním úložném datovém formátu, který je serializován do JSONu. Aktuální návrh podporuje pouze import z tohoto formátu, žádné jiné importery nebyly navrhnuty. Tento import postačuje pro potřeby všech navržených programů, je možné předávat data. Bylo by možné provést import i z jiného formátu, ovšem za cenu větší

pracnosti. Bylo by třeba data pochopit a převést je do univerzálního úložného formátu. K tomuto účelu by mohly sloužit spíše specializované nástroje, konvertory, než další importery.

7.3.1 JSON formát

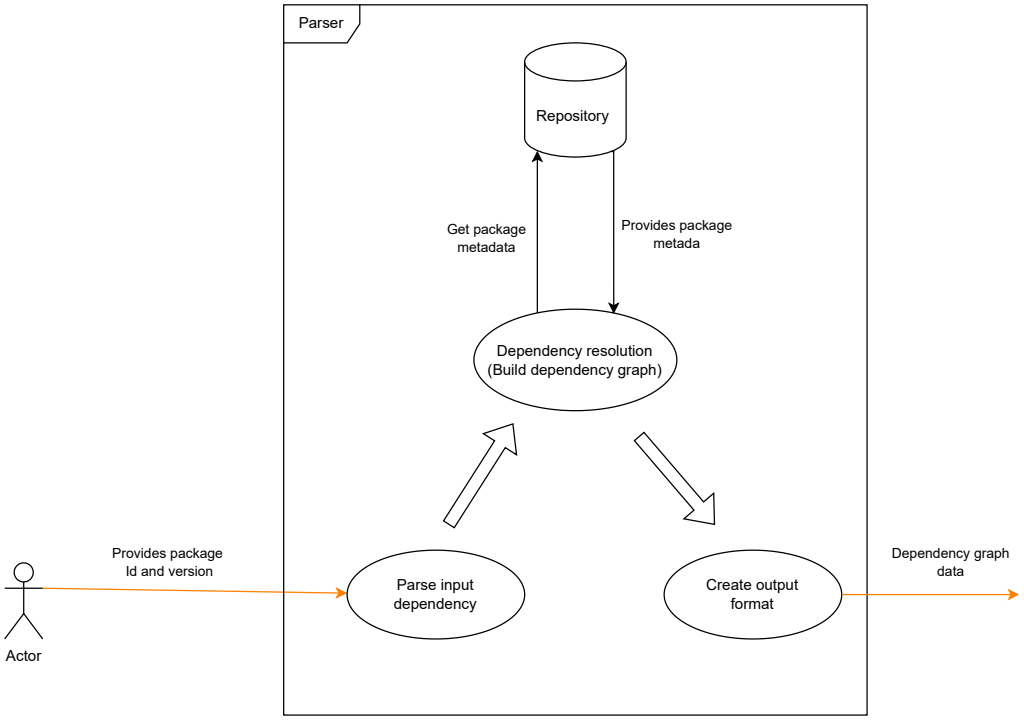
Načtení vstupního JSON souboru, neboli deserializace, je opačný postup k jeho vytvoření v sekci 7.2.1. Je potřeba načíst obal a zkontrolovat jeho verzi, pokud verze načítaného souboru není implementací podporována, je potřeba import ukončit s chybou. Pokud je verze v pořádku, je potřeba zjistit datový formát, uložený uvnitř obalu. Se znalostí tohoto formátu je možné načíst data do správných modelů a tím importovat celý soubor.

Malou nevýhodou je dvojitá deserializace, nejdříve se načte obal a až jako další krok dojde k načtení datové položky v obalu. Tento postup je bohužel nezbytně nutný, je potřeba rozhodnout o schopnosti implementace data přijmout a v případě problému oznámit co nejpřesnější chybu. Pokud by importer pouze naslepo zkusil načíst soubor, novějšího formátu, nepovede se mu to a ohlásí že soubor je chybný. Díky znalosti verze přítomné v obalu lze prohlásit, že se jedná o špatnou verzi souboru. V případě, že se uživatel rozhodne, je možnost obejít kontrolu verze a data zkusit natvrdo načíst. Jsou případy, kdy se to opravdu může podařit, za cenu ztráty určitých informací, takže data budou částečně použitelná, ale nekompletní. Tento přístup by neměl být používán, ale může se objevit situace kdy bude nezbytně nutný.

7.4 Parsery

Jedná se o samostatné nástroje, které mají za úkol obstarat graf závislostí, z předaných argumentů dostanou kořenové body. Jedná se o balíčky, pro které je potřeba vytvořit graf závislostí. Dotazováním na příslušné repozitáře, které balíčky uchovávají je možné získat metadata, ve který jsou informace o dalších závislostech, které je možné následně zpracovávat. Tímto postupem vytvoří graf závislostí. Celé workflow parserů je znázorněno na obrázku 7.8. V rámci jádra existuje rozhraní pro tvorbu parserů, ale implementace těchto rozhraní je spíše dobrovolná, než povinná. Tato rozhraní ovšem definují, jaké vlastnosti by měly příslušné nástroje splňovat, tedy i umět vyřešit. Tím by se měla implementace příslušných nástrojů velmi zjednodušit. Parsery by měli mít závislost na jádře, kde je jim poskytnuta implementace datového modelu a případné pomocné funkce pro datové operace. Tato vazba ovšem není nezbytně nutná, v případě že parser obsahuje vlastní, korektní, implementaci pro univerzální datový formát. Výstupem parseru by měl být

graf závislostí, pro další použití v rámci ostatních nástrojů je potřeba, aby byl výstup v univerzálním datovém formátu a serializován do JSONu.



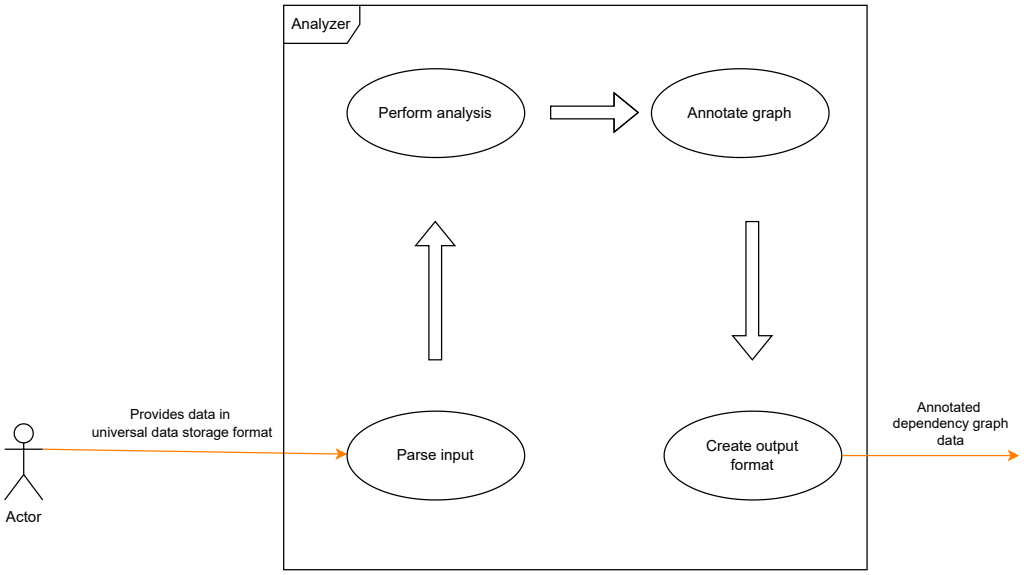
Obrázek 7.8: Workflow parseru

Vytvoření parseru je možné udělat za předpokladu dodržení formátu výstupu prakticky libovolně, v jakémkoliv programovacím jazyce. Důraz je kladen pouze na výstupní data, která budou použita jako vstup ostatních nástrojů. V případě že bude implementace nového parseru v jazyce C#, je možné jako inspiraci použít existující implementace parserů představených v kapitole 8.

7.5 Analyzátory

Jedná se o samostatné nástroje, které mají za úkol analyzovat graf závislostí a poskytnout výsledky analýzy. V rámci jádra existuje rozhraní pro tvorbu analyzátorů, ale vzhledem k tomu, že pracují samostatně, je implementace těchto rozhraní spíše dobrovolná, než povinná. Tato rozhraní ovšem definují, jaké vlastnosti by měly příslušné nástroje splňovat, tedy i umět vyřešit. Tím by se měla implementace příslušných nástrojů velmi zjednodušit. Analyzátory by měly mít, stejně jako parsery, závislost na jádře, kde je jim poskytnuta implementace datového modelu a případné pomocné funkce pro datové operace. Tato vazba ovšem taktéž není nezbytně nutná, v případě že

analyzátor obsahuje vlastní, korektní, implementaci pro univerzální datový formát. Oproti parserům je zde navíc požadavek na schopnost importovat univerzální datový formát, serializovaný do JSONu, tato implementace je součástí jádra, ale nic nebrání pokusit se napsat vlastní, jeli to nezbytně nutné. Workflow práce analyzátorů je zobrazeno na ilustraci 7.9.



Obrázek 7.9: Workflow analyzátorů

Výstupem analyzátorů by měl být anotovaný graf závislostí, kde anotace příslušných prvků informuje a výsledcích analýzy. Pro další použití v rámci ostatních nástrojů je potřeba, aby byl výstup v univerzálním datovém formátu a serializován do JSONu. Pokud analyzátor svou podstatou nedokáže své výstupní informace zakomponovat do univerzálního datového formátu, jedná se například o pouhé metriky, je možné mu tuto vlastnost odpustit. Nesplněním podmínky výstupního formátu se ovšem výstup analyzátoru již nedá znovu analyzovat, tato analýza je konečná a jejím výstupem došlo ke ztrátě informací, ztraceny jsou i anotace z předchozích analyzátorů, které byly v datech uloženy.

Benefitem podmínky pro výstup dat z analyzátoru v anotovaném univerzálním datovém formátu je možnost analyzátory řetězit, nad jedním vstupním souborem provést několik analýz a až po té provést vyhodnocení získaných výsledků. Typicky by se mohlo jednat o potřebu najít v grafu závislostí všechny nežádoucí licence a zároveň zkontrolovat, jestli se v grafu nenachází cykly. Tohoto efektu se dá dosáhnout z příkazové řádky pomocí tvz. rour, kdy je výstup programu připojen na vstup následujícího programu.

7.6 Porovnání s jinými přístupy

Navržená architektura benefituje rozdělením zodpovědností za jednotlivé podproblémy do separátních nástrojů. Jediná podmínka, kterou nástroje mají, je dodržení univerzálního datového formátu. Pro vstup, výstup, nebo obojí, podle účelu nástroje. V případě spojení celé funkcionality do monolitické aplikace by odpadla potřeba mít více nástrojů, vše by bylo obsaženo v jednom nástroji. S tím se ale pojí další problémy, hlavně z pohledu aktualizace a horší rozšiřitelnosti. Doplnit monolitickou aplikaci o další parser by znamenalo upravit celou aplikaci a doplnit příslušnou funkcionalitu. Při úpravě aplikace se dá snadno dostat do stavu, kdy existující funkcionalita koliduje s nově přidávanou, takže je třeba zohlednit existující implementaci. Aplikace je rovněž mnohem hůře udržitelná, změna společné funkcionality znamená upravit vše, co je na tuto funkcionalit vázáno. Oproti tomu v případě separátních aplikací není vývojář limitován implementací ostatních nástrojů, aplikace je tedy lépe udržitelná a dají se v ní rychleji provádět změny.

Mezi separátními nástroji a monolitem ještě existuje jeden možný přístup. Udělat základní aplikaci pro analýzu závislostí softwarových artefaktů a funkcionality dodávat formou modulů. Tento přístup má stále určitá úskalí, hlavně ve formě vazby na společnou část aplikace, která se stará o řízení. Je tedy třeba, aby moduly byly řiditelné z hlavní aplikace, tím vzniká problém, který může být náročné vyřešit. Moduly by musely implementovat rozhraní, pomocí kterého by byly ovládány z hlavní aplikace, pokud ovšem modul nedokáže svou podstatou rozhraní vyhovět, není možné jej vhodně implementovat. Je třeba upravit rozhraní, tím i hlavní aplikaci a dojde ke ztrátě všech již dodaných modulů, ztratí kompatibilitu. Moduly je třeba aktualizovat, jinak jsou v nové hlavní aplikaci nepoužitelné. Separátní nástroje touto vadou netrpí, jejich jediným problémem by byla změna verze univerzálního datového formátu. Separátní nástroje mohou být různých verzí, dokonce mohou i používat různé verze jádra, které se mohou lišit dostupnou funkcionalitou, ale stále budou plnohodnotně použitelné v rámci kooperace s ostatními nástroji, protože univerzální datový formát zůstal nezměněn.

8 Získání grafu závislostí

Úkol získat graf závislostí je v režii nástrojů, zvaných parsery, tyto specializované nástroje se postarají o stavbu grafu závislostí. U uzlů grafu, balíčků, obstarají z dostupných zdrojů co nejvíce metadat je to možné, tato data následně doplní do vytvořeného grafu závislostí. Dostupnými zdroji jsou příslušné repozitáře, ve kterých se balíčky nachází.

Problém je s nedostatkem množstvím informací, které parsery dokáží obstarat. Důvodem nedostatku informací může být nedostatečná schopnost parserů si data obstarat, případně poskytnutá data pochopit, nebo v samotné komunikaci s repozitářem. Repozitáře v principu své funkce musí poskytovat určitá metadata, ovšem rozsah poskytovaných dat se různí, případně je ještě podmíněn použitým komunikačním kanálem. Veřejné rozhraní repozitářů nemusí poskytovat stejná data, jaká používá balíčkovací nástroj pro příslušný repozitář, může komunikovat jiným kanálem. V nejhorším případě, kdy jsou metadata nepřítomna to může být chyba, nebo lenost, programátora a tím se stává balíček hůře analyzovatelným.

Způsob napsání vlastního vytváření grafu závislostí pro příslušný balíčkovací systém je sice možný, ale lepší je pokusit se najít již existující tvz. resolver. Jedná se o specializovaný nástroj, nebo knihovnu, která se postará o vytvoření grafu závislostí. Takto získaný graf závislostí ale neobsahuje metadata, které jsou vyžadována pro další analýzu. Je tedy třeba stále obstarat tato metadata a doplnit je do získaného grafu. Při použití existujících nástrojů je menší riziko chyby při stavbě grafu, protože zodpovědnost za stavbu grafu je v rámci použité knihovny, nebo nástroje. Po získání grafu závislostí je stejně třeba získaná data převést do vhodného formátu a dodat příslušná metadata pro balíčky.

Každý navržený parser by měl být specializovaný na jeden druh repozitáře, případně i na jeho verzi. Nemělo by se v rámci parseru řešit více, než získání grafu závislostí, příslušných metadat balíčků a následného exportu získaných dat. Pokud by se v rámci parseru řešilo více různých repozitářů, nesouhlasilo by to s návrhem architektury, nástroje by byly zbytečně velké a hůře udržovatelné.

Graf závislostí má smysl tvořit i pro výčet balíčků, lze tím sledovat jejich případné dopady na projekt, do kterého se plánují přidávat. Případně je vhodné použít je pro získání grafu závislostí pro specifický projekt, vytvoří se graf závislostí, který by měl odpovídat požadavkům projektu.

V této práci byly navrženy dva parsery, které získávají graf závislostí,

včetně příslušných metadat a dovolují jej uložit do univerzálního úložné formátu popsaného v rámci této práce. Jedná se o parser pro repozitář typu Nuget a pro repozitář typu PyPI.

8.1 Nuget

Nástroj pro získání grafu závislostí pro repozitář typu Nuget byl vytvořen v rámci této práce. Zdrojový kód vytvořeného nástroje je přílohou této práce a je poskytnut s otevřenou licencí MIT, je tedy volně použitelný.

Nástroj je vhodný pro získání grafu závislostí pro jednotlivé balíčky, případně výčet balíčků, z příslušného repozitáře. Zároveň slouží i k získání balíčků z projektů .NET Core a .NET v jazyce C# z prostředí Visual Studio. Nástroj bohužel neumí zpracovat zastaralé .NET Framework typy projektů, kvůli odlišnostem při práci se závislostmi. Nad získanými balíčky provede stejné operace vedoucí k získání grafu závislostí, jako pro výčet balíčků.

8.1.1 Popis

Základem celého nástroje je obal pro práci s repozitářem typu Nuget, zde jsou k dispozici dvě kritické metody. První dokáže získat metadata o specifickém balíčku, dle předaného id a verze. Metoda následně naplní příslušný objekt reprezentující balíček z univerzálního datového formátu a objekt vrátí. Druhá metoda dokáže poskytnout informace o závislostech specifického balíčku a specifické platformy.

Podle vstupních argumentů si program vytvoří seznam balíčků, které potřebuje zpracovat. Pokud je vstupem projekt, jedná se o jeho přímé závislosti. V případě řešení, zjistí všechny projektu v řešení a získá jejich přímé závislosti.V rámci stavby grafu závislostí program zavádí tvz. resolve cache. Jedná se o slovník, ve kterém jsou uloženy již navštívené unikátní balíčky, klíčem je dvojice id a verze balíčku. Samotná stavba grafu závislosti se započne z přímých závislostí, pomocí algoritmu prohledávání do šířky se postupně prvky zpracovávají. Než se prvek začne zpracovávat, zkontroluje se jestli již není přítomen v resolve cache. Pokud je v resolve cache přítomen, použije se pouze reference na prvek a nedojde k jeho podrobnějšímu zpracování. Pro nové prvky, které v resolve cache nejsou, se provede celé zpracování. Zpracováním prvku se rozumí zjištění jeho závislostí, voláním příslušné metody z obalu k repozitáři typu Nuget. V této fázi také dojde k doplnění metadat, po této akci dojde k uložení prvku do resolve cache. Závislosti zpracovávaného prvku jsou vloženy do fronty určené k dalšímu zpracování algoritmem prohledávání do šířky. Po dokončení stavby grafu závislostí je možné přistoupit

k předání výstupu.

8.1.2 Ovládání

Možnosti ovládání nástroje jsou pomocí argumentů příkazové řádky. Nástroj poskytuje výpis přijímaných argumentů pomocí helpu. Přípustné argumenty jsou v následujícím seznamu, tak jak by je vypsal sám program.

- -f, —projectFiles
  - Input project files to be processed. (*.csproj or *.sln)
- —analyzeDependencies
  - List of dependencies to be analyzed (<id>;<version>) This option disables input files option values, framework version has to be specified!
- —frameworkVersion
  - For custom dependency input, the framework version must be specified! (Example: "net5.0", "netstandard2.1", etc)
- -d, —depth
  - (Default: -1) Depth of the search in the dependency tree (Possible value: 1 - n), -1 is maximum.
- -r, —repositoryUrl
  - (Default: https://api.nuget.org/v3/index.json) Nuget v3 repository URL
- -v, —verbose
  - (Default: false) Prints all messages to standard output.
- -t, —outputType
  - (Default: JSON) Type of the output data. Supported types: "JSON", "MD", "DOT", "Text"
- -o, —outputFile
  - Path to the output file
- —help

  - Display this help screen.

- —version
  - Display version information.

Nástroj je možné spustit s definováním cesty k souboru, nebo výčtem balíčků. Oba tyto argumenty přijímají seznam hodnot, nejsou limitovány jednou hodnotou.

Ukázka spuštění programu.

```
NugetParser.exe -f Janus.csproj Hydra.csproj
```

8.1.3 Výstupy

Výstupem nástroje jsou formáty dostupné přímo z jádra, pomocí argumentu „outputType“ se dá nastavit příslušný formát. Nástroj nemá žádná omezení, která by vylučovala jakýkoliv z výchozích formátů. V datech je uložena reprezentace grafu závislostí pro zvolená vstupní data.

8.1.4 Omezení nástroje

Graf závislostí je postaven jen za částečné asistence knihoven z projektu na vývoj nástroje Nuget. Použity byly hlavně knihovny pro komunikaci, ale existuje i veřejná implementace pro samotný resolving. Jejím výstupem je celý graf závislostí. Bohužel se nepodařilo najít žádnou dokumentaci, která by popisovala jak příslušný kód použít. Jedná se o zdrojový kód samotného nástroje Nuget, který tímto kódem opravdu řeší získávání závislostí pro projekty. Tím, že je resolving napsán „natvrdo“, každá změna v rozhodování o výběru balíčku v implementaci nástroje Nuget znamená i nutnou změnu v kódu navrženého parseru.

Data v repozitáři typu nuget nemusí být kompletní, proto jedním z dalších problémů je v určitých případech neschopnost zajistit dostatečná data pro případné analyzátory. Navržený nástroj se dotazuje pouze jednoho repozitáře, nemá další zdroje odkud informace získat.

8.2 PyPI

Nástroj pro získání grafu závislostí pro repozitář typu PyPI byl vytvořen v rámci této práce. Zdrojový kód vytvořeného nástroje je přílohou této práce a je poskytnut s otevřenou licencí MIT, je tedy volně použitelný.

Nástroj je vhodný pro získání grafu závislostí pro jednotlivé balíčky, případně výčet balíčků, z příslušného repozitáře.

8.2.1 Popis

Základem celého nástroje je obal pro práci s repozitářem typu PyPI, zde jsou k dispozici dvě kritické metody. První dokáže získat metadata o specifickém balíčku, dle předaného id a verze. Druhá metoda dokáže poskytnout informace o závislostech specifického balíčku. Důležitou součástí je také nástroj „pipgrip“, který je interně v parseru použit. Jedná se o nástroj napsaný v jazyce Python, který dokáže sestavit graf závislostí a poskytnout ho v JSON formátu, tento graf ale neobsahuje mimo id a verzi o balíčku žádné další informace. Pro správné fungování navrženého parseru je nutné mít nainstalován Python a s ním i příslušný nástroj „pipgrip“. Dokumentace k nástroji je k dispozici na následující adrese.

https://github.com/ddelange/pipgrip

Pro práci s repozitářem typu PyPI bylo také třeba implementovat dle dokumentace datové modely pro formát odpovědí z koncového bodu. Tyto modely nejsou zcela kompletní, ale obsahují dostatečnou funkcionalitu pro potřeby parseru. Důvodem nekompletnosti je zvýšená pracnost při implementaci, protože podle dokumentace jsou některé informace zavádějící. Jedná se hlavně o informace o datových typech jednotlivých atributů.

Podle vstupních argumentů si program vytvoří seznam balíčků, které potřebuje zpracovat.V rámci stavby grafu závislostí program zavádí tvz. resolve cache. Jedná se o slovník, ve kterém jsou uloženy již navštívené unikátní balíčky, klíčem je dvojice id a verze balíčku. Samotná stavba grafu závislosti se provede z přímých závislostí, jsou předány nástroji „pipgrip“. Nástroj „pipgrip“ vytvoří návrh grafu závislostí, který předá zpátky parseru. Pomocí algoritmu prohledávání do šířky se postupně zpracovávají prvky získaného grafu. Než se prvek začne zpracovávat, zkontroluje se jestli již není přítomen v resolve cache. Pokud je v resolve cache přítomen, použije se pouze reference na prvek a nedojde k jeho podrobnějšímu zpracování. Pro nové prvky, které v resolve cache nejsou, se provede celé zpracování. Zpracováním prvku se rozumí doplnění metadat, po této akci dojde k uložení prvku do resolve cache. Závislosti zpracovávaného prvku jsou vloženy do fronty určené k dalšímu zpracování algoritmem prohledávání do šířky. Po dokončení přestavby grafu závislostí je možné přistoupit k předání výstupu.

8.2.2 Ovládání

Možnosti ovládání nástroje jsou pomocí argumentů příkazové řádky. Nástroj poskytuje výpis přijímaných argumentů pomocí helpu. Přípustné argumenty jsou v následujícím seznamu, tak jak by je vypsal sám program.

- —analyzeDependencies
  - List of dependencies to be analyzed (<id>;<version>)
- -d, —depth
  - (Default: -1) Depth of the search in the dependency tree (Possible value: 1 - n), -1 is maximum.
- -r, —repositoryUrl
  - (Default: https://pypi.org/simple) PyPI repository URL
- —noCacheDir
  - (Default: false) Does not consider local cache dir state during dependency graph creation
- -v, —verbose
  - (Default: false) Prints all messages to standard output.
- -t, —outputType
  - (Default: JSON) Type of the output data. Supported types: "JSON", "MD", "DOT", "Text"
- -o, —outputFile
  - Path to the output file
- —help
  - Display this help screen.
- —version
  - Display version information.

Nástroj je možné spustit s definováním výčtu balíčků. Tento argumenty přijímá seznam hodnot, není limitovány jednou hodnotou.

Ukázka spuštění programu.

```
PyPIParser.exe —analyzeDependencies "pipgrip;0.1"
```

8.2.3 Výstupy

Výstupem nástroje jsou formáty dostupné přímo z jádra, pomocí argumentu „outputType“ se dá nastavit příslušný formát. Nástroj nemá žádná omezení, která by vylučovala jakýkoliv z výchozích formátů. V datech je uložena reprezentace grafu závislostí pro zvolená vstupní data.

8.2.4 Omezení nástroje

Výhodou, ale i nevýhodou je použití nástroje pipgrip. Velmi to usnadní problémy spojené s výběrem verzí do grafu závislostí, ale parser je na nástroj pipgrip silně vázán a bez něj nedokáže správně fungovat. Ztrátou tohoto nástroje zanikne i tento parser. Napsat si ovšem vlastní resolving by bylo mnohem pracnější a pravděpodobně by taková implementace obsahovala chyby, protože resolving v rámci PyPI je složitější, než v rámci Nugetu. Hlavně z pohledu omezování výběru verzí.

9 Analyzátory

Analyzovat získaná grafová data reprezentující graf závislostí je úkolem nástrojů nazvaných analyzátory. Jedná se o samostatné nástroje, které se postarají o všechny operace spojené s analýzou dat. Analyzátory jsou specializovány k určitému druhu analýzy, který je zapotřebí k vyřešení, nebo alespoň lepšímu pochopení případného problému. V rámci této práce byly specializace analyzátorů vybrány z podnětů popsaných v kapitole 5.

Specializace konkrétního analyzátoru by měla být vždy jedna, nemělo by dojít k tomu, že analyzátor bude řešit větší množinu problémů. Pokud by analyzátor prováděl více druhů analýz, narušovalo by to myšlenku architektury, kdy každý analyzátor má být zodpovědný na specifický druh analýzy. V případě navyšování funkcionality analyzátorů mimo jejich zaměření se stávají hůře udržitelnými. Je lepší i pro jednodušší druh analýzy vytvořit specializovaný analyzátor, než funkcionalitu dodávat do již existujících. Jednodušší analyzátor bude tak velice snadno pochopitelný pro další programátory a může sloužit jako výborná ukázka toho, jak tvořit další analyzátory.

Analyzátory pro provedení analýzy potřebují dostupná data v očekávaném formátu, v rámci práce byl navržen univerzální datový úložný formát pro graf závislostí a právě ten je očekáván jako vstup analyzátorů. Vstup je vhodné povolit jako cestu k souboru, případně načítáním ze standardního vstupu. Díky možnosti načítat data ze standardního vstupu lze analyzátory velmi jednoduše řetězit a snadno tak spojovat výsledky analyzátorů. Bez potřeby mít jeden velký analyzátor, který by poskytl obdobný výstup. Mimo požadavku na validní formát vstupních dat nemají analyzátory žádné další požadavky, jsou samostatné.

Analyzátory mohou, ale nemusí, mít závislost na jádře. Tato závislost je ovšem velmi výhodná, protože díky ní získají analyzátory přístup k implementaci univerzálního datového formátu. Mimo datový formát jsou zde i metody pro pomoc se vstupem a výstupem dat. V neposlední řadě obsahuje jádro i pomocné metody k procházení grafu závislostí, které mohou být lehce využity při tvorbě analyzátorů a tím urychlit jejich vývoj.

Práci analyzátoru, dovoluje li to jejich podstata, je možné případně ovlivňovat pomocí argumentů příkazové řádky a tím lépe určit styl a podmínky analýzy. Analyzátor si musí vydefinovat tyto argumenty a poskytnout je uživateli sám, jsou velmi specifické k vykonávané analýze a proto je nelze příliš generalizovat.

Výstupem práce analyzátorů může být více různých informací. Silně se to

odvíjí od podstaty provedené analýzy. Pokud je možné výsledky analýzy uložit do existujících dat formou anotace, je to preferovaná volba, protože data je pak možné snadno dále analyzovat, klidně i opakovaně stejným analyzátorem, ale různými styly analýz. Navíc v tomto datovém formátu nedochází k žádné ztrátě informace při exportu. Data je možné exportovat i v jiných formátech, například textovou reprezentací. Tento formát je ovšem velmi ztrátový, množství zobrazitelných informací je zredukováno na úkor lepší přehlednosti zobrazovaných dat. Oproti tomu velmi užitečný je například dot formát, který lze použít i pro následnou vizualizaci. Anotace dat totiž, mimo přidanou hodnotu samotné anotace, také ovlivní obarvení výstupních vizualizovaných dat. Barva je zobrazena podle anotace s nejvyšší důležitostí, kterou má prvek přiřazenou. Barevné rozložení seřazené od nejvyšší důležitosti anotací je následující:

- Varování - Červená barva
- Upozornění - Žlutá barva
- Informační - Modrá barva

Vytvoření vlastního analyzátorů není příliš náročné, tedy pomineme li náročnost řešení problému, který má analyzátor provést. Je pouze potřeba pochopit jak přijmout a jak vyexportovat data. Po vzoru již implementovaných analyzátorů by to nemělo být obtížné. Následně je potřeba analýzu vymyslet tak, že se zakládá na analýze grafu, který bude poskytnut analyzátoru. Od tohoto bodu se jedná pouze o algoritmizaci spojenou s analýzou grafových dat a je již plně v režii příslušného programátora. Po získání výsledku analýzy je pouze třeba data vhodně uložit, jako anotaci, k objektům. Pokud to není možné, například důsledkem typu specifické analýzy, je rozumné data vypsat alespoň textově. Tím ovšem dojde ke ztrátě anotovaného grafu pro další použití. Lze tedy říci, že analyzátory se dají rozdělit na dva základní typy. Analyzátory, jejichž výstup analýzy nezpůsobí ztrátu informace na výstupu, naopak informací přidá formou anotací a na analyzátory, jejichž podstata nedovolí data předat anotovanou reprezentací grafu, tím se stávají nepoužitelné pro následně řetězení. Problém s řetězením je hlavně v tom, že bude ztracena mimo samotného grafu závislostí i informační hodnota předem aplikovaných analyzátorů.

Další zajímavou společnou vlastností analyzátorů může být styl, kterým analyzují data. Prohledávání grafu závislostí je možné několika způsoby, například BFS, či DFS. Analyzátory mohou mít potřeby procházet data obdobně, potom se objevuje další implementační část, kterou by mohlo být možné znovu používat v dalších analyzátorech. Minimálně jako inspiraci z

již napsaných analyzátorů stejného typu. Nabízí se možnost implementaci těchto stylů procházení poskytnout z jádra, ovšem využívat ji znamená silnou závislost na jádře. Změna implementace v jádře tak může zapříčinit problémy s analýzou. Proto je tato volba na příslušném programátorovi. Navíc i když jsou přístupy k procházení obdobné, mohou se lišit v detailech, které by dramaticky ovlivnily dobu prochází grafu. Analyzátory sami nejlépe vědí, která data je ještě třeba procházet a která ne, oproti tomu například obecná implementace BFS je vždy musí projít všechny.

V rámci této práce práce byly navrženy čtyři analyzátory, které řeší představené problémy spojené s využíváním závislostí. Problémy byly představeny v kapitole 5 této práce. Jedná se o analyzátor pro konflikt verzí, kontrolu licencí, kontrolu přítomnosti cyklické závislosti a pro získání počtu závislostí. Posledním z problémů, rozdíly závislostí mezi verzemi, byl řešen, ale nebyl dokončen včas, proto v příslušné sekci bude představen pouze návrh řešení problému, v rámci zohlednění dostupných možností z navržené architektury, bez konkrétní implementace.

9.1 Licenční analyzátor

Tento analyzátor pokrývá potřeby problému představeného v sekci 5.3. Jeho hlavní logikou je projít poskytnutá grafová data a najít v nich takové závislosti, které vyhovují definované podmínce.

Tento analyzátor byl navržen velmi obecně, aktuálně je k dispozici implementace, které dovolí vyhledávat konkrétní licenci, případně pomocí klíčového slova „null“ vyhledá vše co naopak licenci nemá k dispozici. Analyzátor by šel snadno upravit a vyhledávání uzpůsobit na systém klíč-hodnota.

Analyzátor potřebuje v grafu najít všechny unikátní vrcholy a nad každým z těchto vrcholů provést kontrolu, zda li nesplňují definovanou podmínku. Analyzátory používá k prohledávání algoritmus prohledávání do šířky a navštívené vrcholy ukládá za pomoci slovníku. Tím je časově velmi efektivní, přístup do slovníku je v konstantním čase. Náhled implementace je k dispozici na obrázku 9.1.

Analyzátor lze řetezit a tím dosáhnout přesnějších výsledků. Níže je ukázka zřetězení analyzátorů.

```
C:> LicenseAnalyzer.exe -s "License=MIT" |
    LicenseAnalyzer.exe -s "License=Apache2.0" |
    LicenseAnalyzer.exe -s "License=null"
```

```
private void StartDependencyAnalysis()
{
    while (bfsToBeAnalysed.Any())
    {
        DependencyItem current = bfsToBeAnalysed.Dequeue();

        if (bfsVisitedDependencies.Contains(current))…

        bfsVisitedDependencies.Add(current); //Mark as visited

        //Process
        AnalyzeDependency(current);

        //Add subdependencies to the work queue
        foreach (var subDependencyEdge in current.Dependencies)…
    }
}
```

Obrázek 9.1: Ukázka implementace BFS na grafovými daty

9.2 Analyzátor cyklických závislostí

Tento analyzátor pokrývá potřeby problému představeného v sekci 5.4. Jedná se o nejsložitější ze všech analyzátorů, používá algoritmus prohledávání do šířky. Ukázka implementace algoritmu je na obrázku 9.2. Pro nalezení cyklu je potřeba projít graf do hloubky a označovat si již navštívené vrcholy, v případě že algoritmus narazí na vrchol, který byl již navštíven, nepokračuje přes něj dál a chová se stejně jako když další vrchol neexistuje. V případě, že algoritmus narazí na vrchol, který sice byl navštíven, ale je v aktuální rozpracované cestě, znamená to že byl nalezen cyklus.

Cyklů by mohlo být v grafu větší množství a některé mohou mít jen jiné pořadí prvků. Je proto zaveden slovník, jehož klíčem je seřazená množina prvků v textové reprezentaci, kdy prvek je reprezentován pomocí dvojice id a verze. Množina reprezentuje jeden cyklus v grafu. Hodnotou ve slovníku je seznam prvků cyklu, v původním seřazení, jak byly navštíveny algoritmem DFS. Tím se uchová cesta cyklu ve správném pořadí a je možné ji dále prezentovat uživateli. Na obrázku 9.3 je vidět implementace, která zjistí jaká posloupnost prvků vede na cyklus.

Analyzátor tedy anotuje data, kde se nachází cyklus, tak, že ke každému prvku, který je součástí cyklu dodá anotaci. V anotaci je k dispozici informace o cestě, které vede na cyklus.

```
private void DFS(DependencyItem startItem)
{
    DFSMarkVisited(startItem);
    foreach (var item in startItem.Dependencies)
    {
        DependencyItem subdependency = item.Link;
        if (!dfsVisitedDependencies.Contains(subdependency))
        {
            DFS(subdependency);     //New item not visited
        }
        else
        {
            //OK Visited, is it in path? Yes = cycle!!!
            if (dfsPathSetFast.Contains(subdependency))
            {
                var cycleFound = GetFullGraphCyclePath(subdependency);
                InsertCycleListToDictionary(cycleFound);
            }
        }
    }
    DFSRemoveFromPath(startItem);
}
```

Obrázek 9.2: Ukázka implementace DFS na grafovými daty

```
private List<DependencyItem> GetFullGraphCyclePath(DependencyItem cycleItem)
{
    List<DependencyItem> cycleNodes = new List<DependencyItem>();
    var path = dfsPath.ToList();
    path.Reverse();
    bool occurenceFound = false;
    foreach (var node in path)
    {
        //Cycle is from subdepedency first occurence in the list to the end
        if (!occurenceFound && node == cycleItem)
        {
            occurenceFound = true;
        }
        if (occurenceFound)
        {
            cycleNodes.Add(node);
        }
    }
    return cycleNodes;
}
```

Obrázek 9.3: Ukázka implementace zjišťující pořadí uzlů v cyklu

9.3 Analyzátor počtu

Tento analyzátor využívá slovník, pro uložení unikátních uzlů zjištěných při procházení grafu algoritmem prohledávání do šířky. Jeho cílem je pouze poskytnout požadovanou metriku o počtu unikátních uzlů a o počtu anotovaných uzlů. Pro zjištění anotovaných uzlů je bohužel třeba uzly dodatečně projít a zjistit, které z nich jsou anotované. Na obrázku 9.4 je vidět implementace, pomocí které se získají metriky za pomoci seznamu unikátních uzlů.

```
public void FinalizeAnalysis()
{
    foreach (var package in packagesUniqueSet)
    {
        if (package.AnnotationInfo.Count > 0)
        {
            AnalyzerResult.AnnotatedUniquePackagesCount++;
        }

        AnalyzerResult.UniquePackagesCount++;
    }
}
```

Obrázek 9.4: Získání metrik z množiny unikátních uzlů

10 Ověření funkčnosti a rozbor získaných výsledků

V rámci této kapitoly je představena aplikace vytvořených nástrojů, jejich přínosy, použitelnost a případné problémy. Dále jsou zde prezentovány získané výsledky, spojené s užíváním vytvořených nástrojů.

10.1 Náročnost získávání a analyzování grafu závislostí

V rámci testování nástrojů bylo změřeno, že obstarání grafu závislostí je časově náročnější operace. Jedná se o jednotky až desítky vteřin, podle rozsahu projektu a tím i počtu unikátních balíčků, které je třeba do grafu závislostí dodat. Může to být důsledkem časově náročných operací zprostředkujících komunikaci se vzdáleným repozitářem. Díky implementaci mezipaměti určené pro uchování již dotázaných balíčků byla komunikace s repozitářem omezena na nezbytné minimum. Je potřeba se dotazovat na metadata pouze u nově objevených balíčků.

Pro více analýz nad stejným grafem závislostí je tedy časově mnohem efektivnější jej získat, uložit do souboru a všechny následné analýzy provádět nad získaným souborem. Tím dojde pouze k jednomu časově náročnému získání dat, dále je časové omezení už jen v rámci analyzátorů. Analyzátory svou implementací se snaží být co nejefektivnější, v místech kde je to možné používají rozptylovací tabulky, či obdobné abstraktní datové typy, které mají za následek lepší časovou složitost celé implementace.

10.2 Použitelnost k licenčním kontrolám

Namátková kontrola balíčků z repozitářů, za účelem zjištění zda li v grafu závislostí nechybí informace o licencích přinesla zajímavé výsledky. Na obrázku 10.1 a 10.2 je ukázka aplikované workflow na analýzu získaných grafových dat. Cílem analýzy je, za pomoci zřetězení nástroje na kontrolu licence a nástroje na počítání, zjistit kolik balíčků v grafu závislostí nemá přítomnu licenci. Pokud by se nalezl balíček, který nemá licenci, dojde k jeho anotování analyzátorem. Následující analyzátor jen spočítá anotované závislosti, tím dostáváme sumu závislostí, které vyhovují podmínce nepřítomnosti licence.

Výsledek na tomto vzorku ukazuje, že repozitář Nuget nemá informace k dispozici skoro vůbec, oproti tomu repozitář PyPI je na tom o něco lépe. Data z repozitáře PyPI ovšem také nejsou zcela dokonalá.

Dlaším zajímavým zjištěním jsou počty unikátních balíčků, které se zde vyskytují. Vzorky z repozitáře Nuget má oproti vzorkům balíčků z PyPI enormní množství unikátních balíčků. Pro ujasnění, jedná se o analýzu závislostí jednoho jediného balíčku.

Nedostatek informací, který způsobil zkreslení výsledků je možné řešit vylepšením samotných parserů, případně doplnění specializovaných nástrojů, které se budou snažit licenční informace doplňovat do balíčků i odjinud, než jen z repozitáře artefaktů.

```
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\janusNuget.json |.\Janus.Analyzers.Coun
ter.exe
UniquePackagesCount:           185
AnnotatedUniquePackagesCount:  170
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\nugetAsyncInterfaces5.json |.\Janus.Ana
lyzers.Counter.exe
UniquePackagesCount:           3
AnnotatedUniquePackagesCount:  2
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\nugetCastleCore6.json |.\Janus.Analyzer
s.Counter.exe
UniquePackagesCount:           2
AnnotatedUniquePackagesCount:  0
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\nugetXunit5.json |.\Janus.Analyzers.Cou
nter.exe
UniquePackagesCount:           80
AnnotatedUniquePackagesCount:  80
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\pypiPipgripFlask.json |.\Janus.Analyzer
s.Counter.exe
```

Obrázek 10.1: Chybějící licence pro Nuget graf závislostí

```
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\pypiPipgripFlask.json |.\Janus.Analyzer
s.Counter.exe
UniquePackagesCount:           17
AnnotatedUniquePackagesCount:  4
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\pypiPandas.json |.\Janus.Analyzers.Coun
ter.exe
UniquePackagesCount:           5
AnnotatedUniquePackagesCount:  0
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\pypiNumPy.json |.\Janus.Analyzers.Count
er.exe
UniquePackagesCount:           1
AnnotatedUniquePackagesCount:  0
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW>
>> .\Janus.Analyzers.KeyValue.exe -s "License=null" -f .\jsonData\pypihtdp-pt-br.json |.\Janus.Analyzers.
Counter.exe
UniquePackagesCount:           3
AnnotatedUniquePackagesCount:  1
```

Obrázek 10.2: Chybějící licence pro PyPI graf závislostí

10.3 Cyklické závislosti

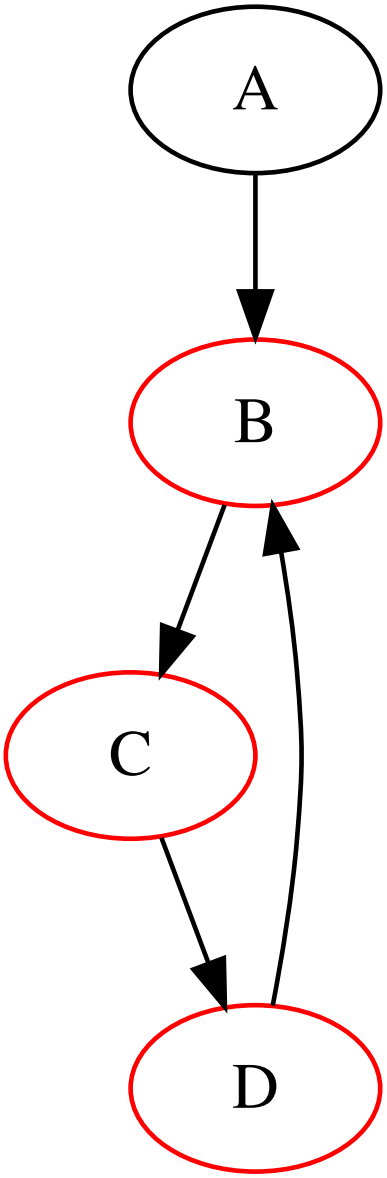
K ověření funkcionality analyzátoru pro cyklické závislosti byla vytvořena testovací data obsahující cyklus, protože se nepodařilo docílit cyklické závislosti v rámci pokusů s kombinací balíčků z příslušných repozitářů. Tato data jsou z datového hlediska stejná, jako reprezentace grafu závislostí, takže toto ověření říká, že pro graf závislostí se bude aplikace chovat stejně.

Na obrázku 10.3 je vidět ověření, na nepřítomnost kruhových závislostí. Opět se jedná o kombinaci specializovaného analyzátoru a analyzátoru na metriky počtu anotovaných dat. Dle výstupů nedošlo k anotaci dat, tím se dá zjistit, že nebyla nalezena žádná kruhová závislost.

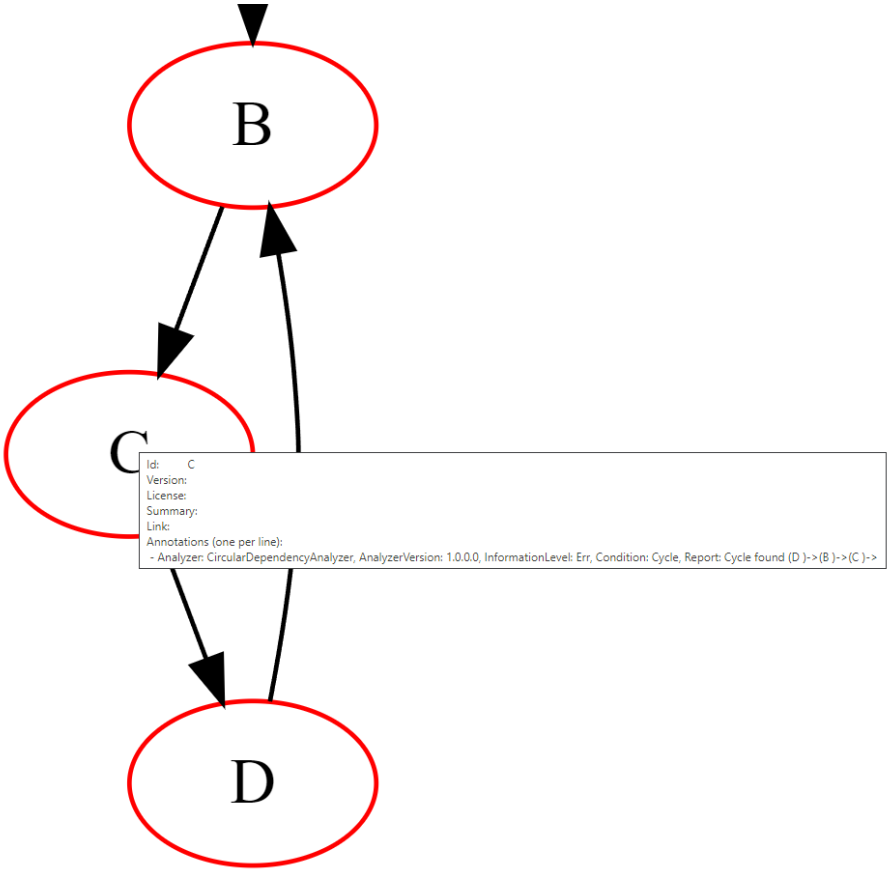
```
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW> .\Janus.Analyzers.Circula
rDependency.exe -f .\jsonData\janusNuget.json | .\Janus.Analyzers.Counter.exe
UniquePackagesCount:           185
AnnotatedUniquePackagesCount:  0
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW> .\Janus.Analyzers.Circula
rDependency.exe -f .\jsonData\nugetAsyncInterfaces5.json | .\Janus.Analyzers.Counter.exe
UniquePackagesCount:           3
AnnotatedUniquePackagesCount:  0
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW> .\Janus.Analyzers.Circula
rDependency.exe -f .\jsonData\nugetCastleCore6.json | .\Janus.Analyzers.Counter.exe
UniquePackagesCount:           2
AnnotatedUniquePackagesCount:  0
PS C:\Users\milan\OneDrive\Vejška-Navazujici\Diplomka\02_Build+test\00_PREVIEW> .\Janus.Analyzers.Circula
rDependency.exe -f .\jsonData\nugetXunit5.json | .\Janus.Analyzers.Counter.exe
UniquePackagesCount:           80
AnnotatedUniquePackagesCount:  0
```

Obrázek 10.3: Otestování na přítomnost kruhových závislostí

Obrázek 10.4 reprezentuje modelový příklad kruhové závislosti. Analyzátor kruhovou závislost odhalil a data anotoval, anotace je na všech uzlech, které jsou součástí cyklu. Vizualizace grafu pomocí dot formátu poskytuje výbornou reprezentaci výsledků, červeně označené uzly jsou anotované s příznakem problému. Problémem je, že je uzel součástí cyklu. V případě potřeby detailnějších informací, ať už o balíčku, či o anotaci je možné využít nápovědu. Najetím myši se zobrazí nápověda s dodatečnými informacemi. Zmíněná nápověda je vidět na obrázku 10.5.



Obrázek 10.4: Zvýrazněná kruhová závislost

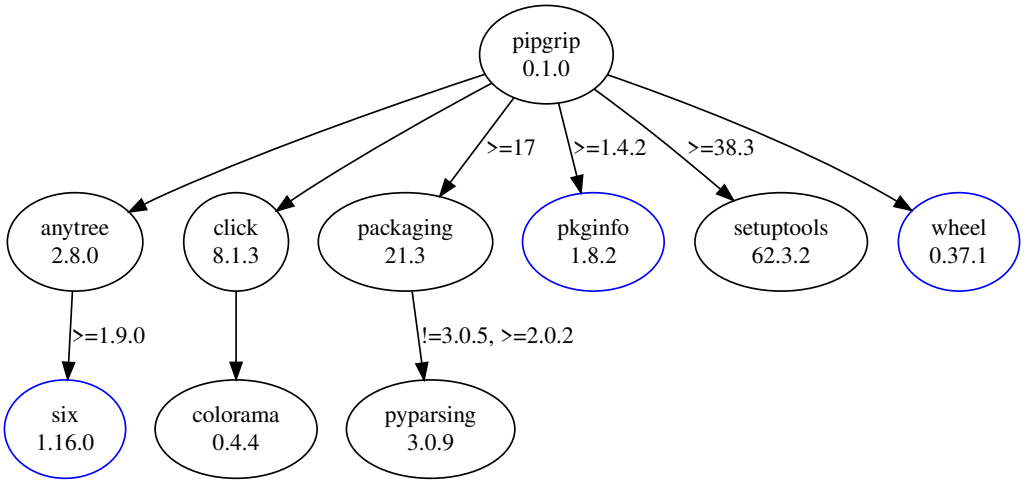


Obrázek 10.5: Anotace kruhové závislosti

10.4 Vizualizace výsledků

Jako významný přínos celého ekosystému se dá považovat exportování do formátu dot, je to výborná volba pro následnou vizualizaci grafových dat. V dot formátu graf obsahuje u uzlů tooltip, ve kterém jsou dostupná metadata a anotace. Díky anotacím je možné ve vizualizovaném grafu zvýraznit anotovaná data a tím na ně velmi rychle upozornit. Tento přístup se může hodit pro rozsáhlejší grafy, kde by v jiné reprezentaci bylo náročné anotace dohledat. Textová reprezentace také není špatná, ale pro velké grafy je hůře čitelná. Ovšem pro rychlý přehled je použitelná, minimálně se dá díky řádkovému zobrazení snadno filtrovat.

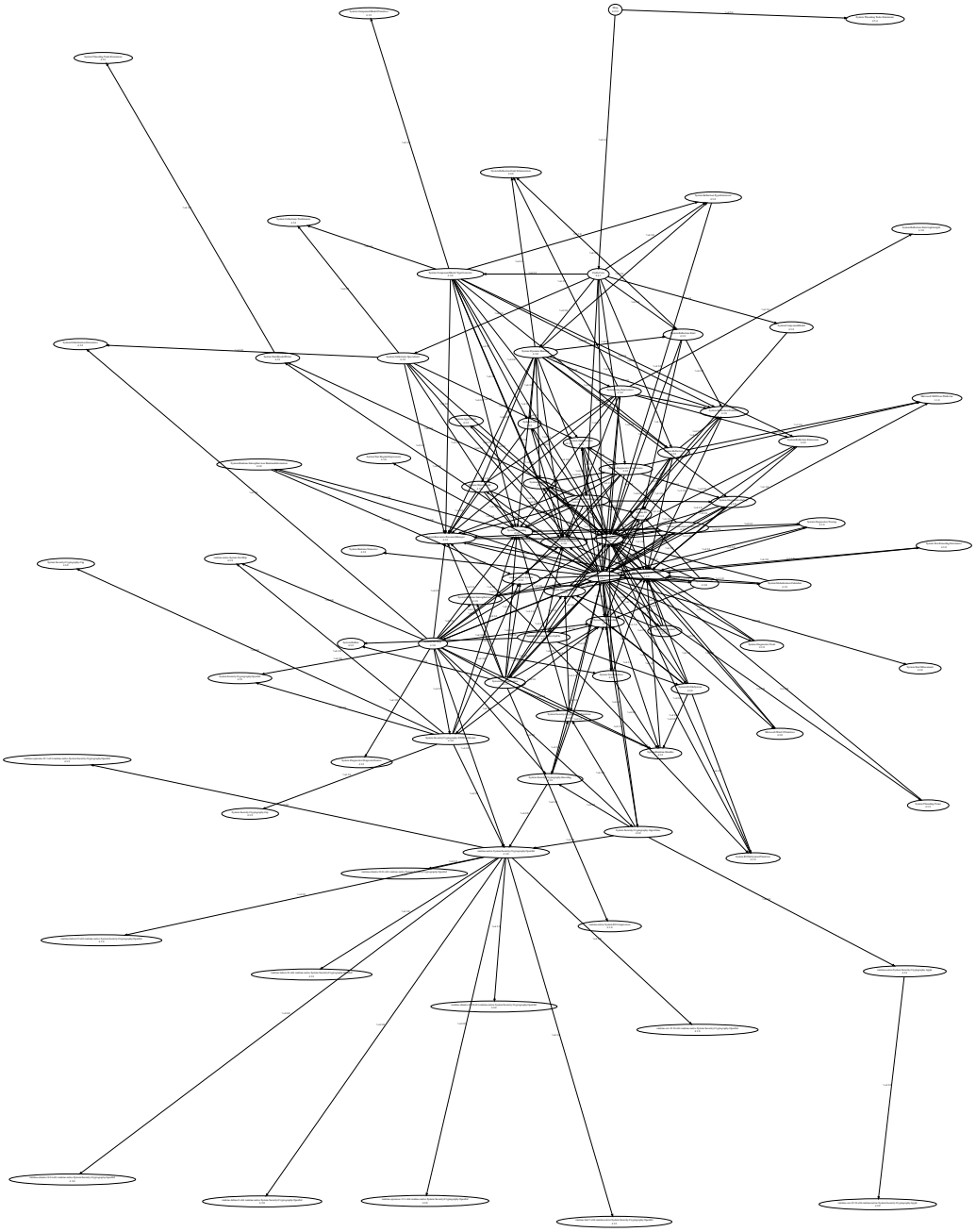
Na obrázku 10.6 je k dispozici vizualizace získaného grafu závislostí s již provedenou analýzou. Smyslem provedené analýzy bylo najít všechny balíčky s licencí typu „MIT“. Vizualizace je velmi přínosná a přehledná, dovoluje velmi rychlou orientaci v získaných datech. Díky anotacím o výsledcích analýzy jsou taktéž data vhodně zvýrazněna u balíčků s anotací.



Obrázek 10.6: Vizualizace grafu závislostí, balíček pipgrip ve verzi 0.1.0

Na obrázku 10.7 je nástin vizualizace grafu závislostí pro jeden jediný balíček z repozitáře Nuget, jedná se o balíček „Moq“ o verzi 4.17.2, který vypomáhá při testování software. Graf je velmi velký, což způsobuje jeho horší čitelnost, ale základní informace, která je z grafu vidět, je právě dopad připojení příslušného balíčku do aplikace. Dojde k přidání rozsáhlého množství závislostí, které je způsobeno požadováním byť jediného balíčku.

Na obrázku 10.7 je nástin vizualizace grafu závislostí pro jeden jediný balíček z repozitáře Nuget, jedná se o balíček „Moq“ o verzi 4.17.2, který vypomáhá při testování software. Graf je velmi velký, což způsobuje jeho horší čitelnost, ale základní informace, která je z grafu vidět, je právě dopad



Obrázek 10.7: Vizualizace grafu závislostí, balíček Moq ve verzi 4.17.2

připojení příslušného balíčku do aplikace.

Vzhledem k rozboru výsledků práce lze říci, že navržené postupy analyzování grafu závislostí jsou použitelné a dále rozšiřitelné, ať už formou analyzátorů nebo parserů.

10.5 Funkčnost nástrojů

Bez získání grafu závislostí, s příslušnými metadaty, není možné provádět přínosnou analýzu, pokud jsou data k dispozici tak je analýza vcelku přímočará. S dostupnými daty je to jen otázka algoritmizace a postupu řešení příslušného problému. Díky přítomnosti dat lze data opakovaně analyzovat několika různými nástroji. Analyzátory totiž přijmou všechna data, které budou ve správném formátu. Je tedy problém přidat další parsery, které poskytnou příslušná data pro analýzy, bez nutnosti zasahovat do analyzátorů.

Analyzátory se dají velmi snadno doplňovat, mezi všemi je určitá část implementace velmi obdobná, liší se až ve chvíli kdy jsou k dispozici data a provádí se specifická analýza. Obslužná logika pro import a export je však stejná.

U analyzátorů je vcelku výzva, jak reprezentovat výsledky analýz. Možnost uložení anotace k objektu se ukázala jako výborná cesta, díky možnosti data anotovat opakovaně z různých analyzátorů. Díky tomu je možné analyzátory řetězit a z posledního analyzátoru si nechat vrátit místo univerzálního úložného formátu třeba dot formát pro následnou vizualizaci výsledků. Implementované nástroje silně závisí na použitelnosti obecného úložného datového formátu. Vzhledem k tomu, že fungují obstojně, lze považovat datový formát za vyhovující. Ze získaných výsledků lze říci, že myšlenka práce s nástroji funguje správně a je vhodně aplikovatelná na řešení problému.

10.6 Testování nástrojů

Nástroje byly během vývoje testovány, byl hojně využíván přístup manuálního testování s reálnými daty. Dále byly využívány tvz. monkey testy, aplikace byly spouštěny s různými parametry a dostávaly náhodná data. V případě problémů došlo k okamžitému opravení funkcionality. Nástroje byly testovány i pomocí vlastnoručně vytvořených dat s cílem nástrojům způsobit chybový stav, aby byly řádně ošetřeny problémy s načítáním vstupu a samotnou analýzou. Nástroje díky otestování dokáží chybové stavy vhodně detekovat a kontrolovaně se ukončit, s předáním informace o problému uživateli.

Úspěšnost testování nástrojů nad reálnými daty zvyšovala jejich důvěryhodnost, v rámci dalšího vývoje by bylo vhodné doplnit nástroje o vlastní jednotkové testy. Bylo by rozumné otestovat části funkcionalit a ověřit je, že neobsahují chyby. Použití množiny reálných dat je sice přínosné z hlediska funkčnosti aplikace, ale stále může existovat stav, který těmito daty nebyl odhalen a mohl by obsahovat chyby.

11 Závěr

Body zadání diplomové práce byly splněny. Bylo provedeno seznámení s komponentově orientovaným vývojem softwarových systémů, více v kapitole 2. Následně se způsoby reprezentace a analýzy grafových dat, podrobněji v kapitole 3. Na základě studia literatury a hlavně podnětů z praxe byla v kapitole 5 popsána množina problémů vyžadujících analýzu informací dostupných z grafu závislostí. V kapitole 4 byla provedena analýza reprezentace komponent a jejich závislostí z různých technologiích. Následně byl navržen vhodný obecný model a úložný formát pro reprezentaci výsledného grafu komponent, podrobně popsán v kapitole 6. Na základě navržené architektury, popsané v rámci kapitoly 7, byla implementována sada nástrojů pro získání grafu závislostí a jeho následnou analýzu. Nástroje pro získání grafu závislostí jsou popsány v kapitole 8. Nástroje pro analýzu grafu závislostí jsou popsány v kapitole 9. U každého z navržených nástrojů jsou nadneseny případné problémy, které s nástroji mohou nastat. V kapitole 10 došlo ke zhodnocení získaných výsledků této práce, včetně použitelnosti úložného formátu a navržených nástrojů.

Navržený ekosystém může sloužit jako slušný základ pro budoucí rozšíření, minimálně se projevil jako použitelný. Je možné jej dál vylepšovat, aby byl specializovanější pro daný problém, který souvisí s analýzou závislostí a grafových dat. Získaný graf je možné následně analyzovat a to dokonce i opakovaně, výstupy analýz se ukládají jako anotace do uloženého grafu závislostí. Rozsah analýz pokrývá analytické potřeby nad grafem závislostí.

Po aplikaci workflow analýzy závislostí nad samotným souborem projektů, v rámci kterého jsou implementovány prezentované nástroje v jazyce C# a verzi .NET 5, bylo zjištěno že jednotky přímých závislostí mohou vytvořit graf závislostí, ve kterém jsou stovky uzlů. Následná vizualizace byla přinejmenším znepokojující, graf závislostí je velmi rozsáhlý a bez této sady nástrojů by nebylo možné jej vhodně analyzovat.

Velkým benefitem celé architektury je snadná rozšířitelnost o další nástroje a hlavně možnost jednotlivé nástroje snadno udržovat. V případě zásahu do logiky jednoho nástroje není ovlivněn žádný jiný nástroj. Rozšíření je možné provádět formou dalších analyzátorů, specializovaných parserů, případně přidáním podpory dalších výstupních formátů.

Díky pochopení architektury z této práce by neměl být problém nástroje v návaznosti na tuto práci dále rozšiřovat.

Literatura

[1] ALGODAILY.COM. *Implementing graph* [online]. algodaily.com. [cit. 2022/06/05]. Dostupné z: https://algodaily.com/lessons/implementing-graphs-edge-list-adjacency-list-adjacency-matrix.

[2] BRADA, P. A Look at Current Component Models from the Black-Box Perspective. In *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, s. 388–395, 2009. doi: 10.1109/SEAA.2009.91.

[3] BRADA, P. – JEZEK, K. Ensuring Component Application Consistency on Small Devices: A Repository-Based Approach. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, s. 109–116, 2012. doi: 10.1109/SEAA.2012.48.

[4] BUNDY, A. – WALLEN, L. *Breadth-First Search*, s. 13–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 1984. doi: 10.1007/978-3-642-96868-6_25. Dostupné z: https://doi.org/10.1007/978-3-642-96868-6_25. ISBN 978-3-642-96868-6.

[5] DOCS.MICROSOFT.COM. *Overview of the Nuget Server API* [online]. docs.microsoft.com, . [cit. 2022/06/05]. Dostupné z: https://docs.microsoft.com/en-us/nuget/api/overview.

[6] DOCS.MICROSOFT.COM. *Nuget Package Dependency Resolution* [online]. docs.microsoft.com, . [cit. 2022/06/05]. Dostupné z: https://docs.microsoft.com/cs-cz/nuget/concepts/dependency-resolution.

[7] DOCS.MICROSOFT.COM. *Package Metadata* [online]. docs.microsoft.com, . [cit. 2022/06/05]. Dostupné z: https://docs.microsoft.com/cs-cz/nuget/api/registration-base-url-resource.

[8] HARARY, F. – PRINS, G. The number of homeomorphically irreducible trees, and other species. *Acta Mathematica*. 1959, 101, 1-2, s. 141 – 162. doi: 10.1007/BF02559543.

[9] HASSMAN, M. *JSON : jednotný formát pro výměnu dat* [online]. zdrojak.cz, 2008/29/09. [cit. 2022/06/05]. Dostupné z: https://zdrojak.cz/clanky/json-jednotny-format-pro-vymenu-dat/.

[10] I_WILL_DO_IT. *Graph theory basics* [online]. geeksforgeeks.org. [cit. 2022/06/05]. Dostupné z: https://www.geeksforgeeks.org/mathematics-graph-theory-basics-set-1/.

[11] KOLÁŘ, J. *Teoretická informatika. 2. vyd.* Česká informatická společnost, 2004. ISBN 80-900853-8-5.

[12] KOŠATA, B. *XML - Úvod* [online]. root.cz, 2001/23/01. [cit. 2022/06/05]. Dostupné z: `https://www.root.cz/clanky/xml-uvod/`.

[13] PACKAGING.PYTHON.ORG. *Core metadata specifications* [online]. packaging.python.org. [cit. 2022/06/05]. Dostupné z: `https://packaging.python.org/en/latest/specifications/core-metadata/`.

[14] PIP.PYPA.IO. *Dependency Resolution* [online]. pip.pypa.io. [cit. 2022/06/05]. Dostupné z: `https://pip.pypa.io/en/stable/topics/dependency-resolution/`.

[15] RAMBA, J. *Grafová terminologie a dostupné technologie* [online]. zdrojak.cz, 2013/21/10. [cit. 2022/06/05]. Dostupné z: `https://zdrojak.cz/clanky/grafova-terminologie-a-dostupne-technologie/`.

[16] RAMBA, J. *Přehled grafových databází* [online]. zdrojak.cz, 2013/8/11. [cit. 2022/06/05]. Dostupné z: `https://zdrojak.cz/clanky/prehled-grafovych-databazi/`.

[17] RELISA. *Uvod do komponent* [online]. wiki.kiv.zcu.cz. [cit. 2022/06/05]. Dostupné z: `http://wiki.kiv.zcu.cz/UvodDoKomponent/HomePage`.

[18] SZYPERSKI, C. *Component software(2nd Ed.): Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0-201-74572-0.

[19] TARJAN, R. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, s. 114–121, 1971. doi: 10.1109/SWAT.1971.10.

[20] W3SCHOOLS.COM. *What is JSON* [online]. w3schools.com. [cit. 2022/06/05]. Dostupné z: `https://www.w3schools.com/whatis/whatis_json.asp`.

[21] WAREHOUSE.PYPA.IO. *JSON API* [online]. warehouse.pypa.io. [cit. 2022/06/05]. Dostupné z: `https://warehouse.pypa.io/api-reference/json.html`.

Seznam obrázků