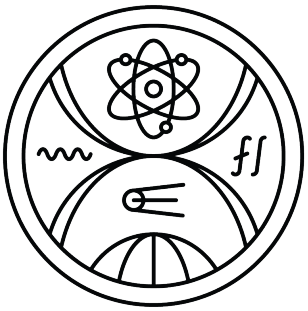
Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

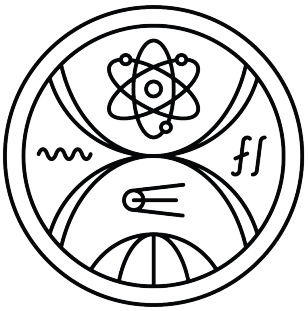


Automatická identifikácia a zobrazenie architektúry softvéru

Master thesis

2025
Bc. Filip Lukáč

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky



Automatická identifikácia a zobrazenie architektúry softvéru

Master thesis

Študijný program:	Aplikovaná Informatika
Študijný odbor:	Aplikovaná Informatika
Školiace pracovisko:	Katedra Aplikovanej informatiky
Školiteľ:	doc. Ing. Ivan Polášek, PhD.

Bratislava, 2025
Bc. Filip Lukáč

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Filip Lukáč
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Automatizovaná identifikácia a zobrazenie architektúry softvéru
Automated software architecture identification and visualization

Anotácia: Dokumentácia architektúry je nevyhnutná na pochopenie a preskúmanie softvéru. Existujúce nástroje sú zvyčajne schopné spätného inžinierstva zdrojového kódu na základné diagramy UML
alebo identifikovať základné antivzory a idiómy na identifikáciu chýb alebo kódových pachov, ktoré pomáhajú QA procesu.
Úlohou tejto práce je preskúmať možnosti automatizovanej alebo poloautomatizovanej identifikácie, extrakcie a dedukcie na úrovni architektúry softvéru.

Cieľ: Navrhnúť a vytvoriť prototyp sady nástrojov, schopných reverzného inžinierstva skutočných rozsiahlych softvérových systémov na klasifikáciu a identifikáciu architektonicky dôležitých komponentov a ich vzťahov.
Navrhnúť potrebné metódy a implementovať základy koncepcie na využitie extrahovaných informácií. Zdokumentovať získané informácie vo forme textových a vizuálnych architektonických náhľadov.

Literatúra:
Anquetil, N. et al. (2020). Modular Moose: A New Generation of Software Reverse Engineering Platform. In: Ben Sassi, S., Ducasse, S., Mili, H. (eds) Reuse in Emerging Software Engineering Practices. ICSR 2020. Lecture Notes in Computer Science(), vol 12541. Springer, Cham. https://doi.org/10.1007/978-3-030-64694-3_8

Holger M. Kienle, Hausi A. Müller:
Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation,
Science of Computer Programming, Volume 75, Issue 4, April 2010, Pages 247-263

Vedúci: doc. Ing. Ivan Polášek, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 10.11.2023

Dátum schválenia: 11.11.2023

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

.....
študent

.....
vedúci práce

Poďakovanie: TODO poďakovanie.

Abstrakt

TODO SK VERZIA:

Kľúčové slová: TODO

Abstract

TODO EN VERSION:

Keywords: TODO

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Kapitola 1

Reverzné inžinierstvo a softvérová architektúra

Softvér je v dnešnej dobe neoddeliteľnou súčasťou moderného sveta, pričom jeho rozsah a zložitosť narastajú. S rastúcou komplexnosťou softvéru sa stáva jeho analýza a pochopenie náročnejším. Dokumentácia zohráva v omto procese dôležitú úlohu nakoľko poskytuje prehľad o štruktúre systému, či vzťahoch medzi jeho komponentami.

Hoci poznáma mnoho automatizovaných nástrojov a techník zameraných na proces reverzného inžinierstva a vizualizácie, často zlyhávajú kvôli rôznorodej komplexnosti systémov, zastaralým technológiám a chýbajúcej dokumentácii.

V tejto kapitole priblížime čitateľom, čo vlastne reverzné inžinierstvo softvérovej architektúry znamená, aké sú jeho benefity, a o čo sa opiera.

1.1 Softvérová Architektúra

Softvérová architektúra predstavuje komplexný návrh a štruktúru systému integrujúcu jednotlivé komponenty do celku. Definuje základné štruktúry softvéru, vzájomné vzťahy medzi komponentami, ich externe viditeľné vlastnosti a zabezpečuje požadovanú funkcionalitu a výkonnosť. Je jedným z najdôležitejších aspektov vývoja softvéru, pretože zachytáva jeho štruktúru. Vrstvenie systému umožňuje logické rozdelenie systému do jednotlivých častí, kde každá z vrstiev má samostatnú úlohu. Komponenty systému predstavujú základné prvky, ktoré spolu komunikujú cez vopred zadefinované rozhrania a komunikačné kanály. Pri modelovaní softvérovej architektúry sa používajú **návrhové vzory**.

Ako uvádzajú Bass, Clements a Kazman v knihe *Software Architecture in Practice*, „The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both“ [1]. Táto definícia zdôrazňuje, že softvérová architektúra nie je len o

komponentoch a ich vzťahoch, ale aj o vlastnostiach, ktoré tieto vzťahy definujú.

Dokumentácia a vizualizácia architektúry je dôležitá pre procesy ako analýza, plánovanie, údržba a refaktoring. Zohrávajú dôležitú úlohu pri pochopení závislostí systému a jeho schopnosti prispôsobovať sa meniacim požiadavkám a technologickému pokroku. Tieto procesy podporujú rozhodovací proces na viacerých úrovniach vývoja, od technických rozhodnutí týkajúcich sa implementácie až po strategické rozhodnutia o smerovaní celého systému.

1.2 Reverzné inžinierstvo a proces obnovovania architektúry

Reverzné inžinierstvo je proces spätnej analýzy, ktorého cieľom je pochopiť štruktúru, funkcionalitu a logiku systému. Umožňuje analyzovať existujúci softvér bez potreby prístupu k jeho pôvodnej dokumentácii alebo návrhovým modelom. Možno ho definovať ako proces skúmania a dekonštrukcie existujúceho systému s cieľom vyextrahovať informácie o jeho štruktúre, správaní medzi komponentmi, či dizajne a architektonických rozhodnutí. Na rozdiel od vývoja softvéru, ktorý postupuje od konceptu k implementácii, reverzné inžinierstvo postupuje opačným smerom – od hotového produktu k jeho modelu alebo návrhu.

Pri obnovovaní softvérovej architektúry môže vzplanúť niekoľko rôznych otázok, ako napríklad:

- Aké sú hlavné komponenty systému a ich vzťahy?
- Aké architektonické štýly boli použité?
- Aké architektonické rozhodnutia boli prijaté a prečo?
- Aký je plán aplikácie do budúcna?

Reverzné inžinierstvo softvérovej architektúry prináša tieto a mnohé iné otázky, ktoré si vyžadujú systematický prístup k identifikácii a analýze architektonických štruktúr a rozhodnutí, ale aj odpovede k mnohým z nich. Tento proces nie je len o rekonštrukcii existujúcej architektúry, ale aj o pochopení dôvodov, prečo bola navrhnutá určitým spôsobom, a o identifikácii možných zlepšení, prípadne zastaralých závislostí a technológií.

1.3 Význam reverzného inžinierstva v softvérovej architektúre

Reverzné inžinierstvo je proces spätnej analýzy softvéru s cieľom rozanalyzovať architektonické prvky a závislosti. Tento proces je dôležitý v prípadoch, kedy je softvér nedostatočne zdokumentovaný, alebo pri nutnosti technologickej modernizácie, či hĺbkového refaktoringu systému. Vďaka reverznému inžinierstvu sme schopní pochopiť architektúru systému, závislosti medzi jednotlivými komponentami a vytvoriť prieh'adný model architektúry zachytávajúci kritické aspekty. Podľa Ibrahim et al., 2023 v článku *Context-Aware Expert for Software Architecture Recovery (CAESAR): An automated approach for recovering software architectures* sa reverzné inžinierstvo v dnešnej dobe často vykonáva polo-automatizovanými a automatizovanými spôsobmi, ktoré sú schopné detekovať architektonicky relevantné dáta, vzory a závislosti v kóde. Autori článku tvrdia, že tieto nástroje významne znižujú manuálnu prácu potrebnú na získanie poznatkov o systéme.

Problémomové však môžu byť dynamické vzťahy a komplexné závislosti medzi komponentami, ktoré sú ťažko identifikovateľné. Autori Lutellier et al. publikácie *Measuring the impact of code dependencies on software architecture recovery techniques* zamerali na analýzu vplyvu rôznych typov závislostí medzi kódom na techniky obnovy softvérovej architektúry a študovali presnosť techník jej obnovy. Cieľom bolo identifikovať a kvantifikovať závislostí medzi modulmi, súbormi, triedami, či metódami: „Many techniques have been proposed to automatically or semi-automatically recover software architectures from software code bases. Such techniques typically leverage code dependencies to determine what implementation-level units (e.g., symbols, files, and modules) form a semantic unit in a software system's architecture. To understand their effectiveness, thorough comparisons of existing architecture recovery techniques are needed."[2]

Vďaka reverznému inžinierstvu sme schopní zobraziť detailný pohľad na závislosti a štruktúry softvérových systémov. Výskum potvrdil, že rôzne typy závislosti v kóde môžu ovplyvniť presnosť používaných techník pri obnove a následnej vizualizácii architektúry. Ako ukazuje výskum, efektivita využívaných techník rôznych techník spolu s polo-automatizovanými a automatizovanými nástrojmi prináša vyššiu presnosť a nižšiu potrebu manuálnej práce. Napriek tomu, výzvou ostáva zachytenie komplexných a dynamických vzťahov medzi komponentmi, čo naznačuje potrebu jeho ďalšieho skúmania.

V nasledujúcich kapitolách si popíšeme známe nástroje a techniky využívané v reverznom inžinierstve.

1.4 Existujúce nástroje schopné reverzného inžinierstva

V tejto kapitole si priblížime prehľad niektorých existujúcich nástrojov používaných v reverznom inžinierstve, ako aj vo vizualizácii.

1.4.1 Nástroje na analýzu kódu

Medzi nástroje, ktoré sú integrované priamo do IDE, ako je napríklad IntelliJ IDE, alebo Eclipse patrí **SonarLint**. Používa sa na statickú analýzu kódu a poskytuje informácie o závislostiach a kvalite kódu. Vyuzíva sa na identifikovanie cyklické závislosti, či duplicít v kóde. IntelliJ umožňuje vizualizovať závislosti medzi modulmi, triedami pomocou vbudovanej možnosti analýzy závislostí. Na analýzu Csharp a .NET kódu existuje nástroj **NDepend**, ktorý ponúka množstvo funkcií ako CI/CD reportovanie webu, Quality Gate, či Vizualizáciu závislostí.

1.4.2 Nástroje schopné rekonštrukcie softvérovej architektúry

Veľkým prínosom v oblasti rekonštrukcie softvérovej architektúry bol *Modular Moose: a new generation of software reverse engineering platform* Anquetil, N. et al. (2020), kde autori kladli dôraz na vizualizáciu architektúry a meta-modelovanie, pričom zohľadnili tri dôležité požiadavky na analýzu softvérov: „We identified three important requirement to analyze software systems: (i) query and navigate a model to find entities of interest; (ii) visualise the software to abstract information; and (iii) annotate entities to represent meaning and reach a higher level of abstraction."[3] Moose slúži na analýzu modelov a poskytuje nástroje na vizualizáciu závislosti, pričom umožňuje tvorbu vlastných meta-modelov.

Účinným automatizovaným nástrojom na obnovu architektúry je Caesar spomínaný v publikácii Context-Aware Expert for Software Architecture Recovery (CAESAR): An automated approach for recovering software architectures, ktorý sa využíva na automatizovanú obnovu architektúry. Pri obnovovaní architektúry je užitočný nástroj ExplorViz, ktorý sme si spomenuli v podkapitole *Nástroje pre vizualizáciu a meta-modelovanie* a zameriava sa na dynamickú analýzu závislostí v runtime prostredí a je užitočným pri analýze interakcií medzi komponentami v reálnom čase.

1.4.3 Vizualizačné nástroje pre grafické zobrazenie a dokumentáciu architektúry

Táto kapitola je venovaná nástrojom používaných pre grafickú reprezentáciu a dokumentáciu softvérovej architektúry, ktoré umožňujú zobrazenie ľahšie pochopiteľných,

prehľadných zobrazení závislostí medzi komponentami, či vrstvami. Medzi ne patrí **Graphviz**, ktorý patrí medzi open-source nástroje na vizualizáciu rôznych typov grafových dát. Používa jazyk DOT, ktorý slúži na definovanie a vytváranie vzťahov medzi jednotlivými uzlami. Grafy môžu byť generované automaticky, čo mu pridáva na flexibilite,

Špecifickou metodológiou na vytváranie diagramov je **C4 model**, ktorý podporuje pohľady na viacerých úrovniach (Context, Container, Component a Code). Model poskytuje štandardizovaný prístup pri vytváraní diagramov, ktoré softvéru pomáhajú s komunikáciou vo vnútri a mimo tímov pre jeho vývoj, modelovanie hrozieb, identifikáciu rizík a hodnotenie architektúry.

Na modelovanie architektúr slúži **ArchiMate**, ktorý podporuje zjednotený prístup k vizualizácií jednotlivých aspektov. Umožňuje popisovať architektúru z rôznych pohľadov, pričom je modulárny a vhodný na vizualizáciu komplexných systémov. Medzi jeho základné princípy patria UML Diagramy. UML je v sotvérovom inžinierstve jazyk používaný na vizualizáciu, špecifikáciu, navrhovanie a dokumentáciu systémov. Má štandardizovaný spôsob zápisov vrátane konceptuálnych prvkov. Podporuje objektovo orientovaný prístup k analýze a jeho cieľom je zaznamenať kompletný návrh. Štandardizuje niekoľko druhov diagramov, ktoré sú zhlukované do troch skupín: štruktúrne diagramy, diagramy správania, a diagramy interakcie.

Vizualizačné nástroje sú vhodné na reprezentovanie poznatkov získaných použitím techník využívaných v reverznom inžinierstve, ktoré si popíšeme v nasledujúcej kapitole.

1.5 Techniky využívané pri reverznom inžinierstve

V reverznom inžinierstve jestvuje niekoľko známych techník, ktoré sa používajú na analýzu existujúceho systému s cieľom pochopiť jeho štruktúru. V tejto kapitole poskytneme prehľad najčastejších techníky využívaných v reverznom inžinierstve:

Statická a dynamická analýza kódu: Pri statickej analýze sa kód analyzuje bez jeho spúšťania, pričom sa snažíme analyzovať zdrojový kód, identifikovať závislosti v štruktúre, hierarchie tried, importov, či prepojenie medzi modulmi. Dynamická analýza sa naopak zaoberá správaním softvéru počas jeho behu. Skúma tok dát a profilovanie kódu.

Syntatická analýza: Syntatická analýza spracováva kód do AST štruktúr, ktoré potom slúžia na podrobnejšiu analýzu hierarchie kódu. Používa sa s inými technikami na pochopenie závislostí v systémoch.

Rekonštrukcia modelov: Na základe získaných informácií z kódu pomocou tejto techniky rekonštrujeme abstraktné modely softvéru, ako napríklad diagramy tried, komponentov, či modulov. Pri rekonštrukcií sa často používajú modelovacie jazyky

ako UML alebo BMPN.

Mapovanie vzťahov a tvorba grafov závislostí Prepojenia jednotlivých častí softvéru, ako sú napríklad funkcie, triedy, či moduly môžeme zachytávať pomocou závislostných grafov, ktorých uzly predstavujú entity softvéru. Hrany zobrazujú závislosti medzi týmito uzlami, ako napríklad volania funkcií a využívanie dedičnosti. Rozšírením tohto konceptu je mapovanie vzťahov, ktoré zahŕňa podrobnejšie znázornenie tokov dát v systéme a mapovanie interakcií.

Mapovanie vzťahov a tvorba grafov závislostí

Clustering je technika aplikovaná pomocou závislostných grafov. Umožňuje zoskupovanie uzlov na základe závislostí. Clustering je špecifický tým, že redukuje zložitosť vizualizácie zlučovaním súvisiacich uzlov do jedného klastru(skupiny), ktorý môže predstavovať entitu, či komponent v softvére. Existujú dva typy clusteringu, a to je *hierarchický clustering*, a *zhlukovanie podľa podobností*.

Hierarchický clustering – používa sa pri vytváraní stromových štruktúr s menšími skupiny komponentov postupne zoskupovanými do väčších celkov.

Zhlukovanie podľa podobnosti – Zhlukuje komponenty s podobnými závislosťami, napríklad moduly, ktoré zdieľajú spoločné funkcie, môžu byť zobrazené ako jeden celok.

Existuje mnoho druhov clusteringových algoritmov, ktoré sa používajú na základe typov údajov a aplikácií. Zhlukovať môžeme do partícií pomocou K-means algoritmu, ktorý zozbierava dáta do K klastrov na základe priemeru (centroidu) bodov v každom klastri. Jeho efektívnejšou variantou je K-means++. Často využívanými sú algoritmy hierarchického klasteringu, ktoré vytvárajú stromovú štruktúru, DBSCAN klastering využívaný pri identifikovaní oblastí s vyššou hustotou bodov a ich oddiaľovaní od oblastí s nižšou hustotou, či klastering pomocou grafových závislostí, kde napríklad algoritmus *Spectral Clustering* využíva vlastnosti spektra Laplaceovej matice grafu a vytvára graf, v ktorom vrcholy sú body a hrany medzi nimi reprezentujú podobnosť, pričom celý graf delí na menšie podgrafy (klastre).

Tieto techniky umožňujú vizualizovať veľké množstvo dát do klastrov na základe podobností, čím umožní softvérovému architektovi identifikovať závislosti medzi entitami v softvéri, klasifikácií komponentov a modulov, či detekcií anomálií v kóde.

Kapitola 2

Teoretické Východiská

V tejto kapitole poskytneme prehľad literatúry, ktorá sa zaoberá dokumentáciou a vizualizáciou softvérovej architektúry. Zameriame sa na existujúce techniky, nástroje a postupy využívané v tejto oblasti, pričom analyzujeme ich prínosy a obmedzenia. Kapitola tiež približuje konkrétne prípady použitia vizualizácie a nástroje, ktoré podporujú analýzu a pochopenie štruktúry a závislostí v softvérových systémoch.

2.1 Prehľad literatúry

Existuje množstvo štúdií, ktoré sa zaoberajú analýzou štruktúr systémov a metódami na ich vizualizáciu. Ako uvádzajú Bass, Clements a Kazman v knihe *Software Architecture in Practice*, „The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.“ [1]. Dokumentácia a vizualizácia týchto štruktúr je preto dôležitým aspektom na pochopenie systému a jednotlivých závislostí. Autori sa venujú základom architektonického návrhu softvéru, pričom jasne zdôrazňuje význam dokumentácie pri optimalizácii softvérového vývoja. Dielo ponúka teoretický základ, ale aj praktické odporúčania a postupy pre architektonický návrh softvéru, čo je dôležité pri refaktorovaní a modernizácií veľkých systémov.

Prehľad literatúry zdôrazňuje dôležitosť vizualizácie a dokumentácie ako nástrojov na pochopenie štruktúry systému. V nasledujúcich častiach sa budeme venovať konkrétnym nástrojom a technikám, ktoré podporujú tieto procesy, a poukážeme na ich praktické využitie.

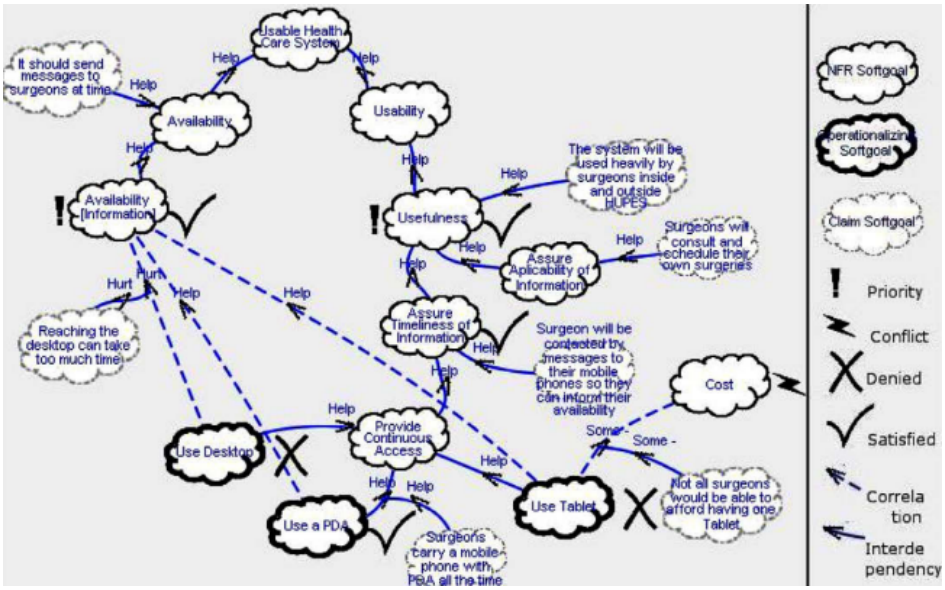
2.1.1 Architektúra a vizualizácia

Niektoré publikácie, ako napríklad *Visualization and comparison of architecture rationale with semantic web technologies* od Lópeza et al. (2009) sa zameriavajú na na

vizualizáciu a porovnávanie architektonických závislostí pomocou grafov vzájomných závislostí cieľov (SIG).

Efektívna vizualizácia architektúry je dôležitá pre jednoduchšie pochopenie analyzovaného systému, čo zohráva dôležitú úlohu pri nahrádzaní častí systémov. Autori článku uvádzajú najmenej tri body záujmov sotférových architektov: „**Understanding architectural decision changes**: reviewing documentation of a system to understand the evolution of its architecture, and to evaluate changes required to satisfy new requirements. **Evaluating current designs**: reviewing architecture decisions, comparing them to rejected alternatives, and understanding specific arguments to support each decision. **Reusing past designs**: reviewing past solutions to learn from them and, eventually, use a similar reasoning to solve a new problem.“ [4]. Pre pochopenie vývoja softvérovej architektúry je dôležité identifikovať, aké rozhodnutia, a prečo nastávali v čase, keď sa softvér vyvíjal. Prehodnotenie týchto rozhodnutí poskytuje cenné informácie o tom, prečo tieto rozhodnutia nastali, aké argumenty ich podporovali a pomáha identifikovať silné a slabé stránky súčasného návrhu. Tento prístup podporuje vytváranie dynamických vizualizácií, ktoré umožňujú používateľom lepšie porozumieť architektonickým vzorom a vzťahom v rámci systému, čím sa zvyšuje celková kvalita dokumentácie a zrozumiteľnosť softvérovej architektúry.

Ako príklad SIG použijeme obrázok publikovaný v článku, ktorý zachytáva „chirurgický riadiaci systém určený na plánovanie a plánovanie operácií pre Univerzitnú nemocnicu Pedra Hernesta (HUPES) v Rio de Janeiro v Brazílii."[4] Na nasledujúcom obrázku možno vidieť použiteľnosť NFR.



Obr. 2.1: Example: partial SIG – health insurance system, López et al. (2009).

Tento príklad zobrazuje ako môžu byť architektonické závislosti vizualizované prostredníctvom grafickej reprezentácie s využitím SIG.

Grafická vizualizácia NFR, či závislostí medzi komponentmi pomocou SIG pomôže urýchliť pochopenie architektonického návrhu aplikácie, alebo hodnotiť budúce zmeny v súčasnej architektúre. „Comparison of SIGs can support, for example, the selection between reuse candidates by identifying domain constraints or contexts that are more similar to the problem at hand. Furthermore, comparison of sequentially elaborated SIGs can identify changes between versions and help to trace the evolution of an architecture design process. "[4]

2.1.2 Nástroje pre vizualizáciu a meta-modelovanie

Nástroje na vizualizáciu a meta-modelovanie sú dôležité pri analýze a pochopení komplexných softvérových systémov. Umožňujú získať prehľad o štruktúre a vzťahoch v rámci systému bez potreby manuálnej analýzy zdrojového kódu, čo šetrí čas a znižuje riziko ľudských chýb. Vizualizácia pomáha vývojárom a architektom pochopiť štruktúru systému, jeho komponenty a vzťahy medzi nimi. Meta-modelovanie je zase metodika, ktorá umožňuje popisovať tieto vzťahy formálne a vytvárať modely, ktoré možno neskôr vizualizovať, či ďalej spracovávať.

ExplorViz

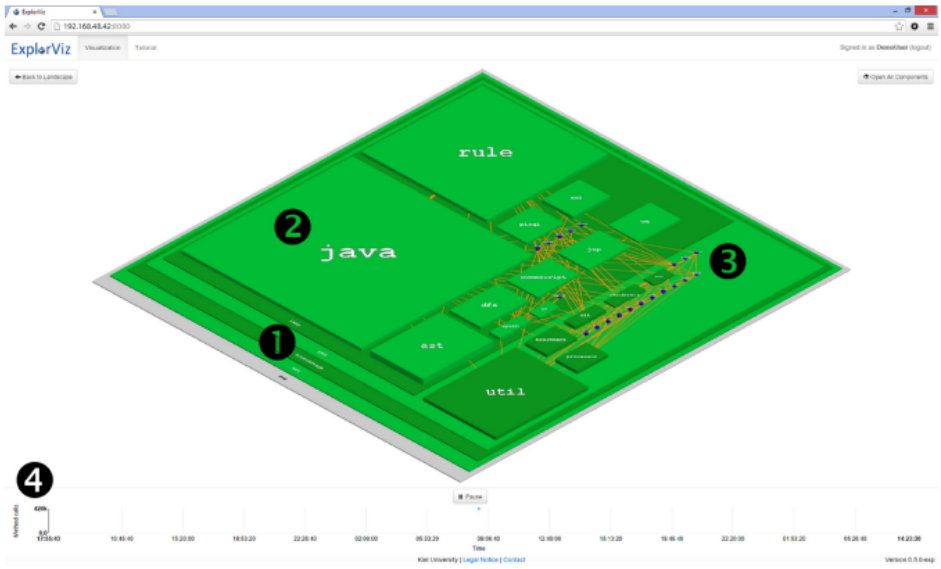
Jedným z významných nástrojov v tejto oblasti je *ExplorViz*, ktorý predstavil Fittkau, Krause a Hasselbring (2017) vo svojom článku *Software landscape and application visualization for system comprehension with ExplorViz*. Tento nástroj poskytuje hierarchické a viacúrovňové vizualizácie, ktoré sú užitočné na pochopenie komplexných aplikácií. Pracuje na viacerých úrovniach, od celkového prehľadu systémového prostredia (landscape-level) až po detailné zobrazenie jednotlivých aplikácií (application-level). „In particular, the ExplorViz approach offers hierarchical visualization and multi-layer monitoring from landscape level to application level.“ [5]. Nástroj bol určený na podporu pochopenia softvérových systémov prostredníctvom vizualizácie ich architektúry a dynamického správania. Bol vyvinutý s cieľom získať rýchly prehľad o štruktúre a správaní veľkých a komplexných aplikácií. Jeho hlavným cieľom je eliminovať potrebu manuálneho študovania zdrojového kódu a znížiť náklady spojené s jeho údržbou.

ExplorViz ponúka niekoľko kľúčových funkcií, ktoré ho odlišujú od iných nástrojov na vizualizáciu softvérových systémov ako:

Hierarchická vizualizácia - poskytuje viacúrovňové zobrazenia systému, od globálneho pohľadu na celkovú infraštruktúru a vzťahy medzi jednotlivými aplikáciami (landscape-level), až po detailné zobrazenie štruktúry konkrétnej aplikácie (application-level); Dynamická analýza správania systému - Nástroj podporuje dynamickú analýzu,

čo znamená, že dokáže zachytávať a vizualizovať správanie aplikácie počas jej behu, čo je potrebné napríklad pri analýze výkonnostných problémov; Podpora viacvrstvovej analýzy - ExplorViz zobrazuje rôzne úrovne vrstiev systému, ako sú napríklad vrstva infraštruktúry, aplikačná vrstva a komunikačná vrstva.

Na nasledujúcom obrázku je znázornená vizualizácia aplikačnej vrstvy prostredníctvom ExplorViz:



Obr. 2.2: Example: Application-level visualization, Fittkau et al. (2017).

Obrázok ukazuje detailnú štruktúru aplikačnej vrstvy, kde sú jednotlivé komponenty (moduly) reprezentované ako samostatné bloky. Komunikačné vzťahy medzi nimi sú znázornené spojmi, ktoré indikujú volania metód alebo výmenu dát. Hierarchická a viacúrovňová vizualizácia na landscape a aplikačnej úrovni im umožňuje získať prehľad o kritických závislostiach a štruktúre systému bez potreby detailnej manuálnej analýzy kódu. Autori publikácie poukazujú na to, že podobné nástroje môžu zefektívniť analýzu komplexných aplikácií.

Napriek tomu, že ExplorViz je užitočný na analýzu a vizualizáciu, má niekoľko obmedzení: - Hoci nástroj podporuje dynamickú analýzu, jej presnosť a využiteľnosť závisí od kvality vstupných údajov a testovacích scenárov - Pri analýze veľmi veľkých systémov môže vizualizácia obsahovať veľké množstvo detailov, čo znižuje prehľadnosť - Vyžaduje manuálnu konfiguráciu a integráciu do analyzovaných aplikácií, čo môže byť časovo náročné

Modular Moose

Modular Moose predstavuje novú generáciu nástrojov použiteľných pri detailnom skúmaní softvérových systémov na základe meta-modelu zvanom FAMIX. Tento meta-model poskytuje reprezentáciu softvérových artefaktov a vzťahov medzi nimi. Modular

Moose nadväzuje na pôvodnú platformu Moose, pričom cieľom autorov bolo vytvoriť prostredie, ktoré by umožnilo získať lepší prehľad o zložitých systémoch. V nadväznosti na to si uvedomovali, že prostredie by malo byť ľahko rozšíriteľné a modulárne. Uvedomovali si, že pri dnešnej heterogénnosti a zložitosti softvérov doposiaľ vyvinuté nástroje na analýzu kódu nie sú postačujúce. Preto základnými princípmi Modular Moose je jazyková nezávislosť, meta-model a podpora integrácie s inými nástrojmi.

Autori opisujú použitie FAMIX meta-modelu, ktorý dokáže popísať rôznorodé softvéry napísané v rôznych programovacích jazykoch, pričom poskytuje reprezentáciu softvérových entít ako sú metódy, triedy, moduly a pod., či vzťahov medzi nimi, ako je napríklad dedičnosť a volania metód. Vďaka tomuto prístupu získali dôležitý základ na prepoužitie pri analýze rôznych systémov. Moose je navrhnutý ako otvorené prostredie, umožňuje kombinovať rôzne druhy analýz a výsledky dokáže graficky reprezentovať formou rôznych spôsobov vizualizácií, ako napríklad pomocou grafických máp, či stromových štruktúr. Podporuje iteratívny prístup, vďaka ktorému môžu analytici postupne rozširovať svoje modely na základe získaných poznatkov a ľahšie identifikovať príliš komplexné časti kódu, ktoré si vyžadujú refaktoring.

Moose so sebou prináša niekoľko praktických výhod a tvorí základ pre napredujúci vývoj a výskum v reverznom inžinierstve a vizualizácii softvérových architektúr. Vďaka univerzálnemu meta-modelu a otvorenej architekrúre je model možné v budúcnosti lepšie rozšíriť o podporu dynamicky konfigurovateľných systémov, či lepších metrík pre kvalitu kódu. Napriek tomu má niekoľko nevýhod, medzi ktoré patrí najmä časová náročnosť pri nasadzovaní do nového projektu, nakoľko je pravdepodobné, že bude treba prispôsobovať meta-model. Aj keď bol navrhnutý pre rozsiahle systémy, kvalita výstupov závisí od kvality vstupných dát a vizualizované výsledky môžu byť častokrát nie veľmi prehľadné.

Dockerizácia a architecture recovery

Vďaka nárastu cloud-native prístupu vývoju a nasadzovania systému sa zvyšuje používanie kontajnerizácie, keďže dokážu poskytnúť izolované a reprodukovateľné prostredie pre beh aplikácií. Aplikácia alebo jej časti dokážu byť zaobalené spolu so všetkými závislosťami, konfiguračnými súbormi a knižnicami do jedného kontajnera, čím sa zjednodušuje proces nasadzovania, zvyšuje škálovateľnosť a znižuje riziko problémov s kompatibilitou medzi rôznymi prostrediami. V kontexte mikroslužnových architektúr na rozdiel od monolitu je aplikácia členená na množstvo samostatne nasaditeľných častí, služieb, kde sú kontajnery využívané najmä na nezávislé spravovanie jednotlivých komponentov. Kým aplikácie v minulosti boli často monolitické, ich architektúru bolo možné odvádzať manuálnou alebo polo-automatizovanou analýzou statického zdrojového kódu. Vo svete mikroslužieb vďaka dockerizácii ponúka docker compose dekla-

ratívny spôsob definovania orchestrácie vzťahov medzi službami, čo umožňuje rýchlejšie(mnohokrát ihneď) a jasnejšie získať prehľad o topológii a závislostiach systému. V prípade kontajnerizovaných aplikácii, Dockerfile a Docker compose súbory obsahujú zdroje informácií explicitne popisujúce štruktúru, závislosti a prevádzkové podmienky jednotlivých služieb v systéme.

Ricardo Jorge de Araújo Ferreira vo svojej dizertačnej práci *Recovery of Software Architecture from Code Repositories* [6] kladol dôraz na význam nástrojov, ktoré umožňujú vizualizáciou dockerizovaných prostredí. Medzi ne patria nástroje ako *docker-compose-viz (používaný na vizualizáciu docker-compose súborov)*, *docker-visualizer*(určený pre vizualizáciu Docker Swarm klastra) alebo *dockerviz*(vyžaduje prístup k Docker API a spusteného docker daemona), ktoré dokážu automatizovane extrahovať architektonické informácie a prezentovať ich v podobe grafov alebo schém. Obsahujú taktiež dôležité parametre, ako sú prepojenia na sieťovej úrovni, externé služby, premenné prostredia, prípadne zdieľané zdroje a mnoho ďalších.

Pri obnovovaní architektúry je dôležité reflektovať aj špecifiká dockerizovaných systémov, kde analýzou dockerfile a docker compose súborov dokážeme získať ucelený obraz o systéme a získať relatívne presné informácie o architektúre systému. Nevýhodou spomínaných nástrojov je, že dva z nich vyžadujú spustené prostredie, a teda nie je možné ich použiť pri statickej analýze, no naopak výhodou je, že dokážu zachytiť zmeny v reálnom čase. Na rozdiel od nich, docker-compose-viz je určený na statickú analýzu docker compose súborov, no má výrazne obmedzenú granularitu analyzovaných informácií a neponúka informácie o službách, ktoré sú definované, aké väzby a závislosti medzi nimi sú, a neanalyzuje konfiguráciu kontajnerov, parametre, objemy, secrets(tajomstvá), ani nedokáže analyzovať správanie jednotlivých služieb pri nasadení v rôznych prostrediach.

Kapitola 3

Cinderella

Návrh TODO

3.1 Technológie a použité nástroje

popis cindy, nástrojov, vlastných implementovaných scriptov a algoritmov.

3.2 Prototyp

popis prototypu

TODO NAVRH PREC A ROVNO IMPLEMENTACIA, IDEA, POPIS a POD.

Kapitola 4

Implementácia

Implementácia TODO, ku každej kapitole pridať teoretický základ, koncepty, štúdie, analýzu výsledkov, možné vylepšenia, TODO: Táto časť sa odkazuje na poznatky o cindy, ako funguje, detectory, modely, results a pod., ktoré ešte nie sú popísané a budú spomenuté v kapitole predtým.

4.1 Statická a dynamická analýza kódu

todo popis

4.2 Clustering

K-Means, DBSCAN, popis kritérií vyhľadávaných skupín, závislosti ako imports,

4.3 Vizualizácia dát (Altova XMLSpy, Visual Paradigm Oxygen XML)

Popis použitia altova XML a iných pluginov, obrázky, použitie.

4.4 Docker vizualizácia

Analýzou prác a nástrojov spojených s vizualizáciou kontajnerizovaných prostredí v sekcii 2.1.2 sme identifikovali niekoľko nedostatkov, kde sme poukázali na zásadné problémy ako:

- Nástroje *docker-visualizer* či *dockerviz* vyžadujú spustené prostredie, čo znemožnuje ich použitie na statickú analýzu,

- Napriek tomu, že *docker-compose-viz* je určený na statickú analýzu docker compose súborov, jeho výstupy sú príliš všeobecné a neposkytujú dostatočne podrobné informácie o službách a ich konfiguráciách, granularita výstupov je limitovaná.

Tieto obmedzenia nás motivovali k vývoju vlastného riešenia. Navrhli sme dvojstupňový proces pozostávajúci z vývoja prototypových skriptov a následnej implementácie detektorov. Cieľom bolo nielen zabezpečiť spracovanie docker compose súborov a ich vizualizácia, ale najmä automatizovanie procesu extrakcie dát z docker compose súborov a ich uloženie do results2.json.

Na začiatku sme implementovali prototypové scripty, ktoré generovali .dot súbory pre Graphviz a vizualizovali kontajnerové služby spolu s ich vzťahmi a závislosťami. Tento prístup však nebol úplne flexibilný nakoľko pracoval iba so súborom, ktorý bol určený na vstupe. Na základe spätnej väzby od školiteľa sme následne implementovali detektory, ktoré automatizovane extrahujú údaje s vyššou granularitou ako nástroj *docker-compose-viz*. Tie automaticky spracovávajú docker compose súbory a ukladajú dáta v JSON formáte v results2.json.

Docker Compose

Navrhli sme tri prototypové skripty, *docker-compose-to-dot.sh*, *docker-compose-to-dot-detailed.sh* a *docker-compose-generate-services-individually*, ktoré spracovávajú docker compose súbory v rôznej granularite. Scripty načítavajú údaje zo súborov na vstupe (docker compose) cez yaml2json, extrahujú údaje a generujú .dot súbory s hierarchickými vzťahmi medzi kontajnermi s rôznou granularitou v závislosti od použitého scriptu.

Script *docker-compose-to-dot.sh* má menej podrobné informácie vo výstupe, spracováva závislosti (depends on) a premenné prostredia pre konkrétnu službu. Jeho rozšírením je *docker-compose-to-dot-detailed.sh*, ktorý extrahuje základné údaje ako Názov služby, názov kontajnera, používateľ (user) a obraz služby (image). Obsahuje informácie o portoch, volumes, a spúšťaných príkazoch (command) a rozdeľuje premenné prostredia do niekoľkých kategórií. Tým, že script generuje výstup veľmi podrobný, pri zložitejšom docker compose súbore bol výstup bez priblíženia nečitateľný, vyžadoval manuálne pohybovanie sa po obrázku a detaily boli príliš nahusto zoskupené, čo prispelo k vzniku *docker-compose-generate-services-individually*, ktorý každý pre každý servis generuje samostatný .dot súbor a zachováva detailné informácie ako jeho predchodca.

```
1 #!/bin/bash
2
3 ## The script generates a detailed dot file for graphviz from a docker
      -compose.yml file.
4 ## version 1.0
```

```
##
## Usage:
##     ./docker-compose-to-dot-detailed.sh <docker-compose-file-path>
##
## Notes:
##     - Ensure the input file exists.
##     - Generates a .dot file that shows detailed service information with color-coded sections.
##
## Example:
##     ./docker-compose-to-dot-detailed.sh ../../../_sample_repos/dataverse/docker-compose-dev.yml

if [ "$1" == "-h" ] || [ "$1" == "--help" ] || ([ -t 0 ] && [ "$#" == 0 ]); then
    echo "Usage: ./docker-compose-to-dot-detailed.sh <docker-compose-file-path>"
    echo "Generates a detailed Graphviz .dot file from a docker-compose.yml file with color-coded sections."
    exit 1
fi

composeFile="$1"

node - <<EOF > compose-detailed.dot
const fs = require('fs');
const yaml = require('js-yaml');

try {
    const composeContent = fs.readFileSync("$composeFile", 'utf8');
    const composeData = yaml.load(composeContent);
    const services = composeData.services || {};

    console.log("digraph DockerComposeDetailed {");
    console.log("  rankdir=TB;");
    console.log("  node [shape=box, style=filled, fontname=\"Courier New\", fontsize=10];");

    const renderedNodes = new Set();

    function renderServiceNode(serviceName, depth = 0) {
        if (renderedNodes.has(serviceName)) {
            return;  // avoid redundant nodes
        }
        renderedNodes.add(serviceName);

        const service = services[serviceName];
```

```
46          if (!service) {
47              return;
48          }
49
50          const serviceNodeId = \`service_\${serviceName.replace(/[^a-zA
                -Z0-9]/g, '_')}\`;
51          console.log(\`  \${serviceNodeId} [label="\${serviceName}",
                shape=box, style=filled, color=lightgreen];\`);
52
53          function printDetailNode(detailLabel, detailContent,
                nodeSuffix, color = "lightblue") {
54              if (detailContent) {
55                  const detailNodeId = \`\${serviceNodeId}_\${nodeSuffix
                        }\`;
56                  console.log(\`  \${detailNodeId} [label="\${
                        detailLabel}:\\n\${detailContent.trim()}", shape=
                        box, color=\${color}, style=filled];\`);
57                  console.log(\`  \${serviceNodeId} -> \${detailNodeId}
                        [label="\${detailLabel}"];\`);
58              }
59          }
60
61          printDetailNode("Container␣Name", service.container_name, "
                container_name", "lightcoral");
62          printDetailNode("Image", service.image, "image", "lightyellow"
                );
63          printDetailNode("User", service.user, "user", "lightcyan");
64          if (service.ports) {
65              printDetailNode("Ports", service.ports.join("\\n"), "ports
                    ", "lightgoldenrodyellow");
66          }
67          if (service.volumes) {
68              printDetailNode("Volumes", service.volumes.join("\\n"), "
                    volumes", "lightpink");
69          }
70          if (service.command) {
71              const commandContent = Array.isArray(service.command) ?
                    service.command.join('␣') : service.command;
72              printDetailNode("Command", commandContent, "command", "
                    lightgrey");
73          }
74          if (service.environment) {
75              let jvmArgsContent = "";
76              let otherEnvContent = "";
77              let authEnvContent = "";
78
79              Object.entries(service.environment).forEach(([key, value])
```

```
                => {
80              const envEntry = \`\${key}=\${value}\`;
81              if (key === 'JVM_ARGS') {
82                  jvmArgsContent += value.replace(/ -/g, '\\n-') + "
                        \\n";
83              } else if (key.toLowerCase().includes('auth') || key.
                    toLowerCase().includes('secret') || key.
                    toLowerCase().includes('password')) {
84                  authEnvContent += envEntry + "\\n";
85              } else {
86                  otherEnvContent += envEntry + "\\n";
87              }
88          });
89
90          if (jvmArgsContent) {
91              printDetailNode("JVM_ARGS", jvmArgsContent, "jvm_args"
                    , "lightsteelblue");
92          }
93
94          if (authEnvContent) {
95              printDetailNode("Authentication Vars", authEnvContent,
                    "auth_vars", "lightcoral");
96          }
97
98          if (otherEnvContent) {
99              printDetailNode("Environment Variables",
                    otherEnvContent, "env_vars", "lightblue");
100         }
101     }
102
103     if (service.depends_on) {
104         service.depends_on.forEach((dependency) => {
105             const dependencyNodeId = \`service_\${dependency.
                    replace(/[^a-zA-Z0-9]/g, '_')}\`;
106
107             if (!renderedNodes.has(dependency)) {
108                 renderServiceNode(dependency, depth + 1);
109             }
110
111             console.log(\`  \${serviceNodeId} -> \${
                    dependencyNodeId} [label="depends on", style=
                    dotted, color=gray];\`);
112         });
113     }
114 }
115
116 Object.keys(services).forEach((serviceName) => {
```

```
117         renderServiceNode(serviceName);
118     });
119
120     console.log("}");
121 } catch (e) {
122     console.error("Error␣parsing␣docker-compose␣file:", e.message);
123 }
124 EOF
125
126 echo "Graphviz␣file␣'compose-detailed.dot'␣was␣created.␣Visualize␣it␣
    using:"
127 echo "dot␣-Tpng␣compose-detailed.dot␣-o␣compose-detailed.png"
```

Algoritmus 4.1: Skript docker-compose-to-dot-detailed.sh

Zobrazený script generuje výstup v podobe .dot súboru:

```
1
2 digraph DockerfileMap {
3   rankdir=TB;
4   node [shape=box, style=filled, color=lightblue];
5   node_0_ARG [label="ARG", shape=box, style=filled, color=lightgreen];
6   node_1_FROM [label="FROM", shape=box, style=filled, color=lightgreen
      ];
7   node_0_ARG -> node_1_FROM [label="next"];
8   node_2_FROM [label="FROM", shape=box, style=filled, color=lightgreen
      ];
9   node_1_FROM -> node_2_FROM [label="next"];
10  node_3_ENV [label="ENV", shape=box, style=filled, color=lightgreen];
11  node_2_FROM -> node_3_ENV [label="next"];
12  env_3_0 [label="SCRIPT_DIR␣=␣/scripts", shape=ellipse, color=
      lightyellow];
13  node_3_ENV -> env_3_0 [label="defines"];
14  node_4_ENV [label="ENV", shape=box, style=filled, color=lightgreen];
15  node_3_ENV -> node_4_ENV [label="next"];
16  env_4_0 [label="SECRETS_DIR␣=␣/secrets", shape=ellipse, color=
      lightyellow];
17  node_4_ENV -> env_4_0 [label="defines"];
18  node_5_ENV [label="ENV", shape=box, style=filled, color=lightgreen];
19  node_4_ENV -> node_5_ENV [label="next"];
20  env_5_0 [label="SOLR_TEMPLATE␣=␣/template", shape=ellipse, color=
      lightyellow];
21  node_5_ENV -> env_5_0 [label="defines"];
22  node_6_ENV [label="ENV", shape=box, style=filled, color=lightgreen];
23  node_5_ENV -> node_6_ENV [label="next"];
24  env_6_0 [label="PATH␣=␣{PATH}:{SCRIPT_DIR}", shape=ellipse, color=
      lightyellow];
25  node_6_ENV -> env_6_0 [label="defines"];
```

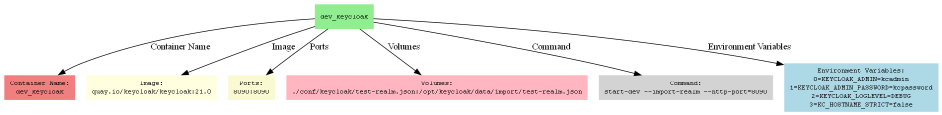
```
26   node_7_ENV [label="ENV", shape=box, style=filled, color=lightgreen];
27   node_6_ENV -> node_7_ENV [label="next"];
28   env_7_0 [label="BOOTSTRAP_DIR␣=␣{SCRIPT_DIR}/bootstrap", shape=
        ellipse, color=lightyellow];
29   node_7_ENV -> env_7_0 [label="defines"];
30   node_8_ARG [label="ARG", shape=box, style=filled, color=lightgreen];
31   node_7_ENV -> node_8_ARG [label="next"];
32   env_8_0 [label="APK_PACKAGES␣=␣curl", shape=ellipse, color=
        lightyellow];
33   node_8_ARG -> env_8_0 [label="defines"];
34   node_9_RUN [label="RUN", shape=box, style=filled, color=lightgreen];
35   node_8_ARG -> node_9_RUN [label="next"];
36   node_10_APK [label="APK", shape=box, style=filled, color=lightgreen
        ];
37   node_9_RUN -> node_10_APK [label="next"];
38   comment_10_0 [label="#␣Make␣our␣working␣directories
39 ␣␣mkdir␣-p␣{SCRIPT_DIR}␣{SECRETS_DIR}␣{SOLR_TEMPLATE}", shape=note,
       color=blue, fontcolor=gray];
40   node_10_APK -> comment_10_0 [label="has␣comment"];
41   node_11_COPY [label="COPY", shape=box, style=filled, color=
        lightgreen];
42   node_10_APK -> node_11_COPY [label="next"];
43   node_12_COPY [label="COPY", shape=box, style=filled, color=
        lightgreen];
44   node_11_COPY -> node_12_COPY [label="next"];
45   node_13_RUN [label="RUN", shape=box, style=filled, color=lightgreen
        ];
46   node_12_COPY -> node_13_RUN [label="next"];
47   node_14_COPY [label="COPY", shape=box, style=filled, color=
        lightgreen];
48   node_13_RUN -> node_14_COPY [label="next"];
49   node_15_COPY [label="COPY", shape=box, style=filled, color=
        lightgreen];
50   node_14_COPY -> node_15_COPY [label="next"];
51   node_16_RUN [label="RUN", shape=box, style=filled, color=lightgreen
        ];
52   node_15_COPY -> node_16_RUN [label="next"];
53   node_17_ENTRYPOINT [label="ENTRYPOINT", shape=box, style=filled,
        color=lightgreen];
54   node_16_RUN -> node_17_ENTRYPOINT [label="next"];
55   node_18_CMD [label="CMD", shape=box, style=filled, color=lightgreen
        ];
56   node_17_ENTRYPOINT -> node_18_CMD [label="next"];
57   node_19_LABEL [label="LABEL", shape=box, style=filled, color=
        lightgreen];
58   node_18_CMD -> node_19_LABEL [label="next"];
59 }
```

Algoritmus 4.2: Ukážka výstupu .dot súboru pre Graphviz

Vygenerované súbory sme neskôr konvertovali do PNG pomocou *dot -Tpng compose.dot -o compose.png* príkazu, kde compose.dot je názov .dot súboru, ktorý sa konvertuje a compose.png je názov výstupného obrázka. V nasledujúcich príkladoch zobrazíme detailný výstup celého docker compose súboru po spracovaní a výstup v podobe zobrazenia konkrétnej služby z predošlého spracovania:

Obr. 4.1: Detailný výstup po spracovaní docker-compose súboru.

Ako možno vidieť, napriek tomu, že granularita spracovaných údajov je vysoká, výstupný obrázok je minimalistický a na prvý pohľad ťažko čitateľný.



Obr. 4.2: Príklad zobrazenia samostatnej služby.

Predchádzajúci príklad obsahuje informácie ku konkrétnej službe, pričom celkové závislosti dokážeme spozorovať na detailnom obrázku v predošlom príklade.

TODO: Popísať prečo prototyp nie je úplne ok, hustotu informácií, chýbajúca automatizácia a prechod na detektory.

Implementácia detektorov

TODO popis detektorov,

```
function docker_compose_dd2json_detector() {
    query-files2 "" "docker_docker-compose" -a file | while read f; do
        yaml2json "$f" | jsontool -E "this.file=\"$f\""
    done | jsontool -g -e '
        this.results = {
            file: this.file,
            services: Object.entries(this.services || {}).map(([name, service]) => {
                let env = Array.isArray(service.environment)
                    ? service.environment.reduce((acc, entry) => {
                        const [key, ...valueParts] = entry.split("=");
                        acc[key] = valueParts.join("=");
                        return acc;
                    }, {})
```

```
14 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣service.environment␣||␣{};
15
16 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣jvmArgs␣=␣env.JVM_ARGS
17 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣?␣env.JVM_ARGS.split(/(?:\\n|(?=-))/g).filter(
       Boolean).map(arg␣=>␣arg.trim())
18 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣[];
19
20 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣(jvmArgs.length␣>␣0)␣{
21 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣env.JVM_ARGS␣=␣jvmArgs;
22 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}␣else␣{
23 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣delete␣env.JVM_ARGS;
24 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣}
25
26 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣command␣=␣Array.isArray(service.command)
27 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣?␣service.command.filter(Boolean)
28 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣service.command
29 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣?␣[service.command]
30 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣[];
31
32 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣{
33 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣name,
34 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣container_name:␣service.container_name␣||␣null,
35 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣image:␣service.image␣||␣null,
36 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣user:␣service.user␣||␣null,
37 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ports:␣service.ports␣||␣[],
38 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣volumes:␣service.volumes␣||␣[],
39 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣command,
40 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣depends_on:␣service.depends_on␣||␣[],
41 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣environment:␣env
42 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣};
43 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣}),
44 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣networks:␣this.networks␣||␣{}
45 ␣␣␣␣␣␣␣␣␣␣};
46 ␣␣␣␣␣␣␣␣␣␣this;
47 ␣␣␣␣' -a file results -j
48 }
```

Algoritmus 4.3: docker compose to json detector

4.5 Záver

Zhrnutie implementácie

Kapitola 5

Výsledky

Výsledky todo

5.1 ttt

Kapitola 6

Inšťalácia

Popis inštalácie

6.1 ttt

Záver

Literatúra

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, Boston, MA, 3rd edition, 2012.

[2] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidović, and Robert Kroeger. Measuring the impact of code dependencies on software architecture recovery techniques. *IEEE Transactions on Software Engineering*, 44(2):159–181, 2017.

[3] Nicolas Anquetil, Anne Etien, Mahugnon H Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatiha Djareddir, Jerôme Sudich, and Moustapha Derras. Modular moose: a new generation of software reverse engineering platform. In *Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19*, pages 119–134. Springer, 2020.

[4] Claudia López, Pablo Inostroza, Luiz Marcio Cysneiros, and Hernán Astudillo. Visualization and comparison of architecture rationale with semantic web technologies. *Journal of Systems and Software*, 82(8):1198–1210, 2009.

[5] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Software landscape and application visualization for system comprehension with explorviz. *Information and software technology*, 87:259–277, 2017.

[6] Ricardo Jorge de Araújo Ferreira. Recovery of software architecture from code repositories. 2022.

[7] Khaled Ibrahim, Hesham Hassan, Khaled T Wassif, and Soha Makady. Context-aware expert for software architecture recovery (caesar): An automated approach for recovering software architectures. *Journal of King Saud University-Computer and Information Sciences*, 35(8):101706, 2023.

[8] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, 2014.

[9] Musengamana Jean de Dieu, Peng Liang, Mojtaba Shahin, Chen Yang, and Zengyang Li. Mining architectural information: A systematic mapping study. *Empirical Software Engineering*, 29(4):1–59, 2024.

Príloha A: obsah elektronickej prílohy

Príloha B: Používateľská príručka