Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Vizualizace dat získaných z úložiště komponent

Plzeň, 2018 Pavel Fidranský

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 16. května 2018

Pavel Fidranský

Abstract

In component-based development, various repositories are used to store components. One of them is Component Repository supporting Compatibility Evaluation (CRCE), a component repository developed at the Department of Computer Science and Engineering at the University of West Bohemia. The goal of this master thesis was to explore meta-data provided by the repository, design a way of their representation and implement it in the form of a web application. The resulting application integrates CRCE with Complex Component Applications Explorer (CoCAEx), a tool which focuses on examination of large component-based applications. Involving CRCE makes it possible for its users to solve potential incompatibilities arising between individual components of an application so that at the end of a work session, all components' dependencies are met.

Abstrakt

V komponentově orientovaném programování jsou pro ukládání komponent využívány různé repozitáře. Jedním z nich je Component Repository supporting Compatibility Evaluation (CRCE), úložiště komponent vyvíjené na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Cílem práce bylo prozkoumat metadata, která úložiště o komponentách poskytuje, navrhnout způsob jejich reprezentace a implementovat jej ve formě webové aplikace. Vytvořená aplikace integruje CRCE s nástrojem Complex Component Applications Explorer (CoCAEx), jehož účelem je průzkum vazeb mezi komponentami rozsáhlých aplikací. Začlenění úložiště CRCE umožňuje uživatelům aplikace řešit případné nekompatibility vzniklé mezi jednotlivými komponentami aplikace tak, že na konci práce jsou všechny jejich závislosti splněny.

Obsah

1 Úvod

Tvorba komplexních systémů na zelené louce a jejich následná údržba je náročným úkolem. V oblasti softwarového inženýrství se tento problém snaží řešit paradigma komponentově orientovaného vývoje software. Jeho základní myšlenkou je rozdělení systému do samostatných funkčních částí, které mohou být opakovaně využívány. Díky tomu by komponenty měly být lépe otestovány, optimalizovány a systém jako celek snadněji udržovatelný. Využití komponent má však také nevýhody, z nichž největší je problematika jejich vzájemné závislosti.

Hotové komponenty mohou být využívány v systémech třetích stran. Pro snadné nalezení komponent vhodných pro zařazení do systému existují různá komponentová úložiště. Jedním z nich je CRCE (Component Repository supporting Compatibility Evaluation), experimentální úložiště komponent vznikající na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Tento nástroj však slouží nejen jako úložiště, o komponentách totiž uchovává různá metadata, která lze využít pro hledání vazeb mezi závislými komponentami.

Cílem této diplomové práce je seznámit se s metadaty poskytovanými úložištěm komponent CRCE, navrhnout způsob jejich vizualizace a ten pak implementovat ve formě webové aplikace. Aplikace přitom vychází z nástroje CoCAEx (Complex Component Applications Explorer), který umožňuje prozkoumávání komplexních komponentových aplikací.

První kapitola představuje komponentově orientovaný způsob vývoje software a jeho principy. Druhá kapitola seznamuje čtenáře s nástroji používanými při dalším zpracovávání tématu práce. Ve třetí kapitole je zpracován návrh reprezentace vazeb komponent a prozkoumány možnosti pro převedení tohoto návrhu do podoby webové aplikace. Čtvrtá kapitola popisuje problémy řešené v rámci malé aplikace, která byla vytvořena jako praktické ověření závěrů předchozí analýzy. Kapitola pátá se věnuje samotné webové aplikaci a konceptům přijatým při její implementaci. Šestá kapitola pojednává o způsobech použitých pro ověření funkčnosti výsledné aplikace. V závěru je zhodnocen výsledek této práce a nastíněny možnosti pro případné rozšíření nástroje.

2 Komponentově orientovaný způsob vývoje software

Komponentově orientovaný způsob vývoje software (CBSE, případně CBD) je způsob vývoje software zaměřený na dekompozici systému do samostatných, opakovaně použitelných celků.

Definice komponenty (někdy také modul) v překladu podle Szyperského [16]:

> *Komponenta je v softwarovém inženýrství jednotkou dekompozice systému se smluvně definovanými rozhraními a výslovně vyjmenovanými závislostmi. Softwarová komponenta může být samostatně nasazena a být využívána třetí stranou.*

Rational Unified Process ve volném překladu definuje komponentu takto [11]:

> *Komponenta je netriviální, takřka samostatná a nahraditelná část systému, která plní jasně danou funkci v kontextu definované systémové architektury. Komponenta se řídí sadou rozhraní, jejichž fyzickou realizaci poskytuje.*

Dá se říci, že komponentou je v softwarovém inženýrství nezávislá, nahraditelná funkční jednotka, která plní jasně danou funkci. Komponenta je určena k propojení s dalšími komponentami, s kterými dohromady vytváří funkční systém. Vnitřní implementace komponenty by ideálně měla být skryta a komponenta by svému okolí měla pouze vystavovat použitelná programová rozhraní. Neexistuje žádný způsob, jak použít funkcionalitu, která v tomto rozhraní není definována. Tomuto principu se říká black-box model. V rozhraní komponenta explicitně definuje, co umí a co pro svoji funkčnost potřebuje.

Komponentově orientovaný způsob vývoje software by měl vést k rychlejšímu a levnějšímu vývoji software. Komponenty mohou být vyvíjeny, využívány, sdíleny a prodávány externím subjektům. Díky opakovanému používání v různých systémech by komponenty měly být také lépe otestovány a optimalizovány než kód postavený na zelené louce. Výhodou je také možnost komponentu eventuálně opravit nebo dokonce vyměnit za běhu aplikace.

Nevýhodou může být nutnost počátečního rozčlenění funkcionality systému do komponent. Při využití komponent třetích stran je to pak především nejistota ohledně budoucího vývoje a správnosti vnitřní funkčnosti komponenty.

Tato práce je zaměřena na komponenty v jazyce Java kvůli jejich přímé podpoře v experimentálním úložišti CRCE. Komponenty samotné společně s jejich vazbami na ostatní komponenty jsou uživateli prezentovány za použití prostředků standardního modelovacího jazyku UML.

2.1 Komponenty v jazyce Java

Jako téměř čistě objektově orientovaný jazyk je Java ze své podstaty komponentově orientovaná. Primární jednotkou dekompozice jsou v Javě třídy, které jsou organizovány do balíků. K distribuci programů používá Java takzvaný Java Archive (formát JAR), jehož součástí mohou být kromě zkompilovaných tříd také obrázky, textové dokumenty a další zdroje programem využívané. V rámci základní platformy Java je JAR (společně se soubory WAR a EAR) jednotkou modularity [10].

2.1.1 Komponentový model

Jako rozšíření tohoto modelu byly do Javy dodatečně doplněny architektonické specifikace pro implementaci komponent, které jsou od svého okolí odděleny jasně daným rozhraním, které definuje požadované a poskytované funkcionality dané komponenty. Tato specifikace se často nazývá komponentový model. V Javě existuje více konkurenčních komponentových modelů, mimo jiné:

- Java 9 Modules (známý jako Project Jigsaw)
- Enterprise JavaBeans (EJB)
- OSGi Service Platform
- SOFA 2

2.1.2 Komponentový framework

Implementací komponentového modelu pak vznikají tzv. komponentové frameworky. Jeden framework může implementovat i více komponentových modelů najednou. Framework je tvořen různými službami, které obvykle zařizují komunikaci a životní cyklus komponent a případné další služby [16].

Vývoj komponenty v rámci frameworku typicky znamená, že komponenta využívá prvky API daného frameworku a poté je nasazena do jeho kontejneru. Komponentovými frameworky, které implementují OSGi kontejner, jsou například:

- Apache Felix
- Knopflerfish
- Equinox

2.1.3 Kontrakt

Součástí rozhraní komponenty je také kontrakt. Mezi dvěma komponentami může být navázáno spojení jen v případě, že jsou splněny požadavky komponenty uvedené v kontraktu. Kontrakt specifikuje, co musí klient udělat, aby mohl využívat rozhraní komponenty. Zároveň se kontrakt může mezi jednotlivými verzemi komponenty měnit [14].

2.1.4 OSGi Service Platform

OSGi je soubor specifikací, které definují dynamický systém komponent na platformě Java. Je vyvíjen konsorciem podniků a spravován OSGi Alliance. Framework OSGi do Javy přidává běhové prostředí, které na platformě umožňuje využití komponentově orientovaného způsobu vývoje software. Na platformu Java přináší plnou modularitu, možnost nasazení více verzí jedné komponenty do jednoho JVM, spouštění, zastavení a výměnu jednotlivých modulů za běhu bez nutnosti restartovat JVM a další výhody.

Komponenta se v OSGi nazývá *bundle*. Její struktura je shodná s běžným JAR souborem, oproti kterému však přidává tzv. Manifest soubor (viz 2.1). Jakýkoliv soubor JAR je validní *bundle* pouze v případě, kdy kromě samotných funkčních prvků obsahuje také Manifest soubor [10].

Manifest obsahuje důležitá metadata o samotných bundlech a hlavně jejich exportovaných a importovaných balících. Na rozdíl od základní Javy, kde je veškerý obsah souboru JAR viditelný jeho uživatelům, obsah bundle je ve výchozím stavu privátní a jeho viditelnost je řízena právě obsahem manifestu. Navíc pokud je v běžném souboru JAR jedna třída přítomna víckrát, použije se jen jedna a ostatní jsou ignorovány. OSGi tomuto problému díky definici importovaných a exportovaných balíků předchází [2].

```
1  Manifest-Version: 1.0
2  Bnd-LastModified: 1333977475461
3  Bundle-Activator: cz.zcu.kiv.osgi.demo.parking.carpark.
       CarParkActivator
4  Bundle-ManifestVersion: 2
5  Bundle-Name: obcc-parking-example.carpark-svc
6  Bundle-SymbolicName: obcc-parking-example.carpark-svc
7  Bundle-Version: 1
8  Created-By: 1.6.0_23 (Sun Microsystems Inc.)
9  Export-Package: cz.zcu.kiv.osgi.demo.parking.carpark.flow;version=1
10 Import-Package: org.osgi.framework,org.slf4j
11 Private-Package: cz.zcu.kiv.osgi.demo.parking.carpark.status.impl
12 Tool: Bnd-1.51.0
13 X-ParkingExample-BundleRev: 1
```

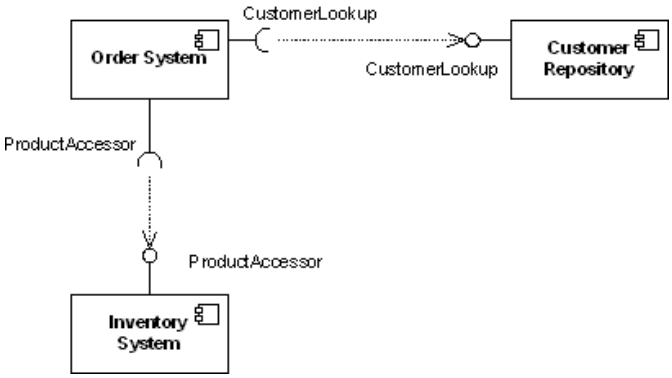
Úryvek kódu 2.1: Příklad OSGi bundle manifest souboru

2.2 Komponenty v UML

Pro modelování softwarových architektur existuje celá řada různých notací, z nichž nejpoužívanější je Unified Modelling Language (UML). Jde o univerzální vizuální modelovací jazyk, který definuje dostatečně obecné prostředky pro vizualizaci komponent v systému a jejich vztahů. Notaci UML zároveň používá aplikace CoCAEx pro vizuální ztvárnění grafu komponent. Pro další potřeby této práce je tak důležité porozumět konceptům používaným v UML.

Graf závislosti komponent je v UML ztvárněn jako klasická grafová struktura, jejímiž prvky jsou uzly a hrany mezi nimi. UML však vizuální ztvárnění obohacuje o některé prvky specifické pro programové komponenty.

Každá komponenta je v UML zastoupena jedním uzlem a vztahy mezi nimi jsou vizualizovány jako orientované hrany mezi komponentami. Z každé komponenty přitom může vycházet libovolný počet hran. Jejich orientace přitom není vyobrazena šipkou na konci hrany jak je v orientovaných grafech obvyklé, ale symbolem obyčejně umístěným uprostřed hrany. UML umožňuje použít libovolnou ikonu pro jakýkoli ustálený vztah [6]. Pro vyjádření vztahu dvou komponent se typicky používají symboly pro svoji podobu nazývané jako „lízátko“ (v angličtině „lollipop“) reprezentující poskytované rozhraní, respektive „mistička“ (anglicky „socket“) pro vyjádření vyžadovaného rozhraní. Uzly i hrany mohou mít v UML textový popis [9]. Nákres reprezentace komponent a jejich vztahů v UML je na obrázku



Obrázek 2.1: Nákres komponent systému a jejich závislostí v UML

2.1, který znázorňuje jednoduchý objednávkový systém, jehož komponenta Order System závisí na komponentách Customer Repository a Inventory System.

Kromě uzlů a hran jako základních prvků pro vizualizaci grafových struktur nabízí UML i další prvky užitečné při návrhu nebo vizualizaci komponentových softwarových systémů. Jsou jimi:

- **port**

  Port slouží jako propojující bod pro vyjádření vztahu mezi komponentou, která je zahrnuta v rámci většího celku (např. balík, jiná komponenta), a komponentou externí.

- **balík**

  Balík je možné kromě agregace běžných programových tříd použít také pro seskupení komponent.

- **poznámka**

  Poznámka je generický textový element, který může být umístěný k jakémukoli prvku diagramu, kde slouží pro jeho dodatečný popis.

2.3 Vývoj nástrojů na ZČU

V oblasti komponentově orientovaného programování probíhá na KIV ZČU poměrně rozsáhlý výzkum, v rámci kterého vznikla celá řada programů, nástrojů a dalších výstupů. S některými z projektů tato práce úzce souvisí, o těch ostatních je příhodné mít alespoň obecné povědomí. Na následujících řádcích jsou projekty stručně představeny.

Components Simplified (CoSi)

Komponentový model, který si klade za cíl používání základních konceptů CBSE při zachování dostatečné funkčnosti. Komponenty CoSi jsou definované jako černá skříňka, jejíž funkce jsou přístupné výhradně přes definovaná rozhraní [5].

OSGi Bundle Compatibility Checking (OBCC)

Projekt zabývající se nahrazováním, automatickým verzováním a verifikací bezpečnosti výměny komponent. V projektu je využívána knihovna JaCC, jako podprojekty pak vznikly nástroje OSGi Bundle Compatibility Checker (OBCC) a OSGi Version Generator.

Component Repository supporting Compatibility Evaluation (CRCE)

Úložiště komponent, mimo jiné také softwarových včetně OSGi a CoSi. Podrobný popis je uveden v kapitole 3.1.

Java Class Compatibility Checker (JaCC)

Projekt zaměřený na statickou analýzu programů v jazyku Java a reprezentaci jeho programových částí opět prostřednictvím jazyku Java. Více informací je k nalezení v kapitole 3.3.

Complex Component Applications Explorer (CoCAEx)

Nástroj pro vizualizaci komplexních komponentových aplikací jako grafové struktury. Aplikace je důkladněji popsána v kapitole 3.2.

Component Application Visualizer (ComAV)

Platforma pro vizualizaci a reverzní inženýrství komponentových aplikací, která poskytuje mechanismus pro další rozšiřování, díky čemuž mohou být snadno přidány další komponentové modely a způsoby vizualizace nezávislé na těchto modelech [15].

3 Představení nástrojů

3.1 CRCE

CRCE (z anglického Component Repository supporting Compatibility Evaluation) je nástroj vyvíjený na KIV ZČU a v základu se jedná o úložiště obecných komponent, které je však rozšířené o část schopnou vyhodnocovat vzájemnou kompatibilitu komponent a jejich rozdíly. CRCE kromě toho nabízí programové rozhraní, díky kterému je možné spouštět evaluaci kompatibility pomocí běžných webových technologií, primárně protokolu HTTP.

Většina nástroje CRCE je napsána v jazyce Java a jako úložiště používá dokumentovou databázi MongoDB. Aplikaci je možné ovládat pomocí dvou rozhraní:

1. web UI
2. REST API

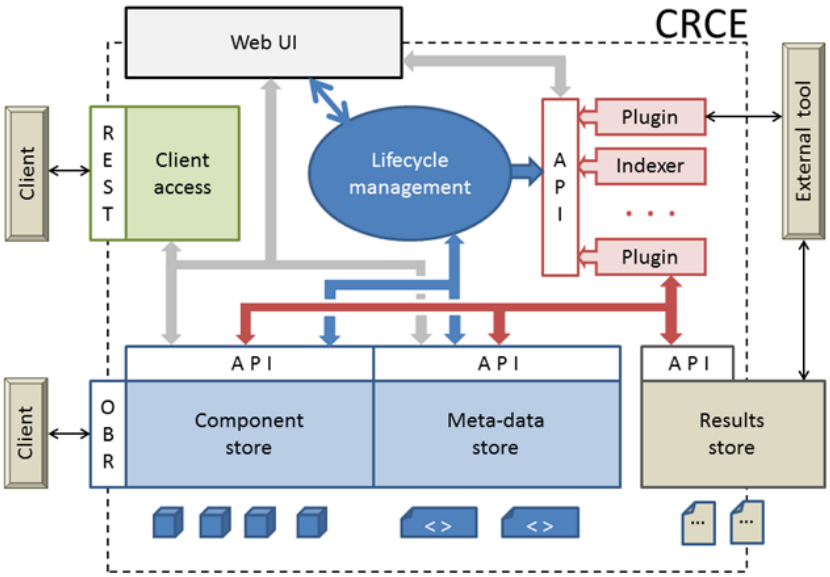
3.1.1 Rozšiřitelnost CRCE

Jak je znázorněno na obrázku 3.1, CRCE je implementováno jako modulární, rozšiřitelná aplikace složená z vzájemně volně provázaných komponent. Hlavními bloky jsou úložiště samotných komponent, úložiště metadat o komponentách a úložiště výsledků. Vnitřní struktury aplikace jsou inspirovány úložištěm OSGi Bundle Repository (OBR), ale byly dále rozšířeny tak, aby bylo možné v nich ukládat i modely obecných komponentových celků [4].

Oproti OBR umožňuje úložiště CRCE navíc provádět výpočetně náročné ověřování kompatibility komponent přímo na straně úložiště. Prostřednictvím doplňků je možné CRCE rozšířit o schopnost zpracovávat libovolné komponenty. Při přidávání a následné indexaci komponent jsou jejich metadata uložena do úložiště a následně využívána při klientsky vyvolaném ověřování kompatibility. To je díky trvalému uložení metadat už méně náročné na výpočetní prostředky.

3.1.2 Web UI

Grafické uživatelské rozhraní aplikace (GUI) je přístupné skrze webový prohlížeč a v rámci této práce slouží především k nahlížení na obsah repozitáře. Každá



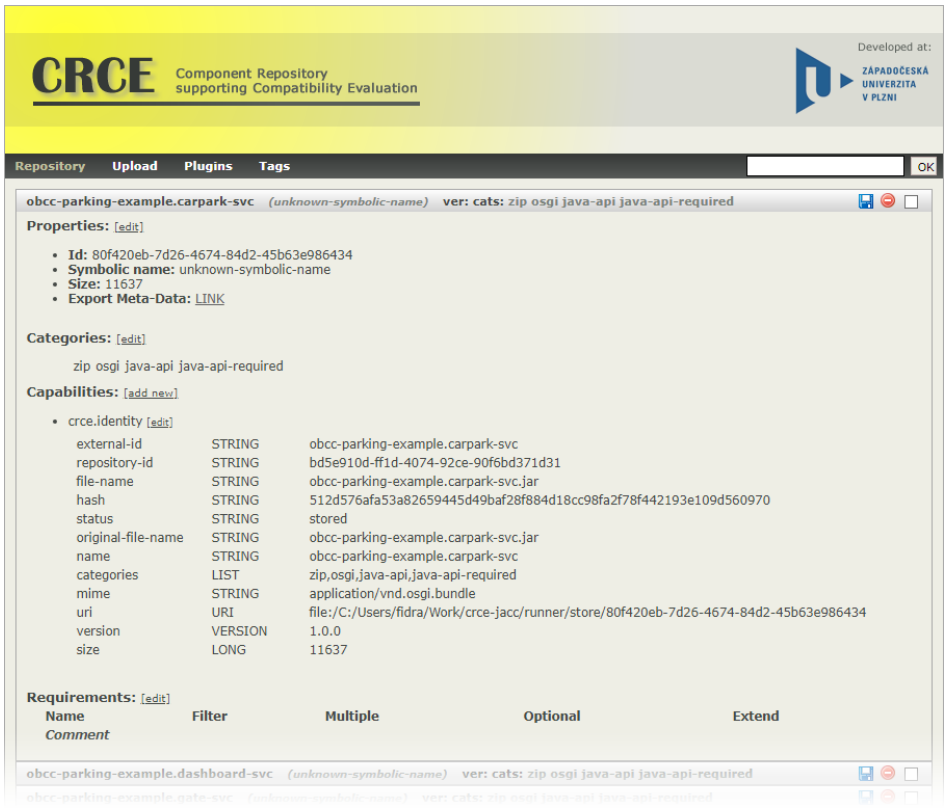
Obrázek 3.1: Architektura nástroje CRCE

komponenta má v rozhraní svůj záznam, který je možné dále otevřít a podívat se na detaily, které CRCE o komponentě uchovává (zachyceno na obrázku 3.2). Těmi jsou mimo jiné následující údaje:

- **`id`** – unikátní identifikátor (UUID) komponenty v rámci CRCE
- **`file-name`** – jméno souboru komponenty
- **`name`** – jméno komponenty
- **`categories`** – kategorie, do kterých komponenta spadá
- **`mime`** – MIME typ souboru komponenty
- **`version`** – verze komponenty ve standardním formátu OSGi [13]
- **`size`** – velikost souboru komponenty v bytech

Kromě toho je možné prostřednictvím GUI nahrávat do úložiště nové komponenty. To je možné buď přímým výběrem souboru uloženého lokálně na počítači, nebo vložením URL, na které je soubor komponenty dostupný. To se může hodit například v případě, kdy je potřebná komponenta uložená ve veřejném Maven repozitáři. Stačí najít vhodnou verzi komponenty a zkopírovat do rozhraní CRCE adresu pro její stažení, čímž lze ušetřit zbytečný mezikrok v podobě stažení souboru na počítač.

Poslední šikovnou funkcí webového rozhraní je zobrazení doplňků aktuálně načtených do CRCE.



Obrázek 3.2: Webové uživatelské rozhraní nástroje CRCE

3.1.3 REST API

Aplikační programové rozhraní (API) obecně slouží ke vzdálenému přístupu k funkcím aplikace pomocí jasně definovaných rozhraní. V prostředí webových nástrojů je vzdálený přístup řešen například pomocí protokolu HTTP.

Nástroj CRCE disponuje rozhraním, které komunikuje s vnějším světem pomocí protokolu HTTP a jeho metod. Jako výměnný datový formát používá CRCE API univerzální XML. Rozhraní je takzvaně RESTful. To znamená, že implementace jeho webových služeb následuje architektonický styl REST. To je univerzální architektonický styl pro síťové aplikace nezávislý na konkrétních protokolech nebo implementaci dílčích komponent [8]. Jako RESTful se pak označují takové webové služby, které jsou postaveny na technologiích HTTP, URI, XML, JSON, Atom aj. Zdroje jsou v RESTful aplikaci identifikovány pomocí URI, akce prováděná na zdroji pak použitou HTTP metodou [1]. Odpovědi na příchozí požadavky navíc obsahují HTTP status kód, jehož hodnota je závislá na tom, zda byl požadavek úspěšně splněn, nebo se během jeho vykonávání vyskytly nějaké chyby.

Pomocí programového rozhraní je možné provádět takřka všechny úkony, které umožňuje uživatelské rozhraní popsané výše. Výjimkou je například nahrávání nových komponent do úložiště z externí URL.

3.2 CoCAEx

Pod zkratkou anglického Complex Component Applications Explorer se skrývá další nástroj vznikající na KIV ZČU. CoCAEx je webová aplikace, která slouží pro vizualizaci strukturovaných dat jako grafu skládajícího se z uzlů a hran. Nástroj se snaží řešit problém s nepřehledností grafů, které obsahují velké množství elementů. Využívá k tomu prostředky návrhu uživatelských rozhraní jako je shlukování uzlů do skupin a odkládání těch uzlů, ze kterých vychází nejvíce hran.

Aplikaci je teoreticky možné uzpůsobit pro její použití jako univerzálního nástroje pro vizualizaci grafových dat. Primárním účelem je však využití pro vizualizaci závislostí a nekompatibilit mezi komponentami programu psaného v jazyce Java. Typickým využitím je reverse-engineering komplexní komponentové aplikace bez předchozí znalosti její vnitřní struktury.

Podobně jako CRCE je i CoCAEx webová aplikace typu klient-server. Serverová část aplikace CoCAEx je vytvořena v jazyce Java a pro sestavování grafu využívá platformu ComAV vyvíjenou taktéž na KIV ZČU. Druhá část aplikace běžící v prohlížeči klientského počítače je napsána v JavaScriptu a slouží především pro vykreslování grafu a obsluhu jeho interakcí.

Kromě hlavního projektu CoCAEx vznikla také jeho odvozená verze. Jejím účelem je zobrazovat místo veškerých závislostí mezi jednotlivými programovými komponentami pouze ty vazby, na kterých vznikají nějaké nekompatibility a odhalení jejich příčin. Komponenty, jejichž vyžadovaná a poskytovaná rozhraní jsou ostatními komponentami splněna, zobrazuje aplikace v odděleném seznamu. Odvozená verze aplikace na serveru přidává knihovnu JaCC, jejíž popis je v kapitole 3.3. V dalším pokračování této práce bude tato verze zmiňována pod názvem CoCAEx-compatibility a dále rozšiřována.

3.3 JaCC

Java Class Compatibility Checker (zkráceně JaCC) je soubor nástrojů a knihoven zaměřených na statickou analýzu kódu v jazyce Java. Celý projekt je vyvíjen na KIV ZČU. Jeho cílem je kompletní reprezentace programových částí (balíků, tříd, metod atd.) jazyka Java v samotné Javě. Dále obsahuje prostředky pro ověření vzájemné kompatibility různých Java knihoven.

V této práci je JaCC používán jako součást obou již zmíněných nástrojů — CRCE a CoCAExu. V obou nástrojích figuruje svým způsobem jako černá skříňka a pro účely jejich integrace není třeba rozumět jeho vnitřnímu fungování. Především pro další rozšiřování projektu CoCAEx-compatibility je však vhodné mít povědomí o doménovém modelu, který JaCC používá pro komparaci a následnou reprezentaci výsledku porovnání kompatibility Java knihoven.

3.3.1 Doménový model

Součástí projektu JaCC je také modul `javatypes-cmp`. Jejím obsahem je množina rozhraní a jejich implementací v jazyce Java sloužících jako model pro reprezentaci jednotlivých částí programu. Ve fázi načtení vytvoří JaCC z prozkoumávané knihovny stromovou strukturu, jejímž kořenem jsou instance `JPackage`, které reprezentují balíky jazyka Java. Z nich jsou dále přístupné kolekce uvnitř definovaných tříd, které představuje rozhraní `JClass`. Pro každou třídu jsou známy jí implementovaná rozhraní, její atributy (`JField`), konstruktory a metody, modifikátory přístupu (`JModifier`) apod. Pro všechny zmíněné prvky jsou dále známy jejich detaily, například pro metody jsou to vstupní parametry včetně jejich typů, vyhazované výjimky, modifikátory a další. Doménový model zkrátka umožňuje uložení všech prvků, které API jazyka Java definuje.

3.3.2 Programové rozhraní pro nalezení nekompatibilit

Po načtení porovnávaných knihoven je z těchto vytvořena jejich reprezentace a následuje fáze porovnání kompatibility a nalezení těch částí, které kompatibilní nejsou.

To má na starosti rozhraní `ApiInterCompatibilityChecker`, které je součástí modulu `compatibility-checker-utils` nástroje JaCC. Jeho jedinou dostupnou implementací je třída `FileApiInterCompatibilityChecker`, jejíž metoda `checkInterCompatibility` akceptuje jako parametr seznam souborů, na kterých

bude porovnávání probíhat. Výsledkem této metody je instance třídy implementující rozhraní `ApiInterCompatibilityResult`. Z této třídy lze pomocí různých metod a mnoha vnořených cyklů získat implementaci rozhraní `CmpResult`. Jak je vidět na výseku kódu 3.1, získaná třída konečně obsahuje kolekci nekompatibilit včetně návrhu strategie pro jejich odstranění.

```
1 CmpResult<File> cmpResult;
2
3 List<CmpResultNode> children = cmpResult.getChildren();
4 CorrectionStrategy strategy = cmpResult.getStrategy();
```

Úryvek kódu 3.1: Získání seznamu nekompatibilit a strategie pro jejich odstranění

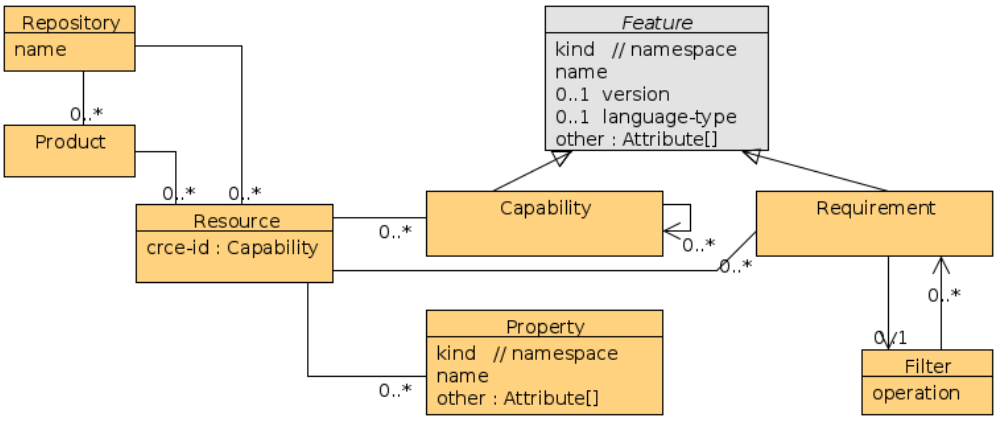
4 Možnosti vizualizace dat z úložiště komponent

Na dalších stránkách se práce zaměřuje na možnosti využití představených nástrojů a jejich schopností pro reprezentaci komponent a jejich vztahů. V první části kapitoly jsou zkoumána metadata poskytovaná úložištěm komponent CRCE se zaměřením na programové komponenty v jazyku Java. Kapitola pokračuje obecným zamyšlením nad vazbami komponent a možnými způsoby jejich reprezentace. Na závěr jsou analyzovány možnosti integrace úložiště CRCE s nástrojem pro prozkoumávání komponentových aplikací CoCAEx.

4.1 Metadata poskytovaná komponentovým úložištěm CRCE

Některá metadata poskytovaná úložištěm CRCE již byla zmíněná v kapitole 3.1.2 o webovém rozhraní. V této části práce je cílem identifikovat v nich ta, která jsou dále využívána pro reprezentaci komponent ve webové aplikaci, a detailně je popsat.

Při popisu metadat, která dokáže úložiště poskytovat je možné vycházet z doménového modelu (viz obrázek 4.1), který data reprezentuje v kódu aplikace. Doménový model nástroje CRCE je velmi obecný a umožňuje uložení různorodých celků složených z nezávislých komponent. Konkrétní funkce je mu vtělena až prostřednictvím jmenných prostor a konkrétních atributů [4].



Obrázek 4.1: Doménový model nástroje CRCE

Namespace

Jmenný prostor ve výpočetní technice obecně slouží k organizaci pojmenovaných prvků do skupin tak, aby byla zaručena unikátnost jejich jmen v dané skupině. Prvkem přitom může být například soubor na disku nebo třída programovacího jazyka. Jmenný prostor tak vytváří kontext, ve kterém může být prvek referencován svým jménem, aniž by hrozilo riziko konfliktu jmen.

V CRCE je využití *namespace* důležité pro dotazy obsahující nějaké požadavky, jmenný prostor totiž umožňuje aplikaci požadavku jen na ty *capabilities* (viz dále), které spadají do stejného jmenného prostoru, díky čemuž nedochází k nejednoznačnostem při jejich interpretaci. Jmenné prostory aktuálně používané v rámci CRCE (potažmo jeho doplňku CRCE-JaCC) relevantní pro tuto práci jsou:

- `crce.identity` - vlastnosti vztahující se k souboru dané komponenty
- `crce.api.java.package` - vlastnosti balíku jazyka Java
- `crce.api.java.class` - vlastnosti třídy
- `crce.api.java.method` - vlastnosti metody
- `crce.api.java.property` - vlastnosti atributu třídy

Jmenné prostory v rámci CRCE slouží k rozlišení metadat specifických pro danou aplikaci úložiště. Společně s generickým datovým modelem jsou to především jmenné prostory, které umožňují další rozšíření nástroje a to buď prostřednictvím nově dostupných metadat, nebo definicí úplně nových jmenných prostor.

Resource

Nejdůležitějším prvkem pro další práci je tzv. *resource*. V terminologii CRCE reprezentuje *resource* jednu komponentu, která byla během vkládání do repozitáře doplněná o další metadata. Každému *resource* je přiřazen jednoznačný identifikátor (UUID). Ten je možné použít pro získání metadat o konkrétní komponentě nebo stažení souboru komponenty z úložiště pomocí API.

Capability

Komponenty mají exaktně definovaná poskytovaná a vyžadovaná rozhraní. V CRCE jsou poskytovaná rozhraní nazvána *capability* a každý *resource* má vždy alespoň jednu – samotný soubor komponenty. *Capabilities* však mohou být i vnořené. Pro

komponenty v Javě to znamená, že je úložiště schopné uchovávat a využívat veškeré informace o prvcích, které API jazyka Java definuje.

Každá *capability* má kromě svého identifikátoru (který ale není nikde využíván) přiřazený také jmenný prostor.

Kromě vnořených *capabilities* jsou dalším prvkem použitelným uvnitř *capability* atributy. Těch může být uvnitř jedné *capability* vloženo libovolné množství [4]. Jejich podrobný popis se nachází v kapitole 4.1.

Requirement

Opakem *capabilities* jsou v CRCE *requirements*. Slouží pro uložení rozhraní, která komponenty vyžadují pro svoji funkčnost. Kromě svého sémantického rozdílu se však jedná o strukturu velmi blízkou *capability*.

Každý *requirement* má svůj identifikátor a jmenný prostor a stejně jako *capabilities* mohou být i *requirements* vnořené. Právě jako *capabilities* pak i *requirements* mohou disponovat dalšími atributy.

Attribute

Doplňkové informace o poskytovaných i vyžadovaných rozhraních skladuje CRCE ve struktuře nazvané *attribute*. Jedná se o obecnou datovou strukturu sestávající z trojice:

- name

  V prvku *name* je schraňováno jméno atributu, které je možné použít pro filtrování seznamu komponent.

- type

  Část *type* slouží pro uložení datového typu, který je použit pro hodnotu atributu. Datový typ je uchováván ve formě *fully qualified name* (FQN) třídy jazyka Java. Ve chvíli zpracovávání diplomové práce podporuje CRCE následující datové typy:

  - `java.lang.String`
  - `java.lang.Boolean`
  - `java.lang.Long`
  - `java.lang.Double`

  - `cz.zcu.kiv.crce.metadata.type.Version`
  - `java.util.List`
  - `java.net.URI`
  - `result.optimize-by`

- value

  Prvek *value* drží samotnou hodnotu atributu. Ta je přitom vždy uložená jako textový řetězec a na definovaný datový typ převedena až tam, kde je to potřeba. Zde je možné identifikovat možný problém při ukládání komplexních datových typů jako je `java.util.List`. V jejich případě je nutná nějaká forma (de)serializace.

4.2 Způsoby reprezentace vazeb komponent

Vztahy mezi dvěma komponentami systému vznikají díky tomu, že komponenty mají definovaná vyžadovaná a poskytovaná rozhraní. Zatímco jedna komponenta může nějaké rozhraní vyžadovat, druhá jej naopak může poskytovat čímž mezi nimi vzniká vazba. Každá komponenta přitom může vyžadovat i poskytovat libovolný počet rozhraní.

Pro reprezentaci komponent a vztahů mezi nimi se perfektně hodí jejich zobrazení jako grafové struktury, kde komponenty jsou vyobrazeny jako uzly grafu zatímco jejich vztahy znázorňují orientované hrany. V případě velkého množství komponent, kde každá komponenta má navíc mnoho (vstupních i výstupních) rozhraní, však nastává problém s nepřehledností grafu. Tento problém se snaží pomocí různých technik vizualizace strukturovaných (zejména grafových) dat řešit nástroj CoCAEx popsaný v kapitole 3.2.

V komponentovém úložišti CRCE jsou komponenty uloženy jako samostatné celky, o kterých je však známa celá řada metadat – například pro programové komponenty jazyka Java ukládá CRCE ve spojení se svým doplňkem pro JaCC veškeré detaily o jejich programových prvcích. Surová metadata stejně jako své dodatečné funkce poskytuje úložiště CRCE nástrojům třetích stran prostřednictvím webového API. Díky tomu je možné implementovat aplikaci, která bude uložená metadata využívat k tomu, aby pomocí CRCE hledala nové vazby mezi komponentami načtenými uživatelem v aplikaci a komponentami uloženými v repozitáři CRCE.

Jako základ uživatelské aplikace lze využít již zmíněný nástroj pro vizualizaci

komponentových aplikací CoCAEx. Do něj je třeba doplnit vrstvu, která bude využívat API úložiště CRCE pro vyhledávání komponent na základě jimi poskytovaných rozhraní. Metadata nalezených komponent budou využita pro zobrazení nových komponent v grafu a vizualizaci jejich vazeb na ostatní komponenty načtené v aplikaci.

4.3 Analýza proveditelnosti integrace

Na úkol integrace jakýchkoli systémů je třeba nahlížet jak z úhlu požadovaného koncového stavu, tak i z hlediska všech systémů, které jsou propojovány. Snahou v této kapitole je zjistit, co bude potřeba doplnit, změnit nebo jakkoli přepracovat v obou slučovaných systémech, aby mohla jejich integrace vůbec započít.

4.3.1 Potřebná rozšíření úložiště CRCE

Pro potřeby vizualizace komponent jazyka Java je potřeba spouštět CRCE společně s jeho doplňkem pro JaCC[1]. Doplněk je třeba zprovoznit ještě předtím, než jsou do úložiště nahrány jakékoliv komponenty. To je nutné proto, že CRCE při nahrávání komponent rovnou analyzuje jejich obsah a strukturu, kterou následně uloží do své databáze a tato data dále používá při zkoumání kompatibility dvou komponent, odpovídání na žádosti o poskytnutí komponent podle požadavků atd.

Pro to, aby CRCE dokázalo najít minimální množinu komponent, která splňuje dané požadavky, je dále nutné, aby byl správně načtený modul lpsolve. Tento doplněk zajišťuje to, že CRCE vrátí vždy pouze ty komponenty, které stačí k tomu, aby byly splněny příchozí požadavky. V opačném případě jsou nalezeny všechny komponenty, které požadavky splňují. To je ale nežádoucí v případě, že konkrétní požadavek splňuje třeba více verzí té stejné komponenty. Pokud je takových nejednoznačných požadavků víc, snadno se může stát, že je jako odpověď nabídnuto množství různých komponent, jejichž nějaká, neznámá kombinace splňuje dané požadavky.

4.3.2 API úložiště CRCE

V kapitole 3.1 věnující se představení úložiště CRCE bylo zmíněno také programové rozhraní, které tento nástroj poskytuje prostřednictvím protokolu HTTP.

[1] `https://github.com/ReliSA/crce-jacc`

Pro další práci je klíčové, aby potřebná metadata byla přístupná z externí webové aplikace což znamená, že je CRCE musí poskytovat v rámci svého API. Následující úsek textu si klade za cíl toto rozhraní prozkoumat a identifikovat části, které budou dále používány při integraci s nástrojem CoCAEx.

Dokumentace veřejného programového rozhraní je k dispozici na serveru Apiary[2]. Tuto službu lze také využít jako statické API, jehož koncové body dokonale odpovídají dané specifikaci. Dokumentace však není zcela kompletní a pro některé detaily je tak lepší zkoumat přímo zdrojový kód[3].

Seznam komponent

Kompletní seznam komponent aktuálně uložených v repozitáři CRCE jde kromě webového grafického rozhraní získat také programově.

Požadavek:

```
GET /metadata
```

Odpověď:

Jeho obsahem je XML dokument, kde uvnitř kořenového tagu `<resources>` je vytvořen pro každou komponentu jeden subdokument uvozený tagem `<resource>` jak je vidět na úryvku kódu 4.1. Přímo na tomto tagu je definován atribut `uuid`, který slouží jako jednoznačný identifikátor komponenty v úložišti CRCE.

Potomkem každé komponenty v CRCE jsou pak její *capabilities*, přičemž každá komponenta má v CRCE alespoň jednu – samotný soubor. V programovém rozhraní CRCE tomu odpovídá subdokument uvozený tagem `<capability>` s definovanými atributy `uuid` a `namespace`. Ve výpisu všech komponent se zobrazuje pouze *capability* první úrovně tzn. soubor komponenty s jeho atributy.

```
1 <resources>
2     <resource uuid="80f420eb-7d26-4674-84d2-45b63e986434">
3     <capability uuid="4e9a095b-9167-403e-86db-70c8d51bd1c1" namespace="crce.
          identity">
4       <attribute name="categories" type="java.lang.String" value="zip,osgi,java-api,
            java-api-required"/>
```

[2]Dokumentace API je ke dni 26. 4. 2018 k dispozici na adrese `https://crceapi.docs.apiary.io`.

[3]`https://github.com/ReliSA/crce`

```
5       <attribute name="external-id" type="java.lang.String" value="
            obcc-parking-example.carpark-svc"/>
6       <attribute name="name" type="java.lang.String" value="obcc-parking-example
            .carpark-svc"/>
7       <attribute name="size" type="java.lang.String" value="11637"/>
8       <attribute name="version" type="java.lang.String" value="1.0.0"/>
9     </capability>
10  </resource>
11  <!-- ... -->
12 </resources>
```

Úryvek kódu 4.1: Seznam komponent uložených v CRCE získaný přes API

Detaily o komponentě

Pro výpis všech detailů, které CRCE o komponentě uchovává, slouží rozhraní na adrese `/metadata/<uuid>`.

Požadavek:

`GET /metadata/<uuid>`

Parametr `uuid` v adrese endpointu odpovídá identifikátoru komponenty, který lze zjistit buď z webového UI aplikace, nebo jejího API, kde mu odpovídá atribut `uuid` tagu `<resource>`.

Odpověď:

Jako odpověď vrací API XML dokument s kořenovým tagem `<resource>`, jehož struktura vychází z formátu stejnojmenného dokumentu použitého pro výpis všech komponent. V tomto případě však kromě `<capability>` popisující soubor komponenty obsahuje také další subdokumenty `<capability>` a `<requirement>`, které slouží jako popis programového rozhraní dané komponenty.

Vyhledávání v katalogu

Nejdůležitější pro tuto práci je endpoint ležící na adrese `/metadata/catalogue`, jež slouží pro vyhledávání komponent(y) v katalogu podle daných požadavků.

Požadavek:

`POST /metadata/catalogue`

Požadavky na vyhledávané komponenty jsou specifikovány v XML dokumentu, který je metodou `POST` odeslán v těle požadavku na server. V kořenovém tagu `<requirements>` mohou být vnořeny tagy `<requirement>` a `<directive>`.

<requirement>

Požadavek na vlastnosti výsledku vyhledávání. Přímo na tomto tagu musí být definovány atributy:

- **uuid** - jakýkoli identifikátor unikátní v rámci dokumentu
- **namespace** - hodnota z množiny podporovaných (detaily jsou popsané v kapitole 4.1)

Uvnitř tagu požadavku může být vložen libovolný počet tagů `<attribute>`, které musí mít definovány atributy `name`, `type` a `value` (detaily v kapitole 4.1).

Požadavky mohou být vnořené a vytvářet tak stromovou strukturu. To je možné využít pro vyhledávání programových komponent podle jejich prvků zanořených ve struktuře programu.

Kromě požadavků na vlastnost komponent je možné zadat požadavek na optimalizaci nalezeného setu komponent. Tomuto účelu je vyhrazen jmenný prostor `result.optimize-by`. Nastavení optimalizační funkce je prováděno pomocí vnořených tagů `<attribute>`, jejichž atributy mohou nabývat hodnot:

- atribut **name:**
  - `function-ID`
  - `method-ID`
  - `mode`
- atribut **type** - vždy `java.type.String`
- atribut **value** - podle hodnoty atributu name:
  - `function-ID` – v závislosti na dostupných doplňcích
  - `method-ID` – v závislosti na dostupných doplňcích
  - `mode` – dvě možnosti: `MIN`, `MAX`

Podporované optimalizační funkce závisí na načtených modulech CRCE a je možné je programově získat pomocí API (popis v kapitole 4.3.2).

<directive>

Přepínač sloužící k přepsání výchozího nastavení vyhledávání. Na jeho tagu musí být definovány atributy `name` a `value`.

V tuto chvíli je podporována jediná direktiva jménem `operator`. Může nabývat hodnot:

- **and**

  Výchozí hodnota. Při jejím použití hledá CRCE jen komponentu, která podporuje všechny požadavky najednou.

- **or**

  CRCE může vyhledat jakékoli komponenty, které splňují alespoň jeden z požadavků. Dohromady však musí splňovat všechny.

Odpověď:
Data obsažená v odpovědi jsou stejná jako v případě rozhraní na adrese `/metadata` (viz kapitola 4.3.2). Pokud úložiště nedokáže najít žádnou komponentu nebo jejich množinu, která splňuje předané požadavky, je kořenový tag `<resources>` prázdný.

Poznámky:

- Jakkoli by dávalo smysl vyhledávat komponenty nejen podle jejich schopností, ale i dle jejich požadavků, CRCE tuto možnost nenabízí. V případě, že mezi dvěma komponentami vznikne nekompatibilita, kterou by bylo možné vyřešit výměnou poskytující komponenty, kvůli tomuto omezení CRCE to prakticky není možné.
- Vyhledávání v katalogu CRCE funguje deterministicky. To znamená, že na stejný dotaz vrátí úložiště vždy totožný výsledek. Skladbu nalezených komponent lze ovlivnit pouze předanými požadavky a zvolenou optimalizační funkcí.

Optimalizační funkce

Množinu komponent nalezených podle požadavků dokáže přímo CRCE optimalizovat. K tomu slouží tag `<requirement>` s definovaným jmenným prostorem `result.optimize-by` (popis v kapitole 4.1).

Požadavek:
Dostupné optimalizační funkce včetně jejich popisu lze zjistit na adrese:

- **function-ID**

  `GET /optimizers/functions`

- **method-ID**

  `GET /optimizers/methods`

Odpověď:

Odpovědí je kolekce s kořenovým tagem `optimizer-functions`, respektive `optimizer-methods`, uvnitř kterého jsou vypsány podporované optimalizační funkce (viz výpis kódu 4.2).

```
1 <optimizer-functions>
2   <optimizer-function id="cf-equal-cost">
3 Optimizes result set by the total amount of components (usually you want to return as
      few components as possible in this scenario.
4   </optimizer-function>
5   <optimizer-function id="cf-file-size">
6 Optimizes result set by the total file size of components (usually you want to return
      the smallest set possible in this scenario).
7   </optimizer-function>
8 </optimizer-functions>
```

Úryvek kódu 4.2: Seznam optimalizačních funkcí CRCE

Stažení komponenty

Soubor komponenty lze z úložiště CRCE také stáhnout na lokální disk. K tomu opět slouží identifikátor `uuid`, který lze získat přes webové GUI i API.

Požadavek:

`GET /resource/<uuid>`

Odpověď:

V těle odpovědi je binární soubor samotné komponenty. V hlavičce `Content-Disposition` je pak obsaženo původní jméno souboru, pod kterým byl soubor komponenty nahrán do úložiště:

`Content-Disposition: attachment; filename="org-obcc...-1.0.4.jar"`

4.3.3 Rozšíření nástroje CoCAEx

Nástroj CoCAEx ve verzi CoCAEx-compatibility bude nutné rozšířit na straně serveru a hlavně straně klientské tak, aby obě reflektovaly požadavky na rozšíření funkcionality. V této kapitole je zmapován současný stav aplikace a možné překážky pro její úpravu.

Kromě toho se následující text zaměřuje na možnosti zlepšení čitelnosti, srozumitelnosti a rozšiřitelnosti kódu. Tyto vlastnosti jsou klíčové pro další rozšiřování aplikace, zvlášť pokud je vyvíjena ve školním prostředí, kde se na vývoji střídá mnoho lidí s různou úrovní znalosti dané technologie.

Serverová část

Část aplikace běžící na serveru má v této chvíli pouze několik zodpovědností. Jedná se především o správu souborů a komunikaci s databází. Důležitou součástí je také hledání nekompatibilit mezi vybranými knihovnami a následné vytvoření uzlů a hran grafu.

Pokud vynecháme části kódu související s vizualizací mimofunkčních požadavků (EFP), která nijak nesouvisí s tématem této práce, skládá se serverová část z několika skupin tříd sloužících k těmto účelům:

1. **Reprezentace grafu, jeho vytvoření a export do formátu JSON**

   Do této kategorie spadají hlavně rozhraní a třídy reprezentující v grafu uzly a hrany. Zásadní je ovšem třída `GraphMaker`, která tyto grafové elementy vytváří a přidává k nim dodatečné informace o kompatibilitě.

2. **Přístup do databáze**

   Prostřednictvím těchto tříd je zprostředkován přístup aplikace do databáze. Aplikace využívá databázi pouze v případě, že je do ní přihlášen uživatel pro ukládání aktuálního stavu zobrazeného grafu.

3. **Operace se soubory**

   Veškeré operace s nahranými soubory komponent obstarává třída `FileManager`, která pro tyto účely používá třídu `FileUtils` z balíku Apache Commons.

4. **Servlety**

   Pro části kódu přístupné přímo prostřednictvím webového prohlížeče jsou používány servlety ve spojení s několika málo JSP. Servlety využívají přímo

nízkoúrovňové Java servlet API. Jejich účelem jsou (kromě zřejmého odpovídání na příchozí HTTP požadavky) ve většině případů pouze operace se soubory/databází.

5. **Podpůrné**

   Aplikace obsahuje několik tříd využívaných pro načtení její konfigurace a definici často používaných funkcí.

Členění kusů aplikačního kódu na dílčí části je tedy dostatečné a neměl by nastat žádný problém s jejím rozšiřováním. Možné vylepšení se týká nedostatečného množství dokumentačních komentářů, případně jejich nulová informační hodnota.

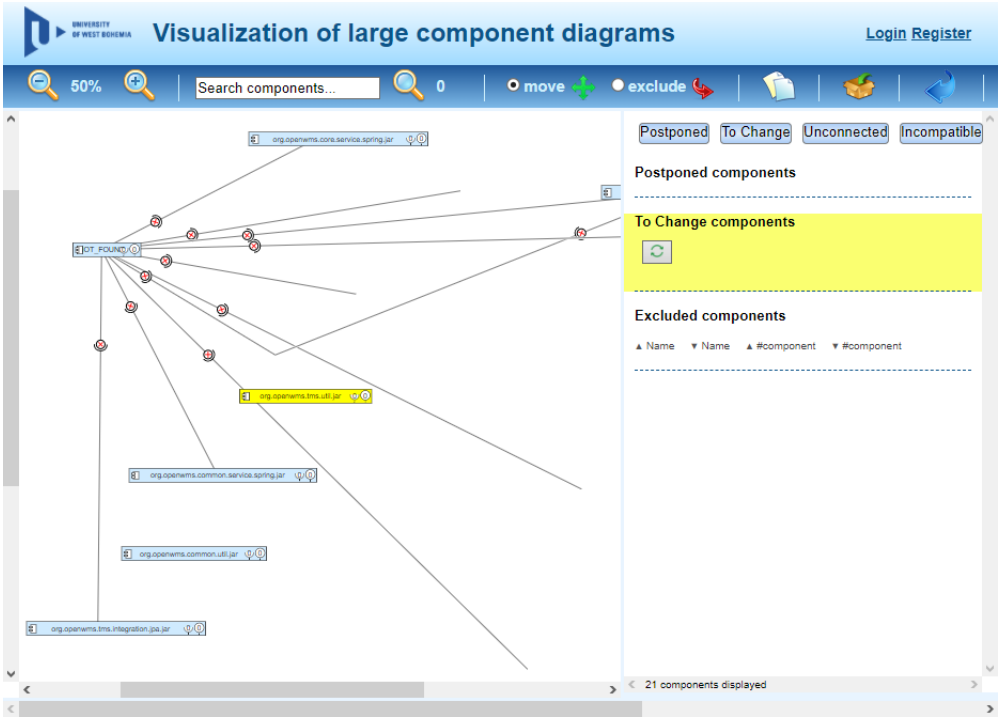
Klientská část

Kód aplikace běžící v prohlížeči má na starosti v první řadě vykreslení grafu a obsluhu uživatelských interakcí. Celá klientská aplikace je psaná v JavaScriptu s využitím dalších běžných webových technologií jako je HTML a CSS. Pro vykreslení grafu a jeho prvků je používáno SVG vložené do běžné HTML stránky.

Vznik původní aplikace CoCAEx se datuje do roku 2012, čemuž odpovídá i způsob, jakým je organizovaná a implementovaná její JavaScriptová část. Pravděpodobně kvůli tehdejší slabé podpoře, ale i nedostatečným schopnostem základního JavaScriptu je v aplikaci použita knihovna jQuery a jako datové úložiště slouží samotný Document Object Model (DOM). To znamená, že data aplikace nejsou dostupná na žádném centrálním místě, ale kód je musí extrahovat z elementů vykreslených na stránce. Kromě toho není zdrojový kód dostatečně dělen na samostatné části a často se v něm dají najít opakující se úseky kódu.

Jak ukázal už zběžný test aplikace v jejím současném stavu, při vyšším počtu komponent zobrazených v grafu se její rozhraní začne chovat nevyzpytatelně. Uzly grafu se při posunu hýbají na jiném místě, než je posunován kurzor myši, hrany původně vedoucí k uzlu raději zůstávají na svém místě, než aby se přesouvaly společně s uzlem. Na obrázku 4.2 je zachycen stav aplikace s načteným demo diagramem short_openwms po několika přesunech uzlů grafu.
Nejedná se přitom o jediné chyby, kterými aplikace trpí. Pro představu o současné funkčnosti aplikace stačí navštívit její běžící instanci[4], nahrát aplikaci skládající se z alespoň dvaceti komponent (případně vybrat jeden z demo diagramů) a několik minut s nástrojem pracovat.

[4]Instance je ke dni 26. 4. 2018 dostupná na adrese `http://relisa-dev.kiv.zcu.cz:8083/cocaex-compatibility/`.



Obrázek 4.2: Chyby interakcí v prototypu CoCAEx-compatibility

Kromě toho je celý kód nedostatečně komentován a zároveň způsob pojmenování proměnných a funkcí ani v nejmenším nenapovídá, k čemu konkrétní kus kódu slouží. Za všechny případy o tom svědčí funkce `fireAwesomePenguin` nebo vlastnost `oscarMike`, které slouží k aplikaci *force-directed* rozvržení, respektive rozlišení mezi kliknutím a tažením uzlu grafu.

Jak se ukázalo už během zpracovávání dílčích úkolů v rámci předmětu ASWI[5], je aplikace velmi těžko rozšiřitelná. Mnohý pokus o její úpravu pak končí rozbitím jiné její části.

4.3.4 Zhodnocení

Pro úspěch integrace úložiště komponent CRCE do aplikace CoCAEx je nutnou podmínkou, aby toto úložiště poskytovalo možnost nalezení komponenty podle definovaných parametrů. K tomu je potřeba, aby úložiště dokázalo poskytovat a zpracovávat metadata, která jsou o komponentě známá. Zároveň tato metadata musí být dostatečně popisná tak, aby se s jejich pomocí daly dohledat správné komponenty.

[5]Projekt probíhající roku 2016 byl spravován na adrese `https://students.kiv.zcu.cz:3443/projects/aswi2016-vizualizace/overview`

V první části analýzy proveditelnosti se povedlo dokázat, že nástroj CRCE disponuje jak potřebnými daty, tak i prostředky pro jejich využití. Kromě toho byly identifikovány potřebné doplňky, díky kterým je možné úložiště využít pro nalezení minimálního setu programových komponent podle definovaných požadavků na poskytované Java API.

Druhým bodem při ověřování možnosti integrace obou nástrojů byl průzkum datového modelu a kódu aplikace CoCAEx. Podmínkou pro další pokračování práce zde byla dostupnost podrobných metadat o kompatibilitě programových komponent jazyka Java, jež nástroj vykresluje. Kapitola se dále zabývala možnostmi rozšíření CoCAExu z hlediska převzatého zdrojového kódu.

Data poskytovaná serverem CoCAEx se ukázala jako použitelná pro jeho propojení s úložištěm CRCE. Pokud přesto bude potřeba nějaká data do výstupu doplnit, díky využití knihovny JaCC to je proveditelné. Prohlídka zdrojového kódu pak především na straně klientské aplikace ukázala značné koncepční nedostatky v jeho strukturování i dostupné dokumentaci. Před dalším doplňováním funkcí tak bude lepší kód aplikace od základu přepsat.

5 Proof of concept

Jako ověření výsledků analýzy proveditelnosti je v této kapitole vypracována malá aplikace, jejíž účelem je dokázat, že se data o kompatibilitě komponent poskytovaná serverem aplikace CoCAEx dají využít pro nalezení odpovídající komponenty v repozitáři CRCE.

PoC aplikace nejdříve vyextrahuje vhodné a potřebné informace z dat o kompatibilitě komponent poskytovaných serverem CoCAEx. Tyto informace pak převede do formátu podporovaného API úložiště CRCE. Jakmile jsou data ve správném formátu, odešle aplikace HTTP požadavek na server CRCE, který následně podle příchozích dat ve své databázi vyhledá komponentu, která splňuje dané požadavky. Výsledek vyhledávání poté vrátí jako odpověď stejného HTTP požadavku, po jejímž obdržení si otěže opět převezme PoC aplikace a výsledek zobrazí v konzoli.

Prostředkem pro implementaci ukázkové aplikace bude JavaScript, pomocí kterého je napsána také klientská strana nástroje CoCAEx. Díky tomu by neměl nastat žádný problém při ostré implementaci navrženého řešení do tohoto nástroje.

5.1 Správnost vyhledávání v repozitáři CRCE

Jako první testovací data jsou použity balíky z repozitáře obcc-parking-example[1], který stejně jako ostatní nástroje zde zmiňované vznikl pod hlavičkou výzkumné skupiny ReliSA KIV ZČU. Pro potřeby PoC aplikace byl vybrán balík `obcc-parking-example.carpark-svc.jar` z první revize této aplikace. Po vložení balíku přes webové rozhraní do repozitáře CRCE je balík viditelný kromě tohoto rozhraní rovněž prostřednictvím API.

Obsah balíku tvoří kromě jiného také rozhraní `IVehicleFlow` definované v balíku `cz.zcu.kiv.osgi.demo.parking.carpark.flow`, jehož kód je vypsán v úryvku 5.1. Jméno balíku i třídy mohou být pro potřeby CRCE API zapsány ve formě XML a odeslány na server.

CRCE by však mělo zvládat i složitější dotazy. Důkazem budiž rozšíření původního dotazu o atribut `interface` definovaný pro třídu `IVehicleFlow`, pomocí kterého

[1] `https://github.com/ReliSA/obcc-parking-example/`

```
1 package cz.zcu.kiv.osgi.demo.parking.carpark.flow;
2
3 public interface IVehicleFlow {
4     void arrive();
5
6     void leave();
7 }
```

Úryvek kódu 5.1: Rozhraní `IVehicleFlow`

by CRCE mělo vyfiltrovat případné třídy, které rozhraními nejsou. Dalším přídavkem je pak specifikace nového požadavku v podobě metody `arrive`, kterou by mělo rozhraní `IVehicleFlow` definovat. Výsledný dotaz na server CRCE je k vidění ve výpisu kódu 5.2.

```
1  <requirements>
2    <requirement uuid="0" namespace="crce.api.java.package">
3      <attribute name="name" type="java.lang.String" value="cz.zcu.kiv.osgi.demo.
           parking.carpark.flow"/>
4      <requirement uuid="1" namespace="crce.api.java.class">
5        <attribute name="name" type="java.lang.String" value="IVehicleFlow"/>
6        <attribute name="interface" type="java.lang.Boolean" value="true"/>
7        <requirement uuid="2" namespace="crce.api.java.method">
8          <attribute name="name" type="java.lang.String" value="arrive"/>
9        </requirement>
10     </requirement>
11   </requirement>
12 </requirements>
```

Úryvek kódu 5.2: Reprezentace rozhraní `IVehicleFlow` jako XML dokumentu

Odpověď API je však prázdná. Přesto, že je balík, ve kterém je rozhraní `IVehicleFlow` definováno, nahrán do úložiště a součástí onoho rozhraní je metoda `arrive`, není tento balík nalezen – jakkoli by podle všeho být měl.

Příčinou tohoto problému se ukázala být třída `RequirementConvertor`, která se v CRCE API jako součást Dozer mapperu stará o mapování objektů hodnot (VO) na odpovídající doménové třídy. Při mapování atributů však třída nebrala v potaz typ atributu a všechny atributy (s výjimkou typu `java.util.List`) tak byly mapovány jako `java.lang.String`. Po opravení tohoto problému byla již odpověď serveru CRCE korektní a úložiště na dotaz vrací tři komponenty. Vidět je to

i ve výpisu kódu 5.3. Oprava chyby byla v podobě *pull requestu*[2] rovněž zaslána do repozitáře CRCE na GitHubu.

```
1  <resources>
2    <resource uuid="406f458e-9b99-4660-b9f0-cffa7f64eb33"
3      <capability uuid="b63d7d0f-0be7-4550-af98-5494ec595975" namespace="crce.
           identity">
4        <attribute name="categories" type="java.lang.String" value="zip,osgi,java-api,
             java-api-required"/>
5        <attribute name="external-id" type="java.lang.String" value="
             obcc-parking-example.carpark"/>
6        <attribute name="name" type="java.lang.String" value="obcc-parking-example
             .carpark"/>
7        <attribute name="size" type="java.lang.String" value="15535"/>
8        <attribute name="version" type="java.lang.String" value="2.0.0"/>
9      </capability>
10   </resource>
11   <!-- ... -->
12 </resources>
```

Úryvek kódu 5.3: Odpověď CRCE serveru po opravě mapování (kráceno)

5.2 Využitelnost dat poskytovaných serverem CoCAEx

Druhým bodem, který byl zkoumán v rámci vytvořené PoC aplikace byla extrakce požadovaných informací z dat poskytovaných serverem CoCAEx a jejich převedení do formátu podporovaného CRCE API. CoCAEx sestavuje data o grafu ve třídě `GraphMaker` a stejná třída se stará také o přidávání informace o nekompatibilitě komponent k hranám, které tyto komponenty v grafu propojují. Aplikace má v této třídě dostupné veškeré informace, které jí poskytuje jako výsledek porovnání kompatibility nástroj JaCC. Jde tedy jen o to, zda a v jaké formě jsou tyto informace předávány do klientské aplikace. V této části PoC tak není cílem zjistit, zda CRCE vrací správné výsledky, ale jestli vůbec existují použitelná data, která mohou být serveru CRCE předložena.

Pro potřeby PoC byla použita testovací data, která jsou pro podobné účely obsažena v repozitáři CoCAEx-compatibility. Pro potřeby mapování je zajímavá především vlastnost hran pojmenovaná `compInfoJSON`. Tato vlastnost obsahuje

[2] `https://github.com/ReliSA/crce/pull/8`

informace o kompatibilitě komponent, které konkrétní hrana propojuje, ve formátu JSON serializovaném do podoby textového řetězce.

```
[{
  "theClass": "lib.specialisedReturnType1.Foo",
  "incomps": [{
    "desc": {
      "level": "0",
      "name": "Methods",
      "isIncompCause": "false"
    },
    "subtree": [{
      "desc": {
        "level": "1",
        "name": "<span class='entity'>M<\/span> List foo ()",
        "isIncompCause": "false"
      },
      "subtree": []
    }]
  }]
}]
```

Úryvek kódu 5.4: Příklad JSONu obsahujícího informace o kompatibilitě

Z náhledu 5.4 je vidět, že se – podobně jako v datovém modelu JaCC – jedná o stromovou strukturu. Nejvyšší úroveň stromu zde tvoří třída, která nekompatibilitu komponent způsobuje. Zajímavější je vlastnost `incomps`, která obsahuje pole samotných nekompatibilit. Každý objekt nekompatibility má jednotnou strukturu, ať jde o třídu nebo třeba datový typ parametru metody. Jsou to tyto dvě vlastnosti:

1. `object` **`desc`**

   Popis daného prvku včetně pravdivostní hodnoty říkající, zda je to zrovna tento prvek, který nekompatibilitu způsobuje. Hodnota je přitom platná pouze pro aktuální prvek, ne tak prvky jemu nadřazené. Objekt dále obsahuje vnořený objekt `details`, který obsahuje detaily o konkrétním prvku v závislosti na jeho typu (třída, metoda, ...).

2. `array` **`subtree`**

   Pole, jehož obsahem jsou prvky, které logicky spadají pod aktuální prvek (např. metoda je součástí třídy). Prvky zde obsažené mají stejnou strukturu jako je tato popisovaná.

Díky využití nástroje JaCC je o všech programových prvcích známa snad každá myslitelná informace. Jediným možným problémem tak je, že tyto informace server CoCAEx nepředává do klientské aplikace. V tom případě však nic nebrání jejich dodatečnému doplnění.

Do dat poskytovaných serverem CoCAEx klientské aplikaci tak byly přidány záznamy o chybějících třídách. Díky tomu bude možné dohledat v úložišti CRCE i komponenty, které při prvotním načtení dat nebyly vůbec dostupné a vyřešit tak možné nekompatibility, které mohly být jejich absencí způsobeny.

5.3 Převod dat do formátu CRCE

Poslední důležitou částí, kterou bylo třeba při zpracovávání PoC aplikace vyřešit, je převod dat načtených z CoCAExu do formátu, který umí zpracovat CRCE API. Vzhledem k tomu, že se na obou stranách jedná o stromovou strukturu, je pro konverzi využitý rekurzivní algoritmus, který postupně prochází uzly stromu a jejich potomky až do chvíle, kdy uzly zpracuje všechny.

Pro vytvoření XML dokumentu v paměti JavaScriptu je možné využít jeho nativní funkce. Jak je vidět v kódu 5.5, skládání XML dokumentu v JavaScriptu probíhá podobně jako v případě HTML.

```
1 var ns = '';
2
3 var xmlDocument = document.implementation.createDocument(ns, '
     requirements', null);
4
5 var directiveEl = xmlDocument.createElementNS(ns, 'directive');
6 directiveEl.setAttribute('name', 'operator');
7 directiveEl.setAttribute('value', 'or');
8
9 xmlDocument.documentElement.appendChild(directiveEl);
```

Úryvek kódu 5.5: Příklad sestavování XML dokumentu v paměti

Jakmile je dokument sestavený v paměti, je pomocí AJAXu odeslán na CRCE server. Pro serializaci a následnou deserializaci odpovědi serveru jsou opět využity nativní třídy JavaScriptu – `XMLSerializer`, respektive `DOMParser`.

Kompletní kód PoC aplikace je k nahlédnutí v příloze A.

5.4 Zhodnocení

V této kapitole byla vytvořena malá aplikace, jejíž účelem bylo na praktickém příkladu ověřit, že data poskytovaná oběma nástroji jsou jimi navzájem využitelná a dostačující a že aplikační rozhraní CRCE vyhledává v repozitáři pouze validní komponenty. Při realizaci této aplikace byla nalezena chyba v nástroji CRCE, kterou se podařilo opravit. Poté již aplikace fungovala korektně a závěry analýzy proveditelnosti o možnostech propojení obou nástrojů se tak podařilo ověřit.

Kód vytvořený v rámci PoC aplikace není pro další práci příliš relevantní, vůbec totiž neřeší zobrazení nalezených komponent ani proces nahrazení komponent v grafu. Použita bude pouze část, která se stará o sestavení XML dokumentu, který je odesílán na server CRCE.

6 Implementace webové aplikace

Následující část práce se věnuje reprezentaci komponent dostupných v úložišti CRCE a jejich vazeb. Způsobem pro jejich zobrazení se stane webová aplikace, jež bude úzce spolupracovat s úložištěm komponent CRCE.

Jako základ webové aplikace bude použit projekt CoCAEx-compatibility, v němž byl implementován prototyp aplikace integrující do CoCAExu úložiště CRCE. Většina jejího kódu (zejména klientská část) bude výrazně přepracována, celá aplikace pak bude rozšířena tak, aby podporovala proces nahrazování komponent za pomoci metadat získaných z úložiště CRCE. Proveditelnost integrace již prokázala kapitola 5, v této části bude nastíněné řešení přeneseno do reálné aplikace.

6.1 Specifikace funkcí

Vytvářená aplikace vychází z nástroje CoCAEx a prototypu CoCAEx-compatibility. Z obou nástrojů přejímá základní funkce pro práci s grafem jako jsou:

- nahrávání a mazání komponent na server
- zobrazení komponent a jejich vztahů jako uzlů a hran mezi nimi
- vytváření skupin uzlů
- zvýrazňování, posun, vyhledávání uzlů a skupin uzlů
- zvětšování a zmenšování celého grafu
- odkládání uzlů a skupin uzlů do bočního panelu
- automatická aplikace *force-directed* rozvržení

Tyto a všechny další funkce popsané v [9] zůstanou zachovány tak, jak jsou. Bude však změněna jejich implementace tak, aby byla zaručena jejich bezchybná funkčnost a budoucí rozšiřitelnost.

Zcela nová bude část aplikace, která načítá data z úložiště CRCE. Komponenty přesunuté do sekundárního panelu bude možné přidávat do seznamu komponent, jejichž nekompatibility budou s pomocí úložiště CRCE řešeny. Tento seznam změn bude možné odkládat na později a opět aktivovat.

Po spuštění procesu výměny komponent bude z dat o nekompatibilitách mezi komponentami sestaven seznam požadavků na novou sadu komponent, který bude následně odeslán do CRCE API. V ideálním případě poskytne úložiště metadata vhodných komponent, která budou použita pro stažení komponent a sestavení nové verze grafu.

Cílem aplikace je iterativně najít množinu komponent, které mezi sebou nemají žádné nekompatibility. Po dokončení práce by v grafu neměl být přítomen uzel `NOT_FOUND` shromažďující chybějící třídy napříč komponentami. Žádná komponenta by neměla být spojena s jakoukoli jinou, to by totiž značilo, že nejsou vzájemně kompatibilní.

6.2 Návrh

První část návrhu je zaměřena na reprezentaci množiny vzájemně závislých komponent jako grafu a implementaci grafu jako interaktivní klientské webové aplikace. Kód by měl být dále rozšiřitelný, což s sebou nese nutné změny kódu vytvořeného v rámci prototypu CoCAEx-compatibility. Sestavení seznamu nutných modifikací je tak dalším cílem této kapitoly.

Druhý oddíl se věnuje plánu na propojení systémů a procesu samotného vyhledání komponent a jejich následnému nahrazení ve webové aplikaci.

6.2.1 Zobrazení grafu a uživatelské rozhraní

Po důkladné prohlídce zdrojového kódu byly identifikovány body, jejichž zavedením by měla být výrazně vylepšena čitelnost a rozšiřitelnost webové aplikace. Patří mezi ně následující:

- doplnění dokumentačních JSDoc[1] a javaDoc komentářů
- zavedení jednotné strategie pojmenovávání tříd, metod a atributů
- členění aplikace do znovupoužitelných služeb a komponent
- uchovávání dat aplikace v paměti místo HTML struktur
- nahrazení externích závislostí vlastním řešením tam, kde je to vhodné
- definice stylů v externím CSS místo HTML atributů

[1] `http://usejsdoc.org`

- přiřazování stylů pomocí CSS tříd místo identifikátorů
- využití standardních funkcí JavaScriptu místo knihovny jQuery

Po důkladném prozkoumání stávajícího kódu a několikerých pokusech o jeho modifikaci do rozšiřitelné podoby za použití zmíněných zásad bylo rozhodnuto, že veškerý kód klientské aplikace bude od základu přepsán. Starý kód bude v procesu reimplementace sloužit jako reference pro úseky kódu, které obsahují aplikační logiku, která funguje korektně. Aplikace bude však přepsána tak, aby každý opakovaně používaný prvek jejího rozhraní byl definován na jednom místě včetně všech svých interakcí.

Původní aplikace CoCAEx je vytvořena jako obecný nástroj pro zobrazování grafových struktur. Nová implementace bude zachovávat obecnou použitelnost, pro další text je však důležité značení: uzel a hrana jsou zde generické prvky grafu zatímco komponenta už je specifikum této práce označující uzel grafu reprezentující komponentu programovacího jazyka.

V zájmu zachování všech zvyklostí a zažitých interakcí s grafem specifikovaných v [9] bude uživatelské rozhraní rozloženo do tří základních bloků (viz obrázek 6.1):

1. **navigační lišta**

   Lišta obsahující tlačítka pro obsluhu funkcí grafu jako je vyhledávání uzlů a přibližování/oddalování grafu. Navigační lišta přitom pouze pracuje s daty zobrazenými v ostatních dvou částech.

2. **viewport**

   Hlavní část rozhraní, ve které jsou vykresleny uzly a hrany grafu. Dalšími prvky, které jsou vykreslovány jako plovoucí nad těmito komponentami jsou:

   - kontextové menu pro přidání uzlu do skupiny
   - okno zobrazující importované a exportované rozhraní komponenty
   - okno zobrazující detaily nekompatibility mezi dvěma komponentami

3. **boční panel (SeCo)**

   Pracovní oblast, do které je možné odkládat prvky grafu a dále s nimi pracovat. Boční panel se dále dělí na několik podskupin:

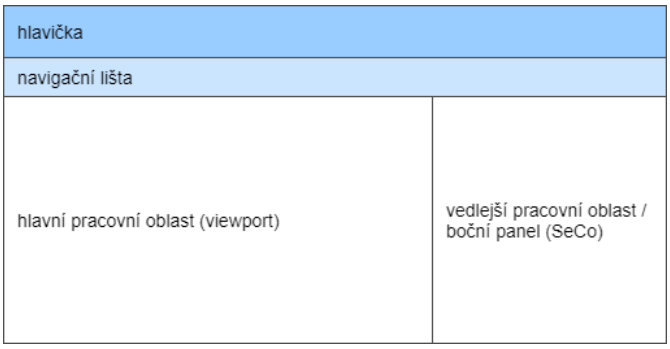
   - odložené komponenty
   - komponenty aktuálně vybrané ke změně
   - odložené změny

- komponenty bez relací
- chybějící třídy
- stavový řádek

Kromě vyjmenovaných komponent, které jsou v aplikaci použity a vykresleny vždy jen jednou jsou součástí aplikace také opakovaně používané prvky:

- uzel
- skupina uzlů
- hrana
- komponenty vybrané ke změně

Kromě jejich opakovaného využití je druhým důležitým rozdílem těchto prvků také to, že mohou být v různých kontextech aplikace vykresleny odlišným způsobem – například uzel může být odložený do bočního panelu, ve kterém je ale zobrazen za použití jiného HTML kódu a podporuje jiné interakce. S tím souvisí také to, že si tyto prvky uchovávají vnitřní stav, který je pak používán při jejich vykreslování.

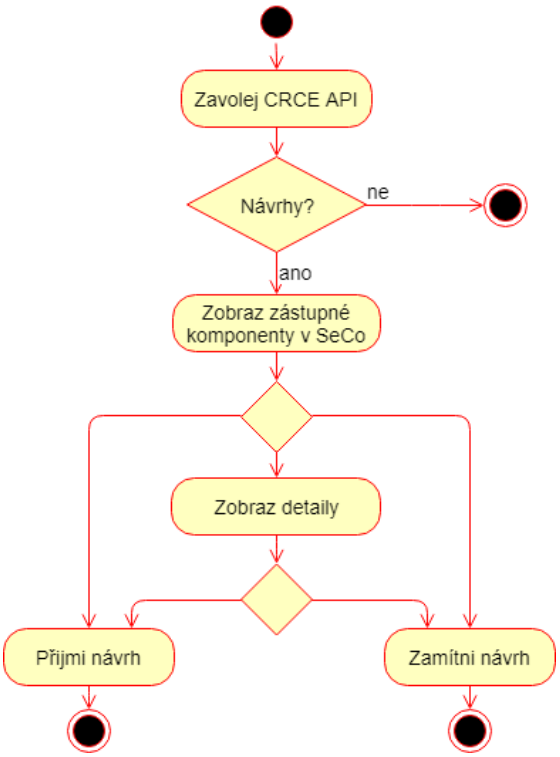


Obrázek 6.1: Rozložení prvků uživatelského rozhraní aplikace na stránce

6.2.2 Proces výměny komponent pomocí CRCE

V uživatelském rozhraní aplikace bude možné prozkoumávat nekompatibility mezi jednotlivými komponentami a vhodné komponenty pak vybírat ke změně. K samotné výměně bude sloužit komponentové úložiště CRCE, které by v ideálním případě mělo nabídnout takové komponenty, aby při jejich použití byla původní nekompatibilita odstraněna.

Protože pro sestavení grafu je v aplikaci používán server CoCAEx, je třeba i výměnu komponent řešit za jeho asistence. Prvotní načtení navrhovaných komponent z CRCE je možné řešit na straně klientské aplikace, jejich samotná výměna a následné vykreslení nové verze grafu už bude probíhat na straně serveru.



Obrázek 6.2: Proces výměny komponenty

Zjednodušené schéma výměny komponent je vyobrazeno na obrázku 6.2. V bodech probíhá proces takto:

1. Po prvotním vybrání komponent ke změně a spuštění procesu výměny je (podobně jako v PoC aplikaci) odeslán na CRCE API požadavek o nalezení komponent splňujících dané požadavky.
2. Pokud CRCE nalezne vhodné komponenty, jsou jejich jména okamžitě zobrazena v bočním panelu. V opačném případě je proces ukončen.
3. V dalším kroku může uživatel navržené změny rovnou přijmout nebo zamítnout. Třetí možností je načtení detailů o navrhované výměně.
4. Pro načtení detailů je již využíván server CoCAEx. Na jeho disk jsou nejprve z úložiště CRCE staženy navržené komponenty, k těm jsou přidány komponenty zobrazené v původním grafu a server následně sestaví novou verzi grafu.
5. Nová verze grafu je pak zobrazena v uživatelském rozhraní. Uživatel může navrženou změnu opět přijmout nebo zamítnout. V rozhraní je pak zobrazena nová (respektive původní) verze grafu, čímž proces výměny končí.

Zde je nutné zmínit, že zatímco CRCE používá pro identifikaci jednotlivých komponent vlastní generované UUID, server CoCAEx toto značení nezná. Pro na-

hrazení komponent je přitom klíčové přiřadit novou komponentu k té, kterou má nahradit. Jediným prostředkem použitelným pro toto přiřazení je jméno souboru komponenty, které CRCE posílá v hlavičce při stahování souboru pomocí jeho UUID.

Nahrazení jedné komponenty druhou je přitom pouze jednou z možností, jak získat množinu kompatibilních komponent. V případě, že nějaká komponenta závisí na rozhraní, které žádná jiná komponenta neposkytuje, je v grafu zobrazen uzel `NOT_FOUND`, který chybějící rozhraní kumuluje. CRCE může nalézt komponentu, která toto rozhraní poskytuje, pravděpodobně však nebude vhodné jí nahradit komponentu původní – spíš bude žádoucí, aby v grafu zůstaly obě. Jestli však bude navržená komponenta nahrazovat tu původní nebo jen přidána do grafu nelze programově rozhodnout a akce je tak volbou uživatele.

6.3 Programová realizace

Nastíněné způsoby reprezentace komponent jako prvků grafu, hotová testovací PoC aplikace, návrh rozdělení aplikace do komponent a proces výměny komponent bude v této kapitole převeden do funkční webové aplikace. Podkapitoly popisují dílčí kroky a rozhodnutí, které vedly k implementaci aplikace zrovna zvoleným způsobem.

6.3.1 Sestavení aplikace

Pro testování webové aplikace v jazyce Java a následné nasazení je nutné mít možnost ji opakovatelně sestavovat do spustitelného WAR souboru. K tomu je v CoCAExu historicky používán nástroj Ant. Jeho *build file* (`build.xml`) byl ale v minulosti pravděpodobně vygenerován v Netbeans IDE a obsahuje závislosti specifické pro toto vývojové prostředí. Jediným způsobem sestavení aplikace tak byla instalace Netbeans. Dílčím úkolem této práce bylo nalezení způsobu, jak aplikaci sestavit nezávisle na používané platformě.

Štěstím v neštěstí je, že ačkoli je *build file* závislý na Netbeans, jsou tyto závislosti definované v externím konfiguračním souboru. Pro sestavení tak stačí pouze nalézt soubory, které jsou pro sestavení nezbytně nutné, ty vytáhnout z Netbeans a prostřednictvím konfiguračního souboru podstrčit Antu.

Sestavení funguje korektně tehdy, když jsou správně nastaveny cesty k souborům:

- `org-netbeans-modules-java-j2seproject-copylibstask.jar`
- `org-netbeans-modules-javawebstart-anttasks.jar`

Nastavení cest je možné provést v souboru, na který odkazuje klíč `user.properties.file` uvnitř konfiguračního souboru `private.properties`. Tyto soubory včetně příkladu konfigurace cest byly přidány do repozitáře aplikace CoCAEx-compatibility.

6.3.2 Třídy v JavaScriptu

Navržený způsob dělení aplikace do samostatných komponent vyžaduje také nalezení vhodného způsobu implementace těchto komponent. Při použití jazyka se silným objektovým modelem si lze snadno představit jejich implementaci jako různě děděných tříd a malých rozhraní. V dynamickém a živě se vyvíjejícím JavaScriptu je to však složitější.

Existuje více způsobů, jak v JavaScriptu definovat privátní atributy/metody. Jednotlivé způsoby se liší především podporou ve webových prohlížečích a svojí čitelností. Ve spojení s požadavky na dědičnost a definici rozhraní se však výčet možností stává výrazně omezenější.

MooTools

Původní aplikace CoCAEx používá pro definici tříd knihovnu MooTools. Toto rozšíření však neumožňuje definici privátních atributů a metod, přidanou hodnotou je tak pouze dědičnost a – svým způsobem – možnost implementace rozhraní.

ECMAScript 6 (ES2015)

Ve specifikaci ES2015 (jejíž je JavaScript implementací) se objevuje možnost definice tříd nativně [7], stejně jako MooTools však nezná způsob, jak označit metodu/atribut jako privátní. Existuje několik cest, jak lze tento nedostatek obejít:

1. **Označení privátních prvků podtržítkem**

   Technika, která do nativních ES2015 tříd přidává čistě notační pravidlo, kdy jména atributů a metod, které by měly být přístupné pouze z třídy samotné, jsou prefixována podtržítkem. Dodržování pravidla není jazykem explicitně

vyžadováno a záleží tak pouze na programátorovi, zda pravidlo následuje. Jednoduchý příklad je uveden ve výseku kódu 6.1.

```
class Node {
  constructor(id, name) {
    this._id = id; // private attribute
    this.name = name; // public attribute
  }

  toString() {
    return `id: ${this._id}, name: ${this.name}`;
  }
}
```

Úryvek kódu 6.1: Označení privátních atributů třídy v JavaScriptu podtržítkem

2. **IIFE a `WeakMap`**

   Třída `WeakMap` umožňuje přiřazení dat k určitému objektu tak, že k datům je možné přistoupit výhradně přes onen objekt. Instance třídy `WeakMap` je přitom viditelná pouze v kontextu daném obalovou IIFE funkcí. Tento postup je k vidění v úryvku kódu 6.2.

```
let Node = (function(){
  let privateId = new WeakMap();

  class Node {
    constructor(id, name) {
      privateId.set(this, id); // private attribute
      this.name = name; // public attribute
    }

    toString() {
      return `id: ${privateId.get(this)}, name: ${this.name}`;
    }
  }

  return Node;
})();
```

Úryvek kódu 6.2: Využití `WeakMap` pro privátní atributy třídy v JavaScriptu

3. **IIFE a `Symbol`**

   Podobným postupem je využití datového typu `Symbol`. V tomto případě jsou data přiřazována přímo do objektu `this`, ovšem jako klíč je použit právě `Symbol`. K hodnotám objektu uloženým přes klíč typu `Symbol` lze přistupovat pouze za použití stejného klíče, který je však viditelný pouze uvnitř obalové IIFE funkce. Díky tomu nejsou atributy přímo přístupné jako vlastnost výsledného objektu. Příklad tohoto postupu je k nahlédnutí v úryvku kódu 6.3.

```
1  let Node = (function(){
2    let idKey = Symbol('id of a node');
3
4    class Node {
5      constructor(id, name) {
6        this[idKey] = id; // private attribute
7        this.name = name; // public attribute
8      }
9
10     toString() {
11       return `id: ${this[idKey]}, name: ${this.name}`;
12     }
13   }
14
15   return Node;
16 })();
```

Úryvek kódu 6.3: Využití `Symbol` pro privátní atributy třídy v JavaScriptu

Způsoby spoléhající IIFE lze přepracovat do podoby ES2015 modulů, jejichž atributy a metody jsou viditelné pouze v rámci kontextu daného modulů (ledaže jsou exportované). Podpora JS modulů v prohlížečích však zatím není dostatečná[2].

Instance funkce s využitím jejího kontextu

Už před příchodem specifikace ES2015 existoval v JavaScriptu způsob, jakým je možné definovat třídy včetně privátních atributů a metod včetně statických. Syntax zápisu tříd přidaná do jazyka v rámci ES2015 je tak pouze syntaktickým

[2]Ke dni 7. 5. 2018 je podpora JS modulů podle serveru Can I use... v České Republice necelých 70%.

vylepšením již existující funkcionality [12], která naopak možnosti jazyka limituje. Výměnou za tato omezení je výrazně lepší čitelnost výsledného kódu.

Technika využívá základní vlastnosti jazyka JavaScript, který vytváří pro každou zanořenou funkci vlastní kontext viditelnosti funkcí a proměnných uvnitř definovaných. Na této vlastnosti jazyka je také postavena funkčnost dříve zmíněných způsobů závislých na IIFE. Kontext funkce je přístupný přes klíčové slovo `this`, které zpřístupňuje objekt, do kterého je možné přiřazovat atributy a metody. Pokud je pak funkce inicializována, vráceným objektem je právě `this` zatímco prvky, které do něj uvnitř funkce přiřazeny nebyly nejsou veřejně přístupné. Jak je naznačeno v kódu 6.4, privátní mohou být nejen atributy, ale také metody.

```
1  function Node(id, name) {
2    var id = id; // private attribute
3    this.name = name; // public attribute
4
5    this.toString = function() { // public method
6      return `id: ${id}, name: ${makeUppercase(this.name)}`;
7    };
8
9    function makeUppercase(text) { // private method
10         return text.toUpperCase();
11   }
12 }
```

Úryvek kódu 6.4: Privátní a veřejné atributy a metody v JavaScriptu

Při použití kontextu funkce `this` je zásadní, aby bylo možné tento kontext při volání nebo referenci metody objektu také nastavit. Pokud by totiž veřejná metoda chtěla zavolat jinou metodu stejného objektu aniž by při jejím spuštění nastavila její kontext, nebylo by možné uvnitř volané metody spoléhat na obsah proměnné `this`.

V JavaScriptu k tomuto účelu slouží tři různé metody definované interně pro všechny funkce (viz 6.5):

1. **`bind`**

   Vytvoří novou funkci, jejíž kontext bude nastaven na první předaný parametr. Ostatní parametry jsou použity jako parametry původní funkce.

2. `call`

   Nastaví kontext funkce na první předaný parametr a tuto funkci rovnou zavolá, přičemž jí předá zbylé parametry.

3. `apply`

   Funguje podobně jako `call`, ale místo samostatných parametrů přijímá jako druhý argument pole.

```
1  function printParameters() {
2    console.log(this, arguments);
3  }
4
5  // bind
6  var tryBind = printParameters.bind('foo', 'bar', 'baz');
7  tryBind();
8  //> String{"foo"} Arguments["bar", "baz"]
9
10 // call
11 printParameters.call('foo', 'bar', 'baz');
12 //> String{"foo"} Arguments["bar", "baz"]
13
14 // apply
15 printParameters.call('foo', ['bar', 'baz']);
16 //> String{"foo"} Arguments[Array(2)]
```

Úryvek kódu 6.5: Zjištění typu proměnné v JavaScriptu

Transpilované jazyky

Pro úplnost je vhodné uvést také úplně jiný přístup pro psaní klientských webových aplikací. Kromě čistého JavaScriptu je možné aplikace psát také pomocí jiných jazyků, které jsou následně do JavaScriptu transpilovány. Jejich výhodou je často statická typová kontrola nebo praktičtější syntax pro objektově orientované programování. Mezi nejvýraznější zástupce transpilovaných jazyků se řadí Dart od společnosti Google a TypeScript vyvíjený Microsoftem.

Použitý přístup

V JavaScriptu existuje mnoho postupů, jak zapsat opakovaně použitelné třídy. Zatím však žádný z nich není ustálený nebo při jeho využití trpí čitelnost výsledného kódu. Použití transpileru by pak do projektu zavedlo notnou dávku přidané komplexity. Třídy tak budou definovány jako běžné funkce se schopností inicializace, což kromě jiného umožňuje bezproblémovou definici privátních atributů a metod.

6.3.3 Typová kontrola

Typová kontrola je v informace proces ověřování správnosti použitých datových typů v různých částech programu. Definicí datových typů a jejich striktní vyžadování umožňuje v programu včasné odhalení případných chyb. Kontrola datových typů probíhá buď staticky při překladu programu, nebo dynamicky za jeho běhu.

JavaScript jako dynamicky typovaný jazyk neumožňuje explicitní definici typu proměnných ani typu parametrů předávaných do funkcí. Jejich typy lze ale dodatečně kontrolovat pomocí podmínky s operátorem `typeof`, výsledkem jehož volání je řetězec udávající jméno typu dané proměnné.

```
1  console.log(typeof 'retezec'); //> "string"
2  console.log(typeof 123); //> "number"
3  console.log(typeof 12.3); //> "number"
4  console.log(typeof true); //> "boolean"
5  console.log(typeof {}); //> "object"
6  console.log(typeof []); //> "object"
7  console.log(typeof Symbol('popisek')); //> "symbol"
8  console.log(typeof null); //> "object"
9  console.log(typeof undefinedVariable); //> "undefined"
10 console.log(typeof new Node); //> "object"
```

Úryvek kódu 6.6: Zjištění typu proměnné v JavaScriptu

Jak je vidět v 6.6, operátor `typeof` nevrací vždy takové výsledky, jaké by člověk zvyklý na jiné programovací jazyky (např. Java, PHP) očekával: řádek 4 by měl udávat `integer` zatímco řádek 5 `float`, výstupem řádku 8 by zase mělo být `array`.

Tyto odlišnosti jsou dány vnitřně podporovanými datovými typy JavaScriptu. Bolestivé je to zejména v posledním případě, který navazuje na zvolený způsob implementace tříd (6.4). Operátor `new` však v JavaScriptu slouží pro vytvoření

nové instance objektu (ať už definovaného uživatelsky nebo systémového), díky čemuž lze jeho typ zkontrolovat pomocí operátoru `instanceof` (viz 6.7).

```
1 console.log((new Node) instanceof Node); //> true
2 console.log((new Node) instanceof Edge); //> false
```

Úryvek kódu 6.7: Ověření typu objektu v JavaScriptu

6.3.4 Vytváření HTML a SVG elementů

Každá dynamická JavaScriptová aplikace potřebuje během svého běhu pracovat s DOM zobrazené stránky. K vytváření nových elementů však existuje mnoho přístupů.

Ve vytvářené aplikaci je důležité mít možnost přidávat novým elementům posluchače událostí uživatelského rozhraní. Tyto funkce je přitom vhodné vázat přímo na vytvářený element, nikoli zprostředkovaně přes jeho identifikátor, protože je tím informace z paměti zanášena duplicitně do DOM.
Graf komponent je vykreslován pomocí SVG, které je jako externí objekt vkládáno do HTML. Zvolená technika musí tedy podporovat kromě HTML také vytváření SVG elementů.

Šablonovací systém

Při tvorbě webových aplikací je vzhled uživatelského rozhraní často oddělován od samostatné vrstvy, která do něj vkládá data. Aby však výsledné rozhraní pružně reagovalo na data, která jsou do něj poslána, používají se různé šablonovací systémy, které umožňují například skrývání obsahu při splnění nějaké podmínky nebo opakované vykreslení stejného rozhraní pro všechny prvky pole.

Tento postup je vhodný především v případě vícestránkových aplikací, které načítají data z externích zdrojů a jejichž vykreslované komponenty nemají mnoho různých interakcí.

JavaScript template literals

Jednodušší formou šablonovacího systému je přímo v JavaScriptu dostupný mechanismus zvaný *template literals*. Jedná se v zásadě o řetězce, které jsou ale uvozeny zpětnými uvozovkami. Uvnitř nich je pak možné používat vložené výrazy,

```
1  var x = 5;
2  var obj = {
3    'foo': 'bar',
4  };
5  var isHidden = true;
6
7  console.log(`x + 10 = ${x + 10}`); //> x + 10 = 15
8  console.log(`Value of foo is: ${obj.foo}`); //> Value of foo is: bar
9  console.log(`Element is: ${isHidden ? 'hidden' : 'visible'}`);
10   //> Element is: hidden
```

Úryvek kódu 6.8: Template literals v JavaScriptu

které jsou uvozovány pomocí sekvence `${}`. Jako výraz zde může být použit libovolný validní JavaScript, což umožňuje uvnitř literálu používat i složitější výrazy (viz 6.8).

Literály řeší výpis řetězců včetně složitějších výrazů, neumožňují však vytváření nových elementů, nemluvě o přidávání nových interakcí.

Web Components

Specifikace WHATWG/W3C Web Components umožňuje pomocí JavaScriptu definovat vlastní, nové HTML elementy. Součástí specifikace je několik technologií, které řeší různé oblasti této problematiky jako je:

- vytvoření třídy obsluhující komponentu v JS
- registrace nové komponenty pro použití v dokumentu
- oddělení stylů komponenty do samostatného kontejneru
- znovupoužitelné komplexní HTML prvky

Komponenty vytvořené tímto způsobem jsou dokonale oddělené od zbytku aplikace, znovupoužitelné a dále rozšiřitelné. Specifikace všech dílčích částí však zatím nejsou dokončeny[3].

[3]Aktuální stav specifikace JS Web Components byl ke dni 7. 5. 2018 k dispozici na adrese `https://developer.mozilla.org/en-US/docs/Web/Web_Components#Specifications`

Nativní JavaScript

Vytváření prvků uživatelského rozhraní je v nativním JavaScriptu možné, přestože ne tak elegantní a všeobjímající jako to jednou umožní Web Components.

Základním kamenem při práci s DOM je v JavaScriptu globální objekt `document`, který reprezentuje celou aktuálně vykreslenou stránku. Pomocí tohoto objektu je také možné vytvářet nové elementy a následně je přidávat do stránky. Objekt přitom umožňuje vytváření nejen nových HTML elementů, ale jakýchkoli XML elementů za předpokladu, že je uveden jejich jmenný prostor. Díky tomu je možné programově vytvářet validní SVG elementy stejným způsobem a se stejnými možnostmi, jako je tomu u HTML elementů (viz kód 6.9).

Elementy takto vytvořené existují pouze v paměti do chvíle, než jsou pomocí metody `appendChild` připojeny k jinému elementu, který už je na stránce vykreslen.

```
1  var label = document.createTextNode('Popisek');
2  var clickCallback = function() {
3    console.log('clicked');
4  };
5
6  // HTML element
7  var htmlElement = document.createElement('div');
8  htmlElement.setAttribute('class', 'node');
9  htmlElement.appendChild(label);
10 htmlElement.addEventListener('click', clickCallback);
11
12 // SVG element
13 var svgNamespace = 'http://www.w3.org/2000/svg';
14 var svgElement = document.createElementNS(svgNamespace, 'rect');
15 svgElement.setAttribute('class', 'node');
16 svgElement.appendChild(label);
17 svgElement.addEventListener('click', clickCallback);
```

Úryvek kódu 6.9: Vytváření elementů v JavaScriptu

Použitý přístup

Aplikace vytvářená v tomto projektu je výrazně interaktivní, kvůli čemuž je nutné registrovat velké množství funkcí obsluhujících události uživatelského rozhraní. Zároveň nezobrazuje velké množství dat, důležitý je spíš způsob jejich vizualizace.

Kvůli tomu není vhodné použití šablonovacího systému. Vznikající specifikace Web Components pokrývá potřeby aplikace dokonale, není však hotová a navíc spoléhá na ES2015 třídy. Pro vytváření elementů tak bude použita kombinace JS template literals společně s API objektu `document`.

6.3.5 Asynchronní volání

V aplikaci je na několika místech použit AJAX jako prostředek pro komunikaci s její serverovou částí. Technika AJAX je už z definice asynchronní což znamená, že po spuštění kódu (v tomto případě požadavku na server) pokračuje běh programu nepřerušeně za tímto blokem aniž by musel být znám výsledek jeho volání. Až ve chvíli, kdy je výsledek vrácen, pokračuje běh programu v původním bloku. V závislosti na výsledku přitom může být zavolána buď funkce pro jeho další zpracování v případě úspěchu, nebo funkce ošetřující chybový stav v případě selhání. V JavaScriptu existují dvě základní cesty pro zpracování asynchronního kódu[4]:

1. **callback funkce**

   V prvním přístupu jsou funkce pro zpracování výsledku předány jako parametr přímo výkonné funkci, která jednu z nich ve správný čas vyvolá. Jak je vidět v programovém výseku 6.10, tento způsob trpí stále hlubším zanořováním kódu v případě, kdy je zřetězeno několik asynchronních volání za sebou.

```
1  function asyncFunction(successCallback, errorCallback) {
2    var result = expensiveOperation();
3
4    if (result) {
5      successCallback();
6    } else {
7      errorCallback();
8    }
9  }
10
11 asyncFunction(function() {
12   asyncFunction(function() {
13     asyncFunction(function() {
14       //...
```

[4] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises`

```
15      });
16    });
17  });
```

Úryvek kódu 6.10: Zanořování bloků při použití JS callbacků

2. **promises**

   Problém se zanořování programových bloků řeší mechanismus zvaný *promise*. Do programu je v tomto případě přidána nová třída `Promise`, která jako svůj parametr přijímá funkci obsahující samotný výkonný kód. Rozdílem však je, že po dokončení asynchronního úkolu zavolá *promise* svoji metodu `then` (případně `catch`), která dále distribuuje běh programu do funkcí, které jsou jí samotné předány jako parametry. Jako výsledek těchto funkcí může být znovu vrácen *promise*, čímž lze jednotlivá volání velmi snadno řetězit jak lze vidět v ukázce kódu 6.11. Výhodou je také možnost globálního zpracování chybového stavu.

```
1  function asyncFunction() {
2    return new Promise(function(resolve, reject) {
3      var result = expensiveOperation();
4
5      if (result) {
6        resolve();
7      } else {
8        reject();
9      }
10   });
11 }
12
13 asyncFunction().then(function() {
14   return asyncFunction();
15 }).then(function() {
16   return asyncFunction();
17 }).then(function() {
18   //...
19 });
```

Úryvek kódu 6.11: Zpracování řetězu asynchronních volání pomocí JS promise

Použitý přístup

Aplikace používá asynchronní přístup pouze v případě odesílání a načítání dat pomocí AJAXu. Veškeré požadavky jsou přitom odbavovány pomocí knihovny jQuery, která pro AJAXová volání obsahuje snadno použitelné rozhraní. Knihovna přitom umožňuje definici funkcí pracujících s výsledkem volání jak pomocí *callbacků*, tak přes vlastní implementaci *promises*. V aplikaci je použit druhý přístup, protože je čitelnější a obecně přijímaný jako lepší způsob zápisu.

6.3.6 Zpracování chyb a výjimek

Stejně jako v dalších programovacích jazycích (např. Java) existuje i v JavaScriptu možnost odchytávání výjimek pomocí bloku `try-catch`. Vyhozená, ale neodchycená výjimka přitom způsobí zastavení aplikace a ona výjimka je vypsána do konzole včetně kompletního *stack trace*. Ve spojení s přidanou typovou kontrolou je tímto způsobem možné předcházet chybám vzniklým nepozorností programátora při ošetřování různých stavů aplikace.

Aplikace může vyhodit výjimku pomocí běžného konstruktu `throw`, výjimka přitom může být jakéhokoli (i primitivního) datového typu. Jazyk samotný přitom definuje například třídu `TypeError`, která je přímo určena jako reakce na nevhodný datový typ parametru funkce. Nic však nebrání vytvoření vlastní třídy, která bude už svým názvem evokovat možný problém.

Další možností, jak může aplikace reportovat chybu ve svém běhu je zamítnutí *promise* (kapitola 6.3.5). Chybu v řetězu *promises* je možné zachytit v jQuery i nativních JS promise dvěma způsoby:

1. funkcí předanou v druhém parametru metody `then`
2. metodou `catch`

Výjimky v aplikaci

V aplikaci je kromě zmíněné třídy `TypeError` dále použita její obecná podoba `Error` a vlastní třída `InvalidArgumentError`. Ta vyjadřuje, že jako parametr byla funkci předána proměnná správného datového typu, jejíž obsah je ale neplatný.

Kromě vstupních parametrů funkcí (respektive metod) je v aplikaci ošetřeno vytváření nových DOM elementů. Jak již bylo zmíněno v kapitole 6.3.4, při vytváření nových SVG elementů je třeba použít funkci akceptující jako druhý parametr

jmenný prostor SVG. V opačném případě bude element sice vytvořen, ale nepůjde jej vykreslit do dokumentu, což může být zdrojem zdlouhavého hledání chyby.

V aplikaci je tak pro vytváření veškerých DOM elementů použita nová třída `DOM`, jejíž metody `createHtmlElement` a `createSvgElement` kontrolují požadované jméno tagu elementu proti seznamu validních HTML (respektive SVG) tagů (viz 6.12). Seznamy ve formátu JSON byly získány z veřejných repozitářů na GitHubu[5] [6].

```
1 this.createHtmlElement = function(tagName, attributes) {
2   if (htmlTags.indexOf(tagName) === -1) {
3     throw new InvalidArgumentError(tagName);
4   }
5
6   //...
7 }
```

Úryvek kódu 6.12: Kontrola názvu tagu při vytváření nového HTML elementu

6.3.7 Komponenty UI

Ve vznikající webové aplikaci jsou komponenty uživatelského rozhraní vytvářeny jako JavaScriptové třídy, které uvnitř své veřejné metody `render` sestavují strom DOM elementů, které danou komponentu reprezentují na stránce.

Vytvořený element je přitom v objektu komponenty uložen a může být později použit pro její překreslení v případě, že se změní vnitřní stav komponenty (viz 6.13). Stejně tak může být vykreslená komponenta ze stránky odstraněna a znovu vykreslena úplně jiným způsobem podle kontextu, v němž se aktuálně nachází.

K jednotlivým elementům komponenty mohou být rovnou přidávány posluchači událostí uživatelského rozhraní, po jejichž aktivaci je zavolána privátní metoda dané komponenty. Její zodpovědností pak může být změna stavu komponenty a následné překreslení v dokumentu.

V aplikaci je vytvořeno množství různých komponent, které jsou vzájemně poskládány do stromové struktury. Jejím kořenovým prvkem je globální proměnná `app`, jejímž prostřednictvím mezi sebou mohou komponenty komunikovat.

[5] `https://github.com/element-io/svg-tags`
[6] `https://github.com/element-io/html-tags`

```
function Node(props) {
  var rootElement;
  var isHighlighted = false;

  this.render = function() {
    rootElement = document.createElement('div');
    rootElement.addEventListener('click', highlight.bind(this));

    return rootElement;
  };

  function highlight() {
    isHighlighted = !isHighlighted;

    rootElement.classList.toggle('node--highlighted');
  }
}
```

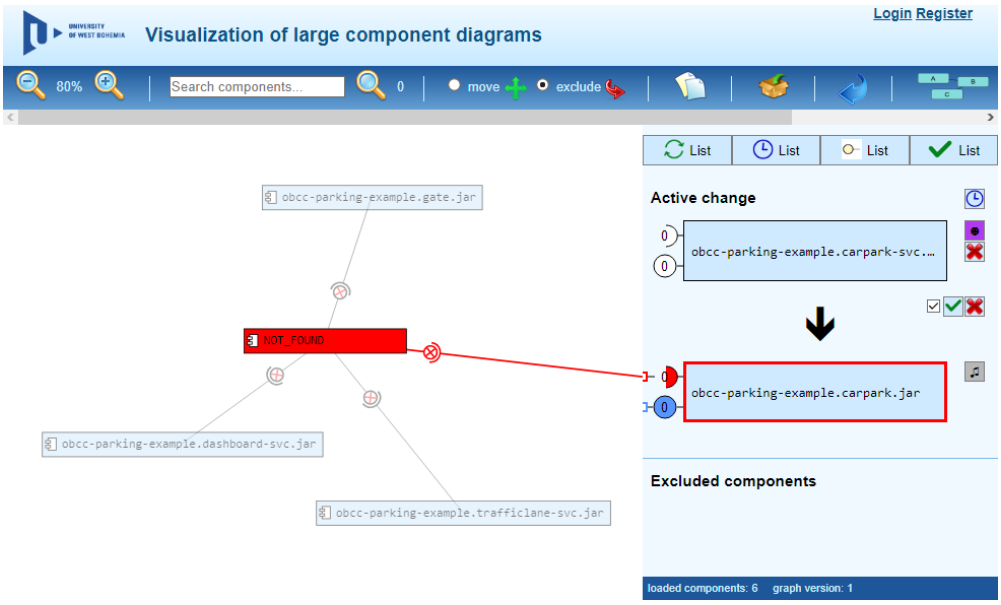
Úryvek kódu 6.13: Stavová komponenta v JavaScriptu

`FloatingPoint`

Jednou z komponent uživatelského rozhraní je třída `FloatingPoint` (plovoucí bod). Jejím účelem je propojení komponent zobrazených ve viewportu s těmi, které jsou odloženy do bočního panelu. To je důležité pro vizualizaci komponent nalezených pomocí CRCE, které jsou nejprve vykresleny právě v bočním panelu, jejich vazby však sahají i ke komponentám uvnitř viewportu. Příklad takové vazby je vidět na obrázku 6.3.

Spojení komponent ve viewportu a bočním panelu hranou přitom není triviální problém – prvky zobrazené ve viewportu je možné přesouvat a celou oblast lze přibližovat a oddalovat, v bočním panelu lze zase skrolovat. Na všechny tyto změny musí `FloatingPoint` reagovat změnou své pozice stejně jako pozice všech navázaných hran.

Plovoucí bod je vnitřně používaný pro uzly i skupiny uzlů. Při jejich odložení do bočního panelu je vytvořen nový `FloatingPoint`, kterému jsou přiřazeny všechny hrany původně spojené s uzlem/skupinou. Ve chvíli, kdy je nutné obnovit pozici plovoucího bodu je vyvolána jeho metoda `setPosition`, která díky znalosti aktuální pozice odloženého uzlu/skupiny, hodnotě zvětšení a dalších od-



Obrázek 6.3: Hrana spojující komponenty v bočním panelu a viewportu

chylek spočte novou pozici plovoucího bodu a na jeho souřadnice přesune příslušné hrany.

6.3.8 Reprezentace metadat CRCE

Metadata komponentového úložiště CRCE jsou v aplikaci využívána prostřednictvím webového API pro hledání vazeb mezi komponentami. Vrstva, která celou komunikaci aplikace s úložištěm zastřešuje by měla být dostatečně obecná, aby ji v případě nového typu metadat bylo možné snadno rozšířit pro jejich zpracování.

Aplikace umožňuje označení komponent, které mají ve vztahu k ostatním komponentám grafu nějaké nekompatibility, k výměně. Pro tyto komponenty pak pomocí CRCE API hledá vazby k jinému souboru komponent, které požadavky vyměňovaných komponent splňují. Nalezené komponenty pak v grafu nahradí původní, nekompatibilní komponenty.

Komponenty označené k výměně jsou v aplikaci shromažďovány uvnitř instance třídy `Change`. Komponentami jsou přitom instance třídy `Vertex`, která v grafu reprezentuje obecný uzel. Informace o (ne)kompatibilitě komponent jsou uchovávány v hranách, které komponenty spojují. Odtud jsou také při spuštění požadavku na výměnu komponent získány.

Po spuštění procesu výměny předá třída `Change` komponenty aktuálně označené k výměně třídě `JavaComponentChanger`. Společně s polem komponent je předána také pravdivostní hodnota udávající, zda si uživatel přeje zahrnout také požadavky

na chybějící třídy (více v závěru kapitoly 6.2.1). Třída `JavaComponentChanger` už je závislá na použitém typu metadat, kterými jsou v tomto případě data o programových prvcích komponent jazyka Java. Při sestavování požadavků odesílaných na server CRCE jsou proto používány jmenné prostory doplňku CRCE-JaCC zmiňované v kapitole 4.1.

Struktura sestaveného XML dokumentu vychází z možností, které externím aplikacím dává API úložiště CRCE. Výsledný dokument se v zásadě podobá tomu, který byl používán v PoC aplikaci (řešeno v kapitole 5), je však výrazně rozsáhlejší jelikož mezi požadavky zahrnuje všechny známé detaily o nekompatibilitě komponent, které aplikaci předává server CoCAEx. V dokumentu je také zahrnut požadavek na optimalizaci výsledné sady komponent pomocí modulu lpsolve (popis v kapitole 4.3.2).

Proces vyhledávání komponent a jejich nahrazování se drží návrhu vytvořeného v kapitole 6.2.2. Jakmile jsou nalezeny vhodné komponenty, zobrazí je aplikace v bočním panelu, kde je možné návrh přijmout, zamítnout nebo načíst další detaily.

Při volbě zobrazení dalších detailů jsou nalezené komponenty staženy na disk serveru CoCAEx, který následně sestaví novou verzi grafu, která je pak zobrazena v aplikaci. Z uživatelského hlediska je důležité, že se při tom nemění pozice už načtených komponent – nové komponenty jsou stále zobrazeny v bočním panelu, ale už je možné zobrazit jejich případné vazby vůči ostatním komponentám (viz obrázek 6.3).

Po přijetí navržené výměny jsou nové komponenty zobrazeny v grafu. V závislosti na volbě uživatele přitom mohou buď nahradit původní komponenty označené k výměně, nebo být do grafu přidány (vysvětlení v kapitole 6.2.1). Verze grafu jsou číslovány, ve chvíli přijetí změny je proto také navýšena verze. Toho využívá server CoCAEx, který komponenty všech verzí grafu ukládá v samostatných složkách. To umožňuje budoucí rozšíření aplikace o možnost vracet se v historii výměn, v této práci je aktuální verze grafu alespoň zobrazena ve stavovém řádku.

Při implementaci procesu výměny komponent byl odhalen problém, kdy server CoCAEx vrací chybná data pro některé abstraktní třídy nebo rozhraní. Pravdivostní hodnoty udávající, zda je třída abstraktní nebo že je rozhraním, občas neodpovídají skutečnosti. Problém pravděpodobně způsobuje knihovna JaCC, jeho důsledkem je pak to, že při vyhledávání v úložišti CRCE nejsou odpovídající komponenty nalezeny. Z toho důvodu jsou v kódu aplikace při sestavování požadavků posílaných na server CRCE tyto informace vynechány, což ale může mít za následek nepřesné výsledky vyhledávání.

6.3.9 Dokumentační komentáře

Pro to, aby byla jakákoli aplikace dále rozšiřitelná je třeba, aby její zdrojový kód byl programátory komentován. Komentáře by měly vývojářům pomáhat při prozkoumávání hotového kódu a jeho úpravách. Každá programovací jazyk je přitom odlišný, kvůli čemuž se liší i způsob zápisu jeho komentářů.

JSDoc

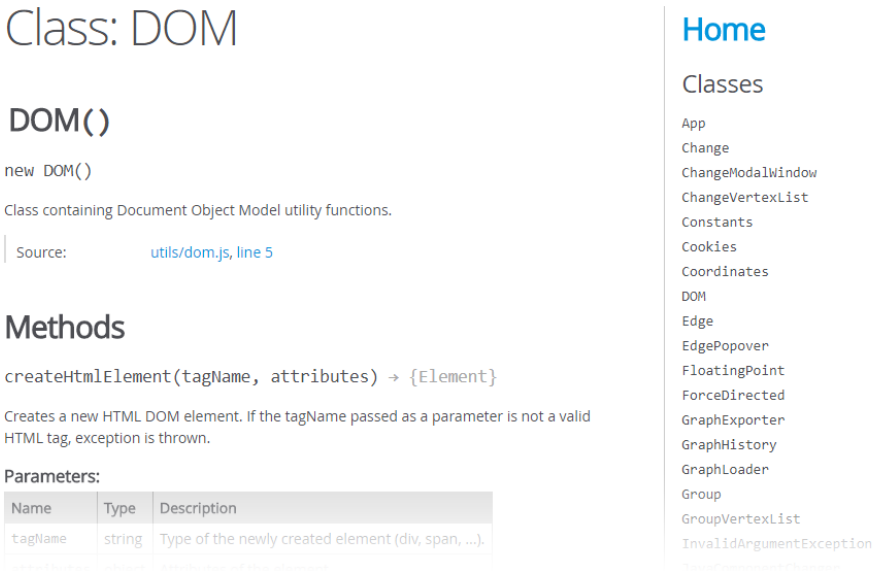
V JavaScriptu je často používán nástroj a způsob zápisu zvaný JSDoc[7]. Podobně jako nástroje používané v jiných programovacích jazycích (např. Javadoc pro Javu, phpDocumentor pro PHP) umožňuje anotaci všech programových prvků dokumentačními komentáři (příklad viz kód 6.14). Kód opatřený dokumentačními komentáři je pak zpracován konzolovým nástrojem, který z něj dokáže vyextrahovat potřebné informace a uložit je ve formě přehledné dokumentace ve formátu HTML.

```
1  /**
2  * Creates a new HTML DOM element. If the tagName is not a valid name
       of HTML tag, exception is thrown.
3  *
4  * @param {string} tagName Type of the newly created element.
5  * @param {object} attributes Attributes of the element.
6  * @returns {Element} HTML DOM element.
7  * @throws {InvalidArgumentException} Thrown when tagName is not a
      valid name of HTML tag.
8  */
9  function createHtmlElement(tagName, attributes) {
10   //...
11 }
```

Úryvek kódu 6.14: Dokumentační komentáře pomocí JSDoc

Kód aplikace vytvářené v této práci používá JSDoc ve všech svých třídách. Výsledná dokumentace ve formě HTML souborů je k nalezení na přiloženém disku, jedna z dokumentačních stránek pak na obrázku 6.4.

[7]Domovská stránka projektu JSDoc je dne 7. 5. 2018 dostupná na `http://usejsdoc.org`.



Obrázek 6.4: Dokumentace třídy `DOM` vytvořená nástrojem JSDoc

6.3.10 Správa závislostí

Programové aplikace většinou nejsou vytvářeny na zelené louce, ale pro dílčí úkoly používají kód vytvořený třetí stranou. Tento externí kód se běžně označuje jako závislost a zvlášť ve velkých aplikacích je vhodné jej spravovat pomocí specializovaných nástrojů. V různých programovacích jazycích se k tomuto účelu používají různé nástroje (např. Maven v Javě, Composer v PHP).

npm

Z mnoha nástrojů pro správu závislostí JavaScriptových aplikací je pravděpodobně nejpoužívanější npm (původně Node Package Manager). Nástroj byl původně využíván pro správu balíčků pro Node.js (na kterém je také postaven), postupně se však rozšířil i do ostatních JS aplikací. Závislosti v npm je možné verzovat a stahovat z veřejných (ale i soukromých) repozitářů. Konkrétní balíčky použité v aplikaci schraňuje npm v souboru `package.json`, který dále slouží k uložení různých informací o projektu (jeho autorech, licencích apod.). Kromě toho npm umožňuje specifikovat v souboru různé úkoly – je tak například možné definovat úkol pro sestavení aplikace nebo spuštění testů.

V této práci je npm použito jak pro správu závislostí, tak i spouštění úloh. V souboru `package.json` jsou definované dva úkoly:

1. **`docs`**

   S pomocí nástroje JSDoc vygeneruje dokumentaci kódu klientské aplikace ve formátu HTML a uloží ji ve složce `docs/`.

2. **`test`**

   Spustí testy uživatelského rozhraní aplikace pomocí Robot Frameworku (popis v kapitole 7.2).

Úkoly je možné spouštět z konzole stejného pracovního adresáře příkazem `npm run <jméno úlohy>`.

7 Ověření funkčnosti

V celém průběhu implementace byla aplikace opakovaně ručně testována. Pro ověření správnosti reakcí uživatelského rozhraní byla aplikace porovnávána s nástrojem CoCAEx v jeho původní podobě, kde interakce alespoň na malých grafech fungují korektně. Pro další vývoj je však toto tento způsob testování nedostatečný, protože je časově náročný a kvůli absenci kontrolního seznamu ověřovaných kroků a testovaných funkcí také náchylný na chyby a opomenutí. Tato kapitola si klade za cíl vytvoření testovacích scénářů, pro které by aplikace měla vždy fungovat totožně, a jejich automatické ověřování tak, aby bylo možné aplikaci dále vyvíjet beze strachu, že provedené změny rozbijí existující kód.

7.1 Testovací scénáře

Vytvořená webová aplikace je výrazně interaktivní a většinu jejího zdrojového kódu tak tvoří obsluha událostí vyvolaných myší a klávesnicí. Část pro obsluhu procesu výměny komponent však plně závisí na použitých vstupních datech a také externí službě v podobě CRCE. Vytvářené testovací scénáře by měly pokrývat obě oblasti.

7.1.1 Testy uživatelských interakcí

První skupina testovacích scénářů se zaměřuje na obsluhu uživatelských interakcí a reakce aplikace na tyto akce. Příkladnou sadou scénářů může být práce s uzlem grafu zobrazeným v hlavní oblasti aplikace:

- při stisknutí a držení tlačítka myši je uzel zbarven žlutě
- po kliknutí:
  - v módu posunu je uzel zvýrazněn červeným rámečkem
  - v módu vyčlenění je uzel přesunut do bočního panelu
- po kliknutí na symbol rozhraní je zobrazeno překryvné okno s detaily o poskytovaných a vyžadovaných rozhraních dané komponenty
- při chytnutí a tažení je uzel včetně svých hran přesunován na nové místo uvnitř grafu

- při vyhledání je uzel zvýrazněn oranžově
- při kliknutí na uzel sousedící se sledovaným uzlem je tento zbarven buď červeně nebo modře podle toho, zda jej jeho soused vyžaduje nebo mu něco poskytuje
- po kliknutí na uzel, který se sledovaným uzlem nesousedí je tento částečně zprůhledněn
- při kliknutí na hranu vedoucí z/do uzlu je tento uzel zvýrazněn červeným rámečkem

Stejný soubor scénářů je vytvořen pro každou část aplikace, jmenovitě:

- prvotní zobrazení grafu
- práce s uzly
- práce s hranami
- práce se skupinami uzlů
- přibližování a oddalování grafu
- vyhledávání uzlů
- vytvoření skupiny z uzlu s nejvyšším počtem hran
- vyčlenění uzlu s nejvyšším počtem hran do bočního panelu
- skrývání a zobrazování boxů v bočním panelu
- odkládání a opětovná aktivace komponent označených k výměně

7.1.2 Testy procesu výměny komponent

Druhou oblastí, na kterou jsou testovací scénáře zaměřeny, je proces výměny komponent. V tomto případě už scénáře závisí i na externí službě, kterou je API úložiště CRCE a především jeho obsah. Správnost funkce úložiště CRCE již byla ověřena v kapitole 5, v této části tak bude ověřována hlavně jeho integrace do vytvořené aplikace.

Už v kapitole 5 byla pro testování používána data z repozitáře obcc-parking-example[1]. Stejná data budou využita i v testovacích scénářích, prvně je však vhodné je představit.

[1] `https://github.com/ReliSA/obcc-parking-example/`

obcc-parking-example

Pro účely demonstrace různých scénářů nahrazování a verzování komponent byla na KIV ZČU vytvořena jednoduchá komponentová aplikace. Aplikace jako taková reprezentuje parkoviště, jeho dopravní značky, příjezdové cesty a vjezdy. Je vytvořena jako soubor OSGi komponent a postupně prošla šesti revizemi. V každé revizi se měnila část kódu – někde přibylo rozhraní, jinde byla přidána závislost na nové komponentě apod. Vývoj aplikace a jejích komponent je zachycen v tabulce 7.1 převzaté z [3]. V průvodním dokumentu je dále popis každé revize aplikace a změn v ní provedených.

komponenta	**1**	**2**	**3**	**4**	**5**	**6**	←revize aplikace
StatsBase	1 1, 2	1 1, 2	1 1, 2	2 2	2 2	2 2	←revize komponenty ←revize fungující v kontextu daném revizí aplikace
Dashboard	1 1, 2	1 1, 2	2 1, 2	2 1, 2	2 1, 2	2 1, 2	
Gate	1 1, 2, 3	2 1, 2, 3	3 1, 2, 3	4 3, 4	4 4	4 4	
CarPark	1	1	1	2	3	3	
TrafficLane			1	2	3	4	
RoadSign					1	1	

Tabulka 7.1: Přehled revizí aplikace obcc-parking-example a jejích komponent

Díky přesné znalosti revizí aplikace, změn provedených mezi jednotlivými verzemi a vzájemné kompatibility komponent v různých verzích je možné aplikaci využít pro nalezení množiny komponent, které mezi sebou zaručeně nejsou kompatibilní. Stejně tak je však možné nalézt způsob, jak jejich nekompatibility spolehlivě vyřešit. Pokud bude mít vytvořená aplikace v úložišti CRCE k dispozici stejné komponenty, mělo by být možné jejím použitím dojít ke stejným výsledkům a nekompatibility odstranit.

Zkušební množina komponent

V tabulce 7.1 je možné nalézt více množin komponent, které mohou sloužit jako testovací scénář. Vybrána byla sada, jejíž komponenty jsou:

- StatsBase revize 1

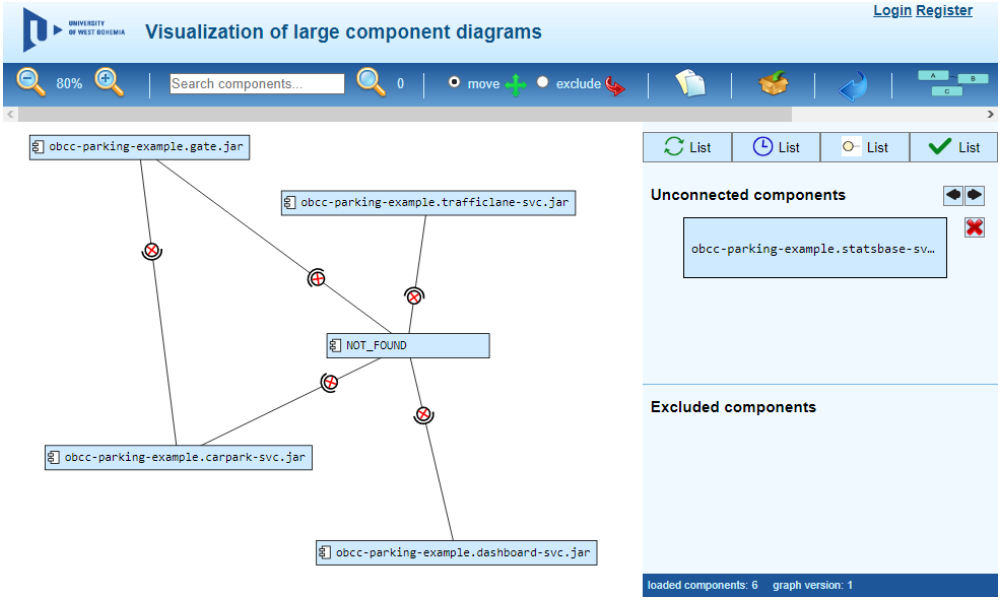
- Dashboard revize 2
- **Gate revize 4**
- CarPark revize 1
- TrafficLane revize 1

Vybrané komponenty náleží do třetí revize celé aplikace s jedinou výjimkou v podobě komponenty Gate. Ta je vybrána ve své čtvrté revizi, jak je však vidět v tabulce 7.1, aplikace ve třetí revizi umožňuje použití komponenty Gate pouze ve verzích 1, 2 nebo 3.

Možných řešení nekompatibilit vzniklých použitím komponenty Gate v nepodporované verzi je více:

1. nahrazení komponenty Gate ve verzi 4 verzí 3
2. použití komponent StatsBase, CarPark a TrafficLane ve verzi 2 místo jejich současných revizí

Z obrázku 7.1 je patrné, že kromě chybějících tříd reprezentovaných uzlem `NOT_FOUND` vznikl problém mezi komponentou Gate a komponentou CarPark, která na ní závisí. Jelikož úložiště CRCE neumožňuje nalezení komponenty podle rozhraní, které poskytuje (více v kapitole 3.1), není možné tento problém vyřešit výměnou komponenty Gate za její starší verzi. Zbývá tak druhý scénář, kdy jsou vyměněny téměř všechny zbylé komponenty, čímž je celá aplikace povýšena na svoji čtvrtou revizi. S tímto scénářem dále pracují automatické funkční testy vytvořené v kapitole 7.2.



Obrázek 7.1: Komponenty obcc-parking-example zobrazené v aplikaci

7.2 Funkční testy

Na kapitolu 7.1, ve které byly specifikovány scénáře sloužící pro ověření funkcí aplikace, navazuje část, jejímž účelem je tyto scénáře přetvořit do podoby automatických testů. Cílem je vytvořit soubor testů, které budou ověřovat správný průběh uživatelských interakcí a funkčnost celé aplikace. Testy je žádoucí spouštět automatizovaně, čímž bude zabráněno chybám vzniklým na straně uživatele a naopak umožněno opakovat stále stejný seznam kroků s totožným průběhem.

Nástrojů, které umožňují psaní automatizovaných testů uživatelského rozhraní existuje nespočet. V této práci byl především kvůli předchozí znalosti a dobré zkušenosti zvolen Robot Framework.

7.2.1 Robot Framework

Robot Framework je obecný testovací nástroj používaný především pro psaní akceptačních testů. Má modulární architekturu a jeho základní funkce jsou dále rozšiřitelné různými knihovnami pro testování širokého spektra aplikací. Framework také umožňuje vytváření vlastních knihoven – ty mohou být psané v Pythonu nebo Javě. Testy Robot Frameworku jsou ukládány do jednoduchých textových souborů, skládány jsou přitom z tzv. klíčových slov, která poskytují rozšiřující knihovny, stejně tak ale mohou být klíčová slova dodatečně definována. To umožňuje vytváření testů, jejichž specifikace je velmi blízká běžnému jazyku.

Jednotlivé testy jsou shlukovány do testovacích sad, které jsou ukládány do souborů s koncovkou `robot`. Testovací sady mohou využívat klíčová slova definovaná v jiných souborech, díky čemuž je možné definovat často používaná klíčová slova pouze na jednom místě. Struktura souboru se skládá z až čtyř bloků:

1. **Settings**

   V sekci nastavení jsou uložena metadata testové sady, importovány používané knihovny, definována klíčová slova spouštěná před/po každém testu atd.

2. **Variables**

   Proměnné používané v rámci testů uvnitř sady.

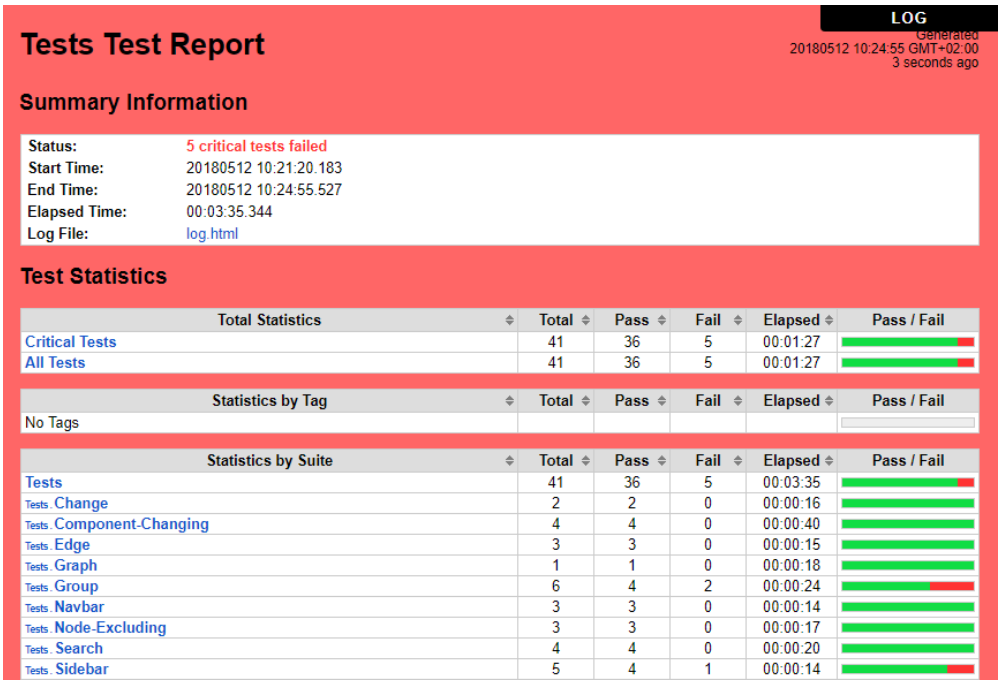
3. **Test Cases**

   Část obsahující samotné testy. Každý test je uvozený svým jménem, za kterým následuje sled klíčových slov, která ověřují konkrétní funkcionalitu.

4. **Keywords**

   V testové sadě mohou být definována také klíčová slova, jež jsou jejími testy využívána.

Robot Framework je spouštěn z konzole a může tak být snadno začleněn do nástrojů pro kontinuální integraci. Při svém běhu vytváří reporty ve formátu HTML a XML, které poskytují detaily informace o průběhu a výsledcích spuštěných testů. Příklad výstupní HTML stránky je zachycen na obrázku 7.2. Pokud nějaký test skončí chybou, vytvoří Robot Framework automaticky snímek obrazovky, který je možné použít pro odhalení příčiny chyby.



Tests Test Report

LOG
Generated
20180512 10:24:55 GMT+02:00
3 seconds ago

Summary Information

Status:	5 critical tests failed
Start Time:	20180512 10:21:20.183
End Time:	20180512 10:24:55.527
Elapsed Time:	00:03:35.344
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	41	36	5	00:01:27	
All Tests	41	36	5	00:01:27	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Tests	41	36	5	00:03:35	
Tests.Change	2	2	0	00:00:16	
Tests.Component-Changing	4	4	0	00:00:40	
Tests.Edge	3	3	0	00:00:15	
Tests.Graph	1	1	0	00:00:18	
Tests.Group	6	4	2	00:00:24	
Tests.Navbar	3	3	0	00:00:14	
Tests.Node-Excluding	3	3	0	00:00:17	
Tests.Search	4	4	0	00:00:20	
Tests.Sidebar	5	4	1	00:00:14	

Obrázek 7.2: Report o průběhu testů vytvořený Robot Frameworkem

Použití

V této práci je framework použit pro testování uživatelského rozhraní, čemuž odpovídají i použité knihovny:

- **Builtin**

  Základní knihovna, která je automatickou součástí všech testovacích sad. Obsahuje klíčová slova, která jsou použitelná ve všech prostředích jako například test rovnosti parametrů nebo cykly.

- **SeleniumLibrary**

  Knihovna, která do Robot Frameworku integruje Selenium jako nástroj pro automatizaci úkonů ve webovém prohlížeči. Klíčová slova získaná jejím použitím typicky simulují interakce prováděné myší a klávesnicí, dále pak kontrolu přítomnosti elementů na stránce, jejich CSS tříd apod.

Knihovna SeleniumLibrary umožňuje spouštění testů za použití různých webových prohlížečů. V této práci je jako výchozí prohlížeč použitý Google Chrome, stejných výsledků však dosahuje i Mozilla Firefox, který je knihovnou také podporovaný.

Vytvořené testy pokrývají základní funkcionalitu aplikace podle testovacích scénářů vytvořených v kapitole 7.1. Při testování procesu výměny komponent jsou tedy opět používány komponenty aplikace obcc-parking-example.

Při psaní testů Robot Frameworku se vyskytly dva problémy. Prvním zádrhelem bylo zaměření SVG elementů přes jméno tagu pomocí jazyka XPath, které z nějakého důvodu nepodporuje syntax, která pro tagy HTML funguje bezchybně. Selektor je však možné napsat alternativním způsob jak je znázorněno v kódu 7.1.

```
1 // non-functional
2 //rect[contains(@class, 'vertex')]
3
4 // functional
5 //*[name()='rect'][contains(@class, 'vertex')]
```

Úryvek kódu 7.1: Alternativní způsob selekce SVG elementu pomocí XPath

Druhá chyba souvisí se způsobem prvotního vykreslení uzlů v grafu. Uzly jsou na stránku umisťovány náhodně což znamená, že někdy mohou být situovány mimo viditelnou oblast grafu. Když se automat na takový uzel snaží kliknout, skončí jeho snaha neúspěchem, protože uzel na stránce není viditelný. Testy v těchto případech mohou končit chybou přesto, že aplikace funguje korektně.

Vytvořené Robot Framework testy jsou uložené v repozitáři aplikace stejně jako na přiloženém disku ve složce `sources/WebContent/tests/`. Ve stejné složce je k dispozici také soubor `README`, který popisuje, jak Robot Framework zprovoznit a následně spouštět testy.

8 Závěr

Cílem práce bylo prozkoumat data poskytovaná úložištěm CRCE, navrhnout způsob jejich vizualizace a ten implementovat ve formě webové aplikace. K dosažení tohoto cíle vedlo několik kroků, které byly v práci zpracovávány.

Začátek práce je věnován seznámení se s problematikou komponentově orientovaného vývoje software. Práce pokračuje představením nástrojů, které v rámci výzkumu na toto téma vznikají na KIV ZČU. V dalším pokračování jsou detailně prozkoumána data poskytovaná komponentovým úložištěm CRCE a navržen způsob jejich vizualizace. V praktické části jsou popsány vybrané detaily implementace webové aplikace, jejíž funkčnost je na závěr ověřena.

Celá práce je výrazně provázána s úložištěm komponent CRCE. Velká pozornost je proto věnována analýze metadat a programových rozhraní, které úložiště poskytuje. Přitom byly opraveny dvě chyby, které znemožňovaly úspěšné vyhledávání komponent.

Jako způsob reprezentace dat z úložiště byla zvolena jejich integrace do nástroje CoCAEx, který metadata využívá pro vyhledání kompatibilních komponent a jejich vizualizaci. Jeho rozšiřování se ale ukázalo jako velmi obtížné, a proto byl celý nástroj víceméně od základu přepsán. Při jeho reimplementaci byl velký důraz kladen zejména na rozšiřitelnost kódu, jeho srozumitelnost, čitelnost a dokumentaci. Pro výslednou aplikaci byl vytvořen také soubor testovacích scénářů a automatizovaných testů její funkčnosti. Aplikaci je také možné sestavit i mimo prostředí Netbeans IDE.

Přestože se při integraci úložiště CRCE s aplikací CoCAEx vyskytlo několik problémů, je možné ji považovat za úspěšnou. Výsledná aplikace umožňuje prozkoumávání komplexních komponentových aplikací, které zobrazuje jako grafovou strukturu. Pro nekompatibility vzniklé mezi komponentami pak nabízí možnost jejich výměny za pomoci úložiště CRCE. Proces výměny využívá metadata úložiště CRCE pro nalezení kompatibilních komponent. Zároveň je dostatečně obecný, aby v případě nového typu metadat nebylo nutné zasahovat hluboko do kódu aplikace. Aplikace umožňuje svým uživatelům iterativně najít takovou množinu komponent, které mezi sebou nemají žádné nekompatibility.

Možná vylepšení vypracované aplikace se týkají zejména procesu výměny komponent. Aplikace aktuálně umožňuje zobrazení grafu pouze v jeho aktuální verzi, její vnitřní implementace však počítá i s možností krokování, díky čemuž by se

uživatel mohl snadno přepínat mezi různými verzemi grafu. Lze si také představit rozsáhlejší integraci úložiště CRCE, jehož metadata by mohla být využita už při prvním načtení grafu.

Přehled zkratek

AJAX Asynchronous JavaScript + XML.

API Application programming interface.

ASWI Pokročilé softwarové inženýrství.

CBD Component Based Development.

CBSE Component Based Software Engineering.

CoCAEx Complex Component Applications Explorer.

ComAV Component Application Visualizer.

CoSi Components Simplified.

CRCE Component Repository supporting Compatibility Evaluation.

CSS Cascading Style Sheets.

DOM Document Object Model.

EFP Extra-functional Property.

EJB Enterprise JavaBeans.

ES2015 ECMAScript 6.

FQN Fully Qualified Name.

GUI Graphical User Interface.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

IIFE Immediatelly Invoked Function Expression.

JaCC Java Class Compatibility Checker.

JS JavaScript.

JSON JavaScript Object Notation.

JSP JavaServer Pages.

JVM Java Virtual Machine.

KIV Katedra informatiky a výpočetní techniky.

MIME Multipurpose Internet Mail Extensions.

OBCC OSGi Bundle Compatibility Checking.

OBR OSGi Bundle Repository.

OSGi Open Service Gateway Initiative.

PHP PHP: Hypertext Preprocessor.

PoC Proof of Concept.

ReliSA Reliable Software Architectures.

REST Representational state transfer.

SeCo Separated Components.

SVG Scalable Vector Graphics.

UI User Interface.

UML Unified Modelling Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

UUID Universally Unique IDentifier.

VO Value Object.

W3C World Wide Web Consortium.

WHATWG Web Hypertext Application Technology Working Group.

XML eXtensible Markup Language.

ZČU Západočeská univerzita.

Literatura

[1] Allamaraju, S. *RESTful web services cookbook*. O'Reilly, 2010. ISBN 978-0-596-80168-7.

[2] Bandara, S. *Java vs OSGi Class Loading* [online]. 2017. Dostupné z: `https://medium.com/@technospace/java-vs-osgi-class-loading-17a1ad4cc2a5`.

[3] Brada, P. *Contextual Substitutability "Car Park" Demo Application* [online]. 2012. Dostupné z: `https://github.com/ReliSA/obcc-parking-example/blob/master/doc/obcc-parking-example.pdf`.

[4] Brada, P. – Jezek, K. Repository and meta-data design for efficient component consistency verification. *Science of Computer Programming*. 2015. ISSN 0167-6423.

[5] Brada, P. – Wajtr, B. – Liška, V. The CoSi component model: technical report no. DCSE/TR-2008-07. Technical report, University of West Bohemia in Pilsen, 2008.

[6] Cheesman, J. *UML Components: a simple process for specifying component-based software*. Addison-Wesley, 2001. ISBN 0-201-70851-5.

[7] Ecma International. ECMAScript® 2015 Language Specification, 2015. Dostupné z: `https://www.ecma-international.org/ecma-262/6.0/`.

[8] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000. Dostupné z: `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`.

[9] Holý, I. L. *Vizualizace rozsáhlých diagramů komponent*. PhD thesis, Zapadočeská univerzita v Plzni, 2014. Dostupné z: `http://hdl.handle.net/11025/23713`.

[10] Knoernschild, K. *Java application architecture: modularity patterns with examples using OSGi*. Prentice Hall, 2012. ISBN 978-0-321-24713-1.

[11] Kruchten, P. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edition, 2004. ISBN 0321197704.

[12] MDN Web Docs. *Classes - JavaScript* [online]. 2018. Dostupné z: `https://developer.mozilla.org/cs/docs/Web/JavaScript/Reference/Classes`.

[13] OSGi Alliance. Semantic Versioning. Technical report, OSGi Alliance, 2010. Dostupné z: `https://www.osgi.org/wp-content/uploads/SemanticVersioning.pdf`.

[14] ReliSA. Úvod do komponent, 2015. Dostupné z: `http://wiki.kiv.zcu.cz/UvodDoKomponent/HomePage`.

[15] Snajberk, J. – Brada, P. ComAV - a component application visualization tools. Technical report, University of West Bohemia, 2012.

[16] Szyperski, C. *Component software: beyond object-oriented programming.* Addison-Wesley, 2nd edition, 2002. ISBN 0-321-75302-X.

Přílohy

A – kód PoC aplikace

```
var API_BASE = 'http://localhost:8081/rest/v2';

var javaClasses = {
       boolean: 'java.lang.Boolean',
       string: 'java.lang.String',
       list: 'java.util.List',
       set: 'java.util.Set',
};

var crceClasses = {
       package: 'crce.api.java.package',
       class: 'crce.api.java.class',
       method: 'crce.api.java.method',
       property: 'crce.api.java.property',
};

var ns = '';

var xmlDocument;
var nodeCounter;

var xmlParser = new DOMParser();
var xmlSerializer = new XMLSerializer();

var sampleEdge = {
       "compInfoJSON": "[]", // omitted for brevity
       "from": "vertex_obcc-parking-example.gate.jar",
       "id": 1,
       "isCompatible": false,
       "packageConnections": [],
       "to": "vertex_obcc-parking-example.carpark-svc.jar"
};
var compatibilityInfoList = JSON.parse(sampleEdge.compInfoJSON);

// initialize requirements XML to be sent to CRCE
xmlDocument = document.implementation.createDocument(ns, 'requirements', null);

constructXmlDocument(compatibilityInfoList);

console.log('CRCE␣request:', xmlDocument);

// send request to CRCE API
postAjax(API_BASE + '/metadata/catalogue', xmlSerializer.serializeToString(xmlDocument),
     function(data) {
       console.log('CRCE␣response:', xmlParser.parseFromString(data, 'text/xml'));
});

function postAjax(url, data, success) {
   var params = typeof data == 'string' ? data : Object.keys(data).map(
```

```
49              function(k){ return encodeURIComponent(k) + '=' + encodeURIComponent(data[k]) }
50          ).join('&');
51
52      var xhr = window.XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject("Microsoft.
            XMLHTTP");
53      xhr.onreadystatechange = function() {
54          if (xhr.readyState>3 && xhr.status==200) { success(xhr.responseText); }
55      };
56      xhr.open('POST', url);
57      xhr.setRequestHeader('Content-Type', 'application/xml');
58      xhr.send(params);
59      return xhr;
60  }
61
62  function constructXmlDocument(compatibilityInfoList) {
63          nodeCounter = 0;
64
65          // optimize returned results
66          var optimizeByRequirementEl = xmlDocument.createElementNS(ns, 'requirement');
67          optimizeByRequirementEl.setAttribute('namespace', 'result.optimize-by');
68
69          var optimizeByFunctionAttributeEl = xmlDocument.createElementNS(ns, 'attribute');
70          optimizeByFunctionAttributeEl.setAttribute('name', 'function-ID');
71          optimizeByFunctionAttributeEl.setAttribute('type', javaClasses.string);
72          optimizeByFunctionAttributeEl.setAttribute('value', 'cf-equal-cost');
73
74          var optimizeByMethodAttributeEl = xmlDocument.createElementNS(ns, 'attribute');
75          optimizeByMethodAttributeEl.setAttribute('name', 'method-ID');
76          optimizeByMethodAttributeEl.setAttribute('type', javaClasses.string);
77          optimizeByMethodAttributeEl.setAttribute('value', 'ro-ilp-direct-dependencies');
78
79          optimizeByRequirementEl.appendChild(optimizeByFunctionAttributeEl);
80          optimizeByRequirementEl.appendChild(optimizeByMethodAttributeEl);
81
82          xmlDocument.documentElement.appendChild(optimizeByRequirementEl);
83
84          // component does not have to fulfill all requirements
85          var directiveEl = xmlDocument.createElementNS(ns, 'directive');
86          directiveEl.setAttribute('name', 'operator');
87          directiveEl.setAttribute('value', 'or');
88
89          xmlDocument.documentElement.appendChild(directiveEl);
90
91          // construct functionality requirements tree
92          compatibilityInfoList.forEach(function(compatibilityInfo) {
93                  compatibilityInfo.incomps.forEach(function(incompatibility) {
94                          appendRequirementTree(xmlDocument.documentElement, incompatibility);
95                  });
96          });
97  }
98
99  function appendRequirementTree(element, incompatibility) {
100         var type = incompatibility.desc.type;
101
102         if (typeof type === 'undefined') {
103                 incompatibility.subtree.forEach(function(incompatibility) {
104                         appendRequirementTree(element, incompatibility);
105                 });
106
107         } else {
```

```
108             // add package for classes
109             if (type === 'class') {
110                     var packageRequirementEl = xmlDocument.createElementNS(ns, 'requirement'
                            );
111                     packageRequirementEl.setAttribute('uuid', nodeCounter++);
112                     packageRequirementEl.setAttribute('namespace', crceClasses['package']);
113
114                     var packageAttributeEl = xmlDocument.createElementNS(ns, 'attribute');
115                     packageAttributeEl.setAttribute('name', 'name');
116                     packageAttributeEl.setAttribute('type', javaClasses.string);
117                     packageAttributeEl.setAttribute('value', incompatibility.desc.details.
                            package);
118
119                     packageRequirementEl.appendChild(packageAttributeEl);
120             }
121
122             var requirementEl = xmlDocument.createElementNS(ns, 'requirement');
123             requirementEl.setAttribute('uuid', nodeCounter++);
124             requirementEl.setAttribute('namespace', crceClasses[type]);
125
126             // attributes
127             var details = incompatibility.desc.details;
128             for (var key in details) {
129                     if (!details.hasOwnProperty(key)) continue;
130                     if (key === 'package') continue;
131
132                     var value = details[key];
133
134                     // fix the incorrectly false value
135                     if (key === 'interface' || key === 'abstract') {
136                             value = true;
137                     }
138
139                     var attributeType;
140                     switch (typeof value) {
141                             case 'boolean':
142                                     attributeType = javaClasses.boolean;
143                                     break;
144                             case 'object':
145                                     if (value.constructor.name === 'Array') {
146                                             attributeType = javaClasses.list;
147                                             break;
148                                     }
149                             default:
150                                     attributeType = javaClasses.string;
151                     }
152
153                     var attributeEl = xmlDocument.createElementNS(ns, 'attribute');
154                     attributeEl.setAttribute('name', key);
155                     attributeEl.setAttribute('type', attributeType);
156                     attributeEl.setAttribute('value', value);
157
158                     requirementEl.appendChild(attributeEl);
159             }
160
161             // children
162             incompatibility.subtree.forEach(function(incompatibility) {
163                     appendRequirementTree(requirementEl, incompatibility);
164             });
165
```

```
166                 // add package for classes
167                 if (type === 'class') {
168                         packageRequirementEl.appendChild(requirementEl);
169                         element.appendChild(packageRequirementEl);
170                 } else {
171                         element.appendChild(requirementEl);
172                 }
173         }
174 }
```