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

Bakalářská práce

Vizualizace rozsáhlých grafů

Plzeň 2021

Jaroslav Hrubý

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 16. srpna 2021

Jaroslav Hrubý

Abstract

In this day and age, the usage of external libraries in software development is a common practice. Therefore it is required to pay attention to the mutual compatibility of the application components and these libraries. The goal of this thesis is the extension of IMiGEr tool - Interactive Multimodal Graph Explorer to this problematic. In theoretical part all related topics are discussed and the design of this extension is proposed. Practical part is devoted to the implementation and achieved results. Conclusion recapitulates the work together with the evaluation the work.

Abstrakt

V dnešní době je využívání externích knihoven při vývoji aplikací již běžnou praxí. Proto je třeba věnovat pozornost vzájemné kompatibilitě komponent aplikací a těchto knihoven. Cílem této práce je rozšíření nástroje IMiGEr - Interactive Multimodal Graph Explorer o tuto problematiku. V teoretické části jsou probrána všechna související témata a je představen návrh na konkrétní rozšíření nástroje. Praktická část se věnuje vzniklé implementaci a dosaženým výsledkům. V závěru je práce zrekapitulována spolu se zhodnocením přínosu práce.

Poděkování

Chtěl bych poděkovat Ing. Lukáši Holému, Ph.D. za ochotu, užitečné rady, trpělivost a čas věnovaný při konzultacích.

Obsah

1 Úvod **1**

2 Seznámení s problematikou **2**

2.1 Testování softwaru 2
2.1.1 Druhy testů 2
2.1.2 Statická analýza 4
2.2 Kontrola kompatibility 6
2.3 JaCC 8
2.3.1 Možnosti ověření kompatibility 9
2.3.2 Princip funkčnosti 11
2.4 Verzovací systémy 12
2.5 Apache Maven 14
2.6 IMiGEr 16
2.7 Techniky pro vizualizaci grafů 18
2.7.1 Overview+Detail 18
2.7.2 Zoom 20
2.7.3 Focus+Context 20

3 Analýza a návrh řešení **22**

3.1 Ověření kompatibility 22
3.2 Procházení veřejných *GitHub* repozitářů 23
3.3 Zobrazení výsledků 24
3.4 Vizualizace grafů nekompatibilit 25
3.5 Kombinace repozitářů s knihovnami 27
3.6 Rozšíření možností vizualizace 28

4 Popis Implementace **30**

4.1 Serverová část 30
4.2 Klientská část 32
4.3 Úpravy nástroje IMiGEr 33
4.4 Uživatelská dokumentace 34
4.5 Testování 37

5 Demonstrace řešení a dosažené výsledky **39**

6 Závěr **42**

Literatura **44**

A Tabulka otestovaných repozitářů **46**

B Struktura přiloženého CD **52**

1 Úvod

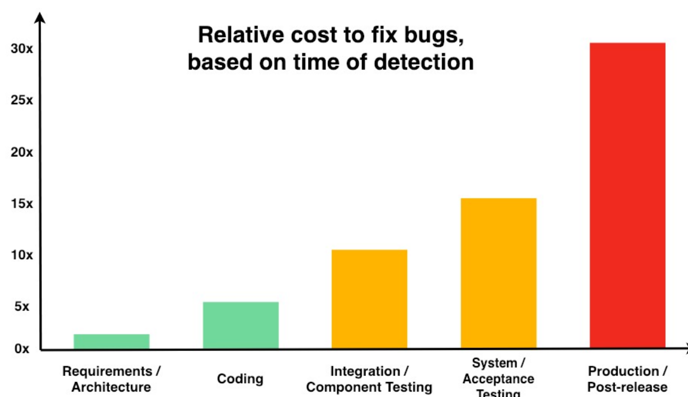
V dnešní době, kdy se rapidně vyvíjí odvětví informačních technologií, je stále více kladen důraz na kvalitu softwaru. Ověřování kvality softwaru představuje různé činnosti, ale bezesporu tou hlavní je testování. Testování je jednou z nejobtížnějších disciplín ve vývojovém cyklu aplikací. Cílem této činnosti je odhalit co největší množství programových chyb. Odhalování chyb je velmi důležité z hlediska nákladů a to jak časových, tak i peněžních.

Jedním z předmětů testování je vzájemná kompatibilita softwaru. Tato problematika je jistě aktuální, protože s čím dál tím větší snahou urychlení vývojového cyklu vznikají znovupoužitelné kódy ve formě knihoven. Tyto knihovny značně urychlují softwarový vývoj, ale zároveň s sebou přináší i otázku zmíněné vzájemné kompatibility. Pomocí různých nástrojů dokážeme vzájemnou kompatibilitu jednotlivých knihoven prověřit. Při vývoji velkých aplikací může být ale počet komponent velmi vysoký, a proto je dobré mít možnost celou situaci přehledně vizualizovat. K vizualizaci vzájemných vztahů mezi komponentami aplikace je více než vhodné využití grafů. Pro vizualizaci a průchod grafů je na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni vyvíjen nástroj IMiGEr - Interactive Multimodal Graph Explorer. Cílem práce je vhodně integrovat do tohoto nástroje problematiku vzájemné kompatibility.

2 Seznámení s problematikou

2.1 Testování softwaru

Každý člověk dělá chyby a s tím je třeba počítat i na poli softwarového vývoje. Pro odhalování chyb v programech se využívá testování. Je důležité, aby testování bylo co nejvíce efektivní. Čím déle chyba v programu existuje, tím exponenciálně stoupají časové a peněžní náklady na její opravu [1]. Odhalená chyba ve fázi vývoje se dá většinou snadno opravit, ale chyba, kterou se odhalit nepodaří, může mít následně při jejím projevu ve finálním produktu fatální následky. Jak se zvyšuje cena spolu s délkou života chyby je ilustrováno na obrázku 1.



Obrázek 1: Náklady na opravu chyby v závislosti na době jejího odhalení [1]

2.1.1 Druhy testů

Testování je velice obsáhlé a testů je celá řada. Testy lze kategorizovat dle jejich významu a zaměření. Testy můžeme například dělit dle přístupu. V tomto případě máme dva různé typy - testy splněním a selháním [2].

- **testy splněním (pozitivní testy)**

  U těchto testů se snažíme ověřit požadovanou funkcionalitu s očekávanými hodnotami. Cílem tedy není software shodit, ale ověřit očekávanou funkčnost.

- **testy selháním (negativní testy)**

  Na rozdíl od předchozího typu se negativní testy zaměřují na ověření chování programu za neočekávaných podmínek. Vstupem takových testů jsou zpravidla chybné hodnoty.

Testy můžeme například dělit také z časového hlediska podle toho, v jakém úseku životního cyklu aplikace se testuje. Radek Kitner [3] na svém webu uvádí, že z tohoto hlediska lze testy kategorizovat mezi:

- **jednotkové (Unit)**

  Jednotkové testy provádějí zpravidla vývojáři ve fázi kódování programu. Cílem jednotkových testů je ověření správné funkčnosti nejmenších částí systému - metod a tříd. Testy ověřují zda nově přidaná nebo změněná část kódu funguje dle očekávání a nepadá na chyby. Jedním z nejznámějších nástrojů pro tvorbu jednotkových testů je nástroj *JUnit*.

- **modul testy**

  Modul testy jsou stále spíše doménou programátorů. Od jednotkových testů se liší jen jejich rozsahem. Zatímco jednotkové testy ověřují funkčnost menších celků, modul testy zpravidla představují otestování komponenty, modulu či celé knihovny.

- **integrační**

  Integrační testy ověřují bezchybnou komunikaci mezi jednotlivými komponentami uvnitř aplikace. Tyto testy provádějí obvykle specializovaní testeři nebo testovací tým. Integraci lze ověřovat nejen mezi komponentami, ale také mezi komponentou a operačním systémem, hardwarem či rozhraním jiného systému. Testování integrace začíná mezi dvěma komponentami a postupně se přidávají další.

- **funkční**

  Funkční testování zkoumá, co systém dělá. Toto testování nepředstavuje ověření funkčnosti jednotlivých funkcí (metod) modulů nebo třídy, ale testuje část funkčnosti celého systému.

Funkční testy se dále dělí na:

  - smoke testy
  - sanity testy
  - regresní testy
  - testy použitelnosti

- **systémové**

  V rámci systémových testů je program testován jako funkční celek. Tyto testy se provádějí většinou v pozdějších fázích vývoje. Hlavním předmětem těchto testů je ověření aplikace z pohledu zákazníka. V průběhu těchto testů se ověřuje, zda aplikace plní svoji úlohu dle zadání zákazníka a zda aplikace dokáže pracovat za neočekávaných situací. Systémové testování probíhá iteračně - nalezené chyby jsou nahlášeny, opraveny a následně se program podrobí dalšímu kolu testování.

- **akceptační**

  Akceptační testy se již provádějí na straně zákazníka. Po úspěšném průběhu předchozích testů je možné aplikaci předat zákazníkovi. Zákazník společně se svým týmem provede akceptační testy podle předem stanovených scénářů. Nalezené chyby zákazník nahlašuje zpět vývojovému týmu. Ten má pak za úkol chybu opravit v co nejkratší době.

2.1.2 Statická analýza

Mimo klasické testování se také hojně provádí tzv. statická analýza kódu. Tato činnost představuje analýzu zdrojového či objektového kódu po přeložení a to bez nutnosti jeho běhu, čímž se liší od analýzy dynamické. K provádění statické analýzy existuje již mnoho dostupných nástrojů. Mnohé z nich lze dokonce integrovat do různých vývojových prostředí (IDE). Mezi takové nástroje patří například *Apache Ant* nebo *Apache Maven*. Princip činnosti nástrojů pro statickou analýzu je vytvoření abstraktního modelu programu a následné hledání chybových vzorů [4]. Výsledkem je pak množina odhalených problémů a nedostatků kódu.

Podle Václava Pecha [5] mezi nejsledovanější oblasti nedostatků v kódu patří:

- **špatné praktiky**

  Užívání špatných praktik při psaní zdrojového kódu může mít za následek, že kód nemusí být za dané situace chybný, avšak může v budoucnosti způsobovat problémy a být tak nespolehlivý.

- **správnost**

  Zdrojový kód se ve výsledku chová jinak, než jeho autor zamýšlel - chyby správnosti.

- **chyby v multijazyčnosti**

  Ve zdrojovém kódu se vyskytují problémy s kódováním znaků. Při vývoji vícejazyčného software je potřeba brát zřetel na tuto problematiku.

- **zranitelnost kódu**

  Ve zdrojovém kódu dochází k použití nezapouzdřených měnitelných objektů. K tomu může dojít například, když třída vrací přímou referenci na její měnitelný atribut (bez vytvoření kopie).

- **vícevláknová korektnost**

  Zdrojový kód je patřičně ošetřený ve všech otázkách problematiky vláken. Typickým příkladem je kritická sekce neboli přístup ke sdíleným datům. V tomto případě musí být zajištěno, aby přístup k datům v daný okamžik mělo nejvýše jedno vlákno.

- **výkon**

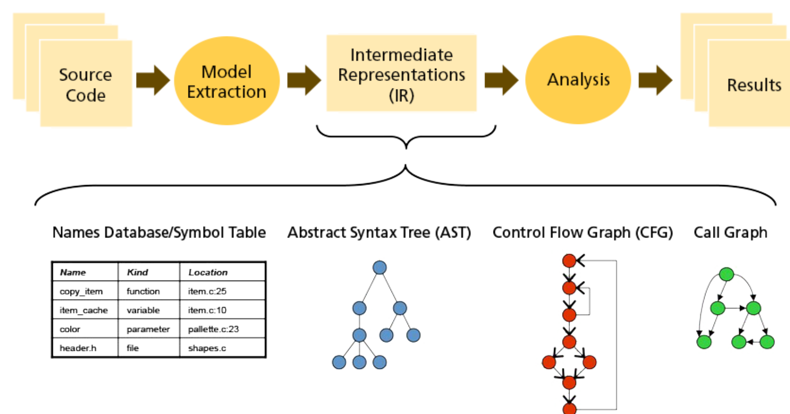
  Zdrojový kód může obsahovat nevhodné konstrukce, které značně zhoršují výkon aplikace.

- **narušování stylu psaní kódu**

  Při psaní zdrojového kódu ve vícečlenných skupinách je dobré držet se jednoho stylu. Tato konvence přispívá nejen k přehlednosti kódu.

- **používání riskantního kódu**

  Programátor by se měl při psaní zdrojového kódu vyvarovat použití podezřelého, nebezpečného nebo také zbytečného kódu.



Obrázek 2: Příklad průběhu statické analýzy [4]

2.2 Kontrola kompatibility

Jedním z předmětů testování je kompatibilita. V dnešní době je snaha využívat znovupoužitelné kódy ve formě externích knihoven v rámci úspory času. To s sebou ale přináší jistá úskalí. Tak jako naše vlastní projekty se i zdrojové kódy externích knihoven postupem času vyvíjejí. To může způsobit značné potíže, pokud využíváme část knihovny, jež byla modifikována. Upravená část kódu se může chovat jinak než očekáváme a to způsobí problémy. Následující příklady jsou popsány podle zdroje [6]. Typickým příkladem takto problémové úpravy je změna typu návratové hodnoty.

```
1  // lib-1.0.jar
2  package lib.specialiseReturnType1;
3  public class Foo {
4   public static java.util.Collection getColl() {
5     return new java.util.ArrayList();
6   }
7  }
8  // lib-2.0.jar
9  package lib.specialiseReturnType1;
10 public class Foo {
```

```
11   public static java.util.List getColl() {
12      return new java.util.ArrayList();
13   }
14 }
15 // client program
16 package specialiseReturnType1;
17 import lib.specialiseReturnType1.Foo;
18 public class Main {
19   public static void main(String[] args) {
20      java.util.Collection coll = Foo.getColl();
21      System.out.println(coll);
22   }
23 }
```

Výpis kódu 1 : změna typu návratové hodnoty [6]

Z příkladu můžeme vidět, že v druhé verzi knihovny došlo ke změně typu návratové hodnoty. To ale způsobí, že při zavolání metody dojde k vyvolání `NoSuchMethodError`. Pro opravu není nutná žádná úprava zdrojového kódu, ale je nutné překompilovat program s verzí knihovny 2.0.

```
1  // lib-1.0.jar
2  package lib.exceptions2;
3  public class Foo {
4   public static void foo() {}
5   }
6  // lib-2.0.jar
7  package lib.exceptions2;
8  import java.io.IOException;
9  public class Foo {
10   public static void foo() throws IOException {
11      throw new IOException();
12   }
13 }
14 // client program
15 package exceptions2;
16 public class Main {
17   public static void main(String[] args) {
18   lib.exceptions2.Foo.foo();
19   }
20 }
```

Výpis kódu 2 : přidání výjimky do metody [6]

V druhém příkladu došlo v druhé verzi knihovny k přidání vyvolání výjimky. Nezachycená výjimka není detekována statickou analýzou během propojení, ale pouze za běhu, kde je výjimka skutečně vyvolána při volání funkce `foo()`. Po takto zavolané metodě je samozřejmě program ukončen s chybou.

```
1  // lib-1.0.jar
2  package lib.primwrap1;
3  public class Foo {
4     public static int MAGIC = 42;
5  }
6  // lib-2.0.jar
7  package lib.primwrap1;
8  public class Foo {
9     public static Integer MAGIC = new Integer(42);
10 }
11/ / client program
12 package primwrap1;
13 import lib.primwrap1.Foo;
14 public class Main {
15    public static void main(String[] args) {
16       int i = Foo.MAGIC;
17       System.out.println(i);
18    }
19 }
```

Výpis kódu 3 : obalení primitivního datového typu [6]

V další ukázce nekompatibility došlo v druhé verzi knihovny k obalení primitivního datového typu `int` do obalové třídy `Integer`. Po takové modifikaci dojde opět k ukončení programu s chybou. Tuto chybu lze vyřešit snadno pomocí rekompilace.

2.3 JaCC

Následující kapitola popisující nástroj JaCC je převzata z diplomové práce Jana Ambrože [7].

JaCC neboli Java Class Comparator je nástroj sloužící k provedení statické analýzy zdrojového kódu a ověření vzájemné kompatibility Java knihoven. Tento nástroj je vyvíjen na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni.

2.3.1 Možnosti ověření kompatibility

JaCC poskytuje tři základní prostředky pro ověření kompatibility:

- **Black list/White list**

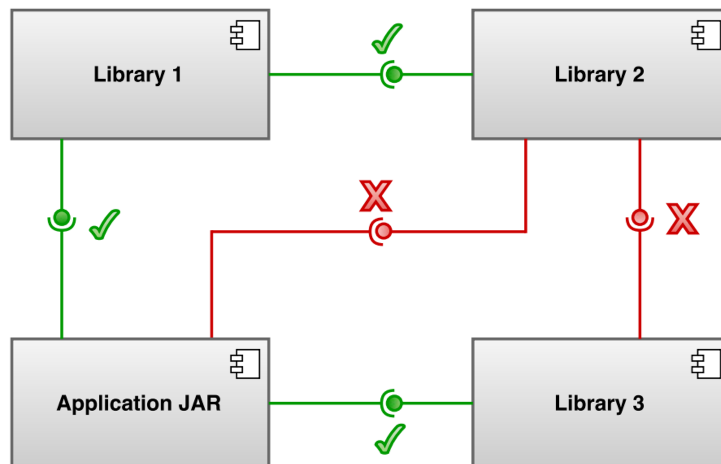
  Tento prostředek umožňuje definovat kontrakt, ve kterém se popíše klintské API, které je zakázáno/dovoleno používat. Výstupem vyhodnocení tohoto prostředku je pak například informace o tom, že je v projektu použito volání nějaké metody, které je ovšem podle kontraktu zakázáno.

- **1:N komparátor**

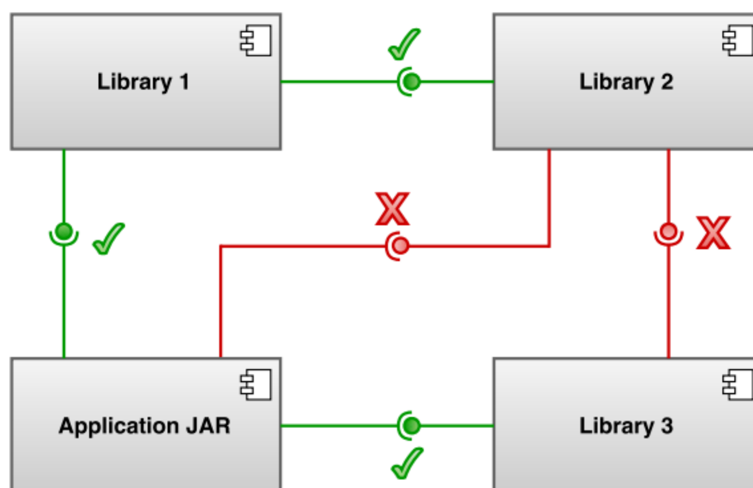
  Umožňuje vyhodnocovat kompatibilitu mezi implementovanou aplikací (1) a použitými cizími knihovnami (N). Toto ověření je ovšem pouze jednoúrovňové a nezachycuje vazby mezi samotnými knihovnami.

- **M:N komparátor**

  Umožňuje vyhodnocovat vzájemnou kompatibilitu všech komponent nehledě na to, zda se jedná o implementovanou aplikaci či cizí knihovnu.



Obrázek 3: Ověření kompatibility pomocí 1:N komparátoru [7]



Obrázek 4: Ověření kompatibility pomocí M:N komparátoru [7]

2.3.2 Princip funkčnosti

Nástroj JaCC vychází ze dvou základních myšlenek:

1. Bytecode vzniká při kompilaci zdrojového kódu v programovacím jazyce Java a obsahuje informaci o stavbě tříd coby základních stavebních prvcích při vývoji v tomto jazyce. Tuto informaci o stavbě tříd je možné s použitím reverzního inženýrství[1] zpětně extrahovat z bytecode a uložit do vhodných datových struktur, se kterými je možné dále pracovat. Obsahem těchto datových struktur pak mohou být například informace o atributech, konstruktorech či metodách pro danou třídu.

2. Tato myšlenka navazuje na myšlenku z prvního bodu a dává do souvislosti kompatibilitu s datovými strukturami získanými prostřednictvím reverzního inženýrství. Máme-li tak například k dispozici datovou reprezentaci téže třídy z různých knihoven, můžeme vzájemným porovnáním struktur těchto tříd rozhodnout o jejich vzájemné kompatibilitě.

[1] **Reverzní inženýrství** je v informatice definováno jako proces analýzy předmětného systému s cílem identifikovat komponenty systému a jejich vzájemné vazby a/nebo vytvořit reprezentaci systému v jiné formě nebo na vyšší úrovni abstrakce [8].

Zpracování nástrojem JaCC probíhá ve 3 hlavních fázích:

1. **Načtení**

Provede se načtení binárních souborů, jejichž kompatibilita má být ověřena a vytvoří se datová reprezentace použitých tříd uvnitř bytecode.

2. **Komparace**

Provede se porovnání datových struktur pro stejnojmenné třídy a výsledky porovnání se následně uloží.

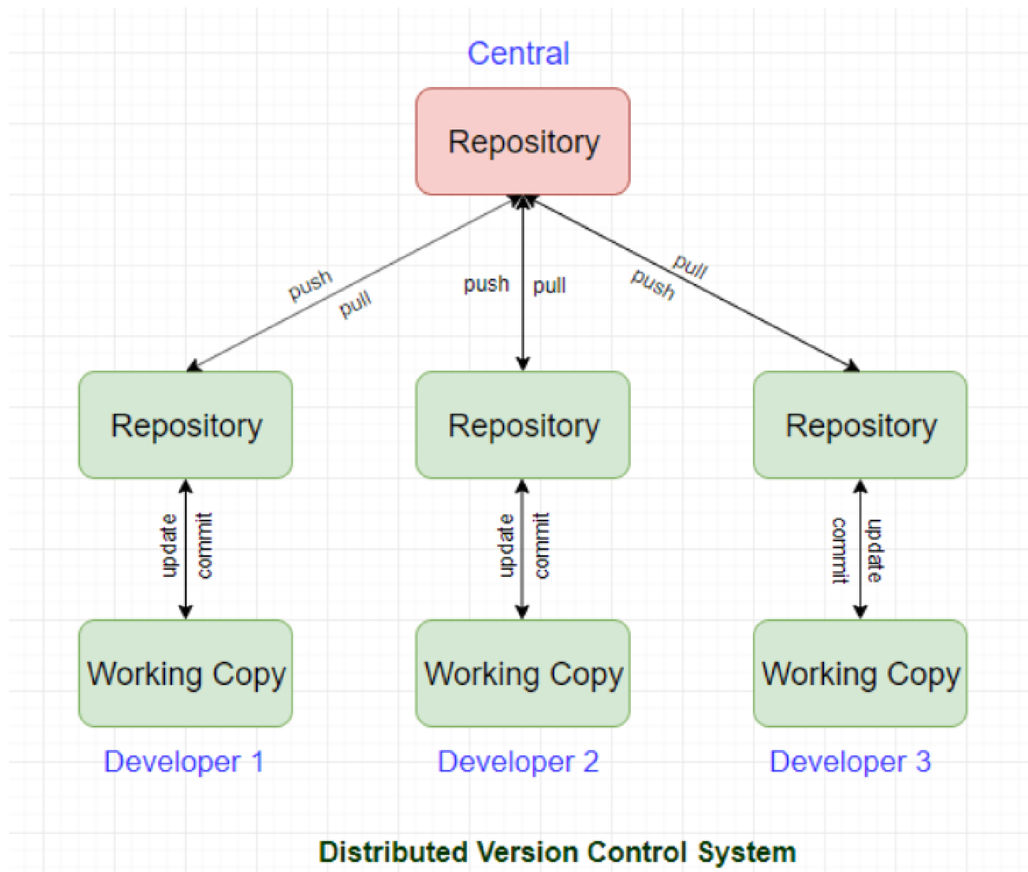
3. **Výpis**

Provede se výpis uložených výsledků porovnání. Aktuálně JaCC dokáže vypsat výsledky do formátů TXT, CSV nebo HTML.

2.4 Verzovací systémy

Jan Faigl [9] ve své přednášce z roku 2016 srozumitelně vysvětluje téma verzovacích systémů, proto následující kapitolu tvoří převážně vybrané poznatky z této přednášky. Verzovací systémy uchovávají historii veškerých změn provedených v informacích nebo datech. Používají se nejčastěji pro sledování změn ve zdrojových kódech softwaru během jeho vývoje. S pomocí těchto systémů můžeme zálohovat veškerá data vzniklá v rámci vývoje, tudíž nedojde k jejich ztrátě v případě poškození lokálního úložiště. Další nespornou výhodou je možnost vyzkoušení nového směru softwaru bez ztráty původní verze. V neposlední řadě řeší verzovací systémy otázku distribuce zdrojových souborů ve vícečlenných týmech vývojářů. V rámci těchto systémů můžeme vzniklé soubory ukládat do jednotlivých úložišť, které nazýváme repozitáře. V repozitáři pak můžeme vidět všechny provedené změny od jeho vzniku. Verzovací systémy disponují také sadou nástrojů pro správu verzí. Pomocí těchto nástrojů dokážeme získat lokální kopii verzovaných souborů. Při změnách modifikujeme pouze zmíněnou lokální kopii příslušné verze a pro potvrzení žádáme o přijetí těchto modifikací (`commit`). Správce verzí následně vytvoří nejbližší vyšší verzi. V případě, že lokální verze není nejnovější verzí, je nutné aktualizovat lokální kopii na verzi novou (`update`).

Dále pokud jsou změny v souladu s lokálními modifikacemi, probíhá sloučení změn (`merge`). V opačném případě je potřeba vyřešit konflikty, které vznikají převážně současnou modifikací stejného místa v souboru. V rámci repozitáře můžeme verze softwaru separovat do tzv. větví (*branches*). Větve umožňují paralelní vývoj vhodný například při postupném přechodu na novější technologie nebo zkoušení nových přístupů. Další výhodou je možnost slučování jednotlivých větví mezi sebou (*branch merge*).



Obrázek 5: Ukázka principu verzovacích systémů [10]

V dnešní době existuje nepřeberné množství verzovacích systémů. Jedním z prvních takových systému byl *Concurrent Version System* (*CVS*). Mezi další můžeme zařadit systémy jako jsou například *Apache Subversion* nebo *Git*. Repozitáře postavené na verzovacím systému *Git* jsou v dnešní době hojně nabízeny zdarma prostřednictvím webových služeb. Mezi takové webové stránky patří například *Bitbucket*, *GitHub* nebo *GitLab*.

2.5 Apache Maven

Maven je rozšířený systém pro správu a sestavování aplikací postavených nad platformou Java [11]. Hlavní důvod vzniku tohoto nástroje je pokus o standardní způsob vytváření projektů, jasnou definici z čeho se projekt skládá, snadný způsob publikování informací o projektu a způsob sdílení JAR souborů v několika projektech [12]. První verze byla vydána roku 2002. Od tohoto roku přibyla mnohá vylepšení a aktuální verze nástroje je nyní 3.6.3.

Základním principem fungování *Mavenu* je popsání projektu pomocí *Project Object Model*. Pomocí tohoto modelu dokážeme popsat softwarový projekt z pohledu jeho zdrojového kódu, závislostí na externích knihovnách, popisu procesu sestavování a různých funkcí s tím spojených. Celý *Project Object Model* je popsán v souboru s jednoduchou XML strukturou. Pomocí této struktury dokážeme definovat jednotlivé části projektu nebo jeho závislosti na externích knihovnách a nástrojích. Dále lze také definovat profily, které můžeme využít při sestavování projektu. Takovýto XML dokument se nachází zpravidla v kořenovém adresáři projektu a je pojmenován `pom.xml`. V případě, že se projekt skládá z vícero dílčích projektů nebo modulů, každý z nich pak obsahuje svůj vlastní `pom.xml` soubor, který dědí vlastnosti od svého nadřazeného souboru.

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>cz.voho</groupId>
  <artifactId>myproject</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>My Project</name>
  <description>Sample project with a dependency</description>
  <url>http://someproject.com/</url>
  <inceptionYear>2000</inceptionYear>

  <dependencies>
    <dependency>
      <groupId>com.company</groupId>
      <artifactId>somelibrary</artifactId>
      <version>1.2.3</version>
    </dependency>
  </dependencies>

</project>
```

Obrázek 6: Ukázka kostry souboru `pom.xml` [11]

Definováním závislostí zajistíme, že budou automaticky vyhledány a nainstalovány. Závislostmi jsou myšlené jiné artefakty (projekty) spravované

systémem *Maven*, které nějaký artefakt vyžaduje ke své kompilaci či funkci. Artefakt je jednoznačně identifikován skupinou (`groupId`), názvem (`artifactId`) a verzí (`version`). Závislosti mají tzv. `scope`, který specifikuje míru a okamžik potřeby dané závislosti [11].

`Scope` může nabývat následujících hodnot:

- **compile (výchozí)**

  Závislost je vyžadována pro překlad i běh aplikace.

- **test**

  Závislost je vyžadována pouze pro překlad a spuštění jednotkových testů.

- **runtime**

  Závislost není vyžadována při překladu aplikace, ale je vyžadována při jejím běhu.

- **provided**

  Závislost je vyžadována pro překlad i běh aplikace, ale bude poskytnuta JVM za běhu.

- **system**

  Obdobně jako v případě `provided` je závislost vyžadována pro překlad i běh aplikace. Je ale za potřebí uvést ručně cestu k požadovanému artefaktu.

- **import**

  Tento `scope` je k dispozici pouze pro typ závislosti *POM*. `Import` označuje, že tato závislost bude nahrazena všemi závislostmi deklarovanými v jejím *POM*.

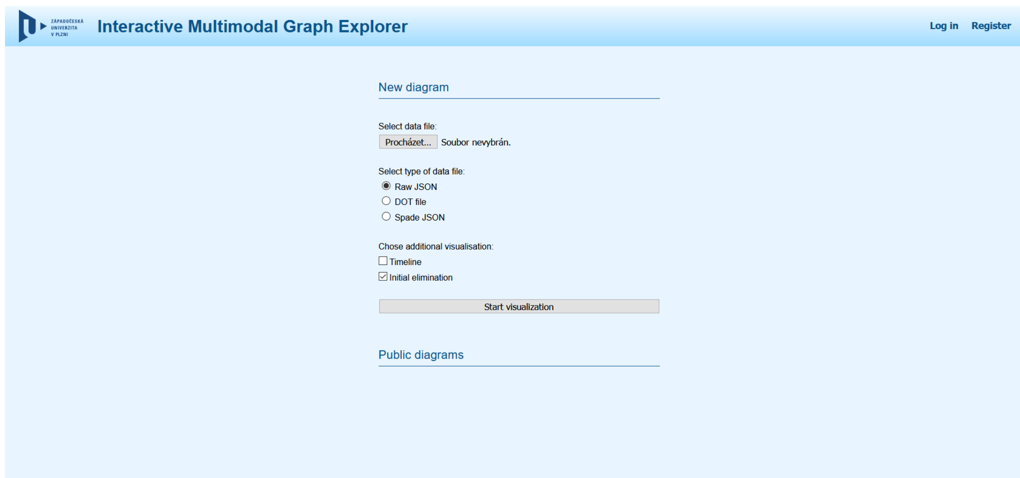
Ve výchozím nastavení nástroj vyhledává knihovny ve svém veřejném centrálním repozitáři. Tuto volbu je samozřejmě možné přenastavit na soukromé nebo firemní repozitáře, které budou obsahovat potřebné knihovny. Celý nástroj je postaven na modulární architektuře a funguje na principu

volání jednotlivých pluginů. Veškeré pluginy jsou volány jednoduchým příkazem `mvn <název_pluginu>:<cíl>`. *Cíl* je v tomto případě název funkce pluginu, kterou chceme volat. *Maven* také nabízí možnost rozdělit proces sestavování na více fází. Díky tomu dokážeme nejen automaticky spouštět pluginy, ale také specifikovat fázi, ve které se má skončit. Tímto způsobem lze definovat například překlad zdrojových kódů bez nutnosti nasazování aplikace, což je ve fázi vývoje žádoucí.

2.6 IMiGEr

Nástroj IMiGEr je webová aplikace pro průzkum grafů. Aplikace umožňuje rychlejší práci s grafy pomocí pokročilé interaktivity, jedinečného skrývání hran a seskupování. Uživatelé jsou schopni vytvořit model složitých grafů mnohem rychleji ve srovnání s tradičními nástroji pro vizualizaci grafů [13]. Podporované formáty vstupních souborů s informacemi o vrcholech a hranách jsou JSON, DOT nebo Spade JSON.

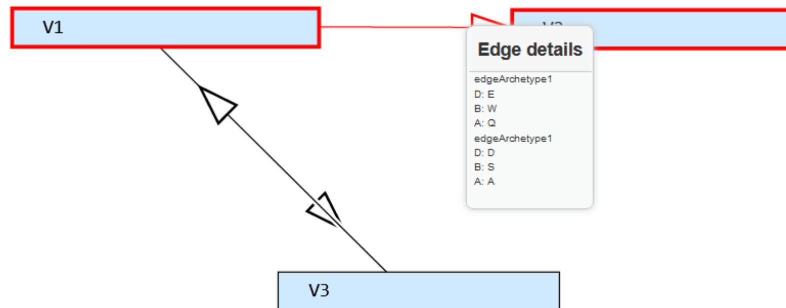
Na úvodní stránce aplikace má uživatel možnost nahrát vstupní soubor obsahující popis grafu v podporovaném formátu. Dále má uživatel možnost zobrazení časové osy nebo využití tzv. počáteční eliminace. Počáteční eliminace v důsledku způsobí, že pokud je počet viditelných prvků větší než 20, aplikace seskupí vrcholy podle jejich typu. To má za následek zejména zlepšení přehlednosti při vizualizaci rozsáhlých grafů.



Obrázek 7: Úvodní stránka nástroje IMiGEr

Po nahrání vstupního souboru a spuštění vizualizace, aplikace analyzuje nahraný soubor a přesměruje uživatele na stránku s již vyobrazeným grafem.

Vyobrazený graf je interaktivní a uživatel má možnost dle libosti vrcholy přesouvat po plátně. Po kliknutí na vrchol grafu se zvýrazní všechny ostatní vrcholy, se kterými je daný vrchol propojený hranou. Uživatel si také po kliknutí na šipku hrany může zobrazit její detail, který popisuje vztah mezi propojenými vrcholy.



Obrázek 8: Ukázka vizualizace jednoduchého grafu

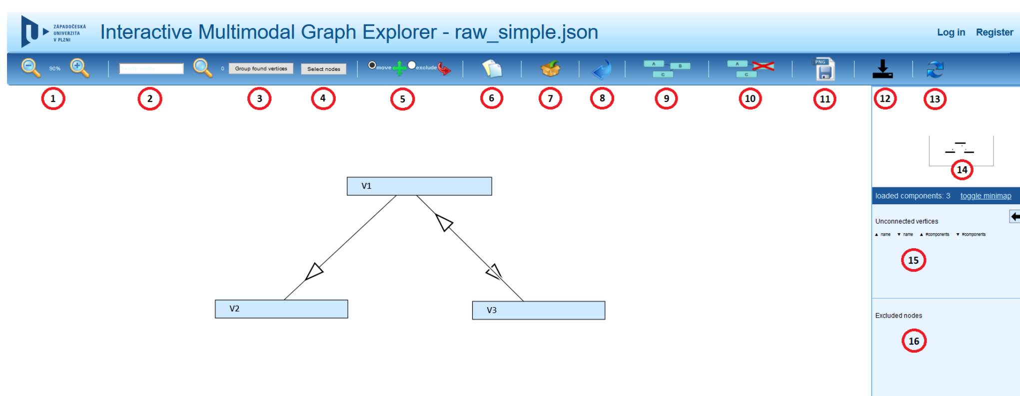
Nástroj IMiGEr dále disponuje paletou několika funkcí, které napomáhají k přehlednému zobrazování grafů. Hlavní plátno, kde je graf vykreslený, ohraničují dvě lišty, pomocí kterých můžeme tyto funkce využívat. Horní lišta poskytuje funkce pro:

- přiblížení a oddálení hlavního plátna (1)
- vyhledání vrcholů podle jejich názvu (2)
- sloučení nalezených vrcholů do skupiny (3)
- vybrání vrcholů podle zadaného kritéria - filtrování (4)
- pohyb vrcholů nebo jejich vyloučení z plátna (5)
- vyloučení vrcholů s největším počtem hran (6)
- sloučení vrcholů s největším počtem hran do skupiny (7)
- návrat na úvodní stránku (8)
- rozložení grafu (9)

- aktivaci počáteční eliminace (10)
- uložení grafu ve formátu PNG (11)
- uložení grafu ve formátu JSON (12)
- obnovení grafu (13)

V boční liště se pak nachází:

- minimapa hlavního plátna (14)
- seznam nespojených vrcholů (vrcholy bez hran) (15)
- seznam vyloučených vrcholů (16)



Obrázek 9: Hlavní stránka nástroje IMiGEr

Aplikace také podporuje registraci uživatelů. Po zadání základních údajů se uživatel může registrovat a přihlásit. Přihlášenému uživateli je poté zpřístupněno ukládání grafů.

2.7 Techniky pro vizualizaci grafů

Tato sekce popisuje techniky pro vizualizaci rozsáhlých grafů *Overview+Detail*, *Zoom*, *Focus+Context*. Tyto techniky jsou popsány podle zdroje [14].

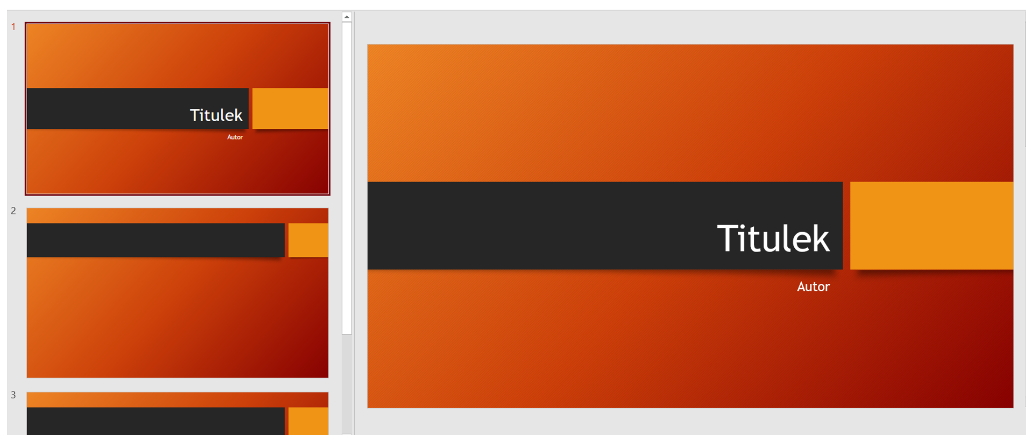
2.7.1 Overview+Detail

Techniku vizualizace *Overview+Detail* lze přeložit jako náhled a detail. Použití této techniky umožňuje souběžné zobrazení náhledu a detailu v jednom okně. Náhled i detail jsou od sebe vizuálně oddělené například ohraničením.

Pokud uživatel provádí akci, například posun mapy v detailu nebo v náhledu, okamžitě je tento posun proveden i v druhé části okna.

Obrázek 10: Ukázka použití techniky *Overview+Detail* v aplikaci Google Maps [14]

Do techniky *Overview+Detail* můžeme také zařadit posuvníky, které umožňují posun v jednom směru. S technikou využívající posuvníky se můžeme setkat například v aplikaci MS PowerPoint. Tato aplikace obsahuje v levé části okna náhledy jednotlivých snímků prezentace. V pravé části je zobrazen detail.



Obrázek 11: Ukázka použití posuvníků v MS PowerPoint

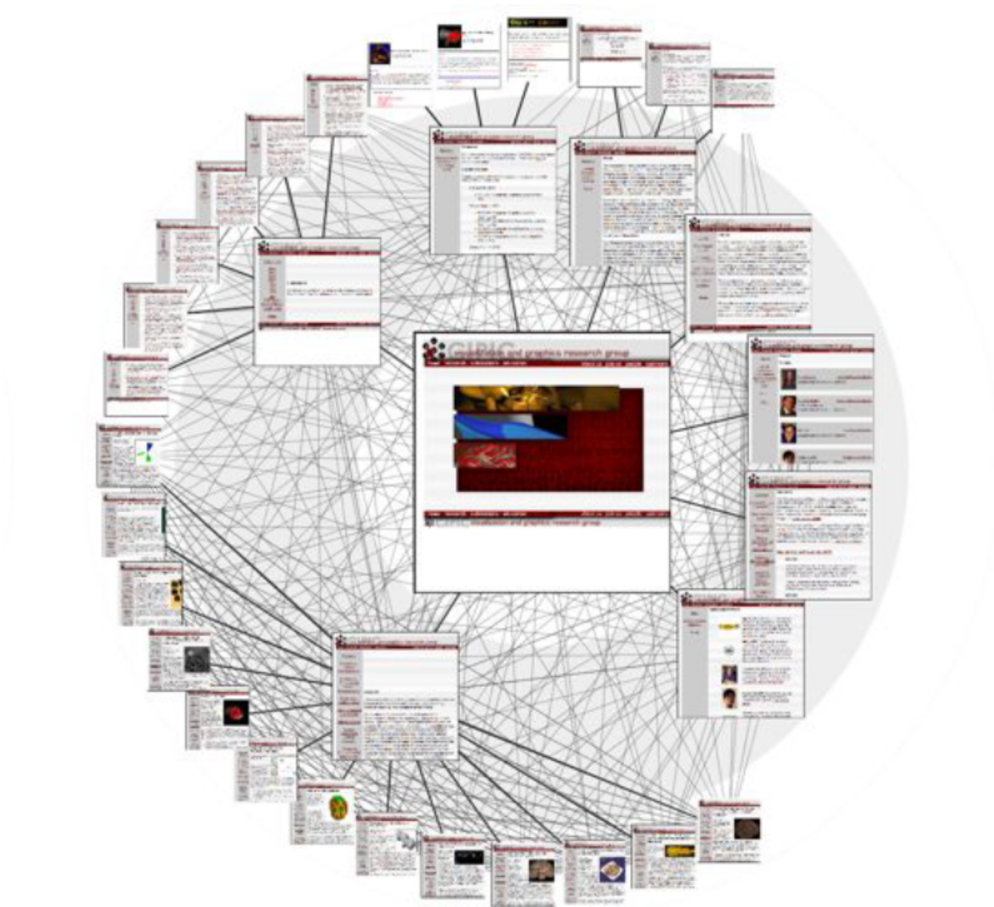
2.7.2 Zoom

Pomocí techniky *zoom* neboli přibližování či oddalování je uživateli poskytnuta možnost rychle přiblížit požadovanou oblast. Technologie je založena na postupném přibližování. V aplikaci je definováno několik úrovní přiblížení a při každém posunu se aplikace přiblíží o jednu úroveň. To umožní uživateli plynulé zvětšení nebo zmenšení požadované oblasti. Úroveň přiblížení může být také zobrazena v okně jako posuvník. Například mapy na internetu tuto techniku hojně využívají. Technika *zoom* bývá také často kombinována s technikou *Overview+Detail*. Tím je dosaženo vyššího uživatelského komfortu při obsluze aplikace.

Obrázek 12: Ukázka techniky *zoom* v aplikaci Google Maps
Zdroj: Google Maps

2.7.3 Focus+Context

Technika *Overview+Detail* rozděluje okno do dvou částí. Technika *Focus+Context* zobrazuje náhled (*Focus*) přímo ve svém okolí (*Context*) v jednom okně. Při použití této techniky dochází ke zvětšení náhledu a nedochází zde k vizuálnímu oddělení přiblížené oblasti a okolí této oblasti. V této technice může být nastaven různý počet úrovní přiblížení při přechodu z oblasti náhledu do oblasti okolí. Pro přiblížení dané oblasti mohou být použité různé algoritmy přiblížení.



Obrázek 13: Ukázka techniky *Focus+Context* [15]

3 Analýza a návrh řešení

Jak bylo již zmíněno v úvodu, cílem této práce je rozšířit nástroj IMiGEr o problematiku vzájemné kompatibility. Vzájemnou kompatibilitu lze ověřovat pomocí zmíněného nástroje JaCC, a proto jej bude třeba integrovat. Dále jsem se rozhodl přidat možnost pro analýzu všech veřejných repozitářů využívajících webovou službu *GitHub*. Získaná data budou následně zpracována a prezentována. Pro zjednodušení jsem se rozhodl analyzovat pouze množinu repozitářů, které využívají *Apache Maven*.

3.1 Ověření kompatibility

První otázkou, kterou je potřeba vyřešit, je volání nástroje JaCC. Tento nástroj potřebuje pro své spuštění dva parametry - cestu do adresáře s JAR soubory testované aplikace a cestu do adresáře s knihovnami, které využívá. V případě že testujeme jednu aplikaci, není předání těchto adresářů problém. Avšak je nutné vyřešit získání JAR souborů aplikace, pokud chceme analyzovat cizí projekt.

Zabalení aplikace do JAR souboru pomocí *Apache Maven* je možné provedením příkazu `mvn package`. Tento příkaz sestaví daný modul aplikace do JAR souboru (pokud není uvedeno jinak) a umístí jej do podadresáře `/target`. Problém nastává v případě, že projekt obsahuje více modulů, respektive podmodulů. V tomto případě je tedy potřeba projít veškeré podmoduly a jejich podadresáře `/target`.

Získání závislých knihoven, které daný modul využívá lze shromáždit pomocí příkazu `mvn dependency:copy-dependencies`. Příkaz nakopíruje všechny závislosti modulu do podadresáře `/target/dependency`. Obdobně jako při sestavování je potřeba v případě vícemodulárního projektu projít všechny tyto podadresáře. Dále je potřeba tyto závislosti vyfiltrovat o případné duplicity.

Po dohledání potřebných souborů je vše připraveno pro zavolání nástroje JaCC, který následně vygeneruje jednoduchou HTML zprávu o nalezených nekompatibilitách. Tato zpráva zobrazuje počet jednotlivých nekompatibilit.

Nekompatibility mohou být:

- **C1** - Chybějící třídy
- **C2** - Nekompatibilní třídy
- **C3** - Rozhraní/třída
- **M1** - Chybějící metody
- **M2** - Nekompatibilní metody
- **F1** - Chybějící pole
- **F2** - Nekompatibilní pole
- **MOD** - Změna modifikátoru
- **M.M1** - Statické/nestatické metody
- **F7** - Statická/nestatická pole

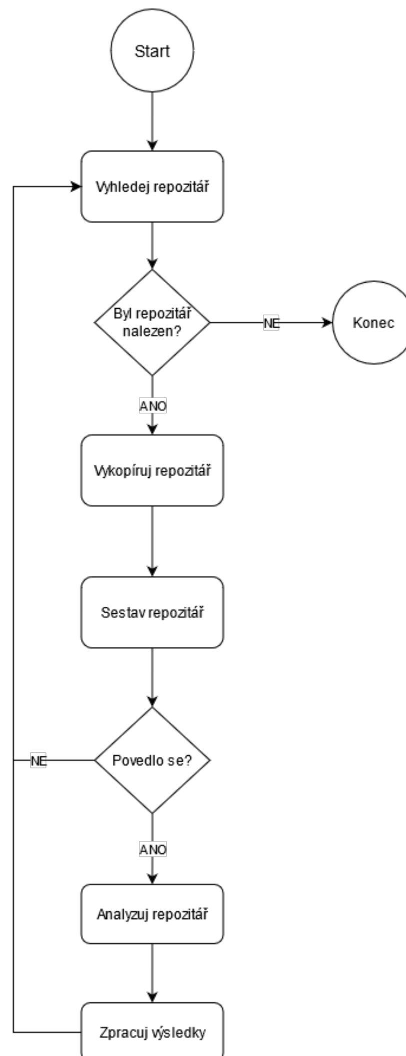
Pro snadné zpracování výsledků bude potřeba nástroj JaCC rozšířit o vypisování těchto údajů, například formou textového souboru.

3.2 Procházení veřejných *GitHub* repozitářů

GitHub nabízí REST API, díky kterému se lze pohodlně dotazovat na různé informace o repozitářích. Pro účely této práce lze využít požadavek `https://api.github.com/repositories`, který v odpovědi vrací informace o veřejných repozitářích [16]. Z důvodu velkého počtu repozitářů jsou tyto informace stránkované. Z těchto informací dokážeme získat URL adresu repozitáře a následně si vykopírovat zdrojové kódy. Dále je ale potřeba se vyhnout zbytečnému kopírování zdrojových kódů repozitářů, které nejsou psány v programovacím jazyce Java nebo nevyužívají *Apache Maven*. Jak bylo dříve popsáno (sekce 2.5), projekty využívající systém *Apache Maven* se vyznačují souborem `pom.xml`, který je zpravidla umisťován v kořenovém adresáři projektů. Můžeme se tedy pomocí požadavku `https://api.github.com/repos/{owner}/{repository}/contents/pom.xml` dotázat na obsah tohoto souboru. Pokud odpověď požadavku vrátí obsah souboru, dá se předpokládat, že projekt využívá *Apache Maven*.

REST API, které *GitHub* poskytuje, má z důvodu ochrany nastavený hodinový limit požadavků. Pro neověřené uživatele je limit nastaven na 60

požadavků za hodinu [17]. To je pro účely tohoto rozšíření nedostačující. Aby se limit zvýšil, je nutné si vygenerovat osobní přístupový token, kterým se lze poté ověřovat. Při ověřování tímto způsobem se hodinový limit požadavků navýší na 5000, což můžeme považovat za dostačující.



Obrázek 14: Vývojový diagram analýzy veřejných repozitářů

3.3 Zobrazení výsledků

Pro zobrazení souhrnných výsledků je třeba navrhnout stránku, kde uživatel uvidí všechny relevantní údaje. Údaje o nalezených nekompatibilitách musí být přehledně zobrazeny. Uživatel si bude moci prohlédnout statistiky o nalezených chybách, zobrazit si výsledky jednotlivých repozitářu či konkrétní repozitáře vyhledávat. Pro srovnávání repozitářů vznikne jednoduchý

bodovací systém. Každé nekompatibilitě bude přidělena váha, která bude představovat její závažnost. Poté bude každému repozitáři vypočítáno jeho skóre podle nalezených problémů.

Nekompatibilita	Váha
C1	1
C2	2
C3	10
F1	5
F2	10
F7	5
M.M1	10
MOD	2
M1	10
M2	10

Tabulka 3.1: Zvolené váhy pro druhy nekompatibilit

Kvůli velkému množství dat je ale třeba zvážit, jak se data budou ukládat a načítat. Jednou z možností je vést evidenci souborů, které budou obsahovat zpracované výsledky. Tato možnost by jistě byla náročná na výkon, a proto bude pro ukládání výsledků lepší využít databázi. Nástroj IMiGEr již využívá *MySQL* databázi, a tak nebude problém připojit další potřebné entity.

3.4 Vizualizace grafů nekompatibilit

Vhodné rozšíření je jistě vizualizace nekompatibilit pomocí grafů. Vrcholy budou reprezentovat jednotlivé komponenty aplikací a závislé knihovny. Hrany budou pak reprezentovat nalezené nekompatibility. Aby bylo možné takto vizualizovat repozitáře pomocí nástroje IMiGEr, je nutné převést celou situaci do podoby grafu. Graf uložený v souboru JSON musí dodržet formát tak, jak jej popisuje Tomáš Šimandl [19]. V souboru je možné definovat:

- **attributeType**
  V této části jsou deklarovány typy atributů, které jsou definovány jako atributy jednotlivých vrcholů nebo hran. Popisujeme zde datový typ atributu, jeho název, případně popis.

- **edgeArchetype**
  Tato část obsahuje typy jednotlivých hran. Zpravidla zde definujeme vztah mezi vrcholy.

- **vertexArchetype**
  V této části můžeme definovat typy jednotlivých vrcholů, respektive jejich název, popis nebo ikonu, která se u vrcholu bude zobrazovat.

- **vertices**
  Tato část obsahuje informace o veškerých vrcholech v grafu - typ, atributy, identifikační číslo vrcholu, popis, název a pozici.

- **edges**
  V této části jsou definovány informace o hranách grafu - popis, typ, index počátečního a cílového vrcholu, identifikační číslo hrany.

- **possibleEnumValues**
  Zde jsou popsány všechny možné hodnoty typu `enum`, kterých mohou atributy vrcholů a hran nabývat.

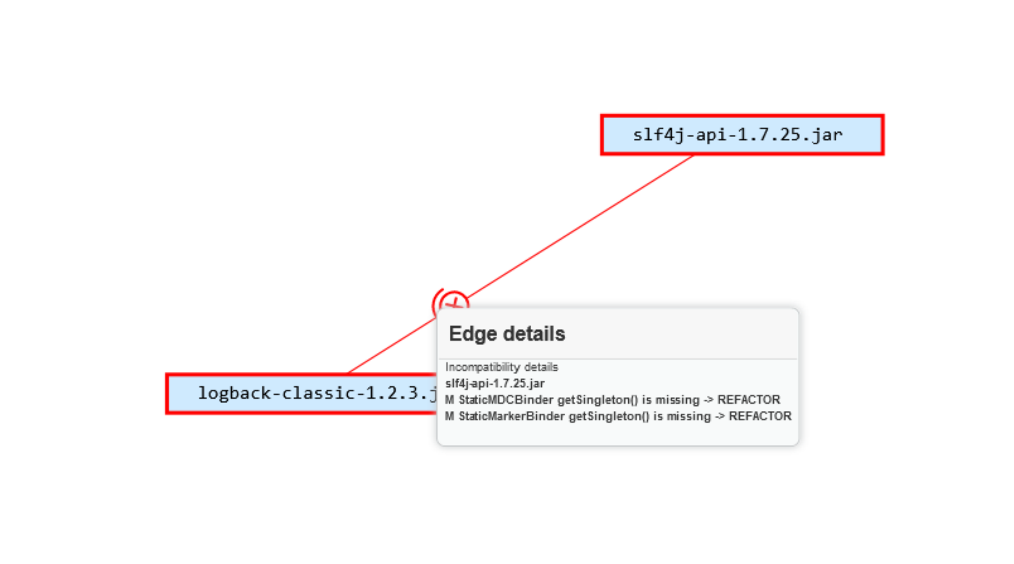
- **groups**
  Tato část slouží pro případné slučování vrcholů do skupin. Každý vrchol musí být maximálně v jedné skupině.

- **sideBar**
  V této části lze definovat vyloučení vrcholů nebo skupin do boční lišty.

- **highlightedVertex**
  Tato část slouží pro definováním identifikačního čísla vrcholu či celé skupiny vrcholů. Tím lze docílit zvýraznění daného vrcholu či skupiny.

- **highlightedEdge**
  V této části lze zajistit zvýraznění hrany na plátně.

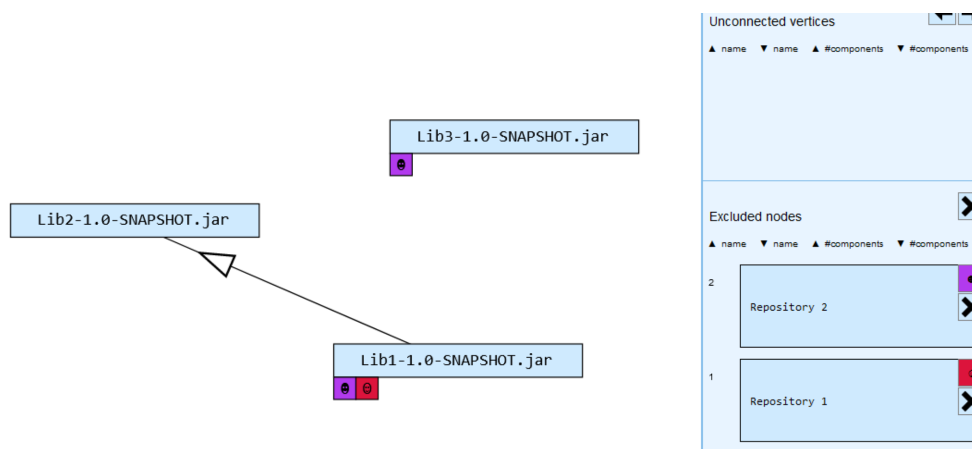
Díky této komplexní struktuře vstupního souboru lze implementovat například zobrazení příčiny nekompatibility v detailu hrany. Tak uživatel přehledně uvidí, která komponenta jeho aplikace využívá problémové knihovny a čím přesně je tato nekompatibilita způsobena. K tomu jsem se rozhodl rozšířit vyskakovací okno detailu hrany o možnost zobrazení HTML. Aktuálně nástroj podporuje pouze prostý text, a to ve fixně daném formátu. S využitím HTML se pak budou nalezené nekompatibility zobrazovat přehledněji, například formou seznamu.



Obrázek 15: Zobrazení příčiny nekompatibility v detailu hrany

3.5 Kombinace repozitářů s knihovnami

Možnou formou rozšíření vizualizace je kombinování více repozitářů najednou. Uživatel si zvolí několik repozitářů, přičemž se mu následně všechny repozitáře sloučí do skupiny, popřípadě vyloučí z plátna. To ve výsledku bude mít za efekt, že uživatel uvidí na hlavním plátně přehledně všechny využívané knihovny a jejich případné problémy.



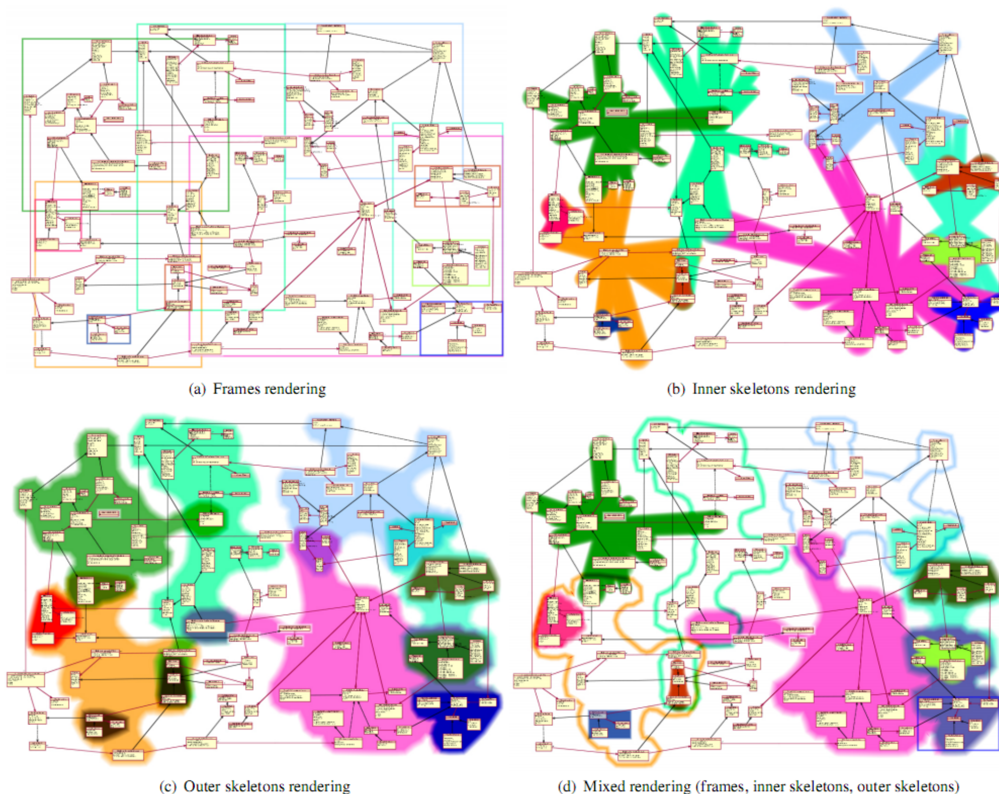
Obrázek 16: Ukázka vyloučení repozitářů

Pro implementaci tohoto rozšíření bude potřeba získat JSON soubory pro příslušné repozitáře. Ty bude následně potřeba sloučit do jednoho a upravit

strukturu. Vrcholy lze od sebe odlišit díky zmíněné části `vertexArchetype` ve vstupním JSON souboru. V této části by šlo nadefinovat, zda se jedná o repozitář nebo závislou knihovnu. Následně by se pomocí skupin sloučily všechny JAR aplikační soubory repozitářů. Výsledné skupiny by pak reprezentovaly jednotlivé repozitáře. Dále bude potřeba vyfiltrovat duplicitní knihovny a upravit příslušné hrany. Na konec je třeba definovat repozitáře v části `sideBar`, aby došlo k jejich vyloučení z hlavního plátna.

3.6 Rozšíření možností vizualizace

Další možností rozšíření je přidání podbarvení skupin vrcholů. To by bylo přínosné zejména při vizualizaci několika projektů. Každý projekt by byl podbarven jinou barvou a uživatel by tak viděl, které komponenty různých projektů na sobě závisí, popřípadě jsou problémové. Ukázka podbarvení grafů je na obrázku 17.



Obrázek 17: Ukázka podbarvení skupin vrcholů [18]

Toto rozšíření by však představovalo velmi náročnou realizaci, a proto jsem se rozhodl jej neimplementovat. Dále by bylo dobré rozšířit nástroj

IMiGEr o možnost změny symbolu, který se zobrazuje na hranách grafu. Aktuálně nástroj umí zobrazovat pouze trojúhelníky, respektive šipky. Pokud by šlo změnit zobrazovaný symbol, mohl by se nahradit například symboly dle UML notace. Tak by šlo lépe přiblížit danou situaci.

4 Popis Implementace

Z důvodu zachování původní funkcionality nástroje IMiGEr, která představuje obecné zobrazování grafů, jsem se rozhodl rozšíření implementovat převážně formou návazné aplikace. Aplikace má dvě hlavní části. Serverová část dokáže analyzovat projekty a výsledky ukládat do *MySQL* databáze. Druhou částí je webový klient, který na základě záznamů z databáze zobrazuje dosažené výsledky.

4.1 Serverová část

Tato část má na starost analyzování projektů pomocí nástroje JaCC. Aplikace dokáže běžet ve dvou módech. První mód analyzuje pouze zadaný projekt. Druhou možností běhu je analýza veřejných *GitHub* repozitářů. Po spuštění serverové části aplikace validuje předané parametry a rozhodne, v kterém režimu bude analýza probíhat. V případě analýzy projektu aplikace ověří vzájemnou kompatibilitu, zpracuje výsledky a uloží je do databáze. V případě analýzy veřejných repozitářů se ověří přítomnost konfiguračního souboru `since.txt` (viz. Uživatelská dokumentace) a spustí se proces analýzy. V prvním kroku aplikace hledá pomocí zmíněného REST API url adresu veřejného repozitáře využívajícího systém *Apache Maven*. Během této činnosti se server dotazuje na veřejné repozitáře a zároveň ověřuje přítomnost souboru `pom.xml`. Po úspěšném nalezení odpovídajícího repozitáře se projekt vykopíruje na lokální disk. Následně pak pomocí zmíněných *Maven* příkazů aplikace projekt sestaví a vykopíruje všechny knihovny, které projekt využívá. Vzniklé JAR soubory jsou poté dohledány a překopírovány do dočasného adresáře. Obdobně aplikace najde využívané knihovny, vyřadí případné duplicity a výslednou množinu knihoven nakopíruje do dalšího dočasného adresáře. Obsah zmíněných adresářů je dále použit pro ověření kompatibility skrze nástroj JaCC. Výsledek je následně zpracován a uložen do *MySQL* databáze. Během zpracovávání výsledků jsou procházeny nalezené nekompatibility, na základě kterých se generují vstupní data pro nástroj IMiGEr. Aplikační soubory a závislé knihovny jsou reprezentovány jako vrcholy grafu. Dále pokud mezi dvěma komponentami existuje nekompatibilita, vznikne mezi příslušnými vrcholy hrana, do jejíž detailu se uloží popis nalezené nekompatibility. Po uložení výsledků jsou vzniklé soubory společně s projektem smazány. Následně je vyhledán další vyhovující repozitář a proces se opakuje. V případě, že se projekt nepodaří sestavit, je ignorován a vyhledá se

nový. Takto aplikace postupuje dokud neprojde veškeré dostupné repozitáře.

Serverová část strukturována 4 balíků:

- **core**

  Balík `core` obsahuje třídy, které spouštějí a obsluhují analýzu repozitářů. Jmenovitě tento balík obsahuje třídy:

  - **Main** - Validuje vstupní parametry a na základě nich spouští analýzu.
  - **Analyzer** - Obsluhuje celou anylýzu.

- **service**

  Tento balík obsahuje třídy:

  - **GitService** - Obsluhuje webovou službu *GitHub*.
  - **MavenService** - Obsluhuje *Maven* příkazy.
  - **ResultService** - Zpracovává výsledky generované nástrojem JaCC.
  - **ScoreService** - Počítá skóre repozitáře.

- **storage**

  Balík `storage` obsahuje třídu, která se připojuje k databázi a ukládá do ni výsledky analýz.

- **utils**

  Tento balík obsahuje třídy:

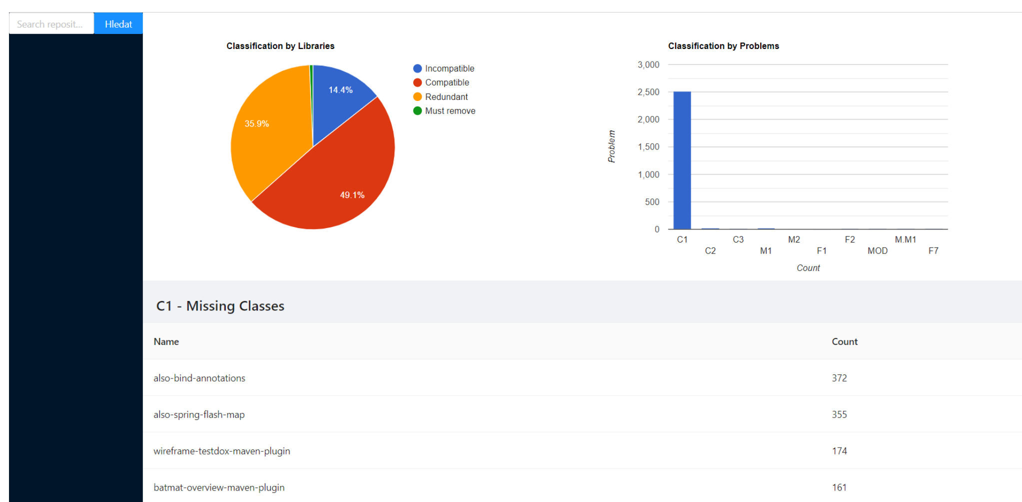
  - **ExtensionFilenameFilter** - Implementace rozhraní `FilenameFilter`, která slouží k filtrování názvů souborů podle přípony.
  - **FileSearcher** - Vyhledává v projektu JAR soubory aplikace a závislé knihovny.
  - **JSONGenerator** - Generuje vstupní JSON soubor pro nástroj IMiGEr.
  - **Graph** - Třída představující celý graf.
  - **Edge** - Třída reprezentující hranu grafu.
  - **SubEdgeInfo** - Třída obsahující informace o konkrétní hraně.

4.2 Klientská část

Klientská část načítá zpracované výsledky z databáze a ty zobrazuje uživateli. Na hlavním panelu se uživateli zobrazují dva grafy, které představují souhrnnou statistiku již analyzovaných projektů. První graf je výsečového typu a zobrazuje podíl klasifikací knihoven. Druhý zobrazovaný graf je sloupcový a zobrazuje počet konkrétních typů nekompatibilit. Dále se v okně vyskytuje boční panel, kde je implementováno vyhledávání repozitářů podle jejich názvu.

Tato část je strukturována celkem do 5 adresářů:

- **components** - V tomto adresáři jsou implementovány tabulky pro zobrazování vybraných údajů.
- **dbconfig** - Tento adresář slouží pro konfiguraci databáze.
- **pages** - Adresář `pages` obsahuje implementaci hlavního okna a podadresář `api`. V tomto podadresáři je implementováno rozhraní pro volání potřebných SQL příkazů.
- **public** - Tento adresář obsahuje soubory potřebné pro správné zobrazování generovaných HTML zpráv.
- **types** - V tomto adresáři se nachází typy a rozhraní.

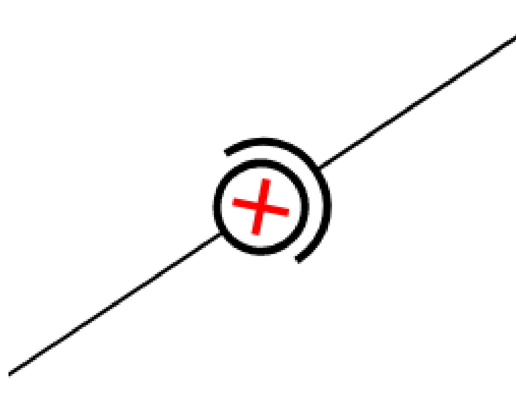


Obrázek 18: Ukázka hlavní obrazovky

4.3 Úpravy nástroje IMiGEr

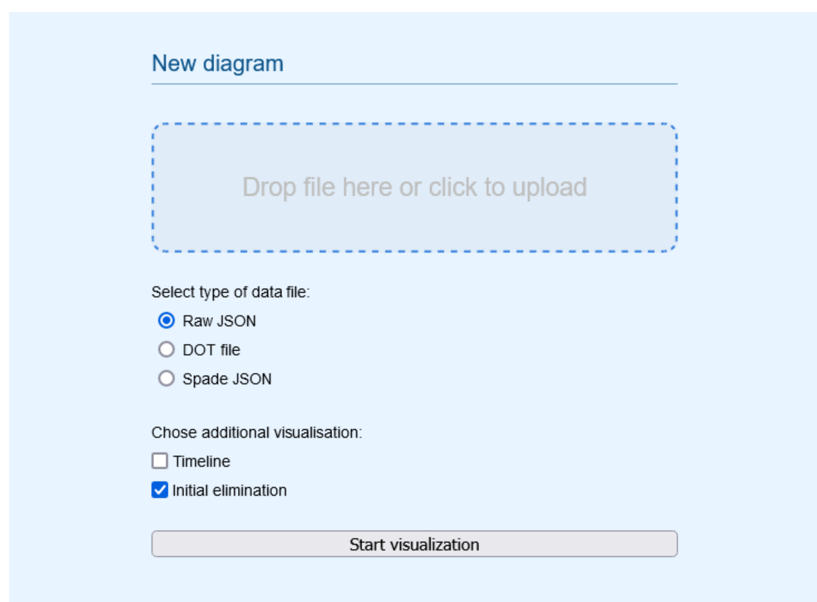
V rámci implementace bylo také potřeba provést úpravy samotného nástroje IMiGEr. Jako první byla implementována podpora zobrazování HTML seznamu v detailu hran grafu. Při generování vstupního JSON souboru v serverové části návazné aplikace je do sekce popisující parametry hrany přidávána navíc položka `incompatibilities`. Tato položka obsahuje data popisující nalezené nekompatibility. Pokud aplikace položku detekuje, spustí se zpracovávání těchto dat, při kterém jsou získávány informace o nekompatibilitách a následně přidávány do HTML seznamu.

Další rozšíření vzniklo v oblasti vykreslování hran. Pokud aplikace detekuje zmíněnou položku `incompatibilities` v parametrech hrany, vykreslí se namísto obvyklé šipky symbol ”lízátka”, který dle UML notace reprezentuje rozhraní mezi dvěma komponentami. Navíc je do tohoto symbolu vykreslen červený křížek, který symbolizuje konflikt mezi příslušnými komponentami daného projektu.



Obrázek 19: Ukázka symbolu ”lízátka”

Při soustavné práci s nástrojem IMiGEr, kdy jsem testoval vygenerované soubory, jsem byl na úvodní obrazovce neustále nucen vybírat příslušný soubor přes stisknutí tlačítka (viz obrázek 7) a následné hledání v adresářové struktuře. Pro snadnější nahrávání vstupních souborů jsem se rozhodl implementovat oblast podporující operaci *drag-and-drop* namísto zmíněného tlačítka. Díky tomuto rozšíření je nahrávání souborů pro uživatele pohodlnější. Uživateli nyní stačí přetáhnout vstupní soubor do vyznačené oblasti a tím jej nahrát. Druhou možností je kliknutí na vyznačenou oblast, jež vyvolá průzkumník souborů jako v původní verzi.



Obrázek 20: Ukázka *drag-and-drop* oblasti

4.4 Uživatelská dokumentace

Před spuštěním samotných částí je třeba mít funkční *MySQL* databázi. V této databázi je třeba pomocí skriptu *create_table.sql* vytvořit tabulku *results*, do které se budou přidávat výsledky.

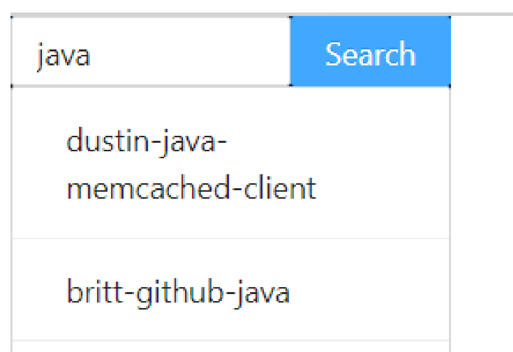
Serverová část je implementována v jazyce Java, a tak se spouští pomocí jednoduchého příkazu `java -jar <název_spustitelného_souboru> <parametry>`. Problém může nastat při sestavování spustitelného JAR souboru - příkaz `mvn package`. Nástroj JaCC, který aplikace využívá, je součástí knihovny *verifa-conformance-cli*. Tato knihovna není volně dostupná, a proto je nutné knihovnu přidat ručně. Druhou komplikací je závislost na samotném nástroji IMiGEr. Ten je ve výchozím stavu sestavován do podoby WAR souboru. Tuto volbu je tedy nutné změnit a sestavit nástroj do klasického JAR souboru. Po vyřešení těchto úskalí se již aplikace automaticky sestaví zmíněným příkazem. Jak již bylo zmíněno, serverová část může fungovat ve dvou režimech. Pro analyzování lokálního projektu je potřeba aplikaci spustit se třemi parametry. Prvním parametrem je název analyzovaného projektu. Druhý vstupní parametr je cesta do adresáře s JAR soubory projektu. Posledním parametrem je cesta do adresáře se závislými knihovnami. Pro analýzu veřejných *GitHub* repozitářů se aplikace spouští pouze s jedním parametrem. Tímto parametrem je osobní přístupový token, který je

potřeba si vygenerovat. Vygenerovat si osobní přístupový token na *GitHub* mohou pouze registrovaní uživatelé podle návodu v dokumentaci [20]. Po spuštění začne program postupně analyzovat veřejné repozitáře podle data, kdy byly vytvořené. V souboru *since.txt* může uživatel definovat identifikační číslo repozitáře, od kterého program začne analýzu. Po ověření kompatibility každého projektu se soubor aktualizuje. Tento soubor musí být umístěn ve stejném adresáři jako spustitelný soubor programu. Při startu bez přítomnosti tohoto souboru, začne program analýzu od začátku. Aby aplikace fungovala, je potřeba mít nainstalovaný *Apache Maven* a správně nadefinovanou systémovou proměnnou *M2_HOME*.

Klientskou část lze spustit pomocí běhového prostředí *Node.js* a balíčkového systému *npm*. Provedením příkazu `npm install` se nainstalují potřebné balíčky. Následně lze aplikaci sestavit pomocí příkazu `npm run build` a spustit příkazem `npm start`. Po spuštění je aplikace dostupná na adrese `http://localhost:3000/`. Na této adrese je uživateli zobrazena hlavní stránka aplikace. Uživatel má možnost provádět tyto operace:

- **vyhledání repozitáře**

  Pomocí textového pole a tlačítka *Search* může uživatel vyhledávat repozitáře podle jejich jména. Nalezená jména se následně zobrazí v bočním panelu.



Obrázek 21: Ukázka vyhledávání repozitářů

- **zobrazení repozitářů podle nekompatibility**

  Po kliknutí na sloupec grafu, který zobrazuje počet nekompatibilit, je uživateli zobrazena tabulka s deseti repozitáři, které obsahují nejvíce nekompatibilit vybraného typu.

M1 - Missing Methods	
Name	Count
wireframe-testdox-maven-plugin	4
hns-ringojs	3
caillette-novelang	3

Obrázek 22: Ukázka tabulky nekompatibilit

- **zobrazení skóre repozitáře**

  Pokud uživatel klikne na nalezený repozitář v bočním panelu, objeví se tabulka zobrazující skóre grafu. Tuto akci lze provést také kliknutím na konkrétní řádek v tabulce z předchozího bodu.

- **zobrazení výsledků analýzy**

  Společně se zobrazením skóre v předchozím případě se objeví 3 tlačítka. První tlačítko slouží ke zobrazení výsledku analýzy v daném okně. Druhé tlačítko slouží pro zobrazení výsledku analýzy v nové kartě prohlížeče.

- **vizualizace v IMiGEru**

  Pomocí třetího tlačítka si může uživatel vizualizovat nekompatibility vybraného repozitáře pomocí nástroje IMiGEr.

logback-logback-classic		
Name		Score
logback-logback-classic		133
Show report	Show report in new tab	Visualise in IMiGEr

Obrázek 23: Ukázka tabulky skóre a tlačítek

Aby byla zajištěna správná funkčnost, je třeba nakonfigurovat soubor `.env`. V tomto souboru se konfigurují hodnoty:

- **DBHOST** - hostovací server databáze
- **DBUSER** - autentizační jméno

- **DBPASSWORD** - autentizační heslo
- **DBNAME** - název databáze
- **IMIGERURL** - url adresa nástroje IMiGEr

4.5 Testování

Výslednou implementaci bylo potřeba samozřejmě také otestovat. Serverová část byla otestována jednotkovými testy. V rámci těchto testů vzniklo několik testovacích tříd:

- **FileSearcherTest**

  Tato třída testuje funkčnost třídy `FileSearcher`. Testuje zda vyhledávání JAR souborů funguje správně.

- **ResultServiceTest**

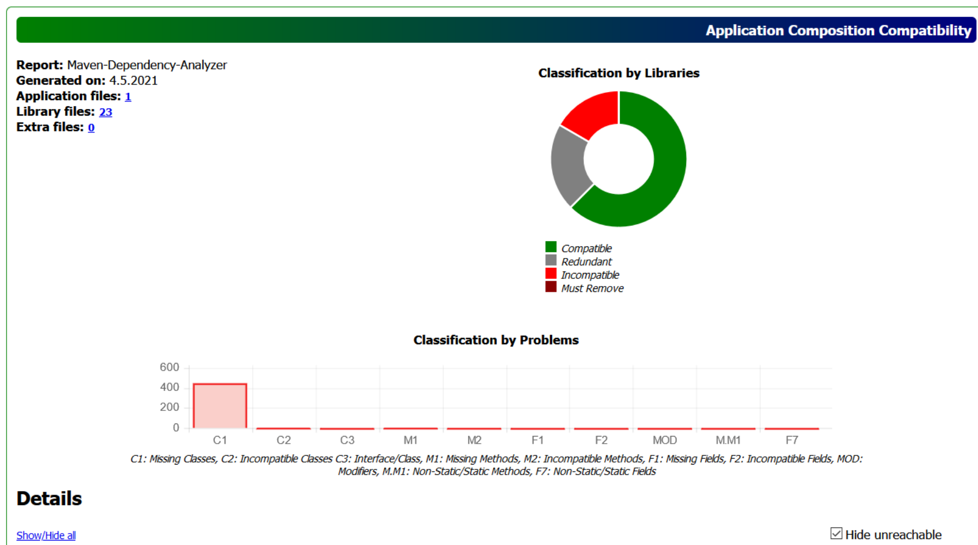
  Tato třída testuje funkčnost třídy `ResultService`. Předmětem testování je načítání výsledků vygenerovaných nástrojem JaCC.

- **ScoreServiceTest**

  Tato třída testuje funkčnost třídy `ScoreService`. Zde se testuje, zda zmíněná třída vypočítá správné skóre na základě výsledků analýzy.

Klientská část byla otestována uživatelským způsobem. Oslovil jsem několik kolegů, aby otestovali vzniklé grafické rozhraní. Při tomto testování bylo zjištěno, že nástroj IMiGEr nedokáže zobrazit diskrétní graf - pouze vrcholy, bez hran. Tato nestandardní situace byla ošetřena jednoduchým upozorněním. Dále byla testována zejména uživatelská přívětivost. Ta byla až na malé připomínky hodnocena převážně kladně. Nahlášené připomínky jsem poté zapracoval do implementace.

Jelikož aplikace využívá externí knihovny, byla také ověřena vzájemná kompatibilita. Jak je vidět z obrázku 22, byly objeveny jisté nedostatky. Konkrétně bylo nalezeno 448 případů chybějících tříd. Tyto nekompatibility jsou však považovány za nedosažitelné, a tak nikterak neohrožují běh aplikace.



Obrázek 24: Výsledek analýzy výsledné implementace

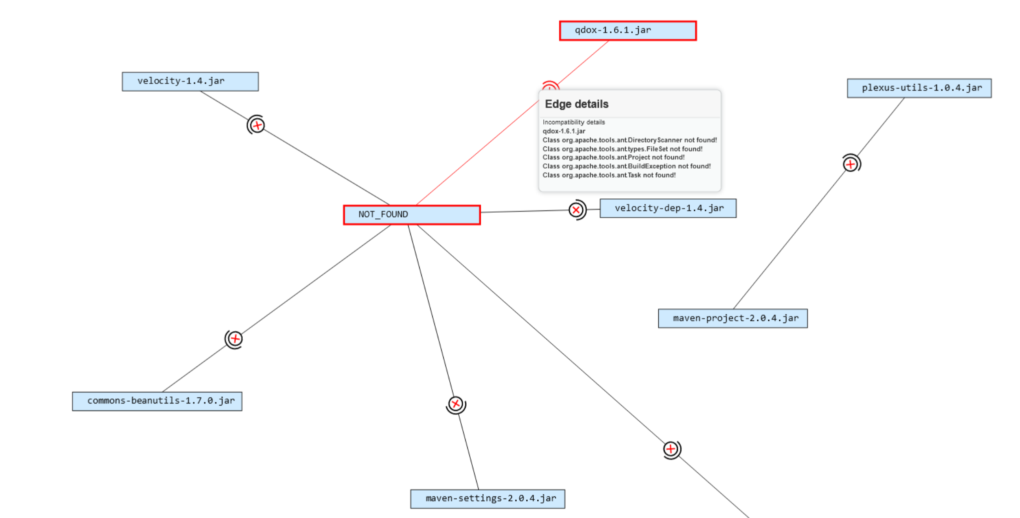
5 Demonstrace řešení a dosažené výsledky

V rámci testování aplikace byla spuštěna analýza veřejných *GitHub* repozitářů. Celkem byla kompatibilita ověřena u 170 repozitářů. Do této testovací množiny bylo začleněno 5 projektů, které byly otestovány samostatně. Těchto 5 knihoven bylo vybráno z 10 nejpoužívanějších knihoven podle centrálního *Maven* repozitáře. Dle očekávání nebyly nalezeny u těchto knihoven žádné nekompatibility. Výjimkou byla však knihovna *logback-classic*, u které bylo objeveno celkem 125 nekompatibilit. Některé z těchto nekompatibilit byly navíc dosažitelné (může dojít na jejich volání). Z toho můžeme usoudit, že používání této knihovny je potenciálně nebezpečné. Dále v tabulce 5.1 můžeme vidět počet nalezených nekompatibilit v testované množině repozitářů. Podrobný přehled všech testovaných repozitářů a v nich objevených nekompatibilit se nachází v příloze.

Nekompatibilita	Počet
C1	5684
C2	86
C3	0
F1	18
F2	0
F7	0
M.M1	70
MOD	0
M1	82
M2	94

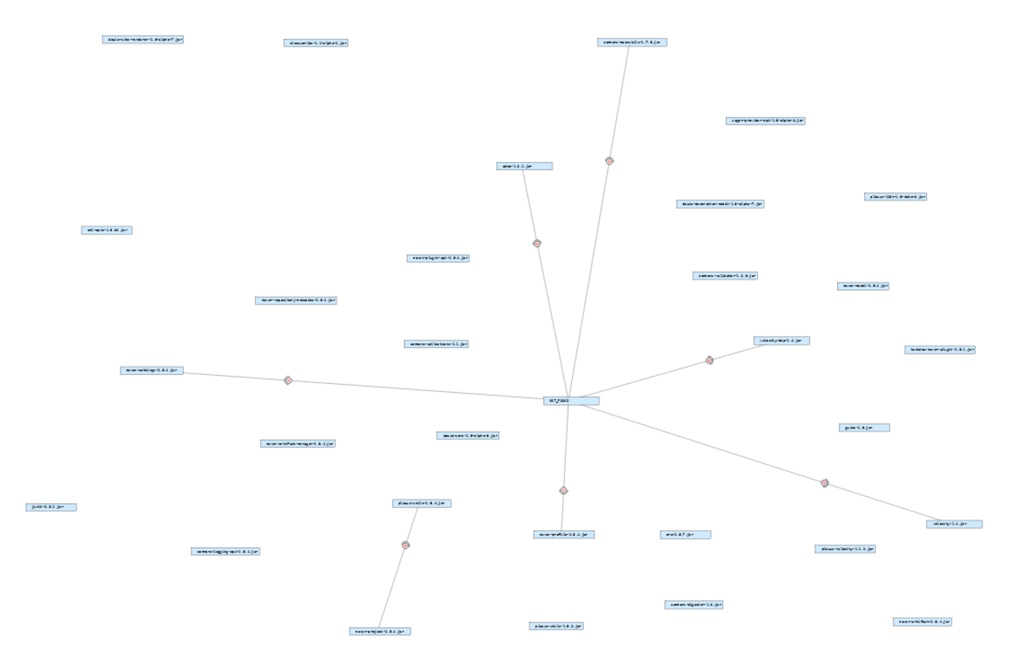
Tabulka 5.1: Počet nalezených nekompatibilit v testovaných repozitářích

Na následujícím obrázku je vidět ukázka zobrazení nekompatibility v rozkliknutém detailem hrany. V detailu může uživatel vidět příčinu nekompatibility. V tomto konkrétním případě dané knihovně chybí 5 konkrétních tříd. Takovýto graf lze pak zobrazit pro libovolný otestovaný repozitář, který obsahuje minimálně jednu nekompatibilitu.



Obrázek 25: Zobrazení nekompatibility v otestovaném repozitáři

Na dalším obrázku je příklad vizualizace situace v konkrétním projektu. V tomto případě u většiny komponent nebyl nalezen žádný problém (nejsou propojeny hranou). Na druhou stranu je vidět, že bylo odhaleno 7 problémových knihoven, které obsahují nekompatibility.



Obrázek 26: Vizualizace konkrétního projektu

6 Závěr

Cílem této práce bylo rozšíření již existujícího nástroje IMiGEr o problematiku vzájemné kompatibility. V teoretické části byla probrána související témata. Mezi ně patří například téma testování, vzájemné kompatibility nebo technik pro zobrazování grafů. Následně byla navrhnuta rozšíření a jejich případná úskalí. V praktické části byla dále podrobně popsána vytvořená implementace. V této části došlo také ke zhodnocení dosažených výsledků a demonstraci řešení.

Rozšíření nástroje IMiGEr o kontrolu kompatibility bylo úspěšné vytvořeno formou návazné aplikace. Aplikace nyní dokáže analyzovat jak jednotlivé projekty, tak veřejné *GitHub* repozitáře. Tato část rozšíření byla z hlediska času nejvíce nákladná, protože bylo zapotřebí se nejprve seznámit s novými nástroji a technologiemi. Dále se povedlo vytvořit grafické rozhraní, které výsledky analýz přehledně zobrazuje. Díky vzniklým úpravám nástroje IMiGEr pak lze celou situaci v konkrétních projektech vizualizovat pomocí grafů. Z důvodu časové náročnosti této implementace nedošlo na možnost kombinace více repozitářů v jednom grafu. Toto rozšíření bude ale jistě možnost implementovat v rámci některého z dalších školních projektů, popřípadě diplomové práce.

Z hlediska přínosnosti vzniklé implementace je potřeba ji srovnávat s ruční analýzou a vytvářením grafů. Bez pomoci vzniklého rozšíření by uživatel musel provést analýzu ručně a výsledky také ručně zpracovávat. To by znamenalo velké množství práce s vytvářením vstupního souboru pro nástroj IMiGEr nebo kreslením grafu nekompatibilit ručně. Přínosnost těchto rozšíření je tedy zřejmá. Další využití se jistě objeví s otestováním většího množství repozitářů.

Seznam použitých zkratek

TXT - Text file
CSV - Comma-Separated Values
URL - Uniform Resource Locator
XML - Extensible Markup Language
JAR - Java ARchive
JVM - Java Virtual Machine
DOT - graph description language
JSON - JavaScript Object Notation
REST - REpresentational State Transfer
API - Application Programming Interface
MS - MicroSoft
SQL - Structured Query Language
WAR - Web Application Resource
HTML - HyperText Markup Language
UML - Unified Modeling Language

Literatura

[1] The exponential cost of fixing bugs, January 29, 2019
https://deepsource.io/blog/exponential-cost-of-fixing-bugs/

[2] Testy splněním a selháním, Tomáš Hlava, 20.8.2011
http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/testy-splnenim-a-selhanim/

[3] Typy testování software, Radek Kitner
https://kitner.cz/testovani_softwaru/typy-testovani-software-trideni-testu/

[4] How Static Analysis Works, Verifysoft Technology
https://www.verifysoft.com/en_grammatech_how_static_analysis_works.html

[5] Statická analýza kódu - za kód bez chyb, Václav Pech, JetBrains, Inc. Dostupné z: http://www.java.cz/dwn/1003/8159_czjug-vaclav-pech-static-code-analysis.pdf

[6] What Java Developers Know About Compatibility, And Why This Matters, DIETRICH, JEZEK, BRADA, 2014
http://arxiv.org/pdf/1408.2607v1.pdf

[7] Jan Ambrož, Výkonnostní a paměťová optimalizace nástroje JaCC, Plzeň 2016. Diplomová práce. ZČU, FAV, KIV.

[8] J. Sochor. Údržba softwaru. Zpravodaj ÚVT MU. ISSN 1212-0901, 1996, roč. VI, č. 3, s. 15-20. Dostupné z: http://webserver.ics.muni.cz/zpravodaj/articles/61.html

[9] Úvod do verzovacích systémů, Jan Faigl, 2016 Dostpuné z: https://cw.fel.cvut.cz/old/_media/courses/a0b36pr2/lectures/lecture11-slides.pdf

[10] Introduction To Version Control System, OSTechNix
https://ostechnix.com/introduction-to-version-control-system/

[11] Maven, Vojtěch Horděјčuk
http://voho.eu/wiki/maven/

[12] Maven - Introduction, The Apache Software Foundation
https://maven.apache.org/what-is-maven.html

[13] IMiGEr, Research group Reliable Software Architectures.
https://github.com/ReliSA/IMiGEr

[14] Andy Cockburn, Amy Karlson, Benjamin B. Bederson. A Review of Overview+Detail, Zooming, and Focus+Context Interfaces. 2007.

[15] T.J. Jankun-Kelly, Kwan-Liu Ma. MoireGraphs: Radial Focus+Context. Mississippi State University, University of California, 2003.

[16] Repositories, GitHub Docs, GitHub, Inc.
https://docs.github.com/en/rest/reference/repos

[17] Resources in the REST API, GitHub Docs, GitHub, Inc.
https://docs.github.com/en/rest/overview/resources-in-the-rest-api#rate-limiting

[18] Heorhiy Byelas and Alexandru Telea. Visualization of areas of interest in software architecture diagrams. In Proceedings of the 2006 ACM symposium on Software visualization, SoftVis '06, pages 105–114, New York, NY, USA, 2006. ACM.

[19] IMiGEr raw input format, Tomáš Šimandl, September 2018.
https://github.com/ReliSA/IMiGEr/blob/devel/documents/IMiGEr/IMiGEr_raw_input_format.pdf

[20] Creating a personal access token, GitHub Docs, GitHub, Inc.
https://docs.github.com/en/github/authenticating-to-github/creating-a-personal-access-token

A Tabulka otestovaných repozitářů

Název repozitáře	Kompatibilní	Nekompatibilní	Redundantní	Nebezpečné	C1	C2	C3	M1	M2	F1	F2	MOD	M.M1	F7	Skóre
dustin-java-memcached-client	3	5	3	0	62	0	0	0	0	0	0	0	0	0	62
jruby-activerecord-jdbc-adapter	1	2	0	0	71	0	0	0	0	0	0	0	0	0	71
britt-github-java	4	2	1	0	34	1	0	2	0	0	0	0	0	0	56
mheath-adbcj	11	8	2	0	114	3	0	2	0	1	0	0	0	0	145
caillette-novelang	14	11	20	0	42	3	0	3	0	0	0	0	0	0	78
mojodna-searchable	7	3	1	0	39	0	0	0	0	0	0	0	0	0	39
soemirno-timesheets	1	0	5	0	0	0	0	0	0	0	0	0	0	0	0
n0ha-yui-compressor-ant-task	9	1	0	1	19	0	0	0	0	0	0	0	0	0	19
takai-jruby-dsl-example	1	1	1	0	19	0	0	0	0	0	0	0	0	0	19
credmp-maven-emacs-plugin	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0
torbjornvatn-guice-injector-tester	1	1	3	0	8	0	0	0	0	0	0	0	0	0	8
wireframe-testdox-maven-plugin	25	7	1	1	174	4	0	4	0	0	0	0	0	0	222
hns-ringojs	11	2	14	0	13	1	0	3	0	0	0	0	0	0	45
KenMacD-gunit	4	0	11	0	0	0	0	0	0	0	0	0	0	0	0
olamy-scm-git-test	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
brianm-mxfingerd	4	1	1	0	26	0	0	0	0	0	0	0	0	0	26
housejester-jsondumper	1	0	13	0	0	0	0	0	0	0	0	0	0	0	0
astubbs-testdox	4	2	2	0	38	0	0	0	0	0	0	0	0	0	38
jasonhudgins-hudgejar	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
const-etl-java	6	3	2	0	72	0	0	0	0	0	0	0	0	0	72
matschaffer-corundum	1	2	0	0	38	0	0	0	0	0	0	0	0	0	38
bobmcwhirter-jboss-naming	5	1	1	0	3	0	0	0	0	0	0	0	0	0	3
blaxter-simplegwtmvn	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cschmidt-middlewhere-demo	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
tcurdt-jdeb	31	9	11	0	83	0	0	0	0	0	0	0	0	0	83
tcurdt-jdependency	6	0	4	0	0	0	0	0	0	0	0	0	0	0	0
shs96c-webdriver	15	4	0	0	58	0	0	0	0	0	0	0	0	0	58
astubbs-wicket.get-portals2	1	0	7	0	0	0	0	0	0	0	0	0	0	0	0
virtix-webdriver	15	4	0	0	61	0	0	0	0	0	0	0	0	0	61
nigelsim-nsadapters	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
myabc-markdownj	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0
also-postal	2	1	0	0	3	0	0	0	0	0	0	0	0	0	3

tcurdt-drift	4	1	12	0	1	1	0	1	0	0	0	0	0	0	13
ekoneil-gnip-java	3	0	4	0	0	0	0	0	0	0	0	0	0	0	0
davidB-yuicompressor-maven-plugin	28	6	2	0	47	0	0	0	0	0	0	0	0	0	47
dkandalov-definilizer	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0
also-veneer-core	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
also-rjs-server	2	1	0	0	3	0	0	0	0	0	0	0	0	0	3
andyhot-maven-jawr-plugin	15	3	21	0	69	0	0	0	0	0	0	0	0	0	69
also-spring-flash-map	4	7	0	0	355	0	0	0	0	0	0	0	0	0	355
johnhampton-maven-scm	24	1	2	0	6	0	0	0	0	0	0	0	0	0	6
wadey-gnip-java	5	1	5	0	6	0	0	0	0	0	0	0	0	0	6
rubbish-jruby-fit	2	2	0	0	67	0	0	0	0	0	0	0	0	0	67
also-bind-annotations	3	7	0	0	372	0	0	0	0	0	0	0	0	0	372
3scale-labs-3scale_ws_api_for_java	5	3	5	1	6	2	0	0	3	0	0	0	0	0	40
Hejki-miniature-assistant	5	0	2	0	0	0	0	0	0	0	0	0	0	0	0
nirvdrum-maven-jython-plugin	5	1	13	0	25	0	0	0	0	0	0	0	0	0	25
born2snipe-flapjack	5	0	5	0	0	0	0	0	0	0	0	0	0	0	0
ralfebert-maven-instant-ws	7	2	2	0	9	0	0	0	0	0	0	0	0	0	9
batmat-overview-maven-plugin	29	4	16	1	161	2	0	2	0	0	0	0	0	0	185
gudmundur-atunit	2	1	1	0	9	0	0	0	0	0	0	0	0	0	9
thillerson-brightkite4j	7	2	2	2	7	1	0	1	0	0	0	0	0	0	19
bazhenov-envy	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0
steingrd-templatext	5	3	1	0	31	0	0	0	0	0	0	0	0	0	31
mikereedell-sunrisesunsetlib-java	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
akkumar-jreversepro	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0
dcrec1-rspec-maven-plugin	2	1	1	0	38	0	0	0	0	0	0	0	0	0	38
happygiraffe-jslint4java	2	1	5	0	7	0	0	0	0	0	0	0	0	0	7
Chouser-clojure-classes	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
masukomi-aspirin	6	1	3	0	3	0	0	0	0	0	0	0	0	0	3
rubbish-tdd-presentation	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0
rubbish-maven-warble-plugin	13	1	2	0	25	0	0	0	0	0	0	0	0	0	25
sethm-symon	0	1	7	0	53	0	0	0	0	0	0	0	0	0	53
kungfoo-geohash-java	3	0	2	0	0	0	0	0	0	0	0	0	0	0	0
stefanofornari-travian-world	3	3	3	0	30	0	0	0	0	0	0	0	0	0	30

nilsga-fitnesse-pom-widget	2	24	17	0	32	0	0	0	0	0	0	0	0	0	32
Reilithion-gridrover-experiments	3	5	1	0	52	0	0	0	0	0	0	0	0	0	52
lewiswang-fitnesse	2	7	0	0	39	2	0	2	1	0	0	0	0	0	73
jsampson-fractal-melodies	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
benjaminplee-tweetpublisher	3	1	1	0	134	0	0	0	0	0	0	0	0	0	134
junit-team-junit4	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0
dsaff-junit	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
davecheney-mdns-java	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
kms-sdr101-java	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
gwenn-sqlite-dialect	5	6	0	0	208	0	0	0	0	0	0	0	0	0	208
chanwit-groovy	0	1	24	0	0	0	0	0	0	0	0	0	0	0	0
knifhen-javarebel-stripes-plugin	3	3	0	0	74	0	0	0	0	0	0	0	0	0	74
datamapper-do	5	1	9	0	60	0	0	0	0	0	0	0	0	0	60
jafl-language_game	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
tobrien-cas-extend	0	1	8	0	0	0	0	0	0	0	0	0	0	0	0
purcell-jargs	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0
rogerleite-doit	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0
gravityrail-databinder	2	2	0	0	45	0	0	0	0	0	0	0	0	0	45
mheath-webbolo	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0
aslakjo-yrapi	1	1	1	0	6	0	0	0	0	0	0	0	0	0	6
fiveruns-dash-java	1	3	1	0	7	0	0	0	0	0	0	0	0	0	7
frsyuki-festivoice	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0
kohsuke-embedded_su4j	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
davidB-winstone-maven-plugin	0	14	1	0	0	0	0	0	0	0	0	0	0	0	0
yestech-yeswebtop	0	2	30	0	0	0	0	0	0	0	0	0	0	0	0
soemirno-xmlvalidator	2	3	6	0	60	0	0	0	0	0	0	0	0	0	60
antlibs-ant-cpptasks	0	3	3	0	0	0	0	0	0	0	0	0	0	0	0
yestech-yesadmin	0	2	31	0	0	0	0	0	0	0	0	0	0	0	0
yestech-yeswizard	1	2	5	0	0	1	0	0	0	1	0	0	0	0	7
yestech-jmlnitrate	4	4	4	0	150	1	0	0	0	1	0	0	0	0	157
dekellum-rjack	3	5	54	14	23	28	0	18	12	9	0	0	0	0	424
rbarooah-s3lib-java	2	5	1	0	14	0	0	0	0	0	0	0	0	0	14
bobmcwhirter-ovirtex	1	2	1	0	3	0	0	0	0	0	0	0	0	0	3

dashorst-wicket-stuff-markup-validator	0	3	2	0	0	0	0	0	0	0	0	0	0	0	0
kohsuke-jinterop-proxy	1	4	1	0	13	0	0	0	0	0	0	0	0	0	13
raykrueger-rulespec	2	3	1	0	45	0	0	0	0	0	0	0	0	0	45
rickyclarkson-jline	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
xfire-jitertools	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
lionelarmanet-wiquery	2	6	1	0	49	0	0	0	0	0	0	0	0	0	49
dekellum-gravitext	3	15	7	0	61	0	0	0	0	0	0	0	0	0	61
dima767-inspektr	2	5	3	0	17	0	0	0	0	0	0	0	0	0	17
we4tech-java-open-search-servlet	1	2	2	0	29	0	0	0	0	0	0	0	0	0	29
frankbille-watzy	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1
rhdev-example-camel-jboss	3	5	1	0	22	0	0	0	0	0	0	0	0	0	22
jmbe-junit-max-test	0	1	7	0	0	0	0	0	0	0	0	0	0	0	0
dsummersl-slimtimer4j	3	7	3	0	114	0	0	0	0	0	0	0	0	0	114
yestech-yescontact	0	1	6	0	0	0	0	0	0	0	0	0	0	0	0
rhdev-example-camel-jboss-spring	7	8	1	1	235	0	0	0	0	0	0	0	0	0	235
fellix-javarecord	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0
adamw-jsr308-maven2-plugin	4	24	15	0	20	7	0	10	0	4	0	0	0	0	154
matthewd-do	5	1	0	0	40	0	0	0	0	0	0	0	0	0	40
gvaughn-yuicompressor-maven-plugin	2	16	2	0	14	0	0	0	0	0	0	0	0	0	14
retoo-bodesuri	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
we4tech-folder-content-guard	0	1	11	0	0	0	0	0	0	0	0	0	0	0	0
umefjord-maven-overview-plugin	4	29	16	1	161	2	0	2	0	0	0	0	0	0	185
janhenrik-blameitknowit	4	7	1	0	44	0	0	0	0	0	0	0	0	0	44
sps-beanstalk-client	3	1	5	0	70	0	0	0	0	0	0	0	0	0	70
avh4-avh4-util	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0
prb-bigbird	0	6	4	3	0	0	0	0	0	0	0	0	0	0	0
ashlux-maven-oc4j-plugin	0	3	5	0	0	0	0	0	0	0	0	0	0	0	0
neotyk-secure-grizzly-http-gitorial	0	6	3	0	0	0	0	0	0	0	0	0	0	0	0
robcos-onthefly	2	3	1	0	7	3	0	7	0	0	0	0	0	0	83
caelum-vraptor	21	39	7	0	387	6	0	13	3	0	0	0	0	0	559
BusTaR-tuxftp	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
ogrodnek-java_fathom	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0
qmx-rsync4j	0	2	12	0	0	0	0	0	0	0	0	0	0	0	0

wsmoak-abc-example	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
aslakhellesoy-ansicolor	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
krosenvold-spring-context-analyzer	5	3	3	0	169	0	0	0	0	0	0	0	0	0	169
apache-tapestry3	7	11	1	0	104	1	0	2	0	0	0	0	0	0	126
alphazero-jredis	4	5	7	0	140	0	0	0	0	0	0	0	0	0	140
michaellperry-tuple	0	2	2	0	0	0	0	0	0	0	0	0	0	0	0
mallardsoft-tuple	0	3	1	0	0	0	0	0	0	0	0	0	0	0	0
benjaminplee-yardproxy	1	3	1	0	0	1	0	0	0	1	0	0	0	0	7
chrsan-css-selectors	0	1	5	0	0	0	0	0	0	0	0	0	0	0	0
mrdon-cargo-maven2-plugin-db	4	22	18	0	45	4	0	0	4	0	0	0	0	0	93
csmuc-activerecord-jdbc-adapter	1	1	0	0	25	0	0	0	0	0	0	0	0	0	25
yestech-yestest	3	9	19	1	31	4	0	0	68	1	0	0	68	0	1404
jdubiner-joeltest01	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
ecomba-moola-j	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
varmaan-HtmlUnit	4	13	18	0	38	2	0	3	1	0	0	0	1	0	92
grzegorzblaszczyk-transcoder	1	4	0	0	32	0	0	0	0	0	0	0	0	0	32
genki-do	1	7	1	0	25	0	0	0	0	0	0	0	0	0	25
betarelease-dbdeploy	0	6	3	0	0	0	0	0	0	0	0	0	0	0	0
kohsuke-loopy	2	1	0	0	23	0	0	0	0	0	0	0	0	0	23
joshvh-ruote	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
jexp-embeddedtestrunner	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
schloepke-atomify	1	6	14	0	27	0	0	0	0	0	0	0	0	0	27
timescro-ruote	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0
zsjoska-gagarin	1	3	0	0	30	0	0	0	0	0	0	0	0	0	30
eee-c-couchdb-lucene	9	20	4	0	145	4	0	5	2	0	0	0	1	0	233
netcompanyno-workmanship-oo	0	1	3	0	0	0	0	0	0	0	0	0	0	0	0
mohlendo-tomcatpolicyprotocol	1	5	1	0	7	0	0	0	0	0	0	0	0	0	7
jexp-xmldsl	1	2	1	0	6	0	0	0	0	0	0	0	0	0	6
luontola-tdd-tetris-tutorial	0	1	9	0	0	0	0	0	0	0	0	0	0	0	0
kohsuke-hadoop-importer	3	1	1	0	171	0	0	0	0	0	0	0	0	0	171
hartmann-bidragsveilederwebdriver	0	1	25	0	0	0	0	0	0	0	0	0	0	0	0
sonatype-JGit	2	3	2	0	2	0	0	0	0	0	0	0	0	0	2
jcblitz-MavenJRubyCucumber	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0

gjoseph-WiseIO	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0
junit-junit	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0
apache-guava	1	0	7	0	0	0	0	0	0	0	0	0	0	0	0
MIT-mockito	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0
apache-commons-io	1	0	16	0	0	0	0	0	0	0	0	0	0	0	0
logback-logback-classic	6	3	47	0	121	2	0	2	0	0	0	0	0	0	145

B Struktura přiloženého CD

- adresář Depend - Obsahuje potřebné závislosti pro sestavení serverové části včetně upraveného nástroje IMiGEr.
- adresář Javadoc - Obsahuje dokumentaci vytvořenou nástrojem *Javadoc*.
- adresář Sources - V tomto adresáři se nachází všechny zdrojové kódy potřebné k překladu a spuštění aplikace. Dále se v adresáři nachází spustitelný JAR soubor aplikace.
- adresář Text - Obsahuje text bakalářské práce ve formátu PDF a zdrojový kód pro jeho vytvoření.