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

Bakalářská práce

Strategie pro efektivní získávání dat z Maven repository

Plzeň 2017 Zdeněk Valeš

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 27. června 2017

Zdeněk Valeš

Abstract

The aim of this thesis is to properly analyze efficiency of possible search strategies which can be used when locating artifacts in maven repositories by given criteria. Chosen strategies will then be implemented in a plugin for CRCE which will be able to locate an artifact, download it, extract its metadata and save it to the CRCE storage. Said plugin will be created with attention paid to scalability and good code readability.

Abstrakt

Cílem této práce je analyzovat efektivitu možných strategií prohledávání maven repozitářů a následně vytvořit rozšíření pro systém CRCE, které bude schopné automaticky vyhledávat artefakty podle zadaných kritérií. Rozšíření bude nalezené artefakty stahovat, provede indexaci jejich metadat a následně je nahraje do úložiště systému CRCE. Modul bude implementován s ohledem na budoucí rozšiřitelnost a dobrou čitelnost kódu.

Obsah

1 Úvod 7

2 Nástroj Maven 8

2.1 Maven artefakt 8

2.1.1 Project Object Model 9

2.1.2 Dědičnost artefaktů 10

2.1.3 Životní cyklus artefaktu 11

2.2 Maven repozitář 12

2.2.1 Vzdálený repozitář 12

2.2.2 Centrální repozitář 12

2.2.3 Lokální repozitář 12

2.3 Dostupné prohledávací služby 13

2.3.1 REST API 13

2.3.2 Nexus indexer 17

2.4 Služby dostupné k získání artefaktu 18

2.4.1 Aether 18

3 Úložiště komponent CRCE 22

3.1 Architektura CRCE 22

3.1.1 OSGi 23

3.1.2 Tvorba modulů pro CRCE 23

3.2 Metadata komponent 24

3.3 Komponenta v CRCE 24

4 Prohledávání maven repozitářů 26

4.1 Možné případy užití 26

4.2 Strategie prohledávání 27

4.2.1 Analýza strategií 27

4.2.2 Navržené strategie 31

5 CRCE modul 32

5.1 Specifikace 32

5.1.1 Základní funkcionality 32

5.1.2 Interakce s uživatelem 33

5.1.3 Grafické rozhraní 33

5.2 Popis implementace 33

5.2.1 Konfigurace 33
5.2.2 Popis tříd 34
5.2.3 Popis vyhledávání 39
5.3 Testování 41
5.3.1 Testovací sestava 41
5.3.2 Testovací scénáře 41
5.3.3 Výsledky testů 44
5.4 Použité technologie 46

6 Závěr 47

Literatura 48

Seznam zkratek 50

Příloha A - UML diagram vytvořeného modulu 51

Příloha B - uživatelský manuál 52

1 Úvod

Vývoj komplexních systémů s sebou nese nutnost udržovat co možná největší konzistenci prostředí, zvláště pak jedná-li se o práci v početných týmech. Jedním z dobrých zvyků při práci na rozsáhlejších projektech je automatizace překladu, sestavení, testování a dalších procesů nutných k uvedení systému do produkce.

Existuje celá řada automatizačních nástrojů pro různé jazyky. Java je jedním z populárních jazyků současnosti se širokým polem působnosti. Právě pro Javu je primárně vyvíjen nástroj Maven, spravovaný komunitou Apache. Pomocí Mavenu lze pohodlně spravovat závislosti i jednotlivé životní cykly projektu, a výsledek poskytnout jako ucelenou komponentu - artefakt. Takto zpracované komponenty mohou být zpřístupněny ve veřejných či soukromých repozitářích, kde je umožněno také průběžné verzování.

Tato práce se zabývá vyhledáváním konkrétních artefaktů v repozitářích a jejich integrací do systému CRCE[1], vyvíjeného skupinou Relisa na Katedře Informatiky Západočeské Univerzity v Plzni. Systém CRCE slouží jako úložiště komponent, které dokáže kontrolovat jejich vzájemnou kompatibilitu. V současné době nicméně chybí integrace s Maven repozitáři a není možné do systému automaticky začleňovat artefakty z těchto repozitářů.

Cílem práce je vytvořit funkční rozšíření systému CRCE, které by tento nedostatek odstranilo a umožnilo automatické vyhledání komponent v Maven repozitářích, jejich stažení a následné zařazení do CRCE. V následujících kapitolách budou rozebrány základní vlastnosti obou výše zmíněných systémů, provedena analýza modulu na vyhledávání a indexaci maven artefaktů, navrženy možné přístupy k řešení dané problematiky a nakonec popsána vlastní implementace.

[1] Component Repository supporting Compatibility Evaluation

2 Nástroj Maven

Maven je komplexní nástroj pro správu a vývoj projektů, založený na konceptu Project Object Model (dále jen POM). Mezi jeho hlavní vlastnosti patří zejména správa závislostí projektu a konfigurace životního cyklu.

Jeho konfigurace pokrývá správu závislostí a celý životní cyklus projektu, od kompilace až k nasazení, nebo nahrání na server. Fázi sestavení lze navíc rozšířit o další akce pomocí existujících, nebo vlastních pluginů (například nahrání testovacích dat do databáze). Výsledkem úspěšného průchodu tímto cyklem je tzv. Maven artefakt.

2.1 Maven artefakt

Maven artefakt (dále jen artefakt) je základní komponenta systému Maven. Jeho unikátní identifikace je tvořena řetězcem obsahujícím tyto atributy (tzv. koordináty): `groupId:artifactId:packaging:classifier:version`[4]. Koordináty `packaging` a `classifier` jsou nepovinné a ve většině případů se používá zkrácená verze jména artefaktu, která obsahuje pouze `groupId`, `artifactId` a `version`.

Koordinát `packaging` určuje typ artefaktu: `pom`, `jar`, `war`, `maven-plugin`, `ejb`, `ear`, `rar`, `par`[4]. Různé typy se mohou lišit fázemi, kterými během životního cyklu projdou. Kromě výše uvedených standardních typů je možné použít libovolné vlastní. V tabulce 2.1 je uveden stručný popis těchto typů.

Artefakty jsou typicky distribuovány skrze repozitáře a k jejich použití v projektu stačí přidat závislost v podobě XML elementu do modelu projektu, který je tvořen XML souborem. Maven pak daný artefakt vyhledá v nakonfigurovaném repozitáři, stáhne a automaticky přidá do classpath projektu. Tímto způsobem jsou jednotlivé funkční bloky skládány do rozsáhlých celků. Nicméně, je třeba brát v potaz vzájemnou kompatibilitu jednotlivých artefaktů, kterou Maven nijak neřeší a zůstává tedy na vývojáři, aby dbal na použití kompatibilních verzí.

Každý artefakt je popsán výše zmíněným systémem POM. Tento popis je realizován povinným souborem `pom.xml`, který je detailněji rozebrán v následující části. Modul, který je výsledkem této práce, bude za indexovatelný artefakt považovat JAR soubor obsahující `pom.xml`, ze kterého budou načtena potřebná metadata.

Název typu	Popis
pom	Slouží k uchování metadat a sdružení více modulů.
jar	Výchozí typ.
war	Slouží k distribuci webových aplikací.
ejb, ear	Slouží k označení Enterprise JavaBeans (EJB), respektive Enterprise Application (EAR) modulů.
par	Slouží k označení Pearl Archive Toolkit modulů.
rar	Slouží k označení Resource Adapter Archive (RAR) modulů.

Tabulka 2.1: Popis typů artefaktu

2.1.1 Project Object Model

POM tvoří XML reprezentaci daného projektu, která je uložena v souboru `pom.xml`[4]. Zde je prováděna veškerá konfigurace projektu, včetně popisu sestavení a případného exportu (jar, war...).

Kořenovým elementem POM je `project`, který obsahuje XML namespace a odkaz na XML Schema Definition (dále jen XSD) POM. XML konfigurace musí dále povinně obsahovat verzi POM a identifikační atributy. Příklad velmi jednoduchého POM je uveden na obrázku 2.1.

```
1 <project>
2     <modelVersion >4.0.0</modelVersion>
3     <groupId>org.zcu.thesis</groupId>
4     <artifactId>thesis</artifactId>
5     <version>0.1</version>
6     <packaging>jar</packaging>
7 </project>
```

Obrázek 2.1: Příklad jednoduchého POM

modelVersion Atribut udává verzi POM. V době psaní práce je nejnovější (a jediná podporovaná) verze `4.0.0`, verze `5.0.0` je zatím ve stádiu vývoje.

groupId Atribut tvoří první část jména artefaktu. Typicky je unikátní na úrovni organizace, nebo vývojářského týmu. Nemá žádný závazný formát, ačkoliv je dobrým zvykem vyvodit `groupId` z balíkové struktury projektu, kde jsou názvy jednotlivých balíků odděleny tečkou.

artifactId Atribut tvoří druhou část jména artefaktu a představuje jméno samotného projektu. Opět nemá žádný závazný formát, ale je dobré nepoužívat mezery (ideálně nahrazovat pomlčkou).

version Atribut udává verzi projektu. Je možné, aby v repozitáři bylo více verzí stejného projektu, proto se dá o verzi uvažovat jako o jakési časové značce. Verze opět nemá žádný závazný formát, ale je dobrým zvykem používat konvenci `major.minor.micro-qualifier`.

packaging Nepovinný údaj, výchozí hodnota je `pom`. Detailnější popis tohoto koordinátu je zmíněn výše.

2.1.2 Dědičnost artefaktů

Artefakty, zejména v případě rozsáhlejších projektů, často sdílejí určitou část své konfigurace, nebo životního cyklu. Je tedy například žádoucí, aby bylo možné v jednom souboru nakonfigurovat sestavovací fázi, kterou projdou všechny moduly, z nichž je projekt složen. Koncept POM nabízí možnost jednotlivé artefakty (respektive jejich POM) dědit a vytvářet tak potřebnou konfigurační hierarchii[4].

Každý POM je implicitně odděděný od takzvaného super POM[4]. Jedná se o základní konfigurační soubor, který obsahuje nastavení cest v projektu, nastavení základních překladových a sestavovacích procesů a také adresu globálního maven repozitáře, ve kterém budou vyhledávány případné závislosti. Všechna zmíněná nastavení lze ve odděděném POM překrýt, případně rozšířit (například použití soukromého maven repozitáře). Tento základní POM umožňuje konfiguraci vlastního POM pouze s nezbytným minimem (viz obrázek 2.1).

Dědění POM nemusí být pouze dvouúrovňové (super POM - vlastní POM), naopak se v projektech často používá řetězení POM (multi-module projekty). V takovém případě není nutné uvádět všechny identifikační atributy. Většinou se uvádí pouze `artifactId`, `groupId` i `version` pak bývají společné pro celý projekt[4].

Na obrázku 2.2 je uveden příklad deklarace dědění od rodičovského POM.

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4.                       https://maven.apache.org/xsd/maven-4.0.0.xsd">
5.   <modelVersion>4.0.0</modelVersion>
6.
7.   <parent>
8.     <groupId>org.codehaus.mojo</groupId>
9.     <artifactId>my-parent</artifactId>
10.    <version>2.0</version>
11.    <relativePath>../my-parent</relativePath>
12.  </parent>
13.
14.  <artifactId>my-project</artifactId>
15. </project>
```

Obrázek 2.2: Příklad deklarace dědičnosti

2.1.3 Životní cyklus artefaktu

Jedna z částí POM představuje popis sestavovacího životního cyklu. Tento cyklus je rozdělen do několika fází ve kterých lze konfigurovat konkrétní akce (typicky pomocí pluginů). Výhoda zde spočívá v deklarativním přístupu – v případě použití skriptů, které projekt přeloží a sestaví, nemusí být zaručena přenositelnost ani znovupoužitelnost na více projektů a pokud ano, mohou být tyto skripty poměrně komplikované. Místo nich stačí v POM deklarovat pluginy (v nejjednodušším případě není nutné ani to), které podle konfigurace provedou nutné akce nad zpracovávaným projektem[3]. V případě potřeby je možné nastavit vynechání konkrétních pluginů a to buď v samotném `pom.xml`, nebo skrze parametr v konzoli.

Zde je výčet a popis základních fází (dohromady je fází 29[3], z úsporných důvodů nejsou uvedeny všechny). Fáze jsou seřazeny vzestupně a při vyvolání jedné z fází se automaticky provedou všechny předchozí:

validate V této fázi je kontrolován POM a ověřuje se dostupnost všech informací nutných k sestavení projektu.

compile V této fázi dojde ke kompilaci zdrojových kódů projektu.

test V této fázi se spustí testy nad zkompilovaným kódem. Pro testy je typicky použit framework JUnit, ale lze nakonfigurovat i jiný.

package V této fázi jsou zkompilované kódy zabaleny do snadno přenosného formátu – typicky JAR.

verify V této fázi jsou zkontrolovány výsledky integračních testů.

install V této fázi je artefakt nainstalován do lokálního repozitáře (více o repozitářích v kapitole 2.2),

deploy V této (finální) fázi je artefakt nahrán do nakonfigurovaného repozitáře, odkud může být dále používán.

Jednotlivé fáze lze vyvolat příkazem `mvn <název fáze>`.

2.2 Maven repozitář

Repozitáře slouží k průběžnému ukládání a distribuci vytvořených artefaktů. Existují dva typy repozitářů – lokální a vzdálený[5], oba budou popsány níže. Struktura repozitáře je přímočará – jedná se o adresářovou hierarchii složenou z identifikátorů artefaktu (`groupId`, `artifactId`, `version`).

2.2.1 Vzdálený repozitář

Vzdálený repozitář je pojem představující jakékoliv úložiště maven artefaktů. Ke stažení artefaktu ze vzdáleného repozitáře je nutné přidat závislost do POM projektu.

Pokud maven nenajde požadovaný artefakt v lokálním repozitáři, začne prohledávat všechny nakonfigurované vzdálené repozitáře. Implicitně je pro vyhledávání a stahování artefaktů nastaven centrální repozitář mavenu.

2.2.2 Centrální repozitář

Centrální repozitář je největším existujícím, veřejně přístupným maven repozitářem[1]. Tato práce bude vyvíjena primárně pro centrální repozitář, avšak s ohledem na případnou rozšiřitelnost na další repozitáře (například privátní).

2.2.3 Lokální repozitář

Lokální repozitář slouží jako cache, kam maven ukládá artefakty referencované v POM a lokálně vytvořené artefakty[5] (`install` fáze životního cyklu).

Cesta ke složce s lokálním úložištěm je nastavena v konfiguračním souboru mavenu (ne `pom.xml`) a typicky vede do složky `.m2` v domovském adresáři.

Lokální repozitář nemusí být nutně umístěn na lokálním počítači. Maven podporuje deklaraci lokálního repozitáře například na firemním serveru. Jedná se tedy o poddruh vzdáleného repozitáře.

2.3 Dostupné prohledávací služby

Aby bylo možné repozitáře efektivně používat, existují služby sloužící k jejich prohledávání. Tyto jsou typicky založeny na indexu, který je vytvořen pro každý repozitář a službám následně zpřístupněn. Informace uvedené v indexech se mohou mezi repozitáři lišit, to může představovat problém při použití pokročilejších metod vyhledávání (například při hledání podle celého jména třídy).

Jednou z možností prohledávání (pokud ji provozovatel repozitáře nabízí) je použití REST API. Největší výhodou této možnosti je její jednoduchost – v zásadě stačí zavolat API a rozparsovat vrácený JSON. Nevýhodou může být omezenost vyhledávání, případně úplná nedostupnost API, pokud ji provozovatel repozitáře nenabízí.

Obecnější alternativu k REST API představuje Nexus Indexer. Ten je za použití Javy schopný prohledávat libovolný Maven repozitář[7] a odpadá zde potřeba konverze získaných objektů do Javy (není tedy nutné ručně parsovat jakékoliv JSON, nebo XML struktury). Nevýhodou je nutnost udržovat aktuální index daného repozitáře. Počáteční stažení indexu je časově náročné a inkrementální updaty vyžadují konfiguraci časování (každý den, týden, měsíc...). Jednotlivé repozitáře navíc svůj index updatují v různých časových intervalech[7].

2.3.1 REST API

Společnost Sonatype provozuje službu, která umožňuje prohledávat centrální Maven repozitář pomocí REST API. Jedná se o stejnou společnost, která vyvíjela původní Nexus Indexer a později jej věnovala komunitě kolem mavenu [11]. Vyhledávač je dostupný na adrese `search.maven.org` a kromě veřejných REST API nabízí i klasické vyhledávání s možností uložení do záložek (takzvané bookmarkable URL).

Representational State Transfer (REST) je architektonický návrh webové služby postavený na protokolu http, který stanovuje jednotné komunikační rozhraní mezi členy distribuovaného systému (například sítě Internet). Toto rozhraní se skládá ze základních operací *GET*, *PUT*, *UPDATE*, *DELETE*,

které umožňují provádět akce nad libovolnými objekty v systému. Objekt je jednoznačně identifikovaný pomocí Uniform Resource Identifier (URI) a je popsán standardním datovým formátem (typicky XML, JSON, HTML...). Komunikace spočívá ve výměně reprezentací těchto objektů mezi jednotlivými členy systému[17].

REST API, které je popsáno v následujících sekcích, umožňuje reprezentovat objekty ve formátech JSON a XML. V práci jsem použil formát JSON, který je pro mě lépe čitelnější než XML a tím mi usnadňuje manuální testování. JSON je textový formát sloužící k serializaci strukturovaných dat, skládající se z párů *název:hodnota*, kde *hodnota* reprezentuje objekt, pole, číslo, řetězec, nebo booleovskou hodnotu (`true`, `false`)[8].

Díky jednoduchosti a dostatečné funkčnosti jsem tuto službu zvolil jako primární pro mou práci.

Popis API

REST API umožňují vyhledávání jak podle kombinace mavenovských kooridnátů, tak podle checksum artefaktu, tříd (celé i částečné jméno) obsažených v artefaktu a několika dalších pokročilých možností.

Vyhledávač používá vlastní formát jednoduchých dotazů, které jsou uvedeny jako URL parametr v adrese. Základní URL pro použití těchto API je `http://search.maven.org/solrsearch/select`. V tabulce 2.2 je uveden seznam možných URL parametrů.

Název parametru	Popis	Možné hodnoty
q	Vyhledávací dotaz, specifikován níže.	–
wt	Formát výsledku, výchozí hodnota je json	json, xml
rows	Maximální počet nálezů, který server vrátí. Pokud není specifikován, vrátí všechny nálezy.	Kladné číslo
start	Pořadí artefaktu od kterého začne listování výsledků. Užitečné ve spojení s parametrem rows.	Číslo větší, nebo rovné 0.

Tabulka 2.2: URL parametry REST API

Samotný dotaz se skládá z jednoduchých klauzulí `klíč:"hodnota"`, které

jsou provázány logickými spojkami `AND` a `OR`. Dotaz na vyhledávání artefaktů podle koordinátů *groupId* a *artifactId* by tedy vypadal následovně: `q=g:"org.hibernate"+AND+a:"hibernate-core"`. V tabulce 2.3 je uveden popis klíčů, ze kterých je možné dotaz sestavit.

Název klíče	Popis
g,a,v	Mavenovské koordináty.
l	Klasifikátor, který maven používá k označení obsahu artefaktu.
p	Typ artefaktu.
c	Jméno třídy (pouze třídy), kterou má nalezený artefakt obsahovat. Nemusí být doslovné, vyhledávač pracuje s podobnými řetězci.
fc	Celé jméno třídy, kterou má artefakt obsahovat. Nemusí být doslovné, vyhledávač pracuje s podobnými řetězci.
1	SHA-1 kontrolní součet artefaktu.
tags	Vyhledávání podle tagů artefaktu.

Tabulka 2.3: Parametry dotazu

Popis získaného JSON objektu

Služba vrací na dotaz odpověď obsahující objekt ve formátu JSON, jehož struktura je popsána níže. Z úsporných důvodů jsou uvedeny jen nejdůležitější části.

```
{
  ▸ responseHeader: { … },
  ▸ response: { … },
  ▸ spellcheck: { … }
}
```

Obrázek 2.3: Elementy v kořenu objektu

Vrácený objekt obsahuje tři elementy (viz obrázek 2.3). Element *responseHeader* obsahuje informace o volání api a vstupních parametrech (element *params* na obrázku 2.4) a kromě elementu *status*, který určuje výskyt chyby, není pro vyhledávání důležitý.

Naopak velmi důležitý pro hledání je element *response* (obrázek 2.5), respektive *docs*, který obsahuje informace o nalezených artefaktech. Za po-

```
▾ responseHeader: {
      status: 0,
      QTime: 0,
    ▸ params: { … }
  },
```

Obrázek 2.4: Elementy responseHeader

všimnutí stojí pole *numFound*, které udává počet reálně nalezených artefaktů. Skutečný počet vrácených artefaktů ale může být omezený parametrem *rows*. Struktura objektů v poli *docs* je poměrně přímočará.

```
▾ response: {
      numFound: 517,
      start: 0,
    ▾ docs: [
        ▾ {
              id: "com.google.cloud.trace:guice",
              g: "com.google.cloud.trace",
              a: "guice",
              latestVersion: "0.3.2",
              repositoryId: "central",
              p: "pom",
              timestamp: 1490735505000,
              versionCount: 8,
            ▾ text: [
                  "com.google.cloud.trace",
                  "guice",
                  ".pom"
              ],
            ▾ ec: [
                  ".pom"
              ]
          }
      ]
  },
```

Obrázek 2.5: Elementy response

Příklad použití

V následujícím příkladě bude ilustrováno použití REST API k vyhledání všech artefaktů, které obsahují balík `org.hibernate.collection`. Od 5. nalezeného artefaktu zobrazíme maximálně pouze 2 další.

Vyhledávací dotaz bude: `fc:"org.hibernate.collection"` a po přidání všech parametrů bude část, která se připojí k základní URL:

```
?q=fc:"org.hibernate.collection"&rows=2&start=4&wt=json
```

Element *response*, který bude JSON, vrácený na předchozí dotaz, obsahovat je znázorněn na obrázku 2.6. Z 666 artefaktů (počet je uvedený ke dni 21.6. 2017) vyhovujícím původnímu zadání byly vráceny pouze dva. Tento způsob, jak výsledný soubor stránkovat, bude v práci použit ke zlepšení efektivity.

```
▾ response: {
      numFound: 666,
      start: 4,
    ▾ docs: [
        ▾ {
              id: "hibernate:hibernate:3.0beta4b",
              g: "hibernate",
              a: "hibernate",
              v: "3.0beta4b",
              p: "jar",
              timestamp: 1131487365000,
            ▾ ec: [
                  ".jar",
                  ".pom"
              ]
          },
        ▾ {
              id: "hibernate:hibernate:3.0beta3",
              g: "hibernate",
              a: "hibernate",
              v: "3.0beta3",
              p: "jar",
              timestamp: 1131487365000,
            ▾ ec: [
                  ".jar",
                  ".pom"
              ]
          }
      ]
  },
```

Obrázek 2.6: Element response obsahující nalezené artefakty

2.3.2 Nexus indexer

Nexus Indexer je nástroj sloužící k sestavování a prohledávání indexů mavenovských repozitářů. Díky dostupnému API pro jazyk Java je indexer možné

využít ve vlastních projektech k integraci s maven repozitáři. K práci s indexem (zejména k jeho prohledávání) je interně použita knihovna Apache Lucene[7].

Pracuje-li indexer s cizím repozitářem, je třeba prvotního stažení a inicializace indexu. To představuje časově náročnou operaci (řádově minuty), proto je nutné provést inicializaci jednou a dále stahovat pouze inkrementální updaty, jejichž četnost je různá pro každý repozitář (pro centrální například jednou týdně). Kvůli této skutečnosti a příliš rozsáhlé funkcionalitě API jsem se rozhodl upustit od použití indexeru, nicméně při rozšíření pluginu pro další repozitáře bude podobného indexeru s velkou pravděpodobností třeba.

2.4 Služby dostupné k získání artefaktu

Prohledávací služby typicky nevrací samotné artefakty, ale pouze jejich popisná data, která je pak možno využít k získání artefaktů. Výhoda tohoto přístupu spočívá v možnosti filtrovat popisná data ještě před stahováním artefaktů, čímž se šetří čas i přenesená data.

Při využití vyhledávání pomocí REST API je možné sestavit HTTP odkaz pro stažení artefaktu. Ruční stahování souboru s artefaktem by ovšem kromě implementace samotného asynchronního stažení vyžadovalo navíc dočasné úložiště a ošetření výpadků spojení. Bylo by tedy potřeba implementovat vlastního download managera. Vzhledem k existenci knihovny Aether, která umožňuje přímé stažení artefaktu a výše popsané problémy interně řeší, je možnost ručního stahování nevýhodná a nebude tedy použita. Namísto ní bude v práci primárně použita knihovna Aether, která je detailněji popsána v následující části.

2.4.1 Aether

Aether je knihovna pro jazyk Java, pomocí které lze do aplikace integrovat funkcionalitu systému Maven (nejen získání, ale i případný push artefaktu do repozitáře). Jádro knihovny nabízí pouze základní, generický přístup k artefaktům a většina dalších funkcí je implementována rozšířeními Aetheru [9]. Je nutné brát v potaz, že knihovna primárně slouží pouze k získání artefaktů z repozitáře, nikoliv k jejich vyhledání.

Repository system

Jednou z první věcí, které jsou nutné k použití Aetheru v aplikaci, je inicializace repository system. Implementace Aetheru se skládá z několika komponent, které je nutné propojit, aby mohl být repository system inicializován. Komponenty jsou mezi sebou propojeny podle návrhového vzoru Service Locator a příklad jejich propojení je uveden v útržku kódu na obrázku 2.7.

```
72  private static RepositorySystem newRepositorySystem() {
73      DefaultServiceLocator locator = MavenRepositorySystemUtils
74                                      .newServiceLocator();
75      locator.addService(RepositoryConnectorFactory.class,
76              BasicRepositoryConnectorFactory.class);
77      locator.addService( TransporterFactory.class,
78              HttpTransporterFactory.class );
79      locator.addService( TransporterFactory.class,
80              FileTransporterFactory.class );
81
82      return locator.getService(RepositorySystem.class);
83  }
```

Obrázek 2.7: Inicializace repository systému

Takto inicializovaný repository system se bude skládat ze základního repository connectoru, komponenty pro přístup k repozitářům přes `file:` URL (typicky lokální maven repozitář) a komponenty pro přístup k repozitářům přes `http:`, nebo `https:` URL.

Repository system session

Komponenty Aetheru jsou bezstavové, to znamená, že veškerá nastavení musí být při každém požadavku na repository system předána skrze parametry metod. Pokud budeme pracovat například jen s centrálním repozitářem, nebudou se tato nastavení příliš měnit a jejich opakování je tedy zbytečné. Právě k uchování a znovupoužití nastavení slouží repository system session. Příklad inicializace session je uveden v útržku kódu na obrázku 2.8.

Lokální repozitář musí být u repository system session vždy inicializován. Při použití modulu `maven-settings-builder` lze cestu k lokálnímu repozitáři načíst z konfiguračního souboru `settings.xml`. Do nakonfigurovaného lokálního repozitáře jsou pak staženy všechny artefakty získané pomocí této session.

```
91   private static RepositorySystemSession newSession(
92           RepositorySystem system,
93           String localRepositoryPath)
94   {
95       DefaultRepositorySystemSession session =
96               MavenRepositorySystemUtils.newSession();
97       LocalRepository localRepo =
98               new LocalRepository(localRepositoryPath);
99       session.setLocalRepositoryManager(system
100              .newLocalRepositoryManager(session, localRepo));
101
102      return session;
103  }
```

Obrázek 2.8: Inicializace repository system session

Získání artefaktu

Po zinicializování repository system a vytvoření příslušné session je možné provést požadavek na repozitář a získat z něj artefakt. Na útržku kódu na obrázku 2.9 je zobrazen příklad stažení artefaktu z repozitáře podle třech koordinátů.

```
305  public Artifact resolve(String groupId, String artifactId,
306                          String version) {
307      Artifact artifact = new DefaultArtifact(groupId,
308              artifactId,
309              extension: "jar",
310              version);
311
312      ArtifactRequest artifactRequest = new ArtifactRequest();
313      artifactRequest.setArtifact(artifact);
314      artifactRequest.setRepositories(repositories);
315
316      ArtifactResult artifactResult = null;
317      try {
318          artifactResult = repositorySystem.resolveArtifact(session,
319                  artifactRequest);
320      } catch (ArtifactResolutionException e) {
321          return null;
322      }
323      return artifactResult.getArtifact();
324  }
```

Obrázek 2.9: Získání artefaktu

Dotaz na repozitář je reprezentován třídou `ArtifactRequest`, jedná se o dotaz na právě jeden artefakt podle koordinátů `groupId`, `artifactId` a

`version`. Řetězec `jar` udává hodnotu koordinátu `packaging` - na příkladu v obrázku je stahován celý artefakt, v případě, že by hodnota byla například `pom`, byl by stažen pouze soubor pom.xml, který danému artefaktu náleží.

Objektu, jež reprezentuje dotaz, je dále předán seznam repozitářů, které budou prohledány a následně dojde k provedení dotazu pomocí repository system session. Výsledkem je instance třídy `ArtifactResult`, ze které je možné získat stažený artefakt. Konkrétní soubor se nachází v lokálním repozitáři, který byl nakonfigurovaný při inicializaci repository system session (obrázek 2.8).

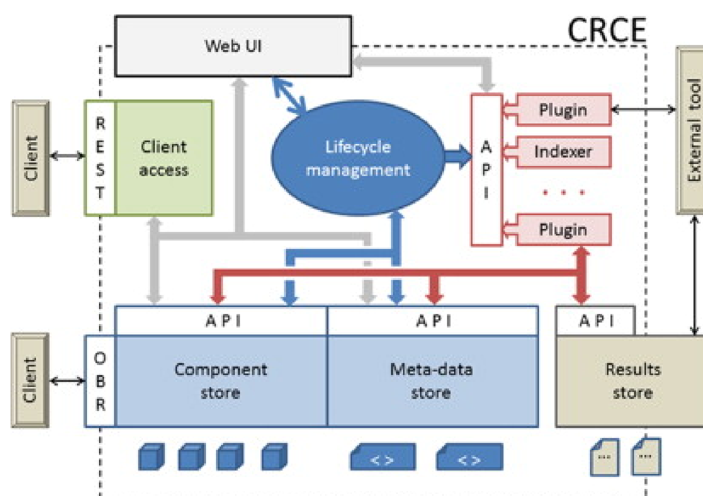
3 Úložiště komponent CRCE

Component Repository supporting Compatibility Evaluation, dále jen CRCE, je úložiště dlouhodobě vyvíjené a spravované Katedrou Informatiky ZČU. Základní principy jsou inspirovány architekturou OSGi[1] Bundle Repository (dále jen OBR)[14], nicméně je navrženo na ukládání libovolných komponent, z jejichž metadat lze později kontrolovat jejich vzájemnou kompatibilitu[16]. Takovou komponentou může být například OSGi bundle, nebo maven artefakt.

Hlavní úlohou tohoto úložiště je komponenty korektně indexovat a kontrolovat jejich vzájemnou kompatibilitu. Samotné úložiště je navrženo jako modulární systém se společným jádrem. Jednotlivé moduly a rozšíření je možno spouštět a vypínat podle potřeby, za chodu aplikace.

3.1 Architektura CRCE

Jak jíž bylo zmíněno výše, architektura CRCE je založena na modulárním frameworku OSGi. To ve výsledku znamená, že systém se, kromě jádra, skládá z volně propojených modulů (viz obrázek 3.1), které realizují potřebné přidané činnosti. Tento systém byl zvolen kvůli potřebě snadné rozšiřitelnosti v případě neplánovaných změn a použití [14].



Obrázek 3.1: Architektura CRCE

[1]Open Services Gateway initiative

Základními stavebními kameny jsou Component store, Meta-data store a Results store. Všechny jsou dostupné skrze definované API a webové rozhraní.

Component store představuje jádro úložiště komponent. Implementace je inspirována architekturou OBR[2] a rozšířena o modifikace, které zajišťují větší rozsah uložitelných komponent. Ostatní části budou popsány v následujících sekcích.

3.1.1 OSGi

OSGi představuje množinu specifikací, které dohromady popisují dynamický systém komponent pro jazyk Java. Aplikace postavená na architektuře OSGi je tvořena znovupoužitelnými, nezávislými komponentami (*bundles*), které mají navzájem skrytou implementaci a jsou propojeny takzvanými *services*, což jsou objekty sdílené mezi komponentami [13].

Komponenta může vytvořit objekt, který následně registruje do *service registry* pod nějaké rozhraní (které objekt implementuje) a tím jej zpřístupní dalším komponentám. Ty mohou jednotlivé *services* procházet, nebo počkat na jejich zpřístupnění v registru. Ve specifikacích OSGi existuje celá řada standardizovaných rozhraní, které mohou komponenty implementovat. Vývojář nicméně není omezen jen na tato standardní rozhraní a může objekty registrovat pod svá vlastní rozhraní.

3.1.2 Tvorba modulů pro CRCE

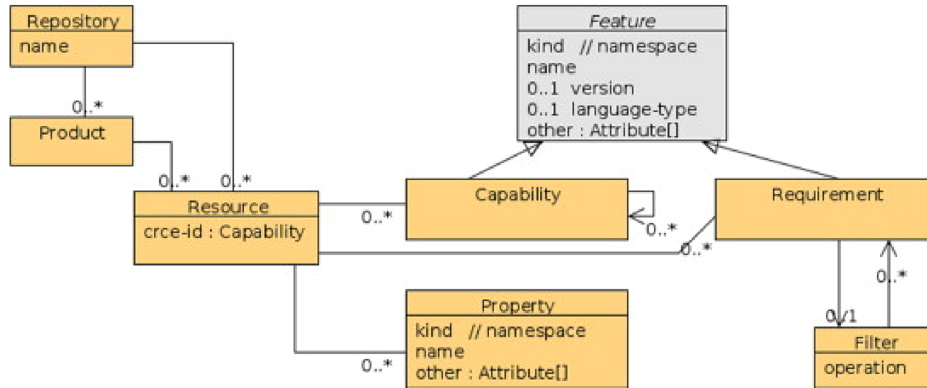
Modul, který je cílem této práce, by měl být schopen identifikovat nahranou komponentu jako maven artefakt, získat z ní potřebná metadata a ta uložit do systému CRCE.

Tvorba modulů je jednou ze základních vlastností úložiště CRCE, která zajišťuje jeho pohodlnou rozšiřitelnost. Principem fungování pluginů v CRCE jsou reakce na jednotlivé události v rámci životního cyklu komponenty (*BeforeUploadToBuffer, AfterUploadToBuffer*, *BeforeCommit...*). Tímto vzniká řetězec pluginů, které jsou schopny nad komponentou provést libovolné operace (typicky extrakce metadat) a následně nedotčenou komponentu předat ke zpracování pluginům, které v řetězci dále následují. Je důležité, aby jednotlivé pluginy komponentu nijak neporušily, to by narušilo správnou funkčnost ostatních pluginů a celého systému.

[2]OSGi Bundle Repository

3.2 Metadata komponent

Po indexování komponenty a kontrole kompatibility jsou získaná data uložena do souboru metadat. Tato metadata představují klíčový element systému CRCE, jejich struktura je naznačená na obrázku 3.2.



Obrázek 3.2: Model metadat

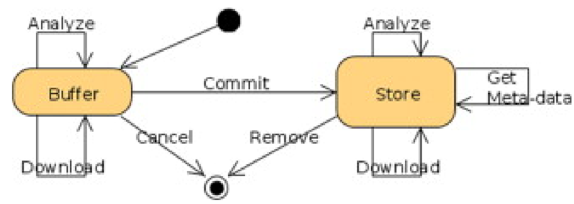
Základními elementy jsou *Resource*, který představuje ukládanou komponentu, *Capability*, které tvoří množinu vlastností a funkcí, které komponenta poskytuje a *Requirement*, které tvoří množinu vlastností a funkcí, které komponenta vyžaduje pro správnou funkci. Každá vlastnost nebo funkce je identifikovaná unikátním klíčem (předem definované jméno) a celá množina vlastností (požadavků) je tvořena páry klíč-hodnota. Z obrázku plyne, že je možné jednotlivé *Capability* řetězit, což umožňuje hlubší a generičtější specifikaci metadat.

Zároveň je také možné definovat vlastní klíče pro speciální hodnoty. Datový model se tak stává značně flexibilní a lze jej bez větších obtíží použít k ukládání metadat o různých typech komponent. Získání a uložení metadat z maven artefaktu je jednou z úloh této práce.

3.3 Komponenta v CRCE

Komponenta je reprezentována výše zmíněnými metadaty a má vlastní životní cyklus, který je znázorněn na obrázku 3.3. Z obrázku je patrné, že komponenta prochází dvěma úložišti - *Buffer* a po indexaci *Store*.

V první části (*Buffer*) proběhne kontrola konzistence komponenty, základní indexace, získání metadat a také kontrola kompatibility. Analýzu komponenty a jejích metadat provádí moduly zmíněné v předchozích sekcích. Modul, který je výsledkem této práce, bude analyzovat maven artefakt právě v této části životního cyklu.



Obrázek 3.3: Životní cyklus komponenty

Pokud je komponenta korektně načtena, jsou metadata uložena do systému CRCE a provede se nahrání do *Store* (záleží na konkrétní implementaci modulu).

4 Prohledávání maven repozitářů

Existuje celá řada možností, jak prohledat repozitář a načíst z něj potřebná metadata komponent. Mezi tyto možnosti patří například ruční použití některého z dostupných on-line vyhledávačů, nebo automatizačního software (program `mvn`, API pro Javu, ...). Při prohledávání je třeba brát ohled na efektivitu a navrhnout vhodnou vyhledávací strategii tak, aby uživatel dostal pokud možno co nejpřesnější výsledek, za co nejkratší dobu. Použití ručních vyhledávačů je tedy evidentně vyloučeno a práce se zabývá pouze automatizovaným prohledáváním.

Cílem této práce je vytvořit modul pro systém CRCE, který bude schopen efektivního prohledání repozitářů a následného získání potřebných artefaktů. Efektivita celého modulu závisí na dobré analýze případů užití, které mohou nastat a následného výběru nejvhodnějších vyhledávacích strategií. Před analýzou a výběrem konkrétních strategií je tedy nutné zamyslet se nad tím, co vlastně má být z repozitáře získáno - jeden artefakt, více různých artefaktů najednou ze kterých uživatel vybere jeden, nebo více... Tato analýza, stejně tak jako strategie z nich vyvozené jsou předmětem následujících sekcí.

4.1 Možné případy užití

Základním předpokladem vyhledávání je co možná nejmenší nutnost interakce s uživatelem. Ten by měl pouze zadat počáteční parametry a program by měl sám, co nejvhodněji, vyhledat artefakty a stáhnout je. Počet nalezených a stažených artefaktů by měl odpovídat očekávání uživatele. Neměl by být příliš velký aby celý systém zbytečně nezahlcoval a uživateli, který by musel nadbytečné artefakty ručně odstranit nepřidělával práci.

Následující výčet znázorňuje možné případy použití, které budou během řešení brány v úvahu:

- Uživatel ví, jaký artefakt má být v repozitáři nalezen a dokáže jej jednoznačně popsat. Zná tedy koordináty `groupId`, `artifactId` a také požadovanou verzi `version.` V takové situaci je vyhledání požadovaného artefaktu triviální a stačí k němu pouze tři zmíněné koordináty, které mohou být použity jako vyhledávací kritéria,

- Uživatel ví, jaký artefakt má být v repozitáři nalezen a dokáže jej, stejně jako v předchozím případě, jednoznačně identifikovat pomocí `groupId` a `artifactId`, ale není si jistý verzí (přesnou formulací), nebo ji vůbec nezná. V takovém případě musí uživatel krom koordinátů zadat ještě doplňující informaci, která bude sloužit ke zpřesnění žádané verze. Takovou informací může být například omezení verze (vyšší než, nižší než), nebo filtr propouštějící nejvyšší, či nejnižší verze.

- Uživatel zná pouze třídu, nebo balík, který by měl požadovaný artefakt obsahovat. V tomto případě nelze artefakt jednoznačně určit a výsledkem vyhledávání by tedy měl být soubor co nejrelevantnějších artefaktů,

- Uživatel stejně jako v předchozím případě zná pouze třídu, nebo balík, který by měl požadovaný artefakt obsahovat, navíc ale dokáže přesně určit jeho `groupId`. Artefakt opět nelze jednoznačně určit, nicméně díky parametru `groupId` by měla být zajištěna větší relevance výsledků, než v předchozím případě.

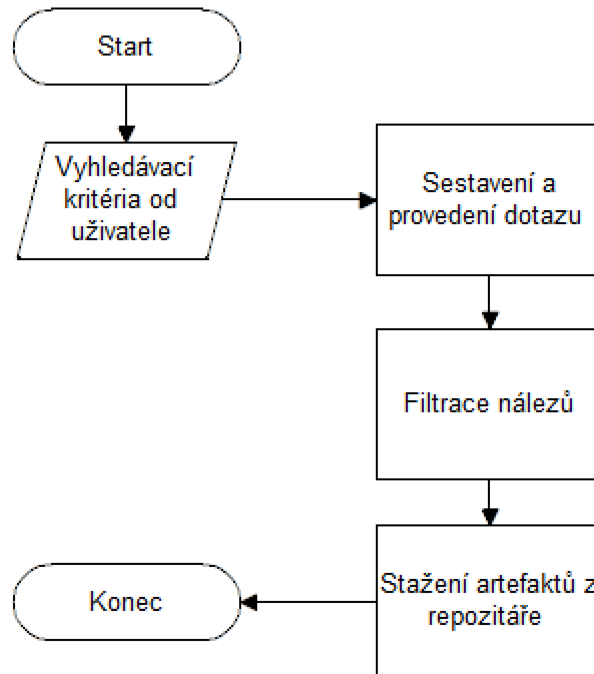
4.2 Strategie prohledávání

V předchozích kapitolách byly rozebrány technologie, které jsem ve své práci použil. Konkrétně se jedná o REST API pro vyhledávání v centrálním repozitáři a knihovna Aether pro získávání artefaktů. Modul bude tedy prozatím omezen jen na centrální repozitář.

Obecný proces vyhledávání je rozdělen do tří fází. V první fázi dojde k sestavení dotazu z parametrů zadaných uživatelem. V druhé fázi dojde k provedení dotazu a filtraci získaného setu metadat. Ve třetí fázi pak dojde ke stažení příslušných artefaktů z repozitáře. Postup je znázorněn na obrázku 4.1. Strategie popsané v této sekci jsou implementovány v první a druhé fázi vyhledávání a jsou zaměřeny zejména na získání souboru výsledků co možná nejmenší velikosti a zároveň největší relevance.

4.2.1 Analýza strategií

V této sekci budou podrobněji analyzovány dříve uvedené případy užití. Z této analýzy budou vyvozeny jednotlivé strategie, které budou detailněji rozebrány, zhodnoceny a následně vybrány nejvhodnější pro použití v mé práci.



Obrázek 4.1: Obecný postup při hledání artefaktu

Vyhledávání podle tří koordinátů

Nejjednodušším případem je vyhledání právě jednoho artefaktu podle standardních koordinátů (ty jsou popsány v kapitole 2). REST API, které v mé práci slouží k vyhledávání, má pro tento případ přímý dotaz a lze jej tedy snadno použít v situaci, kdy uživatel dokáže požadovaný artefakt jednoznačně popsat.

Vyhledávání podle dvou koordinátů s filtrováním

V případě, že uživatel není schopen přesně popsat verzi požadovaného artefaktu, nezbývá nic jiného, než provést vyhledávání podle dvou zbylých koordinátů (tedy `groupId` a `artifactId`). Soubor výsledků získaných tímto hledáním ale může být značně obsáhlý a spousta artefaktů nemusí být, vzhledem ke své nízké verzi a zastaralosti, relevantní.

Tento problém je možné řešit přidáním kritérií, pomocí nichž bude uživatel schopen verzi lépe specifikovat, aniž by znal její přesné znění. Takovými kritérii mohou být například výběr nejvyšší a nejnižší verze, nebo výběr verze s nejvyšším počtem použití (závislostí v ostatních projektech). Druhé zmíněné kritérium předpokládá, že informace o počtu použití artefaktu bude

při vyhledávání dostupná, což v případě REST API, které je v práci použito, neplatí. Proto jsem se rozhodl pouze pro výběr nejnižší a nejvyšší verze. Takto omezený filtr sice neřeší případ, kdy uživatel požaduje stažení artefaktu s verzí „uprostřed“, nicméně v takovém případě předpokládám přesnou znalost verze a tím pádem může být k vyhledávání použita předchozí metoda.

Vyhledávání podle obsaženého balíku

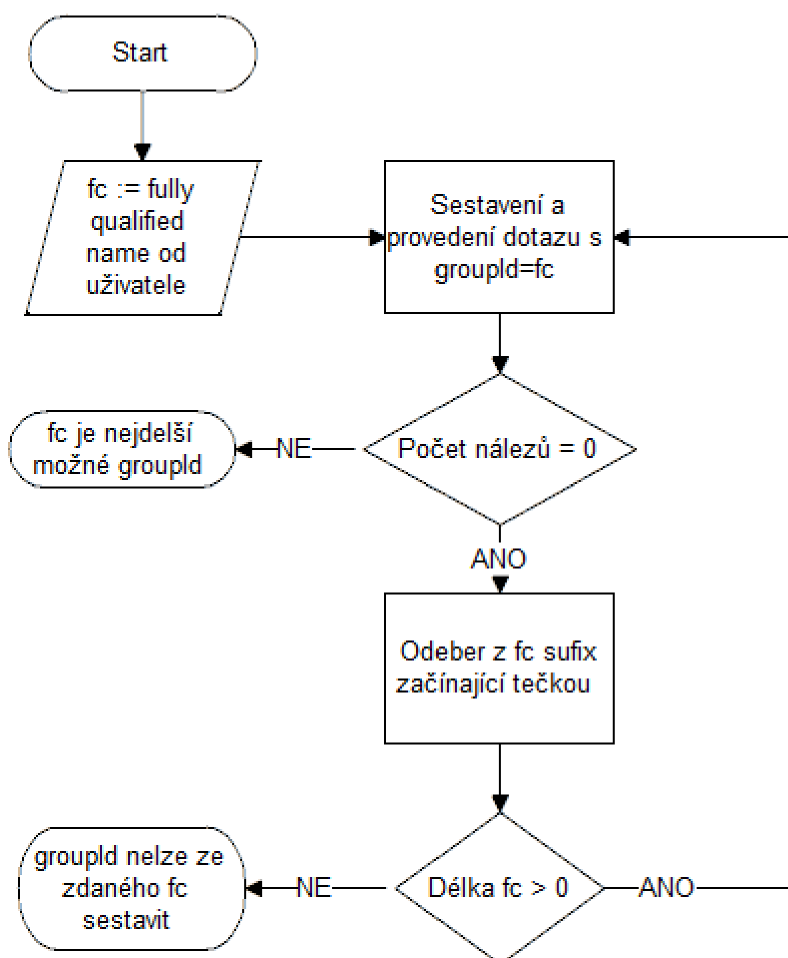
Sofistikovanější přístup je vyžadován v případě, kdy uživatel zná pouze balík, nebo třídu, kterou má požadovaný artefakt obsahovat. Kritériem totiž není popis artefaktu, ale jeho obsah. REST API (stejně tak jako Nexus Indexer) sice nativně podporují i tento typ vyhledávání, nicméně u jiných repozitářů záleží na meta-datech ve zveřejněném indexu - názvy tříd obsažených v artefaktu totiž nemusí být uvedeny.

I když REST API umožňuje vyhledávání podle jména balíku nebo třídy, výsledkem může být poměrně obsáhlý soubor nálezů (řádově stovky až tisíce artefaktů). Navíc také může obsahovat spoustu nálezů, které nejsou relevantní. Například zastaralé verze hledaného artefaktu, artefakty, které patří do slepých vývojových větví, případně artefakty patřící do již ukončených a nepodporovaných projektů. Předpokládá se tedy, že uživatel chce co nejnovější verzi pokud možno stále podporovaného artefaktu, v opačném případě (tedy například při požadavku nalezení starého artefaktu z již ukončené větve vývoje) by uživatel měl být schopen artefakt přesně popsat a k jeho nalezení použít jednu z výše uvedených metod.

V případě velmi obsáhlého souboru výsledků navíc selhává i metoda filtrování pomocí verze, protože je nejprve nutné celý soubor výsledků stáhnout, což je v případě velkého počtu nálezů nepraktické. Řazení vráceného souboru navíc nelze dopředu nijak nastavit a hledání pouze pomocí jména balíku nebo třídy je právě kvůli nutnosti stahovat všechna nalezená metadata artefaktů velmi neefektivní.

Vyhledávání podle obsaženého balíku s filtrováním

Aby se příliš obsáhlý soubor nálezů dal zmenšit, je možné zavést filtrování pomocí `groupId`. Nicméně je nutné splnit dva předpoklady. Prvním z nich je, že jména všech tříd a balíků v artefaktu budou mít společný prefix, kterým je právě `groupId`. Předpoklad vychází z pojmenovávací konvence Mavenu[6], ta je sice nezávazná, ale přesto je obecně dodržována. Druhým předpokladem je, že zadané jméno balíku (nebo třídy) bude v celém znění (fully qualified



Obrázek 4.2: Algoritmus získání nejdelšího groupId postupným zkracováním

name[10]). Nebude tedy uvedeno například pouze `MimerSQLDialect`, nebo `hibernate.dialect`, ale `org.hibernate.dialect.MimerSQLDialect`.

Je evidentní, že celé jméno třídy se obecně nemusí shodovat s žádným `groupId` artefaktu. Nicméně pří dodržení výše zmíněných předpokladů lze postupným zkracováním tohoto jména zprava získat takové `groupId`, které bude odpovídat alespoň jednomu artefaktu. Pokud by jeden z těchto předpokladů nebyl splněn, `groupId` by touto cestou nemohlo být nalezeno, protože by mohla chybět část prefixu (například `org.`), nebo by balíková struktura hledaného artefaktu vůbec nekorespondovala s jeho `groupId`. Algoritmus postupného zkracování je znázorněn na obrázku 4.2. Pokud je `groupId` zkráceno na nulovou délku, artefakt nemohl být pomocí této strategie nalezen a je třeba zvolit jinou.

Filtrování pomocí `groupId` lze provést přímo v dotazu na REST API,

výsledný soubor nálezů je tedy zmenšen ještě před stažením. V kombinaci s filtrem verzí se soubor nálezů zmenší řádově na jednotky (podle počtu artefaktů s různým `artifactId`, které třídu obsahují). Nevýhodou této metody je určitá míra nepřesnosti - i přes výše popsané filtry může dojít ke stažení artefaktů, které uživatel nehledá, nebo naopak k ignorování těch, které jsou požadovány. Druhý případ nastane v situaci, kdy artefakt nedodržuje pojmenovávací konvenci a `groupId` tedy není společným prefixem balíkové struktury. Pokud si je této skutečnosti uživatel vědom a zná `groupId` artefaktu, který by požadovanou třídu nebo balík měl obsahovat, nabízí se možnost nechat uživatele zadat toto `groupId` manuálně.

4.2.2 Navržené strategie

V předchozí části byly detailně rozebrány možné přístupy, které řeší jednotlivé případy užití aplikace. Na základě této předchozí analýzy byly navrženy následující strategie:

- Hledání podle koordinátů `groupId`, `artifacId` a `version`,
- Hledání podle dvou koordinátů `groupId` a `artifactId` s filtrem verzí,
- Hledání podle obsažené třídy, nebo balíku s filtrem verzí,
- Hledání podle obsažené třídy, nebo balíku s filtrem verzí a manuálním groupId filtrem,
- Hledání podle obsažené třídy, nebo balíku s filtrem verzí a nejdelším možným groupId filtrem.

Tyto strategie budou v práci použity a detaily jejich implementace jsou popsány v následující kapitole.

5 CRCE modul

Výsledkem této práce je vytvoření modulu, který rozšiřuje stávající funkcionalitu systému CRCE o integraci s mavenovskými repozitáři. Modul má dvě hlavní funkce – první z nich je načtení a indexace maven artefaktů do CRCE, druhou z nich je nalezení a stažení artefaktu z maven repozitáře. Tyto dvě funkce budou rozšířeny o jednoduché uživatelské rozhraní, přes které bude možno modul ovládat.

5.1 Specifikace

V této sekci jsou detailně popsány vlastnosti, které musí výsledná práce splňovat. Na základě zde uvedených specifikací bylo navrženo řešení a architektura celé práce, jejímž výsledkem je modul rozšiřující stávající systém CRCE.

5.1.1 Základní funkcionality

Modul musí být schopen prohledat centrální maven repozitář podle předem daných kritérií, zejména podle balíků, které artefakt obsahuje. Výsledky hledání musí umět vhodně zpracovat – vybrat z množiny nálezů správné artefakty, následně je stáhnout a zařadit do CRCE. Modul nesmí kolidovat s činností ostatních modulů a nesmí narušit stávající funkcionalitu celého systému. To platí zejména u časově náročného vyhledávání, které by mělo být řešeno vlákny.

Indexace jednotlivých artefaktů proběhne po jejich commitu z bufferu. Modul využije stávajících entit CRCE (zejména `Resource` a `Capability`) aby korektně uložil potřebná metadata maven artefaktu.

Nastavení případných parametrů vyhledávacích komponent by mělo být možné přes uživatelské rozhraní, nebo konfiguračním souborem. Konfigurace těchto komponent by neměla vyžadovat restart celého systému a mělo by tedy být možné ji provádět za běhu.

Modul by měl být také snadno rozšiřitelný - například o implementaci dalších prohledávacích metod, nebo přidání možnosti vyhledávání v jiných repozitářích. Základem jádra modulu by tedy měl být soubor rozhraní popisujících dostupné činnosti, který bude možno v případě potřeby na-implementovat a rozšířit tak celý modul.

5.1.2 Interakce s uživatelem

Předpokládá se, že kritéria pro vyhledávání nebudou automaticky generována systémem, naopak budou zadána uživatelem, společně s případnými kritérii pro filtraci výsledků. Interakce s uživatelem by měla být minimální – v podstatě jen zadání parametrů a následný commit bufferu po provedení vyhledávání, většinu ostatní práce by měl modul provést sám. Nepředpokládá se například, že uživatel bude před každým stažením artefaktů do bufferu tuto akci potvrzovat. Naopak, vše by mělo být pokud možno automatické.

5.1.3 Grafické rozhraní

Z požadavku na interakci s uživatelem plyne i nutnost grafického rozhraní. Úložiště CRCE disponuje základním GUI, postaveným na Java EE. Toto bude v rámci modulu rozšířeno o novou stránku obsahující formuláře pro zadání vyhledávacích kritérií a případné zpětné vazby uživateli (chyba, úspěšné stažení, ...) a servletu, který data z formuláře zpracuje. Stažené artefakty pak budou přidány do bufferu CRCE.

5.2 Popis implementace

Následující sekce popisuje implementaci samotného modulu. Uvedeny jsou nejdůležitější třídy a popis důležitých funkcí. Modul je rozdělen na dvě hlavní části – jádro a uživatelské rozhraní. Jádro je tvořeno samostatným mavenovským modulem `crce-mvn-plugin` (prefix 'crce' je pojmenovávací konvence modulů v CRCE). V příloze je obsažen UML diagram, na kterém jsou znázorněny nejdůležitější třídy a vztahy mezi nimi.

Uživatelské rozhraní je implementováno v modulu `crce-webui`. Jedná se o již existující modul, který obsahuje celé grafické rozhraní systému CRCE. Má práce jej pouze rozšiřuje a to konkrétně třídou `MavenServlet`, která obstarává volání metod z jádra a stránkou `maven.jsp`, která obsahuje samotné uživatelské rozhraní v podobě formuláře na zadávání vyhledávacích kritérií.

5.2.1 Konfigurace

Je žádoucí, aby určité parametry byly pohodlně konfigurovatelné – například údaje o repozitářích, ze kterých mají být artefakty získávány. Třídy, jejichž konfiguraci je možno za běhu měnit, implementují rozhraní `Configurable`, které definuje metodu `reconfigure()` pro načtení nové konfigurace ze souboru. Formát souboru, stejně tak jako jeho obsah není závazný a každá třída,

která toto rozhraní implementuje, jej určí sama. Detailnější popis konfigurace je uveden u tříd, které toto rozhraní implementují.

Metoda `reconfigure()` má ve svém kontraktu stanoveno, že veškerá možná konfigurace má být v metodě přepsána a na tento fakt je při používání aplikace třeba brát zřetel. Například nelze postupně nahrát několik konfiguračních souborů a očekávat, že jejich konfigurace bude sloučena dohromady.

Konfigurace vyhledávacích komponent probíhá za běhu celého systému a je prováděna nahráním konfiguračního souboru skrze formulář, který je umístěn na stejné stránce, jako formuláře pro zadání vyhledávacích kritérií.

5.2.2 Popis tříd

V následující sekci je uveden popis nejdůležitějších tříd a rozhraní, společně s jejich rolí v kontextu celého pluginu. Tato sekce také obsahuje ukázky kódu, z úsporných důvodů je kód oproti originálnímu zjednodušen - neobsahuje obsluhu všech výjimek a některé části kódu nejsou v separátních metodách. Třídy zde popsané a vztahy mezi nimi vychází z přiloženého UML diagramu.

FoundArtifact

Rozhraní `FoundArtifact` specifikuje metody pro získání metadat (zejména základní koordináty) o nalezeném artefaktu. V současné verzi je implementováno pouze třídou `SimpleFoundArtifact`, která je vytvořena podle návrhového vzoru Přepravka.

MavenLocator

Rozhraní `MavenLocator` definuje metody pro vyhledávání artefaktů podle různých kritérií a filtrování nalezených výsledků – v současné době pouze pomocí verze a groupId. Vstupem vyhledávacích metod jsou typicky řetězce specifikující jednotlivá kritéria podle kterých bude hledání probíhat. Výstupem pak jeden nebo více objektů `FoundArtifact`, které reprezentují nalezené artefakty. Právě implementací tohoto rozhraní jsou realizovány strategie navržené v kapitole 4.

V současné verzi pluginu existuje pouze jedna implementační třída, a to `CentralMavenRestLocator`, která vyhledává artefakty v centrálním Maven repozitáři. Implementace je postavena na již zmíněném REST API (sekce 2.3.1) z něhož jsou k vyhledávání využity následující dotazy:

- Dotaz pro nalezení jednoho artefaktu podle koordinátů:
  `g:"<groupId>"+AND+a:"<artifactId>"+AND+v:"<version>"`,

- Dotaz pro nalezení artefaktů v celém rozsahu verzí:
  `g:"<groupId>"+AND+a:"<artifactId>"`,
- Dotaz pro nalezení artefaktů obsahujících třídu nebo balík:
  `fc:"<fullClassName>"`,
- Dotaz pro nalezení artefaktů obsahujících třídu nebo balík s konkrétním `groupId`:
  `fc:"<fullClassName>"+AND+g:"<groupId>"`.

Dotazy obyčejně obsahují další parametry, sloužící ke stránkování a specifikaci výstupního formátu, které nejsou z důvodů přehlednosti uvedeny. Pro usnadnění konstrukce dotazů a také pro lepší testovatelnost jsem vytvořil třídu `QueryBuilder` (návrhový vzor Stavitel). Ta obsahuje metody na zadání konkrétních parametrů a metodu `toString()`, která vrátí sestavený dotaz.

Stránkování a paralelizace vyhledávání Při vyhledávání jehož výsledkem je velký počet artefaktů (například pouze při použití kritéria fully qualified name), může být stahování JSON souboru obsahujícího všechny výsledky zdlouhavé a někdy dokonce neúspěšné – například při stahování souboru o několika stovkách výsledků server pokaždé vrátil chyby 500 a 504. Z těchto důvodů je při volání dotazů použito stránkování (parametry `start` a `rows` viz sekce 2.3.1), které umožňuje rozdělit soubor s výsledky na menší části, jejichž stahování vyhledávací server tolik nezatěžuje.

Vzhledem k tomu, že při stahování jednotlivých částí není nutné čekat na stažení částí předchozích, je stahování každé z nich realizováno ve zvláštním vlákně, čímž je docíleno větší efektivity (příklad je podrobněji rozveden v sekci 5.3).

MavenResolver

K nalezení a získání artefaktu je obecně možné použít vzájemně různé přístupy a z hlediska budoucí rozšiřitelnosti by nebylo výhodné kombinovat je v jednom rozhraní. Z tohoto důvodu jsou metody pro získání artefaktu umístěny ve zvláštním rozhraní `MavenResolver`. Vstupem těchto metod je jeden nebo více objektů `FoundArtifact`, které jsou typicky získány v kroku vyhledávání. Výstupem je pak jeden, nebo více objektů `File`, které ukazují na získané artefakty uložené v dočasném úložišti.

V současné verzi pluginu je rozhraní implementováno pouze jednou třídou, a to `MavenAetherResolver`, která využívá knihovny Aether k získávání

artefaktů z repozitářů. Aether chápe dočasné úložiště jako lokální repozitář a stažené artefakty se tak zde zachovávají a průběžně se neodstraňují, jak by bylo u dočasného úložiště obyčejně očekáváno. V případě požadavku na stažení artefaktu Aether nejprve kontroluje jeho přítomnost v lokálním repozitáři a primárně používá lokální verzi. Cestu k lokálnímu repozitáři stejně tak jako údaje k vzdáleným repozitářům je možné libovolně nakonfigurovat (viz část 2.4.1).

Třída `MavenAetherResolver` implementuje rozhraní `Configurable` a její konfigurace je tak možné měnit za běhu aplikace (tedy bez nutnosti restartovat celý systém CRCE). Konfigurační soubor je očekáván v properties[15] formátu a měl by obsahovat následující údaje:

- `repository.local.path` určující cestu k lokálnímu repozitáři,
- `repositoryN.id` určující id N-tého repozitáře,
- `repositoryN.type` určující typ N-tého repozitáře,
- `repositoryN.url` určující URL N-tého repozitáře.

Je zřejmé, že lze nakonfigurovat $N + 1$ repozitářů, musí však platit, že N začíná od 0 a je inkrementováno vždy po 1. Pokud by tedy byly v konfiguračním souboru uvedeny repozitáře s čísly 0,1, a 3, načteny by byly pouze repozitáře 0 a 1. V případě, že by pro N-tý repozitář nebyly uvedeny všechny potřebné hodnoty, použijí se výchozí. V případě, že žádné repozitáře nejsou uvedeny, systém repozitářů se složí pouze z centrálního, jehož výchozí hodnoty jsou ve třídě pevně zadány.

Třída se vždy při inicializaci nakonfiguruje podle výchozího souboru, který je součástí modulu. Tento soubor je znázorněn na obrázku 5.1 a obsahuje cestu k lokálnímu repozitáři a konfiguraci centrálního repozitáře.

```
2  repository.local.path=target/local-repo
3  repository0.id=central
4  repository0.type=default
5  repository0.url=http://repo1.maven.org/maven2/
```

Obrázek 5.1: Výchozí konfiguračního soubor pro třídu MavenAetherResolver

Absence pom.xml v artefaktech Během testování původní verze modulu jsem objevil závažný problém, který se projevoval neúspěšnou indexací určitých artefaktů i přes jejich úspěšné vyhledání a stažení. Artefakty umístěné v centrálním repozitáři totiž nemusí nutně obsahovat soubor pom.xml,

což odporuje předpokladu uvedenému v sekci 2.1. Repozitář nicméně může uchovávat soubor pom.xml pro daný artefakt zvlášť a je tedy možné jej stáhnout.

Právě této skutečnosti jsem využil při řešení problému: Po stažení artefaktu je zkontrolována přítomnost souboru pom.xml a pokud není tento nalezen, resolver se jej pokusí z repozitáře dodatečně stáhnout. Postup stažení POM je obdobný jako postup pro stažení artefaktu uvedený v části 2.4.1, koordinát `packaging` je pouze zaměněn z hodnoty `jar` na hodnotu `pom`.

FileUtil

Třída `FileUtil` obsahuje statické metody pro práci se soubory (návrhový vzor Knihovní třída). Primárním účelem této třídy je kontrola přítomnosti souboru pom.xml ve zpracovávaném JAR a případné vložení dodatečně staženého pom.xml do archivu.

Activator

Povinná třída odděděná of `DependencyActivatorBase`, která zavádí modul do zbytku systému. V této třídě se pomocí metody `init()` zinicializuje celý modul a zapojí se do lifecyclu CRCE. Na obrázku 5.2 je uvedena relevantní část kódu z třídy `Activator`.

```
42  @Override
43  public void init(BundleContext bundleContext,
44                   DependencyManager dependencyManager)
45          throws Exception {
46      dependencyManager.add(createComponent()
47              .setInterface(Plugin.class.getName(), null)
48              .setImplementation(MavenPlugin.class)
49              .add(createServiceDependency().setRequired(true)
50                      .setService(MetadataService.class))
51      );
52  }
```

Obrázek 5.2: Inicializace pluginu

V inicializační metodě je také možné nastavit komponenty, které musí být pluginu povinně předány od zbytku systému – v tomto případě se jedná o komponentu `MetadataService`, pomocí které lze pohodlně přistupovat k metadatům Resources v CRCE (přidávání kategorií, Capabilit, ...).

MavenPlugin

Třída MavenPlugin představuje jádro pluginu spojující dohromady všechny naimplementované funkcionality. Zde je provedena detekce maven artefaktů v Resources, které jsou commitem bufferu uloženy do CRCE a jejich následná indexace. Třída dědí od AbstractActionHandler a překrývá metodu afterBufferCommit(), kterou se napojuje na životní cyklus CRCE.

Poté, co jsou obecné Resource nahrány do bufferu a dojde k jeho commitu, plugin je postupně prochází a zjišťuje, zda se jedná o maven artefakt. Resource je považován za maven artefakt právě tehdy, jedná-li se o JAR archiv, který obsahuje soubor pom.xml. Z toho jsou načteny potřebná metadata, která jsou následně přiřazena ke zpracovávanému Resource jako nová kořenová Capability. Na obrázku 5.3 je znázorněn kód, který přiřazení vykonává. Kromě metadat je nalezenému artefaktu přidána kategorie MAVEN_ARTIFACT.

```
20    public Resource loadMavenIdentity(Resource resource){
21        URL artifactUrl = null;
22        try {
23            artifactUrl = getUrl(resource);
24
25            Model pomModel = loadPom(artifactUrl);
26
27            Capability rootCap = metadataService
28                    .getSingletonCapability(resource,
29                            NAMESPACE__MVN_ARTIFACT_IDENTITY);
30
31            rootCap.setAttribute(
32                    ATTRIBUTE__ARTIFACT_ID, pomModel.getArtifactId());
33            rootCap.setAttribute(
34                    ATTRIBUTE__GROUP_ID, pomModel.getGroupId());
35            rootCap.setAttribute(
36                    ATTRIBUTE__VERSION, pomModel.getVersion());
37        } catch (Exception ex) {
38        }
39        return resource;
40    }
```

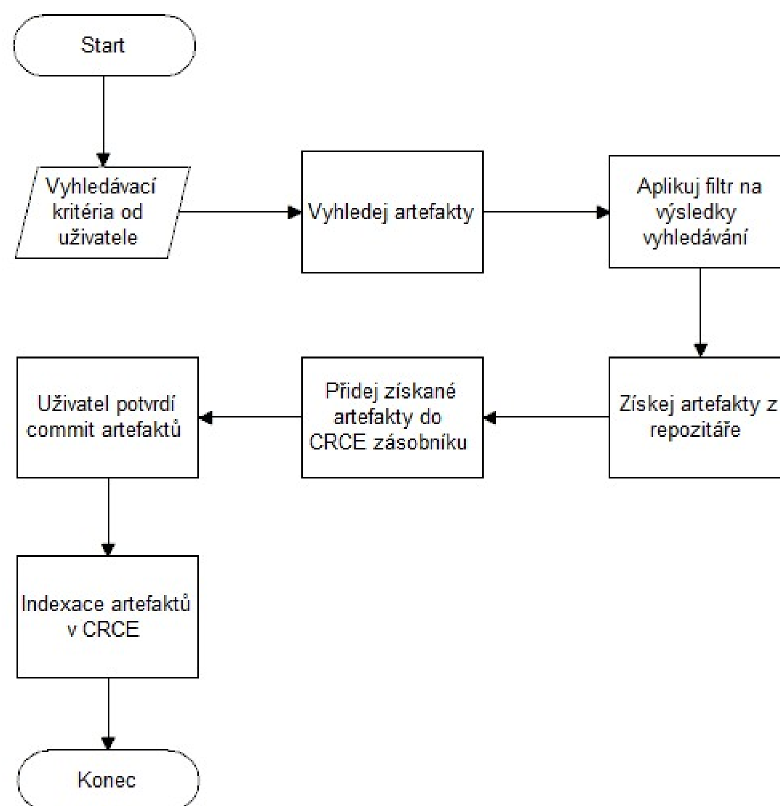
Obrázek 5.3: Načtení potřebných metadat z artefaktu

Je důležité, aby plugin nezasahoval do činnosti ostatním komponentám, které jsou napojeny na životní cyklus. Proto je třeba důsledně ošetřit všechny typy výjimek, které mohou nastat. Typicky špatný pom.xml, v takovém případě je k Resource přiřazena kategorie MAVEN_CORRUPTED. Další výjimka může nastat při prohledávání samotného souboru, kdy plugin očekává jar

archiv s validním `pom.xml`, v takovém případě je `Resource` vrácen nezměněn a ostatní komponenty s ním mohou dále pracovat.

5.2.3 Popis vyhledávání

Proces vyhledávání a indexace artefaktů je rozdělen do několika fází znázorněných vývojovým diagramem na obrázku 5.4. Přechod mezi jednotlivými fázemi je řízen ve třídě `MavenServlet`, která zpracovává požadavek na vyhledávání od uživatele (v architektuře MVC by se tato třída dala přirovnat ke kontroléru).



Obrázek 5.4: Proces vyhledávání

Kritéria zadaná uživatelem jsou v současné době buď koordináty (při vyhledávání konkrétního artefaktu), nebo název balíku, který má artefakt obsahovat a pokud je potřeba také parametry filtrování (verze, `groupId`). K jejich zadání je použit formulář znázorněný na obrázku 5.5. V následující fázi, kdy probíhá vyhledávání, jsou stažena metadata všech nalezených artefaktů.

Pokud je to nutné, je provedena filtrace těchto metadat – podle verze, nebo groupId.

Search by maven coordinates

Group ID

Artifact ID

Version

Version filter:

No version filter (only if all three coordinates are set)

Lowest version

Highest version

Search

Search by fully qualified name

Fully qualified name:

Version filter

Lowest version

Highest version

GroupId filter

No groupId filter

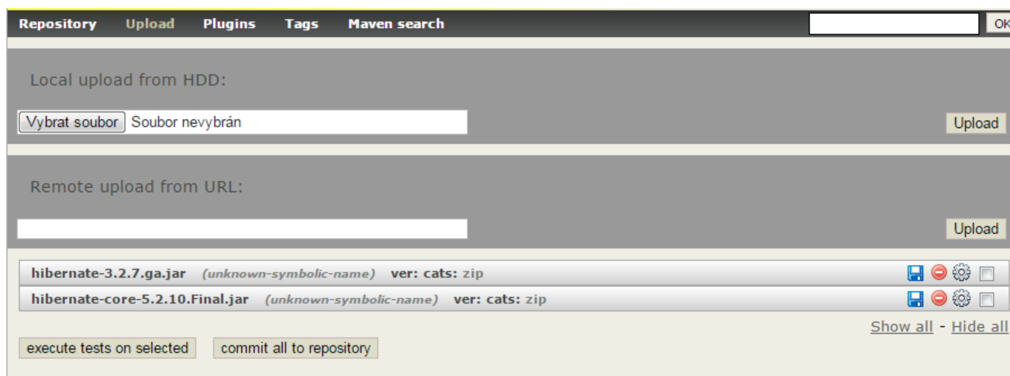
Highest groupId match

Manual groupId filter

Search

Obrázek 5.5: Formulář pro vyhledávání artefaktů

Po získání metadat dojde ke stažení artefaktů a jejich nahrání do CRCE zásobníku. Uživatel pak může sám obsah zásobníku zkontrolovat, nechtěné artefakty manuálně odstranit (případně nad nimi provést jiné operace) a následně potvrdit commit zásobníku, čímž se spustí proces indexace artefaktů. Příklad, jak může vypadat obsah zásobníku po provedení vyhledávání podle jména balíku je znázorněn na obrázku 5.6.



Obrázek 5.6: Vyhledané artefakty v zásobníku

5.3 Testování

Aby byla zaručena a ověřena správná funkčnost celé práce, je nutné všechny podstatné části důkladně otestovat. Testování je rozděleno na dvě části - testování jádra a testování uživatelského rozhraní. Jádro je pokryto jednotkovými testy (framework JUnit), které testují jak samostatnou funkcionalitu jednotlivých tříd, tak jejich vzájemné použití (vyhledat artefakty, následně je stáhnout a oindexovat) podle testovacích scénářů. Uživatelské rozhraní je pokryto smoke testy, které zaručují, že nebyl opomenut uživatelský feedback, výpis chyb, či formální kontrola parametrů.

5.3.1 Testovací sestava

Testy byly provedeny na mém osobním notebooku Lenovo G780. Notebook je osazen dvou-jádrovým procesorem Intel Pentium B970 s taktováním 2x 2,30 GHz a pamětmi s celkovou kapacitou 8 GB. Připojení k internetové síti je ADSL s reálnými rychlostmi 30 Mb/s download a 10 Mb/s upload. Na mém notebooku používám dva 64 bitové operační systémy Windows 7 N SP 1 a Debian 8 Jessie. Vzhledem k multiplatformnosti jazyku Java, ve kterém je celá práce vytvořená, nemá samotný systém na výsledky vliv.

5.3.2 Testovací scénáře

Pro demonstraci funkčnosti modulu jsem vytvořil několik testovacích scénářů, které by měly dostatečně ověřit pokrytí možných případů užití rozepsaných v sekci 4.1. Část z níže uvedených testovacích scénářů je také součástí jednotkových testů pokrývajících jádro modulu.

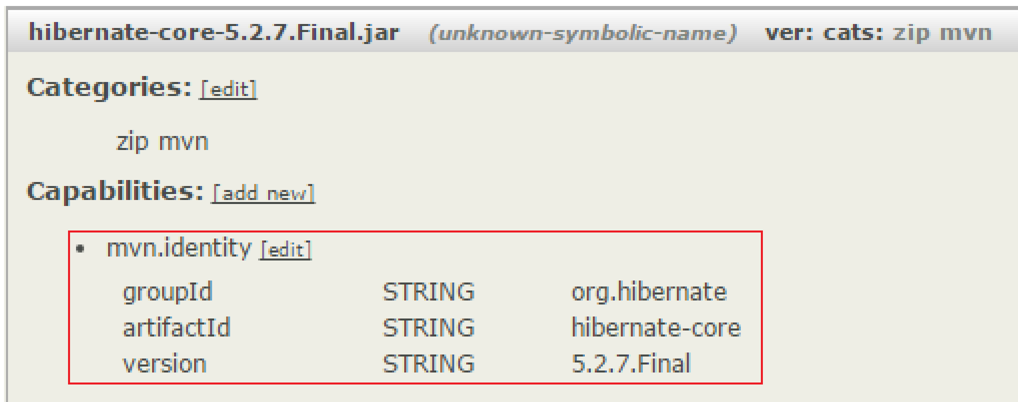
Každý testovací scénář se skládá z tabulky vstupních dat a předpokládá stejný postup: Nalezení a stažení artefaktu, jeho nahrání do bufferu a po následném commitu bufferu jeho indexaci a uložení do CRCE. Výsledkem každého testovacího scénáře je tedy oindexovaný artefakt (jeden nebo více) uložený v CRCE.

groupId	org.hibernate
artifactId	hibernate-core
version	5.2.7.Final

Tabulka 5.1: Tabulka vstupních dat pro scénář 1

Testovací scénář 1 V prvním scénáři je demonstrováno nalezení a stažení artefaktu podle zadaných koordinátů. Jedná se o nejjednodušší možný případ užití, který pracuje se vstupními daty uvedenými v tabulce 5.1.

Výsledek scénáře je uveden na obrázku 5.7. Stažený artefakt sám o sobě neobsahuje soubor pom.xml, ten musel být dodatečně stažen, aby mohlo dojít ke správné indexaci.



Obrázek 5.7: Výsledek scénáře 1

Testovací scénář 2 Druhý testovací scénář demonstruje situaci, kdy uživatel zná `groupId` a `artifactId` artefaktu, ale není si jistý verzí. Program nabízí dvě možnosti filtrování verze – nejvyšší a nejnižší. Test bude proveden pro oba filtry. Tabulka 5.2 obsahuje konkrétní vstupní parametry vyhledávání.

groupId	org.hibernate
artifactId	hibernate-core
version filtr	HIGHEST, LOWEST

Tabulka 5.2: Tabulka vstupních dat pro scénář 2

Výsledky druhého testovacího scénáře (s oběma `version` filtry) jsou uvedeny na obrázku 5.8.

Testovací scénář 3 Třetí testovací scénář demonstruje situaci, kdy uživatel zná pouze jméno třídy, nebo balíku, který se v požadovaném artefaktu má nacházet. Předpokladem je, že uživatel nezná konkrétní `groupId` artefaktu a zvolí jeho automatické nalezení (tedy nejdelší možné `groupId` získané ze vstupního jména třídy, nebo balíku). Zároveň je zapnuto filtrování nejvyšší verze. Vstupní parametry jsou zobrazeny v tabulce 5.3.

Výsledky třetího testovacího scénáře jsou uvedeny na obrázku 5.9.

hibernate-core-3.3.0.CR1.jar *(unknown-symbolic-name)* **ver: cats:** zip mvn

Categories: [edit]

zip mvn

Capabilities: [add new]

- mvn.identity [edit]

groupId	STRING	org.hibernate
artifactId	STRING	hibernate-core
version	STRING	3.3.0.CR1

hibernate-core-5.2.10.Final.jar *(unknown-symbolic-name)* **ver: cats:** zip mvn

Categories: [edit]

zip mvn

Capabilities: [add new]

- mvn.identity [edit]

groupId	STRING	org.hibernate
artifactId	STRING	hibernate-core
version	STRING	5.2.10.Final

Obrázek 5.8: Výsledek scénáře 2

Testovací scénář 4 Čtvrtý testovací scénář demonstruje obdobnou situaci jako v předchozím případě (uživatel zná jen jméno třídy, nebo balíku), nicméně `groupId` filtr je vypnutý. Tento scénář slouží především k porovnání případů, kdy k jednomu vyhledávání je použit `groupId` filtr a k druhému ne. Scénář má téměř totožné parametry (viz tabulka 5.4) jako v předchozím případě.

balík	`org.hibernate.dialect.function`
`version` filtr	`HIGHEST`
`groupId` filtr	nejdelší možné

Tabulka 5.3: Tabulka vstupních dat pro scénář 3

Výsledky čtvrtého testovacího scénáře jsou uvedeny na obrázku 5.10. Kvůli velkému počtu nálezů není zobrazen jejich detail, ale pouze počet. Relevance a porovnání těchto nálezů s výsledky ostatních testů budou rozebrány v následující sekci.

Testovací scénář 5 Poslední testovací scénář srovnává efektivitu postupného stahování jednotlivých stránek souboru výsledků oproti paralelnímu.

hibernate-3.2.7.ga.jar (unknown-symbolic-name) ver: cats: zip mvn

Categories: [edit]

zip mvn

Capabilities: [add new]

- mvn.identity [edit]

groupId	STRING	org.hibernate
artifactId	STRING	hibernate
version	STRING	3.2.7.ga

hibernate-core-5.2.10.Final.jar (unknown-symbolic-name) ver: cats: zip mvn

Categories: [edit]

zip mvn

Capabilities: [add new]

- mvn.identity [edit]

groupId	STRING	org.hibernate
artifactId	STRING	hibernate-core
version	STRING	5.2.10.Final

Obrázek 5.9: Výsledek scénáře 3

Ten byl získán vyhledáváním pomocí obsaženého balíku, bez jakýchkoliv filtrů. V obou případech byl stahován stejný počet, stejně velkých stránek. Tabulka 5.5 obsahuje vstupní data pro tento test a výsledné časy.

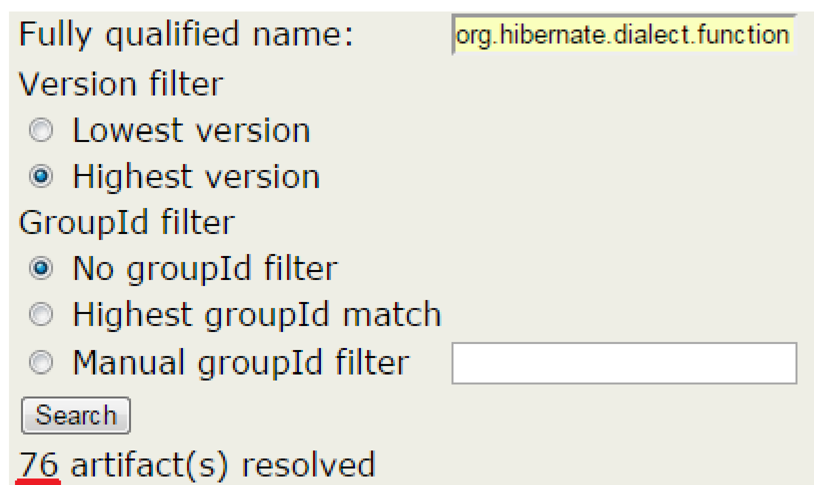
balík	`org.hibernate.dialect.function`
`version` filtr	`HIGHEST`
`groupId` filtr	vypnutý

Tabulka 5.4: Tabulka vstupních dat pro scénář 4

5.3.3 Výsledky testů

První testovací scénář úspěšně demonstroval situaci, kdy uživatel dokáže požadovaný artefakt přesně identifikovat a zároveň dokázal funkčnost části modulu, která detekuje absenci souboru pom.xml ve stahovaném JAR archivu a případně jej dodatečně stáhne. Hodnoty získané indexací, které jsou uvedeny na obrázku 5.7, odpovídají očekávaným hodnotám (viz tabulka vstupních parametrů u prvního testovacího scénáře).

Obdobným případem je i druhý testovací scénář, kde není známa verze.



Obrázek 5.10: Výsledek scénáře 4

Ve výsledcích vyhledávání, které jsou na obrázku 5.8, jsou vidět dva nalezené a oindexované artefakty. Nejnižší vydaná verze hledaného artefaktu `org.hibernate:hibernate-core` je `3.3.0.CR1`, artefakt nejnižší verze byl tedy nalezen správně. Ke dni 22.6. 2017 je nejvyšší vydaná verze `5.2.10.Final` a tedy i artefakt nejvyšší verze byl nalezen správně.

Z výsledků třetího a čtvrtého testovacího scénáře (které demonstrují situaci kdy je známý pouze název třídy nebo balíku) plyne důležitost `groupId` filtru. Při vyhledávání bez použití `groupId` filtrování bylo nalezeno a staženo 76 různých artefaktů (obrázek 5.10). Dá se předpokládat, že nalezení tak velkého počtu artefaktů není cílem vyhledávání a spousta z nich nejspíše pro uživatele nebude relevantní. Oproti tomu během vyhledávání s automatickým určením `groupId` byly nalezeny pouze dva artefakty (obrázek 5.9), z nichž `hibernate-core-5.2.10.Final` je s největší pravděpodobností hledaný artefakt, neboť obsahuje zadaný balík a je novější než druhý nalezený artefakt, který je součástí knihovny Hibernate 3 (ta vyšla v roce 2005 a ke dni 22.6. 2017 je aktuální verze 5 [2]).

balík	`org.hibernate.dialect.function`
celkový počet výsledků	800
počet výsledků na stránku	50
počet stránek	16
počet stahovacích vláken	16
celkový čas stahování bez vláken [s]	150,33
celkový čas stahování s vlákny [s]	42,81

Tabulka 5.5: Tabulka vstupních dat a výsledných časů pro scénář 5

Poslední testovací scénář srovnává dobu potřebnou ke stažení nalezených metadat při paralelním a sériovém stahování. Výsledné časy uvedené v tabulce 5.5 potvrzují, že rychlejšího stažení je dosaženo při použití stahovacích vláken a paralelní přístup je téměř čtyřikrát efektivnější než přístup sériový.

5.4 Použité technologie

Systém CRCE je vyvíjen v jazyce Java, se kterým pracuji již od prvního ročníku bakalářského studia. Proto jsem pro vývoj pluginu zvolil právě tento jazyk. K dnešnímu datu 22.06. 2017 je aktuální verze Java API 1.8[12] a tato byla k vývoji použita. Pro správu závislostí a sestavení modulu jsem použil nástroj Apache Maven, se kterým mám již předchozí zkušenosti. Tento systém také používá CRCE pro sestavování a spouštění.

Pro vývoj v Javě existují různá prostředí, například Netbeans, Eclipse, nebo IntelliJ. S Netbeans nemám prakticky žádné zkušenosti, tuto možnost jsem tedy vyloučil. S prostředím Eclipse mám zkušeností nejvíce, nicméně pro vývoj a práci s velkými projekty není příliš vhodné. Proto jsem zvolil IntelliJ, intuitivní a moderní prostředí, které dokáže rozsáhlejší projekty (kterým CRCE je), zvládat rozhodně lépe než Eclipse.

Modul využívá verzovacího systému Git a celý projekt je (stejně jako CRCE) veřejně dostupný na serveru GitHub.

6 Závěr

Modul pro systém CRCE, jehož vytvoření bylo cílem této práce, splňuje základní požadavky na funkcionalitu – tedy vyhledání artefaktu v repozitáři, jeho stažení, následnou indexaci a uložení do systému CRCE. Kvůli volbě vyhledávací technologie je však prohledávání repozitářů omezené pouze na centrální repozitář, což vnímám jako velkou vadu. Mezi budoucí vylepšení by mělo patřit odbourání tohoto nedostatku a rozšíření prohledávání na libovolný zadaný repozitář.

Z výsledků testů uvedených v sekci 5.3 plyne funkčnost modulu a splnění požadavků na možná kritéria vyhledávání, kterými jsou základní mavenovské koordináty a jméno obsaženého balíku či třídy. Prohledávací strategie popsané v kapitole 4 byly implementovány a jejich použitím bylo dosaženo uspokojivých výsledků. V budoucích rozšířeních by mohlo být navíc zahrnuto řešení tranzitivních závislostí jednotlivých stahovaných artefaktů. Vyhledávací kritéria by se navíc mohla rozšířit například o full textové vyhledávání, kdy by modul našel artefakty s identifikátorem co nejpodobnějším zadanému slovu(ům).

Uživatelské rozhraní, které bylo pro modul navrženo a vytvořeno, pokrývá zadanou funkcionalitu a zároveň by mělo být pro uživatele dostatečně jednoduché a intuitivní. Uživatelské rozhraní bylo vytvořeno ve stejném stylu jako zbytek GUI a na uživatele by tedy mělo působit přirozeně. Součástí vyhledávacího formuláře je i panel s uploadem konfiguračního souboru. V rámci již zmíněných budoucích vylepšeních očekávám rozšíření tohoto panelu, neboť bude nejspíše potřeba předat vyhledávacím komponentám detailnější konfiguraci (například různé repozitáře).

Činnost modulu jsem po dokončení jeho vývoje předvedl vedoucímu bakalářské práce. Modul by měl být zařazen do systému CRCE a používán k dalším činnostem. Tímto považuji práci za úspěšně dokončenou.

Literatura

[1] *The Central Repository* [online]. Sonatype. [cit. 2017/22/06]. Dostupné z: `http://central.sonatype.org/`.

[2] *Hibernate official page* [online]. Hibernate, 2017. [cit. 2017/22/06]. Dostupné z: `http://hibernate.org/`.

[3] *Maven Artifact - Lifecycle* [online]. [cit. 2017/05/02]. Dostupné z: `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`.

[4] *Maven POM* [online]. Apache Maven. [cit. 2017/05/02]. Dostupné z: `https://maven.apache.org/pom.html`.

[5] *Introduction to Repositories* [online]. [cit. 2017/22/06]. Dostupné z: `http://maven.apache.org/guides/introduction/introduction-to-repositories.html`.

[6] *Guide to naming conventions* [online]. Apache Maven. [cit. 2017/13/06]. Dostupné z: `https://maven.apache.org/guides/mini/guide-naming-conventions.html`.

[7] Bradicich, D. *Nexus Indexer API* [online]. Sonatype, 2009. [cit. 2017/22/06]. Sonatype Blog. Dostupné z: `http://blog.sonatype.com/2009/06/nexus-indexer-api-part-1/`.

[8] Bray, T. The JavaScript Object Notation (JSON) Data Interchange Format. Rfc, RFC Editor, March 2014. Dostupné z: `http://www.rfc-editor.org/rfc/rfc7159.txt`.

[9] *Aether* [online]. The Eclipse Foundation, 2014. [cit. 2017/03/26]. Eclipse Wiki. Dostupné z: `http://wiki.eclipse.org/Aether`.

[10] James Gosling, G. S. G. B. A. B. B. J. *Java SE 7 language specification - Names* [online]. Oracle, 2013. [cit. 2017/13/06]. Chapter 6. Names. Dostupné z: `http://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html`.

[11] OBrien, T. *Maven Indexer* [online]. Sonatype, 2011. [cit. 2017/04/2]. Sonatype Blog. Dostupné z: `http://blog.sonatype.com/2011/02/maven-indexer-sonatypes-donation-to-repository-search/`.

[12] Oracle. *Java API Specifications* [online]. Oracle. [cit. 2017/22/06]. Dostupné z: http://www.oracle.com/technetwork/java/api-141528.html.

[13] *OSGi Architecture* [online]. OSGi Aliance. [cit. 2017/03/26]. Dostupné z: https://www.osgi.org/developer/architecture/.

[14] Premek Brada, K. J. Repository and meta-data design for efficient component consistency verification. *Science of Computer Programming.* jul 2014, 97, 3, s. 349 – 365. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0167642314002925.

[15] *Java SE 7 documentation* [online]. Oracle, 2016. [cit. 2017/13/06]. Properties class. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/util/Properties.html.

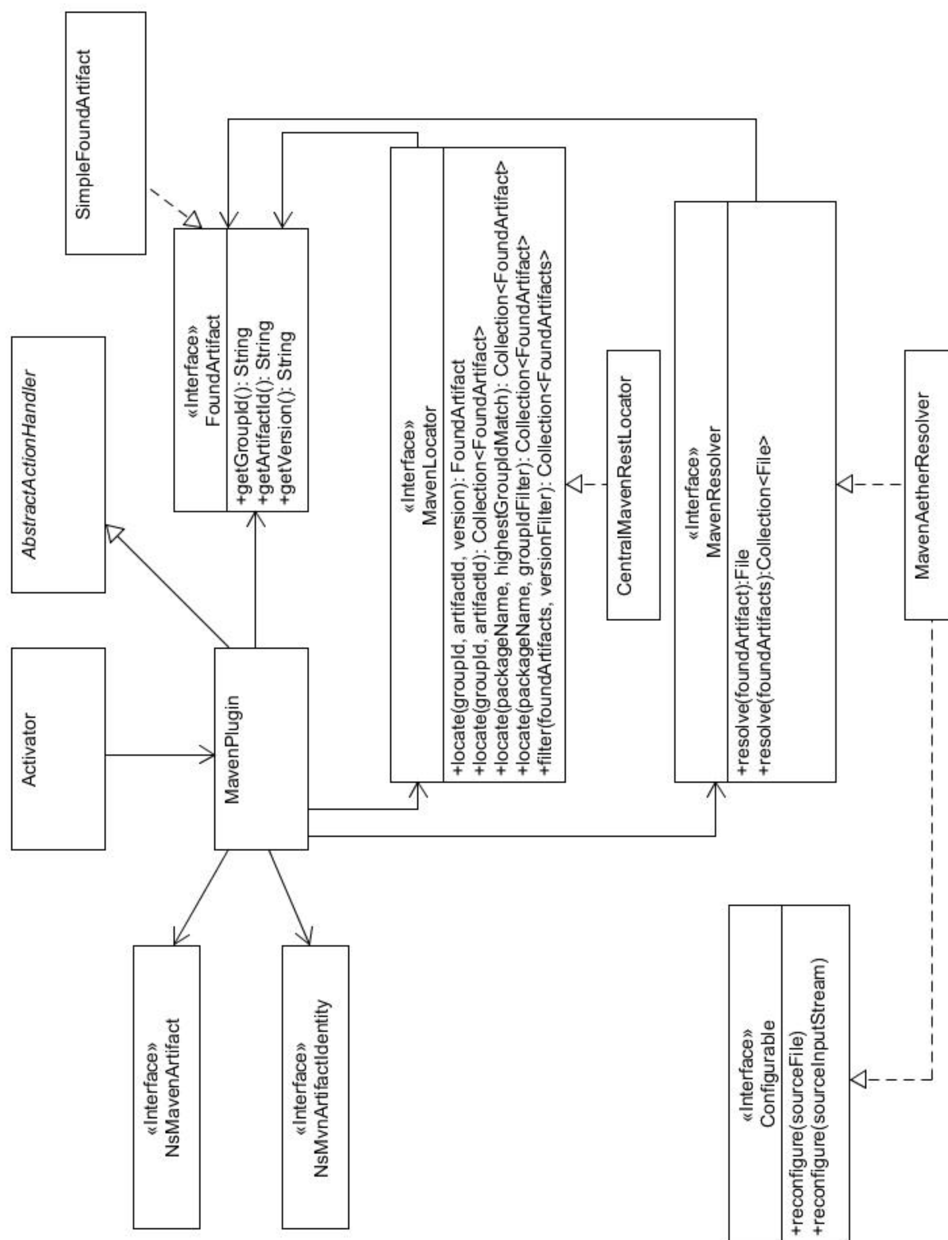
[16] ReliSA. *CRCE - Wiki* [online]. ReliSA, 2015. [cit. 2017/22/06]. Dostupné z: https://app.assembla.com/spaces/crce/wiki.

[17] *Web Services Architecture* [online]. World Wide Web Consorcium, 2004. [cit. 2017/13/06]. Web Services Architecture. Dostupné z: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest.

Seznam zkratek

CRCE	Component Repository supporting Compatibility Evaluation
POM	Project Object Model
JAR	Java Archive
XML	Extensible Markup Language
XSD	XML Schema Definition
REST	Representational State Transfer
API	Application Programming Interface
URI	Uniform Resource Identifier
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
HTTP	Hypertext Transfer Protocol
OSGi	Open Services Gateway initiative
OBR	OSGi Bundle Repository
GUI	Graphical User Interface
UML	Unified Modeling Language

Příloha A - UML diagram vytvořeného modulu

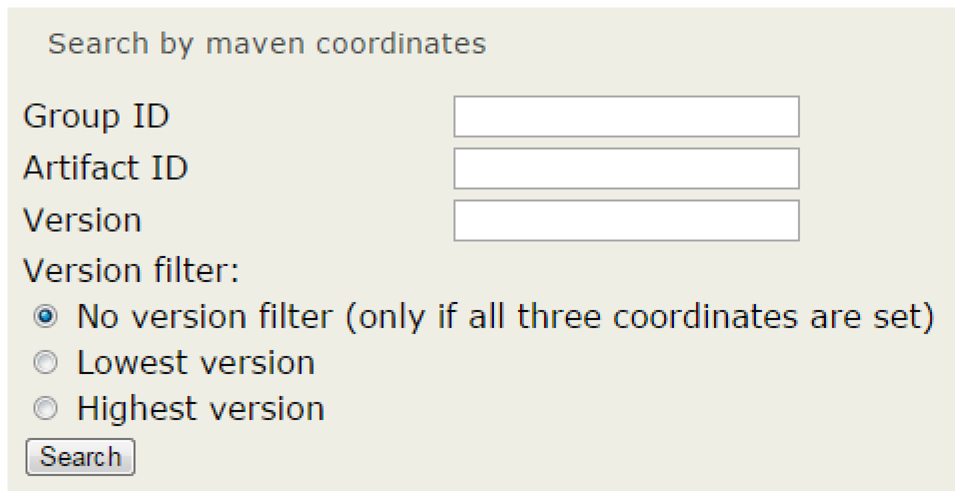


Příloha B - Uživatelský manuál

Tato příloha obsahuje stručný uživatelský manuál, který popisuje ovládání vytvořeného modulu. Modul je ovládán přes grafické uživatelské rozhraní, které obsahuje dva formuláře pro vyhledávání pomocí standardních koordinátů, nebo fully qualified name balíku či třídy.

Vyhledávání podle standardních koordinátů

Vyhledávání podle standardních koordinátů (`groupId`,`artifactId`,`version`) s případným výběrem `version` filtru je možné pomocí formuláře uvedeného na obrázku 6.1.



Obrázek 6.1: Formulář pro vyhledávání pomocí koordinátů

K zadání koordinátů slouží tři označená políčka. V případě, že není vybrán žádný `version` filter, všechny tři koordináty musí být zadané, jinak nebude vyhledávání umožněno. Pokud je vybrán některý ze zbylých `version` filtrů, není nutné koordinát `version` zadávat. Hledání je možné provést stiskem tlačítka „Search“.

Vyhledávání podle fully qualified name

Vyhledávání podle fully qualified name třídy nebo balíku s výběrem `groupId` a `version` filtrů je možné pomocí formuláře uvedeného na obrázku 6.2.

Search by fully qualified name

Fully qualified name:

Version filter

Lowest version

Highest version

GroupId filter

No groupId filter

Highest groupId match

Manual groupId filter

Search

Obrázek 6.2: Formulář pro vyhledávání pomocí fully qualified name

K zadání fully qualified name třídy nebo balíku slouží patřičně označené políčko. `version` filtr je v tomto případě povinný, `groupId` filtr je možné vypnout, nastavit automatické nalezení (volba „Highest groupId match“), nebo zadat manuální `groupId` filtr. Hledání je možné provést stiskem tlačítka „Search“