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

Diplomová práce

Integrace automatické detekce statických chyb do vývojového prostředí

Plzeň 2015 Lukáš Výškrabka

Prohlášení

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

V Plzni dne 12. května 2015

Lukáš Výškrabka

Abstrakt

Tato práce se zabývá problematikou integrace nástroje do vývojového prostředí Eclipse. Integrovaným nástrojem je Java Class Comparator, který na základě přeložených zdrojových souborů a byte kódu knihoven použitých ve vyvíjené aplikaci dokáže odhalit přítomnost statických chyb. Je zde popsáno jak lze tohoto nástroje pro detekci využít a jsou zde uvedeny možnosti, které Eclipse poskytuje pro vytváření jeho rozšíření (pluginů). Dále je zde popsáno, jak byl vytvořený plugin implementován a testován, a v jaké formě jej lze distribuovat koncovým uživatelům. V jedné z posledních částí je ukázáno použití tohoto pluginu na ukázkovém projektu.

Abstract

This thesis deals with tool integration to development environment. Integrated tool is Java Class Comparator, that is able to detect static errors on the basis of compiled source files and byte code of used libraries. There is described how the tool could be used for static error detection and there are shown possibilities for plugins creation provided by Eclipse. There is also described the implementation and testing possibilities of created plugin and how is possible distribute this plugin to end users. In one of the last parts is shown the usage of this plugin with sample project.

Obsah

1 Úvod **1**
1.1 Statické chyby . . . 2
1.2 Způsob detekce statických chyb . . . 4

2 Vývojové prostředí Eclipse **6**
2.1 Úvod . . . 6
2.2 Struktura pluginu . . . 8
2.3 Lazy loading . . . 10
2.4 Eclipse Jobs API . . . 12
2.5 Distribuce pluginů . . . 12

3 Analýza – Integrace nástroje JaCC do Eclipse **16**
3.1 Použití JaCC v pluginu . . . 16
3.1.1 JaCC rozhraní pro kontrolu kompatibility . . . 18
3.2 Podporované typy projektů . . . 18
3.3 Uživatelské rozhraní . . . 19
3.3.1 Spouštění kontroly kompatibility . . . 19
3.3.2 Způsob zobrazování nalezených chyb . . . 21
3.3.3 Seznam ignorovaných balíků nebo tříd (Blacklist) . . . 23

4 Popis implementace **25**
4.1 Příprava parametrů pro JaCC (Maven projekty) . . . 28
4.1.1 Popis důležitých metod . . . 29
4.2 Volání JaCC rozhraní a procházení získaných výsledků . . . 30
4.3 Reportování nalezených chyb . . . 34
4.4 Nové pohledy pro zobrazení výsledků . . . 35
4.5 Popis implementace blacklistu . . . 38
4.6 Rozšíření o podporu jiných typů projektů . . . 40

5 Testování **41**

6 Uživatelská dokumentace **45**
6.1 Instalace pluginu a jeho minimální požadavky 45
6.2 Ukázka funkčnosti na testovacím projektu 46
6.3 Otestování na reálném projektu 50

7 Závěr **51**

A Maven – Definice závislostí na nástroji JaCC **52**

B Příklad volání JaCC rozhraní **53**

1 Úvod

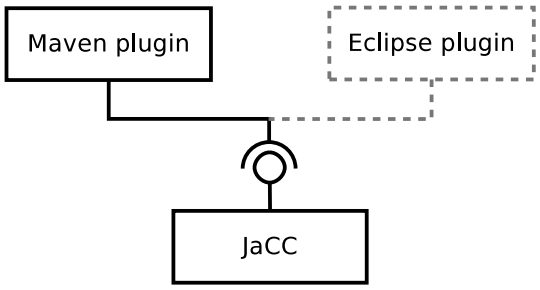
V dnešní době se při vývoji aplikací v jazyce Java běžně používají knihovny třetích stran (rychlejší a levnější vývoj), což nám přináší nutnost dbát na kompatibilitu mezi jednotlivými verzemi. Je totiž plně v rukou vývojáře, jaké verze komponent použije. Při použití nekompatibilních knihoven mohou nastat různé problémy.

Jazyk Java patří mezi staticky typované jazyky. Tyto jazyky vyžadují, aby byly datové typy deklarovaných proměnných jednoznačně definovány. Pokud vyvíjená aplikace obsahuje kompilační chybu způsobenou např. nesprávným datovým typem, jsme o ni informováni již při překladu, a to právě díky kontrole těchto staticky deklarovaných typů, kterou překladač provádí [7]. Tím je předcházeno chybám za běhu aplikace, které by mohly být způsobeny např. nemožností přetypování na požadovaný datový typ. Většina vývojových prostředí se snaží tuto kontrolu provádět při psaní zdrojového kódu (tzv. on the fly). Co ale překladač nebo vývojové prostředí již nekontroluje, je typová správnost (korektnost vazeb) mezi použitými knihovnami, které obsahují již zkompilované binární soubory. Touto problematikou se zabývá nástroj **Java Class Comparator (dále jen JaCC)**. V této práci budou chybami či nekompatibilitami označovány právě ty situace, kdy např. datový typ proměnné nebo parametru metody vyžadované určitou knihovnou nebude shodný se skutečně typem poskytovaným, nebo pokud dojde k pokusu o vytvoření instance neexistující třídy či volání neexistující metody. Jedná se tedy o chyby, které je schopen odhalit za běhu až *ClassLoader*. Pokud taková chyba nastane, je vyvolán chybový stav ve formě potomku `java.lang.LinkageError` [8].

JaCC je nástroj, který vzniká na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Jedná se o nástroj, který provádí statickou analýzu byte kódu použitých knihoven a vyhledává tak nekompatibilní vazby mezi těmito knihovnami [6]. Tento nástroj je vyvíjen v podobě knihovny poskytující potřebné rozhraní (API). Pro možnost jeho využití je tedy nutné vyvinout nějaké uživatelské rozhraní, které by spojovalo uživatele (kteří vyžadují kontrolovat kompatibilitu jimi použitých knihoven) a JaCC (který tuto funkcionalitu poskytuje).

Dosud byl JaCC využit pouze v jednom projektu. Cílem tohoto projektu bylo vytvořit Maven plugin, který by byl schopen využívat rozhraní JaCCu,

a kontrolovat tak kompatibilitu knihoven např. při sestavování projektu. Maven je možné používat z příkazové řádky. Jeho výstup, a tedy výstup vyvinutého pluginu, je pouze textový, a tak se občas může uživateli jevit jako nepřehledný. Současná situace je vidět na obr. 1.1



Obrázek 1.1: Současná situace. Dosud je při vývoji možné využít pouze Maven plugin.

Cílem této práce je vyvinout rozšíření vývojového prostředí Eclipse, které bude umožňovat kontrolu kompatibility za použití nástroje **JaCC**. Toto rozšíření by mělo využít grafické komponenty vybraného vývojového prostředí, a tím vyřešit problém s přehledností výstupu. Cílem je tedy integrace JaCCu do vývojového prostředí, což s sebou nese i požadavek na dodržování zvyklostí v daném prostředí.

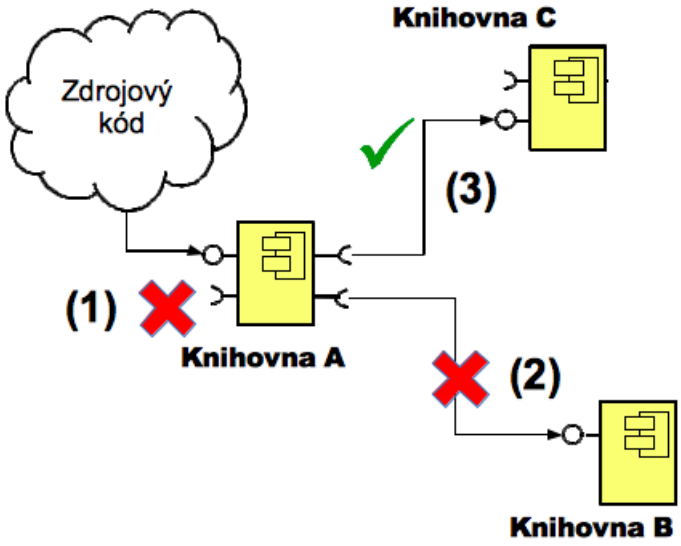
1.1 Statické chyby

Vyvíjené aplikace či velké systémy jsou navrhovány z jednotlivých částí (komponent) a po jejich sestavení jsou spouštěny jako jeden celek. Mějme například situaci, kdy se při vývoji nějaké knihovny změnil název určité třídy. Nekompatibilní knihovna vyžadující starší verzi změněné knihovny očekává původní název třídy a pokud se pokusí vytvořit novou instanci změněné třídy, dojde za běhu k výjimce typu `NoClassDefFoundError`. Tyto chyby mohou být objeveny (v tom nejlepším případě) ve fázi vývoje a nebo (v tom nejhorším případě) v době, kdy je aplikace nasazená v produkčním prostředí, což může mít za následek nedostupnost nějaké důležité funkce, či celé aplikace. Prevencí před těmito chybami je fáze důkladného testování, nicméně téměř vždy se najde část kódu, která testy pokrytá není.

Chyb tohoto typu může teoreticky nastat hned několik, pokud při vývoji novější verze došlo např. k těmto změnám:

- změna názvu metody, odebrání metody, změna počtu parametrů metody – `NoSuchMethodError`
- změna názvu atributu třídy, odebrání atributu – `NoSuchFieldError`

Ostatní chyby, které se mohou objevit za běhu aplikace, je možné dohledat v dokumentaci jazyku Java. [7]



Obrázek 1.2: Rozsah kontroly prováděné překladačem. Nekorektní vazba 1 je odhalena již při překladu. Nekorektní vazbu 2 již překladač neodhalí a tato nekompatibilita způsobí chybu až za běhu aplikace. Zdroj: [6]

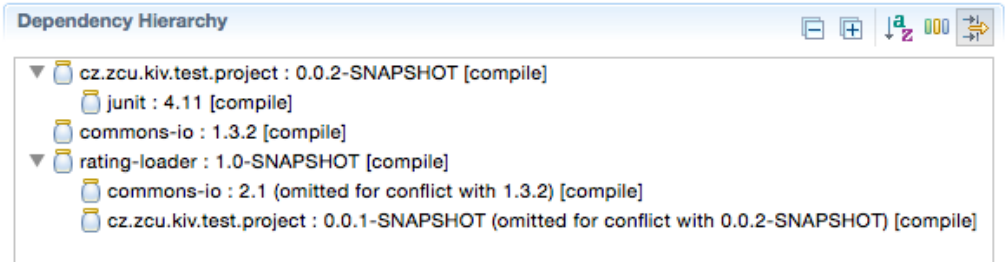
Tyto chyby se vyskytují až za běhu aplikace, jelikož překladač kontroluje pouze vazby mezi vyvíjenou aplikací a použitou knihovnou (to může být například volání metody, kterou poskytuje použitá knihovna), nikoliv vazby mezi použitými knihovnami (metoda ke svému provedení potřebuje ještě nějakou další knihovnu). Kontrola vazeb je názorněji vidět na obrázku 1.2.

Statické chyby v Maven projektech Výše zmíněné problémy **částečně** řeší různé nástroje typu `Maven`, jejichž účelem (mimo jiné) je správa použitých knihoven. V případě Mavenu se tyto knihovny označují jako závislosti a lze je deklarativně definovat. Maven projekty jsou typické tím, že obsahují soubor `pom.xml` obsahující definice závislostí, které budou ve vyvíjené aplikaci použity. Jsou to vlastně knihovny, na kterých bude aplikace závislá, proto se jim říká „závislosti“ nebo anglicky „dependencies“. Takto definované závislosti

jsou poté z centrálního úložiště stažené do lokálního a následně umístěny na `classpath` překládaného projektu. V `pom.xml` to může vypadat následovně:

```
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-io</artifactId>
   <version>1.3.2</version>
</dependency>
```

Většina takových knihoven je závislá ještě na několika dalších knihovnách, které jsou Mavenem automaticky také do projektu přidané, aniž by musely být v `pom.xml` definované. To se nazývá tzv. transitivní uzávěr. Pokud se ale v tomto uzávěru nachází knihovna, která je explicitně definována v `pom.xml` (ačkoliv může být v jiné verzi), je použita ta z `pom.xml` (viz obr. 1.3). A právě toto je prostor pro vznik výše uvedených chyb. Více informací o nástroji Maven lze dohledat na jeho internetových stránkách. [1]



Obrázek 1.3: Hierarchie Maven závilostí, jak jsou zobrazeny v prostředí Eclipse

1.2 Způsob detekce statických chyb

O kompilačních chybách, o kterých nás překladač informuje, se v dnešní době dozvídáme daleko dříve, a to od vývojového prostředí, ve kterém aplikaci vyvíjíme. S rozvojem komponentového programování[1] je jistě zapotřebí nějakého důmyslnějšího nástroje pro statickou kontrolu kompatibility vazeb, než kterým je překladač. Ideálně by tento nástroj měl být součástí vývojového prostředí a stát se tak součástí vývoje. Tento nástroj by měl vývojáře informovat o chybách vzniklých v důsledku použití nekompatibilních knihoven a umožnit jim odhalit, které z použitých knihoven tyto chyby způsobují.

[1]Komponenta je softwarový balík, který poskytuje ucelenou, samostatně užitečnou funkčnost. Aplikace se vytvářejí na základě principu kompozice z více komponent.

Pro získání kýžených informacích o vazbách nám jazyk Java poskytuje Java Reflection API, které umožňuje mimo jiné, načtení informací jako např. jména metod, typy, počty a pořadí parametrů, typ návratové hodnoty atp. Aby bylo možné k těmto informacím přistoupit, je nutné, aby třída, o které chceme zjistit tyto informace, již byla načtena na `classpath`. Potřebné informace je také možno zjistit z analýzy byte kódu. Výhodou tohoto přístupu je, že analyzovaný kód nemusí být načtený na `classpath`. Tohoto přístupu využívá nástroj **JaCC**.

Jedná se o nástroj umožňující výše zmíněnou detekci statických chyb použitím jím poskytovaného rozhraní. Jedná se v podstatě o jednu metodu, jejímž vstupem jsou tři parametry: aplikační soubory (přeložené `*.class` soubory vyvíjené aplikace), knihovní soubory (potřebné `*.jar` soubory pro běh aplikace) a seznam ignorovaných tříd (viz kapitola 4 – Popis implementace). Na obrázku 1.1 jsou tyto vstupy vyznačeny tučně. Výsledkem kontroly je stromová struktura reprezentující nalezené chyby.

2 Vývojové prostředí Eclipse

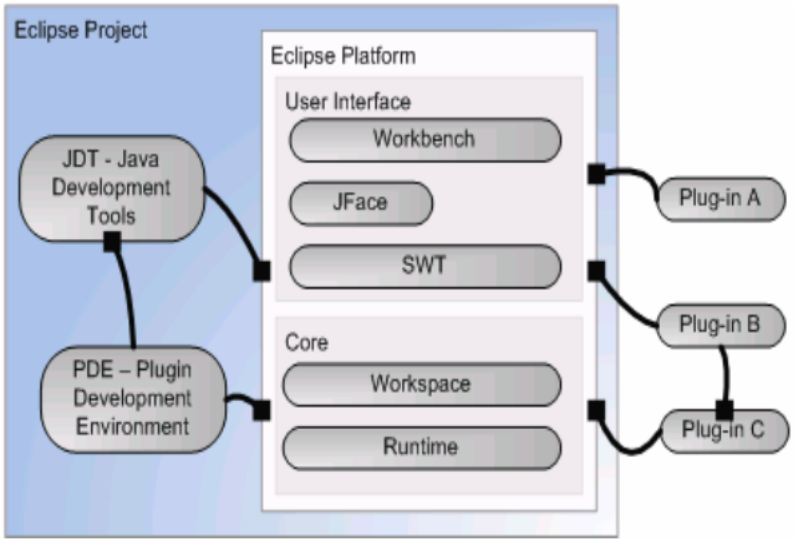
Vývojová prostředí (Integrated Development Environment, IDE) jsou tvořena sadou užitečných nástrojů, které programátorům usnadňují práci, a tak zvyšují jejich efektivitu. Různá vývojová prostředí poskytují různé nástroje, mohou být přímo jejich součástí nebo k nim můžou být dodávané formou rozšíření – tzv. pluginů. Integrace nástrojů zvyšuje jejich použitelnost, používání je rychlejší, intuitivnější, a většinou obdobné jako u ostatních pluginů v daném vývojovém prostředí. Výhodou je, že tato rozšíření nemusí vytvářet vývojáři daného IDE, ale prakticky kdokoliv. Většina vývojových prostředí poskytuje vlastní rozhraní pro tvorbu těchto pluginů.

2.1 Úvod

Eclipse patří k jedním z nejrozšířenějších vývojových prostředí používaných zejména pro vývoj Java aplikací. Flexibilní návrh této platformy dovoluje rozšířit seznam podporovaných programovacích jazyků za pomoci pluginů, například o C++ nebo PHP. Pluginy lze velmi snadno do prostředí doinstalovat – a to pomocí **Eclipse MarketPlace.** Oproti ostatním vývojovým prostředím pro jazyk Java, jako například Netbeans, je filozofie Eclipse úzce svázána právě s rozšiřitelností pomocí pluginů.

Velmi důležité při navrhování pluginu je co nejvíce dodržovat zásady daného prostředí. Jejich uživatel (programátor) pak není nucen učit se novému způsobu ovládání, jelikož je zvyklý na již existující pluginy. Jako příklad si můžeme uvést plugin, který umožňuje používat Maven přímo z Eclipse. Např. spuštění příkazu `mvn install` se provádí přes kontextové menu ikony, která se standardně používá pro překlad aplikace, jako výstup používá standardní Eclipse konzoli a k informování programátora o chybách (např. způsobných špatným nastavením v souboru `pom.xml`) používá tzv. *markery*.

Filozofie Eclipse je zřejmá – rozšiřitelnost. Je to jeho hlavní výhodou, neboť každý programátor může přispět novými komponentami (pluginy). Tyto pluginy rozšiřují dosavadní funkčnost IDE, ale dokonce můžou být znovu rozšířené jinými programátory. Eclipse nabízí vlastní prostředí pro vývoj pluginů nazývané *Plugin Development Environment* (dále jen **PDE**). Eclipse pluginy se vyvíjejí v jazyce **Java** a jsou založené na technologii OSGi. OSGi se zde tedy používá pro správu pluginů. Každý plugin musí obsahovat *manifest soubor* s platnou OSGi hlavičkou, která mimo jiné definuje jeho jméno a verzi.



Obrázek 2.1: Stručný pohled na Eclipse platformu

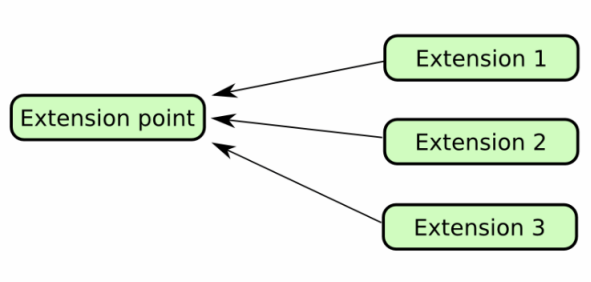
Eclipse pluginy typicky používají **SWT (Standard Widget Toolkit)** k tvorbě uživatelského rozhraní. Je ale možné použít i jiné knihovny pro zobrazování uživatelského rozhraní. SWT nabízí běžné „stavební kameny“, neboli widgety, k tvorbě uživatelského rozhraní, tak jako jiné, podobně zaměřené knihovny. Mezi tyto widgety patří např.: tlačítka, textové popisky, textová pole, dialogy a mnoho dalších. K vykreslovaní prvků GUI používá nativní knihovny operačních systémů prostřednictvím JNI (Java Native Interface), a to obdobným způsobem jako programy psané s použitím nativního API operačního systému. Díky tomu jednotlivé widgety vypadají na každém OS tak, jak je pro něj typické.

Nad SWT je vyvíjena knihovna **JFace**, která na rozdíl od SWT implementuje model MVC[1]. Vývojáři se tak mohou rozhodnout pro použití JFace kvůli pružnějším a abstraktnějším datovým modelům pro komplexní SWT prvky, jako jsou stromy, tabulky a seznamy, a nebo zda chtějí přistupovat k těmto prvkům přímo podle potřeby. SWT i JFace lze samozřejmě použít i mimo prostředí Eclipse, v samostatných aplikacích.

Pluginy podporují modulární architekturu, jelikož jsou to nejmenší jednotky tvořící dílčí funkce IDE. Mohou vytvářet zcela novou funkcionalitu nebo rozšiřovat dosavadní. K rozšíření dochází přeš tzv. „Extension points“,

[1]Model-view-controller architektura

které jsou nabízeny buď jinými pluginy nebo vlastním prostředím. Například pro vytvoření nového `View` se použije extension point s ID „org.eclipse.ui.views“. *Extension points* definují odpovídající rozhraní, které popisuje, co má být v daném rozšíření implementováno. Pro výše uvedený příklad musí být vytvořena třída, která skrze dědičnost rozšiřuje třídu `ViewPart`.



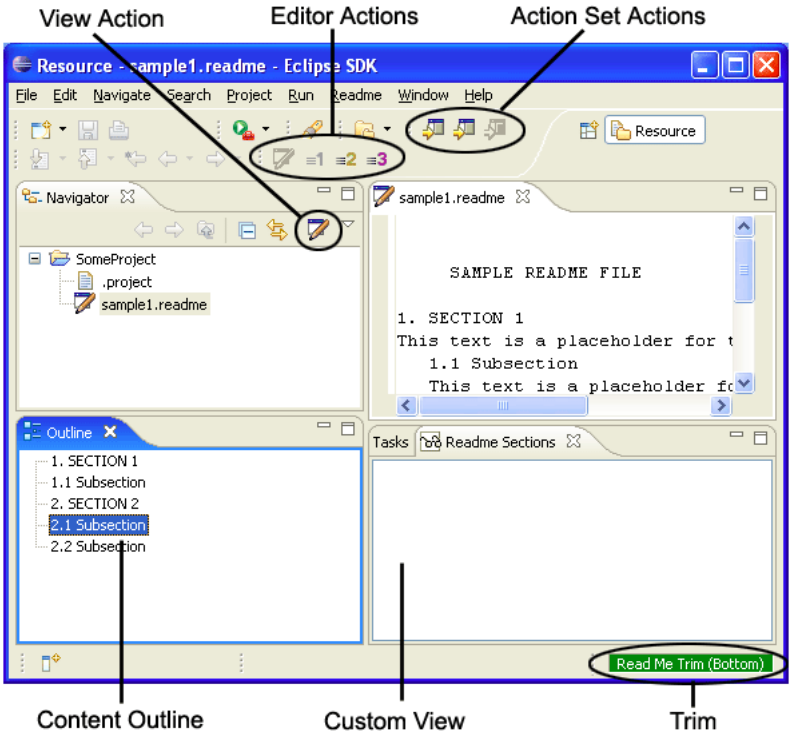
Obrázek 2.2: „Extension point“ je definovaný jedním pluginem, ale využívat jej může více pluginů.

Na obrázku 2.1 je uveden základní pohled na Eclipse platformu. Jsou zde uvedeny pro názornost tři pluginy: Plugin A, Plugin B a Plugin C. První dva jmenované tvoří rozšíření uživatelského rozhraní (to může být například vytvoření vlastního View, nebo přidání nových položek do Menu). Oproti tomu Plugin C využívá extension point poskytovaný jádrem platformy, který může být využit např. pro práci se soubory nebo projekty v daném „Workspace“. To, jaké extension points bude plugin používat a rozšiřovat, je definováno v souboru `plugin.xml`, o kterém si povíme později.

2.2 Struktura pluginu

Základní strukturu pluginu tvoří dva soubory: *META-INF/MANIFEST.MF* a *plugin.xml*. Soubor *MANIFEST.MF* obsahuje OSGi konfigurační informace jako např. název a verzi balíku, aktivační třídu nebo vyžadované balíky. V souboru *plugin.xml* se definuje funkčnost vyvíjeného pluginu a to pomocí „extension points“, které říkají „co bude plugin rozšiřovat“. Např. pokud potřebujeme, aby plugin vypisoval nějaké informace do nového `View`, definujeme jej v *plugin.xml* takto:

```
<extension
point="org.eclipse.ui.views">
  <category
```



Obrázek 2.3: Základní prvky uživatelského rozhraní

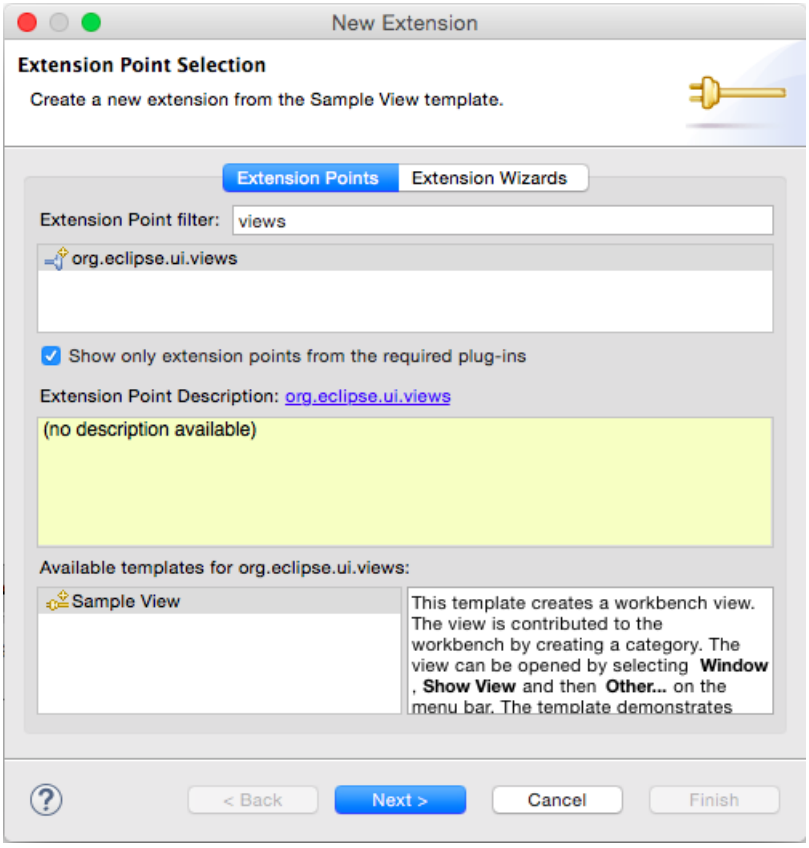
```
        name="Název kategorie"
        id="identifikator.kategorie">
    </category>
    <view
        name="Zobrazovaný název View"
        category="identifikator.kategorie"
        class="trida.obsluhujici.nove.View"
        id="identifikator.noveho.view">
    </view>
</extension>
```

Obdobným způsobem lze použít libovolné z dostupných „extension points“, které lze dohledat v Eclipse dokumentaci [2]. Není však nutné vše v XML souboru definovat manuálně, je možno využít průvodce viz obr. 2.4. „Extension points“ nemusí být pouze využívány, ale pokud chceme umožnit rozšiřitelnost našeho pluginu, můžeme „extension points“ i poskytovat. To potom v *plugin.xml* může vypadat například takto:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?eclipse version="3.2"?>
<plugin>
    <extension-point
        id="cz.zcu.fav.kiv.extensionpoint.example"
        name="Název rozšíření"
        schema="cz.zcu.fav.kiv.extensionpoint.example.exsd" />
</plugin>
```

„Extension point“ je popsán v souboru *example.exsd*.



Obrázek 2.4: Průvodce pro přidání nového rozšíření

2.3 Lazy loading

Aktivace pluginů probíhá podle tzv. *Lazy strategie*. To znamená, že jsou načteny do paměti až ve chvíli, kdy uživatel explicitně vyžaduje jejich funkcionalitu. Teoreticky to vede k relativně rychlému spuštění IDE a k šetření paměti.

Mechanismus „extension points“ zde hraje velmi důležitou roli. Každý plugin obsahuje deklarativní a kódovou sekci. Deklarativní část je obsažena v souboru *plugin.xml*. Tento soubor je načten v paměti při spouštění IDE a je vždy dostupný. To umožňuje informovat uživatele o funkcionalitě daného pluginu, aniž by celý plugin (hlavně kódová sekce) musel být načten a aktivován (což je velmi drahá operace). Plugin může být aktivován následujícími způsoby:

1. Pokud plugin nabízí spustitelné rozšíření, jiný plugin ho může zavolat a to způsobí jeho automatické načtení.
2. Pokud plugin exportuje jednu z jeho knihoven (JAR souborů), jiný plugin se na ni může odkazovat. Načtení třídy patřící pluginu pak způsobí jeho automatické načtení.
3. Plugin může být aktivován explicitně pomocí API, použitím metody `Platform.getPlugin()`. Tato metoda vrátí instanci plně inicializovaného pluginu.

Early startup Většina plugin vývojářů si jistě o svém pluginu myslí, že zrovna ten jejich je tak důležitý, že je nutné jej aktivovat okamžitě při startování IDE. Eclipse platforma poskytuje API, které to umožňuje, nicméně ho nedoporučuje používat (dokonce je označeno jako „deprecated“). Eclipse instalace může obsahovat tisíce pluginů a používáním tohoto API by se startování IDE mohlo velmi prodloužit. Navíc má každý uživatel právo zakázat jednotlivým pluginům aktivaci při startu IDE.

Nicméně pokud to plugin opravdu vyžaduje, je možné tohoto API využít následovně:

```
<extension point="org.eclipse.ui.startup">
    <startup
        class="cz.zcu.fav.kiv.example.PluginStartup">
    </startup>
</extension>
```

Třída `cz.zcu.fav.kiv.example.PluginStartup` je třída implementující rozhraní `IStartup`. Metoda `earlyStartup()` je pak volána hned při startu IDE. Eclipse důrazně doporučuje provádět zde jen ty nejnutnější operace, které nejsou časově náročné. Dalším doporučením je vyvarovat se odkazování se na jiné pluginy v této metodě, jelikož to může vyústit v kaskádní aktivování různých pluginů a k načtení velkého množství nepotřebného kódu.

2.4 Eclipse Jobs API

Eclipse plugin je spouštěn v rámci jednoho vlákna. Toto vlákno se často nazývá `main thread`, či `UI thread`. Pokud je třeba vykonat nějakou déle trvající činnost, je vhodné tak učinit asynchronně vůči vláknu starající se o vykreslování uživatelského rozhraní. Pokud bychom tak neučinili, mohla by nastat situace (a pravděpodobně i nastane), že uživatelské rozhraní celého IDE by bylo neovladatelné. Naopak pokud je potřeba měnit uživatelské rozhraní, je nutné tyto změny provádět právě v již zmíněném `UI thread`. Při pokusu o změnu uživ. rozhraní z nějakého jiného vlákna nám Eclipse framework vypíše následující výjimku:

```
org.eclipse.swt.SWTException: Invalid thread access
```

Při vývoji Eclipse pluginů bychom se bez vytváření nových vláken neobešli, a tak Eclipse poskytuje:

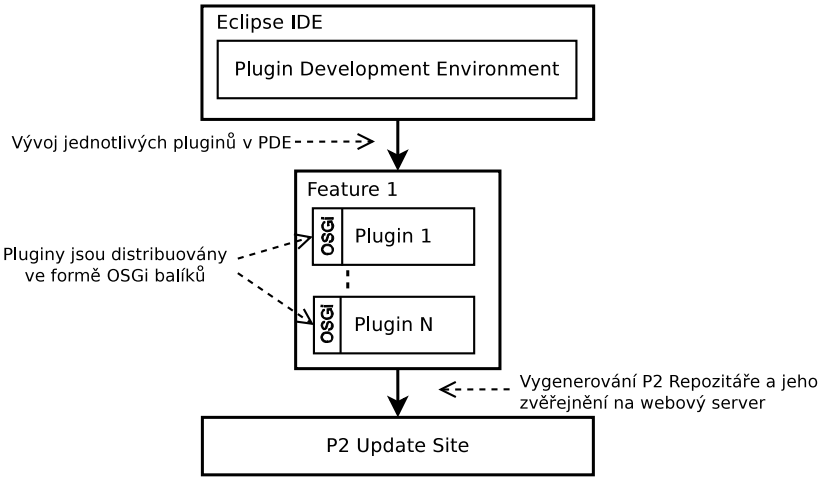
1. Eclipse Jobs rozhranní – rozhraní pro vytváření a spouštění asynchronních procesů bežících na pozadí a umožňující informovat uživatele o jejich stavu a průběhu
2. Synchronizační prostředky pro synchronizaci s hlavním vláknem (možnost upravovat GUI i z nově vytvořeného vlákna)

Těmto procesům je možné před naplánováním přiřadit priority podle jejich předpokládané délky běhu. Existují předdefinované konstanty určující typ procesu, např.: *SHORT*, *LONG* či *INTERACTIVE*. Eclipse plánovač poté na základě těchto priorit určí, ve kterém pořadí mají být tyto „joby“ spuštěny.

Více informací k tvorbě Eclipse pluginů lze získat v dokumentaci k Eclipse [2] či v [4].

2.5 Distribuce pluginů

Instalace pluginů do jednotlivých Eclipse IDE je velmi jednoduchá. Provádí se přes tzv. Update site. Jednotlivé pluginy (může jich být několik) tvoří tzv. Feature, kterou lze pomocí *Plugin development environment* vyexportovat. Výsledek exportu lze umístit např. na nějaký webový server, ze kterého bude poté možné daný plugin instalovat. Souvislosti mezi *Plugin*, *Feature* a *Update site* jsou názorněji vidět na obr. 2.5.



Obrázek 2.5: Souvislosti mezi *Plugin*, *Feature* a *Update site*.

Jednotlivé pluginy jsou distribuovány formou **OSGi balíků** [9]. Jak je vidět na obr. 2.6, P2 Update site obsahuje **metadata** ve formě `xml` nebo `jar` souborů. Mezi ně patří soubor `artifacts.jar`, který obsahuje informace popisující všechny artefakty (soubory) nacházející se v repozitáři. Mezi tyto informace patří např. *id*, *verze* nebo *velikost* artefaktu.

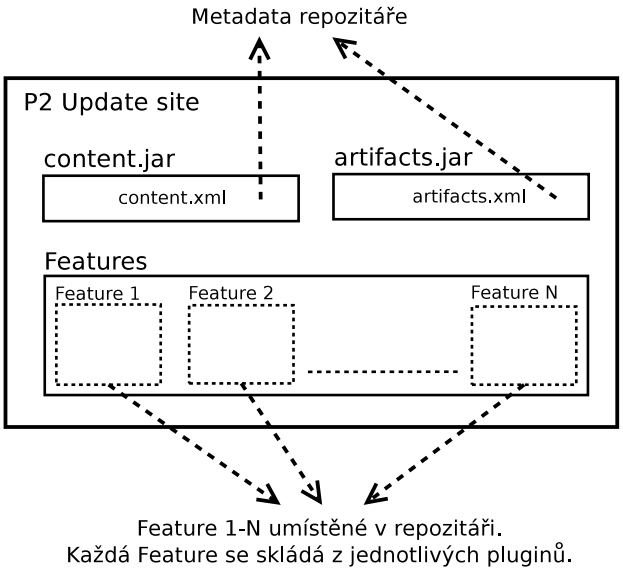
```
<artifacts size='2'>
  <artifact classifier='osgi.bundle'
            id='artifact.id'
            version='0.0.1'>
    <properties size='2'>
      <property name='artifact.size' value='578'/>
      <property name='download.size' value='25382'/>
    </properties>
  </artifact>
  <artifact classifier='org.eclipse.update.feature'
            id='feature.id' version='0.0.1'>
    <properties size='2'>
      <property name='download.contentType'
                value='application/zip'/>
      <property name='download.size' value='795'/>
     </properties>
   </artifact>
</artifacts>
```

Zdrojový kód 2.1: Ukázka obsahu `artifacts.xml`

Druhým souborem je `contents.jar`. Zde se nacházejí bližší informace o jednotlivých instalovatelných artefaktech, jako např.: jméno autora, popis nebo názvy jednotlivých závislostí a jejich verze. Bližší informace o konceptu P2 lze dohledat v [3].

```
<units size='1'>
  <unit id='zcu.plugin' version='0.0.1'>
   <update id='zcu.plugin' range='[0.0.0,0.0.1)'/>
     <properties size='2'>
       <property name='org.eclipse.equinox.p2.name'
                 value='cz.zcu.kiv.jacc.plugin'/>
       <property name='org.eclipse.equinox.p2.provider'
                 value='Lukas Vyskrabka'/>
      </properties>
      <provides size='3'>
        <provided namespace='org.eclipse.equinox.p2.iu'
                  name='cz.zcu.kiv.jacc.plugin'
                  version='0.0.1'/>
        <provided namespace='osgi.bundle'
                  name='cz.zcu.kiv.jacc.plugin'
                  version='0.0.1'/>
        <provided namespace='org.eclipse.equinox.p2.eclipse
    .type'
                  name='bundle' version='1.0.0'/>
      </provides>
      <requires size='13'>
        <required namespace='osgi.bundle'
                  name='org.eclipse.ui'
                  range='0.0.0'/>
        <required namespace='java.package'
                  name='org.apache.maven.model'
                  range='0.0.0'/>
      </requires>
      <artifacts size='1'>
        <artifact classifier='osgi.bundle'
                  id='zcu.plugin'
                  version='0.0.1'/>
      </artifacts>
  </unit>
</units>
```

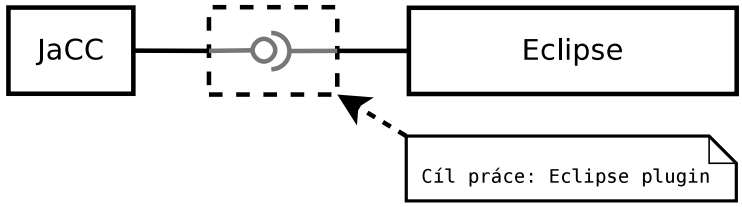
Zdrojový kód 2.2: Ukázka obsahu `content.xml`



Obrázek 2.6: Obsah vygenerované Update site

Pro instalaci je tedy nutné uvést URL daného P2 repozitáře a následně vybrat jeden (nebo několik) z pluginů na něm umístěných. Eclipse pak rozhodne, zda se jedná o nový plugin, či pouze o update, stáhne jeho potřebné závislosti a plugin nainstaluje. Podrobnější informace o instalaci naleznete v kapitole 6 – Uživatelská dokumentace.

3 Analýza – Integrace nástroje JaCC do Eclipse



Obrázek 3.1: Analyzovaná oblast.

Jak již bylo řečeno, cílem práce je volat JaCC rozhraní a jim poskytnuté výsledky zobrazovat pomocí uživatelského rozhraní Eclipse. Na obrázku 3.1 je vidět naše situace, která nám přináší různé problémy, jenž je nutno řešit. Mezi tyto problémy patří zejména:

- jak volat JaCC rozhraní pro námi vyžadovanou kontrolu kompatibility
- pomocí jakých prvků uživ. rozhraní tuto kontrolu spouštět
- pomocí jakých prvků uživ. rozhraní zobrazovat získané výsledky
- jakým způsobem ovládat *blacklist (viz níže)* nástroje JaCC

3.1 Použití JaCC v pluginu

Nástroj JaCC je vyvíjen v podobě **Maven projektu** (viz dále) a je distribuován jako několik Java knihoven, které závisí na několika dalších. Nejsnáze lze JaCC tedy použít také v Maven projektu – a to přidáním **závislosti**. Nástroj Maven se mimo jiné pak postará o dodání všech ostatních potřebných knihoven, na kterých JaCC závisí. Velmi často se používá při vývoji velkých projektů pro zautomatizování různých vývojových procesů, jako např. sestavení projektu, či spouštění automatických testů. Při vývoji Eclipse pluginu se ale standardně Maven nepoužívá, jeho sestavení (vyexportování) se provádí manuálně přímo v prostředí *PDE (Plugin Development Environment)*, což přináší potencionální problém s tím, jak do projektu vyvíjeného pluginu zahrnout závislosti, které by bylo možné definovat pomocí Mavenu. Toto (částečně) řeší projekt **Tycho**.

Přidání závilosti na JaCCu Jelikož se zdrojové soubory a zkompilované třídy nacházejí v repozitáři *Katedry informatiky a výpočetní techniky*, je nutné jej přidat do `pom.xml`, čímž tento repozitář bude zahrnut mezi prohledávané pro vyhledání definovaných závislostí. Nyní už stačí pouze přidat jednotlivé závislosti a je možné začít JaCC v projektu využívat (viz příloha A).

Projekt Tycho Projekt Tycho se zaměřuje na řízení automatického sestavení různých projektů pomocí Mavenu. Mezi tyto projekty patří např.: *Eclipse plugin*, *Eclipse feature*, *Eclipse update site*, *RCP aplikace* nebo *OSGi balíky*. Jedná se o soubor Maven pluginů a rozšíření. Důvodem, proč tento projekt vznikl, je fakt, že Eclipse pluginy využívají metadata z OSGi balíků, zatímco Maven využívá vlastní metadata obsažená v POM. Jedná se o snahu o sjednocení těchto metadat, a tak i o odstranění duplicitních informací mezi těmito metadaty. Výsledkem pak je možnost získat pomocí Mavenu veškeré závislosti (OSGi balíky) z Eclipse repozitáře potřebné pro sestavení Eclipse pluginu a následné jeho sestavení.

Problém ale nastává tehdy, pokud v POM jsou definované „non-OSGi“ závislosti. Tycho je sice získá, ale poté provede kontrolu, zda se jedná o OSGi balíky a pokud ne, jsou tyto závislosti ignorovány. To přináší zásadní problém v použití Tycha k získání JaCCu a jeho závislostí. Řešením by mohlo být použití ještě navíc Maven pluginu s názvem „p2-maven-plugin“. Tento plugin ze závislostí v POM vytvoří OSGi balíky a vygeneruje lokální **P2 repozitář** (používaný Eclipse aplikacemi), ze kterého může Tycho při sestavování projektu příslušné závislosti ve formě OSGi balíků získat. Navíc Tycho neumožňuje pracovat s lokálním P2 repozitářem ve formě složky na lokálním disku, a tak součástí „p2-maven-plugin“ je ještě „jetty-plugin“, který vygenerovaný P2 repozitář umístí na HTTP server.

Jedná se tedy pouze o potenciální řešení, které bych použil, pokud bych potřeboval celý vývojový proces zautomatizovat. Jelikož toto řešení není přímočaré a Eclipse pluginy se standardně pomocí Mavenu nesestavují, tak se jako dostačující zatím jeví možnost automatického zkopírování závislostí z lokálního Maven repozitáře do složky *target* v projektu Eclipse pluginu a manuální přidání těchto `*.jar` souborů mezi jeho závislosti. Toho lze docílit využitím příkazu:

```
dependency:copy-dependencies
```

3.1.1 JaCC rozhraní pro kontrolu kompatibility

Pro spuštění kontroly kompatibility využijeme JaCC metodu s následující hlavičkou:

```
1 ApiInterCompatibilityResult checkInterCompatibility(
2       final File[] appFiles,
3       final File[] libFiles,
4       final ClassFilter classFilter
5 );
```

Zdrojový kód 3.1: Signatura metody checkInterCompatibility poskytované JaCCem

Vstupní parametry metody:

- `appFiles` – Přeložené zdrojové soubory vyvíjené aplikace
- `libFiles` – Seznam všech použitých závislostí (JAR knihoven)
- `classFilter` – Filtr tříd ve smyslu *blacklistu* nebo *whitelistu*.

Vrácené výsledky tvoří stromovou strukturu, kterou je nutné projít a vyhledat v ní tak nalezené nekompatibility. To, jak tuto strukturu lze procházet, a jaké metody pro to použít, bude uvedeno v kapitole 4 – Popis implementace.

3.2 Podporované typy projektů

Dále je nutno se hned v úvodu zamyslet nad tím, zda pluginu umožníme pracovat nad obecně Java projekty nebo se omezíme na konkrétní typ(y) projektů, např. Maven projekty.

První možnost by byla určitě všestrannější. Nicméně v takových projektech není jednoznačné, kde se budou nacházet použité knihovny. Ty se totiž mohou nacházet bud'to přímo v projektu (v nějaké složce) nebo na libovolně jiném místě na disku a mohou být pouze „referencované“. To by přinášelo nutnost pro každý projekt explicitně uvést, kde se nachází všechny jeho knihovny.

Nabízí se tedy možnost omezit se pouze na určité typy projektů, např. na Maven projekty. To by vyřešilo problém s hledáním použitých knihoven, jelikož v Maven projektech je jednoznačně dáno, kde a jak se definují použité knihovny (závislosti) a kde se tyto knihovny fyzicky nacházejí (Maven repozitář). Nastal by zde ale problém, pokud bychom chtěli využít standardního

mechanismu pro ohlašování chyb či varování – tzv. *Markerů*. Ty lze totiž vytvářet pouze u objektů, které implementují rozhraní `IResource` (např. `IFile`) a zároveň musí být daný „resource“ dostupný v aktuálním *workspace*. Závislosti používané v Maven projektech se ale v aktuálním *workspace* nenachází (nachází se v lokálním repozitáři). Tento problém je ale řešitelný pomocí souboru *pom.xml* – ten se ve *workspace* nachází a navíc Maven plugin v Eclipse nabízí možnost, jak v tomto souboru vyhledat příslušné závislosti. Tímto způsobem by se zároveň dodržely Maven zvyklosti v ohlašování různých chyb (např. při chybě z důvodu uvedení špatné verze závislosti).

Z výše uvedených důvodů jsem se rozhodl omezit se pouze na určité typy projektů – primárně na **Maven projekty** – s tím, že bude kladen důraz na možnost rozšířit plugin o další podporované projekty. Jednotlivé Markery budou tedy umisťovány na jednotlivé definice závislostí (na tag `<dependency>`), které příslušnou chybu importují.

3.3 Uživatelské rozhraní

Nástroj JaCC je vyvíjen v podobě knihovny, která nabízí API pro hledání statických chyb v Java knihovnách. Dosud toto API bylo využito pouze v jedné „aplikaci“, a to v Maven pluginu. Rozhraní tohoto pluginu je pouze textové, a tak se výstup může uživateli zdát nepřehledný. To by měl řešit tento Eclipse plugin, který může využít grafických prvků z nabídky SWT, a tak učinit výstup přehlednějším.

3.3.1 Spouštění kontroly kompatibility

Jedna z nejdůležitějších funkčností pluginu mimo zobrazování výsledků bude spouštění kontroly kompatibility. Nejjednodušeji to lze učinit pomocí nějakého tlačítka. Složitější varianta může tuto kontrolu spouštět v nějaký vhodný okamžik **automaticky**. Tak či onak je nutné tuto kontrolu spouštět v novém procesu, abychom předešli tomu, že se IDE stane po dobu kontroly neovladatelným.

Manuálně přes kontextové menu

Položku do kontextového menu lze v Eclipse přidat velmi jednoduše. Jedná se o poměrně malou úpravu v souboru *plugin.xml*. Stačí zde definovat, do jakého kontextového menu chceme položku přidat, do jaké kategorie ji chceme zařadit, jak ji chceme pojmenovat nebo případně ji přiřadit klávesovou zkratku

pro rychlé vyvolání příslušné akce. Dále je možné určit v závislosti na místě vyvolání kontextového menu, kdy bude přidaná položka viditelná. Jelikož budeme chtít spouštět kontrolu nad určitým projektem, vytvoříme nové *submenu* v kontextovém menu okna *Project Explorer* či *Package Explorer*. Toto *submenu* bude viditelné pouze pokud dané kontextové menu bude vyvolané nad **projektem**. Této funkčnosti lze docílit vložením následujícího kódu do souboru *plugin.xml*:

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
      locationURI="popup:org.eclipse.ui.popup.any
                  ?after=additions">
      <menu id="cz.zcu.kiv.jacc.plugin.contextMenu"
            label="JaCC Plugin">
      </menu>
  </menuContribution>
  <menuContribution
    locationURI="popup:cz.zcu.kiv.jacc.plugin.contextMenu">
    <command
       commandId="nejake ID"
       id="ID třídy"
       label="Check compatibility"
       mnemonic="C">

       <visibleWhen>
         <with variable="activeMenuSelection">
          <iterate>
           <adapt type="org.eclipse.core.resources.IProject">
           </adapt>
          </iterate>
         </with>
       </visibleWhen>
      </command>
  </menuContribution>
</extension>
```

Automatické spouštění

Eclipse automaticky provádí několik činností, aniž by je uživatel (vývojář) musel iniciovat. Pro uživatele to přináší větší komfort a usnadnění jeho práce, jelikož se nemusí starat o různé doplňkové funkce. Z těchto důvodů by bylo dobré i JaCC volat ve vhodnou chvíli automaticky.

Volání JaCCu pro kontrolu kompatibility je co se týče paměti a výpočetního výkonu poměrně náročná činnost, a tak je nutné důkladně rozmyslet, jak často a v jaký okamžik ji budeme spouštět. Je také nutné uvést, kdy vůbec má význam kontrolu spouštět, aby bylo možné získat nové (odlišné) výsledky, a abychom zbytečně nezatěžovali IDE. Výsledky se teoreticky mohou změnit po provedení následujících dvou činností:

1. změna zdrojového kódu vyvíjené aplikace (pokud je zapnutý **automatický překlad**) – mohlo dojít k přidání nové nekompatibilní vazby nebo naopak k odstranění nekomp. vazby

2. změna v seznamu použitých knihoven (závislostí v POM) – mohlo dojít ke změně verze jedné z použitých knihoven, a tak i k přidání nové nekompatibilní vazby nebo naopak k jejímu odstranění

První možnost může nastávat velmi často, prakticky po každém uložení změny v nějakém zdrojovém souboru, což by způsobovalo velké zatížení IDE. Jako vhodný kompromis se tedy jeví možnost číslo 2, která nastává méně často.

3.3.2 Způsob zobrazování nalezených chyb

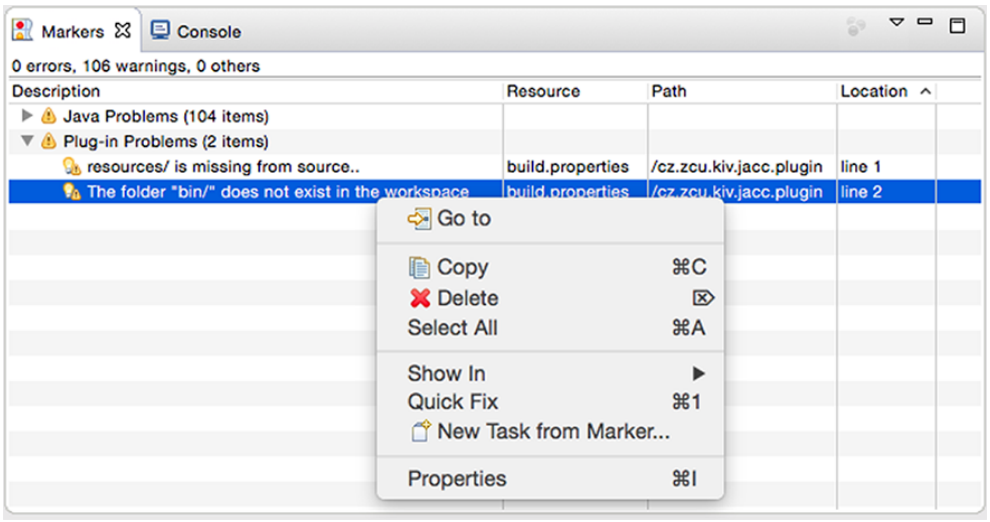
Dalším prvkem uživatelského rozhraní bude výstup JaCCu, neboli zobrazování nalezených chyb. JaCC jakožto výsledek volání kontroly kompatibility vrací stromovou strukturu reprezentující veškeré nalezené chyby. Chyb může být samozřejmě několik a na různých místech, jako např.:

- Chybný název třídy
- Chybný název metody
- Chybný návratový typ
- Chybný typ atributu třídy
- Chybný typ parametru metody

Při zobrazování chyb je žádoucí přehledně je uživateli pomocí nějaké komponenty zobrazit, aby bylo zřejmé, kde se nalezená chyba nachází. Nalezené chyby (= chybové hlášky) je možné (a doporučené) zobrazovat v pohledu „Markers“. Z chybové hlášky, která je tvořena jedním řádkem, nemusí být zřejmé, kde se daná chyba nachází, a tak je možné vytvořit nový pohled, kde by se „cesta“ k chybě v závislosti na vybrané chybové hlášce dynamicky zobrazovala.

Nový pohled pro zobrazení nalezených chyb

Pohled „Markers“ bude sloužit pro zobrazování chybových hlášek vytvořených z nalezených chyb. Jednotlivé záznamy v tomto pohledu jsou označovány jako *Markery* a vždy platí, že jeden záznam odpovídá jednomu Markeru a také jedné řádce. Jedna řádka obsahuje z pravidla několik sloupců. U jednotlivých Markerů lze určit, v jakém souboru a na jaké řádce se daná chyba či varování nachází, čehož se využívá k přechodu na místo chyby, kterou daný Marker popisuje, pomocí „dvojkliku“ či položky v kontextovém menu (viz „Go to“ na obr. 3.2).



Obrázek 3.2: Pohled **Markers** používaný v Eclipse

Všechny chyby obsahují určité informace, které lze zobrazovat, jako např. z jaké třídy a knihovny je nekompatibilita volána nebo jaká změna způsobila nekompatibilitu (smazáno, zobecněno, ap.). Tyto informace mají stejný tvar, ať už chybu způsobuje volání neexistující metody či vytváření instance neexistující třídy.

Chybová hláška, ale musí mít vždy odlišný tvar v závislosti na tom, co tuto nekompatibilitu způsobuje. Například pokud dojde k volání metody s odlišným návratovým typem než je očekávaný nebo k volání metody s jiným typem parametru, tak vždy můžeme bez závislosti na tom, kde se chyba nachází (návratový typ či typ parametru), zjistit o jaký „rozdíl“ se jedná. Tento rozdíl může být v obou případech shodný – např. Zobecněno (Generalised). Text chybové hlášky ale musí uživateli poskytnout konkrétnější informace,

např. kde se chyba nachází nebo proč je toto „volání“ nekompatibilní (např. se typ mohl změnit ze `String` na `Object`). Tento text je již závislý na tom, o jakou chybu se jedná, jelikož musí obsahovat nějaké informace **navíc**, má-li být pro uživatele (vývojáře) dobře čitelný.

Chybovou hlášku bude tedy tvořit text, který ji vystihuje, a bude umístěna do sloupce *Popis (Description)*. Ostatní informace typu z jaké knihovny nebo z jaké třídy je nekompatibilita volána by bylo vhodné umístit do zvláštních sloupců. Problém je, že nelze do standardního pohledu pro zobrazování chyb a varování přidávat nové sloupce. Eclipse ale nabízí možnost vytvořit nový pohled s touto funkčností, do kterého lze přidávat vlastní sloupce, a ve kterém lze zobrazovat pouze chyby určitého typu.

Dalším požadavkem na chybové hlášky je možnost jednoduše upravovat jejich text. Toho může být docíleno např. definováním těchto hlášek v nějakém *properties* souboru.

Nový pohled pro zobrazení detailu chyby

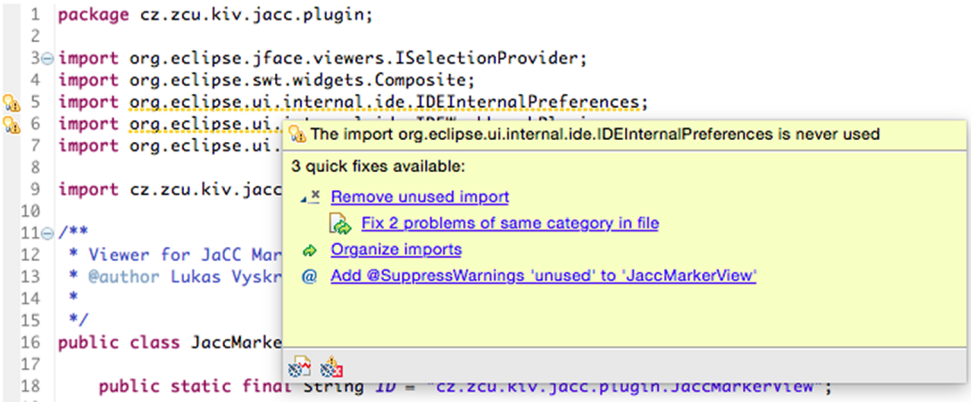
Jak již bylo řečeno, chybová hláška bude závislá na tom, o jakou chybu se jedná, nicméně není možné tuto chybu komplexně popsat v jednom řádku textu. Důvodů proč to nelze, může být hned několik. Jedním z nich je fakt, že potřebujeme uživateli sdělit, kde se daná chyba nachází, což by v jednom řádku textu bylo nepřehledné. Navíc by tato hláška byla velmi dlouhá a mohlo by se stávat, že by se bez nutnosti posouvání nevešla na uživatelův monitor.

Řešením je nezobrazovat polohu nalezené nekompatibility v chybové hlášce, ale v novém pohledu, v závislosti na tom, který Marker by byl aktuálně vybrán. Pro přehlednost by zde bylo možné použít ikony reprezentující jednotlivé uzly této cesty (např. knihovna, třída, metoda, parametr).

3.3.3 Seznam ignorovaných balíků nebo tříd (Blacklist)

Součástí JaCCu je tzv. **blacklist**, kterým lze ovlivňovat jeho výstup. Jedná se o seznam názvů balíků nebo tříd, které pokud volají (importují) nějaké chyby, tak tyto chyby nejsou zahrnuty mezi výsledky. Do *blacklistu* se tedy nepřidávají třídy, které obsahují nějakou nekompatibilitu, ale třídy, které tuto nekompatibilitu importují.

Opakem blacklistu je **whitelist**, který také obsahuje názvy tříd či balíků. Rozdíl je ale v tom, že do výsledku budou zahrnuty pouze ty chyby, jenž jsou importovány ze tříd, které jsou obsažené právě ve whitelistu. JaCC rozhraní nabízí podporu jak pro blacklist, tak pro whitelist, nicméně v tomto pluginu bude implementován **pouze blacklist**.



Obrázek 3.3: Zobrazení „Quick Fix“ v editoru zdrojového kódu

Přidávání názvů tříd či balíků do blacklistu

V Eclipse je zvykem zobrazovat chyby nebo varování pomocí *Markerů*. Pokud existuje nějaké automatické řešení dané chyby, lze jej vyvolat pomocí tzv. „Quick Fix“ (viz obr. 3.2). Pokud má daný Marker definovanou polohu v nějakém souboru, lze „Quick Fix“ vyvolat i z něj výběrem jednoho z nabízených. Nabídka se zobrazí najetím kurzoru myši na podtrhnutý text v příslušném souboru vlivem daného Markeru. Tato situace je vidět na obr. 3.3.

Po přidání záznamu do blacklistu by měl plugin dát uživateli nějakou zpětnou vazbu. Například modifikovat aktuální výsledky do takové podoby, v jaké by se zobrazily po dalším, „novém“, volání JaCCu s přihlédnutím k blacklistu obsahující nové záznamy. Tyto „nové“ výsledky jistě nebudou obsahovat žádné chyby, které jsou importovány z právě přidané třídy popř. balíku do blacklistu. Zpětná vazba by mohla být tedy taková, že po přidání třídy (balíku) do blacklistu budou z pohledu *Markers* smazané chyby, jenž jsou z této třídy (balíku) volány. Přidání záznamu do blacklistu (a následné smazání minimálně jednoho *Markeru*) musí vést do konzistentního stavu, tzn. po opětovném spuštění kontroly kompatibility by měl uživatel získat stejné výsledky.

4 Popis implementace

Jak již bylo řečeno, plugin budeme vyvíjet v prostředí *Plugin Development Environment*, které je součásti Eclipse. Na samém začátku je samozřejmě nutné vytvořit nový Plugin projekt a zvolit pro jakou minimálně verzi Eclipse budeme vyvíjet. V mém případě budu vyvíjet pro nejnovější **Eclipse Luna**, a tak zvolím verzi 3.5 a vyšší. Po vytvoření projektu se nám automaticky vytvoří třída `Activator.java`, která bude řídit životní cyklus pluginu [9].

Pro začátek budeme chtít volat rozhraní JaCCu nad určitým projektem pomocí položky v kontextovém menu. Položku do kontextového menu přidáme stejně tak, jak jsme si uvedli v sekci 3.3.1. Pro obsluhu kliknutí na danou položku je nutné v příslušném kódu v souboru `plugin.xml` uvést název třídy, která nám obsluhu bude provádět – `ExplorerClickAction`. V této třídě je mimo jiné nutné zkontrolovat, zda je příslušný projekt otevřený či nikoliv. Pokud není, tak máme dvě možnosti: zahlásit chybu nebo projekt programově otevřít. Je ale nutné dále pracovat pouze s otevřeným projektem, abychom z něj mohli získávat různé informace, jako např. informaci o jeho typu. V kapitole 3 jsme se omezili pouze na projekty typu Maven, a tak při spuštění kontroly kompatibility z kontextového menu projektu je nutné zkontrolovat, zda se jedná o projekt typu **Maven**. Toho lze docílit následujícím kódem ve třídě `ExplorerClickAction`:

```
1 IProject project;
2  .
3  .
4 if(project.hasNature(IMavenConstants.NATURE_ID)) {
5    ...
6 }
```

Zdrojový kód 4.1: Zjištění zda projekt je typu Maven

Podle toho, zda se jedná o podporovaný typ projektu (zatím tedy pouze Maven projekt), vytvoříme instanci třídy, která bude spouštět kontrolu kompatibility. Protože samotné volání JaCCu bude vždy nezávisle na typu projektu stejné, využijeme dědičnosti, a rozdělíme tuto činnost na dvě dílčí: na přípravu vstupních parametrů pro JaCC (závislé na typu projektu) a na samotné volání JaCCu. Prakticky tyto činnosti obstarávají dvě třídy z nichž jedna je abstraktní:

- `AbstractCompatibilityChecker.java` – Abstraktní třída, která má implementované pouze ty metody, jenž nejsou závislé na typu projektu.

Prakticky se jedná např. o metodu volající rozhraní JaCCu nebo o metodu procházející poskytnuté výsledky. Ostatní metody jsou abstraktní, a tak je nutné je implementovat v třídách dědících od této.

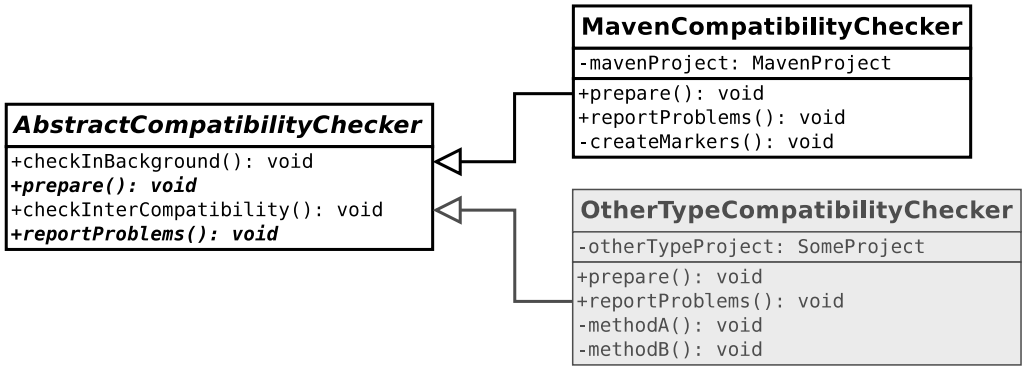
- `MavenCompatibilityChecker.java` – Třída obstarávající přípravu dat, které JaCC vyžaduje na jeho vstupu. Zde se jedná např. o přípravu přeložených zdrojových souborů či seznamu použitých knihoven.

Abychom uživateli nezablokovali celé IDE, vše, počínaje přípravou vstupních dat, až po reportování získaných výsledků, je nutné provádět v novém vlákně, neboli v tzv. **Eclipse Job**. To nám zajistí následující metoda ze třídy `AbstractCompatibilityChecker`:

```
public void checkInBackground() {
  Job job = new Job(JACC_ECLIPSE_JOB_NAME) {
     protected IStatus run(IProgressMonitor monitor) {
      try {
        log.info("CompatibilityChecker starter.");
        check(monitor);
      } catch (Exception e) {
        log.error(e.printStackTrace());
        return Status.CANCEL_STATUS;
      }

      return Status.OK_STATUS;
    }
  };

  // schedule job
  job.setPriority(Job.LONG);
  job.schedule();
}
```

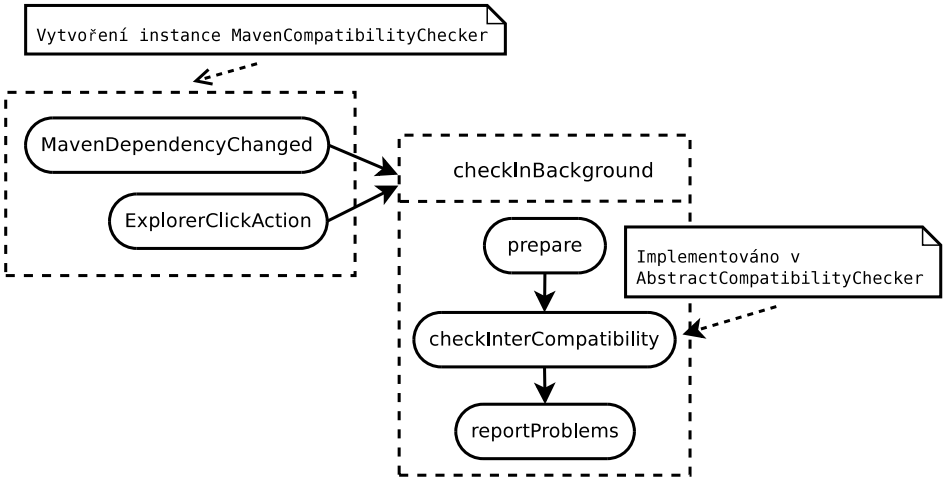
Zdrojový kód 4.2: Vytvoření a naplánování Eclipse jobu

Implementace metody `check(monitor)` je již závislá na typu projektu, a proto se nachází až ve třídě `MavenCompatibilityChecker.java`. Tato metoda provádí přípravu vstupních dat a na jejím konci volá metodu opět z `AbstractCompatibilityChecker.java`, jenž tyto parametry předá JaCCu a následně spustí kontrolu kompatibility. Vztah mezi těmito třídami je vidět na obrázku 4.1.



Obrázek 4.1: Vztah mezi třídami AbstractCompatibilityChecker, MavenCompatibilityChecker a možnost rozšíření podpory o jiné typy projektů.

Jednotlivé činnosti jsou názorněji vidět na obrázku 4.2. Spuštění kontroly (zavolání metody **checkInBackground**) lze provést ze dvou míst: automaticky po změně závislostí v Maven POM (**mavenDependencyChanged**), nebo kliknutím na tlačítko v kontextovém menu (**ExplorerClickAction**). Poté třída `MavenCompatibilityChecker` provede přípravu vstupních parametrů pro JaCC (**prepare**). Tyto parametry jsou následně předány třídě `AbstractCompatibilityChecker`, která zavolá JaCC rozhraní, projde získané výsledky a vytvoří chybové hlášky (**checkInterCompatibility**). Získané výsledky reportuje opět třída `MavenCompatibilityChecker`.



Obrázek 4.2: Jednotlivé činnosti při kontrole kompatibility.

4.1 Příprava parametrů pro JaCC (Maven projekty)

Třída `MavenCompatibilityChecker.java` má za úkol připravit všechny potřebné vstupní parametry pro metodu JaCCu, která provádí kontrolu kompatibility. Hlavička této metody je uvedena v kapitole 3. Třída musí tedy na základě projektu nad kterým má být kontrola spuštěna připravit tyto parametry:

- `appFiles` – Přeložené zdrojové soubory vyvíjené aplikace
- `libFiles` – Seznam všech použitých závislostí (JAR knihoven)
- `classFilter` – Filtr tříd ve smyslu *blacklistu* nebo *whitelistu*.

Přeložené zdrojové soubory Přeložené zdrojové soubory můžou být v Maven projektech v různých složkách (záleží na nastavení v POM). Tuto informaci lze získat z objektu typu `MavenProject` pomocí metod `getBuild()` a `getDirectory()`. V obslužné třídě události kliknutí na položku v kontextovém menu máme k dispozici objekt `IProject`. To, jak z něj získáme objekt typu `MavenProject` bude uvedeno později.

Seznam použitých závislostí Seznam tranzitivních závislostí lze získat zavoláním metody `getArtifacts()` na objektu `MavenProject`. V tomto seznamu se nachází i ty závislosti, které nejsou pro projekt přímo definovány v `pom.xml`, ale jsou potřeba pro správnou funkčnost knihoven zde definovaných. Pokud by se v jedné z těchto knihoven nacházely chyby (a často se tomu tak děje, z důvodu chybějících dalších knihoven), není jasné, kde v souboru `pom.xml` vytvořit příslušný Marker, jelikož zde tato knihovna není definována. Řešením je zobrazit Marker na definici té závislosti, kvůli které byla tranzitivní závislost do projektu přidána. Z tohoto důvodu je nutné vytvořit pro projekt **graf závislostí**, aby bylo možné „rodičovské“ závislosti později dohledat.

```
1 MavenModelManager manager =
2                       MavenPlugin.getMavenModelManager();
3 DependencyNode root = manager.readDependencyTree(
4                           mavenFacade, mavenProject,
5                           scope, monitor
6                       );
```

Zdrojový kód 4.3: Maven – Vytvoření grafu závislostí

Pokud již máme vytvořený graf závislostí, je vhodné zde všechny závislosti (objekty typu `Dependency`) převést na objekty typu `DependencyNode`. Tento objekt totiž obsahuje reference na své potomky, a bude tak snadné dohledávat případné tranzitivní závislosti a jejich „rodiče“. Převod můžeme provést procházením vytvořeného grafu a porovnáváním vyhledávaného objektu `Dependency` a aktuálně v grafu nalezeného `DependencyNode`. Pokud se jejich *Group Id*, *Artifact Id* a *Version* shodují, je převod na `DependencyNode` hotov.

Filtr tříd Pro každý projekt může být vytvořen soubor s názvem `jacc-blacklist.txt`, který bude obsahovat názvy tříd nebo balíků tak, jak bylo uvedeno v sekci 3.3.3. Je tedy nutné jej načíst a vytvořit instanci třídy `ClassFilter`. Mohlo by se zdát, že načítání *blacklistu* je na typu projektu nezávislé, nicméně Maven projekt může obsahovat několik modulů (viz níže), a tak je nutné načíst blacklist i pro tyto moduly. Načítání obstarává třída `BlacklistSupport.java`, která bude popsána dále.

Maven projekt může být typicky strukturovaný tak, že obsahuje několik modulů. Proto je nutné do kontroly kompatibility zahrnout i jeho moduly (tzn. jejich závislosti, přeložené zdrojové soubory a obsahy blacklistů).

4.1.1 Popis důležitých metod

Jednu z důležitých činností provádí metoda `projectToMavenProject`. Jedná se o metodu, která provádí převod typu `IProject` na typ `MavenProject`. Na začátek je nutné opět ověřit zda je `IProject` otevřený. Pokud by nebyl, *MavenPlugin* by jej neměl načtený ve své **cache** a převod by nemohl být uskutečněn. Získat `MavenProject` lze z cache následovně:

```
1 IMavenProjectRegistry registry =
2                 MavenPlugin.getMavenProjectRegistry();
3
4 facade = registry.getProject(project);
5 facade.getMavenProject();
```

Zdrojový kód 4.4: Získání projektu z Maven cache

Nicméně i pokud `IProject` otevřený je, může nastat situace, že není načtený v cache Maven pluginu. Pak je nutné jej do ni načíst, a to obdobně jako výše, ale s jedním rozdílem: metodě `getMavenProject()` se jako parametr předá objekt `IProgressMonitor monitor`.

```
facade = registry.getProject(project);
facade.getMavenProject(monitor);
```

Zdrojový kód 4.5: Načtení projektu do Maven cache

Tím je zajištěno, že na pozadí bude Maven cache obnovena, a pokud požadovaný `IProject` existuje, bude vrácen příslušný `MavenProject`.

`MavenProject` jako své moduly vrací pouze jejich názvy. Je tedy nutné pro tyto názvy získat příslušné Maven projekty, jelikož nad nimi potřebujeme provádět operace jako např. `getArtifacts()`. Převedení na objekt `IProject` je tedy dílčím krokem:

```
IProject getIProjectFromModule(String module) {

 IProject project = ResourcesPlugin
             .getWorkspace()
             .getRoot()
             .getProject(module);

}
```

Zdrojový kód 4.6: Převod názvu Maven modulu na IProject

4.2 Volání JaCC rozhraní a procházení získaných výsledků

Spouštění kontroly kompatibility vazeb mezi knihovnami a zpracování získaných výsledků se provádí bez závislosti na typu projektu. Dílčí kroky se tedy nachází v metodách, které jsou implementované v abstraktní třídě `AbstractCompatibilityChecker.java`. Zejména se jedná o tyto činnosti:

- Kontrola korektnosti vazeb – metoda `checkInterCompatibility`
- Procházení výsledků – metoda `exploreResults`
- Vytváření chybových hlášek – metoda `getWarningMessage` ze třídy `WarningMessageBuilder`

Kontrola korektnosti vazeb

Jak již bylo řečeno, kontrolu korektnosti vazeb provádí knihovna **JaCC**, a to pomocí metody `checkInterCompatibility`, která se nachází ve třídě `ApiIntercompatibilityChecker`. Parametry této metody nám připravila metoda `check` ze třídy `MavenCompatibilityChecker`, a tak je pro provedení kontroly vše připraveno. Provedeme ji následujícím způsobem:

```
1 ApiInterCompatibilityResult interCompatibilityResult =
2       ApiCheckersFactory
3           .getApiInterCompatibilityChecker()
4           .checkInterCompatibility(
5               appFiles,
6               libFiles,
7               classFilter
8           );
```

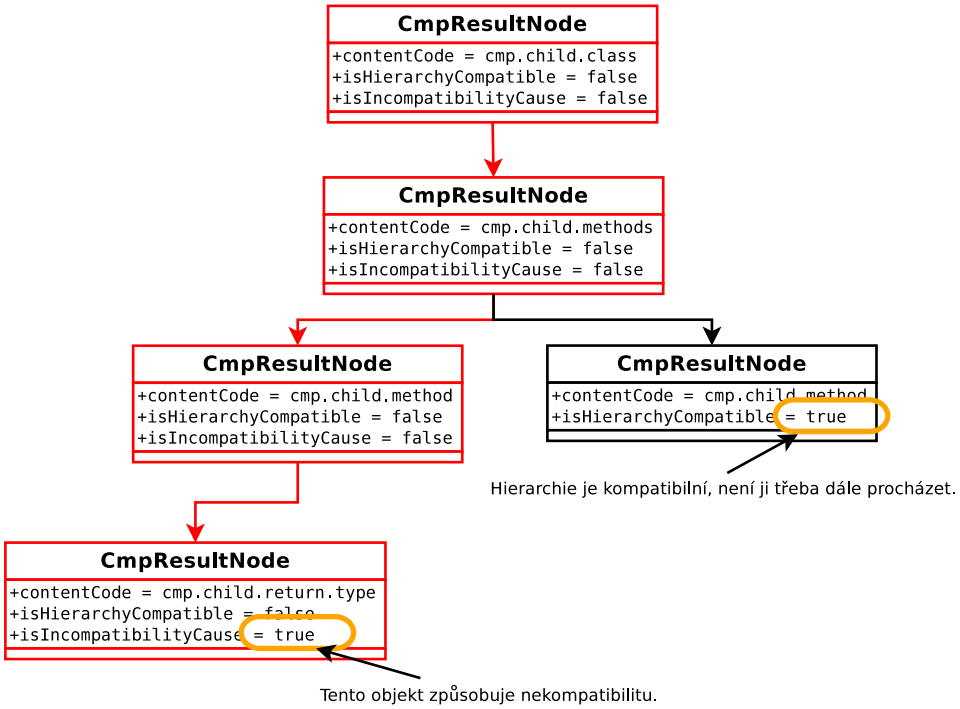
Zdrojový kód 4.7: Volání JaCC rozhraní

Ze získaného `interCompatibilityResult` lze dále získat pomocí metody `getOriginsImportingIncompatibilities` seznam knihoven (jar souborů), které obsahují nějaké nekompatibilní volání. Dále pro tyto knihovny lze získat konkrétní třídy importující nekompatibility a následně pro dvojici (**knihovna**, **třída**) pak již konkrétní výsledky. Jak to může vypadat v kódu je vidět v příloze B. Tyto výsledky obsahují veškeré nalezené chyby, které jsou volány (importovány) z dané třídy. Jedná se o stromovou strukturu, jejíž kořenem je objekt `CmpResultNode` zapouzdřující výsledky pro objekt typu `JClass`. Tento objekt obsahuje reference na své potomky, které jsou přístupné pomocí metody `getResult().getChildren()`. Jak může vypadat struktura získaných výsledků je vidět na obr. 4.3

Procházení získaných výsledků

Jedná se tedy o stromovou strukturu, kterou je nutné prohledat, abychom získali pouze ty uzly, které způsobují nekompatibilitu. Z těchto uzlů poté sestavíme chybovou hlášku, která bude informovat uživatele o nalezené nekompatibilitě. Jelikož při sestavování chybové hlášky bude někdy potřeba získat určité informace i z nadřazených uzlů, budeme strom prohledávat **do hloubky**[1]. To nám umožní udržovat si vždy kompletní cestu stromem vý-

[1]Prohledávání do hloubky (v angličtině označované jako depth-first search nebo zkratkou DFS) je algoritmus pro procházení grafů



Obrázek 4.3: Příklad struktury získaných výsledků

sledků od kořene (tzn. třídy, ve které se chyba nachází) až k aktuálnímu uzlu způsobující nekompatibilitu (např. návratovému typu).

Příkladem, kdy budeme potřebovat informace i z nadřazených uzlů, může být například situace znázorněná na obr. 4.3. Pokud budeme chtít vytvořit chybovou hlášku pro uzel reprezentující návratový typ metody, bude se jistě uživatel v této hlášce chtít dozvědět nejen o návratovém typu, ale např. i o názvu metody obsahující tento nekompatibilní návratový typ. Uzly, které budou k dispozici při konstruování chybové hlášky, jsou vyznačeny červenou barvou.

Vytváření chybových hlášek

Jednotlivé uzly stromové struktury vrácené JaCCem obsahují typicky tyto informace:

- First Object – Reprezentuje objekt, který je z importující knihovny vyžadován.

- Second Object – Reprezentuje objekt, který je ve skutečnosti vyžadovanou knihovnou poskytován.
- Difference (Rozdíl) – Rozdíl mezi *First Object* a *Second Object*.
- Strategy (Strategie opravy) – Říká, jak lze danou chybu opravit.

Chybové hlášky, nebo-li spíše varovné hlášky, jsou definovány v souboru `WarningMessages.properties`, a to vždy pro určitý *Content code* uzlu, který způsobuje nekompatibilitu. Tyto hlášky mohou obsahovat klíčová slova ohraničená symboly „{“ a „}“, která jsou následně nahrazena podle obsahů uzlů `CmpResultNode` nalezených při cestě z kořene stromu k uzlu způsobující nekompatibilitu. Příklad chybové hlášky pro nekompatibilní návratový typ:

```
cmp.child.method.return.type=
Method '${cmp.child.method}{firstObject.name}', wrong return type:
${cmp.child.method.return.type}{firstObject.name}
  -> ${cmp.child.method.return.type}{secondObject.name}.
```

Tato hláška může být přeložena např. do následující podoby:

```
Method 'next', wrong return type:
                 java.lang.String -> java.lang.Object.
```

Pravidla pro klíčová slova jsou tedy taková, že první je uveden symbol „$“ následovaný *Content codem* ohraničený symboly „{“ a „}“. Tento *Content code* říká, ze kterého `CmpResultNode`, nalezeného po cestě od kořene, se má přečíst příslušný atribut. Název tohoto atributu je uveden opět ve složených závorkách. Pro získání hodnoty uvedeného atributu je použito knihovny *BeanUtils* a její třídy *PropertyUtils*:

```
1 ChangeableCmpResult<?> result =
2 (ChangeableCmpResult<?>) node.getResult();
3 Object value = PropertyUtils.getProperty(result,
    attributeName);
```

Zdrojový kód 4.8: Použití PropertyUtils pro konstrukci chybové hlášky

V případě, že by tento mechanismus pro vytvoření varovné hlášky nestačil, je zde implementovaný ještě jeden, a to takový, který umožňuje ve varovné hlášce uvést název metody, která se zavolá, a která může na základě všech nalezených `CmpResultNode` produkovat libovolný textový výstup, který je do hlášky následně vložen.

4.3 Reportování nalezených chyb

Uživatelé budou informováni o nalezených nekompatibilitách pomocí již dříve zmíněných Markerů. O to se stará metoda `reportProblems`, která má následující signaturu:

```
reportProblems(Object project, String jar, ProblemInfo info);
```

Tato metoda je implementována opět ve třídě `MavenCompatibilityChecker`. Jedním z její parametrů je název souboru (knihovny), ve kterém byla daná nekompatibilita nalezena. K tomuto názvu je nutné dohledat artefakt obsažený v příslušném projektu, aby bylo možné najít umístění závislosti definované v `pom.xml`, na kterém bude posléze vytvořen Marker, informující uživatele o konkrétní chybě. Chyba se může nacházet v nějaké z tranzitivních závislostí, a tak je nutné pro tyto závislosti najít jejich „rodiče“, které již v `pom.xml` definované jsou. K tomu je využit `PathRecordingDependencyVisitor` a jednotlivé `DependencyNode` získané z grafu závislostí přísl. projektu:

```
for(int i = 0; i < dependencyNodes.size(); i++) {
  DependencyNode node = dependencyNodes.get(i);

  // Visitor for searching DependencyNode with errors
  PathRecordingDependencyVisitor visitor = new
   PathRecordingDependencyVisitor(filter);
  node.accept(visitor);

  // Path(s) to found node
  List<List<DependencyNode>> paths = visitor.getPaths();

  // If DependencyNode found
  if(paths.size() > 0) {
    createMarkers(info, reporter, pomFile, node, mvnProject
   );
    return;
  }
}
```

Zdrojový kód 4.9: Vyhledání „rodiče“ tranzitivní závislosti

Metoda `createMarkers` již nalezne umístění definice závislosti obsahující chyby (nebo umístění jejího rodiče) a pro tyto chyby vytvoří jednotlivé markery. Vyhledávání závislostí v `pom.xml` umožňuje metoda `findLocation`, kterou obsahuje *m2e* plugin.

```
SourceLocation sourceLocation =
  SourceLocationHelper.findLocation(mavenProject, node.
   getDependency());
```

Zdrojový kód 4.10: Vyhledání <dependency> tagu v pom.xml

Nyní již máme vše připraveno a je tedy možné příslušné markery vytvořit. To obstarává třída `MavenProblemReporter`. V jednotlivých markerech je možné uchovávat různé informace různých datových typů. Tyto informace jsou poté dostupné např. při přidávání třídy či balíku do blacklistu:

```
IMarker marker =
      pomFile.createMarker(JACC_MARKER_ID);
// store information about importer
marker.setAttribute(JACC_IMPORTING_JAR_NAME, importingJar);
marker.setAttribute(JACC_IMPORTING_CLASS_NAME,
   importingClass);
```

Zdrojový kód 4.11: Vytvoření Markeru a uložení atributů

4.4 Nové pohledy pro zobrazení výsledků

Pro přehledné zobrazení výsledků jsou vytvořeny dva nové pohledy. První slouží pro zobrazování nového, námi definovaného typu markerů a druhý pro přehledné zobrazení pozice nalezené chyby.

JaCC Markers Přidání nového pohledu pro zobrazení markerů je velmi jednoduché, většina jeho funkčnosti je definována v souboru `plugin.xml`:

```
<extension point="org.eclipse.ui.views">
   <category
         name="JaCC Plugin"
         id="cz.zcu.kiv.jacc.plugin">
   </category>

   <view
      class="cz.zcu.kiv.jacc.plugin.JaccMarkerView"
      id="cz.zcu.kiv.jacc.plugin.JaccMarkerView"
      icon="icons/jar.png"
      category="cz.zcu.kiv.jacc.plugin"
      name="JaCC Markers">
```

```
      </view>
   </extension>
```

Třída `JaccMarkerView` dědí od třídy `MarkerSupportView` a překrývá metodu `createPartControl`. V této metodě je mimo jiné registrován *listener*, který při vybrání jednoho z markerů překresluje pohled **JaCC Results**, který bude popsán v následujícím odstavci.

```
ISelectionProvider provider =
    getViewSite().getSelectionProvider();
if(provider != null) {
  provider.addSelectionChangedListener(
    new MarkersViewSelectionChangedListener()
  );
}
```

Zdrojový kód 4.12: Registrace listeneru nového pohledu

Dále je v `plugin.xml` nutné definovat, jaké sloupce bude tento pohled zobrazovat. Hlavním důvodem, proč se nepoužil standardní pohled pro zobrazování markerů, byla nemožnost přidání nových pluginem definovaných sloupců. Tyto sloupce zde tedy definujeme.

```
<extension  point="org.eclipse.ui.ide.markerSupport">
<markerField
  class="cz.zcu.kiv.jacc.plugin.fields.ImportedFromClassField"
  id="cz.zcu.kiv.jacc.plugin.fields.importedFromClass"
  name="Importing Class"/>

 <markerContentGenerator
   id="cz.zcu.kiv.jacc.plugin.JaccMarkerView"
   name="JaCC Content Generator">

   <markerTypeReference id="cz.zcu.kiv.jacc.problemmarker" />
   <markerFieldReference
     id="org.eclipse.ui.ide.severityAndDescriptionField" />
   <markerFieldReference
     id="cz.zcu.kiv.jacc.plugin.fields.importedFromClass" />

 </markerContentGenerator>
</extension>
```

Pro každý nový sloupec je vytvořena nová třída dědící od třídy `MarkerField`, jenž pro něj bude z konkrétního markeru získávat následujícím způsobem zobrazovanou hodnotu.

```
@Override
public String getValue(MarkerItem item) {
  if(item.getMarker().exists()) {
    String className =
    (String) item.getMarker().getAttribute(
      MavenProblemReporter.JACC_IMPORTING_CLASS_NAME
    );

    if(className != null) {
      return className;
    }
  }
}
```

Zdrojový kód 4.13: Ukázka metody getValue ze třídy ImportedFromClassField

JaCC Results Tento pohled slouží pro detailnější zobrazení polohy jedné konkrétní nalezené chyby. V novém pohledu pro zobrazování markerů jsme zaregistrovali *listener*, který při výběru jednoho ze zobrazených markerů tento pohled zaktualizuje (překreslí). Přidání pohledu proběhne stejně jako u předešlého s tím rozdílem, že veškerá jeho činnost je dána třídou `ResultsView` dědící od třídy `ViewPart`. Tato třída se stará o správné vykreslování stromu výsledků (resp. pouze jedné cesty v tomto stromu) příslušícího danému markeru.

Tento strom bylo nutné pomocí třídy `PluginDataHolder` nejprve uložit, aby jej bylo možné zpětně zobrazovat. Z důvodu optimálního využívání pamětního prostoru jej ale nelze v paměti uchovávat v té podobě, ve které nám ho vrátí JaCC, jelikož obsahuje mnoho referencí na různé JaCCem vytvořené objekty. Tyto objekty by tak *Garbage collector* nemohl z důvodu existujících referencí uvolnit, a JaCC by tak zabíral zbytečně moc pamětního prostoru. Navíc tento strom obsahuje mnoho pro nás již nepotřebných informací (chybové hlášky jsou v tuto chvíli již vytvořené). Pro naše potřeby je nutné v paměti uchovávat pouze textové popisky jednotlivých uzlů stromu.

Vykreslování stromu probíhá pomocí komponenty `TreeViewer`. Tato komponenta má nastaveny dva tzv. „providery“, které pro ni poskytují nezbytné informace na jejichž základě je možné strom vykreslit. Jedná se o:

- **Content provider** – Poskytuje obsah stromu, tzn. jednotlivé uzly a jejich potomky.
- **Label provider** – Poskytuje např. textové popisky, jejich barvy či ikony pro daný uzel.

Bližší informace o použití komponenty `TreeViewer` lze nalézt v literatuře [2] nebo [5].

4.5 Popis implementace blacklistu

Vzhledem ke snaze dodržení zvyklostí v prostředí Eclipse byl blacklist implementován jako tzv. „Quick Fix“, který lze vyvolat z kontextového menu jednoho z markerů nebo přímo ze zvýrazněné oblasti v určitém souboru – v našem případě z místa definice závislosti v `pom.xml`. Navíc jej lze z kontextového menu vyvolat nad více markery najednou, jelikož pro všechny existuje jedno společné řešení, a to **přidání do blacklistu**.

```
<extension point="org.eclipse.ui.ide.markerResolution">
  <markerResolutionGenerator
    markerType="cz.zcu.kiv.jacc.problemmarker"
    class="cz.zcu.kiv.jacc.plugin.actions.ResolutionGenerator" />
</extension>
```

Nejprve je nutné do `plugin.xml` přidat tzv. „Marker Resolution Generator“, který bude pro námi vytvořený typ markerů nabízet dvě řešení:

- Přidání názvu importující třídy do blacklistu
- Přidání názvu balíku, ve kterém se nachází importující třída, do blacklistu

Tato řešení jsou vytvářena ve třídě `ResolutionGenerator` implementující rozhraní `IMarkerResolutionGenerator` v překryté metodě `getResolutions`. Tato metoda vrací pole možný řešení – v našem případě dvou prvků typu `QuickFix`. Vybrané řešení je poté v této provedeno v této třídě, v metodě `run`.

```
public void run(IMarker[] markers, IProgressMonitor
    monitor);
```

Zdrojový kód 4.14: Signatura metody run ve třídě QuickFix

Prvním parametrem je pole markerů, nad kterými je je aktuální Quick Fix spuštěn. Hlavním úkolem je pro každý marker vyčíst hodnotu atributu, který uchovává informaci i importující třídě či balíku a tuto hodnotu poté zapsat do souboru `jacc-blacklist.txt` v aktuálním projektu pomocí třídy `BlackListSupport`, což se děje v metodě `addToBlacklist`.

```
String className = (String) marker.getAttribute(
  MavenProblemReporter.JACC_IMPORTING_CLASS_NAME);
String packageName = (String) marker.getAttribute(
  MavenProblemReporter.JACC_IMPORTING_PACKAGE_NAME);

IProject project =
  marker.getResource().getProject();

if(!addToBlacklistOnlyClass) {
  addToBlackList(blacklisted, packageName, project);
}
else {
  addToBlackList(blacklisted, className, project);
}


// delete marker imported from same class or package
deleteWithSameImporter(marker, className, packageName,
    addToBlacklistOnlyClass);
```

Zdrojový kód 4.15: Získání atributu z Markeru a přidání nové hodnoty do blacklistu

Metoda `deleteWithSameImporter` zajišťuje smazání všech ostatních markerů, jejichž atribut obsahující název importující třídy nebo balíku je shodný s příslušnými hodnotami aktuálně smazaného markeru. Jedná se vlastně o markery, které se při příštím spuštění kontroly kompatibility s upraveným blacklistem neobjeví.

4.6 Rozšíření o podporu jiných typů projektů

Při návrhu byl brán zřetel na možnost budoucího rozšíření pluginu o podporu jiných typů projektů. V podstatě by se mělo jednat o přidání jedné třídy podobné `MavenCompatibilityChecker`, která by připravovala potřebné vstupní parametry pro metodu `checkInterCompatibility` nacházející se v abstraktní třídě `AbstractCompatibilityChecker` a z ní získané výsledky by reportovala uživateli způsobem, který je pro daný typ projektu typický. Nově přidaná třída je vidět na obrázku 4.1.

Pokud by bylo potřeba zaregistrovat nějaký nový *listener* na začátku životního cyklu pluginu, je zde připraven enumerátor `ListenersEnum` obsahující abstraktní metodu `registerListener`, ve které lze registraci provést.

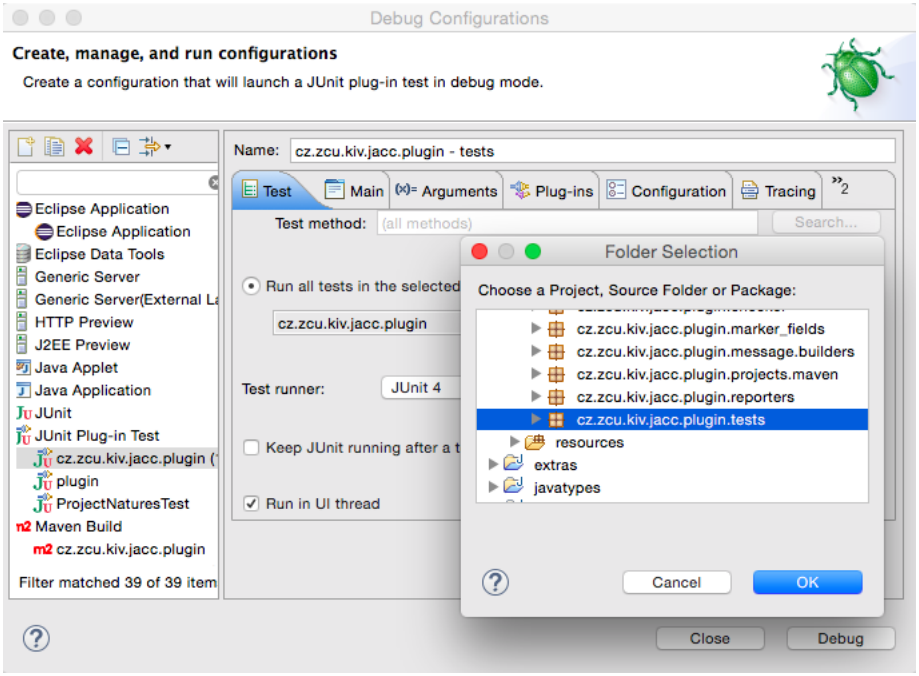
```
public enum ListenersEnum {
  MavenDependencyChanged() {
    void registerListener(final Logger logger) {
      // listener registration
    }
  };

  void register() {
    Logger logger =
      Logger.getLogger(ListenersEnum.class);
    registerListener(logger);
  }

  // listener registration
  abstract void registerListener(Logger logger);
}
```

Zdrojový kód 4.16: Výčet listenerů registrovaných na začátku životního cyklu pluginu

5 Testování

Fáze testování je v oblasti vývoje softwaru velmi důležitá. Mimo jiné slouží k ověření správné funkčnosti vyvinutého softwaru a k zafixování stávající funkčnosti. Během nového vývoje může být dosavadní funkcionalita porušena, což může být identifikováno právě díky existujícím testům.

Eclipse PDE nabízí možnost spouštění JUnit[1] testů pomocí tzv. *JUnit Plug-in Test Launcheru*. Po jeho vyvolání dojde ke spuštění nové instance Eclipse s testovacím workspace, nad kterým jsou následně spuštěny existující JUnit testy. Po dokončení všech testů je Eclipse, který byl spuštěn pro účely testování, ukončen. To, jaké testy budou vykonány, lze určit v konfiguraci launcheru viz obr. 5.1. Více informací o možnostech konfigurace spouštění lze najít v literatuře [2].



Obrázek 5.1: Konfigurace spouštění JUnit testů

Pro účely testování je tedy vždy spouštěno celé prostředí. Toto prostředí je nutné spouštět pro potřeby JaCCu s nastavenou systémovou proměnnou `JAVA_HOME`. To lze učinit v záložce **Environment** v konfiguraci na obr. 5.1.

[1] JUnit je framework pro vytváření jednotkových testů

V rámci této práce bylo při vývoji vytvořeno několik JUnit testů, které testují jak jednotlivé dílčí funkčnosti (metody), tak celkovou funkčnost manuálně spuštěné kontroly kompatibility. Všechny testy se nacházejí v balíku `cz.zcu.kiv.jacc.plugin.tests`.

TestUtils Pro potřeby všech testů je vždy nutné vytvořit v testovacím workspace nějaké testovací projekty, nad kterými budou testy probíhat. Pro tyto účely byla vytvořena třída `TestUtils`, ve které jsou implementovány pomocné metody pro vytváření projektů o zadaných parametrech. Byla zde také vytvořena metoda pro import existujícího Maven projektu, na kterém bude testována celková funkčnost pluginu.

BlacklistSupportTest Jedná se o test, který testuje správné čtení a vkládání do blacklistu, a zda se do něj neukládají duplicitní záznamy.

```
1  @Test
2  public void testAddToBlackList() throws CoreException,
       IOException {
3    IProject project = TestUtils.createTestProject("
       testProject1", new String[] { JavaCore.NATURE_ID });
4    String [] packages = {"package1", "package3", "package1",
       "package3", "package4"};
5    String [] expected = {"package1", "package3", "package4"
       };
6    for(String pckg : packages) {
7      BlackListSupport.addToBlackList(project, pckg);
8    }
9    ArrayList<String> real = BlackListSupport.
       getBlackListedPackages(project);
10   comparePackages(expected, real);
11 }
```

Zdrojový kód 5.1: Část JUnit testu BlacklistSupportTest

ProjectNaturesTest Tento test testuje správné vytváření tzv. „checkerů“, neboli potomků třídy `AbstractCompatibilityChecker` dle typu projektu. V této chvíli tedy pouze `MavenCompatibilityChecker` pro **Maven projekty** a **NULL** pro ostatní typy projektů. Po případném rozšíření pluginu o podporu jiných typů projektů by měl být rozšířen i tento test.

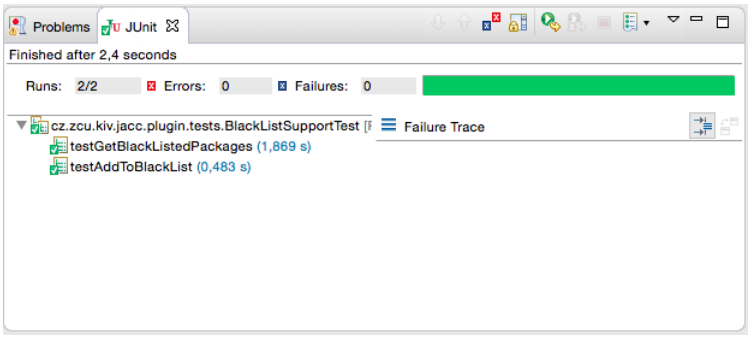
CheckCompatibilityTest Jedná se o automatický test, který testuje celkovou funkčnost pluginu na testovacím projektu. Na tomto projektu bude také ukázána funkčnost pluginu v kapitole 6 – Uživatelská dokumentace. Test se skládá z následujících dílčích kroků:

- import testovacího projektu a jeho modulů do testovacího workspace
- spuštění kontroly kompatibility
- počkání na dokončení předešlého kroku
- porovnání vytvořených Markerů s očekávanými

```
// --- import test projects
TestUtils.importTestProject();

IMavenProjectRegistry mavenProjectRegistry =
    MavenPlugin.getMavenProjectRegistry();
IMavenProjectFacade booking =
    mavenProjectRegistry.getMavenProject(
            "cz.zcu.kiv.examples",
            "booking",
            "1.0-SNAPSHOT");
IMavenProjectFacade server =
    mavenProjectRegistry.getMavenProject(
            "cz.zcu.kiv.examples",
            "server",
            "1.0-SNAPSHOT");
IMavenProjectFacade ratingLoader =
    mavenProjectRegistry.getMavenProject(
            "cz.zcu.kiv.examples.booking",
            "rating-loader",
            "1.0-SNAPSHOT");

// --- check interCompatibility for project booking
ExplorerClickActionFake clickAction =
    new ExplorerClickActionFake();
AbstractCompatibilityChecker checker =
    clickAction.click(booking.getProject());

Job job = checker.checkInBackground();
job.join();
```

Zdrojový kód 5.2: Část JUnit testu CheckCompatibilityTest

Spouštění testů Kliknutím pravým tlačítkem na příslušný test a výběrem *Run as* → *JUnit Plugin-in Test* lze daný test spustit. Průběh a výsledek je pak zobrazen v pohledu JUnit (viz obr. 5.2). Pokud chceme spustit všechny testy najednou, vyvoláme kontextové menu nad celým projektem a dále pokračujeme jako v předchozím případě.



Obrázek 5.2: Zobrazení výsledků JUnit testů

6 Uživatelská dokumentace

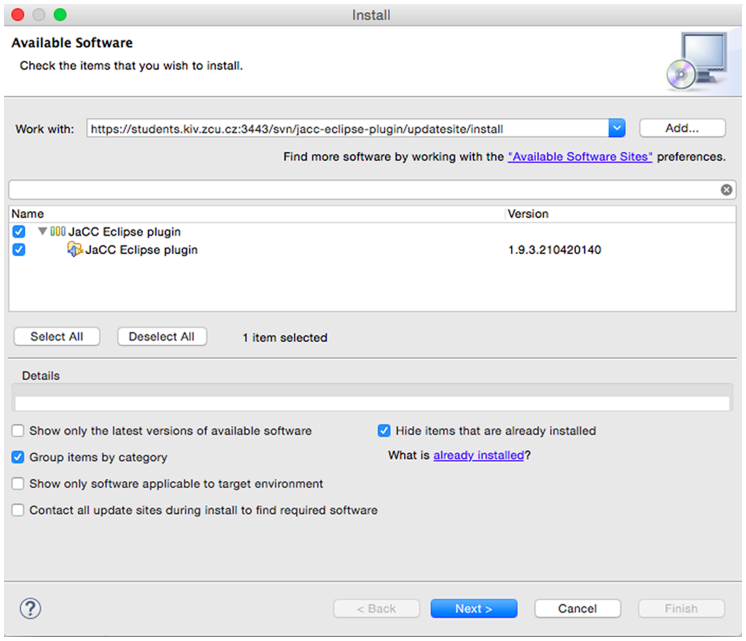
Při vývoji bylo využito verzovacího systému Subversion (SVN). Na tento repozitář byla umístěna také vyexportovaná *JaCC Eclipse Feature*, která lze do Eclipse IDE z repozitáře přímo nainstalovat.

6.1 Instalace pluginu a jeho minimální požadavky

Pro instalaci a správnou funkci JaCC Eclipse pluginu je nutné mít nainstalováno:

- Eclipse Luna
- M2Eclipse plugin ve verzi *1.5.0* (v případě nižší verze bude plugin aktualizován)

Instalaci pluginu lze snadno provést pomocí menu **Help** → **Install New Software** v hlavní nástrojové liště. Poté do textového pole „Work with“ vyplníme následující adresu a po vyžádání zadáme uživatelské jméno a heslo:

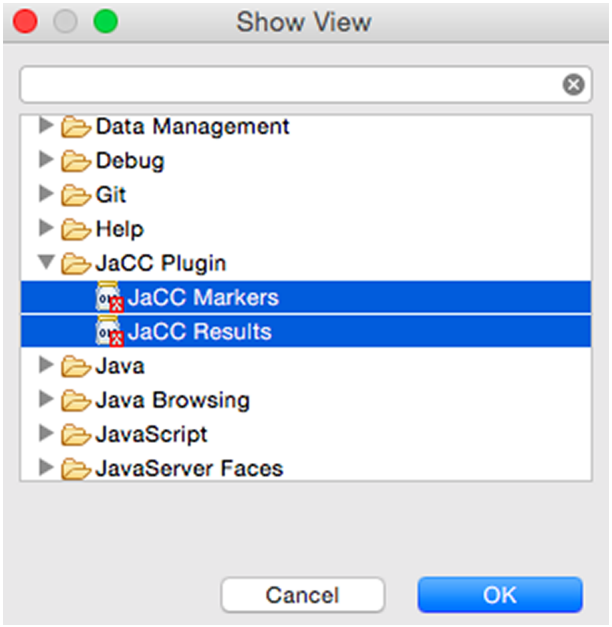


Obrázek 6.1: Instalace JaCC Eclipse pluginu

```
https://students.kiv.zcu.cz:3443/svn/jacc-eclipse-plugin/
updatesite/install
```

Vybereme nalezený **JaCC Eclipse plugin**, klikneme na tlačítko **Next**, potvrdíme „licenční podmínky“ a plugin nainstalujeme. Po úspěšné instalaci je nutné IDE restartovat.

Po restartování IDE bychom již měli v nabídce po kliknutí v hlavní nástrojové liště na **Window** → **Show View** → **Other...** vidět dva nové pohledy, které zobrazíme.

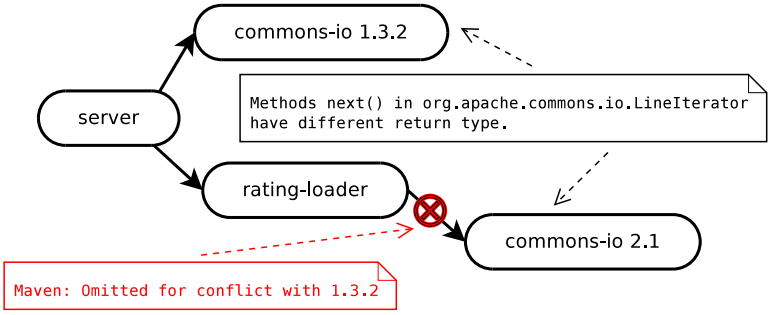


Obrázek 6.2: Dva nové pohledy. JaCC Markers a JaCC Results.

6.2 Ukázka funkčnosti na testovacím projektu

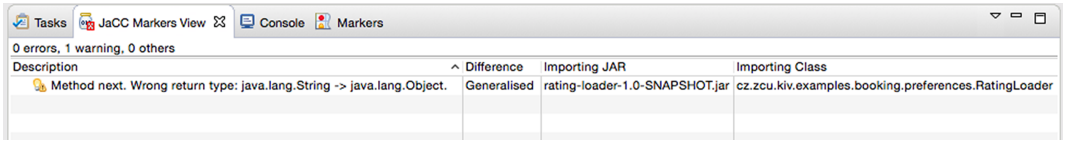
Funkčnost pluginu si ukážeme na projektu, na kterém je testován i Maven plugin. Jedná se o Maven projekt skládající se z „parent projektu“ s názvem **booking** a jeho dvou modulů: **rating-loader** a **server**. Modul *server* závisí na knihovně `commons-io` ve verzi *1.3.2*, ale modul *rating-loader* na verzi *2.1*. Z pohledu serveru zde nastane konflikt v těchto knihovnách a Maven tak použije tu ve verzi *1.3.2* (viz obr. 6.3).

Tyto verze ale nejsou zpětně kompatibilní, jelikož návratový typ metody `next()` ze třídy `org.apache.commons.io.LineIterator` se v těchto verzích liší. Tato chyba ve vývojovém prostředí (které nedisponuje JaCC pluginem) odhalena samozřejmě není, nicméně s použitím pluginu ji odhalit lze.



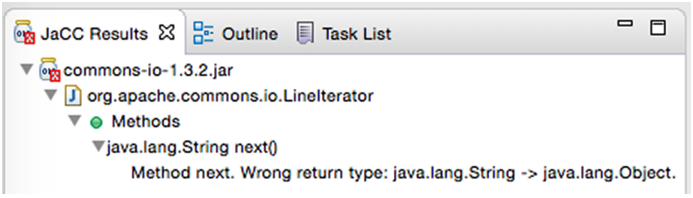
Obrázek 6.3: Chyba z pohledu modulu **server**.

Spustíme tedy kontrolu kompatibility kliknutím pravým tlačítkem myši na modul **server** v *Package Exploreru* a vybereme možnost **JaCC Plugin** → **Check compatibility**. V pravém spodním rohu obrazovky se nám objeví text **JaCC Compatibility checker: 0%**, který indikuje průběh kontroly. Po jejím dokončení se v pohledu **JaCC Markers** objeví výsledky viz obr. 6.4



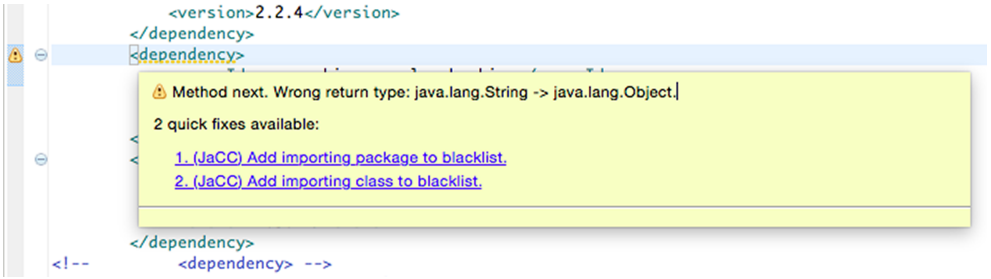
Obrázek 6.4: Nalezené nekompatibility zobrazené v JaCC Markers.

Po kliknutí na zobrazený záznam se nám v pohledu **JaCC Results** detailněji zobrazí místo výskytu dané nekompatibility, tak jak je vidět na obr. 6.5.



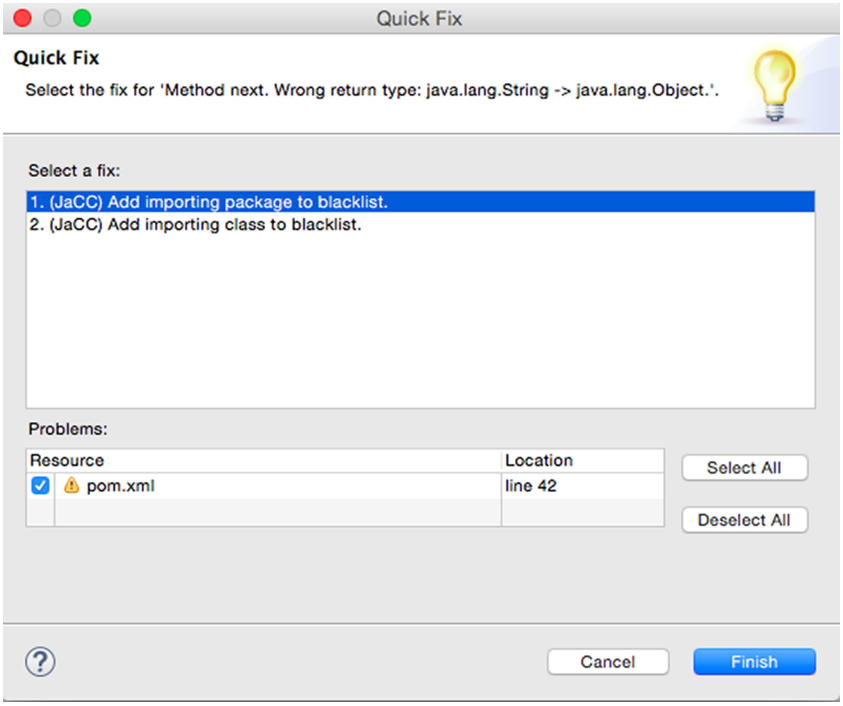
Obrázek 6.5: Zobrazení místa výskytu nalezené nekompatibility.

Pokud na záznam v **JaCC Markers** klikneme dvakrát, otevře se nám `pom.xml` modulu **server** se zvýrazněnou definicí závislosti, ze které je nalezená nekompatibilita importována. (viz obr. 6.6)



Obrázek 6.6: Zvýrazněná definice závislosti, ze které je nekompatibilita importována.

Pokud chceme, aby plugin nějakou z chyb ignoroval, přidáme ji do blacklistu, a to buď výběrem jedné z nabízených možností, jak je vidět na obr. 6.6 nebo kliknutím pravým tlačítkem myši na příslušný záznam v **JaCC Markers**, zvolením **Quick Fix** a následným výběrem jedné z nabízených možností.

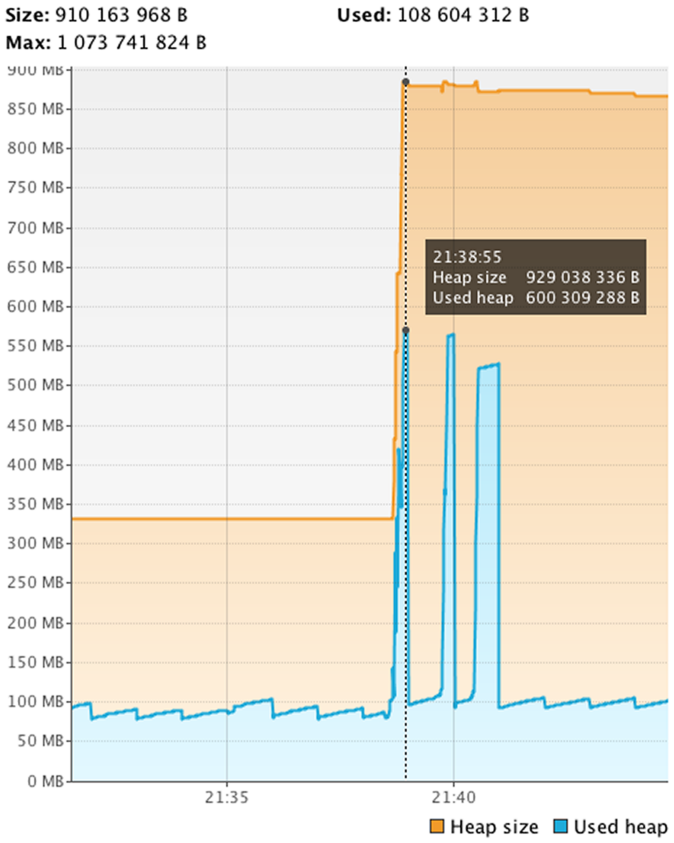


Obrázek 6.7: Nabízené řešení po kliknutí na Quick Fix.

Změníme-li verzi knihovny `commons-io` na *2.1* v modulu **server**, kontrola kompatibility bude automaticky spuštěna (vlivem změny závislosti) a plugin již žádné chyby nenalezne.

Kontrolu kompatibility lze také spouštět z rodičovského projektu. Do této kontroly jsou pak zahrnuty všechny jeho moduly, včetně jejich závislostí, přeložených zdrojových souborů a jednotlivých blacklistů.

Paměťová náročnost Náročnost pluginu co se týče paměťového prostoru byla monitorována při činnosti pluginu nástrojem **Java Visual VM**, který umožňuje mimo monitorování využití CPU nebo počtu aktivních vláken také monitorovat počet MB alokovaných na haldě. Jak je vidět na obrázku 6.8, IDE má při činnosti JaCC pluginu alokováno maximálně 600 MB. Tato velikost je ale dána konkrétním projektem, a tak pro jiný, větší projekt tato velikost může být vyšší.



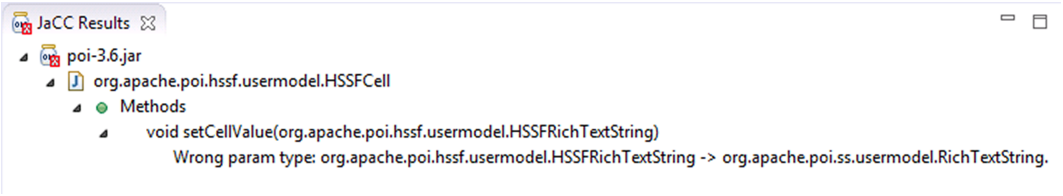
Obrázek 6.8: Využití paměti při opakované činnosti JaCC pluginu.

6.3 Otestování na reálném projektu

Plugin byl také otestován na skutečné aplikaci. Jedná se o webovou MVC aplikaci vyvíjenou společností **Aimtec**. Tato aplikace je vyvíjena jako Maven projekt, který je závislý na cca **190 knihovnách**.

Byla tedy otestována jak funkčnost, tak použitelnost pluginu. Co se týče funkčnosti, bylo ověřeno, že plugin funguje správně. V aplikaci bylo nalezeno několik nekompatibilit typu volání neexistujících metod či konstruktorů. Dále byly nalezeny chyby způsobené nekompatibilním návratovým typem a nebo nekompatibilním typem parametru metody. Pro tyto chyby byla zobrazena např. tato chybová hláška:

```
Wrong param type:
   org.apache.poi.hssf.usermodel.HSSFRichTextString
   -> org.apache.poi.ss.usermodel.RichTextString.
```



Obrázek 6.9: Nalezena nekompatibilita s metodou `setCellValue`

Tato metoda je volána z knihovny **JMesa**, která se v současné době začíná v této aplikaci hojně využívat. Lze tedy konstatovat, že díky vyvinutému pluginu byla zjištěna skutečnost, že pro určité případy není knihovna **JMesa** ve verzi **3.0.4** kompatibilní s knihovnou **Apache POI** ve verzi **3.6**.

Co se týče použitelnosti, ukázala se velká paměťová náročnost. Při použití pluginu na projektu těchto rozměrů (cca 190 závislostí) mělo Eclipse IDE alokováno cca 12 GB paměti.

7 Závěr

Všechny naskytnuté problémy se podařilo vyřešit, a tak bylo úspěšně vytvořeno rozšíření vývojového prostředí Eclipse. Toto rozšíření umožňuje detekovat statické chyby mezi použitými knihovnami v aktuálně vyvíjených Maven projektech, a to buď to po manuálním spuštění kontroly kompatibility (tj. kdy vývojář uzná za vhodné) nebo po automatickém spuštění. Automatické spuštění je nyní automaticky vyvoláváno po změně v definicích závislostí daného projektu, a tak může plugin detekovat chyby např. v nově přidaných knihovnách.

Vývoj tohoto pluginu přinesl vývojářům možnost detekovat tyto chyby v jimi vyvíjených projektech a odhalit je tak za pomoci vývojového prostředí již ve fázi vývoje. Díky tomu lze předcházet některým chybám vyskytujících se za běhu aplikace a tím zvýšit spolehlivost vyvíjeného software. O nalezených nekompatibilitách jsou vývojáři informováni prostřednictvím dvou nově vytvořených pohledů. První z nich je pohled zobrazující Markery popisující nalezené chyby. Druhý pak zobrazuje detailnější popis aktuálně vybrané chyby v prvním pohledu. Pro vyvinutý plugin byla vytvořena tzv. Update site, která byla umístěna do SVN repozitáře. Mimo toho, že tento repozitář byl použit při vývoji pluginu, je možné jeho prostřednictvím plugin instalovat do vývojových prostředí či dodávat aktualizace. Pro instalaci je tedy nutné znát pouze URL a uživatelské jméno s heslem příslušného repozitáře.

Správná činnost pluginu byla ověřena automatickými testy a manuálním vyzkoušením jak na testovacím projektu, který záměrně obsahoval určitou nekompatibilitu, tak na skutečném projektu. Činnost pluginu byla také monitorována profilerem, čímž bylo ověřeno správné uvolňování již nepotřebné alokované paměti.

Plugin umožňuje pracovat pouze s projekty typu Maven, a tak mezi možná vylepšení může určitě patřit rozšíření o podporu jiných typů projektů. Dalším vylepšením by mohlo být rozšíření uživatelského rozhraní pro manipulaci s blacklistem, jelikož nyní z něj lze odebírat záznamy pouze pomocí textového editoru. V neposlední řadě je třeba zmínit možnost detekce chyb tzv. „on the fly“. Jedním z možných pokračování tohoto projektu by mohl být průzkum (popř. implementace), jak detekovat chyby v průběhu psaní zdrojového kódu (tj. po uložení a automatickém překladu). Toto vylepšení by přineslo možnost odhalovat nekompatibility, bez nutnosti manuálního spuštění.

A Maven – Definice závislostí na nástroji JaCC

```
<dependencies>
   <dependency>
      <groupId>cz.zcu.kiv.jacc</groupId>
      <artifactId>javatypes</artifactId>
      <version>1.0.6</version>
   </dependency>

   <dependency>
      <groupId>cz.zcu.kiv.jacc</groupId>
      <artifactId>types-cmp</artifactId>
      <version>1.0.6</version>
   </dependency>

   <dependency>
      <groupId>cz.zcu.kiv.jacc</groupId>
      <artifactId>javatypes-cmp</artifactId>
      <version>1.0.6</version>
   </dependency>

   <dependency>
      <groupId>cz.zcu.kiv.jacc</groupId>
      <artifactId>javatypes-loader</artifactId>
      <version>1.0.6</version>
   </dependency>

   <dependency>
      <groupId>cz.zcu.kiv.jacc</groupId>
      <artifactId>compatibility-checker-utils</artifactId>
      <version>1.0.6</version>
   </dependency>
</dependencies>
```

B Příklad volání JaCC rozhraní

```
1  // ziskani nekompatibilnich jar souboru
2  Set<String> incompatible = compatibilityResult.getOriginsImportingIncompatibilities();
3  for(String importingJar : incompatible) {
4    // ziskani nekompatibilnich trid
5    Set<String> incClass = compatibilityResult.getClassesImportingIncompatibilities(
       importingJar);
6    for(String importingClass : incClass) {
7      // ziskani vysledku pro tridu importingClass a knihovnu importingJar
8      Set<ApiInterCmpResult> inc = compatibilityResult.getIncompatibleResults(
       importingClass, importingJar);
9      for(ApiInterCmpResult res : inc) {
10       // prochazeni ziskanych vysledku
11       exploreResults(...);
12     }
13   }
14
15   // reportovani nalezenych chyb
16   reportProblems(from, importingJar, info);
17 }
```

Literatura

[1] *Apache Maven*. The Apache Software Foundation, 2015. Dostupné z: `https://maven.apache.org/`.

[2] *Eclipse Documentation*. The Eclipse Foundation, 2014. Dostupné z: `http://help.eclipse.org/luna/`.

[3] *Eclipse Wiki*. The Eclipse Foundation, 2014. Dostupné z: `https://wiki.eclipse.org/`.

[4] ERIC CLAYBERG, D. R. *Building Commercial Quality Plug-ins Second Edition*. Addison Wesley, 2006.

[5] GRINDSTAFF, C. *How to use the JFace Tree Viewer*. Eclipse, 2002. Dostupné z: `https://eclipse.org/articles/Article-TreeViewer/TreeViewerArticle.htm`.

[6] JEZEK, K. et al. Software Components Compatibility Verification Based on Static Byte-Code Analysis. In *SEAA, 39th EUROMICRO*, s. 145 – 152, 2013.

[7] *Java Documentation*. Oracle, 2015. Dostupné z: `https://docs.oracle.com/en/java/`.

[8] *Loading, Linking, and Initializing*. Oracle, 2015. Dostupné z: `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html`.

[9] *The OSGi Architecture*. OSGi™ Alliance, 2015. Dostupné z: `http://www.osgi.org/Technology/WhatIsOSGi`.

Seznam obrázků

1.1 Současná situace. Dosud je při vývoji možné využít pouze Maven plugin. 2
1.2 Rozsah kontroly prováděné překladačem. Nekorektní vazba 1 je odhalena již při překladu. Nekorektní vazbu 2 již překladač neodhalí a tato nekompatibilita způsobí chybu až za běhu aplikace. Zdroj: [6] 3
1.3 Hierarchie Maven závilostí, jak jsou zobrazeny v prostředí Eclipse 4

2.1 Stručný pohled na Eclipse platformu 7
2.2 „Extension point“ je definovaný jedním pluginem, ale využívat jej může více pluginů. 8
2.3 Základní prvky uživatelského rozhraní 9
2.4 Průvodce pro přidání nového rozšíření 10
2.5 Souvislosti mezi *Plugin*, *Feature* a *Update site*. 13
2.6 Obsah vygenerované Update site 15

3.1 Analyzovaná oblast. 16
3.2 Pohled **Markers** používaný v Eclipse 22
3.3 Zobrazení „Quick Fix“ v editoru zdrojového kódu 24

4.1 Vztah mezi třídami AbstractCompatibilityChecker, MavenCompatibilityChecker a možnost rozšíření podpory o jiné typy projektů. 27
4.2 Jednotlivé činnosti při kontrole kompatibility. 27
4.3 Příklad struktury získaných výsledků 32

5.1 Konfigurace spouštění JUnit testů 41
5.2 Zobrazení výsledků JUnit testů 44

6.1 Instalace JaCC Eclipse pluginu 45
6.2 Dva nové pohledy. JaCC Markers a JaCC Results. 46

6.3 Chyba z pohledu modulu **server**. 47
6.4 Nalezené nekompatibility zobrazené v JaCC Markers. 47
6.5 Zobrazení místa výskytu nalezené nekompatibility. 47
6.6 Zvýrazněná definice závislosti, ze které je nekompatibilita importována. 48
6.7 Nabízené řešení po kliknutí na Quick Fix. 48
6.8 Využití paměti při opakované činnosti JaCC pluginu. 49
6.9 Nalezena nekompatibilita s metodou `setCellValue` 50

Seznam zdrojových kódů

2.1 Ukázka obsahu `artifacts.xml` 13
2.2 Ukázka obsahu `content.xml` 14
3.1 Signatura metody checkInterCompatibility poskytované JaCCem 18
4.1 Zjištění zda projekt je typu Maven 25
4.2 Vytvoření a naplánování Eclipse jobu 26
4.3 Maven – Vytvoření grafu závislostí 28
4.4 Získání projektu z Maven cache 29
4.5 Načtení projektu do Maven cache 30
4.6 Převod názvu Maven modulu na IProject 30
4.7 Volání JaCC rozhraní 31
4.8 Použití PropertyUtils pro konstrukci chybové hlášky 33
4.9 Vyhledání „rodiče“ tranzitivní závislosti 34
4.10 Vyhledání <dependency> tagu v pom.xml 35
4.11 Vytvoření Markeru a uložení atributů 35
4.12 Registrace listeneru nového pohledu 36
4.13 Ukázka metody getValue ze třídy ImportedFromClassField . . 37
4.14 Signatura metody run ve třídě QuickFix 39
4.15 Získání atributu z Markeru a přidání nové hodnoty do blacklistu 39
4.16 Výčet listenerů registrovaných na začátku životního cyklu pluginu 40
5.1 Část JUnit testu BlacklistSupportTest 42
5.2 Část JUnit testu CheckCompatibilityTest 43