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

Diplomová práce

Vylepšené ověřování kompatibility komponent v OSGi

Plzeň, 2014 Jan Šváb

Prohlášení

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

V Plzni dne 14. května 2014

Jan Šváb

Poděkování

Děkuji Ing. Kamilu Ježkovi, Ph.D. za vedení mé diplomové práce a za podněty a návrhy, které ji obohatily a umožnily její úspěšné dokončení.

Jan Šváb

Abstract

Enhanced component compatibility verification in OSGi

Component-based programming is characterized by emphasis on reusability of code. Applications are composed from components. Functional requirements from component are connected to functionality provided from another components. Special specification of component interface is used for creating these connections.

Even if these dependencies are satisfied, components can still be nonfunctional. The reason for this is that extra-functional properties are not considered. They are not part of the interface specification. This is a big disadvantage especially for third-party components. Unsatisfied extra-functional properties can cause for example issues with reliability, performance or security of applications.

This work aims to demonstrate on selected component model, namely OSGi Service Platform, how definitions of functional dependencies can be enhanced by extra-functional properties to allow better verification of component compatibility.

Abstrakt

V komponentově orientovaném programování je kladen důraz na maximální znovupoužitelnost kódu. Aplikace jsou skládány z komponent. Skládání v principu probíhá propojováním požadavků na funkcionalitu komponenty s poskytovanou funkcionalitou od jiných komponent podle speciální specifikace rozhraní komponenty.

I když budou splněny všechny tyto závislosti, komponenty stále můžou být nefunkční. Nemusí být totiž splněny tzv. mimofunkční charakteristiky, které nejsou nijak v rozhraní komponenty popsány. Specifikace mimofunkčních charakteristik je ale obzvlášť při používání komponent od třetích stran velice důležitá. Jejich nesplnění může vést ke vzniku aplikací, které mají problémy například s výkonem, spolehlivostí nebo bezpečností.

Cílem této práce je demonstrovat na vybraném komponentovém modelu, konkrétně OSGi Service Platform, jak je možné funkční závislosti rozšířit o popis mimofunkčních charakteristik a tím umožnit kvalitnější ověřování kompatibility komponent.

Obsah

1 Úvod **1**

2 CBSE **2**

3 Technologie OSGi **3**
3.1 Specifikace závislostí mezi bundly 3
3.1.1 Hlavičky pro identifikaci bundlu 4
3.1.2 Externí závislosti 4
3.1.3 Java package 5
3.1.4 Generické Capabilities a Requirements 6
3.1.5 Require-Bundle 7
3.2 Instalace bundlu do OSGi frameworku 7
3.2.1 Propojování Capabilities a Requirements 8
3.2.2 Propojování packagů 8
3.2.3 Propojování bundlů 9
3.2.4 Služba Resolver Hook 10

4 Stav projektu **11**
4.1 Model mimofunkčních charakteristik 12
4.2 Jádro modulu *EFP Assignment* pro OSGi 14
4.2.1 Implementace submodulu pro správu souborů komponenty 15
4.2.2 Submodul pro práci s prvky rozhraní komponenty . . . 16
4.2.3 Implementace submodulu pro správu lokálního úložiště 17
4.2.4 Submodul pro práci s EFP přiřazeními 18

5 Navázání EFP na OSGi resolver **19**
5.1 Atributy u packagů 19
5.2 Generické Capabilities a Requirements 19
5.3 Konverze hodnot datových typů EFP 20

6 Implementace **23**
6.1 Vedlejší úpravy 23
6.1.1 Práce s bundlem - odstranění násobného otevírání . . . 23
6.1.2 Práce s bundlem - zavedení indikace provedených změn 24
6.1.3 Parser hlaviček z manifestu 25
6.1.4 Řetězcový identifikátor pro globální a lokální registr . . 26
6.2 Přepracování submodulu `OSGiEfpDataManipulator` 26
6.3 Konvertování EFP přiřazení 28
6.4 Parsování LDAP filtru 30
6.5 Konverze datových typů hodnot EFP 32

7 Ověření funkčnosti **33**
7.1 Výkonnostní testy 33
7.2 Ověření nového způsobu zápisu přiřazení do manifestu 34
7.2.1 Přiřazení EFP s hodnotou z lokálního registru I 35
7.2.2 Přiřazení EFP s hodnotou z lokálního registru II 36
7.2.3 Přiřazení EFP s prázdnou hodnotou a matematickou formulí 37
7.3 Ověření funkčnosti v OSGi frameworku 38
7.3.1 Kompatibilní komponenty 39
7.3.2 Nekompatibilní komponenty 39

8 Návrhy na vylepšení **43**
8.1 Výpočet matematických formulí 43
8.2 Problém výběru lokálních registrů 46
8.3 Úprava definice EFP přiřazení v manifestu 47

9 Závěr **49**

Přílohy **52**

A Struktura XML s lokálním úložištěm **53**

B Překlad a spuštění **54**

C Obsah přiloženého CD **55**

1 Úvod

Hlavním důvodem pro vznik komponentově orientovaného programování (dále zkráceně CBSE [1]) bylo maximalizovat znovupoužitelnost kódu, a snížit tak celkové náklady na vývoj.

Aplikace jsou skládány z množiny základních bloků, tzv. komponent. Propojit lze pouze komponenty, které mají kompatibilní rozhraní. Toho je v případě komponent dosaženo speciálním popisem rozhraní, který se musí řídit stanovenými pravidly. Součástí popisu je kromě identifikace komponenty popis potřebné funkcionality z ostatních komponent a popis komponentou poskytované funkcionality.

Splnění těchto funkčních závislostí nemusí stačit, důležité je také, jak dobře je daná funkcionalita poskytována. To, že do aplikace programátor použije komponentu, a získá tak požadovanou funkčnost, neznamená, že aplikace poté funguje správně. Pokud například komponenta bude mít velmi vysoké nároky na pamět' nebo používat jiné šifrování, aplikace v lepším případě bude náročnější na výkon stroje a bude pomalejší, v horším případě bude vracet nesprávný výstup nebo bude docházet k jejímu pádu. Jinak řečeno, nemusí být splněny tzv. mimofunkční charakteristiky.

Vzhledem k tomu, že v CBSE je běžné používání komponent třetích stran, je popis komponent z pohledu mimofunkčních charakteristik důležitý. Proto by měly být součástí popisu rozhraní komponenty, aby mohly být použity nástrojem pro vyhodnocování jejich kompatibility.

Myšlenka rozšíření popisu rozhraní komponenty mimofunkčními charakteristikami byla pro komponentovou technologii OSGi realizována v projektu z roku 2011 [8]. Realizace má nevýhodu v tom, že veškeré ověřování kompatibility z pohledu mimofunkčních charakteristik probíhá v externím nástroji mimo OSGi.

Cílem této práce je analyzovat možnosti technologie OSGi a upravit nástroj pro přiřazování mimofunkčních charakteristik ke komponentám tak, aby mohly být použity automaticky při ověřování kompatibility komponent přímo v OSGi bez nutnosti používat externí nástroj.

[1] Component-Based Software Engineering

2 CBSE

Komponenta může být definována několika způsoby. Tato práce bude vycházet z definice Szyperskiho [7]. Softwarová komponenta je jednotka skládání s jedním nebo více rozhraními a pouze explicitními závislostmi. Jsou vytvářeny tzv. kontrakty. Softwarová komponenta může být vyvíjena a distribuována nezávisle a je objektem skládání komponentových aplikací třetími stranami.

Kontrakty se komponenta zavazuje poskytnout určité služby a zároveň specifikuje nutné závazky od klienta a prostředí, které komponenta k poskytnutí svých služeb potřebuje. Kontrakty zajišťují nezávislý vývoj komponent a očekávané interakce komponent po nasazení do běhového prostředí. Explicitní závislosti jsou uvedeny ve speciálním popisu rozhraní komponenty, který je distribuován s ní. K popisu je používán programovací jazyk, nebo speciální jazyk zavedený přímo pro popis rozhraní, tzv. IDL[1].

Komponenta v závislosti na programovacím jazyku reprezentuje například package, modul, namespace nebo třída. Kromě funkcionality poskytované přes rozhraní je zbytek implementace komponenty pro uživatele skrytý.

Přesnou podobu komponenty a jejího rozhraní popisuje komponentový model. Komponentový model specifikuje [1] standardy a konvence, kterými se musí vývojáři komponent řídit. Zároveň popisuje životní cyklus komponenty a pravidla, která musí být splněna pro interakce mezi komponentami.

Komponentový framework je implementací komponentového modelu. Lze si ho představit jako zjednodušený operační systém. Framework spravuje zdroje poskytované jednotlivými komponentami, které zprostředkovává komponentám, jež je pro svou funkčnost požadují.

Samotný vývoj komponentové aplikace může probíhat tak, že na začátku se aplikace postupně rozdělí na dostatečně primitivní logické celky, komponenty. Pro každou komponentu jsou pak definovány služby, které bude poskytovat a které bude požadovat. V další fázi jsou implementovány komponenty, které se nepodařilo mezi již existujícími komponentami nalézt. Ve chvíli, kdy jsou všechny komponenty z modelu aplikace k dispozici, proběhne sestavení a výsledná aplikace poté může být nasazena.

[1] Interface Definition Language

3 Technologie OSGi

OSGi je [6] množina specifikací dynamického, komponentového systému pro jazyk Java. Mezi nejznámější implementace tohoto komponentového modelu patří frameworky Equinox[1] a Felix[2].

OSGi [6] definuje jednotku modularizace nazývanou bundle. Bundle je komprimovaný soubor JAR[3] s Java třídami a dalšími zdroji, které společně poskytují funkcionalitu konečným uživatelům.

V rámci této práce bylo nutné se ve specifikaci zaměřit na část, která popisuje, jak jsou definovány závislosti mezi jednotlivými bundly. Tedy poskytované a požadované prvky rozhraní, ke kterým budou přiřazovány mimofunkční charakteristiky. Dále bylo nutné zjistit, jak probíhá zavádění bundlů a jaké existují potenciální možnosti, jak ovlivnit tuto operaci vhodným připojením mimofunkčních charakteristik.

3.1 Specifikace závislostí mezi bundly

Ke specifikaci rozhraní bundlu je používán textový soubor uvnitř JAR na cestě `META-INF/MANIFEST.MF`. Manifest se skládá z hlaviček, což jsou dvojice jméno-hodnota. OSGi definuje vlastní množinu hlaviček, které obsahují:

1. Identifikaci bundlu.
2. Umístění aktivátoru, implementace základního rozhraní pro zajištění životního cyklu bundlu. Jak má být bundle zaveden po spuštění frameworku.
3. Hlavičky s metadaty pro OSGi resolver specifikující závislosti, které musí být splněny před spuštěním bundlu.

Vzhledem k množství hlaviček v manifestu, budou popsány ve stručnosti pouze ty, které se týkají této práce. Manifest bundlu může vypadat například

[1] http://www.eclipse.org/equinox/

[2] http://felix.apache.org/site/index.html

[3] Java Archive - soubor s uloženými aplikacemi a jejich zdroji komprimovaný standardním formátem ZIP

následovně:

```
Manifest-Version: 1.0
Export-Package: cz.zcu.kiv.osgi.example.inventory.
    enterprisequeryif
Bundle-Name: Bundle Example
Bundle-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Activator: cz.zcu.kiv.osgi.example.inventory.Activator
Import-Package: cz.zcu.kiv.osgi.example.inventory.
    inventorydatabase.jdbc,org.osgi.framework;version="[1.5,2)"
Bundle-SymbolicName: cz.zcu.kiv.osgi.example.InventoryData
```

3.1.1 Hlavičky pro identifikaci bundlu

Název bundlu se nachází v povinné hlavičce `Bundle-SymbolicName`. Spolu s informací o verzi bundlu, která je uložena v hlavičce `Bundle-Version`, tvoří unikátní identifikátor bundlu.

3.1.2 Externí závislosti

Každý bundle je závislý na několika externích objektech. Pro své spuštění vyžaduje splnění všech těchto závislostí, které jsou definovány v manifestu. Tyto závislosti dělí OSGi do dvou skupin: Requirements (požadavky) a Capabilities (schopnosti). Mezi závislosti patří také prvky rozhraní komponenty, tudíž poskytované a požadované Java package bundlu. V manifestu jsou definovány externí závislosti pomocí následujících hlaviček:

- `Export-Package` - bundlem poskytované Java package.
- `Import-Package` - bundlem požadované Java package od jiných bundlů.
- `Provide-Capabilities` - množina bundlem poskytovaných generických Capabilities.
- `Require-Capabilities` - množina bundlem požadovaných generických Requirements od jiných bundlů.
- `Require-Bundle` - specifikuje bundle, ze kterého budou získány všechny poskytované Java package.

Základní syntaxe pro výše uvedené hlavičky je následující:

```
header ::= clause ( ',' clause )*
clause ::= dependency ( ';' parameter )*
parameter ::= directive | attribute ( ';' parameter )*
```

Část `clause` obsahuje definici jedné závislosti v hlavičce. V případě více závislostí jsou jejich definice oddělovány čárkami. Jako první je v definici v části `dependency` uveden identifikátor závislosti, za který jsou připojovány její parametry oddělené středníky. Parametry se dělí na direktivy a atributy.

Atributy jsou využívány převážně k uložení identifikačních údajů (např.: verze, vlastnosti) a direktivy specifikují, jak má být s prvky zacházeno při vyhodnocování OSGi resolverem. Tyto dva typy parametrů jsou rozlišeny symbolem pro přiřazení:

$$jmeno_atributu = hodnota$$
$$jmeno_directivy := hodnota$$

OSGi specifikuje, jak mají být vytvářeny atributy s verzemi. Hodnota se skládá ze sekvence tří číslic oddělených tečkami bez mezer a čtvrté volitelné textové části, tzv. kvalifikátoru. Defaultní hodnota verze je `0.0.0`. Dvě verze se shodují, pokud vzájemně odpovídají jednotlivé části.

Rozsah verzí specifikuje interval možných verzí u požadovaných zdrojů. Pokud obsahuje pouze jedinou verzi, pak musí být interpretován jako interval ve tvaru $[verze, \infty)$. Pokud není rozsah specifikován, je předpokládán interval $[0.0.0, \infty)$. Příklad intervalu verzí:

$$[1.2.3, 4.1.3)$$

3.1.3 Java package

V případě hlaviček pro poskytované a požadované Java package (dále pouze package) může část `dependency` obsahovat více jmen packagů oddělených středníkem. Následující parametry jsou určeny pro všechny tyto package. U poskytovaných packagů lze atributem `version` specifikovat verzi. U požadovaných packagů tento atribut obsahuje interval verzí, ve kterém musí být poskytovaný package. Dalšími atributy jsou například `bundle-symbolic-name` a `bundle-version` se symbolickým jménem a verzí bundlu, který exportuje požadovaný package. Příklad:

```
Export-Package: com.foo;version=1.2.3
Import-Package: com.foo;version="[1.1,1.2.4)"
```

K definici packagu lze také připojit vlastní volitelné atributy. Tyto atributy jsou běžně ignorovány při propojování packagů OSGi resolverem. Aby výsledek pokusu o propojení dvou packagů na nich závisel, musí být vyjmenovány ve speciální direktivě `mandatory` na poskytované straně. V definici požadovaného package poté musí být tyto atributy s odpovídajícími hodnotami také uvedeny.

3.1.4 Generické Capabilities a Requirements

Jsou součástí definice rozhraní od verze OSGi 4.3. Capabilities definuje [6] jako množiny atributů ve specifickém jmenném prostoru (namespace) a Requirements jsou výrazy pro filtrování, které jsou uplatňovány na atributy Capabilities z odpovídajícího jmenného prostoru. Jmenný prostor zároveň deklaruje význam porovnávaných atributů. Pro OSGi je rezervováno několik jmenných prostorů, které vždy začínají předponou `osgi`. Při definici vlastního jmenného prostoru je doporučeno tyto názvy registrovat u organizace OSGi Alliance, která OSGi specifikaci spravuje.

Definice požadovaných a poskytovaných packagů jsou určeny pro závislosti programových kódů, nejsou ale vhodné pro popis nekódových závislostí, pro které nemělo do verze 4.3 OSGi žádný přímý prostředek. Typickým použitím je specifikace závislostí deklarativních služeb (declarative services), které nejsou součástí popisu rozhraní bundlu v manifestu. Mohou být ale použity ke specifikaci jakékoliv nekódové obecné závislosti, proto označení generické. Mezi tyto závislosti se řadí také mimofunkční charakteristiky.

U Capabilities lze použít typové atributy, čímž se deklaruje, jak mají být hodnoty atributů porovnávány mezi Capabilities a Requirements. Syntaxe pro typové parametry je následující:

```
attribute ::= name(':' type) '=' argument
type ::= scalar | list
scalar ::= 'String' | 'Version' | 'Long' | 'Double'
list ::= 'List<' scalar '>'
```

Pokud není typ uveden, je zacházeno s atributem jako s řetězcem. Příklady typových atributů jsou:

```
a:Double = 20.8
b:String = "valueA"
c:List<Double> = "10,20,30"
```

Požaduje-li bundle nějakou Capability, je do hlavičky `Require-Capability` přidána klauzule se shodným názvem jmenného prostoru a s direktivou `filter`, ve které se výrazem popíše, jaké atributy musí mít hledaná Capability. Syntaxe výrazu je založena na řetězcové podobě LDAP vyhledávacích filtrů podle specifikace [3].

3.1.5 Require-Bundle

Pokud bundle vyžaduje všechny package jiného bundlu, je možné využít tuto hlavičku bez nutnosti vyjmenovávat jednotlivé package v hlavičce `Import-Package`. Syntaxe této hlavičky je následující [6]:

```
Require-Bundle ::=
     bundle-description (',' bundle-description)*
bundle-description ::= symbolic-name ( ';' parameter )*
```

Jednotlivé definice požadovaných bundlů jsou oddělovány čárkou. V každé definici za názvem bundlu může být uvedena množina parametrů oddělených středníky.

Lze použít například atribut `version` s rozsahem možných verzí požadovaného bundlu.

3.2 Instalace bundlu do OSGi frameworku

Pro každý bundle je po instalaci vytvořen speciální kontejner, nazývaný Bundle Revision, který poskytuje třídy a ostatní zdroje z bundlu. Součástí kontejneru je množina nalezených Capabilities bundlu a Requirements. Framework poté propojuje Requirements bundlu s dostupnými Capabilities již vyřešených bundlů. Package jsou v této fázi mapovány do Capabilities a Requirements ve jmenném prostoru `osgi.wiring.package`. Podobně jsou mapovány přímé závislosti na úrovni bundlů, které používají jmenný prostor `osgi.wiring.bundle`.

Jakmile jsou všechny závislosti uspokojeny, přejde bundle do stavu vy-

řešen, ke kontejneru jsou připojeny informace o všech provedených spojení Requirements s vyhovujícími Capabilities a bundle může být od té chvíle používán v aplikaci.

3.2.1 Propojování Capabilities a Requirements

K Requirement musí být nalezena nějakým již vyřešeným bundlem poskytovaná Capability, která se nachází ve stejném jmenném prostoru, a filtru musí odpovídat množina atributů Capabilities. Například bundle bude mít Requirement:

```
Require-Capabilities:
    com.cap; filter:="(&(a >= 10)(b = valueA))"
```

Tato závislost bude splněna pouze, jestliže nějaký vyřešený bundle bude definovat například tuto Capability:

```
Provide-Capabilities:
    com.cap; a:Double=20.8; b="valueA"
```

3.2.2 Propojování packagů

Poskytované package se musí nacházet v rozsahu verzí v definici požadovaných bundlů. Například pro definice

```
A: Import-Package: com.foo;version="[1.2 , 2.3]"
B: Export-Package: com.foo;version=1.2.3
C: Export-Package: com.foo;version=3.2
```

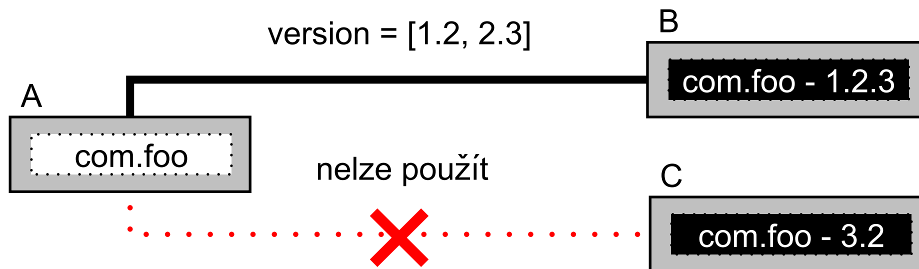
dopadne propojování podle obrázku 3.1.

Také musí být nalezena shoda v atributech, které jsou uvedeny v direktivě `mandatory` v definici poskytovaného package. Příklad, kdy k propojení může dojít:

```
Export-Package: com.foo;port=80;mandatory:=port
Import-Package: com.foo;port=80
```

Příklad, kdy propojení nelze realizovat, protože atribut `port` je specifikován jako povinný:

```
Export-Package: com.foo;port=80;mandatory:=port
```



Obrázek 3.1: Propojování s ohledem na verze

```
Import-Package: com.foo
```

Pokud propojení není u některého požadovaného package při zavádění povinné, lze to v jeho definici specifikovat direktivou `resolution` s hodnotou `optional`, například:

```
Import-Package: com.foo;resolution:=optional
```

Poskytování package z bundlu může být podmíněno splněním závislostí některých bundlem požadovaných packagů. Tuto podmínku lze definovat direktivou `uses`, v jejíž hodnotě budou tyto package vyjmenovány.

3.2.3 Propojování bundlů

Vyhovující požadovaný bundle musí mít specifikovaný symbolický název a musí být splněny omezení verzí podobně jako u packagů. Obdobně lze direktivou `resolution` specifikovat, jestli závislost má být vyřešena při zavádění bundlu.

Hromadné požadování packagů touto cestou nese svá rizika. Typickým rizikem je požadavek na více bundlů, kde vyhovující bundly exportují stejný package. Preferovanou cestou je tedy specifikovat závislosti hlavičkami `Export-Package` a `Import-Package`.

3.2.4 Služba Resolver Hook

Tato služba je určena pro ovlivnění operace propojování Requirements s Capabilities třetí stranou. Byla zavedena v OSGi specifikaci verze 4.3.

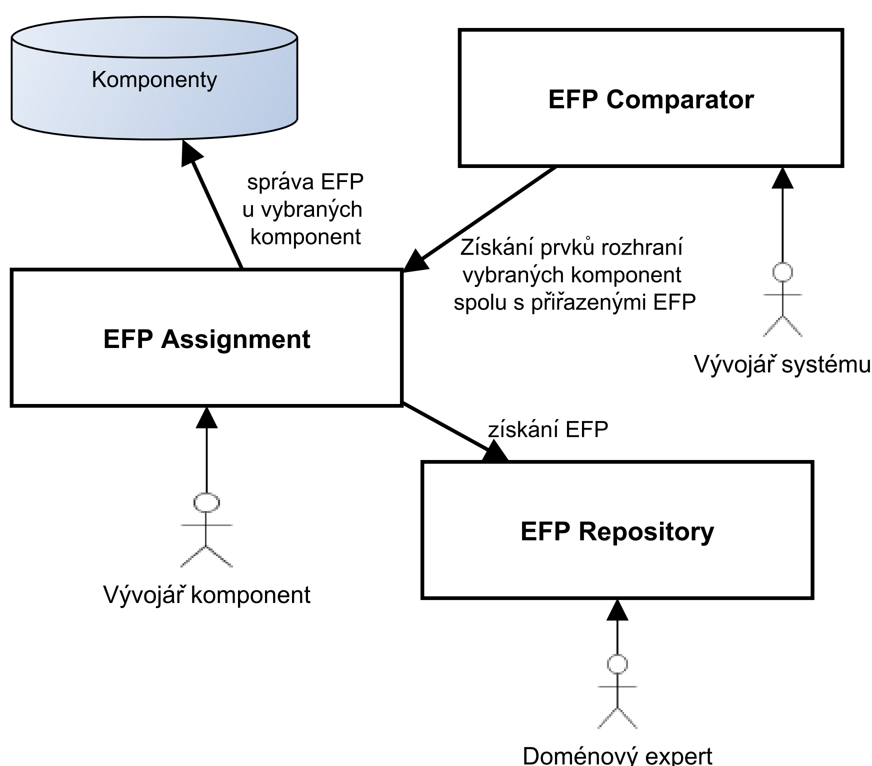
Ovlivnění spočívá v redukci položek seznamu možných Capabilities, které odpovídají filtru v Requirement. Pro využití musí být implementována a zaregistrována v frameworku služba Resolver Hook Factory s rozhraním `ResolverHookFactory`, která vrací instanci implementace rozhraní `ResolverHook`. Pro každý pokus o propojení resolver získá přes všechny registrované služby Resolver Hook Factory novou instanci třídy implementující rozhraní `ResolverHook`. Voláním následujících metod rozhraní může dojít k redukci kandidátů:

- `filterResolvable(Collection)` odstraní z kolekce bundly, které jsou kandidáty pro řešení.
- `filterMatches(BundleRequirement, Collection)` z kolekce odstraní část Capabilities. Jedná se o efektivní cestu, jak skrýt pro určitý Requirement vybrané Capabilities.
- `filterSingletonCollisions(BundleCapabilities, Collection)` umožňuje odstranit z kolekce potenciálně konfliktní singleton Capabilities.

Při implementaci služby Resolver Hook je nutné dávat velký pozor, aby nemohlo docházet k nekonzistentnímu stavu při řešení Requirements. Musí být *stabilní*, to znamená, pro stejnou množinu vstupů musí vždy vracet stejnou odpověď.

4 Stav projektu

Jak již bylo zmíněno, tato práce navazuje na projekt z roku 2011. Jeho hlavní části (moduly) jsou ukázány na obrázku 4.1. Modul *EFP Repository* má na starosti správu mimofunkčních charakteristik. Typickým uživatelem je doménový expert, tedy osoba, která má povědomí o mimofunkčních charakteristikách, které lze v dané oblasti nadefinovat a jejichž použití má nějaký význam.



Obrázek 4.1: Hlavní části projektu z roku 2011

Modul *EFP Assignment* slouží k přiřazení mimofunkčních charakteristik ke komponentám. Vývojář komponenty, který zná její chování, otevře v uživatelském rozhraní vybranou komponentu a poté popisuje prvky rozhraní komponenty dostupnými mimofunkčními charakteristikami z jádra modulu *EFP Repository*. Tento modul je předmětem této práce. V rámci ní se bylo nutné ještě seznámit s modulem *EFP Repository*, jelikož jsou z něj získávány

předdefinované mimofunkční charakteristiky pro jednotlivá přiřazení.

Posledním modulem je *EFP Comparator*, což je nástroj pro ověření kompatibility komponent mimo framework. Je určen pro vývojáře komponentové aplikace. V tuto chvíli, pokud chce vývojář sestavovat aplikaci v OSGi frameworku s použitím mimofunkčních charakteristik, musí provádět ověření v tomto nástroji. Cílem této práce je přenést tuto zodpovědnost přímo na OSGi framework.

4.1 Model mimofunkčních charakteristik

Mimofunkční charakteristiky (zkráceně EFP [1]) jsou [5] speciální druh informací popisující poskytované a požadované kvality částí softwaru. Používaný model mimofunkčních charakteristik je zachycen na obrázku 4.2.

Mimofunkční charakteristiky se vztahují k oblasti, kde mají svůj význam. Ta je reprezentována v modelu tzv. globálním registrem, který je úložištěm pro vyhovující předdefinované mimofunkční charakteristiky.

Hodnoty přiřazované k EFP se často vztahují k určitému výpočetnímu prostředí, kde mají zpravidla svůj obecně uznávaný význam.

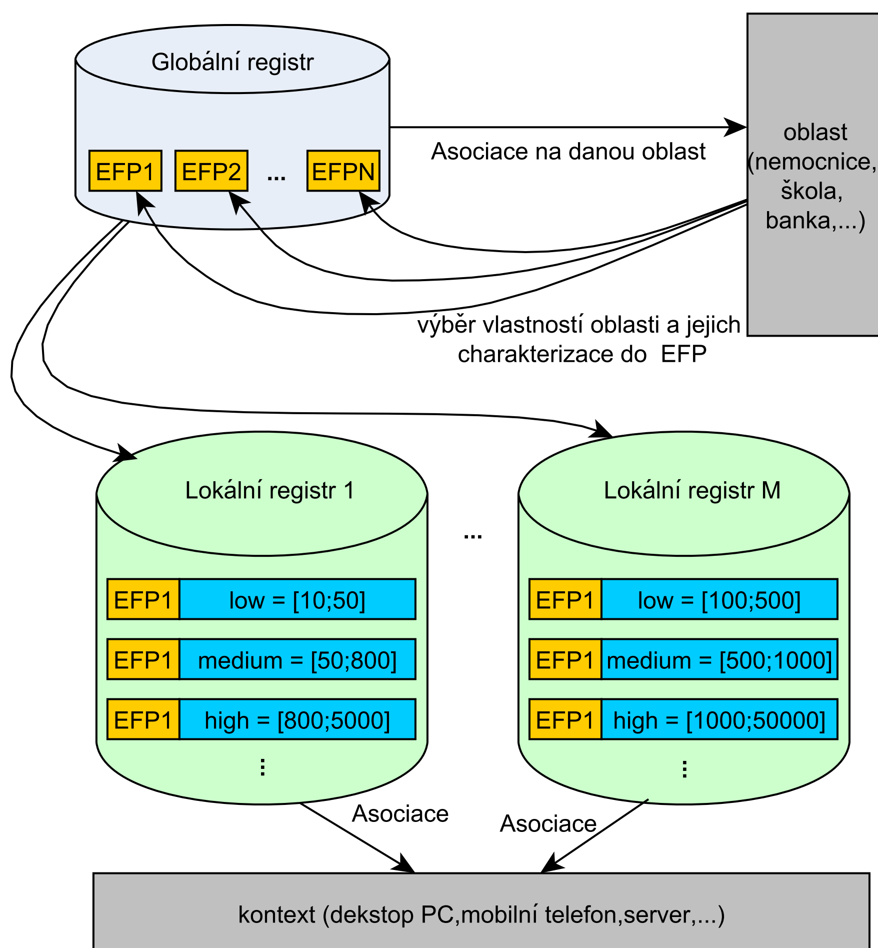
Například pro ohodnocení výkonnosti programů bude k dispozici globální registr s předdefinovanými mimofunkčními charakteristikami, které představují metriky pro měření výkonu (odezva, požadavky na paměť, místo potřebné pro instalaci, atd.). Ve výpočetním prostředí běžných desktopových PC budou mít pro mimofunkční charakteristiku odezvu následující hodnoty význam:

```
0 - 200ms - low
200 - 1000ms - average
1000 - 5000ms - high
```

Pokud výpočetním prostředím budou mobilní zařízení, která jsou oproti desktopovým PC výrazně pomalejší, budou mít předchozí hodnoty například význam:

```
0 - 200ms - ultra low
200 - 1000ms - low
1000 - 5000ms - average
```

[1] Extra-Functional Property



Obrázek 4.2: Model mimofunkčních charakteristik

Hodnoty tedy mají svůj kontext, na základě kterého můžou být děleny a charakterizovány, čímž uživatel získá snadněji představu o samotné hodnotě.

K ukládání předdefinovaných hodnot podle uvedeného příkladu byly definovány tzv. lokální registry. Každý lokální registr se vztahuje k jinému výpočetnímu prostředí a obsahuje pojmenované, pro dané prostřední charakteristické, hodnoty mimofunkčních charakteristik z globálního registru, ke kterému patří.

4.2 Jádro modulu *EFP Assignment* pro OSGi

Jádro v komponentě nalezne definici prvků rozhraní bundlu. U OSGi se jedná o soubor manifest. Po načtení prvků rozhraní komponenty lze k nim začít přiřazovat mimofunkční charakteristiky. Jádro není v práci [8] dostatečně pro účely této práce zdokumentováno, proto bude jeho podoba z původního projektu na základě provedené analýzy v následujícím textu detailněji rozebrána.

Kromě hodnoty z lokálního registru lze přiřadit s EFP následující druhy hodnot:

- Přímá hodnota
- Matematická formule
- Bez hodnoty

Přímá hodnota je určena pro případy, kdy uživatel chce přiřadit určitou hodnotu, nechce jí ale zavést do lokálního registru, nebo lokální registry vůbec nepoužívá. Datový typ hodnoty se shoduje s typem v definici EFP. Matematická formule a přiřazení bez hodnoty jsou určeny pro případy, kdy uživatel chce provést výpočet hodnoty mimofunkční charakteristiky až připojení komponenty. Podrobně je tento způsob přiřazování popsán například v [8].

V práci [8] bylo rozhodnuto, že do komponenty budou ukládány kopie použitých objektů z úložiště mimofunkčních charakteristik pro zpětnou rekonstrukci přiřazení. Vytváří se tzv. lokální úložiště EFP v komponentě, které se ukládá do speciálního XML souboru. Tento soubor je ukládán do komponenty. Kdyby byly ukládány pouze identifikátory EFP a hodnot z lokálního registru, musely by být tyto informace vždy načítány z hlavního úložiště EFP při otevírání komponenty, což může být problém, pokud uživatel bude pracovat v režimu offline a chce pouze přiřazení EFP prohlížet.

Při implementaci pro framework OSGi v [8] bylo zjištěno, že proces přiřazovaní mimofunkčních charakteristik lze rozdělit na několik částí, které mohou být přes řadu frameworků prakticky stejné. Pouze se budou lišit vnitřní implementací. Každou tuto část lze implementovat jako submodul:

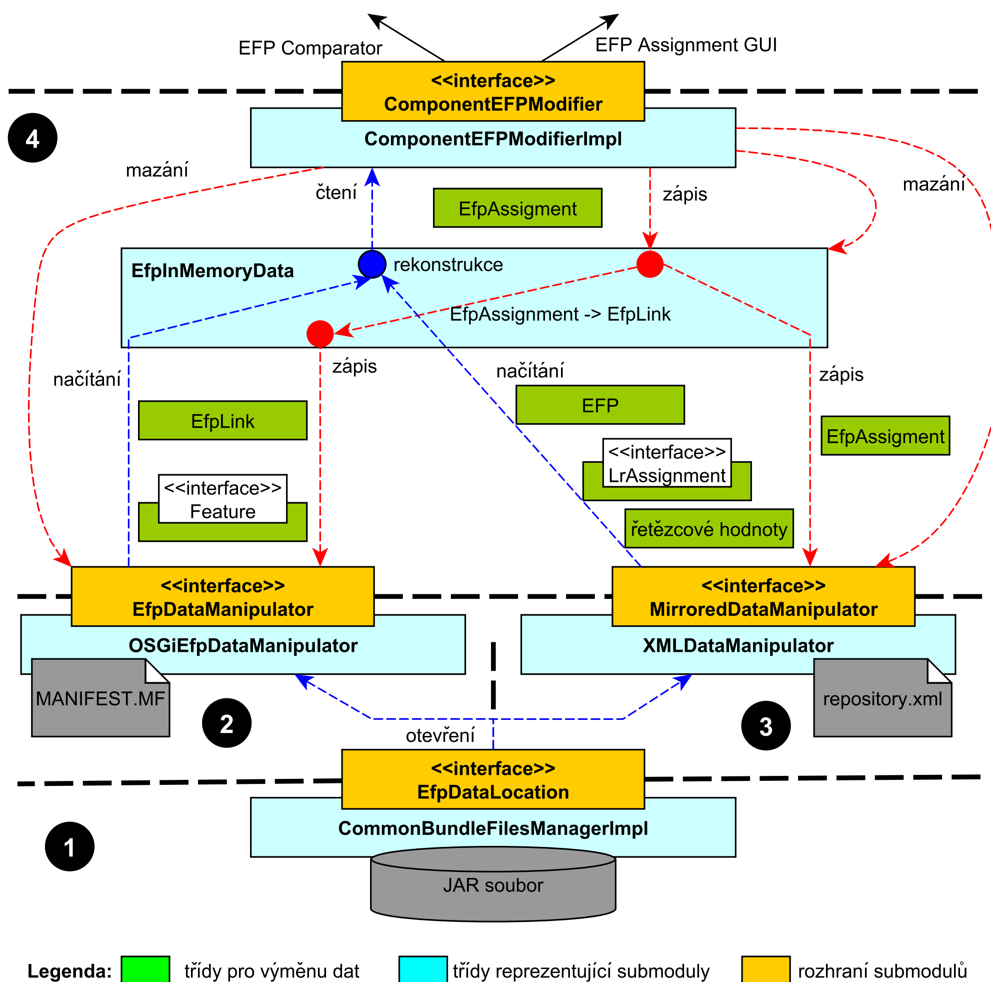
- Submodul pro správu souborů komponenty. Poskytuje soubor s popisem prvků rozhraní komponenty s přiřazenými EFP a soubor s lokálním úložištěm. (1)

- Submodul pro práci s prvky rozhraní komponenty. Aplikace z něj získává prvky rozhraní a identifikátory přiřazení EFP a posílá do něj požadavky na editaci přiřazení. (2)
- Submodul spravující soubor s lokálním úložištěm. Aplikace z něj získává části přiřazení (EFP, globální registr, lokální registr, hodnota z lokálního registru, přímé hodnoty a matematické formule ve formě řetězců) a posílá do něj požadavky na přidání nebo mazání těchto částí. (3)
- Submodul pro práci s EFP přiřazeními. Používán pro kompletní rekonstrukci přiřazení EFP na základě dílů ze submodulů (2) a (3). Zrekonstruovaná přiřazení a nástroje pro jejich editaci poté poskytuje aplikacím jako *EFP Assignment* a *EFP Comparator*. Reprezentuje komponentu v těchto aplikacích. (4)

U každého submodulu lze definovat rozhraní za účelem maximální znovupoužitelnosti. Může se totiž stát, že návrh implementace pro komponentový framework se bude lišit od již provedené implementace pro jiný framework pouze v práci se souborem s prvky rozhraní komponenty, tudíž bude nutné upravit pouze jeden submodul a ostatní submoduly použít z předchozí implementace. Stavba jádra pro OSGi s nejdůležitějšími třídami a naznačenými vazbami mezi submoduly je znázorněna na obrázku 4.3.

4.2.1 Implementace submodulu pro správu souborů komponenty

Třída `CommonBundleFilesManagerImpl` reprezentuje submodul, který má na starosti práci s bundlem. Při otevírání rozbalí obsah bundlu a přes rozhraní `EfpDataLocation` poté vrací cesty na extrahované soubory *MANIFEST.MF* s definicí rozhraní a EFP přiřazeními a *repository.xml* s lokálním úložištěm EFP. Třída je implementována podle návrhového vzoru *singleton*, aby bylo zajištěno, že k extrakci dojde pouze při prvním požadavku na některý ze souborů.



Obrázek 4.3: Struktura jádra *EFP Assignment* pro OSGi

4.2.2 Submodul pro práci s prvky rozhraní komponenty

Submodul pro práci s manifestem je reprezentován třídou `OSGiEfpDataManipulator` s rozhraním `EfpDataManipulator`. Vrací prvky rozhraní bundlu a informace o přiřazených EFP, dále obsahuje nástroje pro editaci přiřazení. Prvek rozhraní bundlu je reprezentován třídou implementující rozhraní `Feature` zajišťující poskytnutí následujících údajů:

- Název prvku a verzi

- Typ (package, bundle)
- Referenci na prvek, který je rodičem.

Přiřazení EFP je vždy reprezentováno objekty třídy `EfpLink`, která obsahuje identifikátory údajů, které jsou součástí celého přiřazení:

- ID globálního registru
- Název EFP
- Typ hodnoty
- ID lokálního registru a název pro hodnoty z lokálního registru
- ID řetězce s přímou hodnotou nebo matematickou formulí uložené v lokálním úložišti

Informace o přiřazení EFP byly v původní práci ukládány přímo do definice prvku rozhraní do speciálního atributu `efp`. Například

```
Import-Package: com.foo;efp:="462.odezva=LRASSIGN464.high"
```

znamená, že požadovaný package `com.foo` musí mít přiřazenou mimofunkční charakteristiku `odezva` z globálního registru s ID `462` s hodnotou, která odpovídá hodnotě z lokálního registru s ID `464` uložené pod jménem `high`. V případě více přiřazení EFP jsou oddělovány v atributu `efp` čárkou.

4.2.3 Implementace submodulu pro správu lokálního úložiště

Pro práci s lokálním úložištěm EFP v XML byl vytvořen submodul ve třídě `XMLDataManipulator`, která implementuje rozhraní `MirroredDataManipulator`. Podle identifikátorů z lokálního úložiště vrací požadované objekty: EFP, přiřazení z lokálního registru, přímé hodnoty a matematické formule v serializovaném tvaru. Struktura XML souboru je ukázána v příloze A.

4.2.4 Submodul pro práci s EFP přiřazeními

Nejvyšší vrstva modulu *EFP Assignment* pro práci s komponentou daného frameworku musí implementovat rozhraní `ComponentModifier`. Ve třídě `ComponentEFPModifierImpl` se nachází obecná implementace, která je teoreticky použitelná pro většinu frameworků. Pro konkrétní framework je definována až implementacemi tří submodulů popsaných výše, které přes jejich rozhraní používá.

Třída `ComponentEFPModifierImpl` využívá ještě objekt třídy `EfpInMemoryData`. Ta byla zavedena za účelem držet potřebná data komponenty v paměti, aby nemusely být při každé operaci znovu načítány (v případě OSGi parsovány z manifestu).

Při načítání rekonstruuje přiřazení EFP načítáním objektů z lokálního úložiště podle dat v objektech `EfpLink` do instancí třídy `EfpAssignment`, který slouží k reprezentaci jednoho přiřazení EFP ve vrstvách výše. Sdružuje trojici údajů každého přiřazení EFP: mimofunkční charakteristiku, prvek rozhraní a hodnotu. Implementaci obsahuje package `cz.zcu.kiv.efps.assignment.values`.

5 Navázání EFP na OSGi resolver

Po prostudování specifikace OSGi a stavu původního projektu mohla být zahájena tvorba nového mechanismu pro přiřazování mimofunkčních charakteristik k prvkům rozhraní komponenty podle požadavků v zadání. Prvním krokem bylo nalezení způsobu přiřazení mimofunkčních charakteristik, aby byly použity při ověřování kompatibility OSGi resolverem. Při návrhu se vycházelo z již provedené studie [4].

5.1 Atributy u packagů

Mimofunkční charakteristiky je nutné připojit k deklaracím packagů a požadovaných bundlů. U packagů je možné používat volitelné atributy s možností vyžadovat jejich výskyt na poskytované a požadované straně při propojování, tudíž by bylo možné pomocí nich do jisté míry vyjádřit mimofunkční charakteristiky. Nevýhodou je porovnání hodnot pouze na shodnost. Není možné specifikovat například:

- Hodnota má být větší nebo menší než nějaká hodnota.
- Hodnota má být v intervalu.
- Hodnota má být jednou z výčtu hodnot.
- Hodnota má být podmnožinou nějaké množiny.

Značně se tak zmenší možnosti, jak vyjádřit hodnotu mimofunkční charakteristiky. Toto je nepřijatelné například pro měřitelné EFP, kde hodnoty často bývají uváděny v intervalech, kvůli tomu, že prováděná měřitelná část je po opakovaném provedení dokončena téměř vždy za mírně rozdílnou dobu.

5.2 Generické Capabilities a Requirements

Lepší volbou je použít generické Capabilities a Requirements bundlu. Přiřazení EFP k prvku rozhraní komponenty může být konvertováno následujícím způsobem:

1. Jmenný prostor se složí z obecného identifikátoru pro EFP přiřazení, identifikátoru globálního registru a, pokud je hodnota z lokálního registru, identifikátoru lokálního registru. Tímto je jasně nadefinován význam a kontext atributů, které budou následovat.
2. Generické Capabilities a Requirements se přímo nevztahují k poskytovaným a požadovaným prvkům rozhraní bundlu, proto musí být součástí atributy specifikující tuto vazbu.
3. Dále budou připojeny atributy s názvem EFP a hodnotou v podobě, kterou bude moci využít OSGi framework k vyhodnocení.
4. Pro zpětnou rekonstrukci přiřazení EFP bude připojen atribut s identifikátorem hodnoty.

Menší nevýhodou je nutnost specifikovat dalšími atributy, k jakému prvku rozhraní se Capability nebo Requirement vztahuje.

V Requirement budou atributy specifikovány v LDAP filtru, který je nutné číst a sestavovat. Pro tyto účely lze napsat vlastní parsovací třídu nebo využít některou existující implementaci. Například od *Apache Software Foundation* veřejně dostupný parser nebo využít implementaci z některého OSGi frameworku.

5.3 Konverze hodnot datových typů EFP

Mimofunkční charakteristiky z *EFP Repository* využívají řadu datových typů pro hodnoty. Jejich implementaci obsahuje package `cz.zcu.kiv.efps.types.datatypes`. Jedná se o typy:

`EfpString` - řetězcové hodnoty.
`EfpNumber` - číselné hodnoty.
`EfpRatio` - číselné hodnoty v intervalu [0; 100].
`EfpBoolean` - pravdivostní hodnoty.
`EfpEnum` - výčet řetězcových hodnot.
`EfpNumberInterval` - číselné intervaly.
`EfpComplex` - složené hodnoty, jednotlivé složky jsou pojmenovány.
`EfpSet` - množiny hodnot.

Každý datový typ implementuje rozhraní `EfpValueType`, přes které je hodnota v kódu standardně používána. Tyto typy musí být konvertovány

do podoby, které rozumí OSGi resolver, u Capabilities do množiny atributů a u Requirements do LDAP filtru. Tabulky 5.1 a 5.2 zachycují návrh těchto konverzí.

Tabulka 5.1: Tabulka konverzí datových typů EFP do seznamu atributů

Datový typ	**Příklad**	**Atributy**
`EfpString`	”foo”	`value:String="foo"`
`EfpNumber`	25.6	`value:Double=25.6`
`EfpRatio`	10.0	`value:Double=10.0`
`EfpBoolean`	true	`value=true`
`EfpEnum`	A1,A2,B1	`value:List<String>="A1,A2,B1"`
`EfpNumberInterval`	[25; 500]	`value_min="25"` `value_max="500"`
`EfpComplex`	{”JMENO”: ”A”, ”HODN”: 25}	`value_JMENO:String="A"` `value_HODN:Double=25`
`EfpSet`	{”*A*”, 25}	`value_1:String="A"` `value_2:Double=25`

Tabulka 5.2: Tabulka konverzí datových typů EFP do LDAP filtru

Datový typ	**Příklad**	**LDAP filtr**
`EfpString`	”foo”	`(value=foo)`
`EfpNumber`	25.6	`(value=25.6)`
`EfpRatio`	10.0	`(value=10.0)`
`EfpBoolean`	true	`(value=true)`
`EfpEnum`	A1,A2,B1	`(& (value=A1) (value=A2)` `(value=B1))`
`EfpNumberInterval`	[25; 500]	`(& (value_min>=25)` `(value_max<=500))`
`EfpComplex`	{”JMENO”: ”A”, ”HODN”: 25}	`(& (value_JMENO=A)` `(value_HODN=25))`
`EfpSet`	{”*A*”, 25}	(& `(value_1=A) (value_2=25))`

Jako základní název atributu s hodnotou byl zvolen identifikátor `value`. V případě komplexnějších typů `EfpNumberInterval`, `EfpComplex` a `EfpSet` je doplněn ještě dodatečným identifikátorem.

U `EfpNumberInterval` se jedná o příponu `_min` pro dolní mez a příponu `_max` pro horní mez. U `EfpComplex` je doplněn o název složky, který je defi-

nován spolu s hodnotou. U `EfpSet`, který u hodnoty složky nemá název, bylo zvoleno pořadové číslo.

Datové typy `EfpComplex` a `EfpSet` mohou používat pro hodnoty všechny ostatní typy včetně jich samotných, může tak docházet k rekurzivní definici hodnoty o několika úrovních. Podobným způsobem musí pracovat konverze, kdy pro první úroveň je základním identifikátorem atributu `value`, pro každou další úroveň je tento identifikátor doplněn o příponu podle pravidel uvedených výše.

Například pro hodnotu typu `EfpComplex` se složkou A a B, kde B obsahuje hodnotu typu `EfpSet`, jehož druhá složka je typu `EfpNumberInterval`,

```
{ "A":5, "B":{10,[25,500]}}
```

bude mít konverze do seznamu podobu

```
value_A:Double=5  ,
value_B_1:Double=10 ,
value_B_2_min:Double=25 ,
value_B_2_max:Double=500
```

a konverze do LDAP filtru

```
(& (value_A=5)
   (& (value_B_1=10)
      (& (value_B_2_min>=25)(value_B_2_max<=500))
   )
) .
```

6 Implementace

Bylo rozhodnuto, že informace o přiřazení EFP k prvkům rozhraní bundlu budou přeneseny do generických Capabilities a Requirements. Důvodem byly dostatečné možnosti specifikace nutného přiřazení EFP na straně požadovaného prvku rozhraní, kdy LDAP filtry lze popsat i složitější závislosti. Nepokrytá přiřazení EFP se pokusí vyřešit implementací služby Resolver Hook.

Z pohledu práce z roku 2011 se jedná hlavně o kompletní přepracování submodulu `OSGiEfpDataManipulator` v package `cz.zcu.kiv.efps.assignment.osgi` a doplnění metod pro konverzi datových typů hodnot EFP do požadované formy pro Requirements a Capabilities.

6.1 Vedlejší úpravy

Před hlavním úkolem byla provedena řada úprav a oprav chyb. Mezi nejvýznamnější patří kompletní přepracování parseru hlaviček z manifestu. Původní implementace je zbytečně složitá, což částečně způsobuje v určitých případech problémy s rychlostí.

Další úpravy se týkaly submodulu pro práci s JAR soubory, kde byly zjištěny dva významné nedostatky:

- Násobné otevírání bundlu
- Aktualizování celého bundlu po zavření, i když k žádné změně nedošlo.

Poslední významnou změnou bylo zavedení řetězcového identifikátoru do globálního a lokálního registru. V původní práci, pokud bylo potřeba redukovat informaci o registru na jednoznačný identifikátor (např.: do manifestu), bylo k dispozici pouze jeho celočíselné ID z databáze. Z pohledu čitelnosti je mnohem lepší v těchto případech používat řetězcový identifikátor.

6.1.1 Práce s bundlem - odstranění násobného otevírání

Původně po otevření bundlu byly pouze extrahovány soubory s manifestem a lokálním úložištěm EFP. Při ukládání byl bundle znovu otevřen a byly

překopírovány soubory z původního bundlu, k nimž byly připojeny nové kopie souboru s manifestem a lokálním úložištěm. Docházelo ke zbytečnému dvojímu načtení archivovaného souboru. V novém algoritmu jsou všechny soubory načteny do vybrané složky už při prvním otevření.

Bylo také upraveno zavírání streamů. Reference na každý stream je po uzavření nastavena na NULL a na vytipovaných místech je volán z Javy *Garbage Collector*.

Při zátěžových testech, ve kterých je zpracováváno velké množství bundlů v cyklu, totiž na platformě Windows docházelo při mazání původního bundlu a nahrazení aktualizovaným bundlem náhodně ke vzniku výjimek. Podrobnou analýzou bylo zjištěno, že nedochází k dostatečně rychlému uvolnění zámků nad soubory s bundly. To je způsobeno vznikem velkého množství streamů při překopírování.

6.1.2 Práce s bundlem - zavedení indikace provedených změn

Původně byl bundle aktualizován při každém jeho uzavírání, což je při pouhém čtení bundlu zbytečné. Tento nedostatek byl odstraněn rozšířením rozhraní `EfpDataLocation` pro práci se soubory bundlu o metodu `setIsModified(boolean)`. Touto metodou se nastavuje pravdivostní proměnná, podle které jsou bundly aktualizovány nebo změny zahozeny. Tato metoda je volána na základě změn provedených v manifestu.

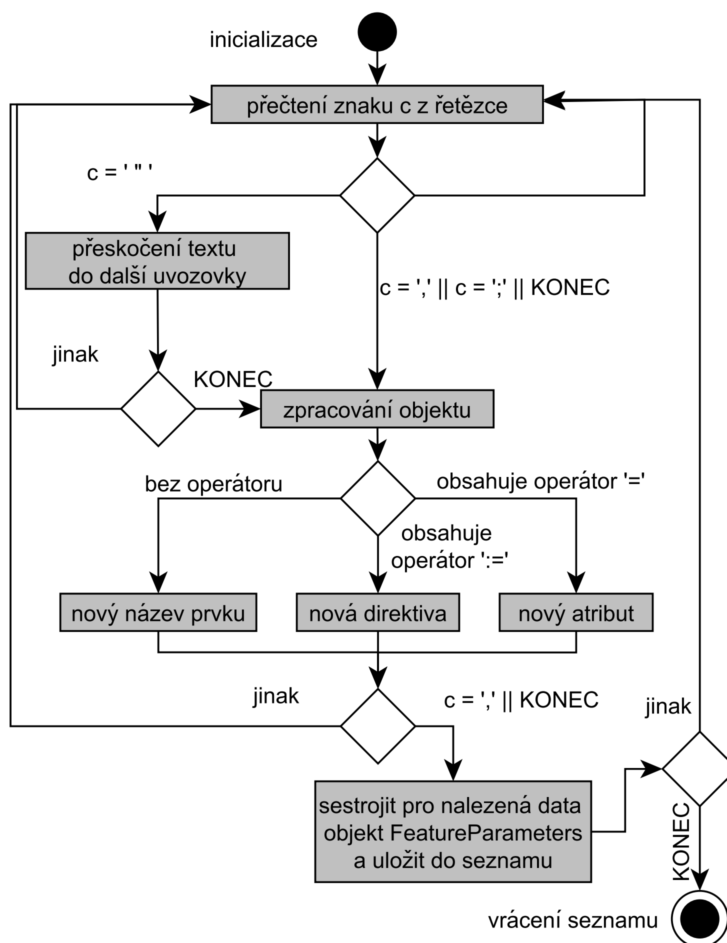
Prvním navrhovaným řešením bylo porovnávání času posledních změn souboru s manifestem po načtení a při zavírání bundlu. Cílem tohoto řešení bylo vyhnout se nutnosti rozšiřovat jakékoliv rozhraní.

Bohužel metoda `File.lastModified()` vrací čas poslední změny s různou přesností na jednotlivých souborových systémech: na EXT3 v sekundách, na NTFS v milisekundách, na FAT v 2 sekundových intervalech, atd. Pokud je načtení a zavření bundlu provedeno v menším intervalu než je přesnost udávaná souborovým systémem (typicky JUnit testy jsou prováděny v ms), mohlo by docházet k neočekávanému chování, kdy změny manifestu občas budou uloženy, občas ne.

6.1.3 Parser hlaviček z manifestu

Při hromadném zpracování několika stovek bundlů modulem *EFP Assignment* byly zpozorovány problémy s celkovou rychlostí. Jedním z důvodu byl parser hlaviček z manifestu. Při analýze bylo zjištěno, že dochází ke zbytečnému násobnému procházení řetězce hlavičky. Nejdřív je řetězec 1. průchodem rozdělen na jednotlivé definice závislostí, poté je každý řetězec se závislostí znovu procházen za účelem rozparsování na části.

Tyto dva průchody je možné spojit a celkově tak algoritmus zjednodušit. Nový parser je implementován ve třídě `HeaderParser`. Princip jeho fungování je znázorněn na obrázku 6.1.



Obrázek 6.1: Diagram aktivit pro nový parser hlaviček manifestu

Třída `FeatureParameters` reprezentuje vždy jednu závislost z manifestu v aplikaci. Obsahuje název závislosti (název package, jmenný prostor pro Capabilities a Requirements), mapu atributů a mapu direktiv. Uvedené třídy lze v projektu najít v package `cz.zcu.kiv.efps.assignment.extension.manifest`.

Po provedení zátěžových testů bylo zjištěno, že se podařilo parser urychlit o přibližně 25%.

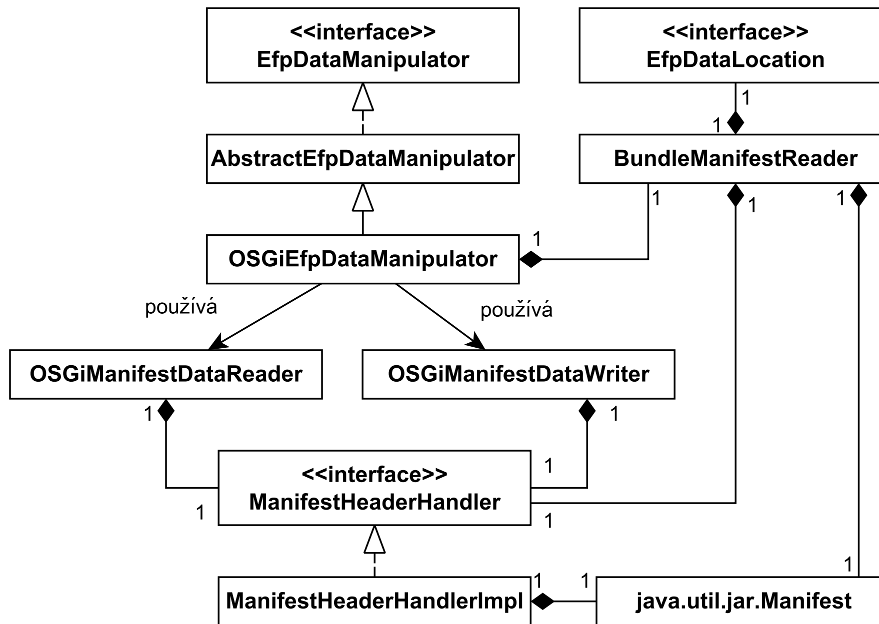
6.1.4 Řetězcový identifikátor pro globální a lokální registr

Definice globálního a lokálního registru byla rozšířena o další atribut s řetězcovou unikátní zkratkou, která by měla vždy vycházet z jejich názvu. Bylo nutné provést následující změny:

- Doplnění definice globálního registru ve třídě `cz.zcu.kiv.efps.types.gr.GR`.
- Doplnění definice lokálního registru ve třídě `cz.zcu.kiv.efps.types.lr.LR`.
- Úprava odpovídajících tříd v implementaci serveru pro modul *EFP Repository*, se kterým se pracuje prostřednictvím webových služeb.
- Úprava formulářů v GUI pro práci s globálními a lokálními registry, package `cz.zcu.kiv.efps.registry.gui.frames`.

6.2 Přepracování submodulu `OSGiEfpDataManipulator`

Nová implementace, která vychází z původní, je ukázána v diagramu na obrázku 6.2. Třída `OSGiManifestDataReader` slouží k načítání prvků rozhraní a přiřazení EFP z manifestu a třída `OSGiManiefstDataWriter` k editaci manifestu. Obě třídy pracují s manifestem přes rozhraní `ManifestHeaderHandler` třídy `ManifestHeaderHandlerImpl`, což je nadstavba knihovní třídy Javy `Manifest` pro snadnější ovládání.



Obrázek 6.2: UML diagram tříd submodulu `OSGiEfpDataManipulator`

Instanci třídy implementující rozhraní `ManifestHeaderHandler` a instanci `Manifest` spravuje objekt třídy `BundleManifestReader`. Používá návrhový vzor *singleton*, čímž je zajištěno, že manifest bude na různých místech v aplikaci reprezentován stejnou instancí. Manifest je načítán ze souboru na cestě, která je získána ze submodulu pro správu souborů bundlu přes rozhraní `EfpDataLocation`.

Použití tříd `OSGiManifestDataReader` a `OSGiManifestDataWriter` pro práci s manifestem je realizováno v metodách třídy `OSGiEfpDataManipulator`, která skrz rodičovskou třídu `AbstractEfpDataManipulator` implementuje rozhraní `EfpDataManipulator`. Účelem třídy `AbstractEfpDataManipulator` je zajistit výskyt pouze jediné instance submodulu `OSGiEfpDataManipulator` v programu přes návrhový vzor *singleton*.

Výše popsané vazby jsou popsány sekvenčním diagramem pro případy čtení z manifestu na obrázku 6.3 a pro případy zápisu do manifestu na obrázku 6.4.

Znázorněná metoda `readFeatureHeader` z rozhraní `ManifestHeaderHandler` je kromě načítání hlaviček s prvky rozhraní také používána pro načítání hlaviček s generickými Capabilities a Requirements, kam jsou nově uklá-

Obrázek 6.3: Sekvenční diagram pro čtení z manifestu

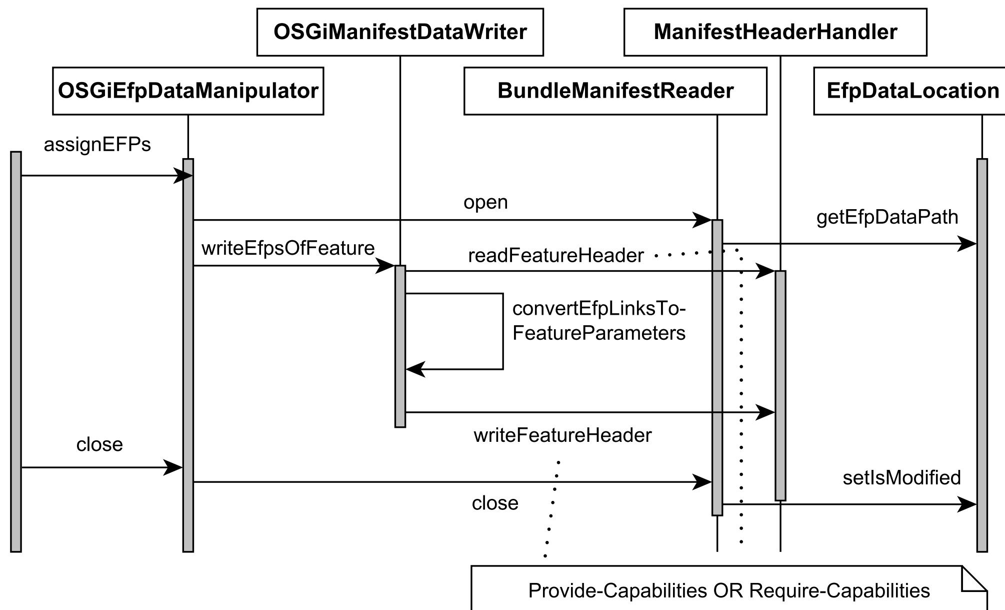
dány přiřazení EFP. Syntaxe hlaviček je identická, pouze se liší interpretace vraceného seznamu objektů `FeatureParameters` s rozparsovanou hlavičkou.

6.3 Konvertování EFP přiřazení

Pro generické Capabilities a Requirements s EFP přiřazeními byl zaveden základní jmenný prostor *cz.zcu.kiv.efps*, který je rozšířen o řetězcový identifikátor globálního registru a případně řetězcový identifikátor lokálního registru, pokud se jedná o přiřazení s hodnotou z lokálního registru.

Pro EFP přiřazení u poskytovaného prvku rozhraní je vytvořena Capability, která obsahuje atributy:

- `feature-type` obsahuje typ prvku rozhraní (`PACKAGE` pro package, `BUNDLE` pro bundle).



Obrázek 6.4: Sekvenční diagram pro zápis do manifestu

- `feature-name` název prvku rozhraní, ke kterému byla EFP přiřazena.
- `version` verze prvku rozhraní, pokud je specifikována.
- `valueID` identifikátor přiřazené hodnoty. Pro přiřazení s prázdnou hodnotou je vynecháván.
- Atributy získané konverzí hodnoty, jejichž název začíná prefixem `value`.

Například pro poskytovaný package *com.foo*, kterému bude přiřazena mimofunkční charakteristika *memory* z globálního registru s ID *Computer* a hodnotou *high=[25;500]* z lokálního registru s ID *PC*, bude Capability ukládaná do manifestu vypadat následovně:

```
cz.zcu.kiv.efps.PC; feature-type=PACKAGE;
    feature-name=com.foo; valueID=high; efp=memory;
    value_min:Double=25; value_max:Double=500
```

V případě přiřazení přímé hodnoty by atribut `valueID` obsahoval řetězec `DIRECT`, ke kterému je připojeno číslo, pod kterým je uložena hodnota v serializované formě v lokálním úložišti. Pro matematickou formuli by místo řetězce `DIRECT` byl použit řetězec `FORMULA`.

Pro EFP přiřazení u požadovaného prvku rozhraní je vytvořen Requirement, která obsahuje vždy atribut `valueID` a direktivu `filter`, která obsahuje v logické formuli v konjunktivní formě atributy:

- `feature-type` obsahuje typ prvku rozhraní.
- `feature-name` - název prvku rozhraní, ke kterému byla EFP přiřazena.
- `version` - požadovaná verze prvku rozhraní, pokud je specifikována.
- Logickou formuli konverze hodnoty s atributy, jejichž název začíná prefixem `value`.

Pro případ uvedený u příkladu poskytovaného package bude požadavek ukládaný do manifestu vypadat následovně:

```
cz.zcu.kiv.efps.PC; valueID=high; filter:=
     (& (feature-type=PACKAGE) (feature-name=com.foo)
     (efp=memory) (& (value_min>=25) (value_max<=500)))
```

6.4 Parsování LDAP filtru

Byla použita implementace parseru pro LDAP filtr z OSGi frameworku Equinox. Použitá implementace se nachází v repositáři *Eclipse-4.3* ve třídě `org.osgi.framework.FrameworkUtil`.

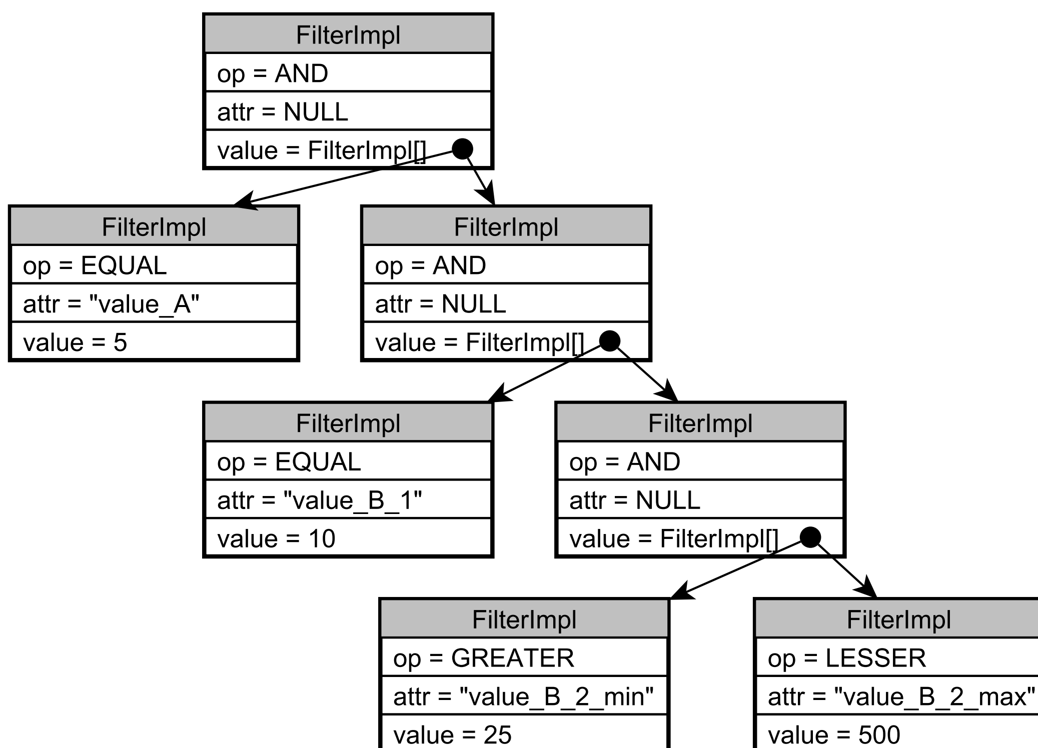
Testováním bylo ověřeno, že se skutečně jedná o vyhovující parser LDAP filtrů. Použitím této implementace je navíc zajištěno, že parser pokrývá všechny potenciální varianty konkrétních LDAP filtrů, které jsou v OSGi frameworku validní.

Získaný parser rozparsuje řetězec s LDAP filtrem do stromu, kde uzly a listy jsou reprezentovány instancemi třídy `FilterImpl`, která byla získána spolu s parserem. Pokud reprezentuje list, jsou plněny vlastnosti jméno atributu, hodnota a operátor. Pro uzel jsou plněny vlastnosti operátor a hodnota, která je použita pro uložení pole potomků. Rozlišení uzlu a listu lze provést zavoláním implementované metody `isItem()`. Například pro filtr

```
(& (value_A=5)
```

```
        (& (value_B_1=10)
           (& (value_B_2_min>=25)(value_B_2_max<=500))
        )
    )
```

bude sestrojen strom ukázaný na obrázku 6.5.



Obrázek 6.5: Výsledný strom pro rozparsovaný LDAP filtr

Obě tyto třídy byly umístěny do package `cz.zcu.kiv.efps.assignment.osgi.manifest.ldap`.

Druhou navrhovanou možnost v podobě veřejně dostupného parseru *shared-ldap* od *Apache Software Foundation* bylo nutné zavrhnout. Při testování bylo zjištěno, že ve filtru, v názvech atributů, lze používat pouze alfanumerické znaky, což odporuje požadavkům této práce a i možnostem LDAP filtrů, které jsou validní pro OSGi.

6.5 Konverze datových typů hodnot EFP

Pro obecnou reprezentaci v aplikaci každý definovaný datový typ z package `cz.zcu.kiv.efps.types.datatypes` implementuje rozhraní `EfpValueType`. Toto rozhraní bylo rozšířeno o metody:

- `convertIntoParameters(String)` - konvertuje datový typ podle tabulky 5.1 do mapy atributů s částmi hodnot. Parametrem je základní identifikátor názvu pro atributy.
- `convertIntoLDAPFilter(String)` - konvertuje datový typ podle tabulky 5.2 do řetězce s LDAP filtrem. Parametrem je základní identifikátor názvu pro atributy.

7 Ověření funkčnosti

7.1 Výkonnostní testy

Otestování rychlosti probíhalo na sériovém zpracování 200 bundlů, které obsahují nějaká přiřazení EFP. Testovací stroj obsahoval čtyřjádrový procesor Intel Core i7 2600k 3.4GHz, 8GB RAM a výkonný SSD OCZ Vertex 3 240GB. Byl měřen celkový čas následujících fází:

1. Rozbalení bundlu
2. Parsování manifestu
3. Zavření bundlu

Fáze 1 s rozbalováním bundlů trvala na testovacím stroji celkem 70s. Rozbalení archivu bundlů v externím specializovaném archivovacím programu WinRAR na stejném stroji proběhlo za podobnou dobu, tudíž doba je přijatelná.

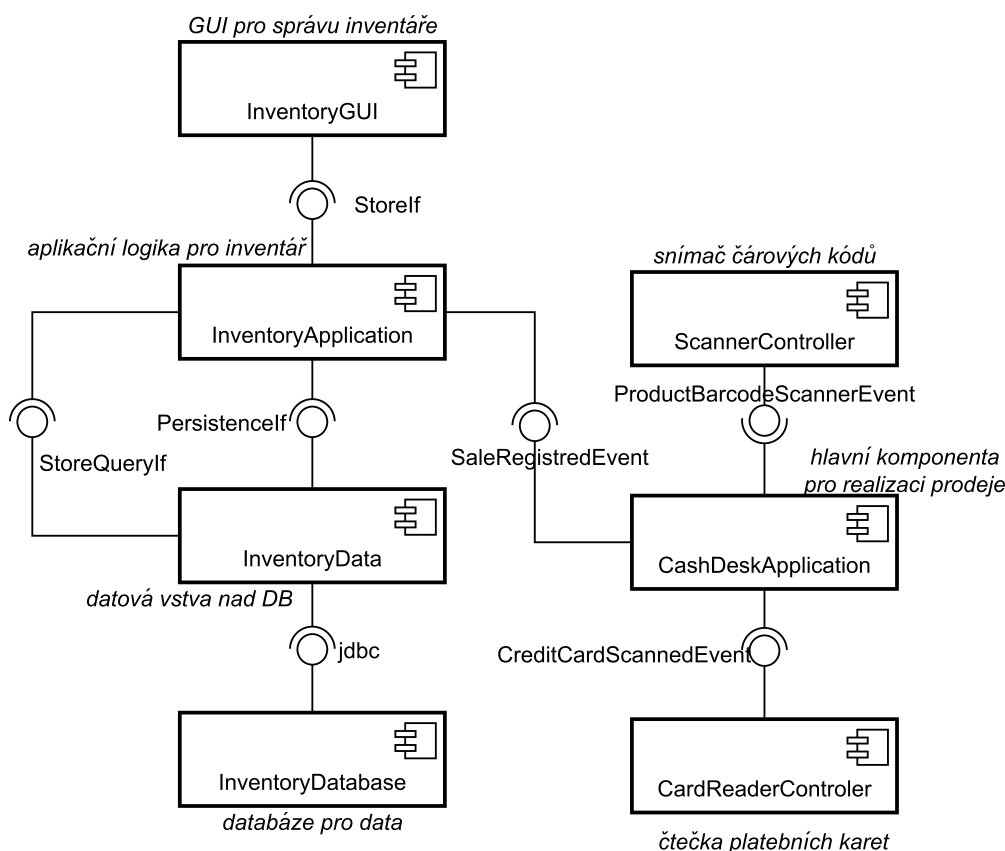
Fáze 2, kde probíhá většina parsování z hlaviček manifestu byla dokončena přibližně za 15s, což je 75ms na bundle. Dobu lze považovat za akceptovatelnou, ale prostor pro případné optimalizace by zde mohl být, protože původní způsob přiřazování EFP trval zlomek tohoto času.

Nárůst doby je způsoben hlavně parsováním LDAP filtru v Requirements. Ve chvíli, kdy mají být nalezena EFP přiřazení pro konkrétní požadovaný prvek, jsou procházeny všechny rozparsované Requirements. Zjištění, jestli Requirement specifikuje přiřazení EFP k tomuto prvku, lze provést až z atributů filtru. Filtr musí být tedy rozparsován, protože identifikátory prvku rozhraní vlastnícího přiřazení se nachází v něm.

Fáze 3, kde hlavní činností je mazání rozbalených souborů, byla dokončena za 15s, což je přibližně doba, za kterou budou manuálně smazány soubory podobného počtu na testovacím stroji. I tuto naměřenou hodnotu lze považovat za přijatelnou.

7.2 Ověření nového způsobu zápisu přiřazení do manifestu

K důležitým částem kódu jsou napsány automatické testy, které byly spolu s modifikacemi kódu také upraveny. Několik dalších automatických testů bylo doplněno. Po dokončení projektu byly provedeny testy simulující reálné použití na komponentách z návrhu obchodního systému založeného na příkladu CoCoMe[1] [2]. Tento příklad byl již použit v práci [8], z něhož ověření v této kapitole částečně vychází. Diagram s komponentovým modelem je zobrazen na obrázku 7.1.



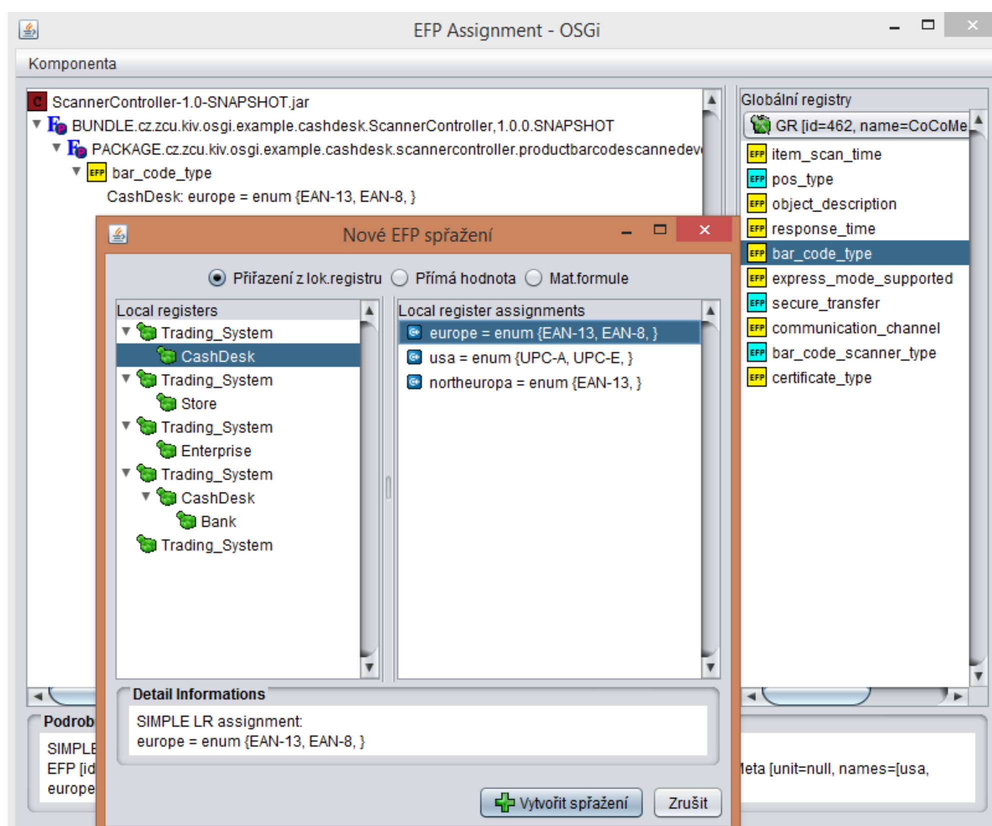
Obrázek 7.1: Diagram příkladu komponentové aplikace

[1]Common Component Example

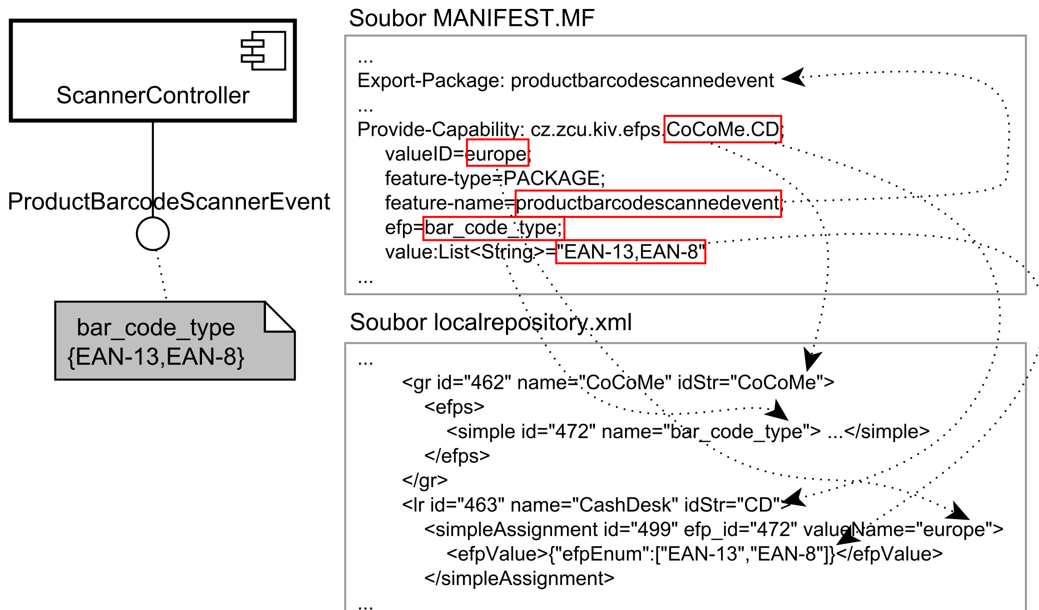
7.2.1 Přiřazení EFP s hodnotou z lokálního registru I

Komponenta `ScannerController` poskytuje prostřednictvím package `ProductBarcodeScannedEvent` službu, která reprezentuje událost oskenování čárového kódu. Skener může skenovat pouze určité kódy, což lze specifikovat mimofunkční charakteristikou. Bude se jednat například o skener pro skenování kódů v Evropě.

Komponenta `ScannerController` je otevřena v uživatelském rozhraní modulu *EFP Assignment*. Následně je z globálního registru *CoCoMe* vybrána mimofunkční charakteristika *bar_code_type* pro výčet typů čárových kódů. Z lokálního registru `CashDesk` je poté vybrána hodnota *europa*, viz obrázek 7.2. Je potvrzeno nové přiřazení. Po uložení komponenty budou soubory s manifestem a lokálním úložištěm obsahovat data, zobrazená na obrázku 7.3.



Obrázek 7.2: Náhled GUI *EFP Assignment* při tvorbě EFP přiřazení



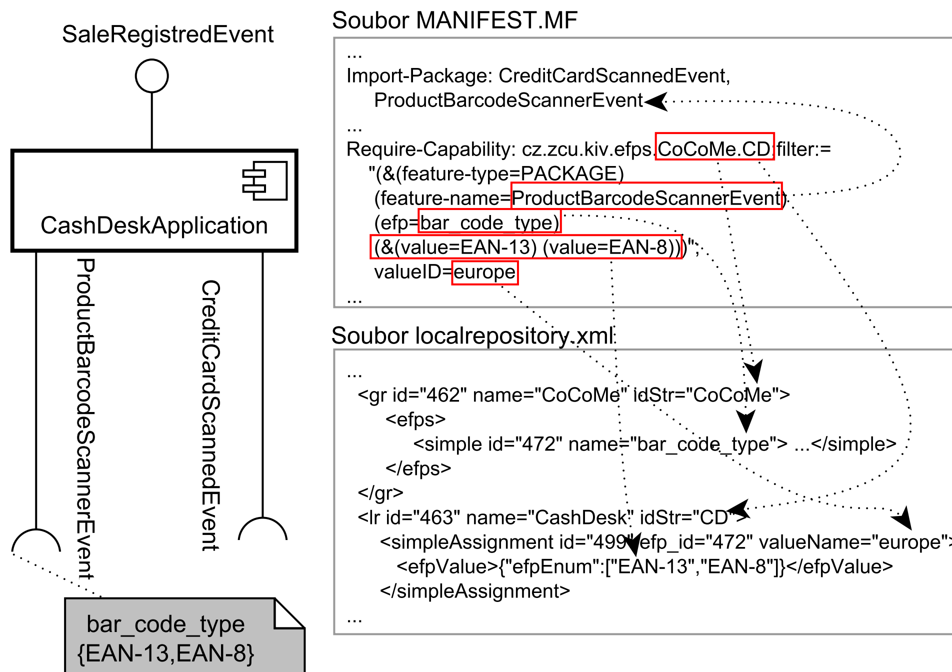
Obrázek 7.3: Stav souborů komponenty po přiřazení hodnoty z lokálního registru do definice poskytovaného package

7.2.2 Přiřazení EFP s hodnotou z lokálního registru II

Pokud bude mít v aplikaci komponenta `ScannerController` předchozí přiřazení, bude moci být provedeno úspěšné ověření kompatibility s komponentou `CashDeskApplication` pouze v případě výskytu přiřazení EFP *bar_code_type* v definici požadovaném package `ProductBarcodeScannedEvent`.

Po přiřazení stejné hodnoty budou po uložení komponenty soubory s manifestem a lokálním úložištěm obsahovat data zobrazená na obrázku 7.4.

Přiřazení se místo v hlavičce `Provide-Capabilities` nachází v hlavičce `Require-Capabilities`, což odpovídá tomu, že bylo provedeno přiřazení k požadovanému prvku rozhraní.



Obrázek 7.4: Stav souborů komponenty po přiřazení hodnoty z lokálního registru do definice požadovanému package

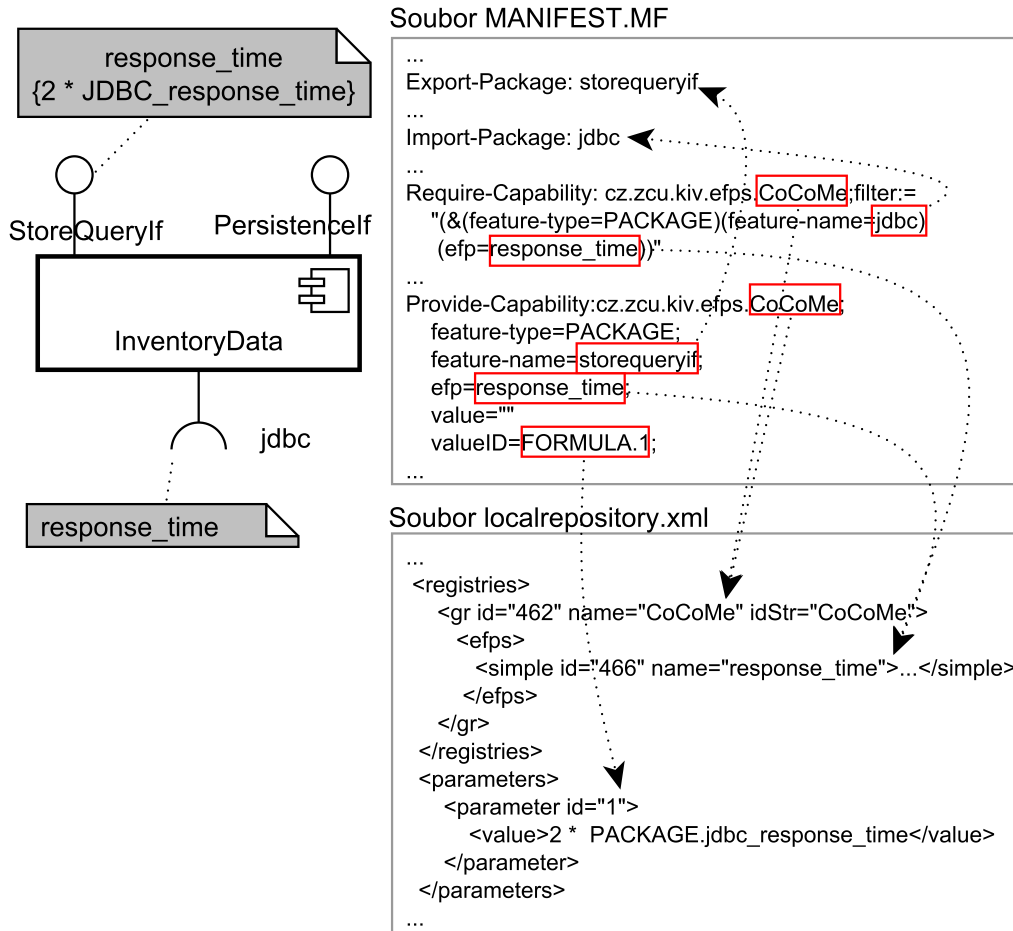
7.2.3 Přiřazení EFP s prázdnou hodnotou a matematickou formulí

K ukázce bude využita komponenta `InventoryData`. Prostřednictvím package `StoreQueryIf` poskytuje službu pro dotazy do databáze. Tato služba reaguje s určitou odezvou. Odezva z velké části vychází z odezvy požadované služby s konektorem pro databázi prostřednictvím package `JDBC`. Například vývojář komponenty zjistí, že díky vnitřní logice komponenty odezva na `StoreQueryIf` se prodlouží dvakrát oproti odezvě na `JDBC`.

V globální registru *CoCoMe* se nachází pro specifikaci odezvy mimofunkční charakteristika *response_time*. Po otevření komponenty je na `JDBC` přiřazena EFP *response_time* bez hodnoty. Poté je EFP přiřazena na `StoreQueryIf`. Jako hodnota je zvolena matematická formule. Buď ručně, nebo přes editor formulí je nastaven vzorec *2 * JDBC_response_time*.

Výsledek popsaného přiřazování je ukázán na obrázku 7.5. Podobně lze

vyjádřit odezvu u druhé poskytované služby přes package `PersistenceIf`.



Obrázek 7.5: Stav souborů komponenty po přiřazení matematické formule

7.3 Ověření funkčnosti v OSGi frameworku

Pro testování byl vybrán OSGi framework Equinox ve verzi 3.8.2. Tento framework byl používán již během návrhu řešení. Při instalaci komponent bylo vždy kontrolováno, jestli nebudou odmítnuty například pro porušený soubor bundlu nebo chyby v manifestu. Bylo realizováno několik scénářů zavedení komponent z obchodního systému CoCoMe.

Nejdříve bylo zkontrolováno zavedení komponent bez přiřazených EFP, jestli všechny komponenty mohou být vyřešeny a připraveny pro spuštění (viz kontrolní výpis na obrázku 7.6).

```
osgi> lb
START LEVEL 6
   ID|State      |Level|Name
    0|Active     |    0|OSGi System Bundle (3.8.2.v20130124-134944)
    1|Active     |    4|Console plug-in (1.0.0.v20120522-1841)
    2|Active     |    4|Apache Felix Gogo Command (0.8.0.v201108120515)
    3|Active     |    4|Apache Felix Gogo Runtime (0.8.0.v201108120515)
    4|Active     |    4|Apache Felix Gogo Shell (0.8.0.v201110170705)
  122|Resolved   |    1|CardReaderController OSGi Bundle (1.0.0.SNAPSHOT)
  123|Resolved   |    1|CashDeskApplication OSGi Bundle (1.0.0.SNAPSHOT)
  124|Resolved   |    1|InventoryApplication OSGi Bundle (1.0.0.SNAPSHOT)
  126|Resolved   |    1|InventoryData OSGi Bundle (1.0.0.SNAPSHOT)
  127|Resolved   |    1|InventoryDatabase OSGi Bundle (1.0.0.SNAPSHOT)
  128|Active     |    1|InventoryGui OSGi Bundle (1.0.0.SNAPSHOT)
  129|Resolved   |    1|ScannerController OSGi Bundle (1.0.0.SNAPSHOT)
```

Obrázek 7.6: Výpis po vyřešení komponent bez EFP do frameworku

7.3.1 Kompatibilní komponenty

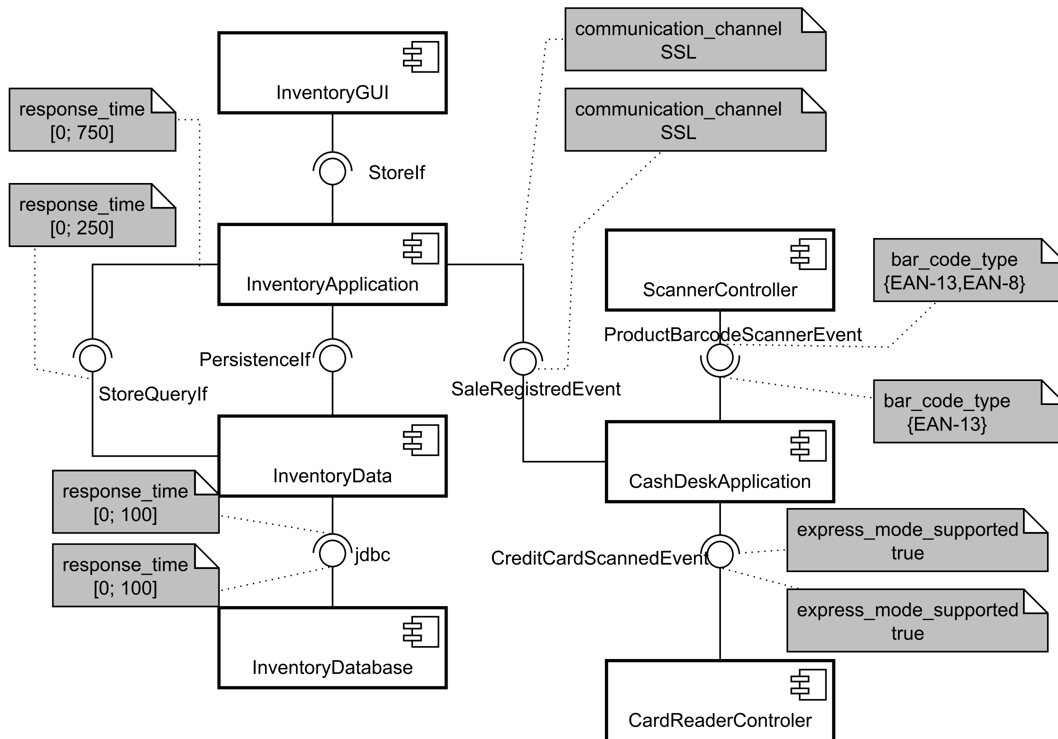
Ke komponentám z aplikace byly přiřazeny mimofunkční charakteristiky podle obrázku 7.7 tak, aby komponenty byly vyhodnoceny jako kompatibilní.

Po zavedení do frameworku by měly být všechny komponenty vyřešeny a připraveny ke spuštění, což dokládání obrázek 7.8 s výpisem stavu komponent. Vyřešení komponent bylo zahájeno pokusem o spuštění komponenty `CashDeskApplication`.

7.3.2 Nekompatibilní komponenty

Cílem dalšího příkladu bylo přiřadit mimofunkční charakteristiky ke komponentám tak, aby byly vyhodnoceny jako nekompatibilní. Diagram komponent s přiřazeními je ukázán na obrázku 7.9. Červeně jsou označeny přiřazení, které zabrání úspěšnému vyřešení komponent `CashDeskApplication` a `InventoryData`.

Po stejném pokusu o nastartování komponenty `CashDeskApplication` dopadne řešení bundlů například podle obrázku 7.10. Jsou vyřešeny pouze



Obrázek 7.7: Diagram kompatibilních komponent s EFP přiřazení

komponenty `InventoryDatabase`, `ScannerController` a `CardReaderControler`. Ostatní nelze vyřešit, dokud nebudou vyřešeny komponenty `CashDeskApplication` a `InventoryData`.

Při pokusu o vyřešení komponenty `CashDeskApplication` je vyhozena výjimka `BundleException`:

```
The bundle CashDeskApplication_1.0.0.SNAPSHOT [174] could not be
resolved.
Reason: Missing Constraint: Require-Capabilities: cz.zcu.kiv
.efps.CoCoMe.CD; filter="(&(feature-type=PACKAGE)
(feature-name=productbarcodescannedevent)
(efp=bar_code_type) (&(value=UPC-A)(value=UPC-E)))"
```

Při pokusu o vyřešení komponenty `InventoryData` je vyhozena výjimka `BundleException`:

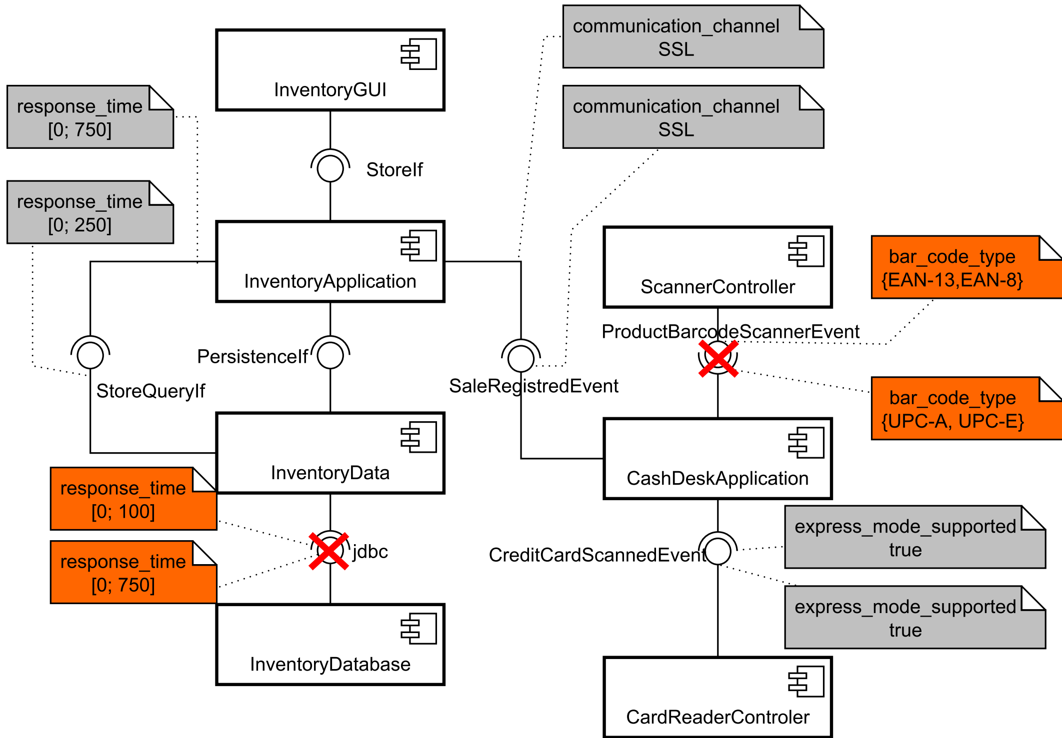
```
The bundle InventoryData_1.0.0.SNAPSHOT [176] could not be
resolved.
```

```
  ID|State      |Level|Name
   0|Active     |    0|OSGi System Bundle (3.8.2.v20130124-134944)
   1|Active     |    4|Console plug-in (1.0.0.v20120522-1841)
   2|Active     |    4|Apache Felix Gogo Command (0.8.0.v201108120515)
   3|Active     |    4|Apache Felix Gogo Runtime (0.8.0.v201108120515)
   4|Active     |    4|Apache Felix Gogo Shell (0.8.0.v201110170705)
 159|Resolved   |    1|CardReaderController OSGi Bundle (1.0.0.SNAPSHOT)
 160|Active     |    1|CashDeskApplication OSGi Bundle (1.0.0.SNAPSHOT)
 161|Resolved   |    1|InventoryApplication OSGi Bundle (1.0.0.SNAPSHOT)
 162|Resolved   |    1|InventoryData OSGi Bundle (1.0.0.SNAPSHOT)
 163|Resolved   |    1|InventoryDatabase OSGi Bundle (1.0.0.SNAPSHOT)
 164|Resolved   |    1|InventoryGui OSGi Bundle (1.0.0.SNAPSHOT)
 165|Resolved   |    1|ScannerController OSGi Bundle (1.0.0.SNAPSHOT)
```

Obrázek 7.8: Výpis s přehledem stavu komponent (vyřešeny)

```
Reason: Missing Constraint: Require-Capabilities: cz.zcu.kiv
.efps.CoCoMe.CD; filter="(&(feature-type=PACKAGE)
(feature-name=jdbc) (efp=response_time)
(&(value_min>=0.0)(value_max<=100.0)))"
```

Chyby popisují dvě nenalezené Capibility, které reprezentují EFP přiřazení zvýrazněná v diagramu komponent.



Obrázek 7.9: Diagram nekompatibilních komponent s EFP přiřazení

```
 ID|State      |Level|Name
  0|Active     |    0|OSGi System Bundle (3.8.2.v20130124-134944)
  1|Active     |    4|Console plug-in (1.0.0.v20120522-1841)
  2|Active     |    4|Apache Felix Gogo Command (0.8.0.v201108120515)
  3|Active     |    4|Apache Felix Gogo Runtime (0.8.0.v201108120515)
  4|Active     |    4|Apache Felix Gogo Shell (0.8.0.v201110170705)
174|Installed  |    1|CashDeskApplication OSGi Bundle (1.0.0.SNAPSHOT)
175|Installed  |    1|InventoryApplication OSGi Bundle (1.0.0.SNAPSHOT)
176|Installed  |    1|InventoryData OSGi Bundle (1.0.0.SNAPSHOT)
177|Resolved   |    1|InventoryDatabase OSGi Bundle (1.0.0.SNAPSHOT)
178|Installed  |    1|InventoryGui OSGi Bundle (1.0.0.SNAPSHOT)
179|Resolved   |    1|ScannerController OSGi Bundle (1.0.0.SNAPSHOT)
180|Resolved   |    1|CardReaderController OSGi Bundle (1.0.0.SNAPSHOT)
```

Obrázek 7.10: Výpis s přehledem stavu komponent (nevyřešeny)

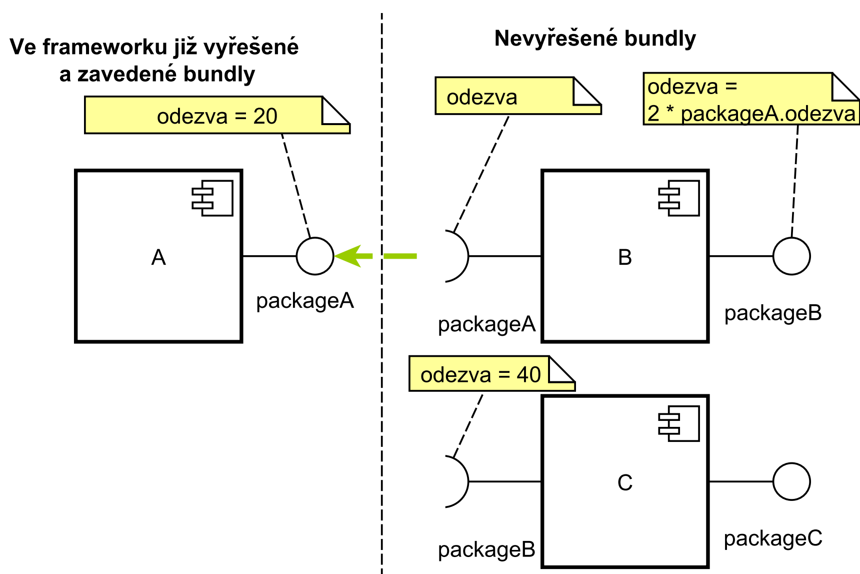
8 Návrhy na vylepšení

V práci se nachází dva významnější nedostatky. Cílem této kapitoly bylo tyto nedostatky popsat a navrhnout jejich řešení.

8.1 Výpočet matematických formulí

Navrhnutý nástroj poskytuje řadu způsobů, jak přiřadit mimofunkční charakteristiky bundlům, u kterých se pak o vyhodnocení postará OSGi resolver při jejich zavádění. Bohužel se nepodařilo převést všechny možnosti přiřazení. Jedná se o mechanismus matematických formulí a prázdných hodnot. Přiřazení EFP s matematickou formulí uživatel může provést, ale OSGi resolver neprovede v Capability její výpočet. Na následujícím modelovém příkladu je problém detailněji popsán.

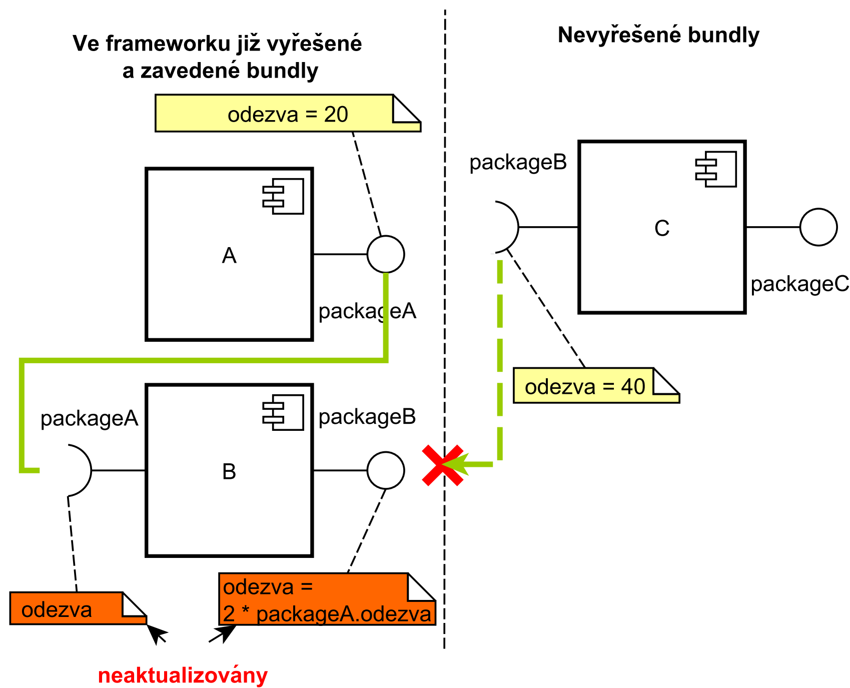
Obrázek 8.1 ukazuje, že ve frameworku se nachází bundle A, který je vyřešen a poskytuje package `packageA`. K němu je přes Capability přiřazena



Obrázek 8.1: Problém matematické formule před připojením bundlu B

EFP *odezva* s hodnotou 20ms. Mezi nevyřešenými bundly se nachází bundle

B, který u požadovaného package `packageA` vyžaduje Requirement s definovanou EFP *odezva* bez hodnoty. Hodnota má být získána tedy až z připojené Capability. V tuto chvíli nic nebrání propojení, čímž bundle B přejde do stavu vyřešený a začne poskytovat package `packageB` s Capability obsahující přiřazenou EFP *odezva* s matematickou formulí, která ale není vyhodnocena (viz obrázek 8.2). Nevyřešená matematická formule způsobí, že nebude možné



Obrázek 8.2: Problém matematické formule po připojení bundlu B

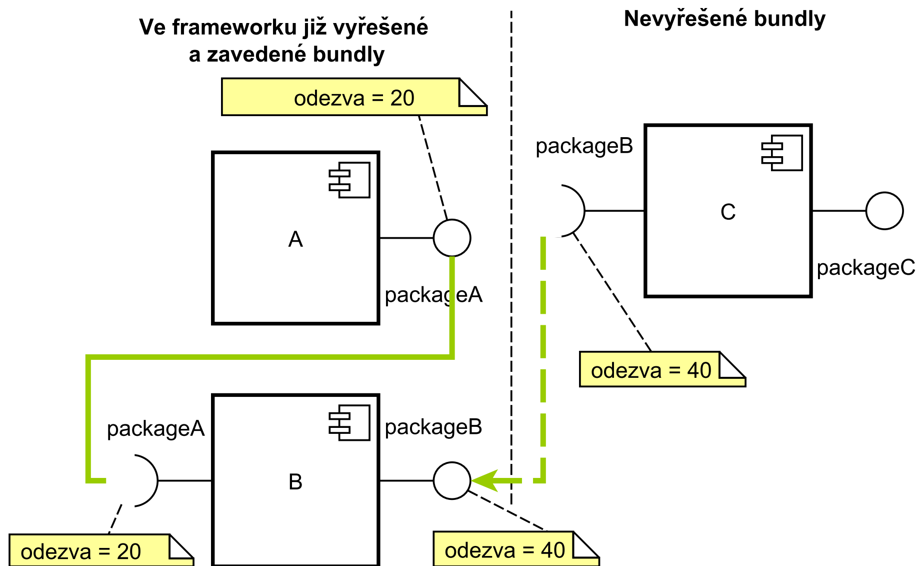
vyřešit bundle C, který požaduje package `packageB` s Capability obsahující přiřazení EFP *odezva* s hodnotou 40ms.

Záměr využít na matematické formule službu Resolver Hook se nepodařilo již provést. Původní navrhované řešení by navíc nebylo možné realizovat. Služba Resolver Hook je založena na tom, že pro Requirement vybírá z kandidátů s již známou hodnotou, která vyhovuje filtru v Requirement. To znamená, že hodnota matematické formule v Capability z některé vyřešené komponenty musí být vypočtena již před touto fází. Jinak se tato Capability mezi kandidáty pro Requirement další řešené komponenty nedostane.

Návrh počítal s tím, že Capability s matematickou formulí bude obsahovat obecnou hodnotu, která bude vyhovovat každé hodnotě v libovolném

Requirement se stejnou EFP. Ve službě ResolverHook by hodnota matematické formule uvažované Capability byla přes *EFP Comparator* dopočtena. Tuto obecnou hodnotu se nepodařilo prostředky OSGi specifikovat.

Alternativou je počítat matematickou formuli v Capability při zavádění komponenty, která ji vlastní, a ne až v komponentě, která z dané Capability potřebuje vypočtenou hodnotu. Ve fázi, kdy je nalezen kandidát pro Requirement, který obsahuje EFP přiřazení, by byl v metodě `filterMatches()` inicializován *EFP Comparator*, který by provedl aktualizaci hodnot v ještě neposkytovaných Capabilities. Byly by aktualizovány Capabilities s matematickými formulemi a prázdnými hodnotami, které používají hodnoty z Requirements. Tímto bude zajištěno, že po vyřešení bundlu budou všechny jeho Capabilities obsahovat správnou hodnotu. Po implementaci nastíněného řešení by tedy po vyřešení bundlu B měl stav vypadat jako na obrázku 8.3, kde nic nebrání vyřešení bundlu C.

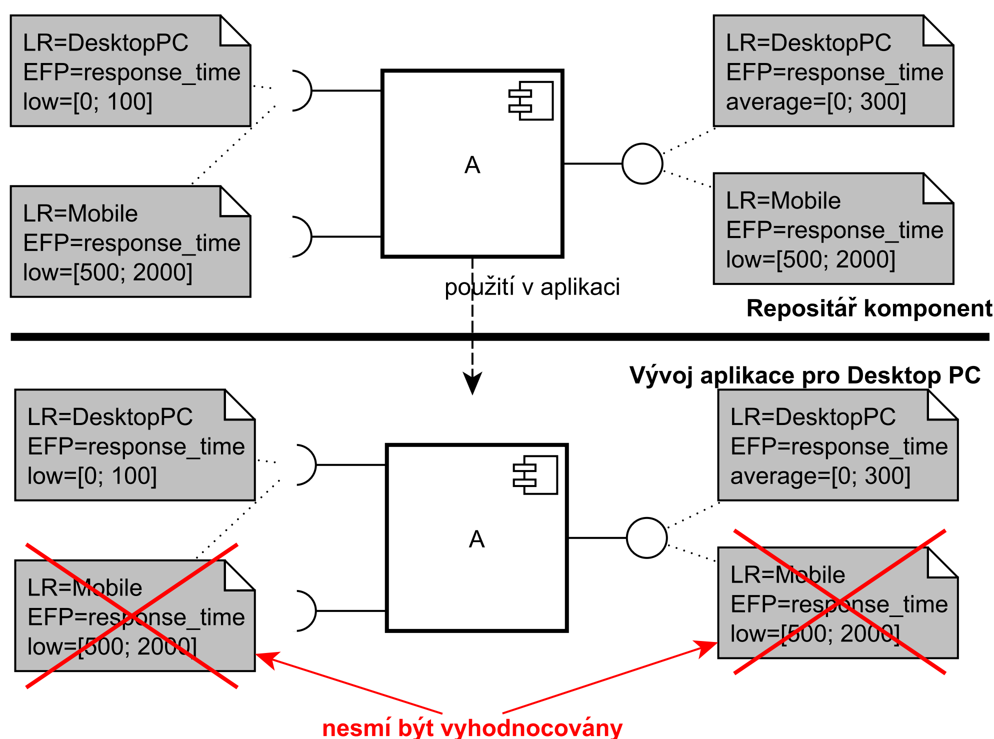


Obrázek 8.3: Problém matematické formule po připojení bundlu B po implementaci řešení

8.2 Problém výběru lokálních registrů

V CBSE je časté použití komponenty v různých výpočetních prostředích. Očekává se, že komponenta bude popisována mimofunkčními charakteristikami pro každé prostředí zvlášť. Ve výsledku by teda měla mít pro jednotlivé použité mimofunkční charakteristiky několik přiřazení, každé s hodnotou z jiného lokálního registru.

Výpočetní prostředí je charakterizováno až použitím komponenty při tvorbě aplikace. Od té chvíle by měly být při ověřování kompatibility brány v potaz pouze přiřazení z lokálního registru, který charakterizuje toto výpočetní prostředí (viz grafické znázornění na obrázku 8.4). Použití pouze validních EFP přiřazení je schopen realizovat *EFP Comparator*. Při ověřování kompatibility komponent v OSGi na základě této práce jsou vyhodnocovány všechny přiřazení bez ohledu na výpočetní prostředí.



Obrázek 8.4: Problém výběru lokálních registrů

Tento nedostatek by bylo možné odstranit použitím direktivy `effective` v definici generických Capabilities a Requirements. OSGi resolver vyhodno-

cuje pouze Capabilities a Requirements, které mají v této direktivě hodnotu `resolve`. V případě jiné hodnoty jsou ignorovány. Nastavením této direktivy by bylo možné filtrovat Capabilities a Requirements s EFP přiřazeními na ty, které jsou validní pro dané nasazení komponenty.

Bohužel nebyla nalezena možnost, jak tuto operaci provádět přímo ve frameworku při zavádění bundlu ve fázi, kdy je načítána specifikace rozhraní z jeho manifestu. Služba Resolver Hook v tomto případě nepomůže, protože je schopná filtrovat pouze vyhovující Capabilities. Musel by existovat takový Hook, který umožní načtené Capabilities a Requirements z rozhraní bundlu vyfiltrovat, než začne řešit jednotlivé Requirements.

Určitým řešením by mohla být implementace malého externího nástroje, ve kterém je komponenta před zavedením do frameworku otevřena. Nástroj by v manifestu prošel všechny Capabilities a Requirements a získal z názvů jmenných prostorů použité lokální registry. Uživatel by pak vybral lokální registry, které mají být použity při vyhodnocování. Při ukládání by u všech Capabilities a Requirements s EFP přiřazeními z nevybraných lokálních registrů byla nastavena direktiva `effective` například na hodnotu `no`. Bundle poté bude aktualizován a může být zaveden do frameworku.

Nevýhodou je, že je opět za účelem kvalitnějšího ověřování komponent vyžadováno použití externího nástroje. Jedním z cílů této práce bylo právě tento požadavek odstranit.

8.3 Úprava definice EFP přiřazení v manifestu

Tato možná úprava má za cíl pouze ukázat na základě ověřování výsledného mechanismu, jak by bylo možné ještě zpřehlednit definice Capabilities a Requirements s EFP přiřazeními. Každá definice je poměrně obsáhlá. Atributy s `feature-type` a `feature-name` by bylo možné sloučit do jednoho atributu, který by v názvu obsahoval typ prvku rozhraní a v hodnotě jméno prvku rozhraní.

Z hlediska OSGi je bezvýznamný atribut `valueID`. Tento atribut je relevantní pouze pro modul *EFP Assignment* při zpětné rekonstrukci, tudíž tuto informaci není povinné ukládat do manifestu. K tomu lze použít například soubor s lokálním úložištěm EFP, jež by byl rozšířen o novou speciální sekci

pro hodnoty atributů `valueID`. Pro každé přiřazení EFP by se zde nacházel jeden záznam, který by obsahoval:

- Identifikační údaje přiřazení: ID globálního registru, název EFP, identifikátor prvku rozhraní komponenty vlastnící přiřazení a ID lokálního registru.
- Údaje o přiřazené hodnotě: typ přiřazení (z lokálního registru, přímá hodnota, matematická formule) a ID hodnoty (jméno hodnoty v lokálním registru, ID přímé hodnoty, ID matematické formule). Případně by mohl obsahovat rovnou serializované přímé hodnoty a matematické formule.

Z manifestu budou pak získány Capabilities a Requirements s jednotlivými přiřazeními. Na základě nich bude nalezen odpovídající záznam v nové sekci v lokálním úložišti. Ze záznamu se přečtou informace o přiřazení a poté bude provedena rekonstrukce přiřazení standardní cestou.

Například přiřazení EFP *bar_code_type* z globálního registru *CoCoMe* do požadovaného package `ProductBarcodeScannedEvent` s hodnotou *europa* z lokálního registru *CashDesk* s ID *CD* má v manifestu v hlavičce `Require-Capability` definici

```
cz.zcu.kiv.efps.CD; valueID=europa;
     filter:=(& (feature-type=PACKAGE)
     (feature-name=ProductBarcodeScannedEvent)
     (efp=bar_code_type)(& (value=EAN-13)(value=EAN-8))) .
```

Po realizaci navrhovaných změn by definice v manifestu měla podobu

```
cz.zcu.kiv.efps.CD; filter:=
     (& (PACKAGE=ProductBarcodeScannedEvent)
     (efp=bar_code_type)(& (value=EAN-13)(value=EAN-8)))
```

a v lokálním úložišti EFP by se v nové sekci nacházel záznam, který by měl přibližně tuto podobu:

```
<value-ref feature='ProductBarcodeScannedEvent'
    gr='CoCoMe' efp='bar_code_type' lr='CD' >
     <valueID>europa</valueID>
</value-ref>
```

Z pohledu projektu by to znamenalo rozšířit modul pro správu lokálního úložiště EFP a upravit mechanismus rekonstrukce přiřazení EFP.

9 Závěr

Cílem této práce bylo upravit existující nástroj z roku 2011 pro přiřazování mimofunkčních charakteristik ke komponentám OSGi tak, aby pokročilé ověřování kompatibility komponent s mimofunkčními charakteristikami byl schopen provést i OSGi resolver. Před touto prací mohl vyhodnocení přiřazených mimofunkčních charakteristik provádět pouze *EFP Comparator*. Schopnosti tohoto modulu byly zachovány, pouze část jeho schopností vyhodnocování byla přenesena na OSGi resolver.

Hlavní nedostatky nového nástroje pro přiřazení EFP s pokusem o návrh řešení byly popsány v předchozí kapitole. Vzhledem k tomu, že OSGi se neustále vyvíjí a že prostředky pro vytvoření nového nástroje pro přiřazování EFP se objevily až při dokončování práce z roku 2011, není vyloučeno, že se v OSGi specifikaci objeví nový mechanismus, který umožní nedostatky tohoto řešení odstranit.

S tím, že se nepodaří pokrýt všechny možnosti přiřazování EFP, se od zahájení projektu částečně počítalo. Hlavní požadavky, které byly specifikovány před zahájením projektu, byly provedeny, tudíž cíl práce lze považovat za splněný.

Zkratky

CBSE Component-Based Software Engineering

JAR Java Archive

OSGi Open Service Gateway Iniciative

EFP Extra-Functional Property

GR Global Registry

LR Local Registry

LDAP Lightweight Directory Access Protocol

XML eXtensible Markup Language

UML Unified Modeling Language

CoCoMe Common Component Example

Literatura

[1] Bachmann, F.; Bass, L.; Buhman, C.; aj.: Volume II: Technical concepts of component-based software engineering. Technická zpráva, Carnegie Mellon University, Software Engineering Institute, Květen 2000.

[2] Herold, S.; Klus, H.; Welsch, Y.; aj.: *The Common Component Modeling Example*. Springer-Verlag Berlin, Heidelberg, 2008, ISBN 978-3-540-85288-9, 16-53 s.

[3] Howes, T.: A String Representation of LDAP Search Filters. Standards track, University of Michigan, ITD Research Systems, RFC 1960.

[4] Jezek, K.; Brada, P.; Holy, L.: Enhancing OSGi with Explicit, Vendor Independent Extra-functional Properites. In *50th International Conference on Objects, Models, Components, Patterns*, Springer-Verlag Berlin, Heidelberg, 2012, 7304, 108-123.

[5] Jezek, K.; Brada, P.; Stepan, P.: Towards Context Independent Extra-functional Properties Descriptor for Components. In *Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010), Electronic Notes in Theoretical Computer Science (ENTCS)*, ročník 264, Říjen 2010, ISSN 1571-0661, s. 55–71.

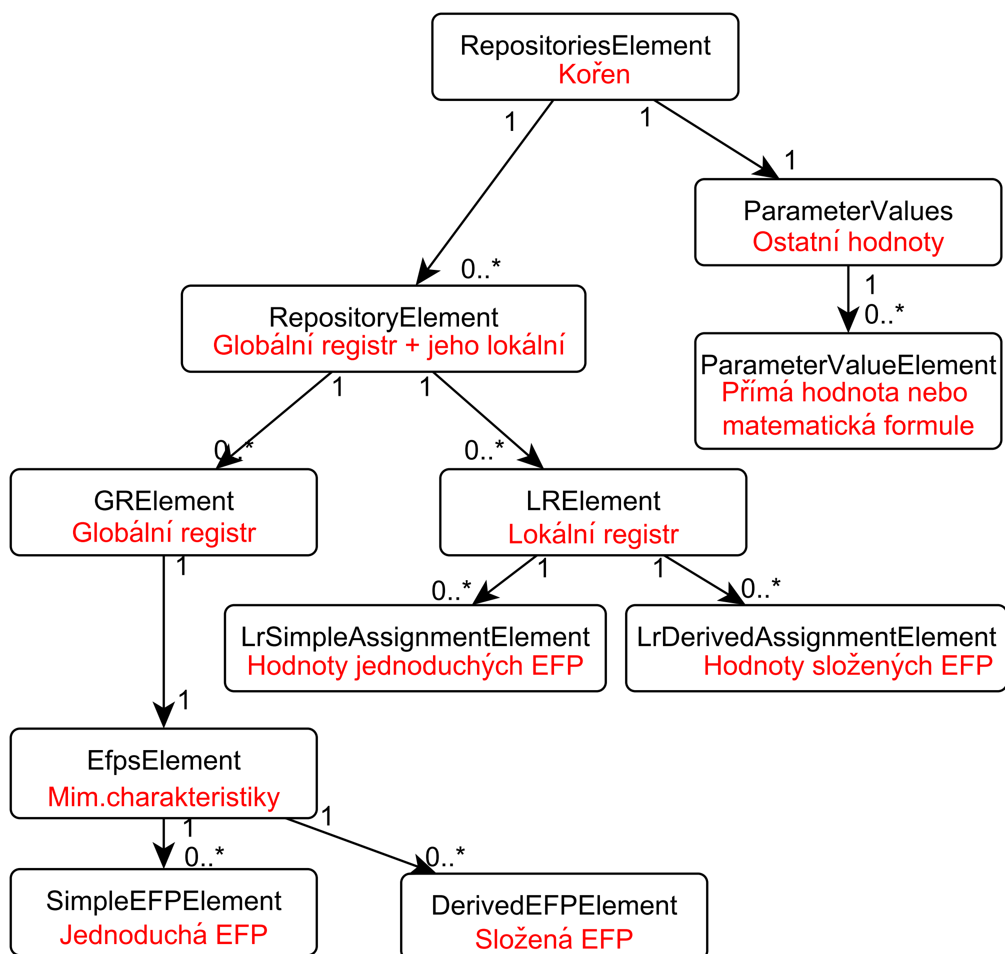
[6] The OSGi Alliance: *OSGi Service Platform Core Specification*. Duben, release 4, verze 4.3.
URL `<http://www.osgi.org>`

[7] Szyperski, C.: *Component Software - Beyond Object-Oriented Programming*. ACM Press, 2002, ISBN 0-201-74572-0, 60-75 s.

[8] Šváb, J.: *Obecná reprezentace mimofunkčních charakteristik na komponentách*. Bakalářská práce, Západočeská univerzita v Plzni, 2011.

Přílohy

A Struktura XML s lokálním úložištěm



Obrázek A.1: Schéma XML pro lokální úložiště

B Překlad a spuštění

Pro sestavení je nutné mít nainstalovaný a nakonfigurovaný program Maven[1]. Kromě něho je pro spuštění sestavené aplikace vyžadována Java ve verzi 1.7 a novější[2].

V kořenovém adresáři v projektu, který lze získat standardně z SVN serveru na adrese

`<http://subversion.assembla.com/svn/efps/modules>`,

stačí poté zadat příkaz `mvn install`, čímž dojde k překladu a sestavení projektu. Spustitelný JAR s nástrojem pro přiřazování mimofunkčních charakteristik ke komponentám se bude nacházet v kořenovém adresáři projektu na cestě:

`./efpAssignmentGUI/target/efpAssignmentGUI-1.5-SNAPSHOT.jar`

[1] Dostupný na `<http://maven.apache.org/download.html>`
[2] Dostupná na `<http://www.java.com/en/download/index.jsp>`

C Obsah přiloženého CD

Obsahuje tento dokument v elektronické podobě a tři složky. Složka `dokumentace_zdroj` obsahuje zdrojový text tohoto dokumentu psaného v LaTeX.

Složka `efps` obsahuje poslední revizi celého projektu. Projekt se skládá z 11 částí:

```
efpAssignment
efpAssignmentCoSi
efpAssignmentGUI
efpAssignmentOSGi
efpAssignmentStub
efpComparator
efpPortal
efpRegistryClient
efpRegistryGUI
efpRegistryServer
efpSpring
efpTypes
```

I když v této práci byly upravovány pouze některé, pro překlad a chod celé aplikace jsou potřeba všechny. Největší úpravy byly provedeny v `efpAssignment` a `efpAssignmentOSGi`. Dále byly prováděny drobnější úpravy v `efpRegistryServer`, `efpRegistryClient` a `efpTypes`.

Ve složce `priklady` se nachází modelová aplikace, která byla použita k ověření funkčnosti.