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

Bakalářská práce

Obecná reprezentace mimofunkčních charakteristik na komponentách

Plzeň 2011 Jan Šváb

Prohlášení

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

V Plzni dne 10. května 2011

Jan Šváb

Poděkování

Děkuji Ing. Kamilovi Ježkovi za vedení mé bakalářské práce a za podnětné návrhy, které ji obohatily a umožnily její úspěšné dokončení.

Jan Šváb

Abstract

Application of Extra-functional Properties to Component Models

With development of component-oriented programming and using third-party components for creating new software began to appear requirements on the extension specification of the interface components with extra-functional properties. They can allow more efficient composition of components.

This work aims to design and implement tool, which will be assign extra-functional properties to features of components. The criteria for its design was a structure, which allows easy extensibility to support components another types in the future.

In the first part is described the component-oriented programming and component models CoSi and OSGi. The second part deals with the implementation of the application. In the end is demonstrated its functionality on a case-study.

Obsah

1 Úvod **1**

2 Komponentové programování **2**
2.1 Komponenta 2
2.2 Komponentové modely a frameworky 2

3 OSGi platforma **3**
3.1 OSGi Bundle 3
3.2 Popis komponenty - soubor MANIFEST.MF 4
3.2.1 Druhy parametrů 4
3.2.2 Parametry verze a rozsah verzí 4
3.2.3 Bundle-SymbolicName a Bundle-Version 5
3.2.4 Import-Package Header 5
3.2.5 Export-Package Header 5
3.2.6 Require-Bundle 6

4 CoSi platforma **7**
4.1 CoSi Bundle 7
4.2 Specifikace bundlu - soubor MANIFEST.MF 8
4.2.1 Bundle-Name a Bundle-Version 8
4.2.2 Provide-Services a Provide-Interfaces 8
4.2.3 Require-Services a Require-Interfaces 9
4.2.4 Provide-Attributes 9
4.2.5 Require-Attributes 10
4.2.6 Ostatní 10

5 Model mimofunkčních charakteristik **11**
5.1 Mimofunkční charakteristiky 11
5.2 Globální registr 11
5.3 Lokální registr 12

6 Popis aplikace pro přiřazení EFP **13**
6.1 Její začlenění 13

7 Implementace **16**
7.1 Deklarace mimofunkčních charakteristik 16
7.2 Druhy přiřazovaných hodnot k EFP 18
7.3 Reprezentace prvků komponent 18
7.4 Rozhraní pro přístup ke komponentě 19
7.4.1 Rozšíření aplikace o podporu komponent 20
7.5 Submodul XMLDataManipulator 20
7.6 ComponentEFPModifierImpl 21
7.7 ManifestHeaderHandlerImpl 22
7.7.1 CommonBundleFilesManagerImpl 23
7.8 Implementace pro CoSi framework 24
7.9 Implementace pro OSGi framework 25
7.10 Uživatelské rozhraní 25
7.10.1 Rozšíření o podporu komponent dalších modelů 25
7.10.2 Formuláře a dialogy 26
7.11 Spuštění 27

8 Ověření funkčnosti **29**
8.1 Testování přiřazení EFP k CoSi komponentám 29
8.1.1 Příklad přiřazení EFP s hodnotou z lokálního registru . 29
8.1.2 Příklad přiřazení EFP s prázdnou hodnotou a mat. formulí 30
8.1.3 Příklad přiřazení složené EFP 31
8.1.4 Příklad na přiřazení EFP se složitější mat. formulí . . . 32
8.2 Testování mazání a editace přiřazených EFP 33

9 Závěr **35**

Přílohy **37**

A Uživatelská příručka **38**
A.1 Výběř pracovního typu komponent 38
A.2 Otevření a zavření komponenty 38
A.3 Přiřazení a editace charakteristiky u prvku komponenty 39

B Struktura XML s lokálním repositářem **41**

1 Úvod

Jedním ze způsobů, jak vytvářet rozsáhlejší aplikace, je komponentově orientovaný návrh. Místo toho, aby firma vyvíjela novou aplikaci tak říkajíc od nuly, využije moduly (komponenty), které byly již vytvořeny například při práci na předchozích projektech. Pokud komponenta s požadovanou funkčností neexistuje, je napsána nebo případně získána od třetí strany. Tímto přístupem může být dosaženo podstatně rychlejšího a snadnějšího vývoje softwaru.

I tento způsob návrhu aplikací má několik nevýhod. Komponenta například vyžaduje, aby implementace potřebné služby z jiné komponenty splňovala určité podmínky (např.: nároky na paměť). Dále může existovat více kandidátů na hledanou komponentu s požadovanou funkčností. Mezi těmito kandidáty je tedy nutné nějakým způsobem vybrat toho nejvhodnějšího, nebo naopak všechny vyloučit a označit je jako nekompatibilní, pokud nesplňují některou z podmínek. Tyto podmínky je nutné specifikovat.

Jednou z možností řešení je ke každé službě komponenty přidat dodatečné informace tzv. mimofunkční charakteristiky, které tyto služby jednoznačně charakterizují.

Cílem tohoto projektu je vytvořit aplikaci, která umožní specifickým částem komponenty v závilosti na jejím typu přiřadit mimofunkční charakteristiky. Poté je vhodně uloží do komponenty, aby nebyla znemožněna práce s nimi v jejich komponentovém frameworku.

Podporovány budou komponenty z komponentových modelů CoSi a OSGi s možností rozšíření o další typy.

2 Komponentové programování

V posledních letech získává tento styl na důležitosti. Důvodem pro jeho vznik byly stále se zvyšující nároky na tvorbu softwaru. Z počátku mohlo komponentou být prakticky cokoliv (např.: třída nebo celá aplikace), což vedlo k problémům se znovupoužitelností. Proto se později začaly objevovat snahy o jejich standartizaci, jejichž výsledkem jsou různé komponentové modely.

2.1 Komponenta

C. Szyperski [7] ji definuje jako jednotku skládající se ze smluvně stanovených rozhraních a pouze explicitními kontextovými závislostmi. Softwarová komponenta může být vyvíjena nezávisle a je subjektem skládání softwaru třetími stranami.

V závilosti na úhlu pohledu existuje více definic, tato práce se ale bude držet Szyperskiho popisu.

Kromě specifikace poskytovaných rozhraní je nutné u komponent stanovit, co prostředí musí poskytnout, aby mohla fungovat. Ve výsledku se specifikace rozhraní každé komponenty skládá z:

- poskytované (provided) části - Služby, které komponenta poskytuje.
- vyžadované (required) části - Závilosti na okolí, bez kterých nebude komponenta fungovat.

2.2 Komponentové modely a frameworky

Model [7] specifikuje pravidla skládání komponent a komponentových platforem, které definují pravidla vývoje, instalace a aktivace komponent.

Komponentový framework je implementací komponentového modelu. Svojí funkcí se podobá [1] operačním systémům, pouze místo procesů řídí běh komponent. Framework spravuje zdroje sdílené komponentami a umožňuje jejich vzájemnou komunikaci. Po instalaci komponenty jí nabízí zdroje, které jiné instalované komponenty poskytují, a zároveň její poskytované zdroje nabízí ostatním komponentám.

3 OSGi platforma

OSGi[1] platforma [6] je specifikace dynamického modulárního systému. Umožňuje instalaci a odebírání modulů za běhu bez nutnosti restartu aplikace, definuje životní cyklus modulu a nabízí infrastrukturu pro spolupráci modulů skrze služby.

Jádro platformy OSGi tvoří OSGi framework, který vytváří běhové prostředí umožňující nasazení a provoz komponent nazývaných bundle. Po úspěšném spuštění bundlu je ostatním instalovaným bundlům ve frameworku poskytována jeho funkcionalita. V současnosti jsou nejpoužívanějšími OSGi frameworky Equinix[2] a Felix[3].

3.1 OSGi Bundle

OSGi [6] definuje jednotku modularizace, kterou nazývá bundle. Bundle je softwarová komponenta, která obsahuje javovské třídy a ostatní zdroje, které dohromady poskytují nějakou funcionalitu uživatelům.

Bundle je vyvíjen jako JAR[4] soubor. Jedná se o standardní komprimovaný soubor v ZIP formátu, který obsahuje:

- Soubory nutné k poskytnutí určité funkčnosti, např.: soubory s Java třídami, pomocné soubory, obrázky, jiné JAR soubory.
- Textový soubor *manifest*, který popisuje obsah JAR souboru a poskytuje informace o bundlu.
- Dále může obsahovat dokumentaci o bundlu, např.: zdrojové kódy.

Manifest je velice důležitý pro framework, který ho vyžaduje pro správnou instalaci komponenty, i pro aplikaci, jež je cílem této práce. Z hlediska frameworku definuje:

Aktivátor komponenty – specifikuje třídu v bundlu, jež implementuje příslušné rozhraní pro spouštění či zastavování komponenty.

[1] Open Services Gateway Initiaitve
[2] http://www.eclipse.org/equinox/
[3] http://felix.apache.org/site/index.html
[4] Java Archive

Zdroje - rozhraním poskytované nebo vyžadované prvky komponenty. Těmito zdroji mohou být v OSGi pouze celé Java balíky.

3.2 Popis komponenty - soubor MANIFEST.MF

Soubor `MANIFEST.MF`, který obsahuje specifikaci bundlu, je uložen ve složce `META-INF` v kořeni bundlu. *Manifest* se skládá z množiny hlaviček, každá hlavička obsahuje určitou hodnotu. Jejich syntaxe je následující:

$jmeno_hlavicky : hodnota$

V následující části budou pouze popsány ty hlavičky, které se týkají této práce. Kromě nich *manifest* obsahuje další hlavičky, které jsou důležité pro správnou instalaci bundlu.

3.2.1 Druhy parametrů

OSGi definuje dva druhy parametrů, které lze přiřadit k prvkům rozhraní komponenty – direktivy a atributy. Atributy jsou využívany převážně k uložení identifikačních údajů (např.: verze, zdrojový bundle) a directivy specifikují, jak má být s prvky zacházeno při vyhodnocování OSGi frameworkem. Tyto dva druhy parametrů jsou rozlišeny symbolem pro přiřazení:

$jmeno_atributu = hodnota$
$jmeno_directivy := hodnota$

3.2.2 Parametry verze a rozsah verzí

Verze prvku se skládá ze sekvence tří číslic (od nejdůležitejšího k nejméně) a čtvrtou volitelnou částí je text (tzv. kvalifikátor). Všechny části jsou odděleny tečkami. Nesmí obsahovat žádné mezery. Defaultní hodnota verze je `0.0.0`. Verze se shoduje s jinou verzí, pokud vzájemně odpovídají jednotlivé číslice a kvalifikátor.

Rozsah verzí je určen pro specifikaci intervalu možných verzí u požadovaných zdrojů. Pokud obsahuje pouze jedinou verzi, pak musí být interpretován jako interval verzí ve tvaru $[verze, \infty)$ a defaulní hodnota je 0, která je ma-

pována na $[0.0.0, \infty)$. Příklad intervalu verzí:

$$[1.2.3, 4.1.3)$$

3.2.3 Bundle-SymbolicName a Bundle-Version

Hlavička `Bundle-SymbolicName` obsahuje název bundlu a `Bundle-Version` obsahuje jeho verzi. Spolu tvoří jednoznačný identifikátor komponenty.

3.2.4 Import-Package Header

Deklaruje importované(vyžadované) balíky pro tento bundle. Syntaxi definuje [6] v této hlavičce takto:

```
Import-Package ::= import ( ',' import )*
import ::= package-names ( ';' parameter )*
package-names ::= package-name ( ';' package-name );
```

Syntaxe umožňuje definovat parametry pro skupinu balíků, pokud jsou tyto parametry stejné. Mezi možné parametry patří direktiva:

- `resolution` - Nastavuje, zda balík musí být vyhodnocen (hodnota `mandatory`) nebo je vyhodnocení volitelné (`optional`).

A atributy:

- `version` - Obsahuje rozsah verzí k výběru exportovaného balíku.
- `bundle-symbolic-name` - Symbolické jméno bundlu, který exportuje vyžadovaný balík.
- `bundle-version` - Rozsah verzí, v němž musí být verze vyžadovaného bundlu.

Příklad:

```
Import-Package: com.acme.foo;com.acme.bar;version="[1.23,1.24]";
resolution:=mandatory
```

3.2.5 Export-Package Header

Deklaruje exportované (poskytované) balíky z bundlu. Syntaxe je stejná jako v hlavičce `Import-Package`, pouze se liší parametry:

```
Export-Package ::= export ( ',' export )*
export ::= package-names ( ';' parameter )*
package-names ::= package-name ( ';' package-name );
```

V tomto projektu byl použit pouze atribut:

- `version` - Verze poskytovaného balíku, defaultní hodnota je 0.0.0.

Příklad:

```
Export-Package: com.acme.foo;com.acme.bar;version=1.23
```

3.2.6 Require-Bundle

Obsahuje bundly, které obsahují další vyžadované balíky. Syntaxi této hlavičky definuje [6] takto:

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

Definována může být následující directiva:

- `resolution` - Pokud je hodnota `mandatory`, pak musí být tento bundle existovat pro správné vyhodnocení.

A atribut:

- `bundle-version` - Rozsah verzí, v nichž se musí verze vyžadovaného bundlu nacházet.

4 CoSi platforma

CoSi[1] [2] je výzkumný framework založený na Javě vyvíjený na Katedře informatiky a výpočetní techniky ZČU. Je odvozen od OSGi, od něhož přebírá některé vlastnosti, jiné rozšiřuje nebo vypouští.

Největším rozdílem mezi těmito systémy je v tom [2], co bundly exportují a importují. Jinak řečeno, jak sdílejí zdroje. V OSGi probíhá sdílení na úrovni balíků a v CoSi na úrovni tříd. Výhodou řešení OSGi je například situace, kdy je z balíku exportováno několik desítek tříd. Na druhou stranu výhodou CoSi je lepší správa sdílených zdrojů.

OSGi narozdíl od CoSi [6] umožňuje definovat zabezpečení bundlů pro jejich ověřování nebo například dynamicky načítat v případě nutnosti importované balíky.

4.1 CoSi Bundle

Jednotkou modularizace je stejně jako v OSGi bundle, který má podobu JAR souboru. Informace o poskytovaných a vyžadovaných zdrojích jsou uloženy v *manifestu. Manifest* obsahuje popis všech složek, které komponenta používá ke komunikaci s vnějším světem - s kontejnerem či s jinými komponentami. Pro popis rozhraní komponenty jsou definovány prvky: služby, typy, zprávy a atributy.

Služby jsou jednou z možností, jak může bundle komunikovat s okolím a poskytovat nějakou funkcionalitu. Službu představuje v Javě objekt rozhraní.

Bundle může poskytovat vnitřní třídy (typy) nebo používat třídy jiných bundlů. Tato situace může nastat při práci s nějakou komplexní službou, u které metody vrací objekty speciálních typů. Typy jsou též součástí služby a bundly, které tuto službu používají, k nim musí mít přístup.

Jiným typem komunikace mezi bundly je systém zpráv. Zpráva je Java objekt odděděný od abstraktní třídy `Message`, kterou poskytuje kontejner a kterou používá standardní služba frameworku `MessageService`, jež je určena

[1] Components Simplified

pro vytvoření systému zpráv.

Bundle může nastavit systémovou konstantu (atribut), kterou poté ostatní bundly mohou číst a používat.

4.2 Specifikace bundlu - soubor MANIFEST.MF

Soubor `MANIFEST.MF` stejně jako u OSGi komponent obsahuje specifikaci bundlu a je uložen ve stejné složce. Formát hlaviček a syntaxe jejích hodnot je též stejný. Liší se pouze v tom, že používá jiné hlavičky a pouze jeden druh parametrů - atributy. Globální atributy verze a rozsah verzí mají stejnou syntaxi jako v OSGi v kapitole 3.2.2.

V následující části budou popsány ty hlavičky, které se týkají této práce. Kromě nich podobně jako v OSGi *manifest* obsahuje další hlavičky, které jsou důležité pro správnou instalaci bundlu do CoSi frameworku.

4.2.1 Bundle-Name a Bundle-Version

Bundle je jednoznačně identifikován textovým identifikátorem, který je uložen v hlavičce `Bundle-Name`, a verzí, která je uložena v hlavičce `Bundle-Version`.

Příklad:

```
Bundle-Name : com.acme.foo
Bundle-Version : 22.3.58.build-345678
```

4.2.2 Provide-Services a Provide-Interfaces

Tyto hlavičky obsahují poskytované služby. Hlavička `Provide-Interfaces` byla definována ve starší verzi CoSi, jinak je její syntaxe stejná. Zde je používána z důvodu zpětné kompatibility. Syntaxi definuje [2]:

```
Provide-Services ::= export ( ',' export )*
export ::= interfaces ( ';' parameter )*
interfaces ::= interface ( ',' interface )*
```

```
interface ::= unique-name
```

Stejně jako v OSGi lze parametry přiřadit více zdrojům najednou. Mohou být definovány parametry:

- `name` - Jméno poskytované služby. Tento atribut je používán, pokud více služeb implementuje stejné rozhraní v systému, pak jsou rozlišeny pomocí jména.
- `version` - Verze služby.

Příklad:

```
Provide-Services:com.acme.Foo;com.acme.Bar;
versionrange="[1.23,1.24]"
```

4.2.3 Require-Services a Require-Interfaces

Obsahují vyžadované služby. Syntaxe je obdobná jako u poskytovaných, pouze se liší v atributech, které lze využít:

- `name` - Jméno vyžadované služby, pokud bundle poskytuje více služeb se stejným rozhraním.
- `versionrange` - Rozsah verzí, který specifikuje jaké verze musí poskytována služba z jiného bundlu být.

4.2.4 Provide-Attributes

Tato hlavička obsahuje poskytované atributy. Syntaxe hodnoty hlavičky je následující [2]:

```
Provide-Attributes ::= export ( ',' export )*
export ::= attributes ; attribute-type-parameter ( ';' parameter
)*
attributes ::= attribute (';' attribute )*
attribute ::= unique-name*
attribute-type-parameter ::= 'type=' provided-type-unique-name
```

Parametr `attribute-type-parameter` je povinný. Dále lze nepovinně uvést parametr `version`, který specifikuje verzi poskytovaného atributu.

4.2.5 Require-Attributes

Obsahuje vyžadované atributy. Syntaxe je analogicky stejné jako u poskytovaných atributů, pouze parametr `version` je nahrazen `versionrange`, který obsahuje rozsah verzí, v němž se musí nacházet verze poskytovaného atributu, který je hledán.

4.2.6 Ostatní

Poskytované zprávy bundlu jsou uloženy v hlavičce `Generate-Events` a vyžadované v `Consume-Events`. Syntaxe je stejná jako v případě atributů. Poskytované typy se nachází v hlavičce `Provide-Types`, jejíž syntaxe je shodná s `Provide-Services`, a vyžadované typy pak v `Require-Types` se syntaxí hlavičky `Require-Services`.

5 Model mimofunkčních charakteristik

Mimofunkční charakteristika může obsahovat spoustu informací, proto bylo nutné jejich podobu přesně definovat. Dále množina všech potenciálně použitelných charakteristik se může lišit v závislosti na tom, pro jakou oblast použití jsou určeny. Liší se též hodnoty k nim přiřazené v závilosti, pro jaké prostředí jsou použity. Z těchto důvodu byl definován vrstvený repositář, kam jsou charakteristiky ukládány.

5.1 Mimofunkční charakteristiky

Mimofunkční charakteristiky (zkráceně EFP[1]) definuje [5] jako speciální druh informací k vyjádření vlastností poskytovaných a vyžadovaných částí programů. Mimofunkční charakteristikou může být například *požadavek na paměť* nebo *průměrná odezva*. Definovány jsou dva druhy EFP:

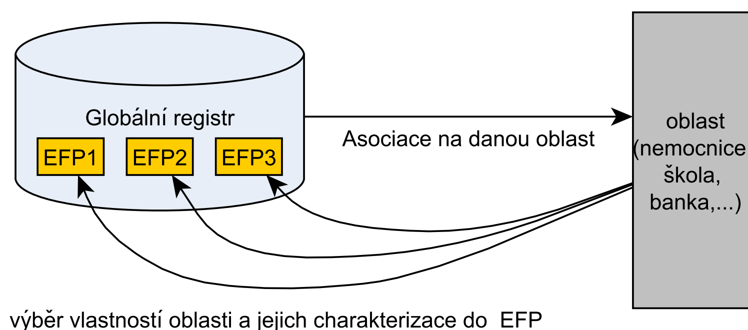
- Jednoduchá (simple) - Popisuje měřitelnou vlastnost často v určitých jednotkách.
- Složená (derived) - Skládá se z jednoduchých a jiných složených EFP.

Společnými atributy charakteristik jsou jméno, porovnávací funkce dvou hodnot přiřazených k EFP a datový typ jejích hodnot. Dále lze specifikovat pro hodnoty měřící jednotku a jména, pod kterými budou vytvořené hodnoty do lokálních registrů ukládány. Složená EFP navíc obsahuje seznam jiných EFP, ze kterých se skládá.

5.2 Globální registr

Globální registr je [5] prostý seznam definic jednoduchých a složených mimofunkčních charakteristik. Je jednoznačně určen identifikátorem, který je asociován s oblastí, pro kterou je definován (viz obrázek 5.1).

[1] Extra-Functional Property



Obrázek 5.1: Schéma významu globálního registru.

5.3 Lokální registr

Lokální registr je [5] úložiště pro hodnoty přiřazené k EFP. Každé výpočetní prostředí (server, telefon, stolní PC) musí definovat vlastní lokální registr.

Každý lokální registr obsahuje identifikátory globálního registru a lokálního registru odvozeného od kontextu, pro který je určen. Dále mu lze přiřadit rodičovský lokální registr, čímž lze definovat jejich celou hierarchii. Potomek pak z něj dědí existující hodnoty, které mohou být přepsány nebo jiné přidány. Poslední částí je seznam hodnot přiřazených k jednoduchým a složeným mimofunkčním charakteristikám ze specifikovaného globálního registru. Narozdíl od hodnot jednoduchých charakteristik obsahují logickou formuli, která definuje vztah k jiným hodnotám charakteristik, ze kterých se složená EFP skládá. Přiřazená hodnota je validní, pokud tato logická formule je vyhodnocena jako pravdivá.

6 Popis aplikace pro přiřazení EFP

Jak již bylo naznačeno v úvodu, cílém této práce je aplikace, která umožní extrahovat z komponenty zdroje, jež jsou jejím vnějším rozhraním poskytovány nebo vyžadovány. Po načtění těchto zdrojů, k nim může uživatel začít přiřazovat mimofunkční charakteristiky nebo editovat existující přiřazení. Při vytváření nového přiřazení je po vybrání charakteristiky nastavena její hodnota. Existuje několik druhů hodnot, které může uživatel k EFP přiřadit:

- Přímá hodnota - konstanta v souladu s datovým typem EFP
- Prázdná hodnota - pouze EFP bez hodnoty
- Hodnota z lokálního registru patřící k EFP
- Matematická formule

Matematická formule [4] specifikuje vztah mimofunkční charakteristiky k jiným EFP různých prvků komponenty. Je vhodná například v následující situaci. Komponenta poskytuje určitou službu *SluzbaA* s přiřazenou mimofunkční charakteristikou *odezva*. Zároveň tato služba pro svou činnost používá jiné služby *SluzbaB* a *SluzbaC*, které mají též přiřazenou EFP *odezva*. Hodnota charakteristiky *odezva* u *SluzbaA* závisí na jejích hodnotách ve službách *SluzbaB* a *SluzbaC*, což lze popsat matematickou formulí, například:

$$SluzbaB_odezva + SluzbaB_odezva$$

Teprve až ve chvíli, kdy je známa trojice (prvek komponenty, EFP, hodnota), může dojít k vytvoření nového přiřazení, které je následně uloženo ke komponentě.

Bohužel není možné vytvořit nástroj umožňující popsanou činnost pro libovolný typ komponenty bez závilosti na komponentovém modelu, který ji definoval. Každý model popisuje zdroje jejího rozhraní jiným způsobem, viz první část tohoto dokumentu. Z toho důvodu je při zavádění podpory pro každý nový typ komponent v této aplikaci nutné seznámit se s tím, jak daný model popisuje její zdroje a teprve podle toho nástroj umožňující přiřazení charakteristik implementovat.

6.1 Její začlenění

Aplikace je součástí většího projektu, jehož schéma je uvedeno na obrázku 6.1. Je výhodné mít charakteristiky definovány již předem v úložišti, o které

se stará jiný modul, a zabývat se tak pouze jejich přiřazením. Přiřazování EFP ke komponentám je samo o sobě k ničemu, pokud neexistuje mechanismus na jejich porovnávání a vyhodnocování.

Obrázek 6.1: Schéma celého projektu

Modul *EFP Repository* implementuje model mimofunkčních charakteristik. Dále má na starosti správu jejich úložiště. K úložišti se přistupuje přes objekt *EFP Client*, který umožňuje provádět:

- Definice a editace globálních registrů
- Správu lokálních registrů a všech EFP v každém globálním registru
- Přiřazení hodnot k EFP a jejich uložení do zvolených lokálních registrů

Správcem úložiště je doménový expert, který musí znát oblast pro definici globálního registru spolu se všemi EFP a lokálních registrů s hodnotami přiřazených k charakteristikám tak, aby byl schopen ji co nejlépe popsat.

Modul *EFP Assignment* je cílem této práce. Uživatel si nastaví typ kompo-

nent, se kterými chce pracovat. Komponenty tohoto typu poté vybere z nějakého úložiště, provede modifikaci přiřazených charakteristik a uloží komponenty zpět. Úložištěm komponent je například adresář, kam jsou ukládány již dokončené komponenty. Typickým uživatele tohoto modulu je vývojář, který komponentu vytvořil a zná její chování i vlastnosti.

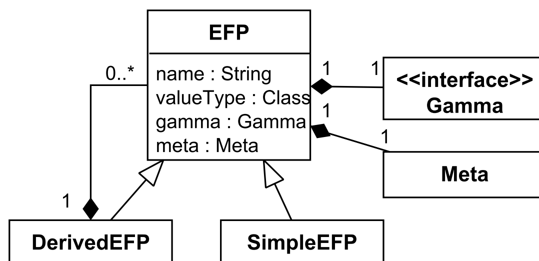
EFP Comparator je externí modul, který umožňuje uživateli provést vyhodnocení kompatibility vybraných komponent. Typickým uživatelem je vývojář systémů, který potřebuje vybrat kompatibilní komponenty a sestavit výsledný funkční produkt.

7 Implementace

V této kapitole je popsána implementace aplikace pro přiřazení charakteristik ke komponentám.

7.1 Deklarace mimofunkčních charakteristik

V kapitole 6.1 bylo zmíněno, že datové typy týkající se mimofunkčních charakteristik jsou deklarovány v modulu *EFP repository*, konkrétně v balících `cz.zcu.kiv.efps.types.*`. V této práci jsou ale používány, tudíž bylo nutné se s nimi blíže seznámit. Na obrázku 7.1 je diagram implementace mimofunkčních charakteristik.

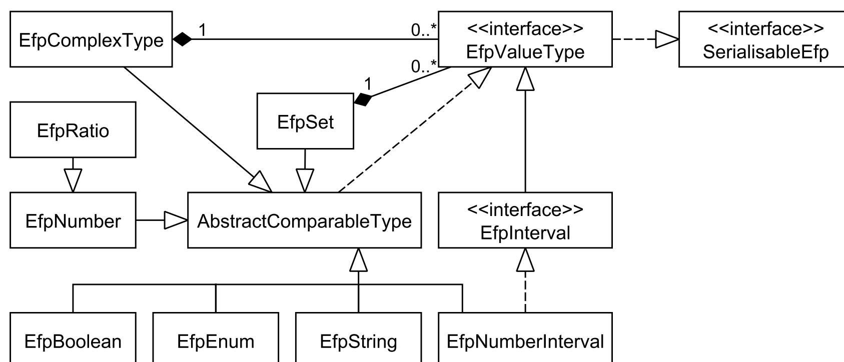


Obrázek 7.1: Mimofunkční charakteristiky podle [4]

EFP je abstraktní bázová třída, od níž jsou odděděny všechny typy mimofunkčních charakteristik. Obsahuje společné atributy definované v kapitole 5.1.

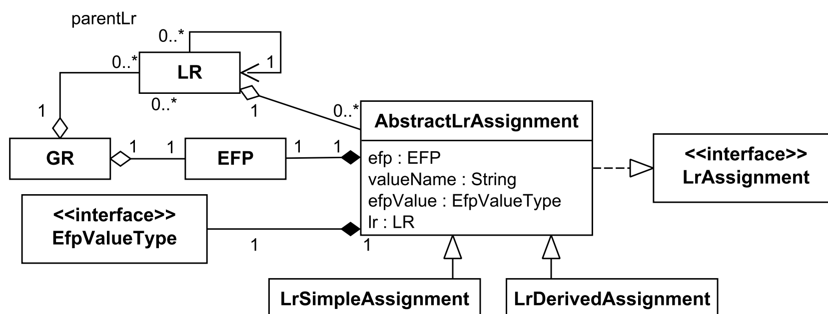
Atribut `valueType` obsahuje referenci na třídu s typem hodnot, kterých bude charakteristka nabývat. Impelementace těchto typů je na obrázku 7.2. Deklarovány jsou základní typy `EfpBoolean` pro pravdivostí hodnotu, `EfpEnum` pro výčtovou hodnotu, `EfpString` pro řetězcovou hodnotu, `EfpNumberInterval` pro číselný interval, `EfpNumber` pro číselnou hodnotu, od něhož je odděděn `EfpRatio` pro procentuální hodnotu. Dále jsou deklarovány složené typy `EfpComplexType` pro skládané hodnoty a `EfpSet` pro množiny hodnot. Složené typy se skládají ze základních nebo jiných složených.

Každý typ implementuje rozhraní `EfpValueType`, jež je rozšířeno o rozhraní `ESerialisableEfp`, které umožňuje serializaci hodnoty do řetězce. Pro zpětné získání hodnoty je deklarována třída `EfpSerialiser`.



Obrázek 7.2: Diagram datových typů hodnot charakteristik podle [4]

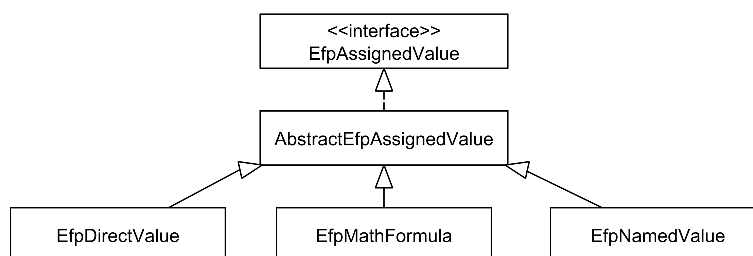
Na obrázku 7.3 je ukázána struktura repositáře. Třída `GR` reprezentuje globální registr, `LR` lokální registr. Od abstraktní třídy `AbstractLrAssignment`, implementující rozhraní `LrAssignment` pro jednotný přístup, jsou odděděny třídy pro jednotlivé druhy přiřazovaných hodnoty do lokálních registrů - `LrSimpleAssignment` pro přiřazení k jednoduché EFP a a `LrDerivedAssignment` ke složené.



Obrázek 7.3: Struktura repositáře

7.2 Druhy přiřazovaných hodnot k EFP

Implementace tříd pro druhy hodnot, které lze přiřadit spolu s mimofunkční charakteristikou k prvku komponenty, jsou deklarovány v balíku `cz.zcu.kiv.efps.assignment.values`. Jejich implementace je ukázána na obrázku 7.4.



Obrázek 7.4: Implementace typů hodnot podle [4]

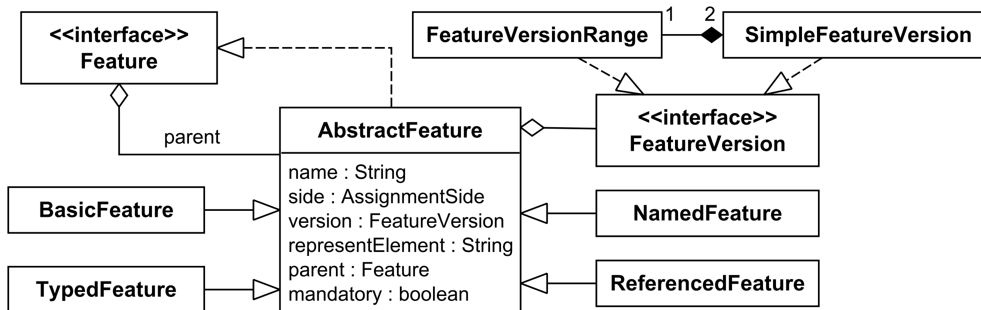
Třída `EfpDirectValue` reprezentuje přímou hodnotu, `EFpNamedValue` je určena pro uložení informací o hodnotě, která je vybrána z některého lokálního registru v repositáři a která patří dané EFP. Do `EfpMathFormula` jsou ukládány matematické formule. Matematická formule je řetězec, ve kterém lze použít kromě základních operací (+, -, *, /, ...) i pokročilejší matematické funkce (odmocnina, logaritmus, ...).

Ke každému prvku komponenty lze přiřadit spolu s jednou EFP několik hodnot z různých lokálních registrů (ale z každého pouze jednu) a jednu přímou hodnotu nebo matematickou formuli.

7.3 Reprezentace prvků komponent

Poskytované a vyžadované prvky rozhraní komponenty je nutné nějakým způsobem v aplikaci reprezentovat. Pro jednotný přístup je definováno rozhraní `Feature` v balíku `cz.zcu.kiv.efps.assignment.types`, které musí být implementováno každou třídou pro specifikaci daného typu prvku. Nejdůležitější jeho metody jsou `getIdentifier` pro získání jedinečného identifikátoru prvku a metoda `matchTo`, která ověří, zda lze provést spárování s jiným prvkem.

Obrázek 7.5 ukazuje implementaci pro jednotlivé typy prvků komponenty. Většina má společnou množinu atributů, které sdružuje abstraktní třída `AbstractFeature`, od níž jsou poté odděděny přímo jednotlivé typy. Jedním z hlavních společných atributů je verze (třída `FeatureVersion`) a rozsah verzí (`FeatureVersionRange`). S oběma se pracuje přes rozhraní `FeatureVersion`.



Obrázek 7.5: Uml diagram reprezetance prvků komponent

Pro spárování prvků musí být jeden poskytovaný a druhý vyžadovaný. Zároveň tyto prvky musí být stejného typu. Další podmínky spárování jsou definovány už jednotlivými typy prvků (například oba musí mít specifický parametr se stejnou hodnotou).

7.4 Rozhraní pro přístup ke komponentě

Komponentu je nutné v aplikaci vhodným způsobem reprezentovat určitou třídou. Tato třída by měla implementovat rozhraní, které umožní s komponentou pracovat nezávisle na jejím typu. Musí obsahovat metody pro:

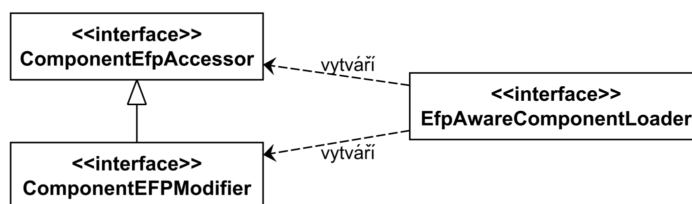
- Načtení všech poskytovaných a vyžadovaných prvků komponenty
- Načtení přiřazených EFP s hodnotami u každého prvku
- Přiřazení, změnu a smazání EFP s hodnotou u každého prvku
- Uložení změn v EFP a uzavření komponenty

Metody pro uložení změn a uzavření komponenty jsou využívány v závilosti na formě distribuce komponent daného typu.

Jelikož například modul *EFP Comparator* pouze čte informace z komponenty, byla implementace rozhraní rozdělena na dvě části. Rozhraní `ComponentEfpAccessor` je určeno čistě pro čtení informací o přiřazení. Pro modifikaci těchto dat bylo definováno rozhraní `ComponentEFPModifier`, které rozšiřuje `ComponentEfpAccessor`, jak znázorňuje obrázek 7.6. Deklarace se nachází v balíku `cz.zcu.kiv.efps.assignment.api`.

7.4.1 Rozšíření aplikace o podporu komponent

K rozšíření aplikace o podporu nového typu komponent je kromě implementace třídy s rozhraním `ComponentEFPModifier` pro její reprezentaci nutné implementovat rozhraní `EfpAwareComponentLoader`. To má za úkol vytvářet dva druhy instancí s reprezentací komponent (viz obrázek 7.6), jež implementují rozhraní definovaná v předchozí kapitole.

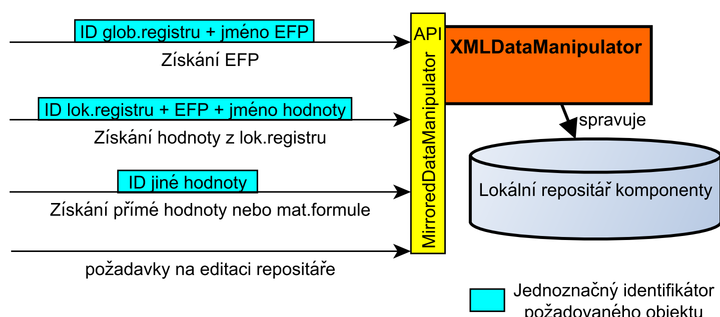


Obrázek 7.6: Uml diagram reprezetance komponent

7.5 Submodul XMLDataManipulator

Schéma na obrázku 7.7 naznačuje, že tento submodul pracuje nad souborem s lokálním repositářem komponenty, což je speciální soubor, kam jsou ukládány mimofunkční charakteristky a jejich hodnoty, které jsou použity v komponentě.

Byl zaveden z toho důvodu, že informace o přiřazených mimofunkčních charakteristikách k prvku rozhraní komponenty může být ukládána přímo k definici prvků komponenty v příslušném souboru. Tento způsob ale má nevýhodu v tom, že kompletní údaj o jednom EFP přiřazení je poměrně objemný



Obrázek 7.7: Schéma submodulu XMLDataManipulator

(EFP má několik atributů, hodnota typu matematická formule je neomezeně dlouhá atd.). Ideálním řešením je tedy místo celého přiřazení vkládat k prvkům pouze vhodné identifikátory EFP a hodnoty. Identifikátory jsou poté použity pro jejich načtení z jiného umístění - lokálního repositáře komponenty. Jiným umístěním by mohl být *EFP repositář*, který by ale měl sloužit pouze k získávání nových ještě nepoužitých EFP. Navíc neumožňuje ukládat přímé hodnoty a matematická formule.

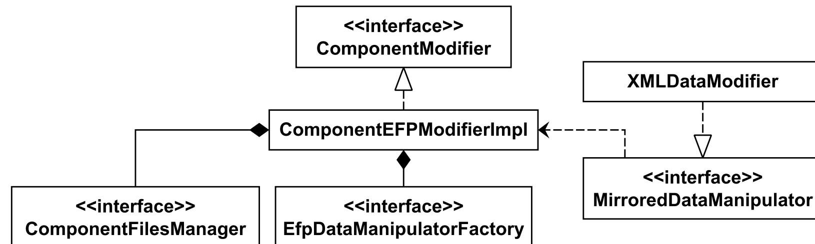
Soubor s lokálním repositářem je ve formátu XML a pro manipulaci s daty je použita technologie *JAXB*, která jednoduše umožní konvertovat XML do Java objektů a naopak. Schéma XML, které je určeno pro generování Java tříd těchto objektů a validaci XML, je uloženo v souboru `localrepository.xsd`. Jeho základní stromová struktura je znázorněna v příloze B.

Implemenentace `XMLDataManipulator` se nachází v balících `cz.zcu.kiv.efps.assignment.repomirror.*`. Se submodulem se pracuje přes rozhraní `MirroredDataManipulator`, jehož metody umožňují kompletní správu lokálního repositáře.

7.6 ComponentEFPModifierImpl

Tato třída v balíku `cz.zcu.kiv.efps.assignment.core` obsahuje obecnou implementaci nástroje pro přiřazování mimofunkčních charakteristik k prvkům komponenty. Důvodem pro jeho vytvoření bylo to, že implementaci nástroje pro přiřazení EFP k různým typům komponent lze rozdělit do částí

(submodulů), z nichž některé mohou být použity nezávisle na několika typech komponent. Z obrázku 7.8 lze vyčíst, že pro implementaci tohoto nástroje pro specifický typ komponent je nutné vytvořit pouze dva submoduly.



Obrázek 7.8: Příklad použití submodulů

Submodul s rozhraním `ComponentFilesManager` spravuje soubory komponenty. Prostřednictvím něho lze získávat a editovat soubory s prvky komponenty a informacemi o EFP přiřazeních.

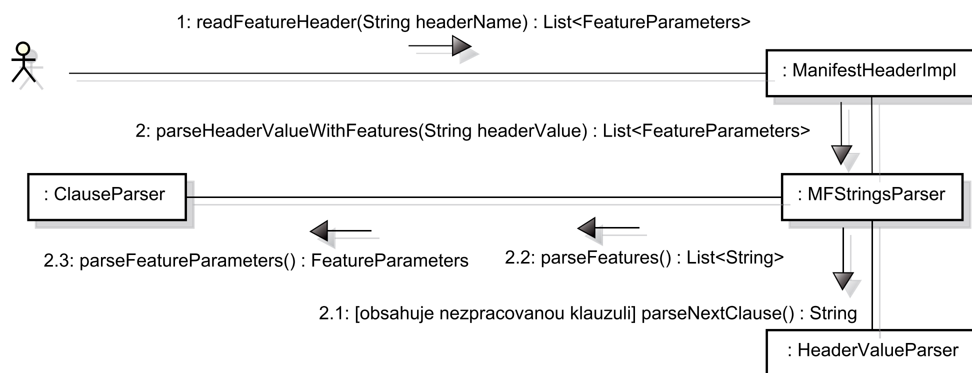
`EfpDataManipulatorFactory` je třída typu továrna, která vytváří instanci druhého submodulu implementujícího rozhraní `EfpDataManipulator`. Úkolem tohoto submodulu je ze souboru získaného prostřednictvím `ComponentFilesManager` načítat jednotlivé prvky rozhraní komponenty a umožnit editaci informací o mimofunkčních charakteristikách přiřazených ke každému prvku. Cílém návrhu bylo, aby k prvkům byly ukládány pouze identifikátory EFP a přiřazené hodnoty. Zbytek je ukládán prostřednictvím interně používaného submodulu `XMLDataModifier` do souboru s lokálním repositářem, k němuž zajištěnu přístup `ComponentFilesManager`.

7.7 ManifestHeaderHandlerImpl

Při bližším pohledu na syntaxi hlaviček s prvky rozhraní komponent v manifestech OSGi a CoSi bundlů si lze všimnout, že jejich formát je stejný, liší se pouze v názvech hlaviček a parametrech u jednotlivých prvků, čehož je možné využít pro vytvoření nástroje, který pracuje nad manifestem s takto definovanou strukturou.

`ManifestHeaderHandlerImpl` v balíku `cz.zcu.kiv.efps.assignment.manifestparser` umožňuje získávat a editovat hodnoty hlaviček manifestu s

prvky komponent OSGi a CoSi. Tato operace je znázorněna na obrázku 7.9. Výsledkem volání metody `readFeatureHeader` je seznam objektů `FeatureParameters`. Každý tento objekt se skládá ze jména vyparsovaného prvku a seznamu jeho parametrů (atributů případně direktiv). Význam parametrů prvků není nijak interpretován, jedná se pouze o dvojice (*klíč*, *hodnota*). O jejich interpretaci se postará až vrstva, které byl vrácen seznam s objekty `FeatureParameters`.



Obrázek 7.9: Parsování řetězce s prvky komponenty a jejich parametery

Třída `HeaderValueParser` rozparsuje hlavičku na klauzule. Každá klauzule obsahuje jedno jméno prvku, nebo množinu jmen prvků se stejnými parametry. Jejich rozparsování a převod do instancí třídy `FeatureParameters` má za úkol třída `ClauseParser`.

7.7.1 CommonBundleFilesManagerImpl

Tato třída s implementovaným rozhraním `ComponentFilesManager` obsahuje submodul pro správu souborů komponenty, která je uložena v JAR souboru. Jako soubor s prvky komponenty a přiřazenými EFP vrací soubor `MANIFEST.MF` ze složky `META-INF`. Do této složky je také ukládán soubor s lokálním repositářem, jehož jméno je interně nastaveno na `localrepository.xml`.

Program, který využívá tento submodul, pracuje s pracovními kopiemi souborů komponenty, které jsou vytvořeny v adresáři komponenty v pomocné složce se jménem *casoverazitko_extract* .

7.8 Implementace pro CoSi framework

Komponentový model CoSi již v sobě má mechanismus pro přiřazení jednoduchých mimofunkčních charakteristik, tudíž bylo možné se zde při implementaci nástroje pro přiřazení charakteristik používaných v této práci inspirovat.

Pro přiřazené charakteristiky byl definován u každého prvku komponenty vlastní atribut, který byl pojmenován `efp`. Z důvodu, které byly nastíněny v úvodu kapitoly 7.5, jsou do tohoto parametru ukládány pouze identifikátory jednotlivých přiřazení. Kompletní informace o přiřazených EFP jsou ukládány do souboru s lokálním repositářem. Struktura hodnoty atributu `efp` je následující:

```
efp ::= (',' assignment)
assignment :: efp '=' value || efp
efp ::= gr-id '.' efp-name
value ::= direct-value || lr-assignment || math-formula
direct-value ::= 'DIRECT{' id '}'
lr-assignment ::= 'LRASSIGN{' lr-id '.' efp-value-name '}'
math-formula ::= 'FORMULA{' id '}'
```

Pro implementaci nástroje pro přiřazení EFP byla použita třída `ComponentEFPModifierImpl`. Jako správce souborů využívá submodul ze třídy `CommonBundleFilesManagerImpl` a submodul pro práci s *manifest* souborem s prvky komponenty je deklarován ve třídě `CoSiEfpAssignsStorageManipulator` v balíku `cz.zcu.kiv.efps.assignment.cosi.manifest`.`CoSiEfpAssignsStorageManipulator` jako spodní vrstvu pro práci s *manifestem* využívá nástroj `ManifestHeaderHandlerImpl`. Třída pro tvorbu objektů reprezentující CoSi komponentu `CosiAssignmentImpl` se nachází spolu s třídou `CoSiEfpDataMnFactory` pro tvorbu instance `CoSiEfpAssignsStorageManipulator` v balíku `cz.zcu.kiv.efps.assignment.cosi`.

K reprezentaci prvků rozhraní komponenty jsou používány následující třídy. Pro služby a typy je určena `NamedFeature`, pro atributy a události `TypedFeature`. Pro celý bundle je určena třída `BasicFeature`.

7.9 Implementace pro OSGi framework

Vychází z implementace pro CoSi komponenty a je uložena v balíku `cz.zcu.-kiv.efps.asignment.osgi.*`. Používá `ComponentEFPModifierImpl` se submodulem `CommonBundleFilesManagerImpl`. Liší se pouze v submodulu pro práci s prvky komponenty v *manifest* souboru, protože pro popis rozhraní komponenty jsou využity jiné prvky. Tento submodul je implementován ve třídě `OSGiEfpAssignsStorageManipulator`.

Na rozdíl od CoSi je pro uložení informací o přiřazených EFP k prvku komponenty definována nová direktiva `efp`. Syntaxí se neliší od atributu pro přiřazené charakteristiky definovaného v CoSi.

K reprezentaci prvků rozhraní komponenty využívá třídy `BasicFeature` a `ReferencedFeature`.

7.10 Uživatelské rozhraní

V rámci aplikace bylo vytvořené jednoduché GUI, které je implementováno v balíku `cz.zcu.kiv.efps.assignment.gui`. GUI splňuje funkčnost, jež byla popsána v kapitole 6. Způsob práce s GUI je popsán v uživatelské příručce v příloze A.

7.10.1 Rozšíření o podporu komponent dalších modelů

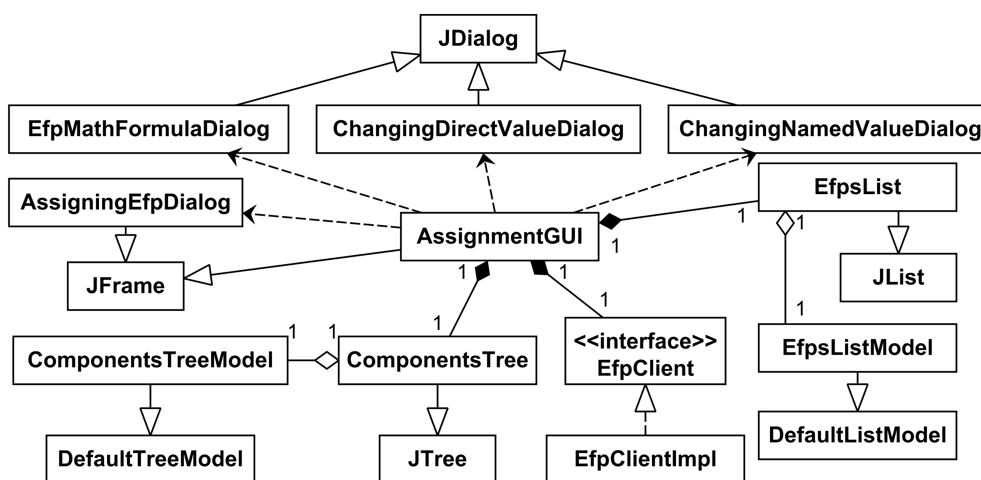
Po implementaci nástroje pro přiřazení EFP ke komponentám dalšího komponentového modelu, je nutné umožnit ho používat v GUI. K tomuto účelu je určen *properties* soubor `supportedcomponents.properties` uložený v kořeni modulu GUI. V něm na každé řádce je uvedena podpora pro jeden komponentový model, řádka má tvar:

$$jmeno_modelu = path$$

Parameter *jmeno_modelu* obsahuje volitelný identifikátor pro rozpoznání modulů pro přiřazení EFP k jednotlivým podporovaným komponentovým modelům. Parameter *path* je *classpath* ke třídě implementující rozhraní `EfpAwareComponentLoader` pro daný komponentový model.

7.10.2 Formuláře a dialogy

Hlavní formulář se nachází ve třídě `AssignmentGUI`, která je znázorněna na obrázku 7.10. Tato třída je vytvořena podle návrhového vzoru *singleton*, čímž je k ní umožněn ze všech objektů v GUI bez nutnosti reference. Toho je například využito při potřebě získávát nějaká data z *EFP úložiště*, protože instance si v sobě drží referenci na objekt implementující rozhraní `EFPClient` z balíku `cz.zcu.kiv.efps.registry.client`, který s úložištěm komunikuje. `AssignmentGUI` využívá předefinované komponenty z knihovny *SWING* `Com-`

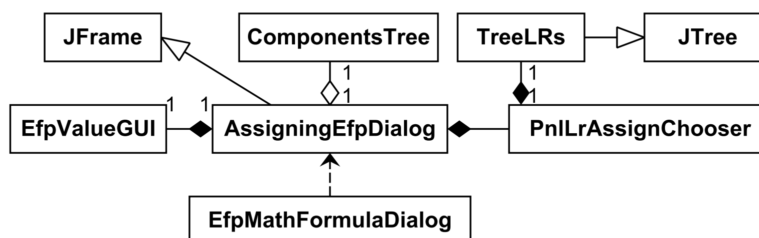


Obrázek 7.10: UML diagram třídy `AssignmentGUI`

`ponentsTree` z balíku `cz.zcu.kiv.efps.assignment.gui.componentstree` pro tvorbu stromu s otevřenými komponentami a `EfpsList` z balíku `cz.-zcu.kiv.efps.assignment.gui.efpslist` pro seznam EFP v aktuálně vybraném globálním registru.

Dále je z hlavního okna otevíráno několik dialogů z balíku `cz.zcu.kiv.-efps.assignment.gui.assigningefpdialog`. Dialogy ve třídách `Changin-gDirectValueDialog` a `ChangingNamedValueDialog` jsou určeny pro editaci již přiřazených hodnot (přímých a z lokálních registrů).

Úkolem dialogu `AssigningEfpDialog` je nastavení hodnoty mimofunkční charakteristiky, která je přiřazována k určitému prvku komponenty. Náznak jeho struktury je ukázán na obrázku 7.11. Drží si v sobě referenci na strom s komponentami `ComponentsTree` pro jeho aktualizaci po vytvoření nového



Obrázek 7.11: UML diagram třídy `AssigningEfpDialog`

přiřazení EFP. Pro nastavení hodnoty z lokálního registru je určen panel `PnlLrAssignChooser`, který pracuje s `TreeLRs`, což je strom se všemi lokálními registry v aktuálním globálním registru.

Pro snadnější vytvoření nebo editaci matematické formule je určen dialog `EfpMathFormulaDialog`[1]. Dialog pracuje s existujícími EFP přiřazeními k prvkům komponenty, tudíž není nutné si při vytváření formule pamatovat, u jakého prvku je jaká charakteristika přiřazena.

Při nastavování přímé hodnoty využívá panely s formuláři z balíku `cz.zcu.kiv.efps.assignment.gui.values` [2]. K vytvoření správného panelu je určena třída `FactoryValueGUI`, která vrací podle datového typu EFP panel pro zadání hodnoty v instanci třídy odvozené od `EfpValueGUI`.

7.11 Spuštění

K sestavení aplikace je používán program *Maven*[3]. Po jeho instalaci je nutné nakonfigurovat přístup ke třem repositářům s knihovnami:

- efps.maven.repository
- efps.maven.snapshots.repository
- efps.maven.third-parties.repository

Poté lze provést překlad prostřednictvím zadání příkazu `maven install` v adresáři projektu `efpParent/trunk`. Pro spuštění aplikace je nutné mít nain-

[1]Tento dialog vytvořil Martin Štulc.
[2]Tento balík vytvořil Martin Štulc.
[3]Dostupný na `<http://maven.apache.org/download.html>`

stalovanou platformu *Java* ve verzi 1.6 a novější[4].

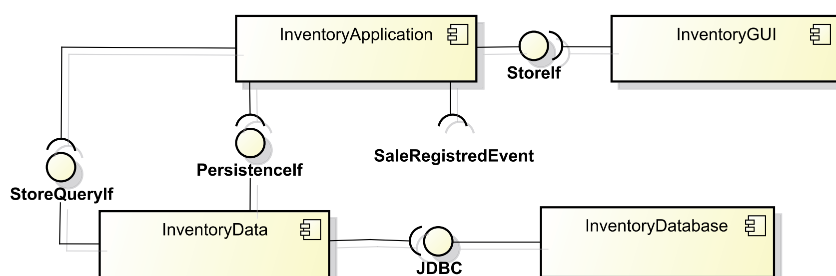
Po sestavení projektu se bude spustitelný JAR soubor s nástrojem pro přiřazení EFP ke komponentám CoSi a OSGi nacházet na cestě:

`efpAssignmentGUI/trunk/target/efpAssignmentGUI-1.0-SNAPSHOT.jar`

[4]Dostupné na `<http://www.java.com/en/download/index.jsp>`

8 Ověření funkčnosti

Funkčnost již částečně byla ověřována během psaní kódu, kde byly k důležitým metodám a třídám psány jednotkové testy. V následujících kapitolách jsou uvedeny postupy pro otestování dokončené aplikace. Protože implementace nástroje pro přiřazení EFP ke komponentám CoSi a OSGi je velmi podobná, je testování ukázáno pouze na CoSi komponentách. Testovací komponenty byly vybrány z návrhu obchodního systému založeného na Common Component Example (CoCoMe) [3]. Pro maximální přehlednost byla použita pouze část systému (viz obrázek 8.1), která se zabývá správou inventáře zboží. V EFP repositáři byly předem připraveny všechny potřebné charakteristiky a hodnoty.



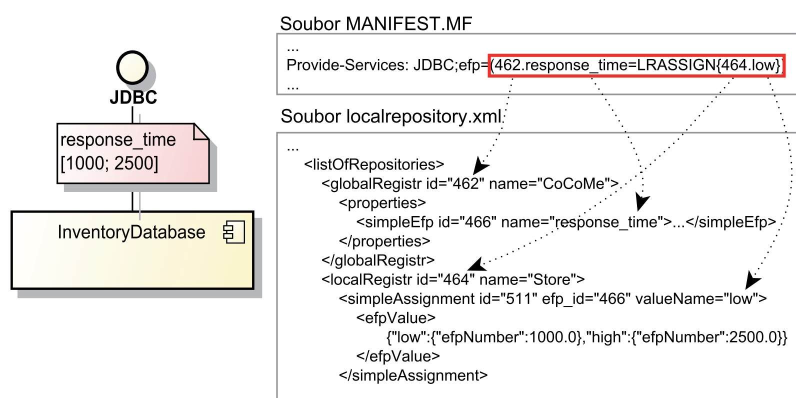
Obrázek 8.1: Schéma inventáře obchodního systému CoCoMe

8.1 Testování přiřazení EFP k CoSi komponentám

8.1.1 Příklad přiřazení EFP s hodnotou z lokálního registru

Komponenta `InventoryDatabase` spravuje databázi všech dat inventáře. Provedení dotazu do databáze v metodách rozhraní `JDBC`, které `InventoryDatabase` poskytuje okolí, má určité zpoždění. Tato vlastnost rozhraní bude popsána mimofunkční charakteristikou.

Přiřazení může probíhat následujícím způsobem. V uživatelském prostředí pro specifikaci zpoždění je vybrána z globálního registru *CoCome* pro obchodní systém EFP *response_time*. Poté je jí z lokálního registru *Store* nastavena hodnota [1000; 2500] se jménem *low*, charakterizující naměřenou hodnotu v daném kontextu jako nízkou. Po uložení stavu komponenty se zapíše do souborů komponenty informace o přiřazení, což je vyznačeno na obrázku 8.2.



Obrázek 8.2: Změny po přiřazení EFP do CoSi komponenty

8.1.2 Příklad přiřazení EFP s prázdnou hodnotou a mat. formulí

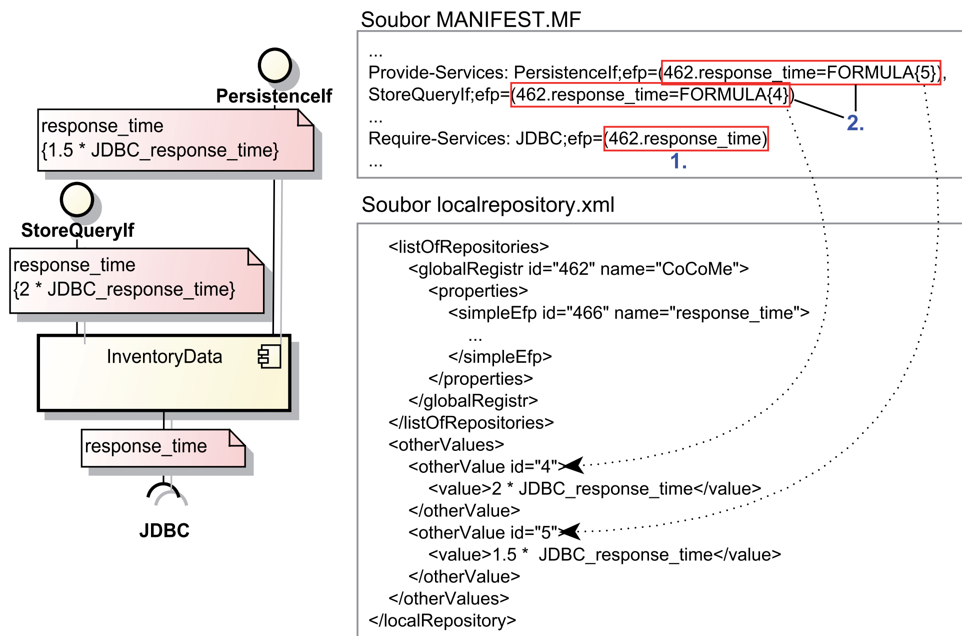
Komponenta `InventoryData` reprezentuje datovou vrstvu nad určitou databází. Pro svou práci vyžaduje komponentu poskytující rozhraní `JDBC`, aby mohla poskytovat svou funkcionalitu přes rozhraní `StoreQueryIf` a `PersistenceIf`. Opět lze charakterizovat zpoždění při volání metod těchto rozhraní. Hodnota již ale částečně závisí na zpoždění při komunikaci přes rozhraní `JDBC`. Například je zjištěno, že zpoždění:

- (a) - u rozhraní `StoreQueryIf` je rovno dvojnásobku zpoždění u `JDBC`.
- (b) - u rozhraní `PersistenceIf` je rovno 1,5 násobku zpoždění u `JDBC`.

Zpoždění na rozhraní `JDBC` je však známo až po připojení komponenty poskytující toto rozhraní s přiřazenou charakteristikou *response_time*. Proto budou vztahy (a) a (b) definovány jako matematické formule. Před jejich vytvořením je nutné specikovat hodnotu *response_time* u vyžadované části rozhraní

`JDBC` v `InventoryData`, tak aby bylo explicitně nastaveno, že tato hodnota *response_time* je rovna hodnotě *response_time* na poskytované části rozhraní `JDBC` z připojené komponenty. Toho lze dosáhnout přiřazením prázdné hodnoty.

Výsledek přiřazení EFP ke komponentě `InventoryData` je ukázán na obrázku 8.3. Nejdříve tedy byla přiřazena *response_time* s prázdnou hodnotou k vyžadovanému rozhraní `JDBC` (bod 1 na obrázku 8.3). Poté mohly být přiřazeny matematické formule (a) k `StoreQueryIf` a (b) k `PersistenceIf` (bod 2 na obrázku 8.3).



Obrázek 8.3: Změny po přiřazení EFP do CoSi komponenty

8.1.3 Příklad přiřazení složené EFP

Tento test je ukázán na komponentě `InventoryApplication`, která má na starosti aplikační logiku pro práci s databází. Poskytuje rozhraní `StoreIf` k vrácení výsledků dotazů a zpracovává událost `SaleRegistredEvent`, která vznikne při prodeji určitého zboží, které je jednoznačně identifikováno čárovým kódem. Vzhledem k různým typům se může stát, že připojená kompo-

nenta generující `SaleRegistredEvent` používá nekompatibilní skener čárových kódů, a proto nebude tato komponenta vyhovovat. K identifikaci uvedeného problému při skládání komponent lze využít mimofunkční charakteristiku.

V příkladu je použita z globálního registru *CoCoMe* složená EFP *bar_code_scanner_type*, která charakterizuje podporované typy čárových kódů skeneru. Skládá se z jednoduché EFP *bar_code_type* definující typ čár.kódu. K *bar_code_scanner_type* je následně vybrána hodnota se jménem *universal* z lokálního registru *Cashdesk*.

Výsledek přiřazení je znázorněn na obrázku 8.4. Ačkoliv je v uživatelském rozhraní přiřazována pouze charakteristika *bar_code_scanner_type* (bod 4 na obrázku 8.4) s hodnotou z lokální registru, musí být do souboru s lokálním repositářem komponenty zapsány všechny objekty, které se jí týkají:

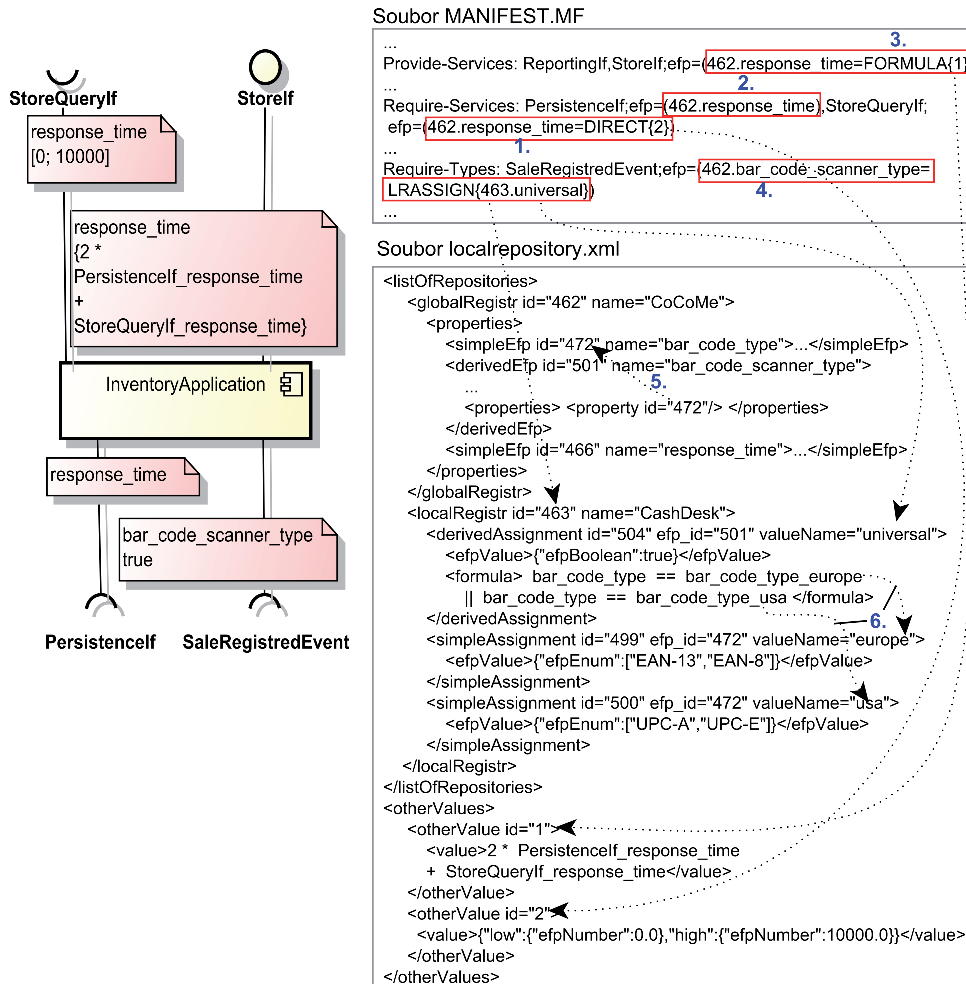
- EFP *bar_code_scanner_type* a přiřazená hodnota se jménem *universal* z lokálního registru *Cashdesk*.
- EFP *bar_code_type*, ze které se *bar_code_scanner_type* skládá (viz bod 5 na obrázku 8.4).
- Z lokálního registru *Cashdesk* všechny hodnoty *bar_code_type*, které jsou použity v logické formuli hodnoty *universal* (viz bod 6 na obrázku 8.4).

8.1.4 Příklad na přiřazení EFP se složitější mat. formulí

Na komponentě `InventoryApplication` je ještě ukázán příklad přiřazení EFP se složitější matematickou formulí. K poskytnutí své funkcionality prostřednictvím rozhraní `StoreIf` využívá přes rozhraní `StoreQueryIf` a `PersistenceIf` jinou komponentu.

U komponenty `InventoryApplication` může být vyžadováno, aby zpoždění na `StoreQueryIf` bylo maximálně 10 s. Je tedy přiřazena EFP *response_time* s přímou hodnotou s intervalem [0;10000] (bod 1 na obrázku 8.4). Dále uživatel může chtít specifikovat zpoždění na StoreIf v závilosti na hodnotách zpoždění na rozhraních `StoreQueryIf` a `PersistenceIf`. Hodnota *response_time* na rozhraní `PersistenceIf` se bude řídit hodnotou na jeho poskytované části v připojené komponentě, tudíž je přiřazena EFP *response_time* s prázdnou hodnotou (bod 2 na obrázku 8.4). Následně je k rozhraní `StoreIf`

přiřazena EFP *response_time* s matematickou formulí se závislostmi na výše uvedených rozhraních (bod 3 na obrázku 8.4).



Obrázek 8.4: Změny po přiřazení EFP do CoSi komponenty

8.2 Testování mazání a editace přiřazených EFP

Testování odstranění přiřazených EFP probíhá přesně opačně. Po každém odstranění přiřazené EFP je zkontrolován u CoSi komponenty *manifest* soubor, kde již nesmí existovat informace o daném přiřazení. Poté je zkontrolo-

ván soubor s lokálním repositářem komponenty, odkud musí být vymazána odstraňovaná EFP a její hodnota ovšem pouze v případě, že nejsou součástí jiného přiřazení v komponentě.

Editace přiřazených EFP je složením operací mazání přiřazené EFP a přiřazení stejné EFP pouze s jinou hodnotou.

9 Závěr

Uživatelé, kteří pracují s komponentami třetích stran, vyžadují pokročilé techniky ověřování kompatibility a možnosti záměn komponent. Jednou z cest, jak splnit tyto požadavky, je přiřazení mimofunkčních charakteristik ke komponentám, což bylo tématem této práce.

Práce je rozdělena do 3 částí. V první části jsou vysvětleny základní pojmy z oblasti komponentově orientovaného programování. Poté jsou popsány komponentové modely OSGi a CoSi a v závěru model mimofunkčních charakteristik. Druhá část se zabývá implementací nástroje pro přiřazení charakteristik ke komponentám OSGi a CoSi. Třetí část se zabývá předvedením a ověřením funkčnosti dokončené aplikace.

Práce splňuje požadavky, které byly kladeny na jejím začátku, a při testování výsledné aplikace byly odstraněny případné chyby. Proto jí lze považovat za úspěšně dokončenou.

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] Brada, P.; Wajtr, B.; Liška, V.: The CoSi Component Model - Specification of CoSi version 2. Technická zpráva.
URL <http://www.kiv.zcu.cz/research/groups/dss/projects/cosi.html>

[3] Herold, S.; Klus, H.; Welsch, Y.; aj.: *Common component modelling example (CoCoME)*. 2010.
URL <http://agrausch.informatik.unikl.de/CoCoME/downloads>

[4] Ježek, K.: A Complex Meta-model for Extra-functional Properties Concerning Common Data Types Their Comparing and Binding. In *2nd World Congress on Software Engineering (WCSE 2010)*, ročník 2, 2010, ISBN 978-0-7695-4303-1, s. 71–74.

[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 2007, release 4, verze 4.1.
URL <http://www.osgi.org>

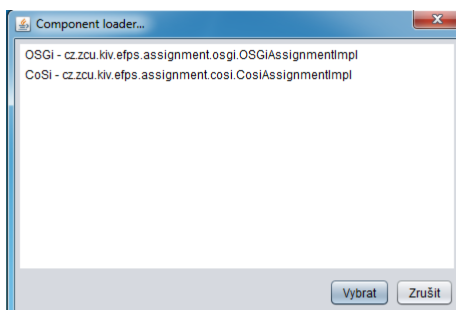
[7] Szyperski, C.: *Component Software - Beyond Object-Oriented Programming*. ACM Press, 2002, ISBN 0-201-74572-0, 60-75 s.

Přílohy

A Uživatelská příručka

A.1 Výběř pracovního typu komponent

Dialog zobrazený na obrázku A.1 umožňuje vybrat mechanismus pro přiřazení charakteristik k určitému typu komponent. Tento dialog je vyvolán automaticky při prvním spuštění aplikace nebo je možné ho zavolat přes menu *Komponenta->Změnit Component Loader* v hlavním okně (obr. A.2).

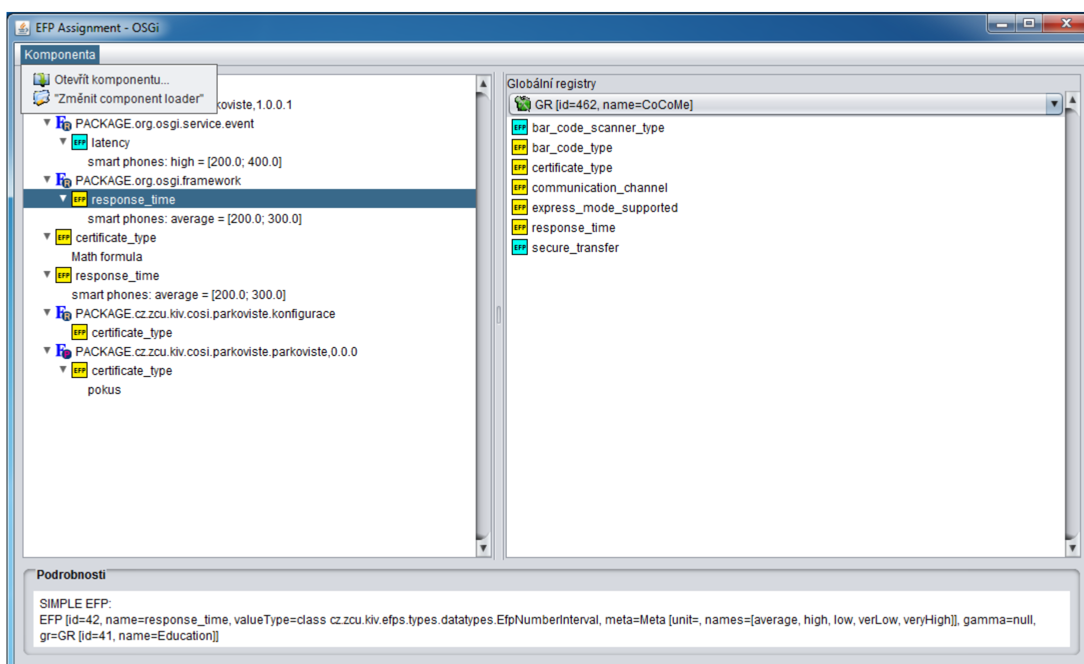


Obrázek A.1: Hlavní okno

A.2 Otevření a zavření komponenty

Uživatel otevře komponentu přes položku menu *Komponenta->Otevřít komponentu* v hlavním okně. Poté vybere v dialogu soubor s komponentou, čímž dojde k načtení jejích poskytovaných a požadovaných prvků spolu s přiřazenými mimofunkčními charakteristikami. Tato data jsou zobrazena ve stromové struktuře v levé části hlavního okna. Kořen komponenty je označen červenou ikonkou se znakem 'C' a jménem souboru komponenty. Potomkem kořenu jsou jednotlivé prvky komponenty, jejichž potomky jsou přiřazené mimofunkční charakteristiky. Pod každou charakteristikou jsou pak přiřazené hodnoty.

Pokud uživatel chce komponentu zavřít nebo uložit do ní změny, klikne pravým tlačítkem myši na kořen komponenty, čímž se mu zobrazí kontextové menu s těmito možnostmi.

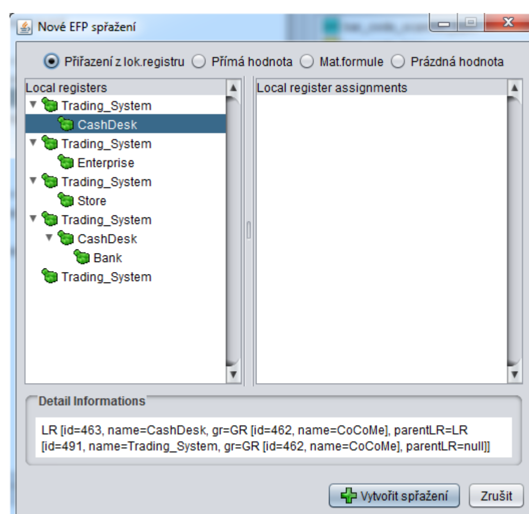


Obrázek A.2: Hlavní okno

A.3 Přiřazení a editace charakteristiky u prvku komponenty

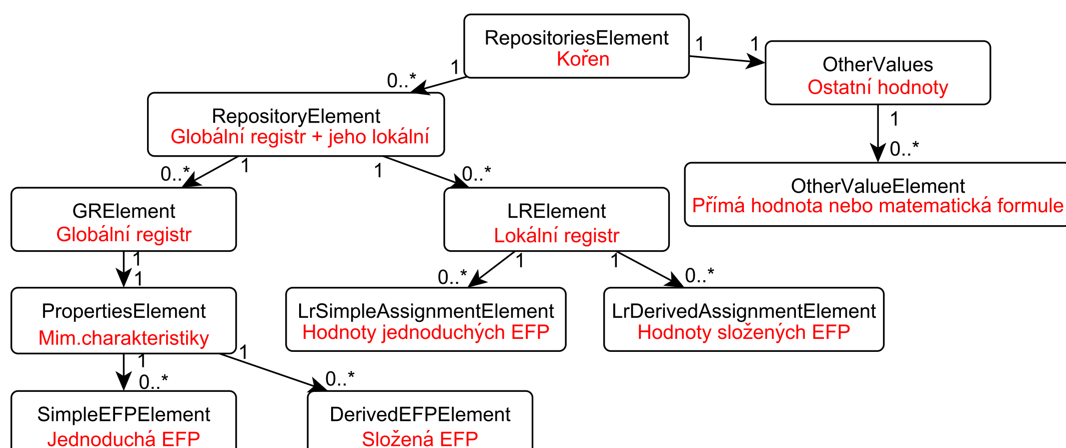
V pravé horní části hlavního okna uživatel nejdříve vybere globální registr. Pod ním se zobrazí charakteristiky, které jsou v něm definovány. Jednoduché EFP jsou rozlišeny žlutou ikonkou od složených s azurovou barvou. Pro přiřazení EFP k prvku komponenty stačí tuto charakteristiku přetáhnout z pravého panelu do levého na zvolený prvek. Dojde k otevření dialogu (obr. A.3), v němž je nastavena hodnota EFP.

Uživatel může hodnotu přiřazené EFP změnit nebo smazat i spolu s EFP. K této akci je určeno kontextové menu, které se zobrazí kliknutím na pravé tlačítko myši nad editovanou hodnotou nebo nad EFP pro její smazání včetně přiřazených hodnot. Při editaci hodnoty uživatel pracuje se stejným dialogem jako při nastavování nové hodnoty.



Obrázek A.3: Hlavní okno

B Struktura XML s lokálním repositářem



Obrázek B.1: Schéma XML pro lokální repositář