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

Bakalářská práce

Vizualizace komponentových aplikací z úložiště CRCE

Plzeň, 2014 Jan Ambrož

Prohlášení

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

V Plzni dne 9. května 2014

Jan Ambrož

Poděkování

Tímto bych chtěl poděkovat svému vedoucímu bakalářské práce,
panu doc. Ing. Přemyslu Bradovi, MSc. Ph.D., za odborné vedení, vstřícnost
a cenné rady při zpracování této práce.

Abstract

The subject of this thesis is component based tool made upon Eclipse RCP named ComAV. The main purpose of this thesis is to prepare new loader plugin into this tool – CRCE loader. CRCE is a repository which can hold different kinds of components inside of it and can provide component descriptions. CRCE loader should load these descriptions and transform them into ENT objects which ComAV can deal with.

The plugin loads metadata from CRCE over REST API. These metadata are transmitted as XML structure files and are parsed with SAX parser. The result of parsing is a map with components in ENT represantion without bindings between them. CRCE loader reuses binding logic from other loaders such as OSGi loader, EJB3 loader or SOFA2 loader and creates bindings between components. These components and their bindings can be visualized then.

Task of this bachelor thesis was successfully finished. There is the possibility in ComAV tool to create new CRCE project now. This new tool feature is internally using steps described in previous paragraph to create new CRCE project. Because CRCE loader is dependent on CRCE current state, consideration has been made and CRCE loader is designed in order with awaited changes with CRCE repository.

Abstrakt

Předmětem této práce je komponentově orientovaný nástroj ComAV postavený nad platformou Eclipse RCP. Hlavním cílem této práce je připravit nový plugin typu loader do tohoto nástroje – CRCE loader. CRCE je úložiště, které umožňuje uskladňovat různé typy komponent a poskytuje jejich popis. CRCE loader by měl tento popis načíst a převést do objektů ENT meta-modelu, se kterými nástroj ComAV dokáže dále pracovat.

Plugin načítá metadata z CRCE pomocí REST API. Tato metadata jsou přenášena v podobě XML souborů a jsou parsována pomocí SAX parseru. Výsledkem parsování je mapa s komponentami, mezi kterými neexistují žádné vazby. CRCE loader využívá logiky utváření vazeb z ostatních loaderů, jako je OSGi loader, EJB3 loader či SOFA2 loader, a vytváří vazby mezi komponentami. Tyto komponenty mohou být poté společně s jejich vazbami vizualizovány.

Úkol této bakalářské práce byl úspěšně dokončen. V nástroji ComAV je nyní možnost vytvořit nový CRCE projekt. K vytvoření nového CRCE projektu nástroj interně používá kroky popsané v předchozím odstavci. Protože je CRCE loader závislý na stavu CRCE, byla tomu během vývoje věnována pozornost a plugin je navržen s ohledem na očekávané změny v podobě CRCE úložiště.

Obsah

1 Úvod

V softwarovém inženýrství stále častěji docházelo k situacím, kdy již byla určitá část funkčnosti implementována v jiné existující aplikaci. Postupem času se proto začaly tyto specifické části kódu oddělovat do samostatných znovupoužitelných jednotek – komponent, které logicky sdružovaly funkčnost či data a mohly být dále použity v jiných komponentových aplikacích. Tento komponentově orientovaný přístup v programování nakonec dozrál do fáze, ve které se z komponent začaly stavět celé systémy. Protože bylo třeba jednotlivé komponenty někde uchovávat, začala vznikat internetová komponentová úložiště. Na tato místa se komponenty uskladňovaly a odtud mohly být dále distribuovány.

Jedním takovým úložištěm je také *CRCE* (**C**omponent **R**epository supporting **C**ompatibility **E**valuation), tj. komponentové úložiště podporující ověření kompatibility. Oproti tradičnímu komponentovému úložišti však disponuje schopností validovat závislosti mezi uskladněnými komponentami – tedy ověřovat jejich kompatibilitu.

Záměrem pojícím se komponentovou aplikací však nemusí být jen rozhodnutí o vzájemné kompatibilitě jejích komponent. Někdy může být velice účelné získat nadhled nad komponentovou aplikací a umět si udělat celkovou představu o její vnitřní stavbě. Efektivní způsob, jak toho dosáhnout, poskytuje nástroj *ComAV* (**Com**ponent **A**pplication **V**isualizer), tj. vizualizér komponentových aplikací. Nástroj umožňuje načítat komponentové aplikace a vizualizovat jejich závislosti.

Cílem této práce je vytvořit propojení nástroje ComAV s úložištěm CRCE. Požadovaným efektem tohoto propojení by měla být vizualizace komponent uskladněných v CRCE v nástroji ComAV. Logika zpracování komponentových aplikací určených k vizualizaci v nástroji ComAV byla doposud čistě jen záležitostí nástroje ComAV. Nástroj fyzicky načítal jednotlivé komponenty a prostřednictvím reverzního inženýrství analyzoval jejich podobu. V případě CRCE loaderu však potřeba této prvotní analytické funkčnosti odpadá. Analýza uskladněných komponent totiž probíhá již na straně CRCE. Úložiště poté poskytuje popis uskladněných komponent založený na oné analýze. Úkolem CRCE loaderu vytvářeného v této práci tedy bude tento popis zpracovat a vytvořit na aplikační úrovni ComAV odpovídající datové reprezentace komponent.

V první kapitole je čtenář seznámen s tím, co pojem *komponenta* představuje a jaké jsou další její souvislosti. Druhá kapitola zmiňuje a popisuje použité technologie ENT a Eclipse RCP, které jsou s nástroji ComAV a CRCE úzce spjaty a jejichž znalost byla pro dosažení cíle této práce kritická. Následuje kapitola, která se věnuje samotné problematice nástrojů CRCE a ComAV. Poslední kapitola pak pojednává o vlastním procesu tvorby pluginu – od specifikace požadavků, analýzy a návrhu až po implementaci a testování. V závěru je vše objektivně zhodnoceno a jsou uvedena možná rozšíření.

2 O komponentách

Kapitola seznamuje čtenáře s problematikou komponentově orientovaného programování. V kapitole jsou použity znalosti z knihy [1], technických článků [3] a [6] a diplomové práce [8].

2.1 Definice komponenty

S termínem *komponenta* se v oblasti softwarového inženýrství setkáváme poměrně často. Konkrétně se tento pojem objevuje v komponentově orientovaném programování neboli také *CBSE* (**C**omponent-**b**ased **S**oftware **E**ngineering). Alternativou pojmu komponenta může být také termín *modul*. Oba tyto pojmy mají víceméně stejný význam, který ovšem bývá vykládán různými způsoby. Podle uznávané definice pocházející z knihy *Component Software: Beyond Object-Oriented Programming* – [1] se jedná o logicky uspořádaný funkční celek – přesněji softwarový balík či modul – zapouzdřující funkce či data.

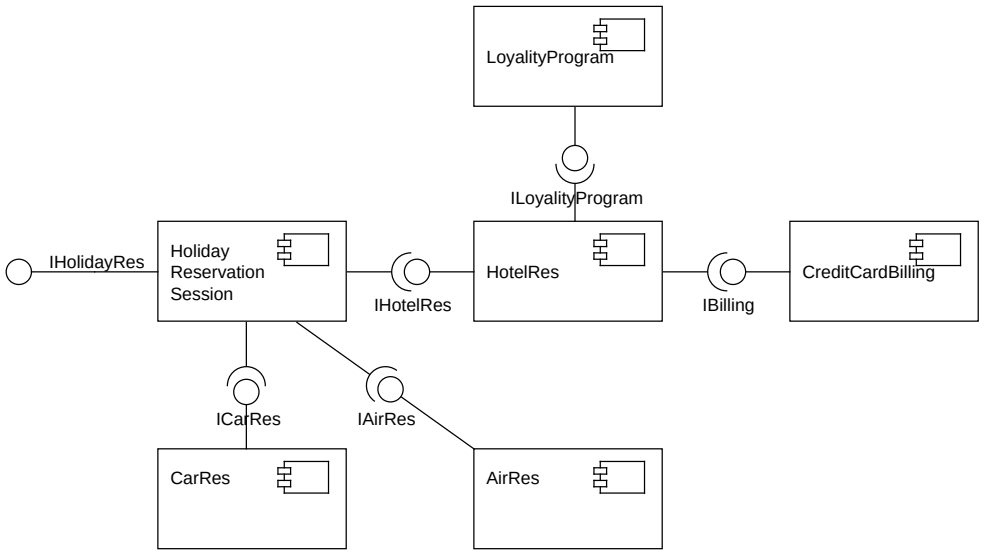
Co se týká stavby komponenty, tak implementace komponenty bývá často skryta a navenek se prezentuje pouze její rozhraní. Komponenta s takovým charakterem bývá označována jako *černá skříňka* neboli *blackbox*.

2.2 Využití komponent

Jedním z možných způsobů jak vyvíjet softwarové aplikace je analýza zadání a rozklad problému na menší dílčí podproblémy. Logicky oddělené autonomní části, které pak samostatně řeší každý takový podproblém, odpovídají komponentě. Sjednocením těchto komponent pak získáváme komponentovou aplikaci, která se navenek tváří jako jeden funkční celek a plní zadané požadavky.

Příkladem takové komponentové aplikace může být jednoduchý rezervační systém zachycený na obrázku 2.1. Jednotlivé spojnice představují vazby mezi komponentami. Směr vazeb říká, zda komponenta poskytuje určitou svoji funkčnost (prázdné kolečko), či zda vyžaduje určitou funkčnost jiné

komponenty (půlkruh). Názvy vazeb odpovídají názvům rozhraní dodávaných komponent.



Obrázek 2.1: Příklad komponentové aplikace

Komponentově orientovaný přístup se na první pohled může zdát zbytečně kostrbatý a komplikovaný. Člověk by raději rovnou přistoupil k implementaci a příliš se nezabýval otázkou logického členění programu. Často má však takový přístup své opodstatnění. Nejenže na samém počátku správná analýza programu eliminuje problémy, se kterými bychom se pravděpodobně v blízké budoucnosti potýkali, ale současně tato strategie přináší řadu výhod.

Jednou z výhod může být například znovupoužitelnost komponent. Pokud se totiž již při návrhu programu ukáže, že již existuje komponenta, která nabízí požadované služby, pak je možné takovou komponentu využít bez nutnosti implementace již existujícího kódu. To nám pochopitelně šetří čas a často i peníze.

Další výhodou může být oprava nebo dokonce samotné nahrazení jedné z komponent. Jak již víme, komponenta logicky zapouzdřuje a samostatně funguje jako jeden funkční celek. Oprava či pouhá aktualizace komponenty tak ve většině případů bude znamenat zásah pouze v rámci jedné z komponent. Naopak úplné nahrazení komponenty je v některých případech možné dokonce za běhu bez nutnosti překladu. Příkladem komponentového modelu

s takovou podporou je například OSGi. S vyměnitelností komponent se však pojí také jedna z nevýhod vývoje komponentových aplikací.

Vývoj komponentových aplikací často zahrnuje využití komponent třetích stran. Tento přístup dokáže ušetřit mnoho úsilí, má však i své stinné stránky. Využívaná komponenta třetích stran může obsahovat chyby nebo může vykazovat neočekávané chování.

2.3 Komponentový model

Komponentový model si můžeme představit jako soubor pravidel determinujících podobu komponenty daného komponentového modelu. Takový soubor zahrnuje pravidla specifikující podobu komponenty, chování komponenty a způsob komunikace komponenty s ostatními komponentami.

Jako příklad komponentového modelu můžeme uvést *EJB*[9], *OSGi*[10] či *SOFA*[11].

2.4 Komponentový meta-model

Máme definovaný komponentový model. Rádi bychom se ale posunuli v abstrakci o úroveň výše a nalezli prostředek, který vůbec umožní popis struktury komponentového modelu nezávisle na typu modelu. Jinými slovy, hledáme jazyk dostatečně obecný na to, aby umožnil jednoznačný popis všech možných existujících komponentových modelů.

Existence takového meta-modelu nám pak umožní dynamicky vytvářet objekty v paměti, které svou strukturou budou odpovídat struktuře komponenty daného komponentového modelu a se kterými budeme dále schopni pracovat na aplikační úrovni.

Na základě získaných znalostí ze specifikací neziskového konzorcia *Object Management Group* – [2] můžeme říci, že existují čtyři abstraktní úrovně datových struktur popisujících komponentové aplikace.

- **MO** – konkrétní implementace elementů komponentového modelu
- **M1** – struktury pro popis komponenty – komponentový model
- **M2** – struktury definující podobu odpovídajícího komponentového modelu – komponentový meta-model
- **M3** – struktura jazyka popisujícího komponentový meta-model – komponentový meta meta-model

Znalost konceptu modelu a meta-modelu je pro pochopení této práce klíčová. V prvním oddíle následující kapitoly je představena technologie, která umožňuje tyto koncepty reprezentovat pomocí datových struktur.

3 Technologie ENT a Eclipse RCP

Kapitola seznamuje čtenáře s technologiemi, které jsou použity v nástrojích ComAV a CRCE.

3.1 ENT meta-model

Nejprve je nutné představit ENT meta-model, který je nedílnou součástí nástroje ComAV a který je nezbytný pro jeho funkční běh. Tento oddíl vychází z nabytých znalostí z článků [3] a [4].

3.1.1 Představení

Jak již bylo zmíněno výše v oddíle 2.4, struktury popisující komponentové aplikace dělíme do 4 úrovní. ENT meta-model řadíme do skupiny *M3*. Jedná se tedy o model, který umožňuje definovat struktury komponentových modelů a komponentově zaměřených aplikací.

Účelem existence tohoto modelu je především schopnost obecného sémantického popisu komponent. Snahou je popis utvořit tak, aby byl maximálně srozumitelný pro uživatele s rozdílnými zájmy. Uživatelem zde rozumíme vývojáře, softwarového architekta, testera, apod.

ENT meta-model je vyvíjen na Katedře informatiky a výpočetní techniky Západočeské univerzity od roku 2002. Zkratka ENT a zároveň i název zkoumaného meta-modelu vychází ze slov **E**xport, **N**eeds a **T**ies. Postupně tato slova zastupují to, co zkoumaná komponenta poskytuje, co vyžaduje a jaké jsou její vnitřní vazby.

3.1.2 Architektura ve zkratce

Architektura ENT meta-modelu je své podstatě dvouúrovňová. První úroveň představuje vůbec popis toho, jak daný komponentový model vypadá. Jedná se tedy o definice struktur komponent daného komponentového mo-

delu. Druhá úroveň je pak reprezentována konkrétními komponentami, které byly na základě definovaných forem z první úrovně vytvořeny.

Abstraktní úroveň

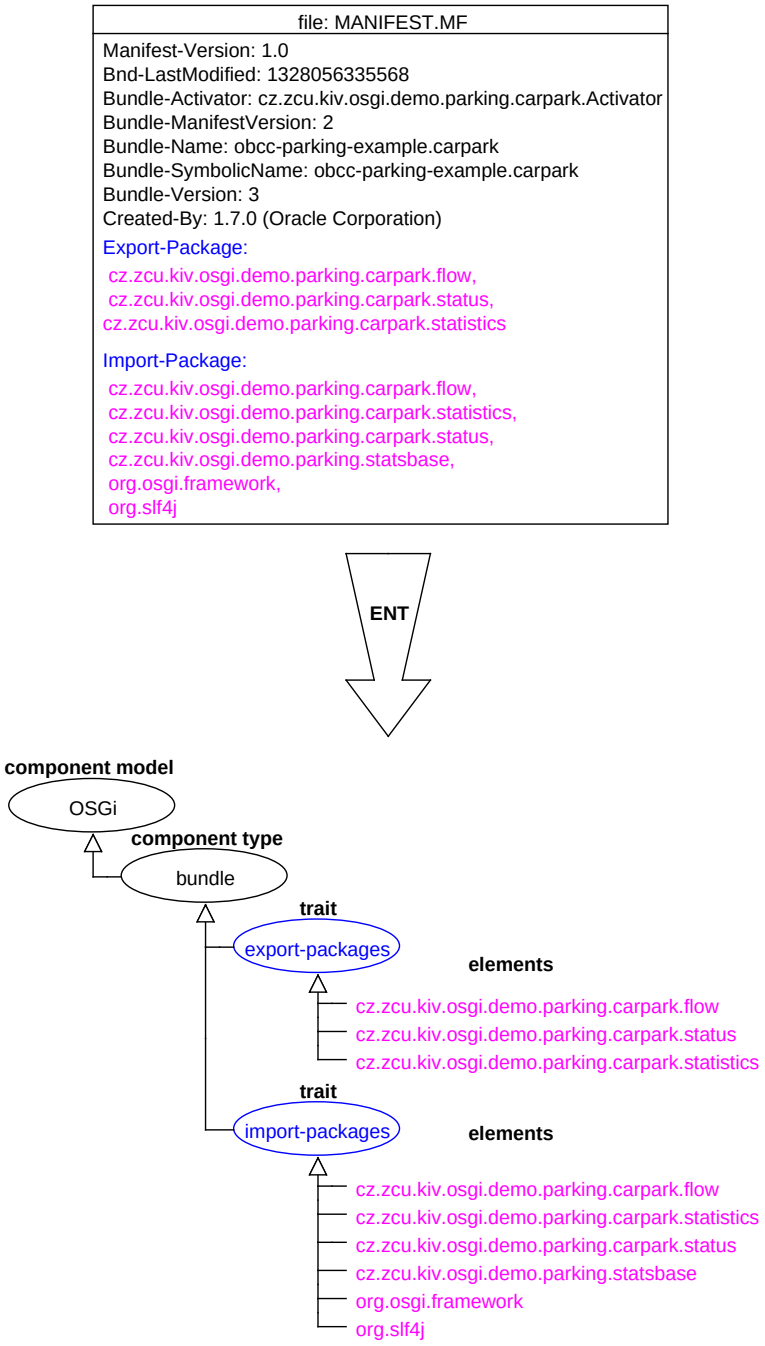
Abstraktní úroveň neboli také úroveň komponentového modelu si můžeme představit jako strom, jehož konkrétní podoba závisí na popisovaném komponentovém modelu.

Kořenem tohoto stromu je vždy konkrétní komponentový model. Elementům nižší úrovně (kořen je rodičem) pak odpovídá množina typů komponent daného komponentového modelu. Pro lepší představu můžeme uvést příklad, kdy pro komponentový model OSGi existuje jediný komponentový typ *bundle* a pro komponentový model EJB existuje trojice komponentových typů: *SessionBean*, *MessageDriverBean* a *Entities*.

Dalším významným elementem, který se v hierarchickém stromě nachází pod odpovídajícím komponentovým typem (potomek komponentového typu), je *trait*, resp. jejich celá množina. Jednotlivé elementy typu trait pak seskupují *elementy*[1] se společnými charakteristickými vlastnostmi v rámci daného komponentového typu. Zůstaneme-li u našeho příkladu s komponentovým modelem OSGi, můžeme si pod elementem typu trait představit například vlastnost bundlu *import-packages* či *export-packages*. *Elementům* by pak odpovídala množina konkrétních balíčků.

Pro ilustrační účel je přiložen obrázek 3.1, který zobrazuje vztah mezi hlavičkami manifestového souboru OSGi bundlu a abstraktní úrovní ENT meta-modelu.

[1] V tomto případě se nejedná o elementy z hierarchického hlediska, ale o prvky, které v ENT nesou označení *element*.



Obrázek 3.1: ENT model pro ukázkový bundle

Neméně významnými elementy jsou tzv. *tagy*, které můžeme nalézt rozprostřené napříč celým hierarchickým stromem. Jednotlivé tagy si můžeme představit jako vlastnosti daného elementu (element, ke kterému dané tagy přísluší, je ve stromě jejich rodičem). Například pro zmíněný bundle by takový tag mohl nést informaci o verzi bundlu či jeho symbolickém jméně.

Aplikační úroveň

Aplikační úroveň ENT meta-modelu již odpovídá realizaci nadefinovaných struktur na úrovni komponentového modelu. Jinými slovy, zatímco abstraktní úroveň se pouze starala o popis formy daného komponentového modelu, aplikační úroveň je již reprezentována objekty v paměti vytvořených podle odpovídající formy.

Je nutné si uvědomit, že na této úrovni již můžeme mluvit o vazbách mezi komponentami v rámci stejného komponentového modelu. Tyto vazby nemusí existovat pouze mezi komponentami, ale mohou se vyskytovat izolovaně uvnitř některé z komponent.

3.1.3 Soubor definic ENTMM

Obecný koncept popsaný výše je nutné pro účely programové implementace reprezentovat vhodnou datovou strukturou. Pro ni je v našem případě použita technologie *EMF* (**E**clipse **M**odeling **F**ramework)[2], která pro popis meta-modelu používá XML soubor s příponou *.entmm*.

Obsahem tohoto pododdílu je ukázka *OSGI.entmm* souboru (viz další stránka). Tento soubor popisuje komponentový model OSGi. Ukázka by měla pomoct pochopit aplikaci ENT meta-modelu v praxi.

Je vidět, že struktura ENT meta-modelu je popsána pomocí značkovacího jazyka XML. Ukázka je oproti skutečnému souboru *OSGi.entmm* značně zjednodušena. Řetězec "..." označuje v souboru místa, na kterých byly vynechány elementy.

[2]Framework *EMF* dokáže vygenerovat množinu JAVA tříd umožňujících na aplikační úrovni reprezentovat datový model popsaný pomocí příslušného XML souboru.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ENTMM:ComponentModel xmi:version="2.0" xmlns:xmi="http://www.omg.
       org/XMI" xmlns:ENTMM="http:///ENTMM.ecore" name="OSGi">
3    <componentSet ctname="Bundle" tagSet="//@tagSet.0␣//@tagSet.2"
         traitSet="//@traitSet.0␣//@traitSet.1␣//@traitSet.2␣//@traitSet
         .3␣//@traitSet.4␣//@traitSet.5␣//@traitSet.7␣//@traitSet.6"/>
4    <tagSet name="symbolic_name" default="e" metatype="string"/>
5    <tagSet name="version" default="0.0.0" metatype="versionidentifier
         "/>
6    ...
7    <traitSet name="export_packages" metatype="class" tagSet="//
         @tagSet.1␣//@tagSet.2">
8      <traitClassifier granularity="structure" arity="multiple"
           construct="type" presence="permanent">
9        <role>provided</role>
10       <lifecycle>development</lifecycle>
11       <lifecycle>assembly</lifecycle>
12       ...
13     </traitClassifier>
14   </traitSet>
15   <traitSet name="import_packages" metatype="class" tagSet="//
         @tagSet.3␣//@tagSet.4␣//@tagSet.5␣//@tagSet.1␣//@tagSet.11␣//
         @tagSet.12">
16     <traitClassifier granularity="structure" construct="type"
           presence="permanent">
17       ...
18     </traitClassifier>
19   </traitSet>
20   ...
21 </ENTMM:ComponentModel>
```

Obrázek 3.2: Ukázkový OSGI.entmm soubor

Na řádce 2 je řečeno, jaký komponentový model soubor popisuje. Tuto informaci podává atribut `name`, jehož hodnota je "OSGi". Element `componentSet` na řádce 3 popisuje jediný typ komponenty v OSGi – "bundle". Atributy `tagSet` a `traitSet` definují, které elementy typu *tag* a *trait* přísluší komponentovému typu *bundle*. Konkrétní prvky jsou odděleny pomocí dvojice lomítek a číslo za elementem odpovídá indexu elementu v *.entmm* souboru, resp. jeho pořadí.

Na základě obsahu *.entmm* souboru se na aplikační úrovni vystaví objekt se stejnou strukturou popsanou pomocí tohoto souboru. Přístup do tohoto souboru a vyhledání elementu s příslušným názvem pak umožňuje získat definici hledaného elementu ENT meta-modelu.

3.2 Eclipse RCP

Oddíl seznamuje čtenáře s platformou Eclipse RCP(**R**ich **C**lient **P**latform), nad kterou je nástroj ComAV postaven. V oddíle se vychází ze znalostí z knihy [5] a diplomové práce [8].

3.2.1 Představení

Spojení *rich client* se poprvé začalo objevovat na začátku 90. let 20 století. Tou dobou se aplikace vyvíjely v tehdy velmi populárních *IDE*[1] *Delphi* a *Visual Basic*. Oblast informačních technologií zaznamenávala dramatický rozvoj a na trh se dostávalo stále více a více klientských aplikací. Snahou bylo vyvíjet aplikace uživatelsky přívětivé. Aplikace, které budou mít nativní uživatelské rozhraní a jejichž odezva bude pro uživatele vzbuzovat pocit uspokojení. Termín *rich client* se používal k odlišení takových aplikací od těch ostatních (terminálové či aplikace s minimální „user experience“[2]).

Eclipse má nejspíše řada lidí spojeno především s Eclipse IDE. Je však třeba říci, že primárně pojem Eclipse označuje open-source komunitu lidí podílejících se na tvorbě nástrojů ulehčujících programátorovi práci.

Pod výše zmíněným Eclipse IDE stojí generická nástrojová platforma, která mimo jiné podporuje tvorbu nástrojů zaměřených na celou škálu programovacích jazyků, a to dokonce na různých operačních systémech.

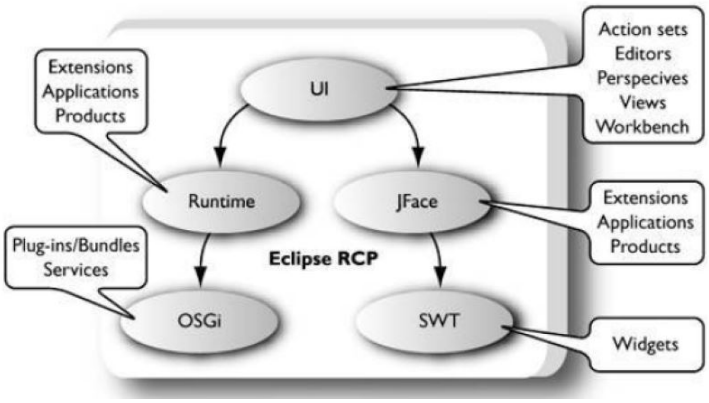
Pod nástrojovou platformou se nachází Eclipse RCP. Jedná se o generickou platformu, která se používá pro tvorbu flexibilních a rozšiřitelných komponentových aplikací. Zmiňované Eclipse IDE je jednou takovou aplikací postavenou právě nad Eclipse RCP.

[1] Jako *IDE* neboli *Integrated Developement Environment* se v informatice označuje software, který programátorovi usnadňuje práci během vývoje. Takový software bývá zaměřen na jeden konkrétní programovací jazyk, ale nemusí to být pravidlem.

[2] Pro spojení *user experience* není v češtině odpovídající doslovný překlad, který by vystihoval stejnou podstatu věci. Doslovným přeložením do „uživatelská zkušenost“ si přesto můžeme udělat hrubou představu o tom, co toto spojení znamená. Jedná se o rozšíření použitelnosti dané aplikace.

3.2.2 Architektura platformy

Platforma Eclipse RCP je tvořena skupinou pluginů, mezi kterými fungují určité druhy vazeb. Každá ze skupin pluginů obstarává určitý typ funkcionality. Programátor je tak například odstíněn od toho, jak zajistit nativní vzhled na různých operačních systémech či jakým způsobem spravovat životní cyklus pluginů. Na obrázku 3.3 jsou tyto skupiny pluginů zobrazeny. Dále jsou pak jednotlivé skupiny pluginů ve stručnosti popsány.



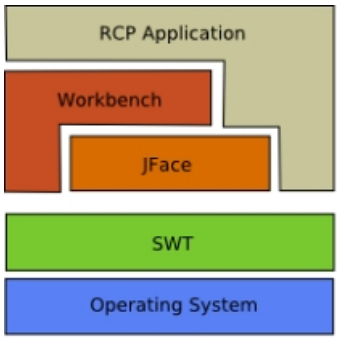
Obrázek 3.3: Skupiny pluginů platformy Eclipse RCP

OSGi skupina

Významnou část výstavby Eclipse aplikací představuje správa závislostí mezi pluginy. Jednotlivé pluginy obsahují explicitně seznamy vyžadovaných pluginů či funkcí, které musí být pro jejich správný běh přítomny. Starostí OSGi je tyto nutné závislosti vyřešit a vytvořit dynamické vazby mezi pluginy. Současně OSGi skupina zajišťuje dynamické spouštění pluginů – plugin je spuštěn teprve ve chvíli, kdy je ho opravdu třeba.

Runtime skupina

Jednou z významných úloh této skupiny je nalezení a spuštění tzv. *application*. Pokud bychom hledali analogii s tradičním programováním v Javě, pak by šlo o spouštěcí metodu `main`. Application je vlastně vstupním bodem komponentové aplikace.



Obrázek 3.4: UI skupina a její hierarchie

Runtime skupina se také stará o správu možných rozšíření jednotlivých pluginů. Mechanismus možných rozšíření je zajištěn pomocí tzv. *extension registry*. Znalost této problematiky byla pro vypracování bakalářské práce klíčová, a tak se o tomto mechanismu můžeme dočíst dále v samostatném pododdíle 3.2.3.

UI skupina

UI (**U**ser **I**nterface) skupina se sestává z několika komponent, které slouží jako základní stavební prvky pro tvorbu uživatelského rozhraní. Hierarchie těchto prvků je znázorněna výše na obrázku 3.4.

SWT skupina

SWT je standartní komponentovou knihovnou pro tvorbu uživatelského rozhraní. Na rozdíl od klasické AWT/Swing knihovny je však knihovnou nízkoúrovňovou. Její použití tak zajišťuje nativní vzhled na daném operačním systému a často i rychlejší odezvu.[1]

[1] O vykreslování grafických prvků se v AWT/Swing stará JVM, což je vždy spojeno s určitou režií. V SWT tato „mezivrstva“ odpadá a grafické prvky jsou poskytovány přímo operačním systémem.

JFace skupina

JFace si můžeme představit jako jakýsi druh rozšíření pro SWT. V knize [5] popisuje UI team JFace takto:„JFace je UI sada nástrojů s třídami, které nabízejí řešení mnoha častých UI programátorských úkolů.“

Některé příklady užití, které dokáží vývojáři značně ulehčit práci, jsou uvedeny dále.

- **Viewer.** Rodina těchto tříd má na starosti jednotvárné úkoly, jako je plnění, třídění, filtrování a aktualizace *widgetů*.[2]
- **Actions.** Umožňují uživateli definovat vlastní chování, které je pak možné přiřadit některé grafické komponentě.
- **Dialogs, Wizards.** Dialogy a průvodci umožňují vytvořit interaktivní aplikaci.

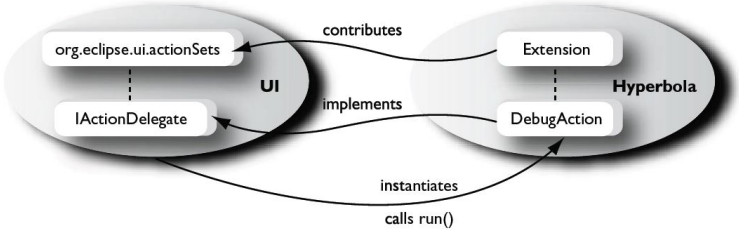
3.2.3 Dodání funkcionality

Dodání funkcionality do komponentově orientovaných aplikací vyvíjených nad platformou Eclipse RCP se provádí pomocí mechanismu zvaného *extension registry*. Tyto registry umožňují dodatečně deklarovat vztahy mezi pluginy. Plugin může zpřístupnit jeho vlastní rozšíření či konfiguraci pomocí tzv. *extension point*. Takový plugin pak vlastně říká: „Dodej mi následující informaci a já udělám ...“ Ostatní pluginy pak mohou přispět vyžadovanou informací ve formě tzv. *extension* vztahujícímu se ke konkrétnímu *extension-point*.

Dovolím si zde uvést příklad z knihy [5]. Plugin *UI* má deklarovaný extension point *actionSets*. Tento extension point umožňuje jiným pluginům přidávat do menu další položky společně s akcí, která se po vybrání položky aktivuje. Má-li některý z pluginů zájem přispět do menu, musí dodržet kontrakt extension point a poskytnout ID, označení, ikonu a třídu, která implementuje `IActionDelegate`. Plugin *UI* pak použije tyto dodané objekty a zobrazí je v podobě další položky menu. Pokud uživatel klikne na tuto

[2] *Widget* v tomto kontextu označuje některý z grafických prvků knihovny SWT. Příkladem takového widgetu může být tlačítko, menu či seznam.

položku, je zavolána metoda `run()` z dodané třídy implementující `IAction-Delegate`. Vše je zachyceno na následujícím obrázku 3.5.



Obrázek 3.5: Dodání funkcionality do ukázkové aplikace, převzato z [5]

Každý plugin vyvíjený nad platformou Eclipse RCP obsahuje XML soubor *plugin.xml*. Tento soubor popisuje, jakým způsobem daný plugin rozšiřuje platformu, jak je implementována funkcionalita a potenciálně jaké extension-point nabízí, či jaké extension dodává.

Vrátíme-li se k našemu příkladu, následující řádka zachycuje deklaraci extension point pro plugin *UI* uvnitř souboru *plugin.xml*.

```
1 <extension-point id="actionSets" name="Action␣Sets"/>
```

Naproti tomu úryvek z *plugin.xml* souboru, který přísluší pluginu *Hyperbola* a definuje konkrétní extension, vypadá následovně:

```
1 <extension point="org.eclipse.ui.actionSets">
2         <actionSet id="org.eclipsercp.hyperbola.debugActionSet">
3                 <action
4                         id="org.eclipsercp.hyperbola.debug"
5                         class="org.eclipsercp.hyperbola.DebugAction"
6                         icon="icons/debug.gif"
7                         label="Debug␣Chats"/>
8         </actionSet>
9 </extension>
```

Dohromady tyto dva protipóly extension-point/extension představují velice mocný mechanismus, který umožňuje deklarativně rozšiřovat komponentovou aplikaci. Vše je tzv. „on demand“, kód se tedy načítá teprve až ve chvíli, kdy je opravdu třeba.

4 Nástroje ComAV a CRCE

Kapitola seznamuje čtenáře s nástroji ComAV a CRCE. Jejich znalost byla pro účely vypracování této práce naprosto nezbytná.

4.1 CRCE

Následující oddíl popisuje nástroj CRCE. V oddíle jsou použity znalosti z článku [6] a z domovských internetových stránek nástroje CRCE [7].

4.1.1 Představení

CBSE obvykle používá úložiště k uskladnění předpřipravených komponent určených pro budoucí použití. Libovolná komponentová aplikace, která má později zájem o integraci některé z uskladněných komponent, využije základních služeb úložiště[1] a komponentu si jednoduše stáhne z úložiště. Správné začlenění komponenty do aplikace – tedy ověření kompatibility –, je pak jeden z úkolů na straně klienta – vývojáře. Pokud se během validace integrované komponenty ukáže, že komponenta svými parametry nevyhovuje požadavkům, je pak opět starostí klienta si vyžádat jinou komponentu z úložiště a opět se pokusit o její začlenění.

Pokud budeme uvažovat o systému, který je inkrementálně skládán z komponent a není nikterak omezen – jeho systémové zdroje jsou neomezené –, pak můžeme říci, že ověření kompatibility během vývoje není výraznou překážkou. Je třeba si však uvědomit, že například ve světě mobilních zařízení, jejichž zastoupení je stále větší a větší, se začíná osvědčovat použití komponentově zaměřených frameworků. Tento trend se objevuje jednoduše z toho důvodu, že je z hlediska vývoje a údržby softwaru velice výhodný.

Je patrné, že konkrétně pro mobilní zařízení může být tento způsob ověření kompatibility až na cílovém zařízení velice problematický. Při limitující rychlosti a nezanedbatelné ceně internetového připojení je přenos redundant-

[1] Mezi tyto základní služby patří schopnost komponenty uskladňovat, komponenty vyhledávat a komponenty stahovat z úložiště.

ních dat (komponenty, které nevyhovují svými parametry) kritický. A to neuvažujeme nad systémovými zdroji, které budou ve světe mobilních zařízení také často překážkou.

Z toho důvodu bylo na Katedře informatiky a výpočetní techniky Západočeské univerzity vyvinuto CRCE úložiště. Jedná se o rozšiřitelné úložiště založené na *OBR* (**O**SGi **B**undle **R**epository). Umožňuje uskladňovat jednotlivé komponenty a především ověřovat jejich kompatibilitu. Ověření kompatibility je tedy záležitostí již na straně distributora, nikoliv vývojáře.

4.1.2 REST API

Nejprve zavedeme pojem *REST*. Ve volnějším slova smyslu je REST popisován jednoduchým rozhraním, které přenáší doménově specifická data pomocí HTTP protokolu. Vyžádání dat ze serveru si tak například můžeme představit jako obyčejný HTTP GET dotaz nad serverem.

Součástí CRCE úložiště je REST API, pomocí kterého je možné s úložištěm programově komunikovat předem definovaným způsobem.

Na ukázku je přiloženo několik možných okomentovaných dotazů nad serverem, kde je spuštěna instance úložiště. Dotazem rozumíme pouze sufix, který je nutný umístit za následující adresu: *server URL + "/rest"*.

- **/bundle/id** – stažení bundlu s příslušným identifikátorem ID (odpovídá symbolickému jménu bundlu)
- **/metadata** – získá metadata[2] popisující uskladněné komponenty
- **/metadata/id** – získá metadata popisující komponentu s příslušným ID
- **/metadata/?filter=&core** – získá metadata popisující uskladněné komponenty (obsahuje pouze základní informace)

Poslední dva typy dotazů byly v práci použity a budou ještě zmíněny.

[2]Pod pojmem *metadata* rozumíme strukturovaná data popisující jiná data. Příkladem takových metadat mohou být například atributy souboru v počítači, jako je: *datum vytvoření*, *velikost souboru* nebo *autor*.

4.1.3 Struktura metadat

CRCE REST API poskytuje výsledky ve formě metadat. Tato metadata popisují uskladněné komponenty v úložišti (viz výše). Cílem tohoto pododdílu je na krátkém vzorku metadat přiblížit čtenáři její strukturu.

Struktura metadat se může mírně odlišovat od aktuální podoby, neboť se na CRCE stále pracuje. Nicméně, princip zápisu informace by měl být stejný. Vzorek je inspirován ukázkou pocházející z [7] a částečně popisuje komponentu *HotelRes* v kontextu ostatních komponent z obrázku 2.1.

```
1  <resource crce:id='hotel-res-2.0.0' id="334232">
2      <capability namespace='crce.identity' uuid='...'>
3        <attribute name="name" value="hotel-res-2.0.0" >
4        <attribute name="crce.type" value="osgi,jar" type="list"/>
5        <attribute name="provider" value="cz.zcu.kiv"/>
6        <attribute name="version.original" type="Version" value="2.0.0
             "/>
7        <attribute name="crce.categories" value="osgi" type="list"/>
8        <attribute name="crce.status" value="stored"/>
9      </capability>
10
11     <capability namespace='osgi.identity'>
12       <attribute name="name" value="hotel-res"/>
13       <attribute name="version" type="Version" value="2.0.0"/>
14     </capability>
15
16     <capability namespace='osgi.wiring.package'>
17       <attribute name='name' value='cz.zcu.kiv.exa.hotelres'/>
18       <attribute name='version' type='Version' value='2.0.0'/>
19      </capability>
20
21     <requirement namespace='osgi.wiring.package'>
22       <attribute name='name' value='cz.zcu.kiv.exa.loyalityprogram'/
             >
23       <attribute name='version' type='Version' value='1.8.0'/>
24     </requirement>
25   ...
```

Obrázek 4.1: Ukázka metadat z CRCE

Na první řádce začíná XML soubor elementem `resource`. Ten odpovídá v již zmiňované komponentové hierarchii (3.1.2) komponentě. Na řádce číslo 2 začíná element `capability` s atributem `namespace`. Hodnota atributu `namespace` blíže specifikuje povahu `capability`. Capability ve volném překladu říká, že je něco poskytováno. Konkrétně zde v tomto případě to jsou základní

informace týkající se objektu v úložišti (namespace je *crce.identity*). Na řádce 7 je pak řečeno, do jaké kategorie objekt v úložišti spadá. Díky této informaci jsme dále schopni reagovat na element `capability` na řádce 11. Uvnitř elementu se pohybujeme na úrovni OSGi bundlu a z atributů `name` a `version` můžeme vyčíst požadované jméno bundlu a jeho verzi.

Další element `capability` je umístěný na řádkách 16–19. Hodnotou atributu `namespace` je *osgi.wiring.package*. Element tedy v tomto kontextu představuje vlastnost bundlu *export-packages* – jedná se o balíček bundlu, který je poskytován ostatním bundlům.

Obdobně to platí pro další element `requirement` na řádkách 21–24, jehož namespace je opět *osgi.wiring.package*. Společně tato kombinace (název elementu a hodnota namespace) vytváří analogii v OSGi v podobě *import-packages* – požadované balíčky.

4.2 ComAV

Oddíl čtenáře seznamuje s nástrojem ComAV, který je středem zájmu této bakalářské práce. Je zde nastíněna architektura nástroje a je ukázáno, jakým způsobem je možné tento nástroj rozšířit.

4.2.1 Představení

Nástroj ComAV je komponentovou aplikací postavenou nad platformou Eclipse RCP. Schopností tohoto nástroje je načítat informace o komponentách určitého komponentového modelu. Načtené komponenty jsou pak vnitřně reprezentovány pomocí ENT meta-model objektů, které mohou být dále interaktivně vizualizovány. Právě vizualizace je hlavním účelem nástroje a měla by uživateli pomoci efektivním a srozumitelným způsobem k pochopení interních závislostí zkoumané komponentové aplikace.

Je nutné dodat, že načítání komponent a jejich vizualizaci zajišťují speciální typy jednotek – pluginy. Tyto pluginy je možné do nástroje přidávat a rozšiřovat tak jeho funkcionalitu. Nástroj tedy funguje jako rozšiřitelná platforma a pro svůj smysluplný běh potřebuje alespoň jeden plugin pro načítání a jeden plugin pro vizualizaci.

4.2.2 Design

Komponentová aplikace ComAV se skládá ze 3 typů pluginů.

1. **Loader** slouží k načtení informací o komponentách příslušného komponentového modelu do ENT meta-modelu.
2. **Visualizer** umožňuje zobrazovat závislosti mezi načtenými komponentami a poskytuje základní předvolby pro zobrazení.
3. **Core** funguje jako prostředník mezi pluginy typu *Loader* a *Visualizer*, definuje základní kostru celé aplikace.

4.2.3 Způsob rozšíření

Jak již bylo zmíněno v oddíle 3.2, platforma Eclipse RCP nabízí prostředek umožňující rozšiřovat funkcionalitu pluginu pomocí mechanismu extension point/extension. Nástroj ComAV byl navržen s předpokladem, že se bude jeho funkcionalita dále rozšiřovat, a tak obsahuje extension point pro pluginy typu *Loader* a *Visualizer*. Implementací těchto pluginů a dodržením určitých pravidel je pak možné do nástroje dodat potřebnou funkcionalitu.

Protože je účelem této bakalářské práce připravit extension point typu *Loader*, je v následující části detailněji popsán postup jeho přidání do nástroje ComAV.

Extension point pro Loader

Plugin *Core* dokáže uchovávat načtené komponenty v podobě ENT meta-model objektů získaných z příslušného pluginu typu *Loader*. Samotné načítání komponent do ENT meta-model objektů zajišťuje třída `Handler` dědící od abstraktní třídy `CommonLoaderHandler` deklarované uvnitř pluginu *Core*. Instanci této třídy si můžeme představit jako prostředníka mezi pluginy *Core* a *Loader*. Tento prostředník zajistí v rámci pluginu *Loader* načtení vybraných komponent do ENT meta-model objektů. Následně pak předává pluginu *Core* připravený seznam s ENT komponentami (reprezentace komponent v ENT meta-modelu).

Komunikace mezi pluginy typu *Core* a *Loader* je řešena pomocí mechanismu extension-point/extension (viz 3.2.3). Plugin *Loader* přidává do aplikace pomocí extension-point vlastní položku menu pro vytvoření projektu. Na tuto položku menu je navázán příkaz, který je taktéž třeba nadefinovat pomocí extension-point vedle položky menu. Obslužným objektem příkazu je pak třída `Handler` nacházející se uvnitř pluginu *Loader*[3].

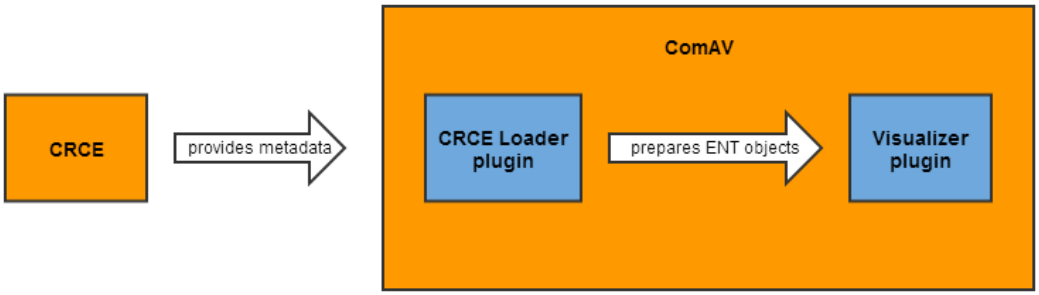
[3]Zmiňovaná třída `CommonLoaderHandler`, od které dědí třída `Handler`, dále dědí od třídy `AbstractHandler` z balíku `org.eclipse.ui.commands`. To umožňuje použití třídy `Handler` jako obslužného objektu pro příkaz.

5 CRCE loader

V úvodním oddíle kapitoly je ve stručnosti řečeno, jaký je záměr bakalářské práce. Další dvojice oddílů pojednává o analýze, návrhu a samotné implementaci pluginu typu loader do nástroje ComAV. A to jak z hlediska logického (1. oddíl), tak z hlediska vizuálního (2. oddíl). Poslední oddíl kapitoly pak objektivně shrnuje dosažené výsledky.

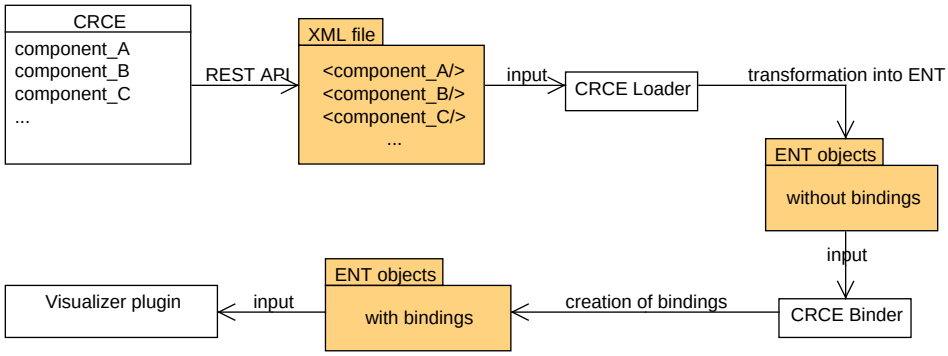
5.1 Specifikace požadavků

Úkolem je připravit plugin typu loader do nástroje ComAV. Plugin bude napojený na CRCE a bude z něj načítat metadata popisující uskladněné komponenty. Metadata bude dále převádět do datových struktur reprezentujících jednotlivé komponenty. S těmito datovými strukturami nástroj ComAV již dokáže pracovat a je schopen je vizualizovat. V zjednodušené formě je tento proces zachycený na následujícím obrázku.



Obrázek 5.1: Význam CRCE loaderu

Poznámka: CRCE úložiště je stále ve vývoji a struktura metadat se stále přetváří. Pro účely vypracování této práce bylo rozhodnuto, že CRCE loader bude spolupracovat s uvolněnou verzí *CRCE 2.0.0 - revize 711*.



Obrázek 5.2: Od úložiště k vizualizaci

5.2 Logická část

Tento oddíl pojednává o logické části pluginu. Logickou částí se rozumí jádro pluginu – tedy funkční část programu –, která je koncovému uživateli skryta.

5.2.1 Analýza

Na základě poznatků z předchozích kapitol jsme již schopni objektivně uvažovat nad problémem a hledat jeho ideální řešení.

Nejprve bude nutné nějakým způsobem získat metadata z CRCE. Jak již bylo zmíněno v pododdíle kapitoly pojednávající o CRCE (4.1.2), pro vyžádání informací o uskladněných komponentách existuje prostředek v podobě CRCE REST API. Výsledkem dotazu z REST API je odpověď ve formátu XML. Tuto odpověď je třeba rozparsovat[1] a převést do ENT objektů. Mezi těmi je pak nutné dynamicky vytvořit závislosti. ComAV je pak schopen jednotlivé komponenty v podobě ENT objektů společně s těmito vazbami vizualizovat. Vše je znázorněno na obrázku 5.2.

[1]Slovo *rozparsovat* pochází z anglického slovesa *to parse* a jeho český překlad je „provést rozbor“. V oblasti informačních technologiích se s tímto slovem často setkáváme v souvislosti s řetězci. Označuje proces získání určité informace z řetězce.

DOM vs SAX

Otázkou číslo 1 bylo, jakou technologii pro parsování XML souboru použít. Mezi nejpopulárnější XML technologie patří DOM a SAX. Každý z těchto přístupů má své výhody a nevýhody.

DOM na svém začátku projde celý XML soubor a uloží ho do paměti ve formě dynamického stromu. Tento strom lze pak libovolně procházet a dodatečně editovat. Protože DOM ukládá celý XML soubor do paměti v podobě dynamického stromu, není příliš vhodný na rozsáhlé soubory (může být paměťově velice náročné).

Naproti tomu technologie SAX používá naprosto odlišný přístup ke zpracování XML souboru. XML soubor je sekvenčně procházen pomocí tzv. *parseru*. K tomuto parseru přísluší tzv. *content handler*, ve kterém jsou definovány události jako: začátek elementu, konec elementu a text uvnitř elementu. Na programátorovi pak je, aby se o celou logiku procházení XML souboru postaral dodatečnou implementací těchto událostí. Do paměti se pak ukládá pouze to, co programátor uzná za vhodné. SAX je tedy vhodnější na rozsáhlé soubory či na proudové zpracování dat.

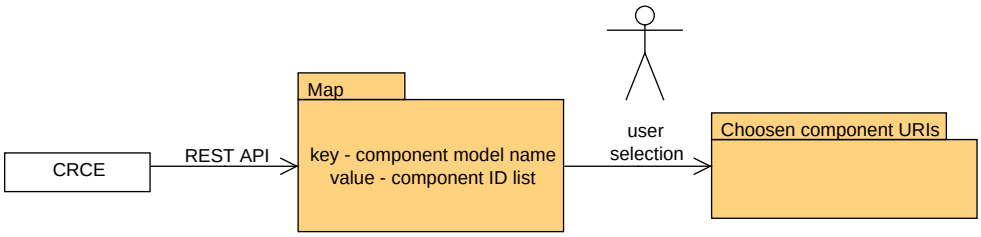
Charakter metadat vrácených z CRCE nelze predikovat – data mohou být rozsáhlá. Zároveň můžeme předpokládat, že plugin bude napojen na vzdálené úložiště a bude tak pracovat s proudem dat. Data nebude třeba upravovat, pouze číst. Po zvážení všech těchto okolností jsem se rozhodl použít pro parsování XML technologii SAX.

Dvoufázové načítání

Dále bylo nutné dobře promyslet, jak bude načítání metadat z CRCE probíhat. Možným způsobem, jak řešit načítání komponent z CRCE, by bylo zpracovávat metadata pro všechny uskladněné komponenty v CRCE a vytvářet jejich ENT reprezentaci. Uživatel by si poté vybral pouze ty komponenty, které chce vizualizovat. Pro větší množství komponent by však takový přístup byl velice nešikovný. Načítaly by se i komponenty, které by posléze nebyly vybrány k vizualizaci, a docházelo by tak pravděpodobně k výraznému zpomalování celého nástroje.

Proto se rozhodlo, že načítání komponent bude dvoufázový proces.

První fáze V první fázi dojde k přednačtení obsahu CRCE ve formě mapy identifikátorů uskladněných komponent. Klíčem v mapě je název komponentového modelu a hodnotou je seznam s identifikátory komponent příslušného komponentového modelu. Na uživateli pak je, aby vybral ty komponenty, které chce vizualizovat (viz obrázek 5.3). Po výběru následuje druhá fáze.



Obrázek 5.3: Přednačtení mapy identifikátorů a výběr komponent k zobrazení

Druhá fáze V této fázi jsou již známy URI adresy požadovaných komponent. Může tedy dojít k načtení metadat požadovaných komponent z CRCE a k převodu do objektů ENT meta-modelu. Tento scénář byl již koneckonců zachycený v úvodu analýzy na obrázku 5.2.

V obou fázích bude třeba přistupovat k CRCE pomocí REST API a zpracovávat XML soubor. V první fázi to však bude pouze okleštěná verze obsahující základní informace o komponentách. V druhé fázi již půjde o plnohodnotná metadata popisující detailně jednotlivé komponenty. Nabízí se zde tedy deklarace třídy, která bude umožňovat zpracování XML pro oba případy a které se pouze pomocí nějakého přepínače určí režim běhu. Teoreticky by totiž třída zpracovávající plnohodnotná metadata měla zvládat i zpracování základních metadat, která jsou podmnožinou plnohodnotných. Rozhodl jsem se přesto použít dvou oddělených tříd pro tuto funkci. Orientace v takovém kódu bude jednodušší a bude jasné, pro jaký typ metadat je třída určena.

Použití existujícího

Posledním významnějším problémem ke zvážení bylo vytváření vazeb mezi ENT komponentami. CRCE umožňuje momentálně uchovávat komponenty komponentových modelů EJB3, OSGi či SOFA2. Pro tyto komponentové modely již existují implementace jiných loaderů, které již mají mechanismus

dynamického utváření vazeb mezi ENT komponentami naimplementovaný. Bylo by tedy hloupé vytvářet znovu kód se stejnou funkcionalitou. Výhodnější proto bude tuto funkční část v rámci každého typu loaderu oddělit a vytvořit pro každý zmíněný typ loaderu vlastní třídu typu *binder*[2], která bude tuto funkcionalitu přejímat. CRCE loader pak bude moci pro konkrétní komponentový model vytvořit konkrétní *binder*, který se již o utvoření vazeb postará.

5.2.2 Design

V této části je popsáno vnitřní členění pluginu na jednotlivé balíčky. Ke každému balíčku je uveden výčet deklarovaných tříd. Ke všem třídám, které souvisí s logickou částí pluginu, je uveden stručný popis. Poslední balíček spadá do vizuální části. Jeho třídy jsou pouze zmíněny a budou dále rozebrány v oddílu této kapitoly zabývající se vizuální části pluginu. O většině tříd se můžeme dovědět více v následující části týkající se implementace.

- **cz.zcu.kiv.comav.loaders.crce** – Jedná se o hlavní balíček pluginu.
  - **CRCEComponentLoader** – Třída slouží k načtení metadat z CRCE a jejich převedení na ENT komponenty. Vnitřně pracuje s XML souborem – používá třídy z následujícího balíčku `.xml`.
  - **CRCEComponentBinder** – Třída slouží jako obecný binder (více viz *Znovupožití existujícího* v pododdíle 5.2.1).
  - **CRCERepoContentLoader** – Třída slouží k získání mapy dostupných komponent z CRCE. Vnitřně funguje velice podobně jako třída `CRCEComponentLoader`, a tak opět využívá třídy z balíčku `.xml`.
  - **CRCEVersion** – Jedná se o převzatou třídu z pluginu `OSGI Loader`. Instance třídy reprezentuje určitou verzi či dokonce rozsah verzí.
  - **Activator** – Tato třída souvisí s životním cyklem pluginu. Obsahuje metody `start()/stop()`, které jsou volány během spuštění/ukončení pluginu.

[2]Slovo *binder* je cizího původu a v překladu znamená něco jako „vázač“. V našem kontextu se jedná o objekt, který zajišťuje utvoření závislostí mezi komponentami – zajistí jejich „svázání“.

- **cz.zcu.kiv.comav.loaders.crce.xml** – Třídy uvnitř tohoto balíčku jsou použity během zpracování XML souboru.
  - **CRCEConstants** – Třída poskytuje konstanty pro práci s XML soubory. Jsou zde například definovaná symbolická jména, která se mohou v XML získaného z CRCE objevit. Dále jsou zde definované názvy konkrétních elementů ENT meta-modelu.
  - **CRCEXmlToEntmmConnector** – Třída slouží pro propojení mezi názvy, které se objevují v XML souboru, a názvy, které jsou definovány v *.entmm* souboru.
  - **CRCEXmlContentHandler** – Tato třída slouží jako *content handler* pro SAX parser. Instanci třídy bychom mohli připodobnit k sekvenčnímu automatu, který reaguje na události spojené s průchodem XML souboru a připravuje ENT komponenty.
  - **CRCEXmlSummaryHandler** – Princip funkcionality této třídy je stejný jako v předchozím případě. Odlišnost této třídy od předchozí spočívá ve skutečnosti, že se v této třídě nevytváří ENT komponenty, ale pouze jednoduchá mapa řetězců. Klíčem v mapě je název komponentového modelu a hodnotou seznam s identifikátory dostupných komponent v CRCE.
  - **CRCEXmlErrorHandler** – Třída funguje jako obslužný objekt pro výjimky způsobené procházením XML souboru.
- **cz.zcu.kiv.comav.loaders.crce.wizard** – Tento balíček definuje prostředky, které umožňují interakci s uživatelem a které zajišťují integraci pluginu CRCE loader do nástroje ComAV. Podrobněji se tomuto balíčku věnuje následující oddíl 5.3.
  - **Handler**
  - **LoaderWizard**
  - **ProjectNamePage**
  - **ComponentChooserPage**

5.2.3 Implementace

V tomto pododdíle jsou postupně popsány jednotlivé kroky, které bylo nutné učinit během implementace logického jádra pluginu. Nejprve je uveden jejich výčet a dále jsou pak jednotlivé kroky podrobněji popsány.

- **Získání mapy identifikátorů**
- **Zpracování metadat s identifikátory**
- **Interakce s uživatelem**
- **Získání mapy komponent**
- **Zpracování metadat s komponentami**
- **Namapování XML na ENTMM**
- **Vytvoření vazeb mezi komponentami**
- **Zobrazení načtených komponent a jejich vazeb**

Získání mapy identifikátorů

Prvním úkolem bylo připravit jednotku, která se postará o získání mapy uskladněných komponent v CRCE. Tuto funkci obstarává třída `CRCERepoContentLoader`.

Instance třídy potřebuje znát ke svému běhu cestu k úložišti a zároveň musí vědět, které komponentové modely jsou podporovány. Tyto údaje jsou proto použity přímo jako vstupní parametry v konstruktoru třídy. Dalším významným parametrem je statická konstanta `URL_SUMMARY_SF`. Jedná se o sufix, který se používá k vytvoření REST dotazu. Pomocí tohoto dotazu se z CRCE serveru získají základní metadata o uskladněných komponentách (*URL serveru + "/rest/metadata?filter=&core"*).

Pro načtení mapy identifikátorů komponent deklaruje třída metodu zachycenou na obrázku 5.4 na další straně. Protože se může snadno stát, že CRCE s danou URL nebude existovat, je zde použit mechanismus vyhození výjimky. Na jiném místě v programu je pak nutné po zavolání metody tuto výjimku ošetřit. Obsah metody je poměrně samovysvětlující. Nad URL se otevře proud dat. Parseru se pak tento proud dat předá jako zdroj v parametru metody `parse()` na řádce 15. Důležité jsou řádky 13 a 14. Na řádce 13 se parseru nastaví obslužný objekt, který reaguje na chyby způsobené během parsování XML souboru. Na řádce 14 se pak nastaví obslužný objekt (instance třídy `CRCEXmlSummaryHandler`), který spravuje samotné zpracování XML.

```
1       /**
2        * Loads component id map from repository.
3        * @throws IOException exception probably caused by wrong url
4        */
5       public void loadComponentMap() throws IOException {
6           URL repoUrl = new URL(repoPath + URL_SUMMARY_SF);
7           InputStream is = repoUrl.openStream();
8
9           try{
10              SAXParserFactory spf = SAXParserFactory.newInstance();
11              SAXParser sp = spf.newSAXParser();
12              XMLReader parser = sp.getXMLReader();
13              parser.setErrorHandler(new CRCEXmlErrorHandler());
14              parser.setContentHandler(summaryHandler);
15              parser.parse(new InputSource(is));
16          }
17          catch(SAXException se){
18              se.printStackTrace();
19          }
20          catch(ParserConfigurationException ce){
21              ce.printStackTrace();
22          }
23      }
```

Obrázek 5.4: Metoda pro načtení mapy identifikátorů

Zpracování metadat s identifikátory

Obsluhu pro parsování XML souboru poskytuje třída `CRCEXmlSummaryHandler`. Třída dědí od třídy `DefaultHandler` z balíčku `org.xml.sax.helpers` a překrývá metody `startElement` a `endElement`. Uvnitř těchto metod se na základě jména elementu, do kterého se vstupuje, či ze kterého se vystupuje, nastavují příslušné pomocné proměnné typu `boolean` začínající prefixem *inside*. Tyto proměnné tedy slouží během procházení ke specifikaci aktuálního umístění kurzoru v XML souboru a uchovávají tak aktuální stav procházení – třída reprezentuje jakýsi víceúrovňový stavový automat.

Během procházení dochází k postupnému zanořování a vynořování ve struktuře XML. Umístění v elementu si můžeme představit jako jeden stav. Jedná-li se o několikanásobné zanoření, můžeme si pak pro každý element na cestě stromem vzhůru představit vlastní stav. Aktuální umístění v souboru je pak popsáno uspořádanou n-ticí takových stavů.

Vrátíme-li se k obrázku 4.1 zachycujícímu XML metadata vrácená z CRCE, můžeme si na tomto příkladu ukázat konkrétní aplikaci stavů – obrázek 5.5 na další straně. Označíme-li stavy na úrovni 0 velkými písmeny, pak bude existovat jediný stav pro element `resource` – *A*. Stavy úrovně 1 označíme čísly. Tedy element `capability` označíme jako *1* a `requirement` jako

2. Chceme-li pak například zpracovat hodnoty atributů pouze pro element `requirement`, stačí vždy ověřit, zda je aktuální konstelace stavů *[A2]*.

```
1  <resource crce:id='hotel-res-2.0.0' id="334232">
2  A  <capability namespace='crce.identity' uuid='...'>                    1
3       <attribute name="name" value="hotel-res-2.0.0" >
4       <attribute name="crce.type" value="osgi,jar" type="list"/>
5       <attribute name="provider" value="cz.zcu.kiv"/>
6       <attribute name="version.original" type="Version" value="2.0.0
            "/>
7       <attribute name="crce.categories" value="osgi" type="list"/>
8       <attribute name="crce.status" value="stored"/>
9     </capability>
10
11    <capability namespace='osgi.identity'>                              1
12      <attribute name="name" value="hotel-res"/>
13      <attribute name="version" type="Version" value="2.0.0"/>
14    </capability>
15
16    <capability namespace='osgi.wiring.package'>                        1
17      <attribute name='name' value='cz.zcu.kiv.exa.hotelres'/>
18      <attribute name='version' type='Version' value='2.0.0'/>
19     </capability>
20
21    <requirement namespace='osgi.wiring.package'>
22      <attribute name='name' value='cz.zcu.kiv.exa.loyalityprogram'/
            >
23      <attribute name='version' type='Version' value='1.8.0'/>        2
24    </requirement>
25  ...
```

Obrázek 5.5: Stavovost XML souboru

Interakce s uživatelem

Vrat'me se nyní k obrázku 5.3. Načtení mapy komponent již máme vyřešené. Další bod, kterým je interakce s uživatelem a výběr komponent k zobrazení, je záležitostí vizuální části. Budeme tedy pouze předpokládat, že výběr komponentového modelu a jeho příslušných komponent z CRCE již proběhl a máme dostupný seznam komponent, které budeme dále načítat. Dalším krokem tedy bude připravit jednotku, která si vyžádá metadata příslušných komponent a převede je do ENT. Tuto funkci má na starosti třída `CRCEComponentLoader`.

Získání mapy komponent

Třída `CRCEComponentLoader` zajišťuje načtení komponent, jejich převedení do ENT a utvoření vazeb mezi ENT komponentami. Každé zmíněné funkci bude věnován podrobnější rozbor. Nejprve se ale podíváme, jak je to s dědičností třídy.

Z diskuse s vedoucím práce vyplynulo, že se do budoucna předpokládá případ, kdy bude v CRCE loaderu třeba zobrazovat společně komponenty rozdílných komponentových modelů. Proto byla navržena třída `GeneralizedComponentLoader`, která rozšiřuje klasický `ComponentLoader` o možnost uchovávat komponenty v mapě (klíčem je název komponentového modelu a hodnotou seznam příslušných komponent)[3] namísto jednoduchého seznamu. S tím souvisí i fakt, že třída uchovává mimo jiné i mapu objektů typu `ComponentModel`. Tyto objekty slouží k popisu komponenty v ENT meta-modelu pro daný komponentový model.

Třída `CRCEComponentLoader` dědí od třídy `GeneralizedComponentLoader` a překrývá metodu `load(URI[] components)`. Na vstupu této metody jsou URI adresy jednotlivých komponent v CRCE, které uživatel vybral k zobrazení. Uvnitř metody je použitý stejný postup jako tomu bylo u `CRCERepoContentLoader`. Opět se vytvoří instance parseru a nastaví se mu příslušné obslužné instance handlerů pro chyby v XML a pro jeho vlastní zpracování (instance `CRCEXmlContentHandler` – více dále). Poté se pomocí konstrukce `foreach` projdou URI adresy komponent a nad každou adresou se otevře proud dat, který se následně předá jako parametr parseru. Výsledkem zpracování všech URI adres je mapa s komponentami v ENT reprezentaci. Nakonec se vytvoří instance `CRCEComponentBinder`, pomocí které se utvoří vazby mezi komponentami.

Zpracování metadat s komponentami

O zpracování XML souboru s metadaty popisujících uskladněné a vybrané komponenty se stará třída `CRCEXmlContentHandler`. Princip funkcionality této třídy je veskrze podobný výše zmíněnému případu s třídou `CRCEXml-`

[3]Existence mapy pro uložení komponent různých komponentových modelů je momentálně nadbytečná, protože CRCE v současné době plně podporuje jen OSGi. Nicméně, do budoucna se počítá s rozšířením podpory CRCE o další komponentové modely. Poté bude dávat použití mapy pro uložení komponent větší smysl.

`SummaryHandler`. Přeci jen se ale najdou nějaké odlišnosti vyplývající z povahy problému, který je poněkud složitější. Úkolem tohoto handleru již není pouhé vytažení identifikátorů komponent, ale samotné utváření komponent v podobě ENT objektů.

Nejprve bych rád řekl něco více o atributech této třídy. Atributy `tmpComponent`, `tmpTrait` a `tmpElement` slouží k uchování aktuálně zpracovávaného ENT objektu. O tom, co to je element či trait, se můžete dočíst více v pododdíle 3.1.2. Dalšími významnými atributy jsou atributy `componentMap`, `factory` a `supportedModels`. Tyto atributy pouze uchovávají odkaz na atributy třídy `CRCEComponentLoader`, a tak poskytují přímý přístup k objektům. Další skupinou atributů jsou atributy s prefixem *inside*. Smysl užití těchto logických proměnných je stejný jako u třídy `CRCEXmlSummaryHandler`. Nakonec atributy `componentType`, `activeNamespace`, `activeModel` a `xmlConnector`. Proměnná `componentType` je typu `String` a určuje, jakému komponentovému modelu přísluší přijatá komponenta v podobě metadat. Proměnná `activeNamespace` je taktéž typu `String` a používá se k uchování hodnoty atributu `namespace` aktuálního elementu (o jeho významu bude dále řeč). Atribut `activeModel` je typu `ComponentModel` a poskytuje definice pro jednotlivé ENT objekty (trait, tag či celá komponenta). Posledním atributem je `xmlConnector`, který je typu `CRCEXmlToEntmmConnector` a kterému je věnován následující samostatný odstavec.

Namapování XML na ENTMM

Instanci třídy `CRCEXmlToEntmmConnector` si můžeme představit jako spojku mezi světy XML a ENT. Přesněji řečeno, tato třída poskytuje namapování určité hodnoty atributu elementu (případně v kombinaci s názvem elementu) v XML na konkrétní název traitu či tagu v ENT meta-modelu popsaném v souboru s příponou *.entmm*.

Pro ilustraci je přiložen příklad, který by měl použití instance této třídy osvětlit. Jedná se o výtažek z ukázkových metadat ze stránek [7]. V tomto příkladu bude zároveň vysvětlen význam atributu `activeNamespace` ve třídě `CRCEXmlContentHandler`.

```
<capability namespace='osgi.wiring.package'>
    <attribute name='name' value='cz.zcu.kiv.obcc.parking.gate.stats
        ' />
    <attribute name='version' type='Version' value='2.0.0' />
    <directive name='uses' value='cz.zcu.kiv.obcc.parking.gate.base'
         />
</capability>
```

Obrázek 5.6: Ukázka elementu *capability*

Ve chvíli, kdy se handler dostane ve zpracování XML na úroveň elementu `capability`, je zavolána metoda `startElement`. Jedním ze vstupních parametrů této metody jsou atributy elementu. Označíme-li tento vstupní parametr jako `atts`, pak se k hodnotě konkrétního atributu dostaneme pomocí konstrukce `atts.getValue("nazev_atributu")`. Hodnotu atributu `namespace` tedy získáme pomocí konstrukce `atts.getValue("namespace")`. Takto získanou hodnotu ("osgi.wiring.package") přiřadíme do atributu `activeNamespace`.

Protože se nacházíme uvnitř elementu **`capability`**, můžeme zavolat metodu `String getTraitNameForNamespace`**`Cap`**`(String namespace)` nad instancí `CRCEXmlToEntmmConnector`. Hodnota parametru `namespace` je "osgi.wiring.package". Návratovou hodnotou metody je pak název konkrétního traitu v ENT meta-modelu. V tomto případě by návratové hodnotě odpovídal řetězec "export_packages". Obdobně v případě elementu **`requirement`** by se volala metoda `String getTraitNameForNamespace`**`Req`**`(String namespace)` a návratovou hodnotou by byl pro tento konkrétní případ řetězec "import_packages". Název traitu vrácený metodou `getTraitNameForNamespace`**`Cap`**`/`**`Req`**`()` se následně použije pro získání definice z ENT meta-modelu.

Vytvoření vazeb mezi komponentami

Potom, co se ve třídě `CRCEComponentLoader` vytvoří ENT reprezentace stažených komponent z CRCE, je nutné vytvořit dynamické závislosti mezi komponentami. O těchto vazbách se více zmiňuje odstavec *Aplikační úroveň* v pododdílu 3.1.2.

Jak již zaznělo v odstavci *Znovupoužití existujícího* v pododdílu 5.2.1, funkcionalita pro utvoření vazeb již byla dostupná. Bylo ji jen třeba napříč loadery různých komponentových modelů správně identifikovat, separovat a

převést do samostatných tříd pro každý typ loaderu. Současně bylo nutné pozměnit kód těchto loaderů tak, aby na tuto změnu reagovaly a používaly třídu konkrétního binderu namísto původní metody. V rámci práce byla tedy provedena restrukturalizace kódu této části nástroje ComAV.

Třída `CRCEComponentBinder` těchto existujících specifických binderů využívá. Prochází mapu s komponentami a na základě hodnoty klíče v mapě (odpovídá názvu komponentového modelu) vytváří odpovídající instanci binderu (`OSGIComponentBinder, EJB3ComponentBinder, SOFA2ComponentBinder`). Nad konkrétním binderem se pak už jen zavolá metoda pro utvoření vazeb. Vstupním parametrem metody je seznam komponent náležících danému komponentovému modelu. Pokud se objeví komponentový model, pro který neexistuje odpovídající binder, běh programu skončí s výjimkou.

```
1  public void createBindings(){
2      for(Map.Entry<String, List<Component>> me : componentMap.
           entrySet()){
3          String componentType = me.getKey();
4          List<Component> componentList = me.getValue();
5
6          if(componentType.equals(GeneralizedComponentLoader.
               MODEL_OSGI)){
7              OSGiComponentBinder osgiBinder = new OSGiComponentBinder
                   ();
8              osgiBinder.createBindings(componentList);
9          }else if(componentType.equals(GeneralizedComponentLoader.
               MODEL_EJB3)){
10             Ejb3ComponentBinder ejb3Binder = new Ejb3ComponentBinder
                   ();
11             ejb3Binder.createBindings(componentList);
12         }else if(componentType.equals(GeneralizedComponentLoader.
               MODEL_SOFA2)){
13             Sofa2ComponentBinder sofa2Binder = new
                   Sofa2ComponentBinder();
14             sofa2Binder.createBindings(componentList);
15         }else {
16             throw new InvalidParameterException("Component type "+
                   componentType + " not recognized as a supported
                   component type.");
17         }
18     }
19 }
```

Obrázek 5.7: Delegace utváření vazeb na jednotlivé bindery

Zobrazení načtených komponent a jejich vazeb

Posledním krokem je zobrazení ENT komponent a jejich vazeb v grafickém editoru nástroje ComAV. O způsob, jakým se komponenty na plátně zobrazí, se stará již hotový plugin typu *Visualizer*.

5.3 Vizuální část

Tento oddíl se věnuje vizuální části loaderu, kterou představuje průvodce vytvořením nového CRCE projektu. Tato volba je zpřístupněna pomocí položky menu v nástroji ComAV. V porovnání s logickou částí je rozsah tohoto oddílu o poznání menší.

5.3.1 Analýza

Jádro loaderu je připravené. Máme k dispozici veškeré prostředky pro načtení komponent z CRCE do ENT. Nyní je třeba připravit uživatelské rozhraní, které tyto prostředky využije a umožní uživateli vytvořit nový CRCE projekt.

Samotné dodání uživatelského rozhraní do nástroje ComAV uskutečníme pomocí již několikrát zmiňovaného mechanismu extension-point/extension (viz pododdíly 3.2.3 a 4.2.3). Je však třeba promyslet, jakým způsobem bude uživatel s CRCE loaderem vůbec komunikovat. Jinými slovy, je třeba popřemýšlet o struktuře průvodce vytvořením nového CRCE projektu.

První stránka průvodce od uživatele získá název projektu a URL cestu k úložišti. Cesta, kterou uživatel zadá, může být zadána chybně nebo může odkazovat na neexistující CRCE server. Proto je třeba s takovou variantou počítat a adekvátním způsobem na ni reagovat.

Potom, co dojde k potvrzení první stránky průvodce a přechodu na další stránku, dojde k načtení mapy komponent uskladněných v CRCE. Tento proces jsme označili jako *první fázi* načítání z CRCE a více se o něm hovoří v odstavci *Dvoufázové načítání* v pododdíle věnujícímu se analýze logické části (5.2.1).

Druhá stránka průvodce bude již poskytovat možnost výběru komponentového modelu a k němu příslušejících komponent uskladněných v CRCE. Uživatel vybere komponenty k zobrazení a potvrdí výběr. Průvodce se ukončí a dojde k načtení komponent do ENT – *druhá fáze* načítání.

5.3.2 Design

Architektura vizuální části je velice prostá. Obsahuje 4 třídy uzavřené do společného balíčku `cz.zcu.kiv.comav.loaders.crce.wizard`.

Těmito třídami jsou:

- **LoaderWizard** – Instance této třídy představuje průvodce vytvořením nového CRCE projektu.
- **ProjectNamePage** – Instance této třídy představuje první stránku průvodce.
- **ComponentChooserPage** – Instance této třídy představuje druhou stránku průvodce.
- **Handler** – Jedná se o obslužnou třídu příkazu, který je navázán na položku v menu aktivující vytvoření nového CRCE projektu.

5.3.3 Implementace

V tomto pododdíle je krok po kroku popsána implementace vizuální části, kterou představuje průvodce vytvořením CRCE projektu.

Nejprve je uveden výčet těchto kroků:

- **Definice extension**
- **Obslužná třída příkazu**
- **Vytvoření průvodce**
- **První strana průvodce**
- **Druhá strana průvodce**

Následující část se pak jednotlivými kroky zabývá podrobněji.

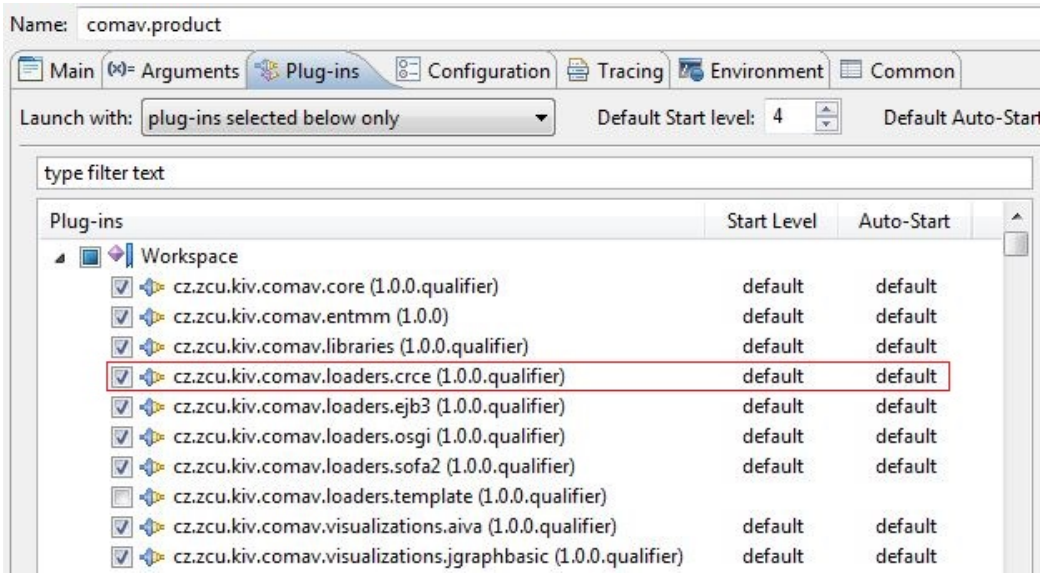
Definice extension

Poznámka: Pod následujícím odstavcem se nachází XML soubor *plugin.xml* příslušející pluginu CRCE loader. K tomuto souboru se vztahují odkazy na řádky z následujícího odstavce.

Prvním krokem bylo v souboru *plugin.xml* CRCE loaderu deklarovat jednotlivé extension. Prvním extension (řádka 2–9) je příkaz *newProject*, na který se jako obslužný objekt navázala třída `Handler`. Dále bylo nutné deklarovat extension pro menu (řádka 10–19), který dodává do menu nástroje ComAV novou položku *CRCE project*. Na tuto položku se pak namapoval zmíněný příkaz *newProject*. Poslední deklarovaný extension (řádka 20–26) víceméně říká nástroji ComAV, že existuje další typ loaderu.

```
1  <plugin>
2     <extension
3           point="org.eclipse.ui.commands">
4        <command
5              defaultHandler="cz.zcu.kiv.comav.loaders.crce.wizard.
                   Handler"
6              id="cz.zcu.kiv.comav.loaders.crce.newProject"
7              name="CRCE Project">
8        </command>
9     </extension>
10    <extension
11          point="org.eclipse.ui.menus">
12       <menuContribution
13             locationURI="menu:new">
14          <command
15                commandId="cz.zcu.kiv.comav.loaders.crce.newProject"
16                label="CRCE project">
17          </command>
18       </menuContribution>
19    </extension>
20    <extension
21          point="cz.zcu.kiv.comav.loaders">
22       <LoaderElement
23             commandId="cz.zcu.kiv.comav.loaders.crce.handler"
24             handler="cz.zcu.kiv.comav.loaders.crce.wizard.Handler">
25       </LoaderElement>
26    </extension>
27 </plugin>
```

V tuto chvíli jsme měli připravené prostředí pro implementaci uživatelského rozhraní pluginu CRCE loader. Bylo již jen třeba nastavit konfigurační soubor *comav.product* kmenového pluginu *ComAV core* nástroje ComAV tak, aby došlo k začlenění pluginu CRCE loader do nástroje ComAV (viz obrázek 5.8).



Obrázek 5.8: Začlenění pluginu CRCE loader do nástroje ComAV

Obslužná třída příkazu

Druhým krokem bylo naimplementovat obslužnou třídu `Handler` příkazu *newProject*. Tento příkaz se zavolá po vybrání položky *CRCE project* v menu.

Třída `Handler` dědí od třídy `CommonLoaderHandler` a překrývá metodu `boolean loaderExecute(ExecutionEvent event)`. Metoda je spuštěna při aktivaci příkazu, na který je třída `Handler` namapována. Uvnitř metody se vytvoří instance průvodce (třída `LoaderWizard`) a průvodce se spustí. Potom, co uživatel v průvodci zadá potřebné informace, je průvodce ukončen a pomocí příslušných getterů se z průvodce vytáhnou informace, jako je název nového projektu, cesta k CRCE serveru, URI adresy vybraných komponent a typ komponentového modelu, kterému vybrané komponenty přísluší. Poté dojde k načtení komponent z CRCE a převedení do ENT pomocí třídy `CRCEComponentLoader`. Pokud vše v metodě proběhne v pořádku, je vrácená logická hodnota *true*. Pokud by byl průvodce ukončen předčasně nebo by během načítání komponent došlo k chybě, je vrácena logická hodnota *false*.

Vytvoření průvodce

Průvodce vytvořením nového CRCE projektu reprezentuje instance třídy `LoaderWizard`. Třída dědí od třídy `Wizard` z balíčku `org.eclipse.jface.wizard`. Rozšiřuje klasický průvodce o některé další atributy, mezi kterými je například název projektu, cesta k CRCE serveru či vybraný komponentový typ (více dále).

Třída překrývá některé metody předka. Metoda `void addPages()` inicializuje během spuštění průvodce jeho jednotlivé stránky, kterými uživatel musí pro dokončení průvodce projít. První stránka průvodce je instance třídy `ProjectNamePage`. Druhá stránka průvodce je instance třídy `ComponentChooserPage`. Obě zmíněné třídy reprezentující jednotlivé stránky dědí od třídy `WizardPage` z balíčku `org.eclipse.jface.wizard` a každé z nich bude dále věnován samostatný odstavec.

Za zmínku také stojí metoda `boolean performFinish()`, která se zavolá po tom, co se průvodce ukončí. Uvnitř metody se nastaví jednotlivé atributy průvodce na hodnoty zadané uživatelem na příslušné stránce průvodce. Tyto hodnoty jsou pak dostupné v obslužné třídě, která průvodce vytvořila (viz výše).

První strana průvodce

První stránku průvodce představuje instance třídy `ProjectNamePage`. Účelem této stránky je od uživatele získat název vytvářeného CRCE projektu a URL cestu k běžícímu CRCE úložišti s uskladněnými komponentami.

Významnou metodou je metoda `void createControl(Composite parent)`, která se volá hned po vytvoření stránky a jejímž úkolem je zinicializovat obsah stránky. Podoba této stránky je velice triviální. Obsahuje dva objekty typu `Label` (popisek), kterým je nastavený text na “Project name” (jméno projektu) a “Path to repository” (cesta k CRCE úložišti). Každý z těchto popisků je umístěný na samostatném řádku společně s objektem typu `Text`, do kterého uživatel vepisuje příslušnou hodnotu. Nakonec je přidáno tlačítko typu `Button`, které se zpřístupňuje tehdy a jen tehdy jsou-li obě textová pole neprázdná.

Po stisknutí tlačítka se uvnitř konstrukce `try/catch` vytvoří nová instance třídy `CRCERepoContentLoader` a jako parametr v konstruktoru se mu

předá cesta k úložišti, kterou uživatel zadal. Potom se nad instancí třídy `CRCERepoContentLoader` zavolá metoda pro načtení mapy identifikátorů komponent. Tato metoda vrací výjimku, pokud by cesta k úložišti byla chybná. Na tuto výjimku reaguje zmíněná `try/catch` konstrukce vyhozením informačního dialogu uživateli. Pokud načtení proběhne správně, zpřístupní se tlačítko "Next" (další) průvodce.

Druhá strana průvodce

Druhou a současně poslední stránkou průvodce je instance třídy `ComponentChooserPage`. Účelem této stránky je od uživatele získat seznam komponent určených k vizualizaci.

I zde se po vytvoření instance třídy `ComponentChooserPage` volá metoda `void createControl(Composite parent)`, uvnitř které se inicializuje vlastní obsah stránky. Na první řádce okna stránky se nachází popisek typu `Label` s textem "Component type:" (komponentový typ, resp. model), kombinovaný seznam typu `Combo` inicializovaný jednotlivými názvy komponentových modelů získaných z první stránky průvodce a tlačítka typu `Button` s nápisy "Select all" (vyber vše) a "Unselect all" (odznač vše). Zbytek okna vyplňuje zaškrtávací seznam s komponentami příslušného komponentového modelu vybraného pomocí kombinovaného seznamu. Seznam je typu `CheckboxTableViewer`. K seznamu a tlačítkám "Select all" a "Unselect all" je navázán posluchač, který aktivuje tlačítko pro dokončení průvodce, pokud je alespoň jedna komponenta vybraná.

5.4 Zhodnocení

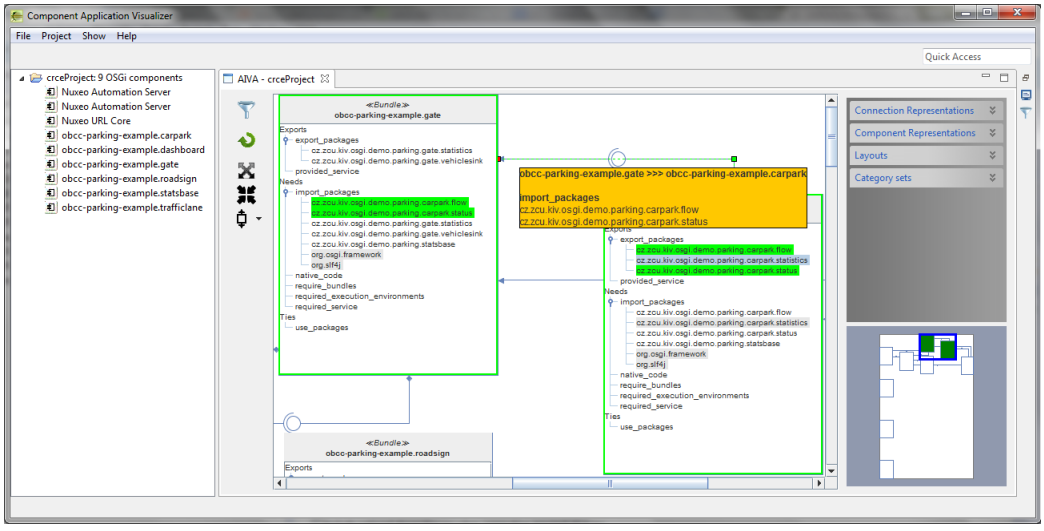
Během analýzy, návrhu a implementace CRCE loaderu jsem narazil na několik problémů, se kterými jsem se musel vypořádat.

První problém představoval plugin *Core* a jeho třída `ComponentLoader`, která slouží jako rodičovská třída pro ostatní pluginy typu *Loader*. Tato třída byla pro potřeby pluginu CRCE loader nedostačující, protože neumožňovala uchovávat komponenty odlišných komponentových modelů. Proto bylo třeba zasáhnout do pluginu *Core* a doimplementovat třídu `GeneralizedComponentLoader`, která přidává možnost uchovávat ENT komponenty v mapě.

Další problém spočíval ve zpracování přijatých metadat z CRCE. Samotné zpracování XML pomocí sériového přístupu SAX se však nakonec ukázalo jako velice přímočaré. Pokročilejší logikou bylo snad jen namapování informací získaných z XML na názvy jednotlivých elementů popsaných v *.entmm* souboru (viz 3.1.3). Pro tuto funkci byla vyčleněna a naimplementována samostatná třída `CRCEXmlToEntmmConnector`.

Posledním výraznějším problémem, který bylo nutné vyřešit, bylo utváření vazeb mezi připravenými ENT komponentami. Protože CRCE loader pracuje s komponentami komponentových modelů, pro který již existují jednotlivé loadery, vyřešil se tento problém poměrně elegantním způsobem. Funkcionalita utváření vazeb mezi hotovými ENT objekty se v rámci jednotlivých loaderů (OSGi, EJB3, SOFA2) oddělila do samostatných tříd (binderů), jejichž instance následně mohly být vytvořeny a použity uvnitř pluginu CRCE loader.

Překonáním všech těchto úskalí se podařilo vytvořit nový plugin do nástroje ComAV – CRCE loader. Následující obrázek 5.9 zachycuje reálný výsledek po užití loaderu. Zobrazuje načtenou ukázkovou komponentovou OSGi aplikaci z CRCE úložiště. Podrobnější popis práce s nástrojem ComAV a jeho novým pluginem CRCE loader nabízí příloha A – *Uživatelská dokumentace*.



Obrázek 5.9: Načtený projekt pomocí CRCE loaderu

6 Závěr

Primárním cílem práce bylo připravit do existujícího nástroje ComAV rozšíření v podobě pluginu CRCE loader. Dosažení tohoto cíle však předcházely další dílčí úkoly, které se zároveň objevily v zadání bakalářské práce a které bylo nutné splnit.

Nejprve se bylo nutné seznámit s tím, co to vůbec je komponentová aplikace a z jakého důvodu se tento přístup v softwarovém inženýrství začal používat. Druhým a velice významným bodem bylo prostudovat implementaci nástroje ComAV a pochopit jeho vnitřní souvislosti. Zejména se jednalo o pochopení implementace jednotlivých pluginů typu loader. Pokud bychom měli být konkrétnější, šlo o způsob, jakým se vytváří v paměti ENT objekty a jak se s nimi dále pracuje. To se nakonec ukázalo jako velice účelné a pomohlo to k optimalizaci kódu. Poslední dílčí úkol představovalo porozumění nástrojové platformě Eclipse RCP. Znalost této platformy sehrála klíčovou roli při implementaci uživatelského rozhraní pluginu CRCE loader a při integraci loaderu do nástroje ComAV.

Všechny body zadání bakalářské práce byly úspěšně splněny. Podařilo se připravit do nástroje ComAV nový modul, který umožňuje načítat z CRCE uskladněné komponenty a převádět je do ENT reprezentace. Funkčnost modulu byla ověřena na notebooku Lenovo Thinkpad E520 s nainstalovaným operačním systémem Microsoft Windows 7 Home Premium. K ověření byly použity testovací OSGi bundly uložené v lokálním CRCE.

6.1 Navrhovaná vylepšení

Na nástroji CRCE vyvíjeném na Katedře informatiky a výpočetní techniky se stále pracuje a v současnosti (ke dni 6. 5. 2014) umožňuje poskytovat pouze metadata pro komponentový model OSGi. Protože CRCE loader může vycházet pouze z toho, co CRCE poskytuje, je jeho podpora taktéž omezena na komponentový model OSGi. Implementačně je však loader na rozšíření podpory pro další komponentové modely připraven a víceméně s takovým rozšířením počítá.

Možným vylepšením by také bylo oddělení rozhraní od implementace v rámci jednotlivých pluginů typu loader. Aktuální struktura jednotlivých pluginů typu loader je plochá a obsahuje čistě jen implementaci. Restrukturalizace jednotlivých pluginů typu loader a vytvoření mezivrstvy v podobě odděleného rozhraní by vneslo do pluginu větší flexibilitu, rozšiřitelnost či modularitu.

Literatura

[1] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 3rd edition, 2002.

[2] Object Management Group. Meta Object Facility (MOF) Core Specification. OMG Specification formal/06-01-01, Object management Group, 2006.

[3] Přemysl Brada. The ENT Meta-Model of Component Interface, version 2. Technical report DCSE/TR-2004-14, Department of Computer Science and Engineering, University of West Bohemia, 2004.

[4] Jaroslav Šnajberk, Přemysl Brada. Implementation of a data layer for the visualization of component-base applications. Technical report, 2010.

[5] Jeff McAffer, Jean-Michel Lemieux, Chris Anisczyk. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley Professional, second edition, 2010.

[6] Přemysl Brada, Kamil Ježek *Ensuring Component Application Consistency on Small Devices: A Repository-Based Approach*. In 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA). IEEE Computer Society Press, 2012.

[7] CRCE Project Assembla Wiki Page, [online]. [cit. 2014-04-20]. Dostupné z: https://www.assembla.com/spaces/crce/wiki

[8] Petr Mošna. *Rozšíření možností reprezentace komponentových aplikací*. Plzeň: ZČU Plzen 2013. Diplomová práce. ZČU Plzeň, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky.

[9] Sun Microsystems, *Inc.: Enterprise JavaBeans^{TM}Specification,* Version 2.0. Sun Microsystems, Inc. 2001.

[10] OSGi Alliance: *OSGi Service Platform Core Specification.* OSGi Alliance, 2009.

[11] T. Bures, P. Hnetynka and F. Plasil: *SOFA 2.0: Balancing advanced features in a hierarchical component model.* Proceedings of SERA 2006, IEEE CS, 2006.

A Uživatelská dokumentace

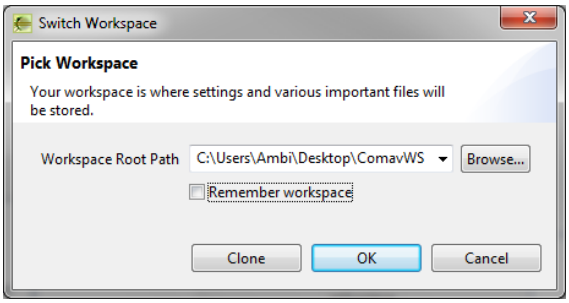
Uživatelská dokumentace seznamuje čtenáře pouze se základními prvky nástroje ComAV. Přesto se však najde výjimka, která je detailněji popsána. Jedná se o rozšíření, které bylo předmětem této bakalářské práce – položka menu pro vytvoření nového CRCE projektu a průvodce, který uživatele provede vytvořením nového CRCE projektu.

A.1 Systémové požadavky

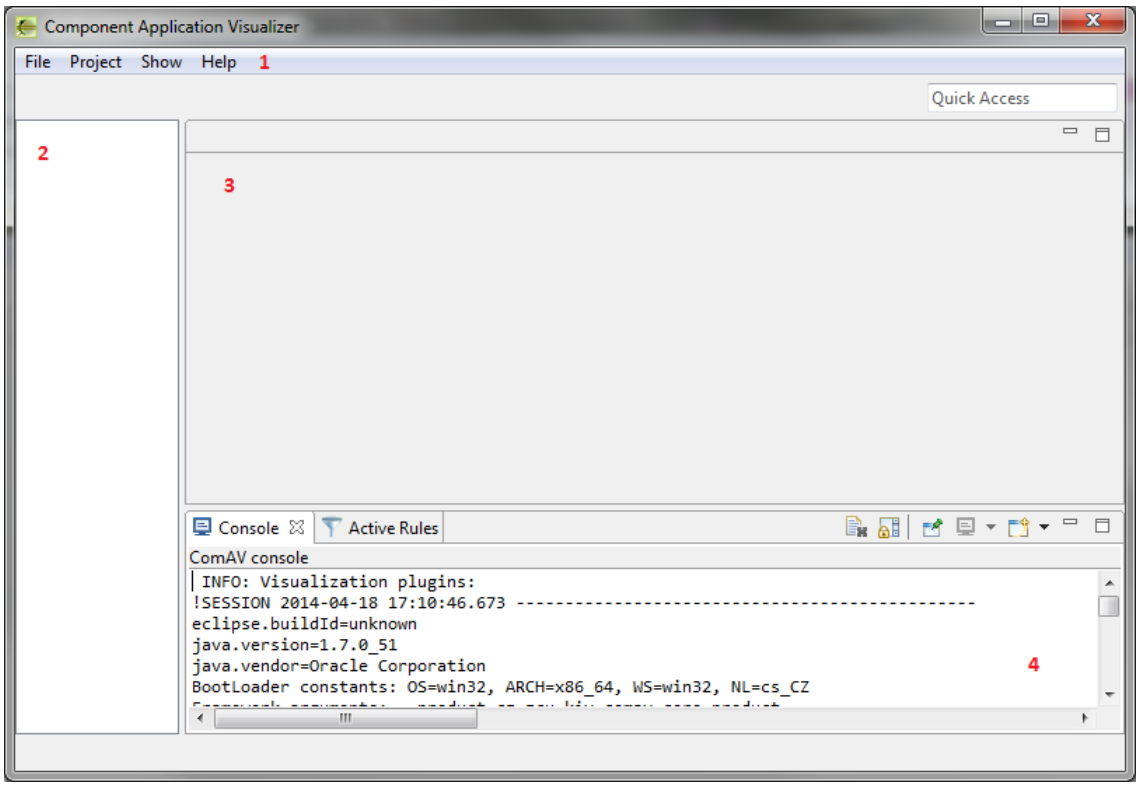
Nástroj ComAV je klientskou aplikací určenou pro operační systém Windows a Linux. Pro jeho běh je zapotřebí mít nainstalované **JRE** (**J**ava **R**untime **E**nvironment) minimální verze **1.6_21**. Doporučené hardwarové požadavky hovoří o operační paměti **512 MB** a procesoru s frekvencí **1 GHz**.

A.2 Spuštění

Aplikace se spouští spustitelným souborem *ComAV.exe* na operačním systému Windows a *ComAV* na operačním systému Linux (viz *readme.txt* na přiloženém CD). Po spuštění se objeví dialog (viz obrázek A.1), pomocí kterého uživatel může nastavit pracovní složku nástroje ComAV. Do této složky nástroj ukládá informace o načtených projektech a případně další předvolby nástroje. V případě opětovného spuštění pak nástroj tyto informace načítá a obnovuje tak konfiguraci nástroje společně s uloženými projekty. Poté, co uživatel vybere pracovní složku, je spuštěn nástroj ComAV (viz obrázek A.2).



Obrázek A.1: Výběr pracovní složky



Obrázek A.2: Hlavní okno nástroje ComAV

A.3 Základní prvky nástroje

Základní prvky nástroje ComAV jsou na obrázku A.2 číselně označeny. Mezi tyto prvky patří:

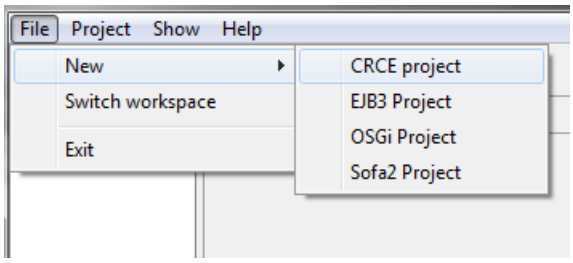
- **Menu** nabízí volby, jako je přepnutí pracovní složky či vytvoření nového projektu (1).
- **Pohled** nacházející se v levé části zobrazuje jednotlivé projekty (2).
- **Editor** poskytuje grafické znázornění komponentových projektů (3).
- **Konzole** slouží pro výpis zpráv (4).

A.4 Vytvoření CRCE projektu

Nástroj ComAV umožňuje vytvářet projekty v rámci různých komponentových modelů. Předmětem bakalářské práce bylo však rozšíření nástroje ve spojení s CRCE, a tak se tento oddíl soustřeďuje na vytvoření nového CRCE projektu.

A.4.1 Otevření průvodce

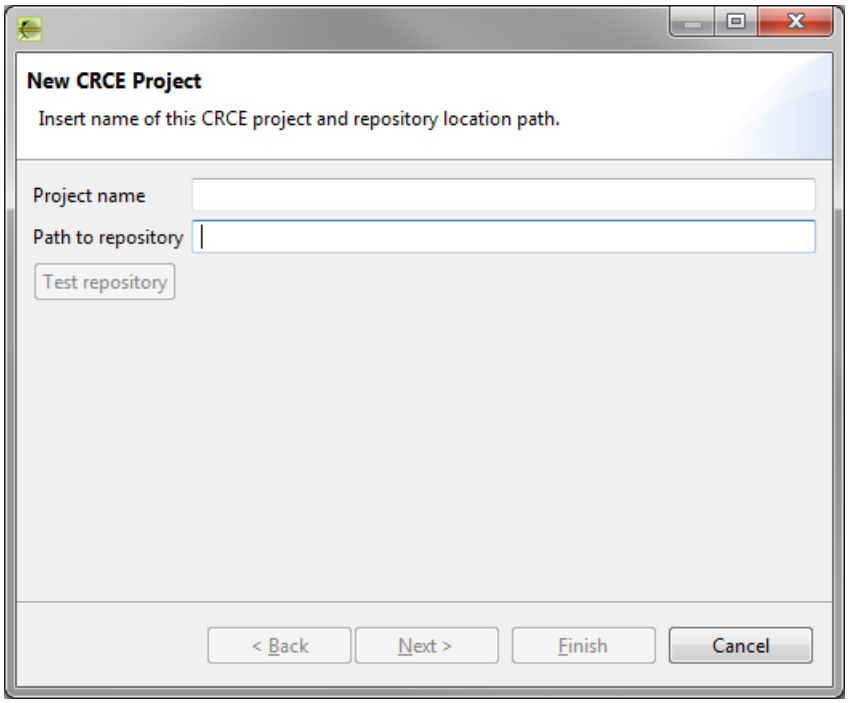
V prvním kroku je třeba otevřít průvodce vytvořením nového CRCE projektu. Toho se docílí vybráním záložky *File* z menu a kliknutím na položku *CRCE project* (viz obrázek A.3).



Obrázek A.3: Otevření CRCE průvodce

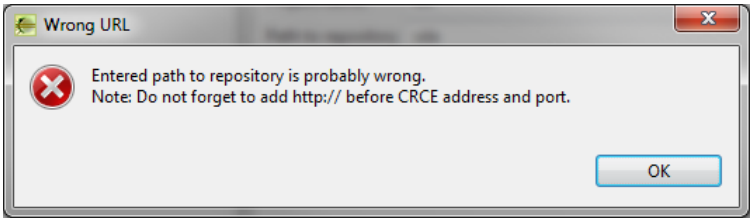
A.4.2 První strana průvodce

Po vybrání položky menu *CRCE project* by se měla objevit první strana průvodce vytvořením nového CRCE projektu (viz obrázek A.4).



Obrázek A.4: První strana průvodce

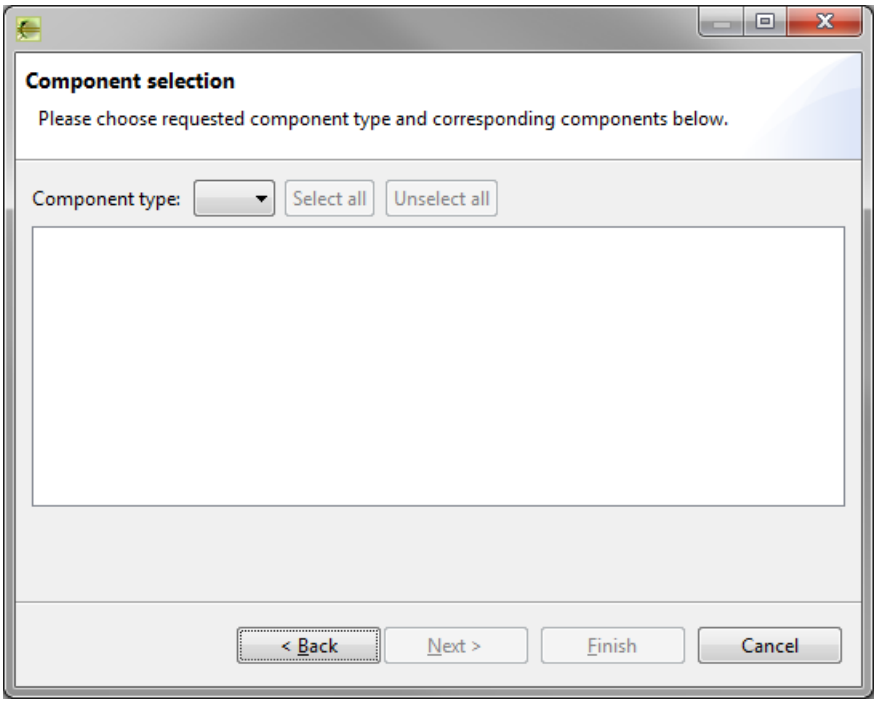
Uvnitř této stránky se nacházejí dvě textová pole. Horní pole slouží pro pojmenování CRCE projektu a pole pod ním slouží ke specifikaci cesty ke spuštěnému CRCE úložišti. Pokud jsou současně obě pole neprázdná, je zpřístupněno tlačítko *Test repository*, které provádí validaci existence úložiště se zadaným URL. Pokud je cesta k úložišti chybně zadaná nebo úložiště se zadanou URL prostě neexistuje, objeví se dialog informující o této skutečnosti uživatele (viz obrázek A.5). V opačném případě je zpřístupněno tlačítko *Next* umožňující přechod na druhou stranu průvodce.



Obrázek A.5: Dialog informující uživatele o chybně zadaném URL k úložišti

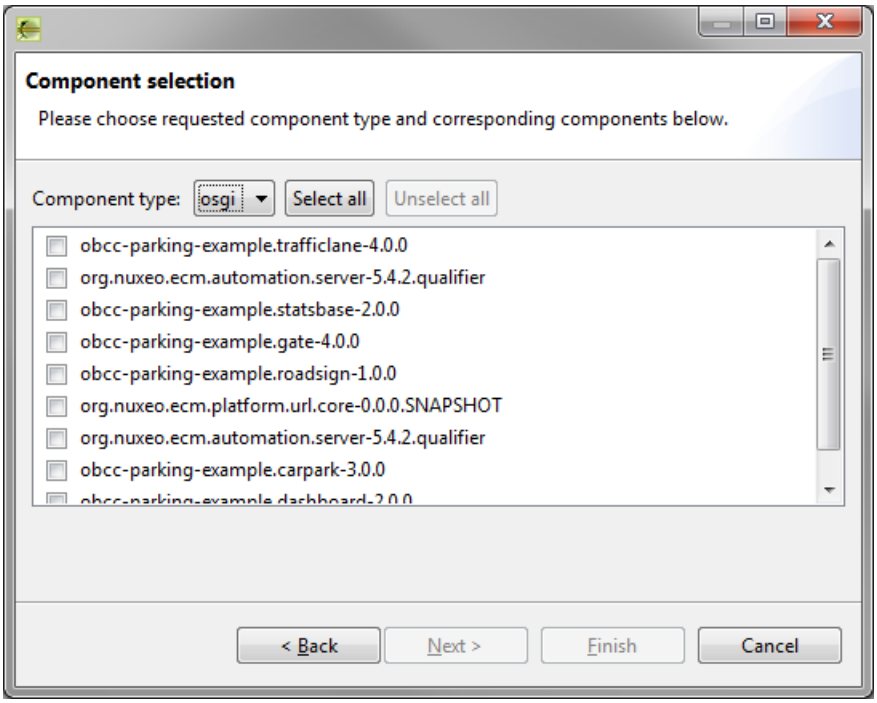
A.4.3 Druhá strana průvodce

Uživatel zadal správně základní údaje na první straně a přesunul se na druhou stranu průvodce (viz obrázek A.6).



Obrázek A.6: Druhá strana průvodce

V horní části se nachází kombinovaný seznam, pomocí kterého uživatel volí komponentový model. Po zvolení komponentového modelu se zobrazí příslušné komponenty v úložišti (viz obrázek A.7).

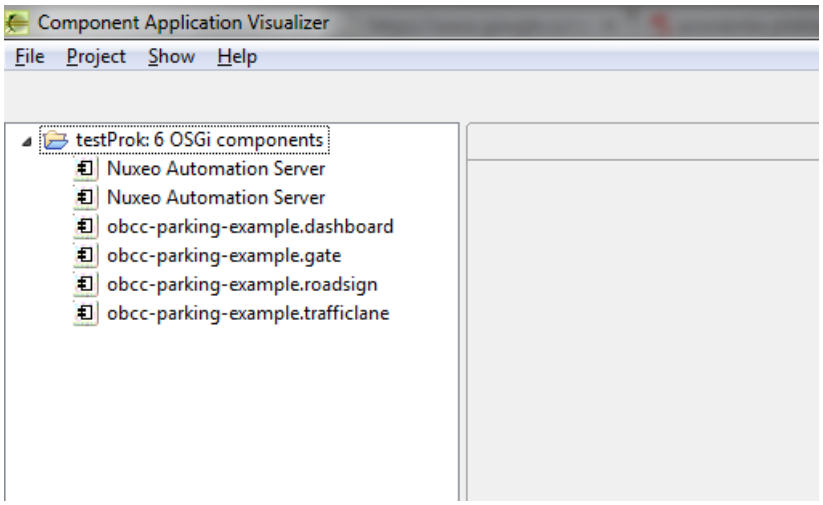


Obrázek A.7: Druhá strana průvodce po zvolení komponentového modelu

Pokud je počet komponent větší, než je možné zobrazit, zobrazí se posuvná lišta. Pomocí té je pak možné seznam posouvat. Komponenty k zobrazení se vyberou kliknutím do příslušného zaškrtávacího pole. Tlačítka nad seznamem *Select all* a *Unselect all* umožňují instantně označit/odznačit všechny komponenty v poli. Pro dokončení průvodce je třeba mít označenou alespoň jednu komponentu v seznamu. Kliknutím na tlačítko *Finish* se průvodce ukončí a proběhne načtení vybraných komponent.

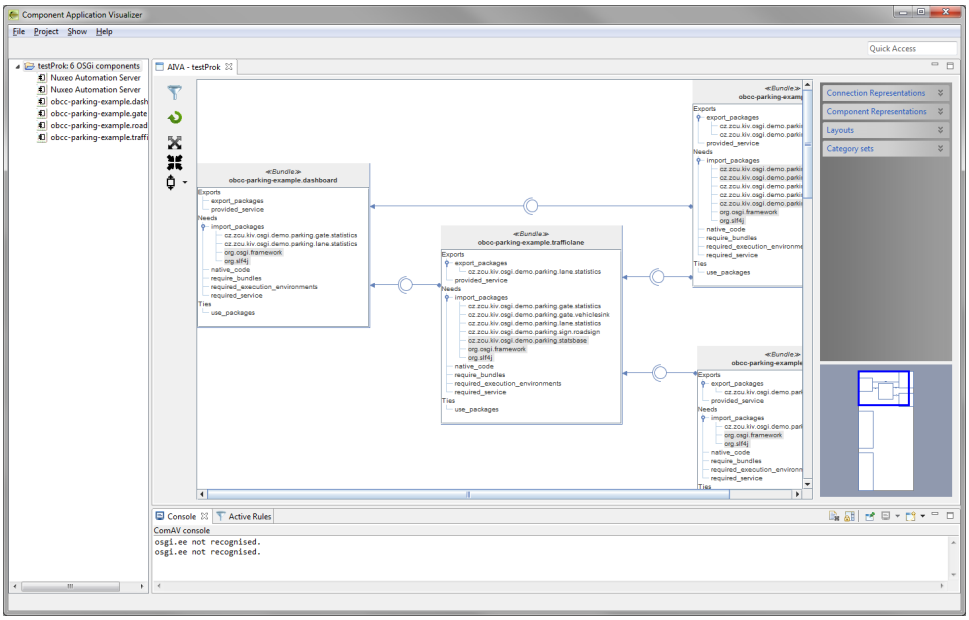
A.4.4 Vizualizace projektu

Po dokončení průvodce by se měl v levé části objevit nový CRCE projekt (viz obrázek A.8).



Obrázek A.8: Vytvořený CRCE projekt

Dvojklik nad kořenovým adresářem projektu nastartuje proces vizualizace a po chvíli by mělo dojít k vykreslení komponent v editoru (viz obrázek A.9).



Obrázek A.9: Vizualizovaný CRCE projekt