|  |
| --- |
| CouchCoders |
| Popis architektury systému CRCE |
| Component Repository supporting Component Evaluation |
|  |
| **Ondřej Kouba** |
| **25.4.2010** |

|  |
| --- |
| Verze 1.2 |

Obsah

[Základní informace 1](#_Toc291613759)

[Jádro Systému 1](#_Toc291613760)

[Webové rozhraní pro ovládání repository 3](#_Toc291613761)

[Rozdělení požadavků mezi jednotlivé servlety 4](#_Toc291613762)

[View Action 5](#_Toc291613763)

[Upload Action 5](#_Toc291613764)

[Store action 6](#_Toc291613765)

[Delete Action 6](#_Toc291613766)

[Filter Action 7](#_Toc291613767)

[Download Action 7](#_Toc291613768)

[Edit Action 8](#_Toc291613769)

[Check Action 8](#_Toc291613770)

[Test Action 9](#_Toc291613771)

[Přidání CoSi podpory 9](#_Toc291613772)

[Změna současné datové vrstvy 10](#_Toc291613773)

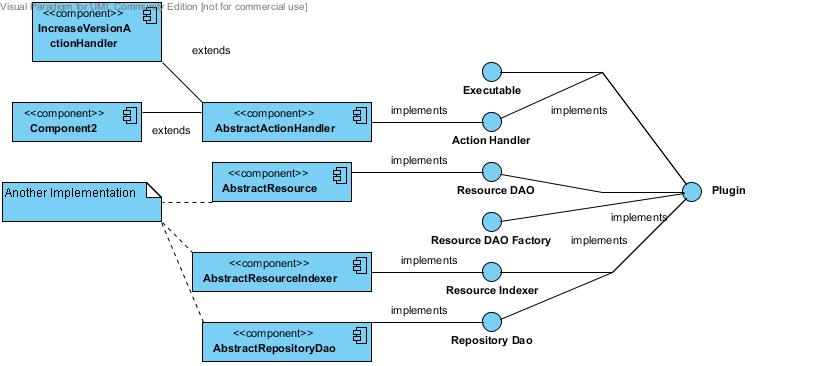
[Závěr 11](#_Toc291613774)

# Základní informace

Component Repository supporting Component Evaluation, dále jen CRCE je postaveno na OSGi architektuře. Konkrétně na frameworku Apache Felix (<http://felix.apache.org>). Základními „stavebními prvky“ jsou tedy ze specifikace OSGI balíky, bundly. Implementace CRCE je provedena pomocí pluginů, které by se dali chápat jako API nad frameworkem. Ty poskytují pohodlnější a konzistentní vývoj rozšíření CRCE.

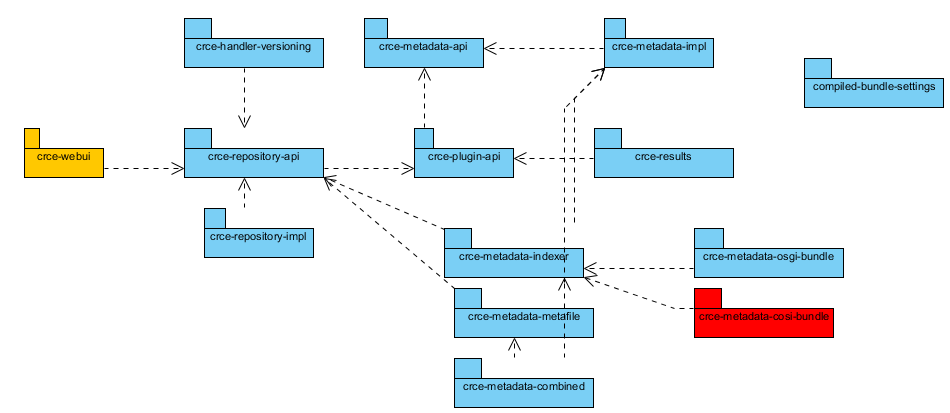
## Jádro Systému

Jádrem systému by se dala rozumět množina bundlů, které implementují rozhraní Plugin. Pomocí třídy PluginManager, jsou při startu načteny všechny třídy, které implementují rozhraní Plugin. Důležité třídy jsou definovány jako Interface, ten implementují abstraktní třídy, které by se daly chápat jako adaptér. Následující konkrétní implementace nás pro zjednodušení nebude zajímat. V našem projektu budeme přistupovat k objektům pomocí rozhraní, nebude nás tedy zajímat konkrétní implementace. Diagram základních funkcí systému definovaných rozhraním naleznete v Diagramu rozhraní a adaptérů.



Obrázek 1 : Diagram Rozhraní a Abstraktních Adaptérů

Vzhledem k tomu, že celý projekt je postaven na OSGI architektuře, jak již bylo zmíněno výše, jsou jednotlivé třídy rozděleny do tzv. bundlů, ten můžeme chápat jako komponentu, stavební jednotku programu. Vzhledem k tomu, že je projekt postaven na Mavenu (http://maven.apache.org) má každý CRCE projekt soubor pom.xml, kde definuje závislosti na ostatních komponentách projektu, jak CRCE, tak ostatních knihoven, ty můžou být z frameworku Felix, ale také to mohou být libovolné knihovny poskytnuté třetí stranou. Dále obsahuje tento soubor obsahuje další metadata, vlastnosti atd. Na diagramu bundlů, můžete vidět spolupráci a závislosti mezi „balíky“. Pro zjednodušení jsou zobrazeny pouze závislosti mezi balíky vytvořenými pro CRCE, tedy nenaleznete tu balíky od třetích stran.

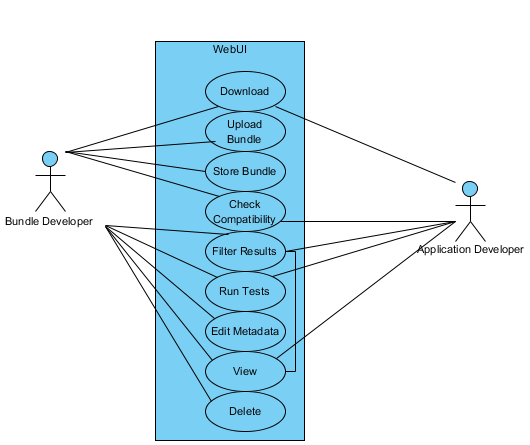


Obrázek 2 : Diagram Bundlů

Oranžový balík crce-webui bude obsahovat webové rozhraní pro crce. Tedy JSP a Servlety. Bundle zvýrazněný červeně bude teprve vytvořen jeho název bude crce-metadata-cosi-bundle a jeho úkolem bude načíst metadata pro COSI komponenty. Více informací v kapitole o vlastní implementaci.

## Webové rozhraní pro ovládání repository

Po sběru požadavků byly identifikovány následující nároky na námi vyvíjený systém, které jsou zobrazeny v diagramu užití webového rozhraní níže.

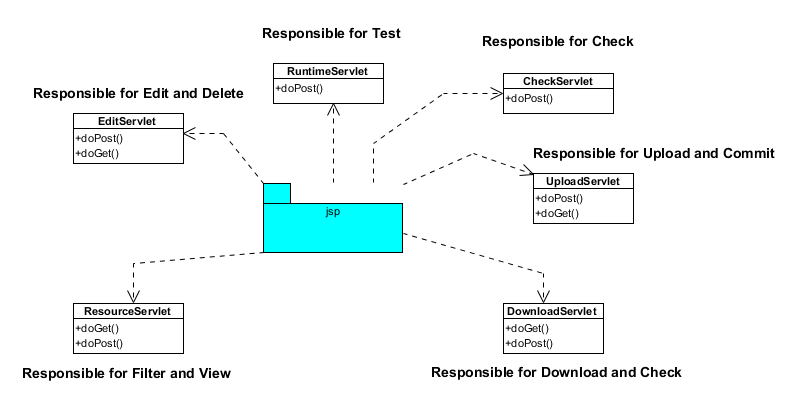


Obrázek 3 : Diagram užití

Vzhledem k tomu, že celá aplikace je napsána v Javě, prostředky pro implementaci těchto požadavků budou kompatibilní s Javou. Konkrétně se bude jednat o Servlety a JSP. Tato technologie byla vybrána kvůli snadné implementaci a komunikaci se stávajícím řešením, také jde o standardně používaná technologie v Java EE aplikacích. Požadavky tedy budou rozděleny mezi několik servletů, které budou obsluhovat prezentační vrstvu v tomto případě webovou, tedy JSP. Pro přehlednou implementaci a oddělení aplikační vrstvy od prezentační jsme se rozhodli používat rozšíření JSP o custom tagy, konkrétně o JSTL(http://jstl.java.net).

## Rozdělení požadavků mezi jednotlivé servlety

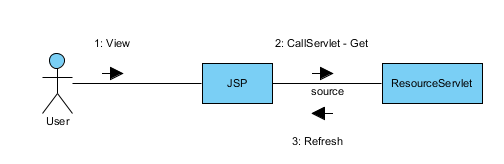
Po indetifikaci požadavků a vybrání technologií pro jejich realizaci byl navrhnut následující model rozdělení funkčnosti mezi pět servletů. JSP stránky jsou brány pouze jako prezentační vrtsva, stejně jako crce implementace jako vrstva datová. Po analýze jednotlivých požadavků a jejich logickém rozdělení byl navrhnut model na obrázku 4.



Obrázek 4 : Diagram obsluhy JSP

## View Action

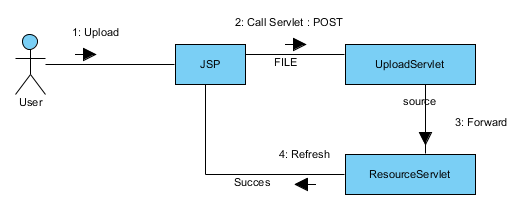
View action je zodpovědná za správné zobrazení komponent ve Store, Bufferu a PluginManageru. Jejím úkolem je čtení dat pomocí API poskytnutém zadavatelem. Pro bezproblémový přenos dat mezi Servletem a JSP bude využita HttpSession jako transportní prostředek. JSP bude identifikovat parametrem link odkud požadavek na načtení dat přišel. Bohužel není možné jinak zjistit odkud požadavek přišel, třída HttpRequest tato data bohužel neukládá. Pokud nebude parametr link zadán bude provedeno defaultní přesměrování a poskytnuty defaultní data. Za tuto akci je zodpovědná třída ResourceServlet a její komunikační diagram můžete vidět na obrázku 5.



Obrázek 5 : Resource Diagram

## Upload Action

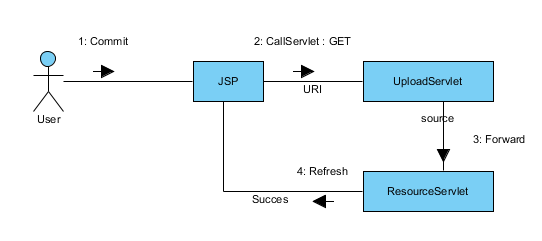
Upload action je zodpovědná za nahrání bundlu do Bufferu, ten je unikátní pro každý request, tedy pro každého uživatele. Úkolem tohoto požadavku je tedy přidat do Bufferu metodou put resource vložený uživatelem pomocí file dialogu. Servlet Upload bude dědit od templatu FileUploadServlet. Po úspěšném uploadu zavolá Resource servlet pro refresh komponent bufferu a synchronizaci. Tato akce je symbolicky znázorněna na obrázku 6 pomocí komunikačního diagramu.



Obrázek 6 : Upload Diagram

## Store action

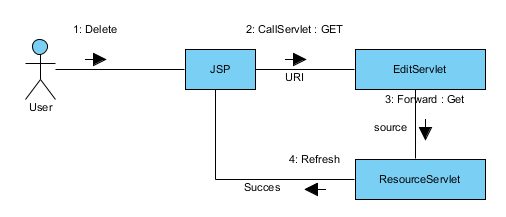
Komponenty z dočasného úložiště (Buffer), je možné uložit do trvalého úložiště(Store) akcí commit. Tuto funkcionalitu bude zajišťovat Upload servlet metodou Get. Úkolem tohoto servletu bude přemístit bundle z Bufferu do Store, tedy odebrat bundle z bufferu a přidat do store. Po úspěšném provedení akce bude uživatel přesměrován na původní stránku pomocí ResourceServletu. Více na obrázku číslo 7.



Obrázek 7 : Commit Diagram

## Delete Action

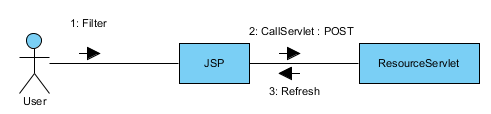
Delete akce je společná pro store a buffer. Jejím úkolem je odstranit bundle z úložiště(repositury). Tuto akci obsluhuje EditServlet pomocí metody Get. Jeho úkolem je odebrat komponentu z úložiště a zavolat ResourceServlet s parametrem link pro obnovení výpisu komponent a správné přesměrování. Více na obrázku číslo 8.



Obrázek 8 : Delete Diagram

## Filter Action

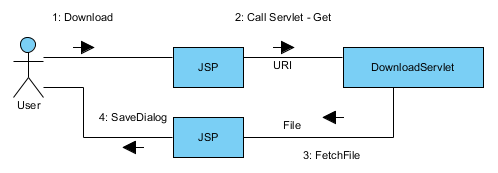
Filter akce je vlastně speciálním případem View Akce. Proto ji bude obsluhovat stejný servlet, tedy ResourceServlet. Požadavek na tuto akci je zpacování filtru, odstranění položek které nevyhovují a vrácení bundlů plus přesměrování na stránku ze které byl požadavek vyslán. Více na obrázku číslo 9.



Obrázek 9 : Filter Diagram

## Download Action

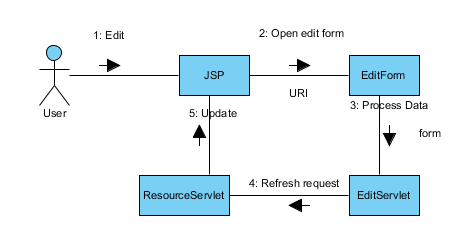
Download akce bude dostupná pro obě perspektivy programu tedy jak pro trvalé úložiště – store, tak i pro dočasné úložiště – buffer. Obsluhu akce bude zprostředkovávat DownloadServlet. Jeho úkolem bude podle URI najít komponentu ve právném repository, a vrátit zpět jsp, které ho vyvolalo, správně pojmenovaný File. Více na obrázku číslo 10.



Obrázek 10 : Download Diagram

## Edit Action

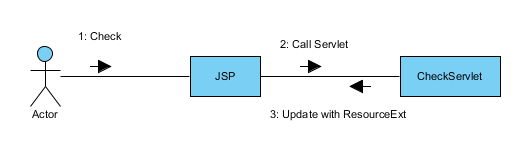
Edit akci bude obsluhovat servlet EditServlet. API poskytnuté zadavatelem obsahuje základní kontrolu, nedovolí zapisovat nesmyslná data. Servlet ale dále bude muset kontrolovat duplicitu jednotlivých položek pro properties soubotu. Tento Servlet bude zpracovávat celkem tři formuláře a to CapabilitiesForm, PropertiesForm(Podobný jako Capabilities, ale nepovoluje duplicitní hodnoty) RequirementsForm a CategoryForm. Formuláře budou posílat informaci o tom ze kterého formuláře přichází požadavek a také ze které stránky. Pokud edit akce bude úspěšná změněná data budou uložena a uživatel přesměrován na původní stránku.



Obrázek 11 : Edit Diagram

## Check Action

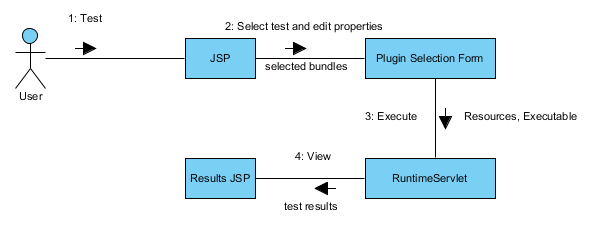
Úkolem této akce je zkontrolovat, zda jsou pro vybrané bundly splněny všechny závislosti. K tomu složí interface Resolver poskytnutý zadavatelem. Implementace této třídy vrací seznam Requirement, které nebyly splněny. Ty budou uloženy do speciální „obalové“ třídy ResourceExt, která bude obsahovat informaci o tom, zda závislost je splněna nebo ne. Resource s nesplněnou závislostí budou zvýrazněny na stránce jinou barvou. Tuto funkcionalitu bude zajišťovat CheckServlet s pomocí ResourceServletu.



Obrázek 12 : Check Diagram

## Test Action

Slouží k otestování komponent uživatelem definovanými testovacími pluginy. To jsou ty Pluginy, které implementují rozhraní Executable. Viz obr. 1. Uživatel může v obou perspektivách (Buffer i Store) vybrat balíky, které chce otestovat. Po stisknutí tlačítka execute bude uživateli zobrazen formulář s výběrem dostupných testovacích pluginů a textová pole pro zadávání parametrů pro jednotlivé testy. Po odeslání informací RuntimeServletu budou testy spuštěny a výsledek bude zobrazen na další stránce.

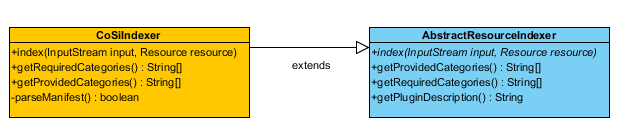


Obrázek 13 : Test Diagram

# Přidání CoSi podpory

Kromě uživatelského rozhraní pro repository je dalším požadavkem zajistit správné zpracování komponent pro framework CoSi (komponentový framework vyvíjený na KIV za účelem výzkumu závislostí a nahraditelnosti komponent, je inspirován OSGi). Systém obsahuje tzv Indexery, které procházejí vložený soubor a určují, o jaký formát se jedná a zjišťují metadata. Dosavadní indexery dokáží získat základní informace o jakémkoliv souboru k tomu OSGIIndexer dokáže načíst metadata z manifestu pro jakýkoliv OSGI bundle. Realizace podpory pro CoSi tedy bude naprogramování CoSiIndexeru, který dokáže správně naparsovat manifest tohoto formátu a uložit ho v OBR kompatibilním formátu. Tato třída bude uložena v balíku crce-metadata-cosi-bundle a bude mít stejné dependence jako osgi bundle. Viz obr.2.

Třída CoSiIndexer bude dědit od třídy AbstractResourceIndexer. Bude obsahovat vlastní implementace metod z rozhraní ResourceIndexer. Toto rozhraní má tři metody. Index(), getProvidedCategories a getRequiredCategories. Úkolem metody index() bude naparsovat manifest CoSi bundlu a uložit zjištěná data do Resource vložené jako parametr této metody. getProvidedCateogires vrací jaké kategorie tento indexer dokáže vyplnit. Zde se bude vracet konstanta cosi - tento indexer vyplňuje data pouze pro CoSi formát. GetRequiredCategories má podle definice rozhraní vracet názvy kategorii, se kterými umí indexer pracovat. Vzhledem k tomu, že bundly budou zabaleny buď v zip nebo jar formátu, bude indexer vracet právě tyto dvě kategorie. Class diagram naleznete na obrázku 14.

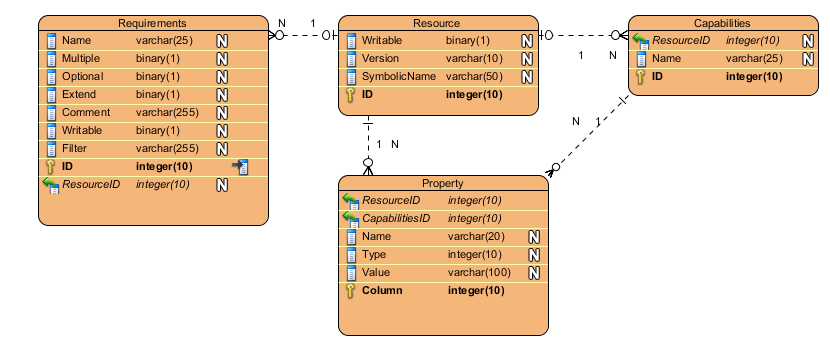


Obrázek 14 : CoSi class diagram

# Změna současné datové vrstvy

V současné implementaci jsou data uložena do xml souborů. Zadavatel vyjádřil úmysl tato data migrovat do databáze. Struktura databáze je již předem specifikována API od zadavatele. Jednotlivé tabulky budou mít mezi sebou stejné relace jako mají Resource s ostatními „bean“ třídami. Pro migraci dat do databáze je třeba zvolit přístup k DB. Z důvodů snadné implementace, přehlednosti a možnosti psát mapovací kód do anotací a ne do xml souborů, byl vybrán přístup objektového mapování, konkrétně JPA.

Z API poskytnutého zadavatelem a z OBR definice (<http://felix.apache.org/site/apache-felix-osgi-bundle-repository.data/obr-entities.png>) lze snadno odvodit následující ERA model (obrázek 15), který bude použit pro implementaci databáze. Jednotlivé implementační třídy potom musí obsahovat správné mapovací anotace JPA.



Obrázek :ERA model

# Závěr

Tento popis Architektury byl vypracován v rámci semestrální práce z předmětu KIV/ASWI. Všechny UML diagramy byly vytvořeny v programu VisualParadigm for UML 8.1. Starší verze architektury se mohou lišit od aktuální implementace, základní smysl by však měl zůstat zachován.

CouchCoders, Duben 2011, Plzeň