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

Diplomová práce

Klient pro aktualizace aplikací z úložiště komponent

Plzeň, 2014 David Švamberk

Prohlášení

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

V Plzni 26.6.2014 David Švamberk

Anotace

Tato práce se zaměřuje na problematiku spojenou s úložišti softwarových komponent a jejich klienty. Nejvýznamnější oblastí komponentového vývoje je pro tuto práci technologie OSGi. Na ní je postavena i aplikace GUICA (Graphical User Interface for Component Administration), jejíž rozšíření je hlavním cílem. GUICA by pak ve výsledku měla být schopna spolupracovat s úložištěm komponent CRCE (Component Repository Supporting Compatibility Evaluation), které podporuje dodatečnou kontrolu kompatibility na základě popisných metadat. Dohromady by měly tyto dva systémy umožňovat konzistentní aktualizaci aplikací nasazených v běhovém prostředí OSGi Frameworku.

Abstract

This work focuses on issues related to the repository of software components and their clients. The most important area of component development for this work is OSGi technology. On OSGi is also built aplication called GUICA (Graphical User Interface for Component Administration), whose extension is the main goal. In result the GUICA would be able to work with the repository component CRCE (Component Repository Supporting Compatibility Evaluation), which supports an additional compatibility checking based on metadata. Altogether, these two systems would allow consistent updating applications deployed in the runtime environment of OSGi Framework.

Obsah

1. Úvod

Hlavními tématy, kterých se práce týká, jsou úložiště komponent a s nimi spojená kompatibilita komponent, dále také klienti s podporou aktualizací aplikací. CRCE (Component Repository supporting Compatibility Evaluation), úložiště s podporou určování kompatibility komponent, které je vyvíjeno na Katedře informatiky a výpočetní techniky Západočeské univerzity, je pro účely této práce stěžejním. Je navrženo tak, aby klienti, kteří budou toto úložiště využívat, mohli mít veškeré dostupné informace reprezentované popisnými metadaty k určení vzájemné kompatibility aktualizovaných komponent. Jedním z takových klientů se má stát GUICA (Graphical User Interface for Component Administration). Je to grafické rozhraní pro konzoly OSGi. Open Services Gateway initiative je technologie pro platformu Java, která umožňuje v rámci OSGi Frameworku přidávat, aktualizovat a odebírat moduly (tzv. bundly) za běhu aplikace.

Cílem této diplomové práce je rozšířit GUICA o možnost instalovat komponenty z různých typů úložišť. Nejdůležitějším typem je již zmíněné CRCE. Kromě funkce instalace je také cílem podpora aktualizace již nainstalovaných komponent.

Motivací pro tento projekt byl na jedné straně značný rozvoj technologie OSGi a komponentového programování, na druhé straně neustálý vývoj a sofistikovanost různých vestavěných (embedded) zařízení, jako jsou například inteligentní domácí spotřebiče.

V úvodní části textu je osvětleno teoretické pozadí problematiky. Jsou zde obecně popsány technologie OSGi včetně jejího využití v distribuovaném prostředí a komponentové programování jako takové. Pozornost je rovněž věnována problematice kompatibility komponent. Úvodní část uzavírá porovnání různých úložišť softwarových komponent, ať už se jedná o komerční či výzkumné (CRCE), spolu s klienty pro aktualizace. Druhá část práce je zaměřena spíše na praktické aspekty dané problematiky. Spadá sem analýza, návrh architektury klientské části systému a popis její implementace.

2. Komponentové programování

Při softwarové analýze nějakého komplexnějšího problému se často používá technika nazývaná dekompozice. Při dekompozici se komplexní problém rozdělí do několika dílčích jednodušších problémů, které jde lépe vyřešit odděleně. Komponentové programování [1] se zakládá na tomto principu a snaží se ho využít při návrhu softwarových komponent, ze kterých se pak skládají komplexní systémy.

Softwarovou komponentou může být např. softwarový balíček, webová služba nebo modul, který zapouzdřuje sadu souvisejících funkcí nebo dat. Všechny systémové procesy jsou umístěny do samostatných komponent tak, aby všechna data a funkce uvnitř každé byly sémanticky související (podobně jako obsah tříd).

Komponenty spolu vzájemně komunikují přes rozhraní z důvodu lepší koordinace celého systému. Tudíž zprostředkovává-li komponenta nějaké služby, odstiňuje svou vnitřní reprezentaci pomocí rozhraní. Tento princip má za následek zapouzdřenost komponent vzhledem k okolí.

Je-li ke komponentě přistupováno přímo nebo přes sdílený exekutivní kontext či síťové připojení, pak jsou techniky jako serializace nebo marshalling nezbytné k určení správné komponenty.

Opětovné použití je jednou z nejdůležitějších charakteristik kvalitních softwarových komponent. Programátoři by měli navrhovat a implementovat softwarové komponenty takovým způsobem, aby mohly být znovu použity v různých aplikacích. Někdy to vyžaduje značné úsilí a povědomí o komponentovém způsobu programování, ale ušetří se náklady na analýzu a vývoj. Další důležitou vlastností komponenty je její velmi dobrá a důkladná dokumentace. Samozřejmě nesmí chybět dostatečné pokrytí testy, které by měly být robustní a komplexně kontrolovat validitu vstupů komponenty. V případě potřeby je také nutné, aby byly komponenty schopné přenést odpovídající chybové hlášky nebo návratové kódy zpět volajícímu.

2.1. OSGi Framework

Technologie OSGi [2] je souborem specifikací, které definují dynamický systém komponent pro platformu Java a poskytují modulární architekturu rozsáhlých distribuovaných systémů. OSGi umožňuje komponentizaci softwarových modulů a aplikací, zajišťuje vzdálenou správu a interoperabilitu aplikací či služeb. Při použití OSGi modulů lze dosáhnout zvýšení produktivity vývoje ve všech jeho fázích.

Vývoj této technologie zaštiťuje OSGi Aliance [3], což je celosvětové konsorcium inovátorů technologií, kteří definují a zdokonalují otevřené specifikace umožňující vytváření modulárních aplikací postavených na technologii Java. Aliance poskytuje specifikace, referenční implementace, soubory testů a certifikátů. Aliance rovněž podporuje spolupráci významných společníků uvnitř i vně s cílem dodávat na trh inovativní řešení založená na otevřených standardech. Členové aliance představují zástupce mnoha odvětví, patří sem poskytovatelé hardware a infrastruktury, provozovatelé sítí, prodejci softwaru, softwaroví vývojáři, dodavatelé spotřební elektroniky či jiných zařízení a výzkumné instituce.

Technologie

Jak již bylo zmíněno, technologie OSGi [4] je definována souborem otevřených specifikací. Tyto specifikace podporují vývojový model, ve kterém jsou aplikace sestaveny z mnoha odlišných (v nejlepším případě i znovu použitelných) komponent. OSGi umožňuje komponentám abstrahovat od jejich vnitřní implementace a komunikovat s ostatními pomocí služeb. Tento jednoduchý model ovlivňuje téměř všechny aspekty procesu vývoje softwaru.

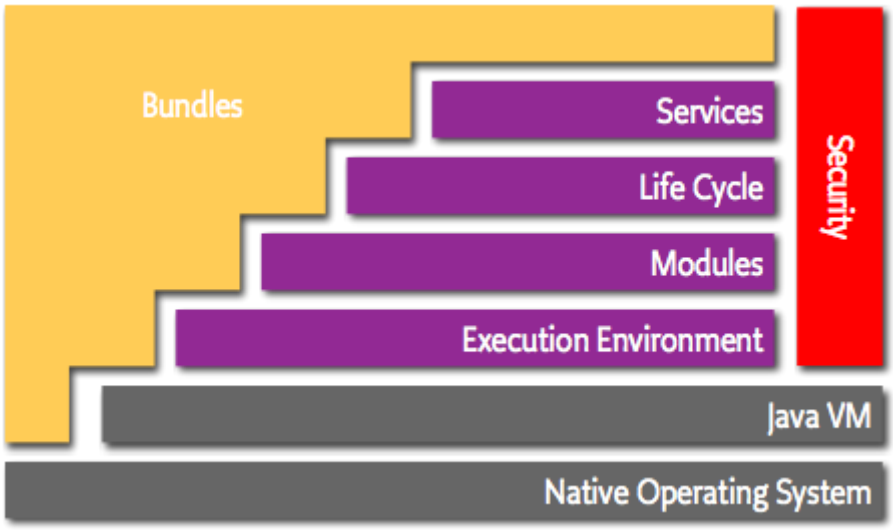
OSGi je inovativní technologie, které se reálně daří pomocí komponentového systému nalézat řešení mnoha skutečných problémů v oblasti vývoje softwaru. Zdrojový kód aplikací je jednodušší, lépe se testuje a jeho opětovné použití je snazší a častější. Nasazení je snazší na řízení a chyby jsou detekovány dříve.

Při navrhování technologie OSGi nebylo účelem hledání možnosti, jak spustit více aplikací v jednom VM (Virtual Machine - virtuální stroj). K tomu účelu

slouží aplikační servery. Účelem OSGi bylo vytvářet aplikace složené ze znovupoužitelných komponent. Tyto komponenty jsou sestavovány dynamicky a mohou být přidávány či odebírány i za běhu aplikace.

Architektura

Technologie OSGi je založena na vrstveném modelu, který je popsán na obrázku níže.



Obrázek 1: Model vrstev OSGi[1]

Následující seznam obsahuje stručný popis jednotlivých částí modelu.

Bundles - OSGi komponenty aplikace

Services - vrstva služeb dynamicky propojující bundly

Life-Cycle - tato část modelu OSGi definuje životní cyklus komponenty pomocí API[2] podporujícího operace s bundly jako jsou install, start, stop, update a uninstall

Modules - zajišťuje import, export balíčku a služeb komponent

Security - vrstva určená k zajištění bezpečnosti

Execution Environment - vrstva prostředí pro běh aplikace definuje, jaké metody a třídy jsou k dispozici pro konkrétní platformu

---

[1] Přejato z [4]

[2] API (Application Programming Interface) neboli aplikační programovací rozhraní - formálně se jedná o množinu rozhraní vystavovaných aplikacemi nebo např. knihovnami k využití okolními systémy; tato množina rozhraní v sobě zapouzdřuje funkcionalitu určenou ke zveřejnění a odstiňují okolí od vnitřní implementace

Bundly

Bundly pro OSGi představují základní koncept umožňující dekompozici sytému. Modularita je pevně zakotvena v jádru specifikace OSGi a je součástí jejího konceptu. Z hlediska platformy Java je OSGi bundle obyčejný JAR soubor. Podle JAR standardu jsou veškerá data souboru kompletně viditelná pro ostatní JAR soubory. Oproti tomu OSGi bundle zviditelňuje pouze data explicitně určená k exportování. Bundle, který závisí na nějaké jiné OSGi komponentě, musí explicitně uvést, jaké konkrétní balíčky či služby hodlá využívat. Implicitně se žádné balíčky ani služby nesdílejí.

Výše zmíněná data jsou uložena v souboru manifest.mf, který je součástí každého JAR souboru.

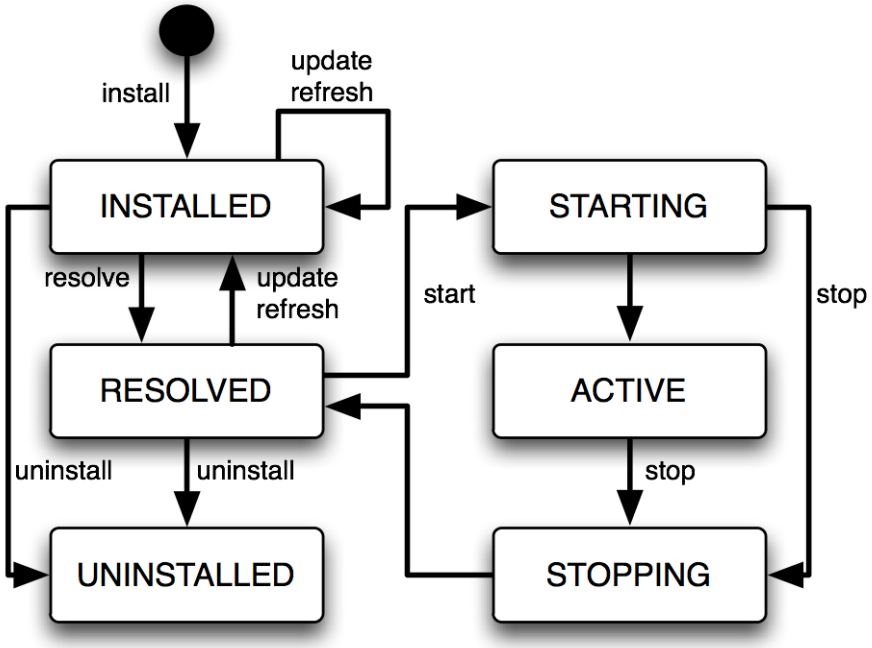
OSGi přidává do manifestu následující informace:

- Bundle-Name – název komponenty určený k prezentačním účelům
- Bundle-SymbolicName – název komponenty určený k její identifikaci např. cz.zcu.kiv.guica.update-manager
- Bundle-Version – verze komponenty složená z major, minor, patch + sufix verze např. 1.2.0-SNAPSHOT
    - Bundle-SymbolicName + Bundle-Version lze považovat za unikátní identifikátor komponenty v rámci OSGi Frameworku
- Export-Package – výčet balíčků, které daný bundle exportuje pro využití ostatními komponentami; záznam obsahuje symbolický název a volitelně i verzi či rozsah verzí
- Import-Package – stejné jako u předchozího, jen s tím rozdílem, že na uvedených komponentách daný bundle závisí a bez nich nemůže být zaveden do OSGi běhového prostředí
- Manifest může obsahovat ještě další informace jako např. Bundle-Activator, Bundle-ManifestVersion, Bnd-LastModified

Těchto informací je využíváno v řešení závislostí OSGi komponent a též při jejich ukládání do úložišť.

Životní cyklus OSGi komponenty

OSGi bundle se zavádí do běhového komponentového prostředí označovaného jako Framework. Toto označení může být někdy poněkud matoucí. V rámci tohoto běhového Frameworku prochází komponenty různými stavy, jejichž přehled se nachází na následujícím diagramu.



Obrázek 2: Životní cyklus OSGi komponenty[1]

- Installed – zaveden do Frameworku, zatím nebyly vyřešeny závislosti
- Resolved – bylo ověřeno, že jsou všechny závislosti vyřešeny
- Uninstalled – odstraněn z Frameworku
- Starting – spouští se
- Active – aktivován a připraven k použití
- Stopping – zastavuje se

Přechody mezi stavy se dějí buď při startu Frameworku, nebo na požadavek uživatele. OSGi podporuje ovládání běhového prostředí pomocí konzole, která umožňuje spravovat komponenty v rámci Frameworku. Také umožňuje instalování, spouštění, zastavování či aktualizaci komponent, aniž by musela být restartována JVM.

[1] Převzato z http://zenit.senecac.on.ca/wiki/index.php/File:Bundle-lifecycle.png

Události

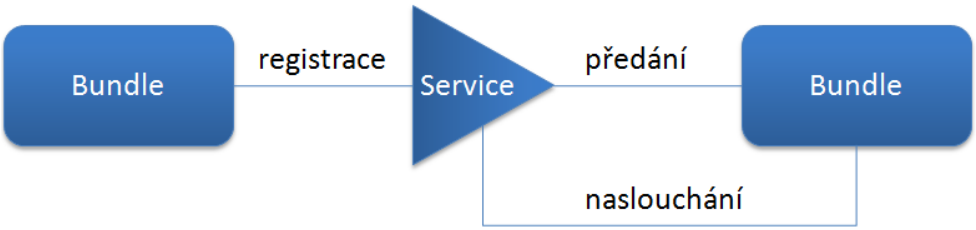
Na životní cyklus komponenty navazují události. Událost tzv. Event je totiž vyvolána při každém přechodu komponenty do jiného stavu. Událostí je však více druhů a ne všechny jsou spojeny přímo s komponentami. Souhrnně by se daly rozdělit do třech základních skupin:

- Události Frameworku
- Události komponent
- Uživatelské události

Nezávisle na typu události může být každá z nich zpracovávána mechanizmem zvaným EventAdmin. Ten běží jako služba a může být využit k zachytávání konkrétních událostí, které pak mohou být zpracovány na aplikační úrovni.

Služby

Důvodem pro vznik modelu služeb byl hlavně fakt, že je v jazyce Java obtížné vytvářet kolaborativní model s výhradním využitím sdílení tříd.
Standardním řešením v jazyce Java je použití tzv. Factory (továrna), která využívá dynamický class loading a privátní konstruktory v kombinaci se statickou metodou newInstance(). OSGi však přináší inovativnější model, kterým je registr služeb. Bundle může zaregistrovat v rámci OSGi registru službu definovanou rozhraním a ostatní OSGi komponenty ji mohou začít ihned využívat. V praxi jsou důsledky takové, že jedna komponenta vzdáleně zavolá metodu jiné komponenty, jejíž výstup je pak vrácen volající komponentě. Bundle může dokonce čekat na konkrétní službu a jakmile ji někdo zaregistruje, může ji začít okamžitě využívat, tento mechanismus se nazývá Service Tracker.
Hlavními rozhraními pro služby jsou ServiceRegistration, ServiceReference a ServiceListener. Libovolný počet komponent může zaregistrovat stejný typ služby a zároveň libovolný počet bundlů může využívat stejnou službu. Toto je znázorněno na následujícím obrázku.



Obrázek 3: Model OSGi služeb[1]

V takto navrženém modelu může dojít k situaci, kdy si v rámci stejného rozhraní nebo třídy zaregistruje služby více komponent. Z tohoto důvodu má každá zaregistrovaná služba sadu standardních a uživatelských vlastností. Jednou z možností, jak služby podle těchto vlastností vyhledávat, je zvláštní filtrovací jazyk, díky němuž je možné vybrat pouze ty služby, které daný bundle požaduje.
Systém služeb je dynamický. To znamená, že bundle může odejmout své služby z registru, zatímco je ostatní komponenty stále používají. Pak je nutné zajistit, aby komponenty nadále již takové služby nevyužívaly a odstranily veškeré reference na ně. Dynamický systém služeb se také přidává k podpoře instalování a odinstalování komponent za běhu, aniž by tím negativně ovlivnil zbytek systému.
Systém služeb pomáhá zjednodušit návrh celé aplikace, umožňuje použít standardní nástroje pro ladění a napomáhá k získání lepšího přehledu o propojení celého systému.

[1] Přejato z [4]

3. REST API využívající OSGi

REST webové služby [5] jsou v současné době velmi populární a jsou minimálně rovnocennou konkurencí pro klasické webové služby, jejichž představitelé jsou např. JAXWS či AXIS. Webové služby obecně se zakládají na principu vzdáleného volání procedur podobně jako u RPC (Remote Procedure Call), jen s tím rozdílem, že zde je využíván http protokol. Klasické webové služby používají nadstavbu nad http protokolem zvanou SOAP (Simple Object Access Protokol), která umožňuje vzdálené volání funkcí. Technologie REST webových služeb využívá možností samotného http a mapuje volání procedur na URL, viz níže.
Jelikož je tato práce zaměřena na OSGi a při implementaci bylo využito REST API, je zapotřebí zde popsat některé moderní přístupy spojující tyto dva aspekty. Takovými technologiemi jsou např. Jersey Framework a Restlet Framework uvedené v dalším textu.

3.1. JAX-RS

Proces vývoje REST webových služeb bez použití vhodných nástrojů není jednoduchý, zvláště když mají bez problémů podporovat publikování dat různými prezentačními kanály a poskytovat abstrakci nízkoúrovňové síťové komunikace.
Za účelem zjednodušení vývoje těchto služeb a jejich klientů bylo navrženo standardní a přenositelné JAX-RS API pro platformu Java. JAX-RS je definováno v JSR (Java Specification Request), což je obdoba RFC (Request For Comments). Prvotní koncept definovalo JSR 311 s názvem „The Java™ API for RESTful Web Services“. Tento návrh byl dále rozvíjen v JSR 339, který je označován jako „The Java™ API for RESTful Web Services 2.0“ viz [7]. Jelikož je JAX-RS pouze specifikace podobně jako OSGi, bylo nutné pro ověření navržených principů realizovat nějakou referenční implementaci. Za tímto účelem byl vytvořen projekt Jersey Frameworku charakterizován v následující kapitole.

3.2. Jersey

Framework Jersey REST Web Services [6][5] (dále jen Jersey) je Open Source Framework určený pro vývoj REST webových služeb, který poskytuje podporu pro JAX-RS API a slouží jako referenční implementace JAX-RS. Jersey je však více než prostou referenční implementací JAX-RS. Nabízí také své vlastní API, které rozšiřuje JAX-RS nástroje o další funkce a utility pro ještě větší zjednodušení vývoje REST služeb. Jersey také poskytuje podporu pro SPI[1], díky čemuž ho lze rozšířit tak, aby co nejlépe vyhovovalo individuálním potřebám uživatelů.
Jersey je striktně rozděleno na klientskou a serverovou část. Pokud je například zapotřebí pouze funkcionalita klientské části, lze omezit velikost přilinkovaných knihoven na minimum. Kompletní distribuce se skládá z následujících modulů respektive knihoven:

- jersey-core - knihovna používaná oběma stranami jak serverovou, tak klientskou, obsahuje podporu pro SPI, rozhraní pro adresaci a implementace hlavních funkcionalit
- jersey-json - modul pro podporu JSON (JavaScript Object Notation)
- jersey-client - modul implementující klientské rozhraní
- jersey-server - modul implementující rozhraní serveru dostačující pro běh odlehčeného http (Hypertext Transfer Protocol) serveru
- jersey-servlet - modul pro podporu serverů

Server

Serverová část Jersey Frameworku [9] k vystavení REST webových služeb využívá kombinaci technologie servletů a tříd označených standardními JAX-RS anotacemi. Aplikace implementující serverovou část Jersey REST API musí běžet v nějakém aplikačním serveru např. Tomcat, Glassfish, atd.

Resource

Jak již bylo zmíněno výše, jádrem návrhu serverové části jsou třídy využívající JAX-RS anotace [7]. Takové třídy reprezentují návrh JAX-RS Web Resource,

[1] SPI - Service Provider Interface

který je implementován jako třída Resource, jejíž metody jsou určené ke zpracování http požadavků.

Třída Resource je standardní Javovská třída, která používá JAX-RS anotace k implementaci odpovídajícího webového zdroje. Třídy zdrojů jsou typu POJO (Plain Old Java Object), které mají alespoň jednu metodu označenou anotací @Path.

Hlavní JAX-RS anotace s popisem významu jsou v následující tabulce.

Anotace	Popis
@PATH(adresa)	Nastaví cestu k základně URL + /adresa. Základní URL je založeno na názvu aplikace, servletu a vzoru URL z konfiguračního souboru web.xml
@POST	Takto označená metoda obsluhuje požadavky http POST
@GET	Takto označená metoda obsluhuje požadavky http GET
@PUT	Takto označená metoda obsluhuje požadavky http PUT
@DELETE	Takto označená metoda obsluhuje požadavky http DELETE
@Produces(MediaType.TEXT_PLAIN[, další typy])	Tato anotace definuje, jaký MIME typ je vrácen metodou anotovanou @GET. Tyto typy mohou být např. „text/plain“, „application/xml“ nebo „application/json“
@Consumes(typ[, další typy])	Tato anotace definuje, jaký MIME typ je přijat danou metodou
@PathParam	Používá se pro mapování parametrů URL na parametry volané metody a umožňuje např. překlad identifikátoru na konkrétní objekt.

Tabulka 1: JAX-RS anotace (zdroj:[9])

Konkrétní servlet naslouchající na definovaném URL (Uniform Resource Locator) analyzuje příchozí http požadavek a vybere odpovídající třídu respektive metodu k jeho obsloužení.

Z pohledu životního cyklu instancí třídy Resource je systém defaultně nastaven tak, že pro každý příchozí požadavek se vytvoří nová instance. Veškeré požadované závislosti jsou řešeny pomocí DI (Depencency Injection). Poté se zavolá odpovídající metoda k obsloužení požadavku. Po úspěšném vyřízení požadavku je objekt zpřístupněn ke zpracování mechanizmem zvaným garbage collector.

Provider

Provider je třída implementující rozhraní JAX-RS, která je označena anotací @Provider pro její automatickou detekci systémem. Instance takto anotované třídy jsou vytvářeny jednou pro celou aplikaci, mohly by tedy být označovány jako „singleton“. Slouží ke zprostředkovávání kontextuálních dat a informací. Rozdělují se na dva hlavní typy:

- Entity provider - dále rozčleněn na:
    - Message body reader - definuje vztah mezi JAX-RS a komponentou, která poskytuje služby mapování parametrů http požadavku do odpovídajících Java datových typů; třídy, které poskytují tuto službu, mají implementovat rozhraní MessageBodyReader a jsou pak označeny anotací @Provider
    - Message body writer - má opačný význam než Message body reader, poskytuje mapování z návratového typu metody do parametrů resp. těla http odpovědi

Context provider - poskytuje kontext třídám Resource, popřípadě jiným poskytovatelům; implementuje rozhraní ContextResolver<T> a může být označen anotací @Provider pro automatické detekování, např. aplikace, která chce poskytnout vlastní JAXBContext poskytovatelům výchozích JAXB(Architecture for XML Binding) entit, musí implementovat ContextResolver<JAXBContext>

Klient

Klientská část reprezentuje JAX-RS Client API [9] pro komunikaci s REST webovými službami. Toto API je navrženo pro co nejjednodušší využití webové služby vystavené prostřednictvím protokolu http. JAX-RS client API může být využito jakoukoliv webovou službou založenou na protokolu http nebo nějakého jeho rozšíření např. WebDAV.

Jako rozšíření standardního JAX-RS Client API, Jersey Client API poskytuje možnosti použití různých základních implementací standardního rozhraní HTTP Client Connector. Součástí Jersey Frameworku je několik takových implementací.

Některá klientská rozhraní jako např. Apache HTTP Client nebo HttpURL-Connection může být poměrně obtížné používat, neboť produkují příliš mnoho kódu při relativně jednoduché funkcionalitě. To je důvod, proč Jersey poskytuje podporu pro zapouzdření HttpUrlConnection a Apache HTTP Client.

OSGi distribuce

Jednou z mnohých distribucí Jersey Frameworku [8] jsou i OSGi bundly. Tyto bundly lze zavést do komponentového kontejneru bez téměř jakýchkoli ostatních závislostí. Podpora OSGi byla přidána do Jersey od verze 1.2. Od té doby bylo možné používat standardní OSGi prostředí pro spuštění webových aplikací založených na Jersey. V současné době je Jersey kompatibilní se specifikací OSGi verze 4.2.0.

3.3. Restlet Framework

Technologie zmíněná v kapitole 3.2 není však jedinou svého druhu. Pro srovnání bude v dalším textu popsán, řekněme konkurenční přístup a tím je Restlet Framework [10]. Tato technologie není sice určena pouze pro OSGi, naopak se zaměřuje na co nejširší spektrum platforem, ale má pro OSGi velmi rozsáhlou podporu. Restlet je komplexní, škálovatelný, ale ve své podstatě jednoduchý a odlehčený RESTful Framework určený k tvorbě webových API rozhraní pro jazyk Java. Má relativně malé jádro, jeho komplexnost však zajišťuje velké množství rozšiřujících pluginů. Umožňuje implementovat architekturu webu (REST) a těžit z jeho jednoduchosti a rozšiřitelnosti. Využitím tohoto inovativního Frameworku lze jednoduše propojovat webové služby, webové stránky a webové klienty do jednotných webových aplikací. Podporuje všechny REST principy a je vhodný pro webové aplikace typu klient i server. Jsou zde také podporovány všechny komerčně rozšířené webové standardy včetně JSON, RSS (Rich Site Summary) a WADL (Web Application Description Language). Restletové aplikace lze integrovat s většinou používaných Frameworků jako jsou Spring, Guice, JAXB a JAX-RS. Mimo jiných výhod jsou ještě k dispozici speciální edice, které jsou průběžně synchronizovány pomocí automatizovaného procesu portování. Jedna z těchto

speciálních edic se zaměřuje právě na OSGi a tím spojuje výše zmíněné technologie, je proto jako jediná detailněji popsána v dalším textu.
V následujícím odstavci jsou zmíněny některé zajímavé vlastnosti a možnosti Restlet Frameworku.

- Je vhodný pro oba typy webových aplikací a to jak client-side (aplikační logika je vestavěna přímo ve webové stránce), tak server-side (aplikační logika je na serveru). Novinkou je možnost využití stejného Java API pro oba typy aplikací.
- Je zde také podporován koncept "URI as UI" založený na standardu URI (Uniform Resource Identifier) šablon. To má za následek velmi flexibilní a přesto jednoduché směrování s automatickou extrakcí URI proměnných do atributů http requestu.
- Tunelovací služba umožňuje prohlížeči používat jakoukoliv metodu http protokolu (PUT, DELETE, PATCH, atd.) prostřednictvím jednoduchého http POST. Tato služba je pro Restlet aplikace transparentní.
- Restlet poskytuje také interní webový server, který umožňuje využívat statických souborů podobně jako Apache HTTP Server ke sdružení metadat na základě přípon těchto souborů. Vzdálená manipulace se soubory pak probíhá za pomoci PUT a DELETE metod (aka režim mini-WebDAV). Dekódovací služba transparentně dekóduje komprimovaná nebo kódovaná vstupní data. Logovací služba zaznamenává všechny přístupy do aplikací ve standardním souboru webového protokolu. Formát protokolu odpovídá W3C Extended a formát souboru protokolu je plně přizpůsobitelný. Nechybí zde samozřejmě ani rozsáhlá a flexibilní podpora bezpečnosti jak pro autentizaci a autorizaci.

Jak bylo již zmíněno, tato technologie se snaží pojmout co nejširší spektrum platforem. Tento přístup se také neodmyslitelně týká prezentační a persistenční vrstvy. Technologie Restlet je otevřena všem prezentačním prostředím a technologiím, kterými jsou např. AngularJS, Android, iOS, Eclipse RCP, GWT (Google Web Toolkit), atd. Z pohledu persistence dat nezáleží na zvolené databázové platformě a lze použít nejrůznější technologie jako např. JDBC (Java

Data Base Connectivity), Hibernate, Spring IO, Cassandra, MongoDB, atd. Díky všem těmto opatřením nejsou Restlet aplikace závislé na konkrétní platformě a mohou být přenášeny z jednoho prostředí do druhého velmi jednoduše a bez jakýchkoli omezení.

Speciální edice

Této platformní nezávislosti napomáhá velké množství speciálních edicí [11]. V současné době je Restlet Framework k dispozici v několika verzích. Vydání pro Java SE (Java Second Edition) je specifické spouštěním Restlet aplikací ve standardní JVM. Vydání pro Java EE se naopak zaměřuje na provoz Restlet aplikací v servletových kontejnerech. Edice GAE nasazuje Restlet aplikace do Google App Engine cloud platformy. Edice pro GWT umožňuje spustit aplikace ve webovém prohlížeči a to bez pluginů. Vydání pro Android umožňuje nasazení aplikací na mobilní zařízení se systémem Android. A samozřejmě edice pro OSGi, která umožňuje nasazení Restlet aplikací v tomto dynamickém prostředí, je blíže popsána níže.

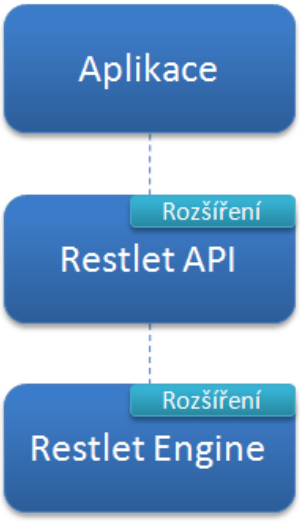
Na následujícím diagramu je zobrazen průběh vydávání nových verzí a různých distribucí s použitím automatizovaného Restlet Forge včetně specializovaných javadocs a speciálních edicí.

Obrázek 4: Restlet Forge[1]

[1] Převzato z [11]

Architektura

Restlet Framework [10] se skládá ze dvou hlavních částí. Jako první je zde Restlet API, neutrální API podporující REST a http. Usnadňuje zpracování komunikace na obou stranách jak na straně klienta, tak na straně serveru. Toto rozhraní je navíc podpořeno Restlet enginem, který je spolu s ním dodáván v jednom balíčku "org.restlet.jar".



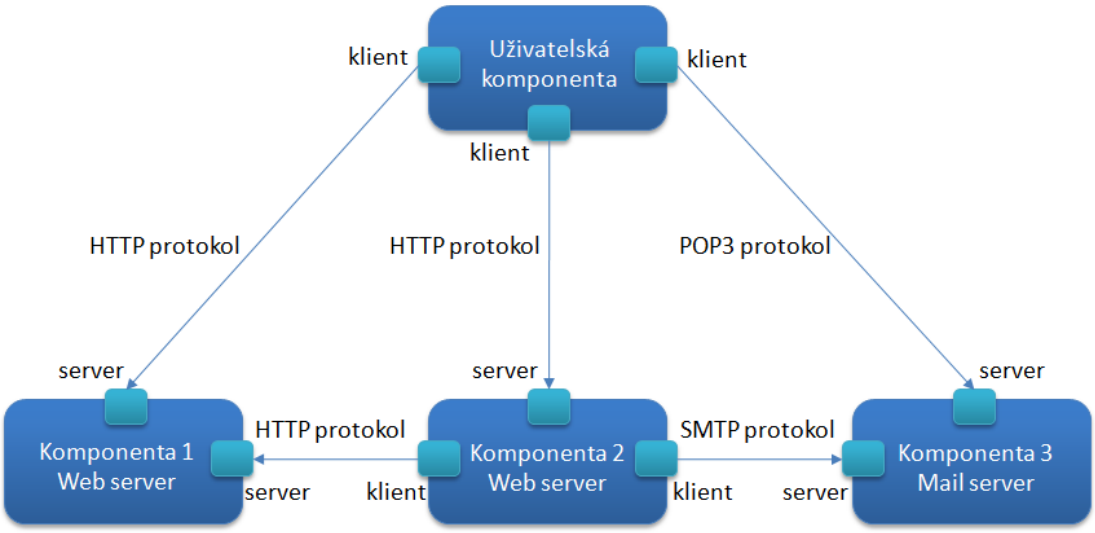
Obrázek 5: Model Restlet Frameworku[1]

Separováním API od implementace je dosažen stejný efekt jako u Servlet API a webových kontejnerů jako Jetty nebo Tomcat, nebo mezi JDBC API a konkrétními ovladači JDBC.

REST architektura

Při abstrakci o úroveň výše a úvaze typické webové architektury z hlediska REST lze popsat architekturu následujícím grafickým znázorněním. Tzv. konektory jsou znázorněny jako čtverce navzájem propojené hranou. Umožňují komunikaci mezi komponentami, které jsou reprezentovány bloky s příslušným označením. Hrany pak představují protokol (http, SMTP - Simple Mail Transfer Protocol, atd.) použitý pro vlastní komunikaci.

[1] Převzato z [10]



Obrázek 6: Model Rest architektury[1]

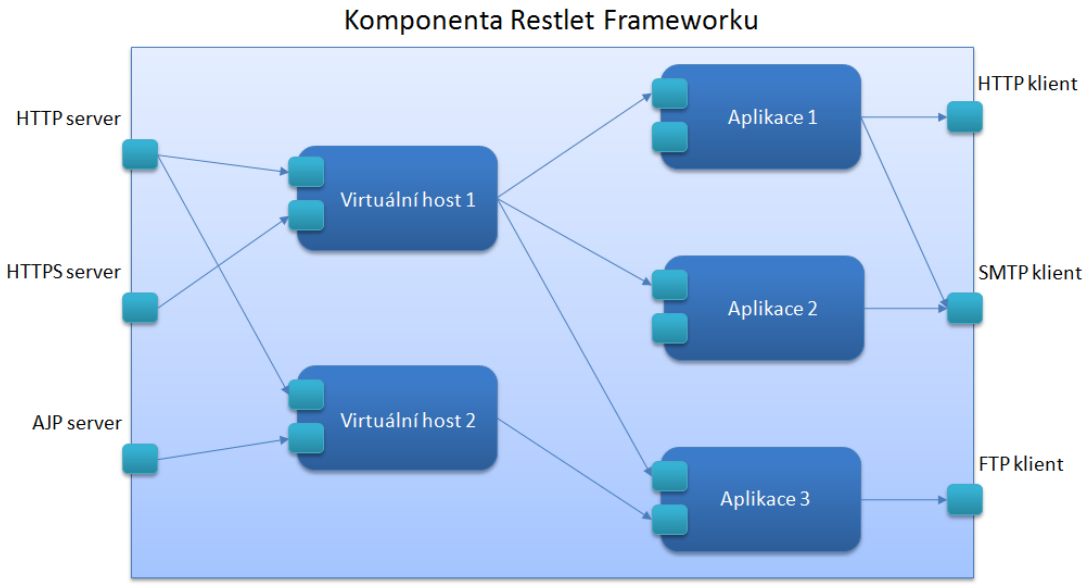
Z diagramu je patrné, že jedna komponenta může mít libovolný počet připojených klientských či serverových konektorů. Například komponenta 2 má serverový konektor přijímající požadavky z uživatelské komponenty a též klientské konektory pro odesílání požadavků na komponenty 1 a 3.

Restlet architektura

Kromě podpory standardních prvků REST architektury, jak je uvedeno výše, Restlet Framework také poskytuje sadu tříd, které značně zjednodušují hostování více aplikací v rámci jedné JVM. Cílem je poskytnout snáze přenositelné a flexibilnější REST rozhraní jako alternativu k již existujícímu Servlet API. Komponenta Restlet Frameworku zobrazená na diagramu níže se mírně liší od komponenty REST architektury. Jedna komponenta totiž může spravovat několik virtuálních hostů i aplikací.

Entita „Virtual host“ podporuje flexibilní konfiguraci, ve které může být jedna IP adresa sdílena několika doménovými jmény a naopak jeden název domény reprezentován několika IP adresami k docílení load-balancingu. Aplikace jsou, jak bylo avizováno, přenositelné a konfigurovatelné pro různé Restlet implementace a různá virtuální prostředí. Restlety také poskytují důležité služby jako je logování přístupu, automatické dekódování requestů, konfigurovatelné nastavení prezentační vrstvy, atd.

[1] Převzato z [10]



Obrázek 7: Komponenta Restlet Frameworku[1]

Na diagramu je zobrazena Restlet komponenta, jí byl přidělen jeden http serverový konektor, který naslouchá na konkrétním portu a obsluhuje http komunikaci. Ten je dále vnitřně propojen s virtuálním hostem komponenty, jenž může být společný pro více konektorů. Virtuální host pak přesměruje komunikaci příslušné aplikaci.

OSGi speciální edice

Zpočátku bylo OSGi pro Restlet [11] sice důležitou platformou, ale využívalo se spíše jen jako běhové prostředí pro jeho aplikace. Průkopníky této spolupráce byly ojedinělé inovativní projekty v čele s NASA JPL, které napomohly nárůstu zájmu o OSGi v posledních letech. Ale již od verze 1.1 je každý Restlet modul distribuovaný jako JAR soubor, zároveň platným OSGi bundlem a jeho manifest je rozšířen o povinné atributy a popisné informace OSGi bundlu. Kromě jiného velký vývoj zaznamenal také nový distribuční kanál specifický pro OSGi edici. Oblíbené vývojové prostředí Eclipse nově podporuje snadnou instalaci a automatickou aktualizaci Restlet aplikací i díky jeho stránkám s aktualizacemi. Tento distribuční kanál je doporučován tehdy, je-li vývoj zaměřen právě na OSGi aplikace využívající Restlet.

[1] Převzato z [10]

4. Úložiště softwarových artefaktů

V této kapitole budou analyzovány některé implementace úložišť softwarových komponent, které jsou z pohledu této práce významné. Úložiště standardu OBR (OSGi Bundle Repository) zaštiťované OSGi aliancí je významné kvůli svému zaměření na OSGi komponenty. Úložiště Maven artefaktů vyvíjené nadací Apache je komerčně nejrozšířenější, a proto bylo zařazeno ke srovnání s ostatními. A CRCE úložiště je zařazeno z důvodu provázanosti s touto prací. Klientská aplikace vyvíjená v rámci práce je sice schopná pracovat i s OBR, ale primárně je zaměřena právě na CRCE.

4.1. OBR

OSGi Bundle Repository zkráceně OBR [12] je úložiště OSGi komponent. Jeho návrh je postaven na datovém modelu, kterým lze vhodně popsat uložené komponenty i s jejich specifickými vlastnostmi a vzájemnými vazbami. Takový datový model také samozřejmě podporuje efektivní vyhledávání komponent na základě jejich metadat. Nad datovým modelem je postavena logika umožňující rozsáhlé množství funkcí, např. uživatelům OBR úložiště usnadňuje instalaci a nasazování komponent bez nutnosti manuálního řešení jejich závislostí, konzistenci zaručuje úložiště automaticky.

4.1.1. Formát popisných metadat

OBR úložiště jsou postavena nad běžným souborovým systémem, v jehož adresářích jsou uloženy jednotlivé komponenty. Celá tato struktura komponent je popsána jedním XML (Extensible Markup Language) souborem umístěným v kořenovém adresáři. Tento XML soubor má pevně specifikovaný název „repository.xml“, který čtou klientské aplikace a extrahují z něj potřebné informace k získání potřebných komponent. Obsahem tohoto souboru jsou metadata, jejichž nejnovější formát definuje OSGi Enterprise R5 Repository service specification implementovaná např. OBR Apache Felix 4.0 [12].

Kořenovým elementem tohoto XML je element „repository“. Obaluje celý obsah úložiště a jeho nejvýznamnějšími atributy jsou datum poslední změny a název

úložiště. Dále jsou vnořeny elementy typu „resouce“, které reprezentují jednotlivé komponenty v úložišti. Každé komponentě odpovídá jeden element „resource“, který ji detailně popisuje a to včetně deskriptivních informací o její fyzické reprezentaci čítající velikost souboru či URI pro stažení. Kromě těchto popisných informací jsou zde ještě další vnořené elementy definující vazby komponenty na okolí. Spadají sem elementy „capability“ a „requirement“. Element „capability“ určuje schopnosti dané komponenty, které může ostatním poskytovat, bývají nejčastěji reprezentovány programovými balíčky. Opakem schopností jsou požadavky definované pomocí elementů „requirement“, které určují, na jakých zprostředkovaných schopnostech komponenta závisí. Vyžadované subelementy či atributy elementů „capability“ jsou vyjádřeny pomocí LDAP (Lightweight Directory Access Protocol) filtrů, kterými se specifikuje například konkrétní verze či název dané schopnosti.

Ukázkový příklad repository.xml:

```
<repository lastmodified='20131018122615.159'
    presentationname='sampleRepository'
    symbolicname='cz.sample.repository'>
    <resource id='cz.sample.resource-1.0'
        symbolicname='cz.sample.resource' uri='http.//cz.sample.resource'>
        <description>Sample bundle</description>
        <size>1024</size>
        <documentation>
            http://cz.sample.resource.doc.html
        </documentation>
        <category id='service' />
        <capability name='package'>
            <p n='package' v='cz.sample.resource.pcg' />
            <p n='version' t='version' v='1.0.0' />
        </capability>
        <requirement name='package'
            extend='false' multiple='false' optional='false'
            filter='(&amp;
                    (package=cz.sample.resource.pcg)
                    (version&gt;=1.0.0))'>
            Import package cz.sample.resource.pcg
        </requirement>
    </resource>
</repository>
```

4.1.2. Logika úložiště

Ukázalo se, že takto popsané úložiště je schopno poskytovat svým uživatelům podporu nasazování komponent bez nutnosti řešení jejich závislostí. Tuto funkci totiž zprostředkovává tzv. „resolver“, jehož základním stavebním kamenem jsou výše zmíněné dva elementy „capability“ a „requirement“. Pokud nejsou naplněny všechny požadavky dané komponenty je resolver schopen dohledat v úložišti takové komponenty, které požadované schopnosti zprostředkovávají a vyřešit tak všechny závislosti. Díky tomuto mechanismu běžícímu nad Frameworkem klientského prostředí není tento proces na zodpovědnosti uživatele.

Dalším objektem na straně klienta je tzv. správce úložiště (Repository Admin). Ten umožňuje slučovat více fyzických úložišť umístěných na různých serverech do skupin a pro uživatele je reprezentovat a přistupovat k nim jako k jednomu.

4.1.3. Implementace OBR

Jelikož OBR je vlastně jen další částí specifikace definující OSGi, musely nutně vzniknout některé její implementace, které by tento návrh přenesly do reálného zpracování. V současné době je používáno několik hlavních implementací.

Jednou z nich je Apache Felix OBR [12], které navazuje na implementaci běhového prostředí Apache Felix. Jedná se o komplexní implementaci návrhu OBR, do kterého vnáší řadu individuálních vylepšení. Jelikož se však jedná o externí implementaci mimo OSGi alianci, může do budoucna docházet k desynchronizaci s oficiálním rozhraním, čemuž se bude společnost Apache snažit zabránit. Nad Felix OBR bylo vybudováno mnoho nadstaveb, které ještě více zpřístupňují uživatelům jeho používání. Patří sem např. RemoteOBR (více logiky přeneseno na server, menší nároky na klienta) nebo Maven Bundle Plugin (umožňuje přistupovat k Maven úložišti jako k OBR).

Další implementací je Knopflerfish OBR, která je provozována společností Makewave, ale její licence je OpenSource[1]. Taktéž navazuje na implementaci

[1] Jedná se o Open Source licenci podle šablony The BSD 3-Clause License blíže: http://opensource.org/licenses/BSD-3-Clause

OSGi Knopflerfish, stejně jako je to v případě Apache Felix. Nejnovější verze 5 je v souladu s OSGi R5 specifications.

4.2. Maven

Maven [13] je nástroj vyvíjený společností Apache pro správu softwarových projektů. Výhodou Mavenu je možnost používat nepřeberné množství pluginů použitých např. při sestavování projektu. Díky tomuto nástroji lze z projektu sestavit JAR soubor nebo WAR (Web Application Resource), či vygenerovat třídy pro JAXB (práce s XML) nebo pro JAXWS (webové služby) a to až v průběhu sestavování. Dalším mocným mechanismem je práce s knihovnami. Stačí jen na určitou knihovnu či modul aplikace přidat závislost a Maven je automaticky přidá k projektu. Tyto knihovny se dají označit také jako tzv. „provided“, což znamená, že se nepřibalí k výsledku sestavení, ale musí být zajištěno, že budou dostupné v prostředí, do kterého se bude projekt nasazovat. Mimo jiné Maven podporuje rozdělování aplikací do jednotlivých modulů, které lze pak sestavit v předem určeném pořadí podle jejich závislostí. Všechna tato nastavení jsou definována v souboru pom.xml, jenž se nachází v kořenovém adresáři každého Maven projektu a je založen na objektovém modelu POM (Project Object Model).

Každý z projektů využívajících Maven je označován jako Maven artefakt a nemusí se jednat pouze o Java projekty. Takový artefakt je jednoznačně identifikován svými „groupId“, „artifactId“ a „version. Tyto údaje se využívají pro linkování externích zdrojů (Maven projektů), které se nacházejí v úložišti Maven artefaktů.

Ukázka pom.xml:

```
<project>
    <groupId>cz.zcu.kiv.guica</groupId>
    <artifactId>update-manager</artifactId>
    <version>1.2.0-SNAPSHOT</version>
    <name>GUICA update manager</name>
    <packaging>bundle</packaging>

    <dependencies>
        <dependency>
            <groupId>org.osgi</groupId>
```

```
            <artifactId>org.osgi.core</artifactId>
            <version>4.2.0</version>
        </dependency>
        <dependency>
            <groupId>cz.zcu.kiv.guica</groupId>
            <artifactId>support</artifactId>
            <version>1.2.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>com.sun.jersey</groupId>
            <artifactId>jersey-client</artifactId>
            <version>1.17.1</version>
        </dependency>
    </dependencies>

    <url>http://www.assembla.com/code/guica</url>
</project>
```

4.2.1. Úložiště Maven artefaktů

Maven úložiště se v mnohém podobá již zmiňovanému OBR, má však také svá specifika. Maven ke své funkci využívá dva typy úložišť.

Jedním z nich je tzv. lokální úložiště (Local repository). To je umístěno na zařízení, které je určené ke spouštění a využívání Mavenu. Záleží na individuální konfiguraci, ale defaultním umístěním tohoto úložiště je složka s názvem „.m2“ nacházející se v adresáři přihlášeného uživatele. Ukládají se sem artefakty vzniklé při sestavování projektů s příznakem „install“, což znamená „Nainstaluj vzniklý artefakt do úložiště“. Projekt lze totiž sestavit, aniž by se uložil do lokálního úložiště. Určuje to příznak „package“ a artefakt vznikne pouze v adresáři „terget“ v daném projektu. Dále se do lokálního úložiště stahují artefakty ze vzdáleného úložiště, které jsou vyžadovány pro vyřešení závislostí při sestavování.

Druhým typem je vzdálené úložiště (Remote repository). Toto úložiště běží na nějakém vzdáleném serveru a je většinou veřejně přístupné pro vyhledávání a získávání artefaktů. Pokud není nějaký artefakt či jeho konkrétní verze nalezena v lokálním úložišti, Maven se ji pokusí dohledat v některém z předem nakonfigurovaných vzdálených úložišť. Veřejných Maven úložišť je totiž velká řada, některé jsou oficiální jako např. http://central.maven.org/maven2/, a některé méně jako např. https://nexus.bitexpert.net. Pro veškeré použití Mavenu nebo pro konkrétní projekt lze nastavit seznam repositářů a jejich

obrazů, ve kterých se budou artefakty vyhledávat. Teoreticky by měly být artefakty ve vzdálených repositářích uloženy persistentně, protože nikdy není zaručeno, že i tu nejzastaralejší verzi nějaké nepopulární knihovny nevyužívá nějaký běžící projekt. V praxi to však není pravidlem, mluvím z vlastní zkušenosti. Pro použití artefaktu Mavenem se tento však musí nejprve stáhnout do lokálního úložiště, ze kterého se pak načte.

4.2.2. Fyzická reprezentace artefaktů

Maven také na rozdíl od OBR standardně nepoužívá k popisu obsahu svého úložiště soubor repository.xml. V některých případech se však tento soubor v kořenovém adresáři úložiště může objevit. Je to díky Maven Bundle Pluginu, který dovoluje používat Maven repositář jako OBR, pokud jsou v něm uloženy OSGi bundly vzniklé tímto pluginem. Ty jsou pak přístupné regulérním OBR způsobem.

Standardní přístup Mavenu k uloženým artefaktům je však jiný. Maven vyhledává artefakty přímo v adresářové struktuře, která je optimalizována pro jeho účely. Slouží k tomu identifikátory artefaktů, jejichž označení bylo zmíněno výše. Jsou to tedy „groupId“ (označující skupinu projektů či organizaci), „artifactId“ (určující název artefaktu) a „version“ (reprezentující verzi artefaktu). Maven umisťuje soubory artefaktů do stromové struktury od „groupId“ až po konkrétní verze, které se stávají listy tohoto stromu obsahujícími konkrétní artefakty.

Např.

Definice Maven artefaktu:

```
<groupId>cz.zcu.kiv.guica</groupId>
<artifactId>bundle-state</artifactId>
<version>1.2</version>
```

Výsledné umístění v adresářové struktuře:

```
.m2\repository\cz\zcu\kiv\guica\bundle-state\bundle-state-1.2.jar
```

V koncových složkách bývají kromě samotných artefaktů jejich popisné soubory s názvem a verzí artefaktu pouze s jinou příponou a to *.pom. Tyto soubory

obsahují stejné informace jako původní pom.xml, které jsou součástí projektů. Dále se mohou v podsložkách artefaktu, kde jsou složky jednotlivých verzí, objevovat soubory typu maven-metadata-local.xml obsahující doplňující informace pro rychlejší vyhledávání.

4.3. CRCE

Pro účely aplikací, které by bylo možno aktualizovat za běhu díky OSGi, je nutné mít jistotu, že nová verze aktualizované komponenty negativně neovlivní své okolí či dokonce nezpůsobí pád aplikace. V některých případech totiž není vhodné spoléhat se pouze na kompatibilitu na úrovni rozhraní (importované a exportované balíčky). Někdy je pro stoprocentní jistotu zapotřebí mnohem sofistikovanější mechanismus určování kompatibility komponent, který však nenabízí komerční komponentová úložiště. Z tohoto důvodu byl na Katedře informatiky Západočeské univerzity odstartován projekt s názvem CRCE.

CRCE (Component Repository supporting Compatibility Evaluation) [14][17] je úložiště komponent s podporou určování kompatibility komponent. Jeho návrh je do značné míry ovlivněn OBR, také z důvodu zaměření na OSGi. Také používá XML metadata k popsání komponent a jejich vzájemných vazeb a závislostí, jenž vycházejí z návrhu OBR. CRCE metadata jsou však obohacena o detailní a velmi sofistikované informace o kompatibilitě komponenty vůči okolí a jejích jednotlivých verzích mezi sebou.

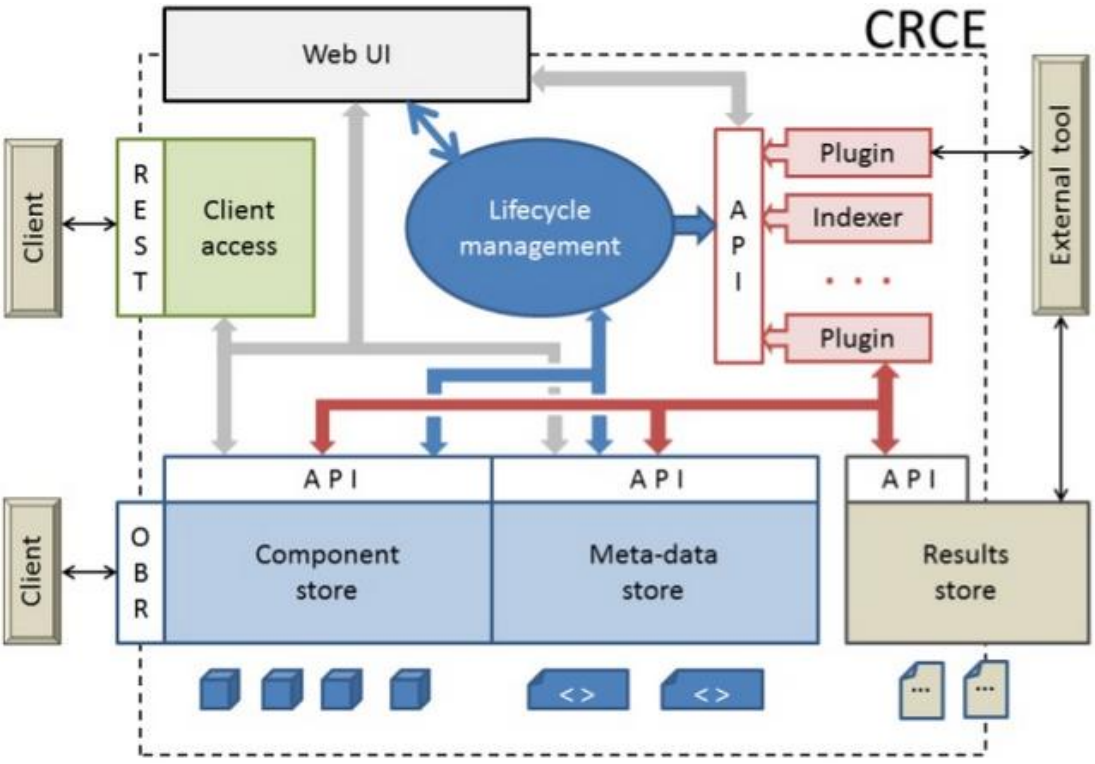
Většina komerčních úložišť slouží pouze k persistenci komponent, softwarových artefaktů či obecných datových souborů a ověřování kompatibility je zodpovědností klientských aplikací. Klient po získání komponenty ověří, zda je možno její starší verzi nahradit a pokud ne, hledá se jiná alternativa nebo se aktualizace neprovede. Zde se projevuje revoluční přístup CRCE, které provádí ověřovaní kompatibility na své straně. Tento přístup přináší velké množství výhod:

- optimalizace využití přenosového pásma - klient není nucen se vícenásobně dotazovat po zjištění nekompatibility, server vrátí kompatibilní komponentu nebo zprávu, že žádná nebyla nalezena

- optimalizace výpočetního výkonu při ověřování kompatibility - server má všechny verze všech komponent u sebe, díky tomu s nimi může operovat snáze a rychleji
- minimalizace nároků na klientské zařízení - toto řešení je vhodné pro mobilní či embedded zařízení s určitými výkonnostními omezeními

4.3.1. Architektura

CRCE úložiště je koncipováno jako modulární aplikace založená na OSGi technologii s podporou Apache Maven. Mimo jádra systému, které je tvořeno několika moduly, je zde ještě řada pluginů určujících funkcionalitu úložiště.



Obrázek 8: Architektura CRCE úložiště[1]

Jelikož na téma CRCE úložiště bylo zpracováno více dokumentů, viz [14][15], není nutné zde detailně popisovat funkci a účel jednotlivých částí systému. Místo toho bude důraz kladen na ty oblasti, které jsou z pohledu této práce významné, a které bude klientská aplikace reálně využívat. Jsou to oblasti týkající se metadat, vlastního úložiště a REST rozhraní.

[1] Převzato z CRCE prezentace EuroMicro, Přemysl Brada

4.3.2. Úložiště

Proces uložení komponenty do CRCE úložiště není tak jednoduchý, jako tomu bývá u ostatních úložišť. Pro tento účel slouží webové uživatelské rozhraní a OBR rozhraní. Po vložení komponenty do úložiště oběma způsoby se uloží nejprve do tzv. Bufferu. V bufferu může být komponenta podrobena různým testům, které se týkají kompatibility a je zde opatřena doplňujícími CRCE metadaty. Dále následuje commit komponenty do vlastního úložiště, z něhož je pak veřejně dostupná. Samozřejmě jsou podporovány běžné operace, jako např. smazání.

4.3.3. Formát popisných metadat

Formát metadat CRCE úložiště je silně ovlivněn metadaty používanými v OSGi OBR, jak bylo již předesláno. OBR metadata jsou popsána v kapitole 4.1.1, tudíž by bylo zbytečné je znovu zmiňovat, proto zde budou uvedena pouze metadata, která přidává ke komponentě CRCE úložiště. Rozšíření, která CRCE přináší, jsou rozdělena na několik základních oblastí podle [14]. Nejzásadnější jsou „core“ (tzv. jádro metadat popisujících komponentu) a „compatibility“ (metadata o kompatibilitě komponent). Tato rozšíření jsou určena jmenným prostorem (xmlns) definovaným pomocí XML šablony tzv. XSD (XML Schema Definition) veřejně dostupné zde: http://relisa-dev.kiv.zcu.cz/schema/crce/metadata/1.0.0 CRCE v aktuální podobě navazuje na starší verzi specifikace OBR metadat OSGi RFC-0112. Připravuje se ale nová verze, která je již plně kompatibilní s OSGi Enterprise R5 Repository service specification.

Core metadata

CRCE core metadata [16] rozšiřují některé elementy OBR o doplňující atributy, které jsou uvozené „crce:*“ nebo přidávají dodatečné elementy „attribute“ do elementů „capability“, „requirement“ a „property“. CRCE také přináší možnost rekurzivně deklarovat vnořené výše uvedené tři typy elementů pro jejich hierarchické vyjádření.

Následují okomentované fragmenty XML metadat primární identifikace komponenty v CRCE úložišti a příklad vnořování elementů.

Element capability rozšířený o specifická CRCE metadata.

```
<capability namespace='crce.identity'
    uuid='550e8400-e29b-41d4-a716-446655440000'>
    <attribute name="name"
        value="obcc-parking-example.gate-2.0.0" />
    <attribute name="crce.type" value="osgi,jar"
        type="list" />
    <attribute name="provider" value="cz.zcu.kiv" />
    <attribute name="version.original"
        type="Version" value="2.0.0" />
    <attribute name="crce.categories"
        value="initial-version,versioned,osgi"
        type="list" />
</capability>
```

Element capability s dalšími dvěma vnořenými elementy téhož typu.

```
<capability namespace='ab.component'>
    <attribute name='name' value='someComponent' />
    <capability namespace='ab.component.bean'>
        <attribute name='name'
            value='com.ab-software.example.BeanClass' />
        <capability namespace='ab.component.bean.interface'>
            <attribute name='name' value='svc1' />
            <attribute name='type'
                value='com.example.Interface' />
        </capability>
    </capability>
</capability>
```

Metadata kompatibility

Zde se jedná o zcela nový přístup k popisování uložených komponent, který není v OBR vůbec postižen. Detailní informace o kompatibilitě [16] zprostředkovává CRCE přidáním vlastnosti se jmenným prostorem „crce.compatibility“. Takto označená vlastnost obsahuje nejprve jeden nebo více elementů „s:compatibility“, které určují, vůči jaké komponentě potažmo verzi se má kompatibilita určovat a na jaké aspekty je toto určování zaměřeno. Lze totiž určovat kompatibilitu na úrovni syntaxe či funkcionality komponent. Dále mohou následovat ještě elementy „s:diff“, které definují podmínku kompatibility.

Následují okomentované fragmenty XML metadat jedné komponenty v CRCE úložišti. Jedná se o modul Gate vzorové výukové aplikace OBCC Parking, ve které se studenti ZČU seznamují s OSGi.

```
<repository name='CRCE Repository'
    increment='13582741' xmlns="TBD-CRCE-METADATA-XSD-URI">

    <resource crce:id='obcc-parking-example.gate-2.0.0'
        uuid="334232">
        <property namespace="crce.compatibility">
            <attribute name="predecessor-version"
                value="1.2.6" type="Version" />
```

Atributem s názvem "predecessor-version" se udává konkrétní verze té samé komponenty, proti které má být kompatibilita určována.

```
            <s:compatibility
                base-version="1.2.6" contract="syntax"
                diff="MUT">
```

Element s:compatibility určuje úroveň, ve které se bude kompatibilita určovat a atributem diff je možné specifikovat, o jakou změnu se jedná. Typy změn mohou být následující:

- MUT - Mutation
- GEN - Generalization
- SPE - Specialization
- INS - Insertion
- DEL - Deletion

```
                <s:diff level="package" role="capability"
                        namespace="osgi.wiring.package"
                        name="cz.zcu.parking.gate.stats"
                        syntax="java" value="INS" />
```

Element s:diff definuje kompatibilitu na úrovních balíčků (level="package"), tříd (level="type") či dokonce jednotlivých metod (level="operation"). Díky možnosti rekurzivního vnořování elementů s:diff je také možné simulovat reálnou hierarchickou strukturu komponenty (balíček -> třída -> metoda).

```
<s:diff level="package" role="capability"
                    namespace="osgi.wiring.package"
                    name="cz.zcu.kiv.obcc.parking.gate.base"
                    syntax="java" value="GEN">
                    <s:diff level="type"
                        name="cz.zcu.parking.gate.base.SomeClass"
                        value="GEN">
                        <s:diff level="operation" name="myConstructor"
                            value="GEN" />
                    </s:diff>
                </s:diff>
            </s:compatibility>
            <s:compatibility
                base-version="1.2.6" contract="extra-functional"
                diff="GEN" />
            <s:compatibility
                base-version="1.2.0" contract="syntax"
                value="SPE" />
        </property>
    </resource>

</repository>
```

5. Klienti úložišť

Aby mohl být naplněn cíl práce implementace klientské aplikace, bylo zapotřebí nejdříve prozkoumat různé typy již existujících klientů. Pozornost byla věnována hlavně klientským aplikacím, které jsou založené na podobných principech jako úložiště zmíněná v předchozí kapitole a pracují s komponentami, pluginy či softwarovými balíčky. Tato kapitola proto obsahuje popis přístupu různých klientských aplikací a jejich závěrečné porovnání.

5.1. IDE klienti pro pluginy

Většina integrovaných vývojových prostředí (IDE - Integrated Development Environment) má možnost uživatelského přizpůsobení pomocí různých pluginů či rozšiřujících balíčků. Kvůli uživatelské přívětivosti a odstranění nutnosti každý plugin zvlášť stahovat z nějaké webové stránky a až poté ho nainstalovat, existují servery, které obsahují většinu rozšíření daného IDE. Těmto serverům přísluší klientské aplikace, které jsou součástí IDE a mají za úkol zjednodušit proces získávání dodatečného obsahu a některé také řeší vzájemné závislosti těchto rozšíření i na úrovni různých verzí.

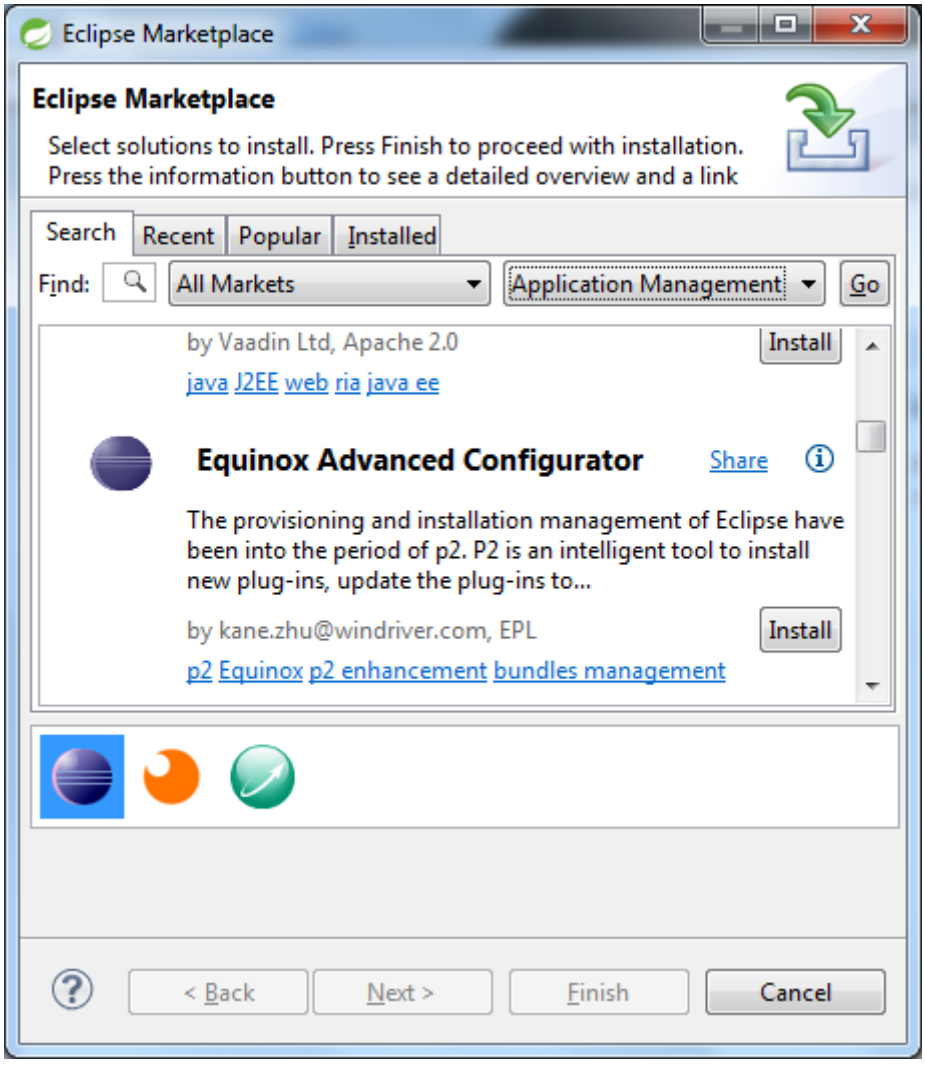
5.1.1. Eclipse

Vývojové prostředí Eclipse je jedním z nejpoužívanějších vývojových prostředí pro platformu Java, ale existují i verze pro jazyk C a další. Eclipse je vyvíjen The Eclipse Foundation.

Nadace Eclipse si výše nastíněný problém uvědomuje a kvůli těmto účelům vytvořila EPP (Eclipse Packaging Project) [18]. Tento projekt se zaměřuje na stahování pluginů na základě definovaných uživatelských profilů. Poskytuje platformu, která umožňuje vytváření balíčků (zip/tar) z aktualizačního webu. Jádrem technologie projektu je možnost vytvoření balíčků, které vznikají pomocí sdružování funkcí Eclipse z jednoho nebo více aktualizačních webů Eclipse. EPP je určeno pro všechny typy Eclipse prostředí a to Java Developer, Java EE Developer, C/C++ Developer a RCP Developer. Součástí projektu je instalátor, který zlepšuje proces instalace na základě uživatelských zkušeností.

Jednou z nejvýznamnějších aplikací EPP je tzv. Eclipse Marketplace[19]. Jedná se vlastně o webový katalog, díky kterému mohou uživatelé jednoduše získávat nové pluginy a rozšíření. V samotném vývojovém prostředí je integrován tzv. MPC (Marketplace Client), který je rozhraním pro procházení a instalaci nových balíčků a pluginů. Komunikace klienta se serverem probíhá přes REST rozhraní [20], které je ale ještě ve vývoji. Je sice funkční, ale může se často měnit. Obsahuje operace k získání seznamu všech dostupných katalogů, kategorií a nejvyhledávanějších či nejnovějších rozšíření.

Klientská aplikace dokáže navíc řešit vzájemné závislosti jednotlivých rozšíření. Pro správnou funkci je někdy zapotřebí kompatibilní verze nějakého pluginu. MPC je schopen tyto závislosti správně vyhodnotit a popřípadě zamítnout pokus a aktualizaci na úkor ostatních rozšíření.



Obrázek 9: Eclipse Marketplace

5.2. Linux balíčkovací systém

Operační systém Linux podporuje rozšiřování a aktualizaci pomocí tzv. balíčků [23]. Tyto balíčky obsahují samotné aplikace, různé pluginy či knihovny. Různé distribuce Linuxu mají někdy odlišný přístup k této problematice, ale princip balíčků zůstává u všech stejný.

Obecným návrhem řešení tohoto problému je systém pro správu balíčků, také zvaný manažer balíčků. Je to skupina softwarových nástrojů pro automatizaci procesu instalace, upgrade, konfigurace a odstraňování programových balíčků z operačního systému konzistentním způsobem. Tyto balíčky bývají uloženy v databázi s popisem závislostí a informacemi o verzi softwaru, aby se zabránilo nesouladu a nevyřešeným závislostem. Z tohoto důvodu obsahují balíčky také metadata jako je jméno balíčku, popis jeho účelu, číslo verze, distributora, kontrolní součet a seznam závislostí potřebných pro správnou funkci software. V následujících podkapitolách jsou znázorněny formáty dvou nejpoužívanějších typů balíčků pro OS Linux. Dále pak následuje výčet některých používaných klientských aplikací pro správu balíčků.

5.2.1. Struktura balíčků systému Debian

Balíčky určené pro OS Linux založené na distribuci Debian [24] mají svou specifikou strukturu a metadata sloužící k jejich detailnějšímu popisu. Tyto balíčky jsou označovány příponou *.deb.

Ve struktuře balíčků pro Debian jsou zakomponovány konfigurační soubory použité pro instalaci balíčku, kontrolní součet, samotná binární data, licence, ale hlavně popisná metadata, jenž jsou uložena v souboru „control“. Struktura ukázkového balíčku je přehledně zobrazena na screenshotu níže.

Obrázek 10: Struktura DEB balíčku

Metadata obsažená v balíčku popisují jak software samotný, tak i jeho vztahy a závislosti na okolí. Jádro metadat tvoří následující informace:

- Package - skutečný název balíčku, může se lišit od názvu souboru *.deb
- Source - zdrojový balíček, po jehož sestavení tento vznikl
- Version - verze balíčku
- Architecture - pro jakou hardwarovou architekturu byl balíček určen

Metadata mohou dále obsahovat podrobnější popis balíčku (Description), velikost po nainstalování (Installed-Size), atd.

Samostatnou částí metadat z pohledu jejich významu tvoří definice vztahů s ostatními balíčky a ty jsou následující:

- Depends - určuje absolutní závislost na jiných balíčcích, a to i s určením konkrétního rozsahu verzí např. python (>=2.3), balíčky zde uvedené musí být nainstalovány spolu s tímto nebo předem
- Pre-Depends - stejné jako Depends s tím rozdílem, že balíčky zde uvedené musejí být nainstalovány jedině předem
- Recommends - přítomnost uvedených balíčků je doporučena, ne však vyžadována
- Suggests - pouhý návrh dalších balíčků neovlivňujících funkcionalitu

- Enhances - návrh dalších balíčků obohacujících sekundární funkcionalitu
- Breaks - zde jsou deklarovány nekompatibilní balíčky, obvykle ale nějaká konkrétní verze
- Conflicts - zde jsou deklarovány absolutně nekompatibilní balíčky, tudíž jakákoli verze, a musí být odstraněny pro správnou instalaci
- Replaces - některé soubory zde uvedených balíčků jsou při instalaci daného balíčku nahrazovány
- Provides - daný balíček poskytuje všechny soubory a funkce uvedeným balíčkům

5.2.2. Struktura balíčků RPM

Balíčky RPM (Red Hat Package Management) [25] jsou určeny pro OS Linux založený na distribuci Red Hat. Soubory RPM balíčků jsou označovány příponou *.rpm. Mají také odlišnou strukturu a formát metadat, např. oproti výše zmíněnému DEB formátu balíčků.

Struktura RPM balíčku vypadá následovně:

- Identifier - někdy též označován jako lead nebo rpmlead, souží k označení RPM balíčků, obsahuje verzi použitého RPM formátu a informace o cílové architektuře
- Signature - slouží k ověření integrity balíčku a optimálně i k ověření autenticity, je to většinou řetězec získaný hash funkcí (MD5, SHA) aplikovanou na části Header a Payload
- Header - metadata určená k detailnějšímu popisu, blíže popsána v dalším textu
- Payload - někdy též nazýván jako archive, obsahuje skutečné soubory používané v balíčku, jedná se o soubory, které příkaz rpm při instalaci balíčku nainstaluje, data v archivu jsou často komprimována do GNU formátu gzip.

Sekce označovaná jako hlavička (Header) obsahuje metadata, která jsou velmi podobná těm v DEB balíčcích. RPM metadata se dělí do následujících skupin:

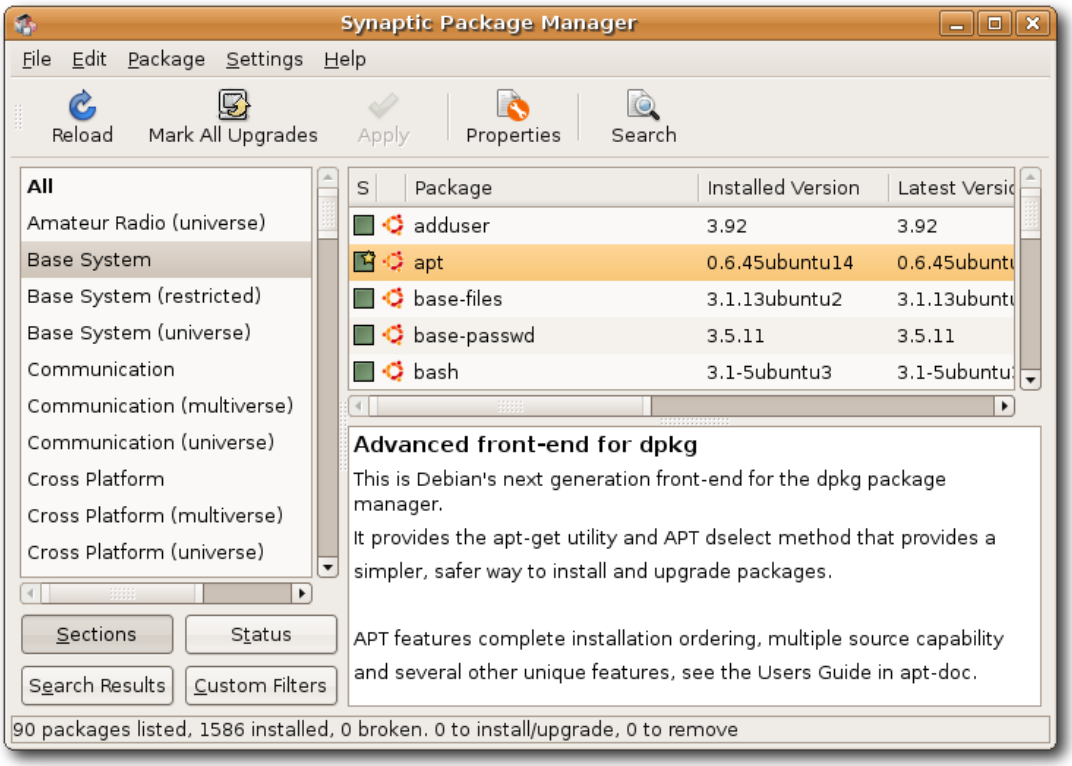
- Header Tags - v této skupině se nacházejí metadata týkající se balíčku samotného

    - RPMTAG_NAME - název balíčku
    - RPMTAG_VERSION - verze balíčku
    - RPMTAG_BUILDTIME - datum a čas sestavení
    - RPMTAG_ARCH - cílová architektura
    - RPMTAG_ARCHIVESIZE - velikost sekce archive (payload), atd.
- Private Header Tags - obsahuje neveřejné informace používané balíčkem
    - RPMTAG_HEADERIMMUTABLE - obsahuje data použitá pro výpočet v sekci Signature
    - RPMTAG_HEADERI18NTABLE - zde jsou uchovány názvy lokalizací pro dohledávání internacionalizovaných textů
- Dependency Tags - důležitá metadata pro určení vztahů a závislostí na okolních balíčcích
    - RPMTAG_REQUIRENAME - obsahuje seznam názvů balíčků, na kterých tento závisí
    - RPMTAG_REQUIREVERSION - obsahuje definice konkrétních verzí
    - RPMTAG_REQUIREFLAGS - tento příznak je vázán na verzi a označuje její kontext, nabývá hodnot:
        - RPMSENSE_LESS
        - RPMSENSE_GREATER
        - RPMSENSE_EQUAL, atd.
    - RPMTAG_PROVIDE, RPMTAG_CONFLICT, RPMTAG_OBSOLETE - další atributy mají také tři varianty, jako je tomu u RPMTAG_REQUIRE tudíž seznam názvů, verze a její příznak a jejich význam je obdobný jako u DEB balíčků

5.2.3. Synaptic

Synaptic [26] je grafická nadstavba pro systém pro správu balíčků Advanced Packaging Tool (apt). Spojuje v sobě jednoduchost point-and-click grafického uživatelského rozhraní s možnostmi apt-get nástrojů příkazového řádku. Synaptic umožňuje instalovat, odstraňovat, konfigurovat nebo aktualizovat softwarové balíčky, procházet, třídit a vyhledávat v seznamu dostupných softwarových balíčků, spravovat úložiště, nebo aktualizovat celý systém.

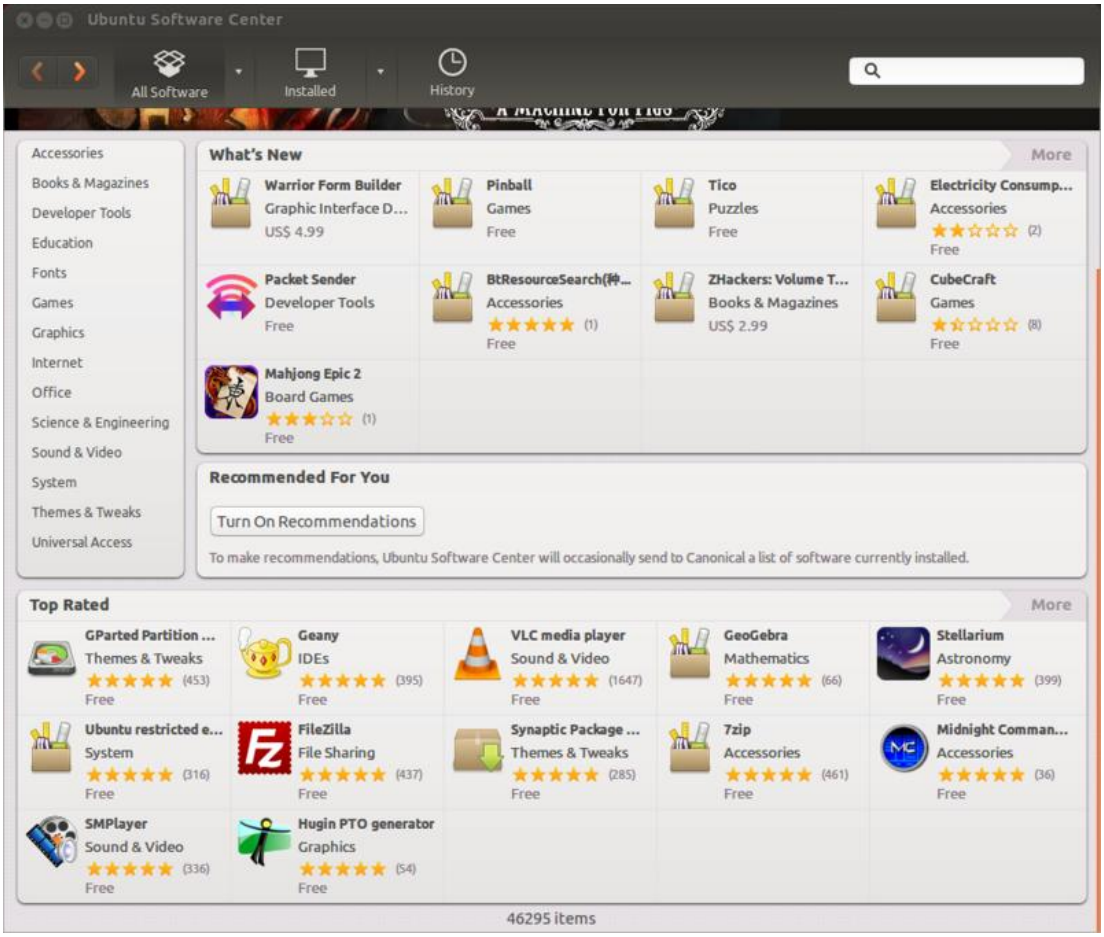
Podporuje řazení operací do fronty před jejich následným provedením. Synaptic informuje o závislostech a případných konfliktech s jinými balíčky, které jsou již nainstalovány v systému.



Obrázek 11: Synaptic Package Manager

5.2.4. Ubuntu Software Center

Pro Linuxovou distribuci Ubuntu dříve neexistoval jednotný přístup pro získávání balíčků. Uživatelé používali např. Synaptic, GDebi, atd. Z tohoto důvodu bylo pro Ubuntu navrženo centrum pro správu softwaru Ubuntu Software Center [27], které se stalo ústředním bodem pro získávání nového softwaru v této distribuci. Je zde kladen důraz na jednoduchost a uživatelskou přívětivost GUI (Graphical User Interface) aplikace. Funguje zde systém uživatelského hodnocení aplikací, díky němuž se lze vyvarovat nefunkčním či dokonce škodlivým balíčkům. Úložiště těchto balíčků (dostupné i přes webové rozhraní http://apps.ubuntu.com) obsahuje pouze oficiální a ověřené aplikace a knihovny pro zvýšení bezpečnosti jeho použití.

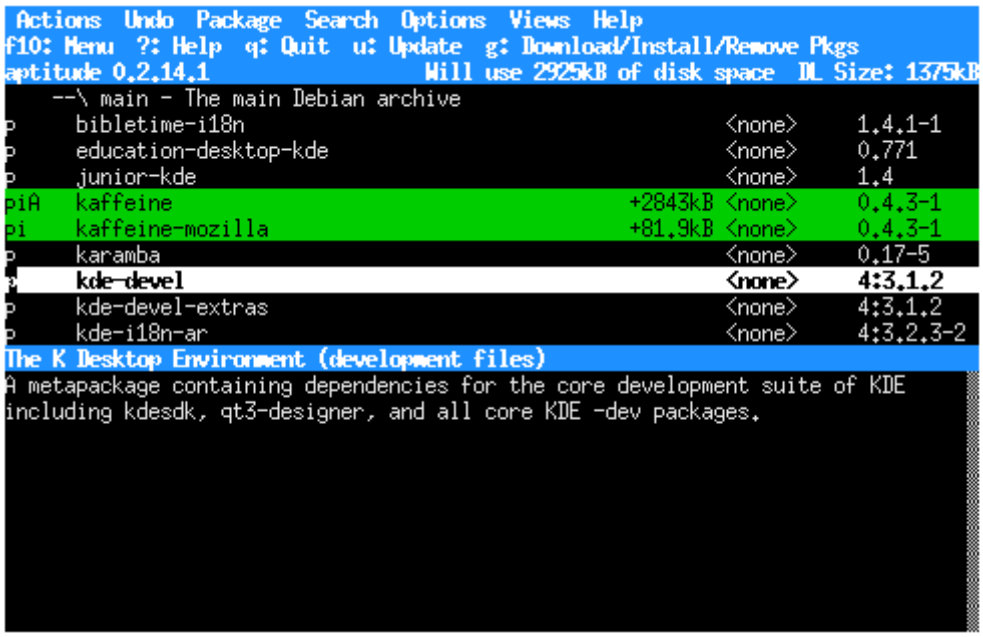


Obrázek 12: Ubuntu Software Center

5.2.5. Aptitude

Aptitude [28] je stejně jako Synaptic front-end k apt. Zobrazuje seznam softwarových balíčků a umožňuje uživateli interaktivně vybrat balíčky, které chce nainstalovat nebo odstranit. Má obzvláště silný vyhledávací systém s využitím flexibilních vzorů vyhledávání. Byl původně vytvořen pro Debian, ale postupem času obsáhl i další distribuce jako např. RedHat.

Kromě grafického rozhraní poskytuje také rozsáhlé rozhraní pro příkazovou řádku (CLI). Toto rozhraní podporuje řádku funkcí používaných v apt nástrojích (apt-get, apt-cache, apt-listchanges, atd.). Aptitude také emuluje většinu apt-get argumentů příkazového řádku, lze ho tudíž použít jako plnohodnotnou náhradu za apt-get.

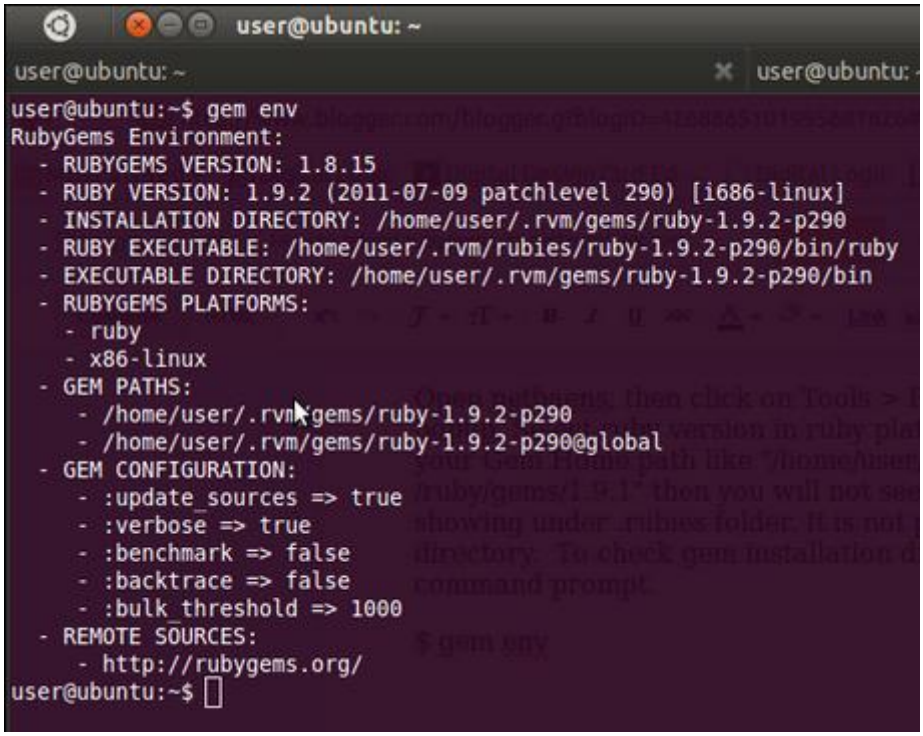


Obrázek 13: Aptitude

5.3. Ruby Gems

Ruby Gems [29] je správce balíčků pro programovací jazyk Ruby, který poskytuje standardní formát pro distribuci programů a knihoven Ruby tzv. "Gem". Tento nástroj je určený pro snadnou správu instalací Ruby Gems a serverů pro jejich distribuci. Tento přístup je analogií k Easy Install pro programovací jazyk Python. Gems jsou nyní součástí standardní knihovny Ruby od verze 1.9.

Ruby Gems nemají bohužel žádné grafické uživatelské rozhraní, k manipulaci s nimi se využívá výhradně příkazová řádka. Syntaxe příkazů je velmi podobná například Linuxovému apt-get. Je zde podpora pro instalaci ze vzdálených úložišť, odebírání již nainstalovaných balíčků, vyhledávání či výpis dostupných a již nainstalovaných balíčků.



Obrázek 14: Ruby Gems konzole

Nově je možné komunikovat s úložišti balíčků pomocí REST API [30]. Toto komunikační rozhraní podporuje zabezpečení pomocí autorizace. Lze díky němu získávat metadata o balíčcích ve formě JSON, XML či YAML (Yet Another Markup Language), samotné balíčky. Díky REST technologii lze například odebírat uživatelská oprávnění balíčků pomocí http DELETE operace.

5.4. Instalační nástroje Apache pro OSGi

Nadace Apache Software Foundation se v poslední době zaměřila na OSGi a vyvinula např. běhové prostředí Felix a úložiště komponent Felix OBR. Kromě těchto zmíněných se Apache zaměřuje na nadstavby či implementace dalších specifikací OSGi a to i takových, které poskytují prostředí s jednoduchým nasazením a aktualizací komponent. Do této kategorie patří následující technologie.

5.4.1. Karaf

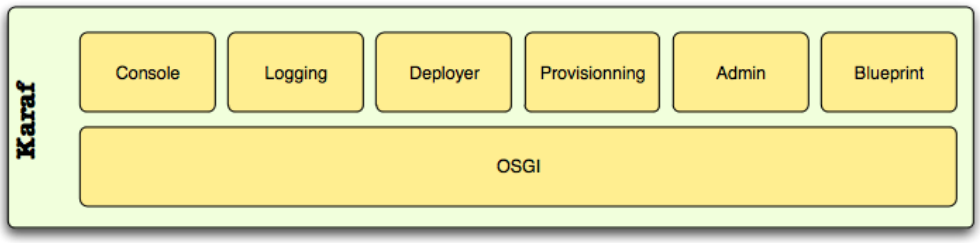
Apache Karaf [32] je odlehčený kontejner pro běh různých komponent a aplikací založený na OSGi technologiích. Karaf sám o sobě neposkytuje běhové prostředí, takže musí být podpořen nějakou implementací OSGi běhového

prostředí a to buď Apache Felix, nebo Eclipse Equinox. Karaf díky možnosti rozšíření může být jak jednoduchým komponentovým kontejnerem, tak i robustním systémem s podporou mnoha funkcí.

Jednou z jeho součástí je i Apache Aries [31]. Projekt Aries se skládá ze sady komponent určených pro platformu Java podporujících programovací model OSGi enterprise aplikací. Zahrnuje v sobě implementaci z následujících specifikací OSGi enterprise:

- Blueprint Specification
- JTA Transaction Services Specification
- JMX Management Model Specification
- JNDI Services Specification
- JPA Service Specification
- Service Loader Mediator Specification
- Subsystem Service Specification

Na obrázku níže jsou znázorněny nadstavby nad OSGi Frameworkem, které Karaf poskytuje.



Obrázek 15: Model Apache Karaf[1]

Základním principem Karafu je však jeho mechanismus XML deskriptorů tzv. Feature Descriptors. Feature zde znázorňuje určitou množinu funkcionality poskytovanou např. nějakou komponentou či aplikací. Těmito deskriptory lze popsat závislosti Features a tím definovat komponenty, které mají být do kontejneru zavedeny. Podobný princip je použit i u další popsané technologie Apache Ace.

[1] Převzato z [32]

Karaf sice není explicitně strukturován na klientskou a serverovou část, ale uživatel má jisté možnosti jak spravovat komponentový kontejner, je jich hned několik. První možností je konzole, která podporuje mimo jiné i standardní příkazy OSGi konzole (např. Felix Gogo), navíc však umožňuje interakci s OSGi PackageAdmin (informace o import/export balíčcích komponent) a s ConfigAdmin (nastavení kontejneru). Dále obsahuje nástroje pro vývojáře, logování a SSH. Tato konzole má však i svou obdobu dostupnou přes webové rozhraní. Další možností správy je ještě JMX (Java Management Extensions).

5.4.2. Ace

Framework Apache ACE [33] je navržen hlavně pro OSGi, ale návrh je spíše obecný a nezaměřuje se striktně na jednu platformu. Tento Framework umožňující centralizovanou správu komponent je určen k distribuci artefaktů do koncových zařízení založených na komponentové architektuře. Využití nalézá u distribuce vhodně dekomponovaných aplikací, kde je výsledný produkt pokaždé sestaven na míru z takových komponent, které svou funkcionalitou naplňují potřeby daného prostředí.

Koncové zařízení sleduje v předem definovaných intervalech změny na distribučním serveru, a když nalezne novější verze komponent, automaticky je zavede do OSGi Frameworku a odstraní nepotřebné, např. kvůli změně závislostí.

Apache Ace má architekturu typu klient server. Klientská aplikace je postavena na webovém rozhraní a dává uživatelům možnost ovládat funkcionalitu popsanou výše a také nastavovat komplexní konfiguraci distribuce artefaktů.

5.5. Porovnání implementací

Z přehledu klientských aplikací je patrné, že většina úložišť, potažmo jejich klientů pracujících s artefakty, ať už se jedná o softwarové balíčky či komponenty, vykazuje podobný přístup k problematice jejich správy. Na základě toho lze odvodit některé žádoucí vlastnosti, které by klientská aplikace měla mít. Měla by:

- umožňovat jednoduché získání artefaktů z úložiště
- mít funkce pro instalaci a odebírání artefaktů
- poskytovat podporu pro aktualizaci již nainstalovaných artefaktů
- řešit vzájemné závislosti, konflikty a kompatibilitu různých verzí
- mít grafické rozhraní zaměřené na přehlednost a intuitivní, uživatelsky přívětivé ovládání

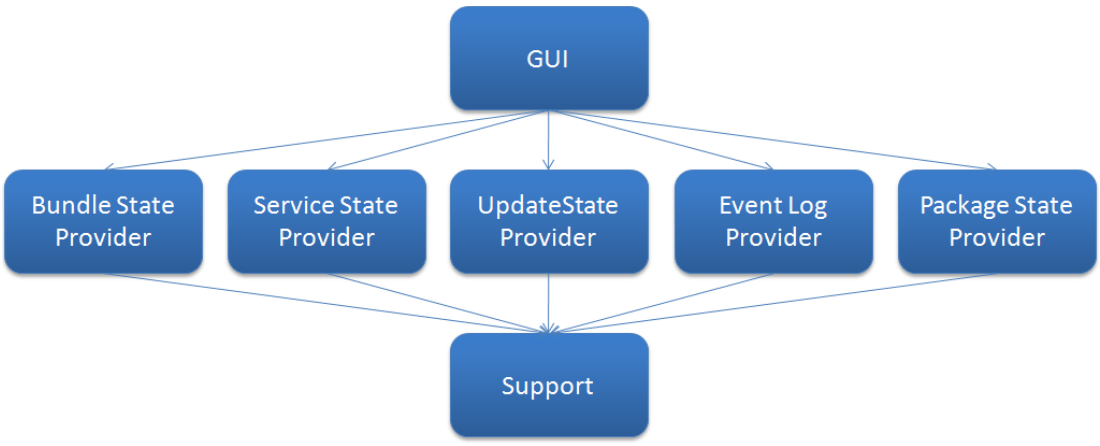
Z těchto poznatků lze vycházet i při implementaci klientské aplikace, jejíž vytvoření je cílem této práce. Aplikace by měla splňovat výše uvedená kritéria a obohatit toto odvětví využitím výhod, které přináší CRCE úložiště komponent.

6. GUICA

GUICA je desktopová Java aplikace určená k výzkumným účelům zaměřeným na komponentový Framework OSGi. Její vývoj zaštiťuje Katedra informatiky a výpočetní techniky Západočeské univerzity.
GUICA slouží jako grafické rozhraní pro správu komponent OSGi Frameworku a jako nadstavba OSGi konzole, která je určena k manipulaci s komponentami. To znamená, že svým uživatelům umožňuje získat přehled o komponentách zavedených do běhového prostředí, nabízí detailní pohled na OSGi bundle založený na informacích z jeho manifestu a poskytuje funkce pro jednoduchou manipulaci s komponentami jako je např. jejich spuštění či zastavení.

6.1. Moduly

Tato aplikace je modulární a je postavena právě na technologiích OSGi. Jednotlivé moduly zapouzdřují autonomní funkcionalitu, jejíž vnitřní reprezentace je pro zbytek aplikace skryta za rozhraním. Každý z modulů je OSGi bundle, který komunikuje s okolím pomocí služeb či vystavených softwarových balíčků používaných ostatními OSGi komponentami. Následující kapitoly jsou věnovány jednotlivým modulům aplikace GUICA s popisem jejich funkce v systému.



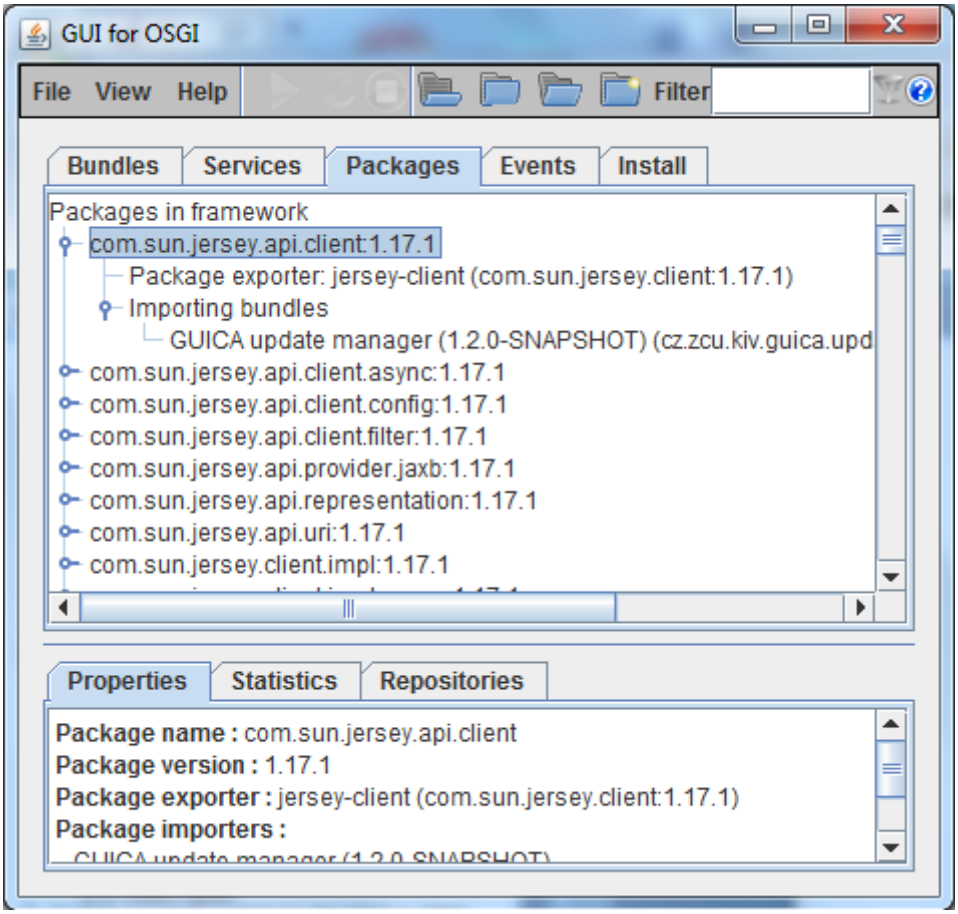
Obrázek 16: Model aplikace GUICA

6.1.1. Modul GUI

Modul grafického uživatelského rozhraní reprezentuje prezentační vrstvu modelu aplikace. Obsahuje jednotlivé obrazovky a funkce využívající

funkcionalitu poskytovanou ostatními OSGi komponentami. Grafický design je postaven na standardních Java nástrojích pro desktopové aplikace a to SWING a AWT. Pro přenesení prezentační vrstvy aplikace např. na web či mobilní zařízení (Android OS) stačí nahradit tento modul jiným, který je zaměřen na danou platformu, a ostatní bude fungovat standardně nezávisle na zobrazení.

Základní layout je rozdělen na tři oblasti. Patří sem horní horizontální záložkové menu, hlavní zobrazovací oblast a dolní panel obsahující buď ovládání, nebo doplňkové informace. Každá záložka horního menu představuje jeden modul, jenž poskytuje hlavní business logiku a data zobrazená v hlavní zobrazovací oblasti.



Obrázek 17: Záložka „Packages“ aplikace GUICA

Při pohledu na tento modul jako na OSGi komponentu je patrné, že plně využívá služeb, které poskytují okolní komponenty, neboť v definici importovaných balíčků jsou všechny dostupné.

6.1.2. Modul Support

Jedná se o čistě podpůrný modul obsahující obecná rozhraní a některé implementace používané celým systémem. Odstraňuje nutnost duplicity kódu a jeho kopírování. Zprostředkovává implementace základních operací s OSGi komponentami zavedenými do běhového prostředí, jako např. install/update/uninstall bundle nebo umožňuje získat seznam zavedených komponent či zaregistrovaných služeb.

6.1.3. Bundle state provider

Tento modul je určen pro správu OSGi Frameworku na úrovni jednotlivých bundlů. Umožňuje uživateli zobrazit strom bundlů zavedených do Frameworku s jejich aktuálními stavy. Bundle state provider však poskytuje o bundlu daleko detailnější informace. Umožňuje jednoduše zjistit seznam služeb registrovaných vybranou OSGi komponentou a naopak seznam služeb, které daná komponenta využívá. Dále lze získat přehledný výpis exportovaných a importovaných balíčků.

6.1.4. Service state provider

Díky tomuto modulu může uživatel získat komplexní přehled o službách zaregistrovaných v registru služeb OSGi Frameworku. Výpis služeb je opět strukturován ve formě stromu pro přehlednější zobrazení. Každá služba je samozřejmě navázána na konkrétní bundle, který ji zprostředkovává. Dále je možné zjistit základní vlastnosti registrované služby, jako je např. její id. V neposlední řadě je také důležité mít možnost zjistit, jaké komponenty danou službu využívají.

6.1.5. Package state provider

Pro správu obsahu běhového prostředí na úrovni softwarových balíčků je určen modul Package state provider. Díky němu je uživateli umožněno získat přehled o všech balíčcích, které OSGi komponenty zavedené do Frameworku exportují. Každému balíčku odpovídá jeden bundle, který ho zpřístupňuje ostatním.

Takový bundle je nazýván jako exportér a lze jednoduše dohledat, na kterých komponentách je závislý on sám.

6.1.6. Event log provider

Tento modul se oproti ostatním velmi liší. Zprostředkovává totiž služby umožňující logování událostí odehrávajících se během provozu OSGi Frameworku. Využívá možností OSGi služby zvané Event Admin, která je určena k zachytávání a posílání událostí viz kapitola 2.1. Rozděluje události do dvou základních skupin a to na systémové události a aplikační, přičemž systémové jsou všechny, které produkuje OSGi Framework. Tyto události zaznamenává na záložce „Evenst“ v Gui a přikládá k nim dodatečné informace zobrazené ve spodním panelu „Properties“.

6.1.7. Update state provider

Funkce poskytované tímto modulem jsou zaměřené na získávání komponent z externích komponentových úložišť. Uživatel zde může definovat soubor několika úložišť různých typů, ze kterých pak může komponenty zavádět do OSGi Frameworku.
Aktuálně podporované typy úložišť:

Čistý souborový systém

Tento typ úložiště představuje čistou adresářovou strukturu, která neobsahuje žádná dodatečná metadata o uložených artefaktech. Takové úložiště je definováno pouze cestou ke kořenovému adresáři (z pohledu úložiště, nikoli souborového systému), v jehož podadresářích se artefakty nacházejí. K načtení OSGi komponent je nutné tuto strukturu rekurzivně prohledat a nalézt JAR soubory. Každý JAR soubor však nemusí být OSGi komponenta, z tohoto důvodu se v aplikaci provádí kontrola, zda manifest obsahuje povinná pole pro OSGi bundle.

OBR

OBR formát úložiště je v aplikaci GUICA také podporován. Zde je k dispozici soubor obsahující metadata obsahu celého úložiště. Není tedy nutné procházet

strukturu, ve které jsou komponenty fyzicky uloženy. Definicí OBR úložiště je cesta k jeho repository.xml. K načítání OSGi komponent se pak použije proces založený na přečtení URI identifikátorů jednotlivých OSGi komponent pospaných ve zmíněném souboru repository.xml. JAR soubory komponent se nestahují při načítání komponent, ale až při instalaci do běhového prostředí. To umožňují standardní OSGi nástroje. Metoda instalující bundle do běhového Frameworku totiž akceptuje jako parametr obecné umístění komponenty jak z lokálního, tak i vzdáleného úložiště.

Když jsou komponenty v „aplikačním“ úložišti načteny, zobrazí se uživateli pod záložkou „Install“. Odtud je lze instalovat do běhového prostředí nebo je naopak odinstalovat.
Zmíněná funkcionalita modulu Update State Provider byla již před touto prací implementována, ale udržovatelnost a kvalita jejích zdrojových kódů dosahovala nízké úrovně. Navíc kód obsahoval chyby a tak bylo v rámci této práce zapotřebí velkou část stávajících funkcí stejně přepracovat.

7. Analýza

První fází vývoje klientské aplikace s podporou aktualizací komponent z úložiště CRCE byla analýza. Rámec analýzy zahrnoval bližší specifikaci požadavků na klientskou aplikaci, zvolení vhodného běhového prostředí a výběr externích knihoven a nástrojů.

7.1. Požadavky na klientskou aplikaci úložiště CRCE

CRCE úložiště je již v současné době velmi rozsáhlé, ale doposud nemohlo být plně využito. Chybí totiž klientská aplikace, která by využívala výhod CRCE plynoucích z ověřování kompatibility uložených komponent. Cílem této práce je implementace takové klientské aplikace.

Základní požadavky kladené na klienta jsou následující:

- klientská aplikace musí umožňovat komunikaci s CRCE úložištěm pravděpodobně přes webové služby
- dále je vyžadována možnost provádění konzistentních aktualizací komponent pocházejících z CRCE
- jádro implementace musí být obecné a vycházet z technologií Java a OSGi
- klientská aplikace bude buďto desktopová (rozšíření aplikace GUICA) nebo určená pro mobilní platformu Android

Takto specifikované požadavky byly součástí zadání této práce. K jejich naplnění byla nejdříve provedena analýza interních nástrojů (GUICA, CRCE) i externích knihoven třetích stran.
Z analýzy vyplynula některá fakta ovlivňující detaily jednotlivých požadavků. Například fakt, že pro komunikaci s CRCE bude použito komunikační rozhraní založené na konceptu REST API. Nebo zvolení desktopové platformy a úpravy aplikace GUICA, která je detailněji popsána v následující kapitole.

7.2. Běhové prostředí

Jedním z možných prostředí byl operační systém Android, který se v poslední době dostal do téměř všech mobilních zařízení a byl by tudíž i vhodným kandidátem pro umístění do embedded zařízení. Tato možnost byla bohužel hned v počátku projektu zamítnuta, a to z důvodu značné pracnosti převyšující svým objemem rozsah této práce. Bylo by totiž nutné z velké části znovu implementovat již vytvořené funkce aplikace GUICA, která již graficky podporovala většinu operací OSGi konzole. Výhodnější tedy bylo rozšířit zmiňovanou aplikaci GUICA, což umožnilo více se zaměřit na vlastní klientskou část. Detaily implementace GUICA a technologií, na kterých je postavena, jsou uvedené v kapitole 6 a 9.

7.3. Použité knihovny a nástroje

V rámci analýzy stál vývoj před dalším rozhodnutím. Bylo nutné zvolit nejvhodnější nástroj pro práci s XML soubory. Doposud se na straně klientské aplikace s XML vůbec nepracovalo a na straně úložiště CRCE byla použita poměrně zastaralá knihovna kXML. Byla tedy provedena analýza knihoven pro práci s XML, ze které vyšlo jako nejvhodnější JAXB. Popis prováděných testů a měření se nachází v podkapitole 7.4.

Dále byl předmětem analýzy návrh rozhraní mezi klientskou a serverovou částí (CRCE). Funkcionalita umožňující komunikaci s úložištěm za účelem získávání komponent a jejich popisných metadat mimo webové rozhraní byla cílem jiné diplomové práce, která probíhala paralelně s touto. Bylo tudíž zapotřebí navzájem spolupracovat a synchronizovat společné rozhraní na obou stranách. Kdyby byly využívány webové služby, jednalo by se o synchronizaci Java interface na straně klienta s jeho implementací na straně serveru. To by mohlo být v některých fázích vývoje značně omezující a jistě by to ovlivnilo rozhodnutí zvolit pro komunikaci REST API. Postup návrhu a bližší informace o komunikačním rozhraní jsou v kapitole 8.1.

Poslední část analýzy se týkala popisných metadat. Jejich formát zaznamenal během realizace značný vývoj. Na jedné straně byl přizpůsobován nové

specifikaci OBR formátu verze R5 popsanému v kapitole 4.1.1., a na straně druhé byl rozšiřován jmenný prostor CRCE úložiště.

7.4. Výběr knihovny pro zpracování XML metadat

V úložišti CRCE jsou veškeré komponenty opatřeny popisnými metadaty, tato data jsou poté uložena ve formě XML souborů. Obsah popisných XML souborů OSGI komponent odpovídá formátu OBR rozšířeného o interní jmenný prostor pro účely CRCE viz kapitola 4.3.3.

Při komunikaci mezi CRCE a klientskou aplikací je zapotřebí analyzovat a vytvořit velké množství XML souborů. Z tohoto důvodu bylo v úvodních částech návrhu architektury důležité vhodně zvolit nástroj pro práci s XML. Po krátké analýze bylo vybráno JAXB pro svou výkonnost a sofistikovanost a kXML 2, protože to bylo v minulosti v CRCE již použito. Zbývalo jen exaktně rozhodnout, kterou z těchto dvou knihoven vybrat. Následuje stručná charakteristika obou knihoven.

kXML 2

Tato knihovna je součástí sady knihoven nazvané kObjects, které spravuje Stefan Haustein viz [21]. kXML je založené na tzv. XML pull parseru, který kombinuje výhody SAX (Simple API for XML) a DOM (Domain Object Model). Jelikož se tato knihovna neprosadila v prostředí J2SE či J2EE, zacílila se na mobilní a embedded zařízení, což by pro účely této práce nepředstavovalo samo o sobě problém, protože i zde jsou ambice pojmutí i mobilních zařízení. Větším problémem je však neudržovanost a takřka nulová podpora ze strany distributora.

JAXB

Knihovna pro zpracování XML s názvem JAXB je referenční implementací specifikace JSR-222 s názvem Java Architecture for XML Binding viz [22]. Tato knihovna používá standardní POJO objekty rozšířené o anotace z balíčku `javax.xml.bind`. Z toho vyplývá i další výhoda a to její přítomnost mezi standardními Java knihovnami. Je součástí JDK (Java Development Kit) od

verze 6. Detailnější popis práce s knihovnou je popsán v kapitole 9.3.1. Projekt se neustále vyvíjí a navazují na něj další technologie jako např. JAXWS.

Pro podporu rozhodování byly implementovány dvě stejné aplikace, přičemž každá využívala jednu ze zmíněných knihoven. K testování výkonnosti byly použity dvě základní metriky a to čas potřebný k načtení či vytvoření souboru a množství operační paměti, které bylo pro tyto operace zapotřebí. Dohromady proběhla čtyři různá testování obou nástrojů a každé z nich bylo provedeno několikanásobně pro zvýšení relevance výsledků.
První dva výkonnostní testy byly navrženy k určení systémových nároků generování (vytváření) XML souborů z objektů tříd reprezentujících komponenty v úložišti. Obsah souborů je do jisté míry náhodný, ale odpovídá formátu metadat používaných v CRCE. Důvodem je snaha o věrnější simulaci zatížení vznikajícího při REST komunikaci.

- V prvním testu byl měřen čas a využitá paměť při vytváření jednoho XML souboru o předem určeném počtu řádků. Po předchozí úvaze o potřebách komunikace mezi CRCE a klientskou aplikací byl test spuštěn s počtem řádek pohybujícím se v rozmezí 10 až 100 000. Pokaždé byly otestovány obě dvě knihovny.
- Druhý test byl zaměřen na zkoumání náročnosti vytváření většího množství relativně malých XML souborů o cca 20 řádcích. Vstupním parametrem tohoto měření nebyl tedy počet řádek, ale požadovaný počet generovaných souborů. Opět s ohledem na možné reálné využití knihoven bylo generováno 10 až 10 000 souborů.

Třetí a čtvrtý test je zaměřen na takzvané parsování, což ze slova parse - analýza znamená v tomto konkrétním případě převedení textového obsahu XML souboru na instance třídy reprezentující komponentu pro možnost další práce s daty jako proměnnými v aplikaci za pomoci jedné z testovaných knihoven. Parsování bude jednou z nejdůležitějších a nejčastěji používaných operací při komunikaci systémů.

- Třetí test je tedy zaměřen na parsování, ale konkrétně na rozsáhlé soubory s velkým počtem řádků. Při každém měření je tedy načten jeden soubor a výstupem je jeden velký objekt. Testování probíhalo podobně jako u prvního testu na souborech o 10 až 100 000 řádcích.
- V posledním čtvrtém testu byla logicky porovnávána výkonnost knihoven při parsování velkého množství malých souborů. Zde byl upraven počet analyzovaných souborů oproti druhému měření na 1 až 10 000, protože bylo zajímavé změřit nároky na parsování i menšího počtu souborů.

Pro měření obou sledovaných veličin tudíž času a množství využité paměti byly využity standardní Java nástroje specifikované níže a pro ověření správnosti naměřených výsledků byla ještě použita aplikace JProfiler[1] verze 7.2.2.

Měření času

Doba trvání zkoumaných operací byla určována pomocí dvou časů. První byl zaznamenán před spuštěním výkonné metody a druhý po návratu z ní. Výsledná doba se pak určila odečtením těchto dvou časů. Zmíněný čas byl získán pomocí třídy `System` ze standardního Java balíčku `java.lang` a to konkrétně metodou `System.currentTimeMillis()`.

Příklad:

```
time1 = System.currentTimeMillis();
parser.parse();
time2 = System.currentTimeMillis();
```

Měření paměti

K určení množství využité paměti potřebné pro vykonání zkoumaných operací byla využita třída `Runtime` opět ze standardního Java balíčku `java.lang`. Instance této třídy je vytvořena jednou pro celou aplikaci a získá se metodou `Runtime.getRuntime()`. Ta umožňuje zjistit informace o celkovém množství paměti (`runtime.totalMemory()`) a o volné paměti (`runtime.freeMemory()`) v jednotkách bytů. Odečtením těchto hodnot lze odvodit množství využité

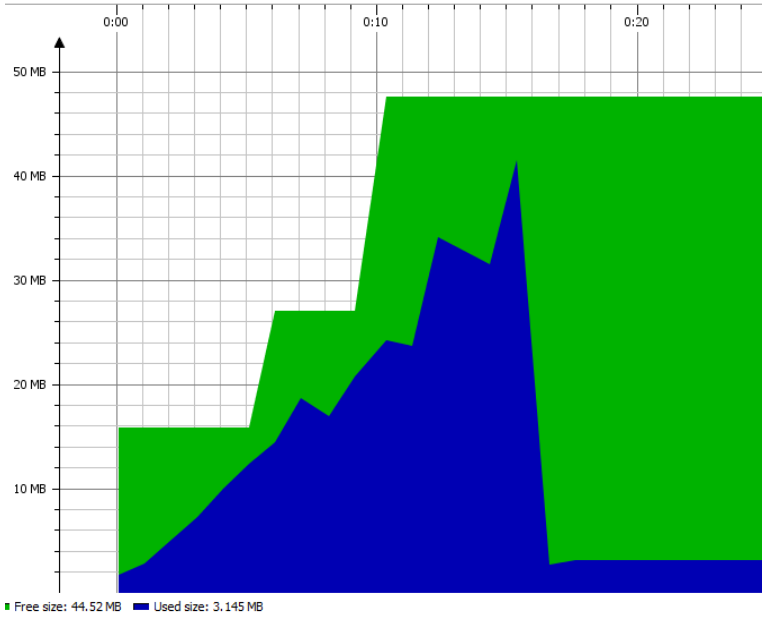
[1] JProfiler viz www.jprofiler.com

paměti. Získávání hodnot potřebné paměti bylo založeno na principu zaznamenávání stavu před a po zavolání výkonné metody jako u měření času.

Příklad:

```
memory1 = runtime.totalMemory() - runtime.freeMemory();
parser.parse();
memory2 = runtime.totalMemory() - runtime.freeMemory();
```

Takto zjištěné hodnoty byly následně porovnány s výsledky naměřenými pomocí aplikace JProfiler. V tomto nástroji lze spustit Java aplikaci k zaznamenávání veškerých informací o jejím běhu včetně množství aktuálně využitých systémových zdrojů.



Obrázek 18: Graf aplikace JProfiler - využití paměti

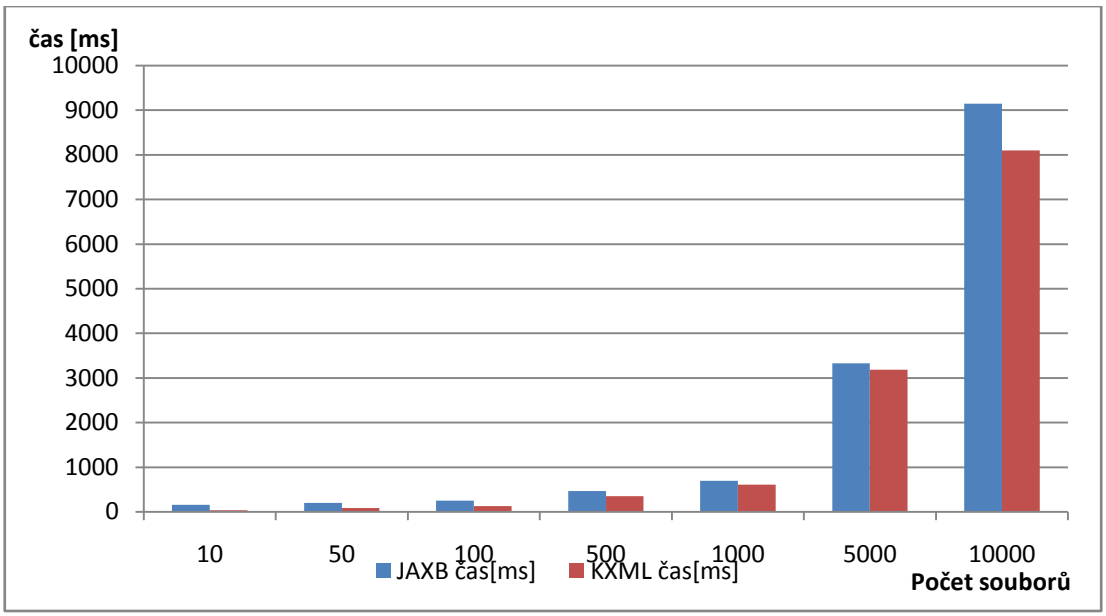
Na obrázku 18 je graf, který zobrazuje průběh alokace paměti při spuštění čtvrtého testu (parsování desetitisíc XML souborů). Maximální hodnota množství využité paměti se zde pohybuje okolo 42MB, což odpovídá výsledkům měření, které následují níže.

Hardwarová konfigurace stroje, na kterém měření probíhalo, byla následující:

Procesor:	Intel Core 2 Duo T6600 2,2GHz
Operační paměť:	4GB
Operační systém:	Windows 7 32b

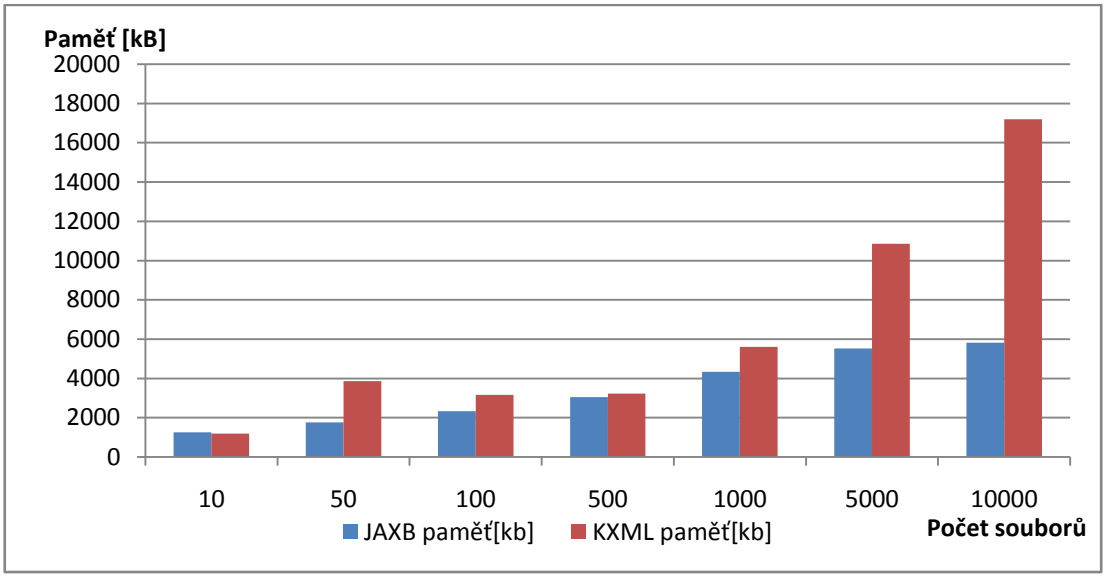
7.4.1. Výsledky měření

Ze všech osmi grafů, které byly výstupem měření, zde budou uvedeny pouze čtyři z důvodu značné podobnosti. Následující grafy zobrazují výsledky druhého testu (vytváření většího množství souborů). První graf zobrazuje časovou náročnost operace a druhý graf reprezentuje naměřené hodnoty využité paměti.



Obrázek 19: Graf časové náročnosti testu vytváření XML souborů

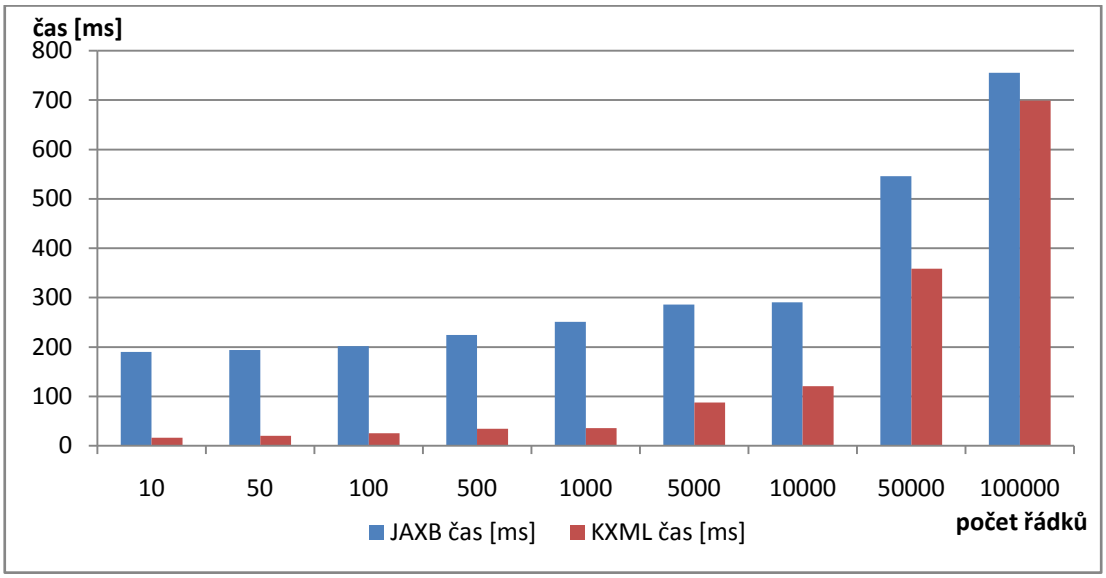
JAXB v tomto testu zpracovalo data o něco pomaleji než kXML. Knihovna kXML vykazovala u většiny testů o něco menší časovou náročnost hlavně při menším množství dat, z tohoto grafu je však patrné, že i pro 10 000 souborů byla rychlejší.



Obrázek 20: Graf využití paměti při testu vytváření XML souborů

Nároky na paměť jsou však u kXML mnohem větší a trend hodnot má mnohem rychleji rostoucí charakter, což samo o sobě představuje problém. JAXB vykazuje v tomto případě ideálnější a lépe predikovatelné chování.

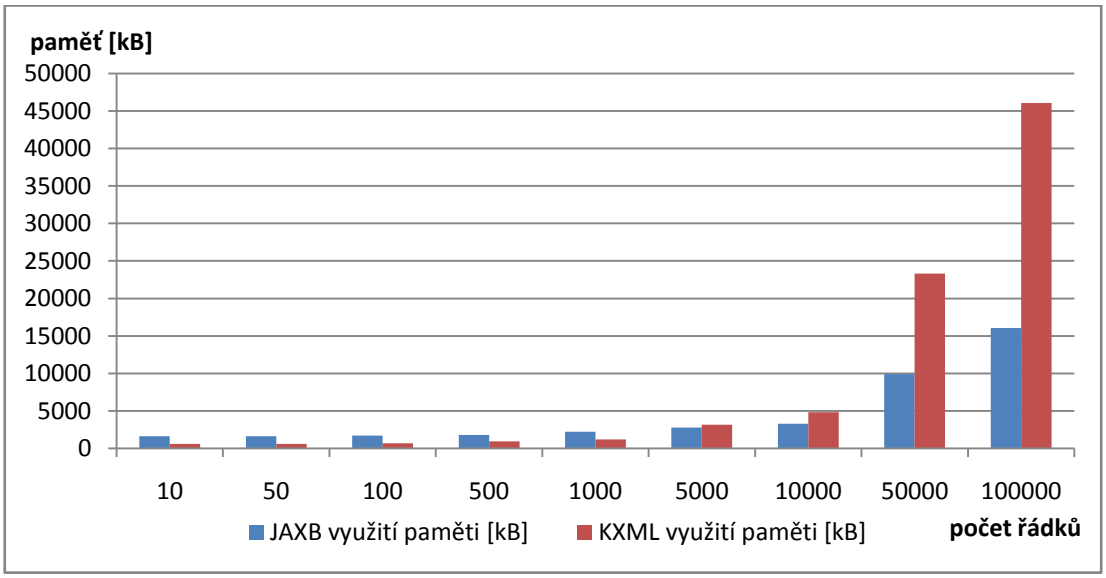
Následující dvě grafická zobrazení výsledků třetího měření (parsování souboru o určeném počtu řádků). První graf opět zobrazuje výsledky měření času a druhý měření množství využité operační paměti.



Obrázek 21: Graf časové náročnosti testu parsování XML souborů

Podobně jako u předchozích grafů i zde je vidět vyšší časová náročnost použití knihovny JAXB. Zde jsou rozdíly ještě patrnější než v předchozím případě, ale

stále se pohybujeme v rozdílech maximálně 200ms. Tyto hodnoty jsou pro naše účely ještě únosné, protože jsou vyváženy paměťovými nároky.



Obrázek 22: Graf využití paměti testu parsování XML souborů

Na tomto grafu vyniká „síla“ knihovny JAXB vykazující opět téměř lineární chování. Hodnoty množství paměti v maximálních počtech řádků parsovaného souboru se liší téměř řádově.

7.4.2. Shrnutí

Naměřená data zobrazená v grafech vypovídají o tom, že obě sledované metriky pro kXML vycházejí při menším množství zpracovávaných dat lépe, ale rychle se zhoršují. JAXB vykazuje naopak vyšší nároky na paměť při menších objemech zpracovávaných dat. To je však dáno vytvořením kontextu potřebného pro práci knihovny, který se vytváří pro třídu Resource a zabere přibližně 1000 kB operační paměti. Poté stoupají nároky na systémové zdroje lineárně oproti kXML vykazující téměř exponenciální chování. To představuje největší problém. Při komunikaci přes REST API jsou velmi nevhodné velké časové rozdíly při zpracovávání požadavků. Další výhodou JAXB je také mnohem větší uživatelská přívětivost, přehlednost kódu napomáhající lepší udržitelnosti a celková sofistikovanost knihovny. Na základě dat získaných při měření a empiricky nabytých zkušeností při implementaci testovacích aplikací bylo rozhodnuto pro použití knihovny JAXB.

8. Architektura

Celý systém se skládá ze dvou základních částí. CRCE úložiště vystupuje jako server a GUICA jako klient s podporou aktualizací komponent. Tyto hlavní celky se dále skládají z několika dílčích komponent, oba jsou implementovány v prostředí OSGi a jejich komponenty jsou tedy OSGi bundly. V prvních fázích realizace bylo CRCE založeno na implementaci OSGi Apache Felix a GUICA běžela v prostředí implementace Equinox, která je vyvíjena pod záštitou společnosti Eclipse. V průběhu vývoje se nároky na prostředí sjednotily a obě aplikace nyní využívají implementaci Felix. Tento fakt neměl však vliv na vzájemnou komunikaci obou systémů, která probíhala přes REST rozhraní.

Při bližším pohledu na aplikaci GUICA je z hlediska architektury postavena na OSGi, jak již bylo zmíněno. GUICA má grafické uživatelské rozhraní (dále jen GUI) vyčleněné do zvláštní komponenty. Toto GUI je implementováno s využitím standardních Java grafických nástrojů knihovny Swing.

Jelikož není implementace CRCE předmětem této práce, bude zde architektura CRCE zmíněna jen okrajově. CRCE také obsahuje jednu komponentu vyčleněnou pro uživatelské rozhraní. V tomto případě se jedná o webové rozhraní, pomocí kterého lze do úložiště nahrát komponenty, zobrazovat jeho obsah a detaily komponent včetně jejich popisných metadat.

8.1. Specifikace REST API úložiště

REST API je na straně serveru (to v tomto případě znamená na straně úložiště CRCE) navrženo velmi obecně a to proto, aby k němu mohl přistupovat jakýkoli klient, který bude znát toto rozhraní. GUICA je jen první implementací klienta pro toto úložiště.

REST API se skládá z několika typů požadavků, které umí server zpracovat a kterým přísluší vlastní obslužná metoda. Návratovým typem většiny těchto metod bývá XML soubor s popisnými metadaty obsahujícími od popisu nějaké konkrétní komponenty až po popis celého úložiště. Ojediněle je ale návratovým typem přímo JAR soubor OSGi bundlu.

Specifikace REST API vznikla na základě případů užití navržených pro klientskou aplikaci. Tyto případy užití budou včetně sekvenčních diagramů a odpovídajících požadavků a odpovědí podrobně popsány v následujícím textu.
Požadavky klientské aplikace na server by se daly rozdělit do dvou základních kategorií. Jednou z nich jsou požadavky založené na získávání bundlů či metainformací o nich. Tato kategorie je ve specifikaci nazývána „Základní operace (Basic operations)“. Druhou z kategorií jsou tzv. „Nahrazovací operace (Meta-data for Substitution)“. Ty obsahují požadavky odpovídající dotazům „Které nejnovější bundly jsou kompatibilní s tímto?“ nebo „Který bundle zprostředkovává danou službu?“
CRCE webový server naslouchá na dvou různých URL. Jedno odkazuje uživatele na administrátorskou aplikaci pro správu úložiště, poté toto URL vypadá následovně „http://{adresa_serveru}:{port}/crce“. Druhé URL odkazuje na REST rozhraní a je následující „http://{adresa_serveru}:{port}/rest“. Pro zaslání konkrétního požadavku je zapotřebí za zmíněné URL uvést název operace. Parametry požadavků se zapisují podle standardu http protokolu, tzn. …/rest/{nazev_operace}?{nazev_parametru}={hodnota_parametru}.

8.1.1. Základní operace

Požadavek na konkrétní bundle

Tomuto požadavku odpovídá operace „get bundle“. CRCE odpoví kódem 200 a přiloženým OSGi bundlem, byl-li v úložišti nalezen, pokud ne, je návratovým kódem 404. Požadavek na konkrétní bundle může mít tři základní podoby.

1) GET /bundle/{id} – za id lze dosadit OSGi identifikátor

2) GET /bundle?name={název}&version={verze} – pokud je znám název komponenty a její konkrétní verze, je možné požadavek zapsat i tímto způsobem. Rozdíl oproti předchozímu je však minimální, protože OSGi identifikátor je výsledkem konkatenace názvu komponenty a její verze.

3) GET /bundle?name={název} – pokud není známa konkrétní verze komponenty, je možné vynechat parametr verze. CRCE v tomto případě vrátí nejnovější dostupnou verzi dané komponenty v rámci celého úložiště.

Požadavek na popisná metadata

Operací odpovídající tomuto požadavku je operace „get metadata“. Tato funkce je určená k získání popisných metadat, která jsou detailně popsána v kapitole 4.3.3. Díky velké škále možností, jak tuto operaci volat a díky komplexně zpracované množině parametrů lze získat nejen kompletní metainformace o všech OSGi komponentách v úložišti, ale i informace o konkrétní službě zprostředkovávané konkrétní komponentou.
Jelikož jsou u této funkce parametry o mnoho bohatší a složitější než u předchozí, je třeba jejich význam lépe osvětlit.

- core
  - core nebo core = all – všechny metainformace, které představují tzv. jádro metadata, kam spadají informace o bundlu samotném a o jeho fyzické reprezentaci, nikoli informace o závislostech a vazbách na ostatní bundly
  - core = {název} – metadata typu „osgi identity“ daných komponent
- cap
  - cap nebo cap = all – všechny elementy typu „capability“
  - cap={nazev} – všechny elementy typu „capability“ s názvem odpovídajícím zadané hodnotě parametru např. pro všechny metainformace cap:osgi.wiring.package bude parametr vypadat takto: cap='osgi.wiring.package'
- req
  - req nebo req – všechny elementy typu „requirement“
  - req={nazev} – všechny elementy typu „requirement“ s názvem odpovídajícím zadané hodnotě parametru, stejně jako u předchozího
- prop
  - prop nebo prop = all – všechny elementy typu "crce:property"

- prop={nazev} – všechny elementy typu „crce:property“ s názvem odpovídajícím zadané hodnotě parametru, stejně jako u předchozího

- filter
  - filter={výzar} – omezuje výběr metadata na ty bundly, které odpovídají zadanému výrazu
  - hodnotou výrazu je řetězec s LDAP syntaxí, kterou lze definovat filtr např. pro určitý rozsah verzí bundlu nebo jeho konkrétní název
  - takový filtr by vypadal následovně:

    filter=(&(name={nazev})(version>=1.0))

Pro ilustraci budou dále uvedeny vzorové požadavky s popisem hodnot odpovědí serveru.

1) GET /metadata – tento tvar volání znamená požadavek na kompletní metadata celého úložiště

2) GET /metadata/{id} – takto se získávají metainformace o jedné konkrétní komponentě, id má tvar jako v případě operace „get bundle“

3) GET /metadata/{id}?core&prop – výsledkem takto zadaného požadavku jsou všechna „core“ metadata a vlastnosti bundlu odpovídajícího zadanému identifikátoru

4) GET/metadata?filter=(&(name={nazev})(version>=1.0))
   &core&prop=crce.compatibility – odpovědí serveru na tento požadavek budou všechna „core“ metadata a vlastnosti typu „crce.compatibility“ a to takových bundlů, které odpovídají zadanému filtru, jehož význam je popsán u parametru filter

Příklad XML metadat z odpovědi CRCE úložiště:

Dotaz: GET /metadata/com.sun.jersey.bundle-1.17.0

Fragment XML odpovědi:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:repository xmlns:ns2="TBD-CRCE-METADATA-XSD-URI">
    <ressource ns2:id="com.sun.jersey.bundle-1.17.0">
        <requirement namespace="osgi.wiring.package">
            <directive name="filter"
                       value="(&amp;(package=javax.xml.bind))"/>
            <attribute name="name" value="javax.xml.bind"/>
        </requirement>
        ...
        <capability namespace="osgi.identity">
            <attribute name="name" value="com.sun.jersey.bundle" />
            <attribute name="version" value="1.17.0" type="Version" />
        </capability>
        <capability namespace="osgi.content">
            <attribute name="hash" value="b98c...069290" />
            <attribute name="url"
                value="http://localhost:8080/rest/
                       bundle/com.sun.jersey.bundle-1.17.0" />
            <attribute name="size" value="1500454" type="Long" />
            <attribute name="mime" value="application/vnd.osgi.bundle" />
            <attribute name="crce.original-file-name"
                       value="jersey-bundle-1.17.jar" />
        </capability>
        <capability namespace="crce.identity">
            <attribute name="name" value="com.sun.jersey.bundle-1.17.0" />
            <attribute name="crce.categories" value="zip,osgi"
                       type="List&lt;String&gt;" />
            <attribute name="crce.status" value="stored" />
        </capability>
        <capability namespace="osgi.wiring.package">
            <attribute name="name" value="com.sun.ws.rs.ext" />
            <attribute name="version" value="1.17.0" type="Version" />
        </capability>
        <capability namespace="osgi.wiring.package">
            <attribute name="name" value="com.sun.jersey.core.util" />
            <attribute name="version" value="1.17.0" type="Version" />
        </capability>
        ...
    </resource>
</ns2:repository>
```

8.1.2. Nahrazovací operace

Požadavek na nahrazení konkrétního bundlu

Když se klientská aplikace dotáže CRCE úložiště na data potřebná k nahrazení aktuálně používané verze nějakého konkrétního bundlu specifikovaného předaným identifikátorem, který z tohoto úložiště pochází, CRCE vrátí v odpovědi metadata resp. již zmiňovanou „core“ část bundlu či skupiny bundlů, jenž jsou striktně kompatibilní, mají stejný název a stejné závislosti na okolní komponenty. Pokud žádný z bundlů v úložišti neodpovídá těmto požadavkům, odpovědí je kód 404 a zpráva, že v úložišti nebyla nalezena žádná odpovídající verze.

Jako u předchozích operací i zde existuje několik možných variant, jak požadavek sestavit, aby splňoval požadované podmínky. Parametry jsou v tomto případě pouze dva. Jedním z nich je identifikátor aktuálně používaného bundlu a druhým je operace, jenž má být se zadaným bundlem provedena. Parametr má sadu předdefinovaných hodnot, které určují operaci. Ty jsou následující:

- upgrade – metadata všech kompatibilních komponent s verzí vyšší než je verze zadaného bundlu
- downgrade – metadata všech komponent s verzí nižší než je verze zadaného bundlu
- lowest – metadata jednoho konkrétního bundlu, který má nejnižší verzi oproti zadanému bundlu, to znamená jeho nejbližší novější verzi
- highest – metadata jednoho konkrétního bundlu, který má nejvyšší verzi oproti zadanému bundlu, tímto je míněno, že má nejvyšší verzi ze všech novějších komponent
- any – metadata všech komponent, jejichž verze je rozdílná od verze zadaného bundlu

Níže budou opět charakterizovány možné varianty požadavku spolu s popisem obsahu odpovědi úložiště.

1) GET replace-bundle?id={id} – odpovědí na takto zapsaný požadavek jsou metadata všech kompatibilních komponent s verzí vyšší, což odpovídá operaci upgrade, která je považována za defaultní, pokud není jiná operace implicitně zadána
2) GET replace-bundle?id={id}&op={upgrade, highest, atd.} – tímto způsobem se zapisují jednotlivé operace, odpovědi na ně jsou uvedeny výše

Požadavek na bundle zprostředkovávající danou službu

Předá-li klient požadavek na bundle zprostředkovávající danou službu, CRCE odpoví zprávou obsahující metadata bundlů poskytujících požadovanou službu, resp. majících požadované „capability“. Pokud žádný z bundlů v úložišti neodpovídá těmto požadavkům, odpovědí je kód 404. Pro tento požadavek se používá metoda POST http protokolu místo obvyklé GET. Požadavek nemá žádné parametry, ale v jeho těle jsou zaslána XML metadata obsahující elementy „requirement“, kterým musí odpovídat „capability“ požadovaných bundlů.

Příklad požadavku:

POST provider-of-capability/

```
<requirement namespace='osgi.wiring.package'>
    <attribute name='name' value='cz.zcu.kiv.parkoviste' />
    <attribute name='version' value='1.0' op='less-than' />
    <directive name='filter'
               value='(&(osgi.wiring.package=cz.zcu.kiv.parkoviste)
                      (version&gt;=1.0.0))' />
</requirement>
```

Úložiště CRCE podporuje ještě více operací dostupných přes REST API, které budou moci jejich klienti využívat. Z hlediska první realizace klienta pro CRCE úložiště však byly implementovány pouze výše uvedené operace. Všechny jsou využívány pro funkci získávání komponent z úložiště a pro možnost jejich následné aktualizace v aplikaci GUICA, s výjimkou posledního popsaného požadavku na bundle zprostředkovávající danou službu. Ten svoje reálné využití zatím nenalezl, ale je jen otázkou času, kdy jeho implementace na obou stranách najde své uplatnění.

9. Implementace klientské části

Implementace klientské části naplňuje zadané cíle a vychází z poznatků získaných během fáze analýzy tak, jak byly obě tyto části popsány v kapitole 0. Pro realizaci požadované funkcionality bylo nutné rozšířit modul Update state provider, přidat do GUI potřebné obrazovky a upravit zobrazení některých prvků.

9.1. Obecné znaky implementace

Z důvodu zachování obecnosti a celkové udržovatelnosti kódu byly všechny třídy, tak specificky zaměřené na práci s CRCE úložištěm, že je nešlo navrhnout obecněji, striktně vyčleněny do balíčků:
`cz.zcu.kiv.guica.{název_modulu}.crce.*`.

Vlastní implementace by se dala rozdělit do následujících oblastí:

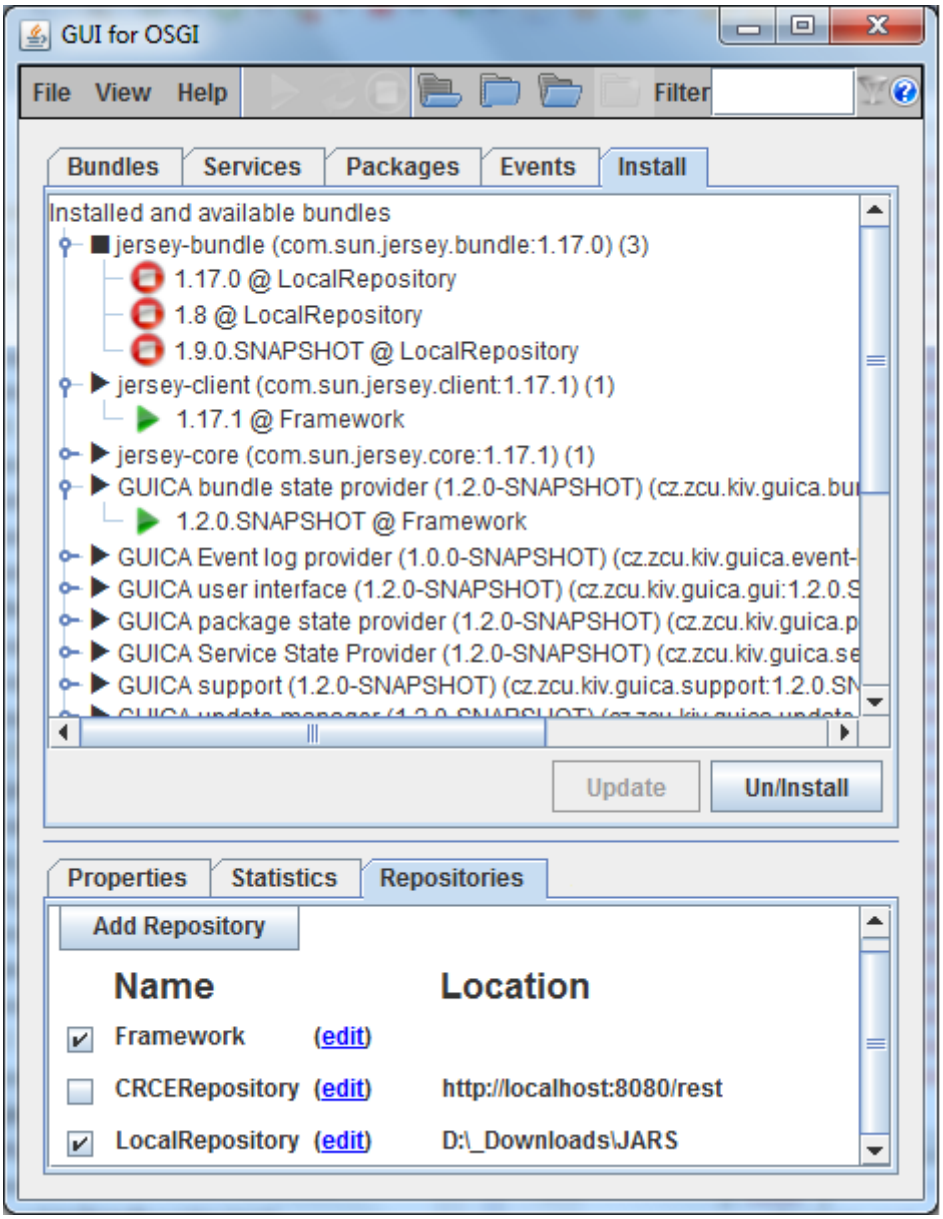
- úprava zobrazení OSGi komponent na záložce „Install“ v GUI
- přidání nového typu úložiště pro CRCE
- implementace REST API klienta
- umožnění aktualizace komponent

Detailnější charakteristika jednotlivých částí vývoje klientské části aplikace GUICA se nalézá v následujících podkapitolách.

9.2. Úprava záložky pro instalaci komponent

Design záložky „Install“ bylo zapotřebí upravit, aby lépe podporoval implementaci aktualizací komponent. Před začátkem úprav byly komponenty zobrazovány v dvouúrovňové stromové struktuře, která neodpovídala novým požadavkům. Tato struktura (kořen -> komponenta) byla nahrazena strukturou tříúrovňovou (kořen -> ? -> komponenta). Bylo však nutné vyřešit problém, jaký objekt by měl reprezentovat nově vzniklou střední úroveň (reprezentována „?“). Za tímto účelem byly navrženy tzv. skupiny komponent, které pod sebou seskupují různé verze téže komponenty (mající stejné symbolické jméno). Tyto skupiny podporují aktualizaci verzí podřízených komponent, která se provádí

právě nad skupinou a ne nad konkrétní komponentou. Nový design bude nejlépe patrný z následujícího screenshotu záložky „Install“ aplikace GUICA.



Obrázek 23: Záložka „Install“ aplikace GUICA

Na obrázku č. 23 je vidět strom skupin i s jejich podřízenými komponentami, které pocházejí ze dvou úložišť. Jedno je nazváno jako „Framework“ a obsahuje komponenty zavedené do OSGi běhového Frameworku po startu aplikace. Druhé s názvem „LocalRepository“ je úložiště založené na souborovém systému, což je patrné i z jeho umístění. Pro lepší pochopení významu zobrazených entit je níže uvedena jejich detailnější charakteristika.

Obrázek 24: Detail skupiny komponent

Rozborem detailního pohledu na rozbalenou skupinu OSGi komponent lze získat následující obecný vzor. Na prvním místě se nachází stavová ikona. Je-li jedna z komponent nainstalována, je ikona ▶, v opačném případě je ■. Pak následuje popis skupiny komponent složený z:

`{BN} ({BSN}:{BV}) ({počet komponent ve skupině})`

- BN (bundle-name) - název OSGi komponenty, označuje se výrazem „human readable“ a představuje pojmenování komponenty pro prezentační účely, které však nemusí být jedinečné
- BSN (bundle-symbolicName) - např. com.sun.jersey.bundle, spolu s verzí tvoří jednoznačný identifikátor komponenty
- BV (bundle-version) - označení konkrétní verze komponenty; pro zobrazení u skupiny se vybere buď první verze v seznamu, nebo ta právě nainstalovaná

Každá samostatná komponenta je pak reprezentována jedním listem zobrazovacího stromu. Zde už je podstatná pouze verze komponenty, protože BN a BSN jsou pro všechny stejné.

Obecný vzor pro zobrazení jednotlivých komponent vypadá následovně:

Nejprve ikona ▶ je-li daná komponenta nainstalována, v opačném případě ■. Pak následují tyto informace:

`{BV}@{název zdrojového úložiště}`

Tímto byla úprava designu záložky pro instalaci komponent z externích úložišť dokončena a aplikace byla připravena k dalším úpravám.

9.3. Implementace podpory REST API

Dalším krokem realizace cílů práce byla implementace komunikačního rozhraní. Jak již bylo zmíněno v analýze, pro komunikaci klientské části aplikace GUICA s úložištěm komponent CRCE byl zvolen protokol založený na konceptu REST API. Jeho specifikace je blíže objasněna v kapitole 8.1.

REST API mezi GUICA a CRCE je na straně klienta reprezentováno jedním Java interface, které deklaruje příslušné metody pro jednotlivé požadavky. Návratovým typem většiny těchto metod je textový řetězec, který obsahuje metadata vrácená CRCE úložištěm. Pokud CRCE vrací OSGi bundle je návratový typ `JarFile.` Každý, kdo toto rozhraní používá, musí tedy potřebná data ze získaných XML metadat extrahovat pomocí mapování XML na objekty. Tomuto procesu je věnována sekce 9.3.1. Rozhraní `RestApi` je implementováno třídou `RestApiImpl`. Tato třída obsahuje instanci typu Client z Jersey Client API sloužící k vlastní komunikaci se serverem na straně CRCE.

K zavolání odpovídající metody na serveru úložiště je nutné správně sestavit URL adresu, na kterou odešle klient požadavek.

URL požadavku protokolu komunikace s CRCE se obecně skládá z:

```
{protokol}://{server}:{port}/rest/{operace}?{parametry}
```

Základ URL tvoří protokol, adresa či název serveru, port a konstanta „/rest“, např.:

```
http://localhost:8080/rest/
```

Takto by vypadala adresa úložiště nastavená v GUICA při přidávání nového úložiště typu CRCE běžící jako služba na tomtéž stroji. Požadavky na CRCE lze volat i bez návaznosti na konkrétní úložiště založené v GUICA, např. z testovacích účelů. V tomto případě se použije vzor URL nastavený v souboru „urlConfig.conf“, který je uložen ve složce „resources“ komponenty Update State Provider.

Celé URL včetně parametrů se pak zkompletuje při volání konkrétní operace REST API na základě obslužné metody a jejích parametrů. Výsledné URL by při volání metody `replaceBundle` parametrem ID vypadalo následovně:

```
http://localhost:8080/rest/replace-bundle?id={ID}
```

Když je celé URL sestaveno, vytvoří se na jeho základě tzv. WebResource (opět třída z Jersey Client API), nad kterým lze volat operace http protokolu (GET, POST, atd.).

9.3.1. Mapování XML na objekty

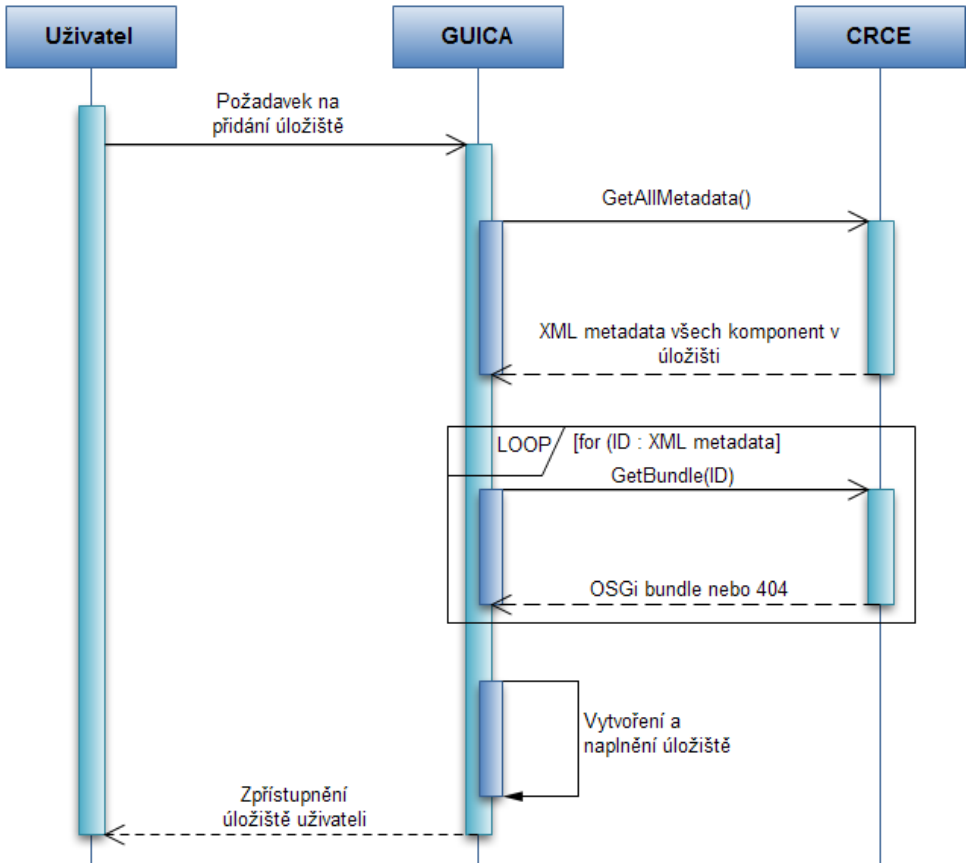
K mapování XML metadat na objekty používané v aplikaci slouží knihovna JAXB. Zdůvodnění jejího použití se nachází v kapitole 7.4. Tato knihovna pracuje se standardními POJO objekty rozšířenými o anotace z balíčku `javax.xml.bind`.
Díky těmto anotacím lze vytvořit třídy, které odpovídají struktuře XML dat. K samotnému mapování se pak využívají instance třídy `JAXBContext`, inicializované s parametrem třídy reprezentující kořenový element XML dat.
Je-li zapotřebí vytvořit XML data z předdefinovaných objektů v aplikaci, stačí použít tzv. `Marshaller` získaný z `JAXBContext`, který je vytvoří. XML data pak mohou být uložena do souboru, nebo jak je tomu v případě REST API, zaslána v těle požadavku pomocí metody http POST.
V opačném případě, tedy je-li zapotřebí XML data mapovat na objekty, použije se tzv. `Unmarshaller` získaný taktéž z `JAXBContext`. Ten pak vytvoří sadu objektů provázaných podle jejich hierarchie v XML, a nastaví jejich příslušné proměnné reprezentující hodnoty XML elementů či jejich atributů.
Pro reálné použití při komunikaci aplikace GUICA s CRCE byly nejprve anotované třídy jednorázově vygenerovány JAXB nástroji z XSD šablony definující strukturu XML zpráv pro REST API a následně přidány mezi ostatní zdrojové kódy na obou stranách komunikačního kanálu. Tyto třídy reprezentovaly strukturu popisných metadat CRCE úložiště tudíž rozšířený OBR formát.
V pozdějších fázích vývoje byly tyto vygenerované třídy vyňaty ze zdrojových kódů, kam logicky nepatří. Začaly se generovat pokaždé znovu a to při sestavování projektu pomocí Apache Maven. Funkci generování tříd na základě XSD šablony při sestavování projektu umožňuje JAXB XJC[1] plugin.

[1] XJC - XML Java Compiler

9.4. Podpora úložiště typu CRCE

Aplikace GUICA je velmi obecné řešení poskytující možnost spravovat různé typy externích úložišť. CRCE mezi nimi ale doposud nebylo. Proto bylo nutné po implementaci REST API zavést ještě nový typ úložiště, který by vyhovoval požadavkům a umožňoval plnohodnotnou spolupráci s CRCE úložištěm.

Pro přidání nového typu úložiště je mimo jiné zapotřebí vytvořit novou implementaci rozhraní `Repository` definující obecný vzor úložiště, kterému musí každý nový typ odpovídat. Toto rozhraní je součástí modulu `Update State Provider`. Nejdůležitější je metoda pro načtení komponent z externího úložiště `loadBundles()`. Tato metoda je automaticky volána v okamžiku uživatelského požadavku na přidání nového úložiště do aplikace GUICA. Pro účely úložiště typu CRCE byla tato metoda implementována pomocí operací REST API. Na následujícím sekvenčním diagramu je znázorněna návaznost operací spojených s přidáním nového úložiště typu CRCE včetně komunikace GUICA a CRCE probíhajícího přes REST API.



Obrázek 25: Sekvenční diagram přidání CRCE úložiště

Na tomto sekvenčním diagramu jsou tři aktéři. První „Uživatel“ představuje uživatele ovládajícího aplikaci GUICA. Aktér „GUICA“ představuje business logiku aplikace GUICA, stejně jako aktér „CRCE“ představuje serverovou část úložiště CRCE. Komunikace mezi GUICA a CRCE probíhá přes REST API, jak již bylo zmíněno.

GUICA používá standardní operace tohoto komunikačního rozhraní, tudíž není třeba je zde znovu objasňovat. Jeden blok by přesto mohl vyžadovat bližší popis a je jím blok označený jako „LOOP“. V rámci tohoto bloku reprezentujícího cyklus jsou iterativně procházena metadata získaná v předešlém kroku voláním metody `GetAllMetadata()`. Z těchto metadat jsou extrahovány identifikátory jednotlivých OSGi komponent v CRCE úložišti, které jsou následně dosazeny za parametr ID v metodě `GetBundle(ID)`, ta je během každé iterace zavolána. Na konci tohoto cyklu má aplikace GUICA na své straně shromážděny všechny dostupné komponenty z CRCE úložiště. Může tak dokončit proces přidání úložiště, jehož obsah je následně zobrazen uživateli.

9.5. Aktualizace komponent

Podpora pro nízkoúrovňový OSGi update komponent zavedených do běhového Frameworku byla v aplikaci implementována již dříve. Využívá standardních funkcí OSGi kontextu. Pro aktualizaci skupin komponent získaných z externích úložišť bylo však nutné přenést tento proces i na vyšší úroveň. Na záložce „Install“ přibylo tlačítko „Update“, ale okolo jeho přesné funkce a podmínkách přístupnosti bylo i po analýze mnoho nejasností. Proběhla tedy malá dílčí implementační analýza, která byla věnována tomuto tématu. Jejím výstupem byly dva případy užití (tzv. UC - Use Case), a ty exaktně určily proces aktualizace skupin komponent za různých podmínek.

UC01 – Update skupiny komponent

Uživatel provádí update skupiny komponent. Po stisknutí tlačítka „Update“ (dále je Update) se zjistí novější verze ve zvolené skupině, které se zobrazí k výběru. Dále se provede nízkoúrovňový OSGi update komponenty tzn.,

původně nainstalovaný bundle se odinstaluje a místo něj se nainstaluje bundle s vybranou verzí.

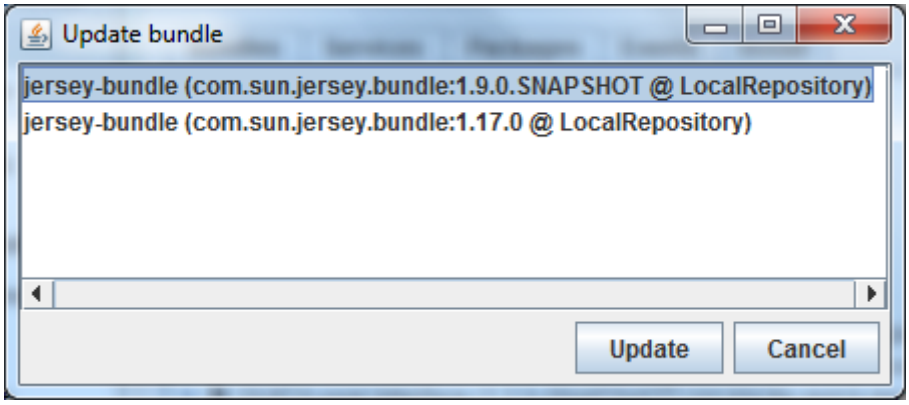
Podmínky pro umožnění aktualizace jsou následující:

- je-li vybrán konkrétní bundle s verzí ve třetí úrovni stromu – tlačítko Update je neaktivní
- je-li vybrán uzel ve druhé úrovni stromu představující skupinu komponent (human readable název, v závorce počet položek pod ním)
    - není-li žádná z položek pod uzlem nainstalována - tlačítko Update je neaktivní
    - je-li některá z položek nainstalovaná, ale její verze je již nejvyšší - tlačítko Update je neaktivní
    - je-li nejbližší vyšší verze z CRCE úložiště, proveď CRCE Update viz UC02
    - je-li některá z položek nainstalovaná a je k dispozici její vyšší verze - tlačítko Update je aktivní a uživatel může provést aktualizaci

UC02 – Update CRCE komponenty

Jelikož komponenty z CRCE úložiště mají navíc metadata o kompatibilitě, jejich update bude probíhat odlišně oproti UC01.

- jsou-li splněny všechny podmínky pro CRCE update:
    1) tlačítko Update se změní v CRCE update
    2) GUICA odešle požadavek na CRCE pro získání všech novějších kompatibilních verzí aktuálně nainstalované komponenty a to metodou replaceBundle(ID).
    3) výsledky jsou zobrazeny uživateli v podokně aktualizace k výběru požadované verze



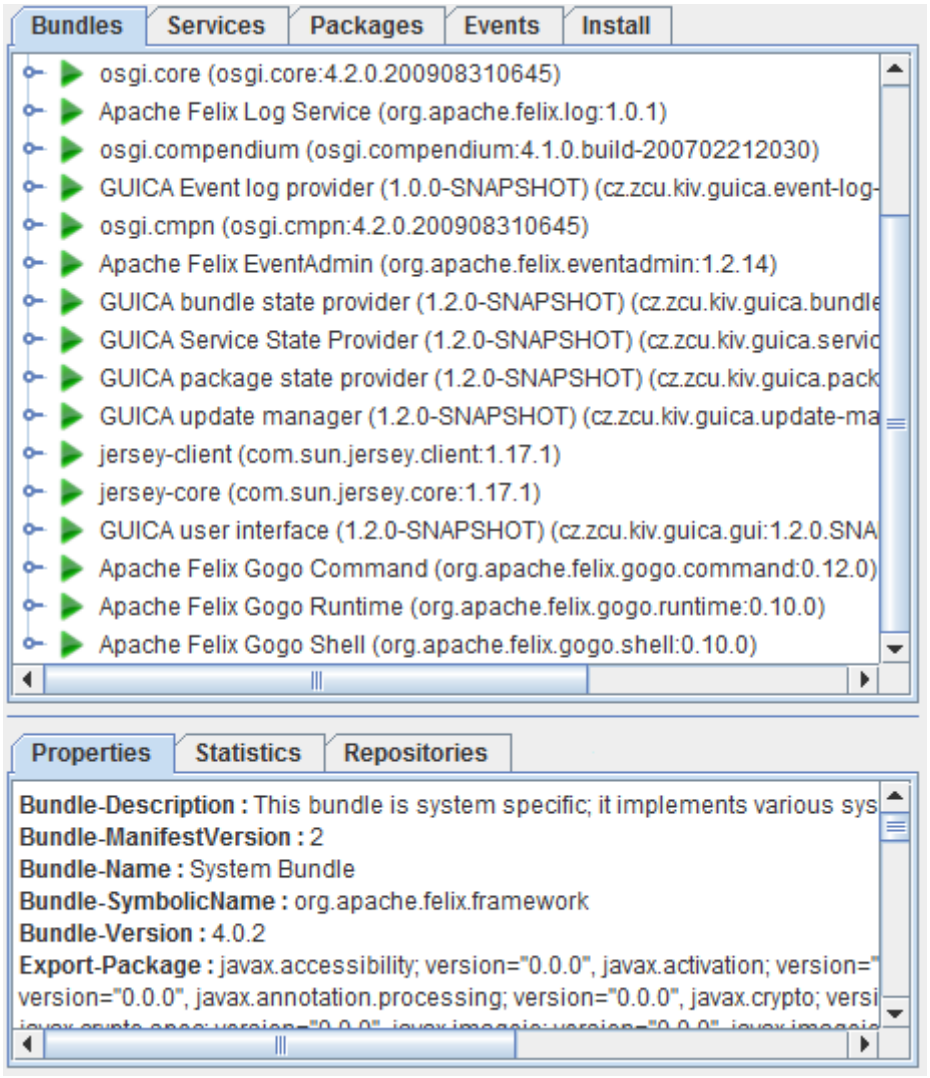
Obrázek 26: Obrazovka aktualizace komponent

Tato obrazovka je pro oba typy aktualizace stejná, jen zobrazené verze komponent se získávají odlišným způsobem popsaným výše. Uživatel zde má na výběr kompatibilní vyšší verze nainstalované komponenty, z nichž musí vybrat právě jednu. Po kliknutí na tlačítko Update se provede aktualizace.

9.6. Shrnutí

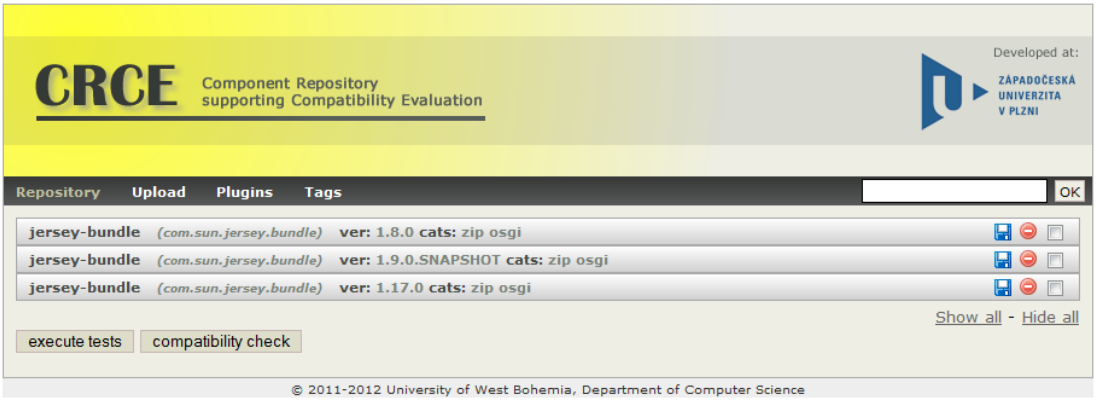
V rámci této kapitoly bude shrnut proces instalace a aktualizace komponent z CRCE úložiště, protože tato funkcionalita je předním cílem celé práce. Pokud je komponenta dostupná v CRCE, je možné ji nainstalovat do běhového OSGi Frameworku spravovaného aplikací GUICA. Obsahuje-li CRCE nějaké novější kompatibilní verze nainstalované komponenty, pak je možné provést i bezpečnou aktualizaci s minimálním rizikem problémů vzniklých v důsledku nekompatibility.

Následuje popis procesu instalace a aktualizace komponenty z CRCE úložiště, který je pro větší názornost doplněn o snímky obrazovek aplikací GUICA a CRCE.



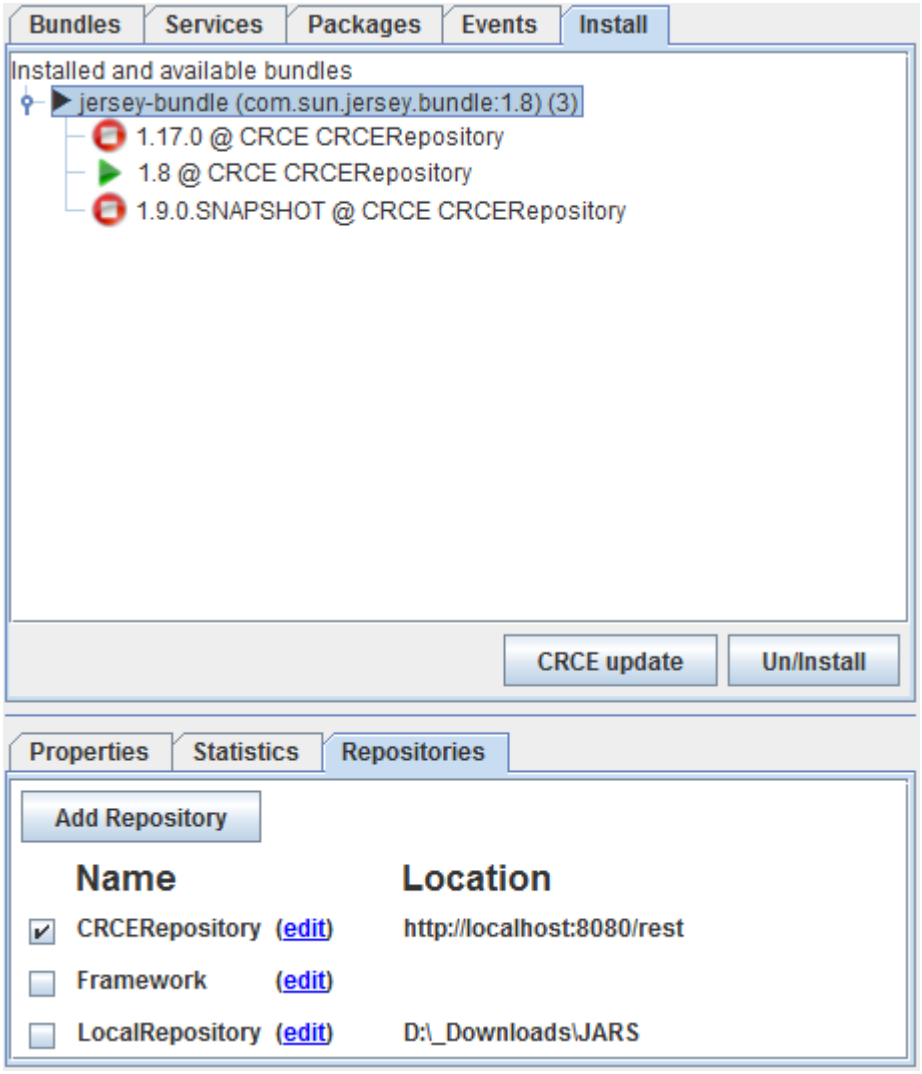
Obrázek 27: Záložka „Bundles“ aplikace GUICA - počáteční stav

Na obrázku 27 je snímek obrazovky zobrazující výčet komponent aktuálně zavedených v OSGi běhovém Frameworku. Tento snímek je uveden pro ilustraci počátečního stavu, po zapnutí aplikace GUICA.



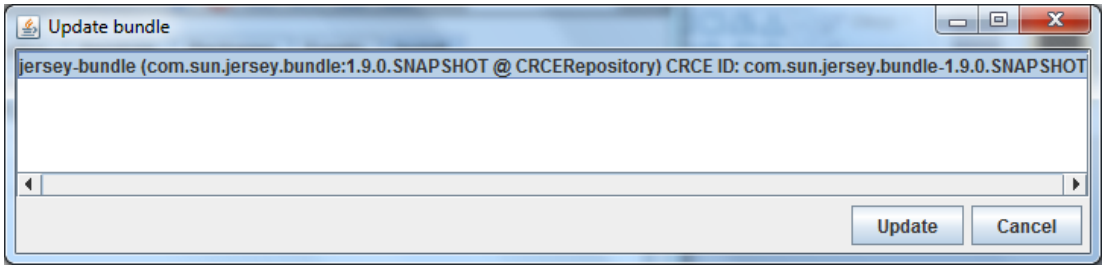
Obrázek 28: CRCE Web UI - výčet dostupných komponent

Obrázek 28 znázorňuje obsah úložiště zobrazený v CRCE webovém uživatelském rozhraní, pomocí kterého byly také komponenty předtím nahrány. Zde je vidět, že se v úložišti nachází tři různé verze komponenty `jersey-bundle`.



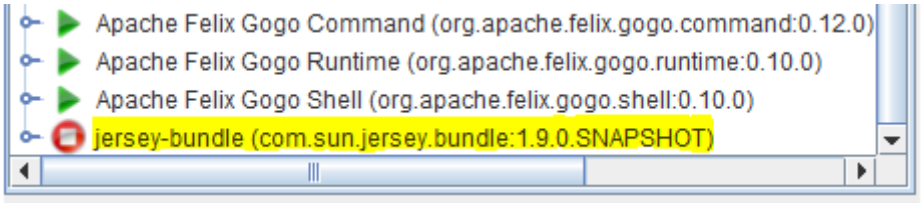
Obrázek 29: Záložka „Install" aplikace GUICA - komponenty z CRCE úložiště

Snímek na obrázku 29 zobrazuje obsah CRCE úložiště na záložce „Install" aplikace GUICA. Jsou zde vidět stejné verze komponenty `jersey-bundle` jako v CRCE Web UI. CRCE úložiště však muselo být nejprve do aplikace GUICA přidáno pomocí tlačítka „Add Repository". Má název „CRCERepository" a běží na stejném stroji jako GUICA, což znázorňuje jeho umístění (Location). Pro zobrazení komponent z CRCERepository bylo nutné označit pole pro zaškrtnutí tak, jak je vidět na obrázku. Poté byla nainstalována jedna z verzí komponenty konkrétně verze 1.8, což zpřístupnilo tzv. CRCE update skupiny.



Obrázek 30: Obrazovka aktualizace komponent - CRCE update

Po stisknutí tlačítka „CRCE update“ se zavolá metoda `replaceBundle` z REST API s parametrem identifikátoru aktuálně nainstalované OSGi komponenty v dané skupině. V tomto případě byl identifikátor „com.sun.jersey.bundle-1.8“. CRCE vrátilo nejbližší vyšší kompatibilní verzi „com.sun.jersey.bundle-1.9.0.SNAPSHOT“, což je vidět na obrázku 30.



Obrázek 31: Část záložky „Bundles“ aplikace GUICA zobrazující konečný stav

Poslední snímek zobrazuje část záložky „Bundles“, která nyní obsahuje novou aktualizovanou komponentu `jersey-bundle`. Obrázek 31 slouží hlavně k ilustraci rozdílu mezi počátečním stavem (před instalací a aktualizací komponenty z CRCE úložiště) a stavem konečným zachyceným právě zde.

10. Závěr

Cílem práce bylo vytvořit implementaci klienta úložiště CRCE, která by využila jeho možnosti poskytovat informace o kompatibilitě komponent. Realizace vedla k rozšíření aplikaci GUICA tak, aby splňovala nároky kladené na klienta umožňujícího aktualizaci softwarových komponent získaných nejen z CRCE úložiště. To vyžadovalo dodatečnou funkcionalitu jakou je např. získání popisných metadat komponenty nebo její novější kompatibilní verze. Cíle práce byly naplněny, GUICA nyní podporuje využívání komponent z CRCE a umožňuje jejich bezpečnou aktualizaci.

Jelikož GUICA vystupuje jako klient a CRCE jako server, bylo nutné použít vhodný komunikační protokol. K tomuto účelu bylo vybráno REST API, jehož návrh byl z velké části vytvořen zpracovatelem diplomové práce, viz [15]. Toto komunikační rozhraní umožňuje z CRCE získávat komponenty a informace o nich ve formě metadat a je-li za potřebí tak i kompatibilní verze již stažených/nainstalovaných komponent.

Kromě CRCE lze nyní využít i jiné typy úložišť jako například čistý souborový systém či úložiště formátu OBR. Při analýze byl kladen velký důraz na optimalizaci hardwarových nároků aplikace z důvodu možného budoucího využití v mobilních či různých vestavěných zařízeních. Vize do budoucna je totiž taková, že by firmware takových zařízení mohl využívat OSGi a mít komponentovou architekturu. V kombinaci s přístupem na internet například s pomocí WIFI by se mohl software automaticky aktualizovat, bez nutnosti restartu JVM, ve které by byla aplikace spuštěna a také bez nutnosti zásahu technika, jenž by musel firmware přeinstalovat. CRCE úložiště zajišťuje konzistentní kompatibilitu komponent v rámci celé aplikace, což slouží jako prevence před pádem aplikace při aktualizaci jedné z komponent a tudíž je upgrade software velmi bezpečný.

Na základě zkušeností nabytých při realizaci klientské aplikace by bylo možné do budoucna doporučit případný rozvoj spolupráce systémů GUICA a CRCE. REST API na straně CRCE totiž podporuje více funkcí, než kolik klient v současné době využívá. Jistě by bylo vhodné zaměřit se na implementaci aktualizace

komponent na úrovni celé aplikace, či nalézt využití pro již implementované, ale zatím nevyužité funkce. Z pohledu samotné aplikace GUICA lze doporučit rozšíření ve formě podpory dalších typů úložišť např. Maven Repository nebo přenos uživatelského rozhraní na nějakou mobilní platformu jakou je např. OS Android.

Literatura

[1] Trueman, Philip, et al.: *Component-based software engineering* [online]. 27.5.2014 [cit. 2014-05-30]. URL <http://en.wikipedia.org/wiki/Component-based_software_engineering>

[2] OSGi Alliance: *OSGi Technology* [online]. 19.12.2013 [cit. 2014-04-16]. URL <http://www.osgi.org/Technology/HomePage>

[3] OSGi Alliance: *About the OSGi Alliance* [online]. 23.12.2013 [cit. 2014-04-16]. URL <http://www.osgi.org/About/HomePage>

[4] OSGi Alliance: *The OSGi Architecture* [online]. 23.12.2013 [cit. 2014-04-18]. URL <http://www.osgi.org/Technology/WhatIsOSGi>

[5] Allamaraju, Subbu: *RESTful web services cookbook*. O'Reilly 2010

[6] Oracle Corporation: Jersey - RESTful Web Services in Java [online]. 26.5.2014 [cit. 2014-05-30]. URL <https://jersey.java.net/>

[7] Pericas-Geertsen, S., Potociar, M.: *JAX-RS: Java™ API for RESTful Web Services, Version 2.0 Final Release.* [online PDF]. Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065 USA. 22.5.213. URL <http://download.oracle.com/otn-pub/jcp/jaxrs-2_0-fr-eval-spec/jsr339-jaxrs-2.0-final-spec.pdf>

[8] Oracle Corporation: *Jersey 2.9 User Guide* [online]. 24.4.2014 [cit. 2014-05-10]. URL <https://jersey.java.net/documentation/latest/user-guide.html>

[9] Vogel, L.: *REST with Java (JAX-RS) using Jersey - Tutorial* [online]. 28.52014 [cit. 2014-06-02]. URL <http://www.vogella.com/tutorials/REST/article.html>

[10] Restlet, SAS.: *Learn Restlet Framework* [online]. Restlet, Inc. 2014 [cit. 2014-06-05]. URL <http://restlet.com/learn/guide/2.3/>

[11] Restlet, SAS.: *Restlet improves OSGi support: new edition and update site* [Online]. Restlet, Inc. 9.11.2011 [cit. 2014-06-07]. URL

<http://blog.restlet.com/2011/11/09/restlet-improves-support-for-osgi-with-new-edition-and-eclipse-update-site/>

[12] *Apache Felix OSGi Bundle Repository (OBR)* [online]. The Apache Software Foundation, 5.5.2011 [cit. 2014-05-24]. URL <http://felix.apache.org>

[13] *Apache Maven* [online]. The Apache Software Foundation, 2002 -2014 [cit. 2014-05-26]. URL <http://maven.apache.org/>

[14] KUČERA, Jiří. *Uložiště komponent podporující kontroly kompatibility*. Plzeň, 2011. Diplomová práce. Západočeská univerzita. Vedoucí práce Doc. Ing. Přemysl Brada, MSc., Ph.D.

[15] ŘEZNÍČEK, Jan. *Využití úložiště komponent pro podporu aktualizace aplikací*. Plzeň, 2013. Diplomová práce. Západočeská univerzita. Vedoucí práce Doc. Ing. Přemysl Brada, MSc., Ph.D.

[16] Brada, P., et al: *CRCE – Component Repository supporting Compatibility Evaluation* [projekt wiki]. 1.2.2013 [cit. 2014-05-11]. URL <https://www.assembla.com/spaces/crce/wiki>

[17] Brada, P., Ježek, K.: *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.

[18] *Eclipse Packaging Project (EPP)* [online]. The Eclipse Foundation, 2014 [cit. 2014-05-15]. URL<http://www.eclipse.org/epp/>

[19] *Eclipse Marketplace Client* [online]. The Eclipse Foundation, 2014 [cit. 2014-05-15]. URL <http://www.eclipse.org/mpc/>

[20] Guindon, C.: *Eclipse Marketplace REST API Documentation* [online]. The Eclipse Foundation, 14.3.2013 [cit. 2014-05-15]. URL <http://wiki.eclipse.org/Marketplace/REST>

[21] Haustein, S.: *kXML* [online]. 2001-2014 [cit 2014- 06-10]. URL <http://kxml.org/>

[22] Vajjhala, S., Fialli, J.: *Java™ Architecture for XML Binding (JAXB) 2.0* [online PDF]. Sun Microsystems, Inc. 19.2.2006 [cit. 2014-06-10]. URL <http://download.oracle.com/otndocs/jcp/jaxb-2.0-fr-eval-oth-JSpec/>

[23] Bergemann, T., et. al.: *Package management system* [online]. 20.5.2014 [cit. 2014-06-07]. URL <http://en.wikipedia.org/wiki/Package_management_system>

[24] Aoki, O.: *Debian package management, Chapter 2* [online]. Debian Free Software Guidelines 3.12.2013 [cit. 2014-06-07] URL <http://www.debian.org/doc/manuals/debian-reference/ch02.en.html>

[25] Marley, B., Tosh, P.: *RPM Package File Structure, Chapter 24* [online]. Fedora Documentation 21.7.2002 [cit. 2014-06-09]. URL <http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch-package-structure.html>

[26] Ubuntu Community: *SynapticHowto* [online]. 14.12.2013 [cit. 2014-06-09]. URL <https://help.ubuntu.com/community/SynapticHowto>

[27] Thomas M.P.: *SoftwareCenter* [online]. 29.8.2005 [cit. 2014-06-10]. URL <https://wiki.ubuntu.com/SoftwareCenter>

[28] Macon, G., et. al.: *Aptitude (software)* [online]. 9.4.2014 [cit. 2014-06-10]. URL <http://en.wikipedia.org/wiki/Aptitude_%28software%29>

[29] Brodowsky, K. et. al.: *RubyGems* [online]. 28.5.2014 [cit. 2014-06-12]. URL <http://en.wikipedia.org/wiki/RubyGems>

[30] *RubyGems Guides, RubyGems.org API* [online]. RubyGems.org 2014 [cit. 2014-06-12]URL <http://guides.rubygems.org/rubygems-org-api/>

[31] *Apache Aries* [online] The Apache Software Foundation, 2014 [cit. 2014-05-26]. URL <http://aries.apache.org/>

[32] *Apache Karaf* [online] The Apache Software Foundation, 2008-2011 [cit. 2014-05-26]. URL <http://karaf.apache.org/manual/latest/overview.html>

[33] *Apache Ace* [online] The Apache Software Foundation, 2012-2014 [cit. 2014-05-26]. URL < http://ace.apache.org/>

Seznam zkratek

CRCE - Component Repository Supporting Compatibility Evaluation
GUICA - Graphical User Interface for Component Administration
OSGi - Open Services Gateway initiative
JVM - Java Virtual Machine
JAR - Java Archive
API - Application Programing Interface
JSR - Java Specification Request
RFC - Request For Comments
SPI - Service Provider Interface
JSON - JavaScript Object Notation
HTTP - Hypertext Transfer Protocol
POJO - Plain Old Java Object
URL - Uniform Resource Locator
URI - Uniform Resource Identifier
DI - Depencency Injection
JAXB - Java Architecture for XML Binding
RSS - Rich Site Summary
WADL - Web Application Description Language
JDBC - Java Data Base Connectivity
Java SE - Java Standard Edition
GWT - Google Web Toolkit
SMTP - Simple Mail Transfer Protocol
OBR - OSGi Bundle Repository
XML - Extensible Markup Language
LDAP - Lightweight Directory Access Protocol
WAR - Web Application Resource
POM - Project Object Model
IDE - Integrated Development Environment
EPP - Eclipse Packaging Project
RPM - Red Hat Package Manager
GUI - Graphical User Interface
YAML - Yet Another Markup Language
JMX - Java Management Extensions
JNDI - Java Naming and Directory Interface
JPA - Java Persistence API
BN - Bundle-Name
BSN - Bundle-SymbolicName
BV - Bundle-Version
UC - Use Case

Seznam obrázků