České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Alexandr Makarič**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Modulární architektura restauračního systému**

Pokyny pro vypracování:

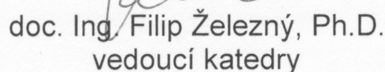
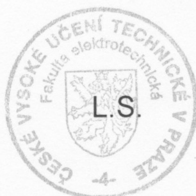
Analyzujte, navrhněte, implementujte, otestujte a dokumentujte řešení problematiky modulární architektury restauračního systému Cashbob. Hlavním cílem je možnost rozšíření funkcionality nasazeného systému bez nutnosti jeho reinstalace.

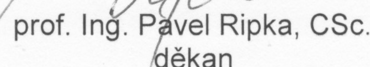
Seznam odborné literatury:

[1] Cummins, Holly; Ward, Tim (March 28, 2013), Enterprise OSGi in Action (1st ed.), Manning Publications, p. 376, ISBN 978-1617290138
[2] McAffer, Jeff; VanderLei, Paul; Archer, Simon (February 24, 2010), OSGi and Equinox: Creating Highly Modular Java Systems (1st ed.), Addison-Wesley Professional, p. 460, ISBN 0-321-58571-2

Vedoucí: Ing. Martin Komárek

Platnost zadání: do konce letního semestru 2014/2015

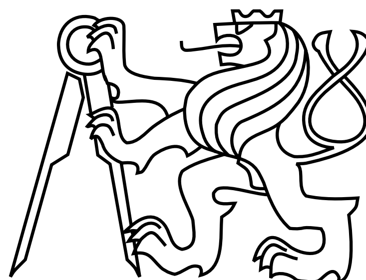


doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 28. 2. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Modulární architektura restauračního systému

Alexandr Makarič

Vedoucí práce: Ing. Martin Komárek

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

22. května 2014

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Martinu Komárkovi a mé matce Janě Makaričové.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 20. 5. 2014 ....

Abstract

The aim of this thesis was to analyze the current project of restaurant management system CashBob and propose a set of guidelines and rules to transfer the project to take advantage of OSGi component model for the benefit of adding new functionality without the need to stop the running system. The resulting implementation of these rules was then tested on the client part of the application and structured for the use by future developers of the project.

Abstrakt

Cílem této práce bylo analyzovat stávající projekt restauračního systému CashBob a navrhnout postup, jakým tento projekt upravit tak, aby využíval komponentového modelu OSGi a bylo možné ho za běhu rozšiřovat o novou funkcionalitu bez nutnosti zastavovat běžící aplikaci. Daný návrh bylo potřeba implementovat, otestovat na klientské části aplikace a navrhnout pravidla, kterými se budou řídit vývojáři nových částí aplikace.

Obsah

1 Úvod **1**

1.1 Services (Služby) 2
1.2 Bundl 3
1.2.1 Manifest - OSGi hlavičky 4
1.2.2 Životní cyklus bundlu 4
1.2.3 Skrývání informací 5
1.3 Principy 5
1.3.1 GRASP 5
1.3.2 SOLID 6
1.3.3 KISS + DRY 6
1.4 Nástroje 6
1.5 Testování 6

2 Restaurační systém CashBob **7**

2.1 Architektura 8
2.1.1 Problém 8
2.1.2 Řešení OSGi 8

3 Iterace I 0 Minimální nasazení OSGi **9**

3.1 Úvod 9
3.1.1 Cíl 9
3.2 Iterace 9
3.2.1 BundleActivator 11
3.2.2 Závislosti 13
3.2.3 Externí knihovny 14
3.3 Závěr 14

4 Iterace II - Izolace modulu **15**

4.1 Úvod 15
4.1.1 Cíl 15
4.2 Iterace 15
4.2.1 Úpravy 17
4.3 Závěr 20

5 Iterace III - Dynamická komponenta **21**
5.1 Úvod 21
5.1.1 Cíl 21
5.2 Iterace 21
5.3 Závěr 24

6 Závěr **25**
6.1 Restaurační systém CashBob 26
6.2 Výsledek 26

Literatura **27**

A Seznam zkratek **29**

B Instalační manuál **31**

C Obsah CD **33**

Seznam obrázků

1.1 Architektura OSGi frameworku [6] 3
1.2 Životní cyklus bundlu [4] 5

3.1 Dialog spouštěcí konfigurace OSGi projektu 10

Seznam tabulek

1.1 Výběr OSGi hlaviček 4

2.1 Moduly projektu CashBob 7

Kapitola 1

Úvod

V průmyslu se na každém kroku setkáváme se standardizací, v našich končinách nejčastěji reprezentovanou normami ČSN a ISO. Výhody jsou zřejmé: jednotné součástky, jednotné nástroje, jednotné postupy. To vše vede ke snadné zaměnitelnosti a znovupoužitelnosti. Rozhodneme-li se vyrobit například nový automobil, nemusíme znovu vymýšlet kola, matky a šrouby. Použijeme stávající části a pouze pozměníme způsob, jakým vše sestavíme dohromady. Případně vytvoříme novou komponentu, vycházející ze stávajících součástí, ale nejlépe tak, abychom ji mohli použít i v dalším projektu.

S obdobným jevem se setkáváme i při vývoji softwarových aplikací. Již od prvních programovacích jazyků se snažíme brát dobré myšlenky, separovat je z existujících programů, zobecňovat je, abychom je mohli snáze používat znovu a znovu. Nové technologie a koncepty, jako například funkcionální programování a objektový pohled na svět, s sebou přinášejí nové možnosti, jak elegantněji řešit problémy. Říká se sice, že: „Všechny cesty vedou do Říma“, ale v případě vývoje softwarových aplikací ne všechny postupy, na jejichž konci je fungující aplikace, jsou ideální. Každá nová technologie nám přináší nové „cesty“, jak stavět software, a my se musíme snažt nacházet ty neschůdnější postupy, které se snadno používají. Ty sdružujeme do knihoven, univerzálních modulů, které pouze s malými obměnami můžeme používat znovu a znovu.

Myšlenka programovacího jazyku Java, že je možné program přeložit do univerzálního byte-kódu a ten spustit kdekoliv, je dalším krokem ke znovupoužitelnosti myšlenek. „Write once, run anywhere“ býval slogan Javy. Bohužel časem se Java rozdělila na několik odnoží: konkrétně Java SE, Java ME a Java EE.

Každá z těchto odnoží se přizpůsobuje jinému prostředí, ve kterém je aplikace spouštěna, a nabízejí tedy různé modely a koncepty řešící modularizaci a znovupoužitelnost kódu [2]. I vývojáři různých aplikací a frameworků, například JBoss nebo NetBeans, řeší otázku modularizace po svém a vznikají různé proprietární modularizační vrstvy s vlastnými mechanismy načítání tříd, packagování, nasazování (deployment) a řešení závislostí [6].

Cílem OSGi Alliance, původně sdružením výrobců malých domácích embedded zařízení jako například set-top-boxy nebo home gateway zařízení nazývaném Open Service Gateway Initiative, je vytvořit společnou a otevřenou platformu pro poskytovatele služeb, vývojáře, správce systémů a prodejce softwaru a příslušenství ke koordinovanému vytváření, nasa-

zování a správě služeb. OSGi vytváří specifikaci[1] takového frameworku, jehož primárním cílem je využití platformní nezávislosti a schopnosti dynamického načítání kódu jazyka Java k usnadnění vývoje a dynamického nasazení aplikací na zařízení s omezenými paměťovými nároky [6], přičemž není nutné se omezovat pouze na embedded zařízení. Velké množství vývojářů aplikací a frameworků již adaptovalo OSGi model jako základ svých projektů nebo se k němu začíná přiklánět. Můžeme zmínit například platformu Eclipse nebo aplikační servery JBoss, GlassFish, Websphere a další [1].

Framework se snaží nespoléhat na prostředí, ve kterém je spuštěn a snahou vývojáře aplikace má být podobné smýšlení. Především jde o oddělení vlastní implementace služby od její specifikace (typicky reprezentovanou pomocí Java interface). Důsledkem je, že aplikace je méně svázána s prostředím, ve kterém je spouštěna, neboť pro nasazení na konkrétní platformu se použije patřičná implementace služby. Příkladem může být logování událostí.

V případě, že je aplikace nasazena na standartním PC, není problém události logovat do souboru na disk. V případě nenáročného embedded zařízení s omezenou kapacitou je příhodnější logovat na vzdálený server. Vývojáři služby pro logování vždy implementují podle stejné specifikace stejně jako vývojáři aplikace, která bude službu využívat. Interface pak vytváří bod, kde dochází k odstínění kódu vyžadujícího službu a její implementace.

Další vlastnost, kterou framework přináší, je možnost rozdělení aplikace na drobnější instalovatelné komponenty, kde každá má svůj životní cyklus. Tyto komponenty se nazývají bundles[2]. Bundly je možné přidávat či odebírat dle potřeby za běhu, bez nutnosti zastavovat celý systém. Když je bundl nainstalován do systému, může zaregistrovat libovolné množství služeb, které poskytuje pro ostatní bundly.

Framework pracuje s následujícími klíčovymi prvky:

- Služby - Java třídy poskytující danou funkcionalitu
- Bundly - základní jednotka modularizace nesoucí funkcionalitu
- Bundle Context - souhrnné prostředí, v němž jsou bundly spouštěny uvnitř Frameworku

1.1 Services (Služby)

Služba je uzavřená komponenta dostupná skrze její rozhraní (interface). OSGi aplikaci si můžeme představit jako sadu kooperujících služeb. Framework se stará o mapování konkrétních služeb na jejich implementace a poskytuje mechanizmus k vyhledávání jejich dostupných implementací. Jak již bylo řečeno, je služba definována rozhraním a její implementací.

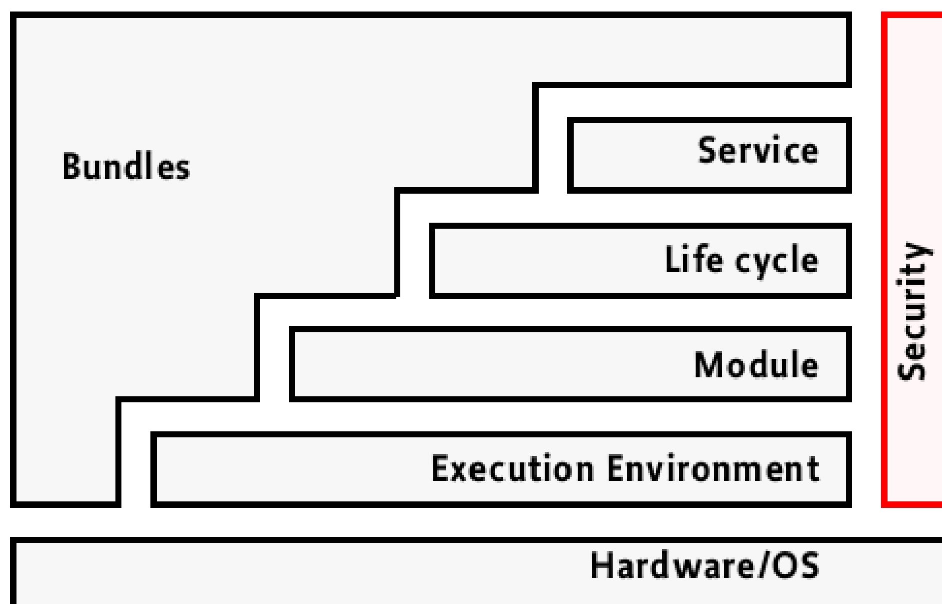
K registraci služeb daného bundlu do frameworku se typicky využívá třídy implementující rozhraní BundleActivator, která, pokud je přítomna, je použita k inicializaci bundlu a registraci služeb. Protože ale tento postup byl poněkud těžkopádný neboť bylo nutné pro

[1] V době psaní této práce byla aktuální verze R5, která přinesla pouze pár drobných změn. Většina této práce se odkazuje především na verzi R4.

[2] V češtině je možná nejvhodnější použít termín modul, nicméně tento pojem budu používat v jiném kontextu. Dále v textu budu používat počeštěný termín bundl/bundly.

každý bundl obsahující službu typicky implementovanou jednoduchou POJO třídou definovat novou aktivační třídu s pokaždé téměř identickou funkcionalitou, vznikly různé projekty, které automatizovaly a zjednodušovaly tento proces.

Následně přinesla verze R4 specifikace požadavek, aby tuto funkcionalitu zajišťoval přímo framework. Vznikl model Declarative Services (DS). Ten využívá konfigurace pomocí XML souboru a zbavuje potřeby přidávat do bundlu kód, který nesouvisí s funkcionalitou služby.



Obrázek 1.1: Architektura OSGi frameworku [6]

1.2 Bundl

Specifikací definovaná jednotka modularizace se nazývá bundl. Bundl může obsahovat kód a další zdroje nutné k implementaci služeb, přičemž jeden bundl může implementovat více různých služeb nebo také žádnou. Bundl je ve své podstatě obyčejný JAR soubor, který má svůj MANIFEST.MF soubor doplněn o hlavičky OSGi. Tyto hlavičky framework zparsuje a dle nich správným způsobem bundl nainstaluje do systému a aktivuje ho.

Takovouto minimální úpravou je možné nasadit libovolný kód zabalený v JAR souboru bez nutnosti zasahovat do výkonného kódu. Pro každý bundl, nainstalovaný do frameworku je vytvořen samostatný Classloader, který defaultně načítá pouze třídy a zdroje v bundlu samotném, základní třídy frameworku a Java packages z ostatních bundlů explicitně vyjmenované v hlavičce Import-Package.

Hlavička	Význam
Bundle-ManifestVersion	Verze OSGi hlaviček manifestu. Pro specifikaci verze R3 je použita hodnota 1, pro verzi R4 hodnota 2, přičemž následující verze specifikace mohou tuto hodnotu změnit.
Bundle-SymbolicName	(*povinná*) Jedinečný identifikátor bundlu. Typicky se používá identifikátor balíčku nejvyšší úrovně v daném bundlu.
Bundle-Version	Verze bundlu. V kombinaci s Bundle-SymbolicName jedinečně identifikují konkrétní bundl.
Bundle-Name	Člověkem čitelný název bundlu.
Bundle-Activator	Třída určená ke korektnímu spuštění bundlu. Implementující rozhraní BundleActivator s metodami start a stop.
Export-Package	Čárkami oddělený seznam Java balíčků, které bundl veřejně zpřístupňuje ostatním.
Import-Package	Čárkami oddělený seznam Java balíčků, které daný bundl vyžaduje ke svému běhu. Předtím než Framework může změnit stav bundlu na RESOLVED, musí všechny tyto balíčky nalézt v ostatních nainstalovaných bundlech a zpřístupnit je v Class-Loaderu tohoto bundlu.

Tabulka 1.1: Výběr OSGi hlaviček

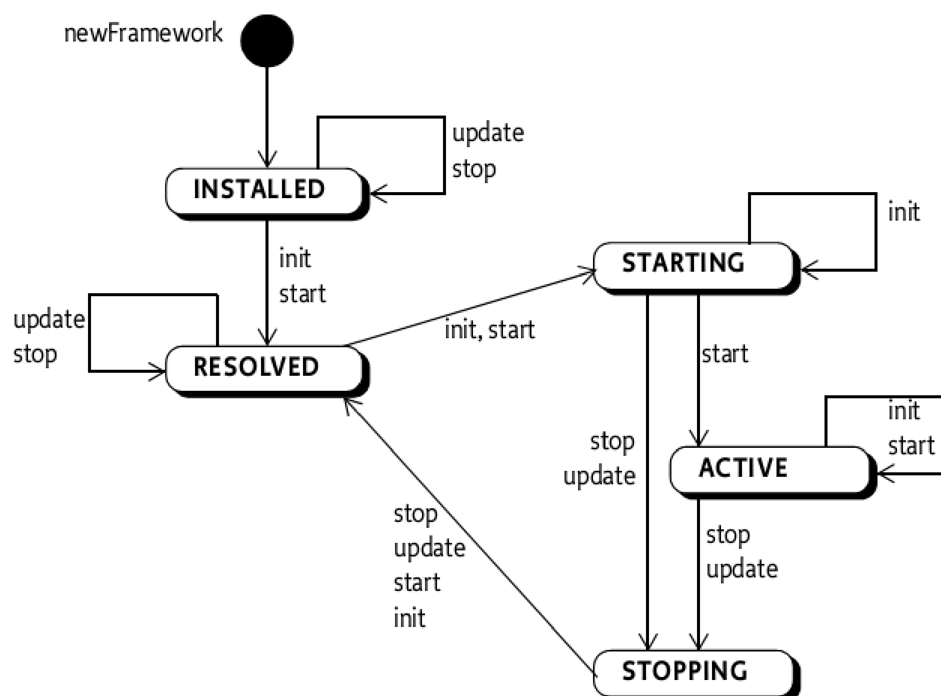
1.2.1 Manifest - OSGi hlavičky

1.2.2 Životní cyklus bundlu

Framework splňující specifikaci OSGi obhospodařuje nasazování bundlů do běžící instance aplikace, registraci služeb a řízení životního cyklu bundlu. Bundl se může nacházet v jednom z následujících stavů:

- INSTALLED - V tomto stavu se nachází po tom, co byl úspěšně nainstalován.
- RESOLVED - V tomto stavu jsou všechny závislosti na ostatních třídách a nativní kódu vyřešeny a zpřístupněny bundlu. V tomto stavu je bundl připraven ke spuštění. Bundl, který byl zastaven, se vrací do tohoto stavu.
- STARTING - Indikace spouštění bundlu.
- STOPPING - Indikace zastavování bundlu.
- ACTIVE - Bundl byl úspěšně spuštěn a běží.
- UNINSTALLED - Bundl byl odinstalován z frameworku a již nemůže nabýt jiného stavu.

Informace o spuštění nebo zastavení bundlu (stavy ACTIVE a RESOLVED) se zachovává mezi restarty frameworku.



Obrázek 1.2: Životní cyklus bundlu [4]

1.2.3 Skrývání informací

Implicitně jsou všechny balíčky bundlu pro ostatní neviditelné. Pouze balíčky explicitně vyjmenované v hlavičce *Export-Packages* framework zpřístupňuje a spravuje jejich závislosti. Analogicky pouze balíčky, které jsou bundlem deklarovány jako závislosti hlavičkami *Import-Package* nebo *Require-Bundle*, mu jsou zpřístupněny.

1.3 Principy

Jak již bylo zmíněno v úvodu, s každou technologií se vyvinuly i doporučené postupy ověřené praxí. V případě objektového modelu, který využívá Java, bych rád zmínil několik z nich, které jsem se snažil během práce dodržovat, protože v některých případech mohou být rozhodujícím argumentem pro výběr řešení při přechodu na OSGi model[5].

1.3.1 GRASP

(General Responsibility Assignment Software Patterns) Sada vodítek a návrhových vzorů, ze kterých vybírám především:

- High cohesion - Rozhodovací vzor, jehož cílem je udržovat funkcionalitu dané komponenty úzce zacílenou k řešení daného problému.

- Low coupling - Značí nízkou závislost na ostatních komponentách systému
- Information expert - Princip přiřazující odpovědnost za vykonání úkolu komponentě, která má dostatečné množství informací k jeho dokončení.

1.3.2 SOLID

Je akronymem pro:

- Single responsibility - Třída má nést odpovědnost pouze za jedinnou věc.
- Open-closed principle - Softwarová entita by měla být otevřená pro rozšíření, ale uzavřena pro úpravy.
- Liskov substitution - Objekty v programu by měly být nahraditelné instancemi jejich podtypů bez změny korektnosti programu.
- Interface segregation - Je lépe mít více roztříštěných rozhraní než jedno řešící vše.
- Dependency inversion - Moduly by měly záviset na abstrakcích, nikoli na implementacích.

1.3.3 KISS + DRY

Dvě související vodítka objektově orientovaného návrhu: „Keep It Simple Stupid“ a „Don't Repeat Yourself“. KISS princip souvisí s výše zmíněnými principy High cohesion a Low coupling. K DRY principu se váže také pravidlo tří.

1.4 Nástroje

K nasazování na OSGi model jsem použil implementace frameworku Equinox, která je považována za referenční. Dalším důvodem bylo, že je použita v knize [2], kterou jsem používal jako výchozí bod. Pokud je aplikace závislá čistě na OSGi modelu dle dané specifikace, nemělo by činit žádné problémy nasadit ji na jiný framework. Na to je třeba pamatovat a vyhnout se už při vývoji závislostem na proprietární funkcionalitě frameworku.

Jako vývojové prostředí jsem použil Eclipse for RCP and RAP Developers, což je prostředí zaměřené právě na vývoj modulárních softwarových aplikací, pluginů a právě i aplikací využívajících OSGi model, neboť právě platforma Eclipse od její verze 3.0 přešla na OSGi komponentový systém.

1.5 Testování

Protože hlavní podstatou této práce není přidávat novou funkcionalitu, bude testování mít formu úspěšného sestavení a spuštění upravovaných částí aplikace. Pokaždé, kdy je spuštěna klientská část aplikace, je spuštěn i server.

Kapitola 2

Restaurační systém CashBob

CashBob je projekt restauračního systému určeného pro komplexní správu procesů v restauračním zařízení. Je postaven na architektuře server-klient s neomezeným počtem klientů. Server obstarává perzistenci dat v databázi a veškerou doménovou logiku. Klientská část aplikace obsahuje pouze vrstvu obsluhující komunikaci se serverem a logiku uživatelského rozhraní. Funkcionalita projektu je rozdělena do modulů:

Modul	**Význam**
CashBob	Hlavní modul klientské části aplikace
CashBobLibrary	Knihovna sdíleného kódu a zdrojů
CashBobManager	Správa nastavení a základních doménových dat
CashBobPokladna	Klíčová funkcionalita aplikace, obsluha menu, objednávek a plateb
CashBobRestLib	Podčást knihovního modulu obsahující třídy zajišťující komunikaci mezi klientem a serverem
CashBobServer	Hlavní modul serverové části aplikace
CashBobStorage	Správa skladových zásob
CashBobWorkShift	Správa pracovních směn zaměstnanců
CashBobBuildClient	Bez funkcionality, určeno k sestavení klienta
CashBobBuildServer	Bez funkcionality, určeno k sestavení serveru
CashBobBuild	Bez funkcionality, určeno k sestavení celé aplikace

Tabulka 2.1: Moduly projektu CashBob

Sestavení celého projektu je řízeno pomocí nástroje Maven[3]. Každý modul obsahuje soubor pom.xml, který definuje závislosti na externích knihovnách, způsob překladu, parametry sestavení a další konfiguraci daného modulu. Poslední tři moduly neobsahují žádný výkonný kód, pouze mají definovány závislosti na ostatních modulech, a tím určují, ze kterých modulů se klient resp. server skládá.

2.1 Architektura

Jak již bylo zmíněno, projekt CashBob stojí na architektuře server-klient. Serverová část používá k uchování dat databázový systém H2, se kterým pracuje skrze framework Hibernate. K obsluze požadavků klientů je použito frameworku Spring. Server poskytuje pro připojení klientů REST API, které je celé implementováno vlastním proprietárním kódem. Klientská část má minimální závislosti na externích knihovnách. Obsahuje komunikační vrstvu a controller k řízení GUI jednotlivých modulů.

2.1.1 Problém

Architektura aplikace z hlediska návrhu neobsahuje žádné závažné nedostatky. Hlavní problém však tkví ve vlastní organizaci kódu projektu. Moduly reprezentující základní funkcionální bloky, jako například *CashBobPokladna* nebo *CashBobStorage*, obsahují pouze klientskou část. Business logika je rozdělena mezi *CashBobRestLib* a *CashBobServer*. Hlavním problémem je spíše než nesprávná architektura, špatná organizace kódu a opomenutí doporučených principů objektově orientovaného vývoje.

2.1.2 Řešení OSGi

Mým primárním cílem bylo upravit a převést klientskou část projektu na OSGi komponentový model, včetně lepší organizace kódu. Klientská část byla zvolena proto, že má velmi minimální závislost na externích knihovnách, což snižuje riziko možných chyb z hlediska nekompatibility při přechodu na OSGi model.

Kapitola 3

Iterace I - Minimální nasazení OSGi

3.1 Úvod

Stávající kód klienta je modulární především z hlediska organizace kódu.

3.1.1 Cíl

Primárním cílem je vytvořit proof-of-concept OSGi verzi projektu bez výrazných zásahů do kódu. Výstupem iterace bude klientská aplikace běžící na OSGi frameworku Equinox.

3.2 Iterace

Práci jsem zahájil stažením kompletního stromu projektu z repozitáře.

Projekt je primárně vyvíjen ve vývojovém prostředí NetBeans, avšak jeho zprovoznění v prostředí Eclipse je jednoduché. Postačilo importovat Maven projekty sloužící k sestavení klientské a serverové části a Eclipse se postaralo o import závisejících projektů jednotlivých modulů a stažení externích závislostí z centrálního repozitáře.

Sestavení celé aplikace bylo úspěšné a po spuštění aplikace pracovala bez problémů.

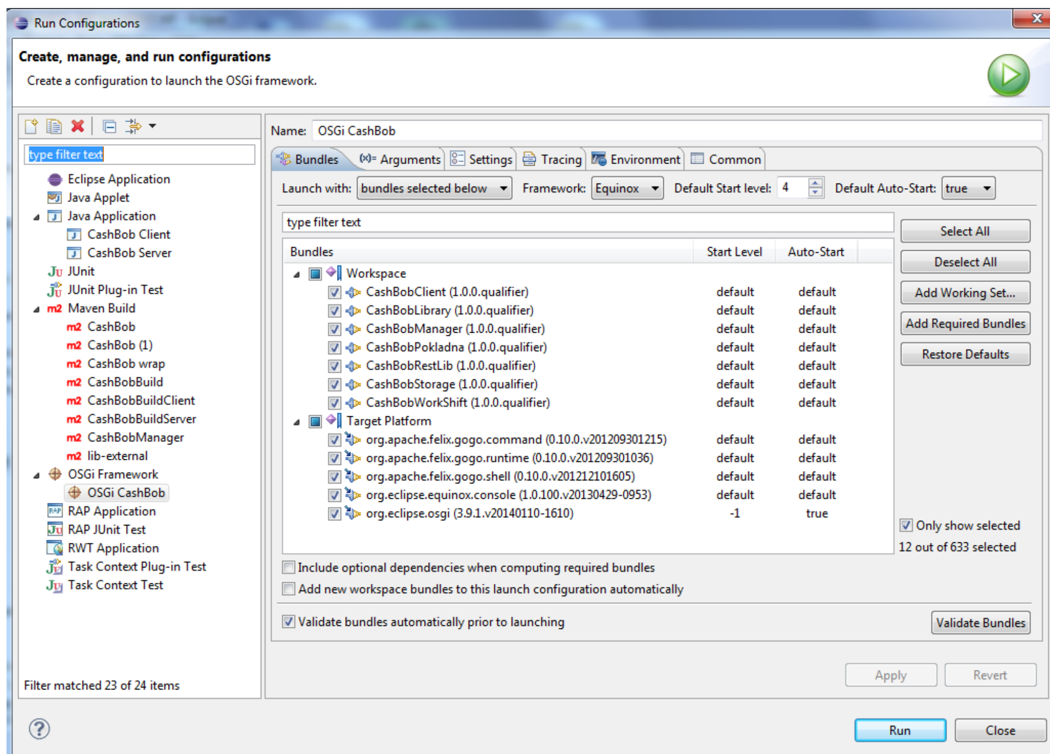
Pustil jsem se tedy do nasazení OSGi. Prvním krokem bylo převedení jednotlivých podprojektů na Plug-in projekty. To jsem provedl pravým kliknutím na projekt a volbou možnosti **Configure >Convert to Plug-in Projects...**

V dialogovém okně jsem poté vybral všechny projekty. Tím je Eclipse začalo chápat jako OSGi projekty, což má především za následek správnou interpretaci direktiv Manifestu, korektní napovídání a možnost editace konfiguračních souborů prostřednictvím dialogových formulářů. IDE dále vytvořilo ve všech projektech soubor META-INF/MANIFEST.MF s minimální sadou OSGi hlaviček, čímž se z každého projektu stal bundl:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: CashBobLibrary
Bundle-SymbolicName: CashBobLibrary
Bundle-Version: 1.0.0.qualifier
```

Protože doposud bylo sestavení a spuštění řízeno nástrojem Maven, bylo potřeba vytvořit novou spouštěcí konfiguraci pro OSGi.

Tu jsem vytvořil následujícím způsobem: Nejprve jsem otevřel dialogové okno z nabídky **Run >Run Configurations...** . V levém sloupci jsem zvolil typ konfigurace *OSGi Framework* a klepl jsem na ikonu *New launch configuration*, čímž se vytvořila nová spouštěcí konfigurace pro OSGi projekt. V pravé části okna nabízí Eclipse výběr, které bundly má použít k sestavení aplikace. Z části, která obsahuje aktuálně otevřené projekty, jsem zvolil všechny, které doposud tvořily klientskou část. V části *Target Platform* jsem zvolil pouze bundl `org.eclipse.osgi`. Na kartě argumentů jsem nastavil pracovní adresář do kořene modulu *CashBob* kvůli dostupnosti konfiguračního souboru. Při spuštění však framework selhal s chybovou hláškou: „Could not find bundle: org.eclipse.equinox.console“, ačkoliv jsem nechal automaticky zkontrolovat závislosti přes tlačítko **Validate Bundles** a volba *Validate bundles automatically prior to launching* byla zaškrtnuta. Po krátké chvíli jsem ručně doplnil zbývající závislosti dle obrázku 3.2.



Obrázek 3.1: Dialog spouštěcí konfigurace OSGi projektu

3.2.1 BundleActivator

V tomto stavu se již aplikace sestaví, ale nespustí. Na konzoli se zobrazí příkazový řádek frameworku. Po zadání příkazu `ss` vypíše framework všechny nainstalované bundly a jejich stav:

```
osgi> ss
"Framework is launched."


id State       Bundle
0 ACTIVE      org.eclipse.osgi_3.9.1.v20140110-1610
1 ACTIVE      CashBobRestLib_1.0.0.qualifier
2 ACTIVE      org.apache.felix.gogo.shell_0.10.0.v201212101605
3 ACTIVE      CashBobWorkShift_1.0.0.qualifier
4 ACTIVE      org.eclipse.equinox.console_1.0.100.v20130429-0953
5 ACTIVE      CashBobManager_1.0.0.qualifier
6 ACTIVE      org.apache.felix.gogo.command_0.10.0.v201209301215
7 ACTIVE      CashBobLibrary_1.0.0.qualifier
8 ACTIVE      org.apache.felix.gogo.runtime_0.10.0.v201209301036
9 ACTIVE      CashBobClient_1.0.0.qualifier
10 ACTIVE      CashBobStorage_1.0.0.qualifier
11 ACTIVE      CashBobPokladna_1.0.0.qualifier
osgi>
```

První řádek nás informuje o spuštění frameworku. Následuje tabulka obsahující identifikátor, který byl přidělen danému bundlu, jeho stav odpovídající diagramu 1.2 a unikátní název bundlu sestavený z jeho symbolického jména a verze. Protože jsem prostředím generované manifesty neupravoval, mají všechny bundly projektu výchozí jména a verze.

Na první pohled se tedy zdá, že je vše spuštěno, ale klient nefunguje. Důvod je následující: v případě klasické Java aplikace je potřeba definovat statickou metodu *main*, která je výchozím spouštěcím bodem. Protože ale OSGi aplikace je tvořena z velkého množství splupracujících bundlů, kdy každý z nich by měl pracovat sám o sobě, pokud možno nezávisle na ostatních, je tento problém řešen pomocí mechanizmu *BundleActivator*.

Specifikace OSGi nám umožňuje v každém bundlu definovat jednu třídu implementující rozhraní *BundleActivator*, která je po vyřešení závislostí bundlu použita k vykonání námi definovaného kódu. Existence této třídy v bundlu je indikována hlavičkou *Bundle-Activator* v manifestu.

Tohoto mechanizmu jsem tedy využil k zavolání původní metody *main* klienta. Protože se metoda *main* nachází v projektu *CashBob*, otevřel jsem `MANIFEST.MF` tohoto projektu.

Využil jsem nástroje vývojového prostředí a namísto ručních úprav souboru zvolil kartu **Overview**. Zde byla mimo jiné i položka *Activator*, která byla momentálně prázdná. Když jsem kliknul na popisek pole, otevřelo se dialogové okno pro vytvoření nové Java třídy. Jako jméno jsem zvolil *Activator* a jako package *cz.cvut.fel.restauracefel.restauracefel.bundle*[1].

[1]Konvencí je ukládat aktivační třídu mimo hierarchii vlastního kódu bundlu. Typicky je to package v kořeni hierarchie packages.

Všechny ostatní volby jsem ponechal ve výchozím stavu. Po odsouhlasení dialogu se vytvořila nová třída *Activator* a byla zobrazena. V manifestu přibyla hlavička *Bundle-Activator* s hodnotou *cz.cvut.fel.restauracefel.restauracefel.bundle.Activator*, což je absolutní cesta v rámci bundlu k aktivační třídě. Vygenerovala se prázdná třída. Eclipse v tomto okamžiku hlásilo, že nemůže nalézt import *org.osgi.framework.BundleActivator*, což je rozhraní, které musí naše aktivační třída implementovat, aby ji mohl framework použít k aktivaci bundlu.

Definice tohoto rozhraní se nachází v základním bundlu frameworku. Ten jsem sice použil pro sestavení klienta, ale ve výchozím stavu nebyly balíčky, které tento bundl obsahuje, dostupné.

Aby vývojové prostředí a následně framework dodaly potřebné balíčky pro kód, musel jsem deklarovat závislost bundlu *CashBob* na balíčku *org.osgi.framework.BundleActivator*. To lze provést dvěma způsoby, obojí přidáním hlavičky do manifestu:

- **Import-Package** - Deklaruje závislost daného bundlu na konkrétním balíčku bez ohledu na to, který bundl jej poskytuje.

- **Require-Bundle** - Deklaruje závislost na daném bundlu, a tím na všech jeho exportovaných balíčcích.

Obecně se doporučuje volit možnost *Import-Package*, neboť se tím zbavujeme závislosti na konkrétním bundlu. V případě rozsáhlejších projektů však seznam importovaných balíčků může rapidně narůstat.

Prostředí Eclipse poskytuje nástroj na automatickou správu těchto závislostí. Na kartě **Dependencies** editoru manifestu se nachází sekce *Automated Management of Dependencies*. Zde lze definovat závislost na kokrétních bundlech bez jejich přidání do manifestu. Spuštěním odkazu *add dependencies* s volbou *Import-Package* se spustí analýza kódu daného projektu a jsou-li nalezeny v kódu importy balíčků, které poskytují tyto automaticky spravované závislosti, jsou přidány do manifestu přes hlavičku *Import-Package*.

Definoval jsem závislosti na ostatních bundlech projektu, tedy stejně jako tomu bylo u původního způsobu sestavení pomocí nástroje Maven, a přidal závislost na *org.eclipse.osgi*, vygenerovalo mi Eclipse seznam všech balíčků, které tento projekt (bundl) importuje z ostatních.

Dále jsem naimplementoval metody vyžadované rozhraním a doplnil zavolání původní metody *main*. Výsledný kód třídy vypadal následovně:

```java
package cz.cvut.fel.restauracefel.restauracefel.bundle;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

import cz.cvut.fel.restauracefel.restauracefel.main.Main;

public class Activator implements BundleActivator {

  @Override
  public void start(BundleContext context) throws Exception {
    Main.main(null);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
    // Not needed
  }

}
```

Výpis 3.1: BundleActivator projektu CashBob

3.2.2 Závislosti

Po spuštění se nyní korektně zavolala metoda main a začal se vykonávat kód klienta. Krátce po zavolání metody main, která obsahovala pouze vytvoření instance třídy *Controller*[2] a následné volání její metody *run*, která spouští inicializaci aplikace a spuštění logiky přihlašování, selhala aplikace s chybou: „NoClassDefFoundError: cz/cvut/fel/restauracefel/library/interfaces/IModuleInterface“.

To bylo známkou stejného problému jako v případě vytváření aktivační třídy, respektive demonstrace výchozího chování bundlů. Tedy že všechny balíčky, které bundl obsahuje jsou implicitně skryty před ostatními. Doposud jednotlivé části spoléhaly na to, že třídy z ostatních modulů budou dostupné po sestavení aplikace.

Tím že jsem aplikaci rozdělil na bundly podle jednotlivých projektů modulů, jsem jednotlivé části izoloval od sebe. Nejjednodušším řešením nyní bylo u všech bundlů exportovat všechny packages, v nich obsažené.

Otevřel jsem tedy manifest každého projektu a na kartě **Runtime** v části **Exported Packages** zvolil **Add...**. Eclipse prohledalo daný projekt a nabídlo všechny dostupné balíčky. Označil jsem všechny, potvrdil a soubor uložil.

Důsledkem toho bylo přidání hlavičky *Export-Package* se seznamem exportovaných balíčků.

Druhým krokem bylo deklarovat závislosti bundlů mezi sebou. Toho jsem dosáhl stejným způsobem jako v předchozím případě pomocí automatické správy závislostí.

[2]Dle návrhového vzoru Controller, viz GRASP.

3.2.3 Externí knihovny

Dalším problémem z hlediska závislostí byly externí knihovny. Ve stávající konfiguraci byly externí závislosti spravovány nástrojem Maven a vyřešeny během sestavení aplikace. Aby daná knihovna byla použitelná jako OSGi bundl, je potřeba, aby její JAR soubor obsahoval manifest s příslušnými OSGi hlavičkami. To bohužel zatím dělá pouze malá skupina poskytovatelů těchto knihoven, a tak veškerá práce spočívá na vývojáři samotném. Problémem je tedy přidání hlaviček OSGi do manifestu knihovny. To je možné řešit třemi způsoby:

- **Využít automatizace pomocí Bundle Plugin for Maven** - Příslušné vkládání hlaviček do existujících knihoven je možno automatizovat pomocí plug-inu nástroje Maven, který vychází z nástroje BND. Výhodou je plná automatizace procesu včetně vnořených závislostí. Nevýhodou je trochu vyšší složitost konfigurace. Bohužel se mi nepodařilo tuto možnost plnohodnotně zprovoznit, a proto jsem kvůli úspoře času toto řešení odložil.
- **Extra bundl - vložené JAR** - Další možností je vytvořit nový bundl sloužící jakožto poskytovatel závislostí tak, že soubory JAR knihoven jsou obsaženy uvnitř tohoto bundlu. Výhodou je jednoduchost řešení s možnou výjimkou prvotního získání všech knihoven. Nevýhodou je, že si framework nemusí být schopen poradit s několikanásobným vnořením JAR archivů.
- **Extra budl - Rozbaleno** - Poslední možností je, stejně jako v předchozím případě, vytvořit nový bundl s tím rozdílem, že obsah externích knihoven je rozbalen do tohoto bundlu. Tuto metodu jsem zvolil pro další postup.

Vytvořil jsem tedy nový projekt pomocí nabídky **File >New >Project...** a jako typ projektu jsem zvolil *Plug-in project from Existing JAR Archives*. Vybral jsem JAR soubory z distribučního sestavení aplikace a vytvořil nový projekt s názvem *CashBobLibExternal*. Závislosti na tomto „externě knihovním bundlu“ jsem doplnil do správy závislostí projektu *CashBob*. Dále jsem přidal *CashBobLibExternal* do spouštěcí konfigurace.

Toto se projevilo jako nešťastné řešení, neboť některé moduly závisely na externích knihovnách tranzitivně skrze *CashBobLibrary*, respektive *CashBobRestLib*. Když jsem tedy všem modulům přidal závislost na tomto nově vytvořeném knihovním bundlu, framework nedokázal správně provázat závislosti a spuštění skončilo chybou. Řešením bylo vytvořit pro každý modul samostatný bundl obsahující externí závislosti specifické pro daný modul.

Dalším problémem bylo načítání grafických souborů (resources) z ostatních bundlů. Od toho bylo prozatím opuštěno a odloženo na další iteraci. Závislosti na grafických prvcích z ostatních bundlů byly odstraněny.

3.3 Závěr

Výstupem iterace je funkční prototyp klientské aplikace CashBob. Přes prvotní problémy se podařilo provázat bundly reprezentující jednotlivé moduly aplikace včetně závislostí na externích knihovnách. Jediným nedostatkem je absence některých grafických prvků, které však na funkčnost nemají zásadní vliv.

Kapitola 4

Iterace II - Izolace modulu

4.1 Úvod

Kód klienta je nyní spustitelný v OSGi frameworku, ale je stále velmi úzce provázaný. Je potřeba se na projekt dívat jako na platformu umožňující snadné rozšíření o novou funkcionalitu.

4.1.1 Cíl

Oddělit modul CashBobPokladna a odstínit ho od hlavního controlleru aplikace.

4.2 Iterace

Modul pokladny byl svázán s hlavní aplikací prostřednictvím jediné třídy `PokladnaController`. Ta byla odkazována ze třídy `Controller`, která je výchozím bodem klientské aplikace, a třídou `ViewController`, která obsahuje obsluhu GUI klienta.

Přestože controllery modulů všechny implementují společné rozhraní `IModuleInterface`, je k nim přistupováno přímo. To je hrubým porušením *Dependency inversion* principu.

```
package cz.cvut.fel.restauracefel.restauracefel.Controller;

public class Controller implements IModuleInterface {
    ...
    public void logout() {
        if(PokladnaController.getInstance().isActive())PokladnaController.
            getInstance().kill();
    }
    ...
}
```

Výpis 4.1: Původní třída Controller

```java
package cz.cvut.fel.restauracefel.restauracefel.Controller;

public final class ViewController {
  ...
    public void showPokladna() {
        getController().showModule(PokladnaController.getInstance());
    }
  ...
    public void showUserGate(String username) {
        mainFrame.setToolBar(new MainMenu());
        mainFrame.setTitle(MAIN_TITLE_USER + " " + username);
        mainFrame.createToolBar();
    }
}
```

Výpis 4.2: Úryvek třídy ViewController

Dalším prvkem, kterým byl modul nepřímo spojen s aplikací (pominu-li grafické ikony a lokalizační zdroje umístěné v modulu CashBobLibrary), bylo tlačítko v hlavní nabídce umožňující jeho spuštění.

Vytvoření vlastního tlačítka se nacházelo ve třídě `MainMenu`, která reprezentuje panel hlavní nabídky, a ve třídě `MainFrame` představující úvodní „obrazovku“ aplikace, kde mu byla přidělena funkcionalita (spuštění modulu pokladny) přes `ActionListener`.

```java
package cz.cvut.fel.restauracefel.restauracefel.gui;

public class MainMenu extends javax.swing.JPanel {
  ...
    public JButton modulePokladnaButton = new MainMenuButton(new ImageIcon("
        images/n_bills.png"), "Pokladna");
  ...
    public MainMenu() {
    ...
        topPanel.add(modulePokladnaButton);
    ...
    }
}
```

Výpis 4.3: Původní třída MainMenu

```
package cz.cvut.fel.restauracefel.restauracefel.gui;

public class MainFrame extends JFrame implements InterfaceGUIObject{
  ...
    public void createToolBar() {
    ...
        toolBar.modulePokladnaButton.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                view.showPokladna();
            }
        });
    ...
}
```

Výpis 4.4: Původní třída MainFrame

4.2.1 Úpravy

Tyto třídy jsem upravil následujícím způsobem:

```
package cz.cvut.fel.restauracefel.restauracefel.Controller;

public class Controller implements IModuleInterface {

  private Map<String, IModuleInterface> modules = new HashMap<String,
      IModuleInterface>();

  public void registerModule(String moduleName, IModuleInterface module) {
    if (modules.containsKey(moduleName)) {
      throw new IllegalStateException("Modul "+moduleName+" je jiz registrovan
          ");
    }
    modules.put(moduleName, module);
  }

  public void unregisterModule(String moduleName) {
    modules.remove(moduleName);
  }

  public Map<String, IModuleInterface> getModules() {
    return modules;
  }

    private Controller() {
    ...
    // Docasna umela inicializace
    modules.put(PokladnaController.class.getName(),
  PokladnaController.getInstance());
  }

  public void logout() {
    for (IModuleInterface module : modules.values()) {
```

```
31        if (module.isActive()) {
32          module.kill();
33        }
34      }
35      ...
36    }
37
38    public void showModule(String moduleName) {
39      if (!modules.containsKey(moduleName)) {
40        throw new IllegalStateException("Modul "+moduleName+" neni registrovan."
          );
41      }
42      showModule(modules.get(moduleName));
43    }
44
45    public void showModule(IModuleInterface module){
46      if (!module.isActive()) {
47        module.run(restClient.getCurrLog(),restClient, rightManager);
48      } else {
49        module.takeFocus();
50      }
51    }
52  }
```

Výpis 4.5: Upravená třída Controller

I přesto že nesmí být třída `Controller` svázána s konkrétními implementacemi modulů, je nutné aby si na ně spravovala reference. To jsem zajistil vytvořením registru modulů realizovaným pomocí standartní kolekce `Map` a vytvořením obslužných metod `registerModule`, `unregisterModule` a `getModules`[1].

Metodu `logout` jsem upravil, aby zpracovala všechny moduly v registru. Dále jsem přetížil metodu `showModule`, aby byla schopna vyvolat modul pouze pomocí `String` hodnoty jeho názvu z registru.

Pro vyhledávání modulů jsem použil úplný název třídy, jak je možno vidět v konstruktoru, kde jsem ručně přidal modul pokladny do registru. Toto volání se tak stalo jediným bodem propojení modulu pokladny s hlavní řídicí částí aplikace.

```
1  package cz.cvut.fel.restauracefel.restauracefel.Controller;
2
3  public final class ViewController {
4    ...
5      public void showModule(String moduleName) {
6      getController().showModule(moduleName);
7    }
8    ...
9    public void showUserGate(String username) {
10       MainMenu mainMenu = new MainMenu();
11       for (String moduleName : getController().getModules().keySet()) {
12       mainMenu.createModuleButton(moduleName);
13     }
14
```

[1] Pro jednoduchost vědomě porušuji u této metody princip zapouzdření a zpřístupňuji vnitřní implementaci registru.

```
15            mainFrame.setToolBar(mainMenu);
16            mainFrame.setTitle(MAIN_TITLE_USER + " " + username);
17            mainFrame.createToolBar();
18        }
19 }
```

Výpis 4.6: Upravená třída ViewController

Do třídy `ViewController` jsem doplnil univerzální metodu `showModule` pracující opět se `String` reprezentací jména modulu a doplnil jsem vytváření hlavní nabídky v metodě `showUserGate` o vytvoření příslušných spouštěcích tlačítek modulů z registru.

```
1  package cz.cvut.fel.restauracefel.restauracefel.gui;
2
3  public class MainMenu extends javax.swing.JPanel {
4    ...
5      private Map<String, JButton> moduleButtons = new HashMap<String, JButton
          >();
6      ...
7      public void createModuleButton(String moduleName) {
8      if (moduleButtons.containsKey(moduleName)) {
9        throw new IllegalStateException("Tlacitko pro modul "+moduleName+" bylo
            jiz vytvoreno.");
10     }
11
12     MainMenuButton button = new MainMenuButton(new ImageIcon("images/n_bills.
           png"), moduleName);
13     button.setEnabled(true);
14     moduleButtons.put(moduleName, button);
15
16     topPanel.add(button);
17     System.out.println("Created [button] for "+moduleName);
18   }
19     ...
20     public Map<String, JButton> getModuleButtons() {
21     return moduleButtons;
22   }
23   ...
24 }
```

Výpis 4.7: Upravená třída MainMenu

Ve třídě `MainMenu` jsem odstranil inicializaci tlačítka z konstruktoru. A vytvořil jsem metodu `createModuleButton` vytvářející nové instance spouštěcích tlačítek pro dynamicky registrované moduly. Tato metoda je volána v předchozím výpisu metody `showUserGate` třídy `ViewController`.

```
1  package cz.cvut.fel.restauracefel.restauracefel.gui;
2  ...
3  public class MainFrame extends JFrame implements InterfaceGUIObject{
4    ...
5    public void createToolBar() {
6        for (Map.Entry<String, JButton> entry : toolBar.getModuleButtons().
             entrySet()) {
7        entry.getValue().addActionListener(new ActionListener() {
8
```

```
 9          private String moduleName;
10
11          public ActionListener init(String moduleName) {
12            this.moduleName = moduleName;
13            return this;
14          }
15
16          @Override
17          public void actionPerformed(ActionEvent arg0) {
18            view.showModule(moduleName);
19          }
20        }.init(entry.getKey()));
21        System.out.println("Setting action listener for "+entry.getKey());
22      }
23        ...
24    }
25  }
```

Výpis 4.8: Upravená třída MainFrame

Namapování tlačítek na konkrétní akce, to jest spuštění daného modulu, v metodě `createToolBar` třídy `MainFrame` jsem provedl opět s využitím registru modulů.

Pro názornost jsem se nezabýval odstíněním závislostí na grafických zdrojích a lokalizaci. Ve výsledném řešení by tyto informace měl poskytovat sám modul, například doplněním rozhraní `IModuleInterface` o metody `getMenuIconResource` a `getMenuButtonLabel`. Nikoli jako doposud, kdy jsou tyto zdroje uloženy v centrálním modulu klienta.

4.3 Závěr

Výstupem této etapy je odstínění modulu *CashBobPokladna* od hlavní klientské části. Úpravy byly provedeny takovým způsobem, že ostatní moduly je nyní možné odstínit s již minimální námahou.

Kapitola 5

Iterace III - Dynamická komponenta

5.1 Úvod

Modul pokladna není už sice tak úzce vázán k základu klientské části použitím vzoru Dependency inversion, ale stále není úplně nezávislý. Jeho načtení do klienta je řešeno ručně v konstruktoru.

5.1.1 Cíl

Cílem pro tuto iteraci je vytvořit komponentu reprezentující modul pokladny a pomocí mechanizmu *Declarative services* (DS) zajistit jeho injekci do hlavní třídy `Controller` klienta.

5.2 Iterace

Modul *Pokladna* rozšiřuje funkcionalitu klientské aplikace. Toto rozšíření můžeme považovat za službu, která má pevně definované rozhraní (`IModuleInterface`). Naproti tomu modul *CashBob* je konzumentem služby tohoto typu. S tímto konceptem pracuje mechanizmus *Declarative Services*, který nám umožňuje definovat bundly jakožto komponenty poskytující nebo vyžadující služby. Pro definice služeb se používají standardní Java rozhraní.

Nejprve jsem začal upravovat modul *CashBobPokladna*. Ten poskytuje navenek službu, jejíž výchozí bod splňující výše zmíněné rozhraní, je ve třídě `PokladnaController` v balíčku `cz.cvut.fel.restauracefel.pokladna.controller`. Ten jako jediný bylo nutné vyexportovat z bundlu, takže jsem všechny ostatní záznamy v hlavičce `Export-Package` odstranil z manifestu. Dále bylo potřeba vytvořit z modulu pokladny komponentu ve smyslu *DS*. K tomu jsem použil vývojové prostředí: z kontextové nabídky prostředí jsem vybral **New >Component Definition**. Pomocí nástrojů prostředí jsem vytvořil definici OSGi komponenty, která sestává z následujícího XML souboru:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="CashBobPokladna">
    <implementation class="cz.cvut.fel.restauracefel.pokladna.controller.PokladnaController"/>
    <service>
        <provide interface="cz.cvut.fel.restauracefel.library.interfaces.IModuleInterface"/>
    </service>
</scr:component>
```

Výpis 5.1: Deklarace OSGi DS komponenty modulu CashBobPokladna

Komponentu jsem nazval *CashBobPokladna*. Z její definice lze vyčíst, že tato komponenta poskytuje službu definovanou rozhraním `IModuleInterface` a třída implementující tuto službu je `PokladnaController`. Eclipse tuto definici automaticky nalinkovalo do manifestu pomocí hlavičky `Service-Component`.

Dále bylo potřeba upravit klientský modul *CashBob*, aby mohl tyto komponenty registrovat a využívat jejich služeb. Původně jsem chtěl jako komponentu použít přímo třídu `Controller`. Ta je ale vytvořena dle návrhového vzoru *Singleton* a s tím si mechanizmus *DS* nebyl schopen poradit. Proto jsem ji obalil třídou `CashBobClientComponent`:

```java
package cz.cvut.fel.restauracefel.restauracefel.component;
...
public class CashBobClientComponent {
    Controller controller;

  public void startup() {
    controller = Controller.getInstance();
    controller.run();
  }

  public void registerModule(IModuleInterface module) {
    Controller.getInstance().registerModule(module);
  }

  public void unregisterModule(IModuleInterface module) {
    Controller.getInstance().unregisterModule(module);
  }
}
```

Výpis 5.2: Třída CashBobClientComponent

DS má vlastní způsob aktivace kódu, takže jsem z projektu vymazal třídu `Activator` a spouštěcí metodu `startup` jsem umístil do třídy komponenty. Dále jsem provedl několik drobných úprav registru modulů a přidal metody pro výmaz modulu z registru a s tím spojené odstranění spouštěcího tlačítka v hlavní nabídce. Nakonec jsem vytvořil definici komponenty:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" activate="startup" name="CashBobClient">
    <implementation class="cz.cvut.fel.restauracefel.restauracefel.component.CashBobClientComponent"/>
```

```
4   <reference bind="registerModule" cardinality="0..n" interface="cz.cvut.fel.
        restauracefel.library.interfaces.IModuleInterface" name="
        IModuleInterface" policy="dynamic" unbind="unregisterModule"/>
5 </scr:component>
```

Výpis 5.3: Definice OSGi komponenty CashBobClient

Komponentu jsem pojmenoval *CashBobClient*. Její implementační třídu jsem nastavil na obalující třídu `CashBobClientComponent` a navíc nastavil metodu `startup` jako spouštěcí. Každá komponenta může využívat libovolné množství služeb poskytovaných ostatními komponentami.

Tag `<reference>` umožňuje deklarovat závislost komponenty na ostatních službách. Atribut `interface` určuje rozhraní služby, kterou komponenta požaduje, atributy `bind` a `unbind` vyjmenovávají metody komponenty, které budou použity k injektování služby do komponenty, respektive její odpojení. Tímto jsem dokončil převod bundlů na komponenty.

Posledním krokem bylo přidání bundlů `org.eclipse.equinox.ds` a `org.eclipse.equinox.utils`, které poskytují implementaci mechanizmu DS do spouštěcí konfigurace projektu.

Po spuštění se aplikace chovala jako obvykle a na první pohled se nic nezměnilo. Po přihlášení se zobrazila hlavní nabídka. Zásadní přínos ale nastal v tom, že jsem si bundl nyní mohl za běhu aplikace beztrestně zastavit. Nejprve jsem si zjistil id, které framework přidělil bundlu pokladny:

```
osgi>ss

"Framework is launched."

id State        Bundle
...
3 ACTIVE       CashBobPokladna_1.0.0.qualifier
...
```

Rozhodl jsem se zastavit bundl pomocí příkazu `stop 3`. Framework na to zareagoval odpojením modulu od komponenty *CashBobClientComponent*, což mělo za následek vymazání modulu z registru a odstranění spouštěcího tlačítka z menu. Bundl byl nyní ve stavu `RESOLVED`:

```
osgi> ss
...
3 RESOLVED     CashBobPokladna_1.0.0.qualifier
...
```

Zadáním příkazu `start 3` došlo opět ke spuštění bundlu, injekci služby mezi komponentami a zobrazení spouštěcího tlačítka. Při spuštění modulu samotného dojde k chybě způsobené špatným načtením správce uživatelských rolí a oprávnění z modulu *CashBobLibrary*. To je způsobeno závislostí pokladního modulu na knihovním, kterou jsem ale nepřevedl na mechanizmus dynamických komponent.

5.3 Závěr

Výsledkem iterace je prototyp klientské aplikace, která korektně reaguje na přidání a odebrání, resp. spuštění a zastavení modulu.

Kapitola 6

Závěr

Modularizace a znovupoužitelnost jsou neustále skloňované koncepty ve všech odvětvích. Pokud se vývoje softwarových aplikací v jazyce Java týká, zdá se, že technologie OSGi je tím správným směrem, kterým se vydat. Nabízí poměrně jednoduchý model a relativně intuitivní sadu nástrojů k vytvoření plně modulárního řešení softwarových aplikací. Na první pohled se OSGi zdá jako „země zaslíbená“, vždyť slogan „OSGi™ - The Dynamic Module System for Java™“ k této myšlence vybízí, ale ne vše je úplně zadarmo.

Použití této technologie od nás sice vyžaduje myslet určitým způsobem a používat jisté postupy, které však jsou již známé. Správné nasazení technologie OSGi stojí na správném objektovém návrhu aplikace. Jde tedy o použití praxí ověřených postupů a pouček pro návrh objektových aplikací. Nejedná se o nic jiného než o zásady zmíněné v úvodní kapitole, především principy *High Cohesion* a *Low Coupling*.

Za největší výzvu při nasazení OSGi a obecně při softwarovém návrhu bych označil „lenost“ nebo ještě lépe „absenci snahy o čistý design“, která nás ovlivňuje ve dvou směrech:

- **Vnitřní vliv** - Jedna část problémů vychází přímo od nás, samotných vývojářů aplikace. Jsme-li konfrontováni s nečekaným problémem, máme tendenci hledat řešení cestou nejmenšího odporu. Často je řešením dočasné obcházení problému pouze v místě výskytu, které se však časem stane trvalým. V lepším případě sáhneme k osvědčenému návrhovému vzoru, který nám ale díky nesystémovému použití bez dostatečného nadhledu nabídne pouze falešný pocit dobře odvedené práce a problém pouze rozmělní.

- **Vnější vliv** - Druhým zdrojem problémů bývají často nástroje z externích zdrojů, především knihovny a frameworky. V případě dlouhodobě vyvíjených knihovních projektů se nemusíme příliš obávat nesystémového designu knihovny samotné. Problém tkví v jejich používání. Ačkoliv nám usnadňují řešení některých úloh, často se stává, že okouzleni snadným řešením problému se necháme zlákat a používáme metody konkrétní knihovny kdekoliv uznáme za vhodné, aniž bychom si uvědomili, že tím svazujeme náš kód s konkrétní knihovnou namísto toho, abychom mezi náš kód a knihovnu zanesli jistou míru abstrakce a zapouzdření.

Takovou nesystematičnost je možné zanést do softwarového projektu v kterékoliv etapě vývoje. Konkrétní fáze vývoje pak ovlivňuje rozsah výsledného problému. Podcenění řešení ve fázi analýzy a návrhu často mívá nejrozsáhlejší důsledky a ovlivňuje projekt po celý zbytek vývoje. Tato architektonická chyba se ale může objevit i při řešení jakéhokoli, na první pohled banálního, problému a bez včasného zásahu může postupně „zarůstat“. Jistá nedokonalost architektury nemusí být nutně na závadu. Je-li kód funkční a je-li možné ho upravovat a rozšiřovat s relativně malou námahou, necítíme potřebu se kriticky zamýšlet nad jeho architekturou. Problém nastává až v okamžiku, kdy jsme nuceni rozdělit projekt na samostatné celky.

6.1 Restaurační systém CashBob

Počátek vývoje projektu CashBob se datuje už od 22. září 2011[1]. V průběhu následujících let až do současnosti se na něm podílelo více jak 20 osob[2] v rámci předmětu *Řízení SW projektů* nebo vlastního projektu během studia. Diplomová práce Jana Vrtišky [7], jednoho z předchozích členů vývojového týmu, z roku 2013 ve své analytické části poukazuje i po dvou letech vývoje na jistou nestabilitu určitých modulů projektu, nízkou míru abstrakce a vysokou provázanost kódu. Tento vývoj bychom mohli přisuzovat tomu, že po stanovení základní kostry projektu se nově příchozí členové zaměřili pouze na určité úlohy, jejichž řešení bylo obvykle omezeno na jeden konkrétní modul, a nepříliš často se objevovaly změny, které by ovlivňovaly architekturu celého projektu. Protože však byla funkčnost projektu zachována, nebylo nutné tyto nedokonalosti dále adresovat.

Problémy plynoucí ze zanedbání dohledu nad celkovou architekturou se projevily již v první iteraci. Zásadním problémem bylo, že veškeré sdílené zdroje (grafické ikony a loga) byly uloženy v knihovním modulu, na kterém byly závislé ostatní moduly aplikace. To by nepředstavovalo zásadní potíž, bohužel ale ostatní části kódu spoléhaly na to, že po sestavení bude možné přistupovat k těmto zdrojům přímo díky sjednocení veškerého kódu do jednoho JAR souboru. Dále měl tento modul deklarované závislosti na externích knihovnách, které díky společnému prostředí *classpath* nepřímo poskytoval ostatním závislým modulům. Posledním komplikací bylo svázání s knihovnou poskytující obohacení grafického vzhledu aplikace. Výstupem druhé iterace byla větší nezávislost pokladního modulu na jádru klientské části. Na konci třetí iterace byl již funkční prototyp aplikace reagující na přidání a odebrání pokladního modulu ve formě služby.

6.2 Výsledek

Ve výsledku se podařilo vytvořit prototyp klientské části aplikace CashBob, s izolovaným pokladním modulem na základě specifikace OSGi. Práce prokázala důležitost používání a důsledného dodržování ověřených principů při vývoji softwarových aplikací. Další vývoj je možné směrovat k rozdělení jednotlivých modulů a vlastního jádra aplikace a vytvoření instalačního/aktualizačního nástroje.

---

[1] Datum prvního commitu do repozitáře projektu.

[2] Vybrány osoby s 10 a více commity v repozitáři projektu.

Literatura

[1] Charles Humble. *IBM, BEA and JBoss adopting OSGi* [online]. February 2008. [cit. 3. 5. 2014]. Available at: ¡`http://www.infoq.com/news/2008/02/osgi_jee`¿.

[2] Jeff McAffer, Paul VanderLei, Simon Archer. *OSGi and Equinox. Creating Highly Modular Java^{TM}Systems.* Pearson Education, Inc., 1st edition, 2010.

[3] The Apache Software Foundation. *What is Maven?* [online]. [cit. 8. 5. 2014]. Available at: ¡`https://maven.apache.org/what-is-maven.html`¿.

[4] The Open Services Gateway Initiative. *OSGi Service Gateway Specification. Release 1.0* [online]. May 2000. [cit. 20. 4. 2014]. Available at: ¡`http://www.osgi.org/Download/`¿.

[5] The OSGi Alliance. *Guidelines for designing Java API for use in the OSGi environment* [online]. [cit. 3. 5. 2014]. Available at: ¡`http://www.osgi.org/Design/Guidelines`¿.

[6] The OSGi Alliance. *OSGi Service Platform Core Specification. Release 4, Version 4.2* [online]. June 2009. [cit. 21. 4. 2014]. Available at: ¡`http://www.osgi.org/Download/`¿.

[7] VRTIšKA, J. *CashBob - modul pro skladové hospodářství. Diplomová práce.* Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2013.

Příloha A

Seznam zkratek

API Application Programming Interface

GUI Graphics User Interface

IDE Integrated Development Environment

POJO Plain Old Java Object

OSGi Open Service Gateway initiative

KISS Keep It Simple Stupid

DRY Don’t Repeat Yourself

GRASP General Responsibility Assignment Software Patterns

DS Declarative Services

Příloha B

Instalační manuál

1. Zkopírujte obsah CD na disk
2. Ve složce Eclipse se nachází vývojové prostředí. Spusťte soubor eclipse.exe
3. V Eclipse zvolte položku File >Import…
4. Vyberte složku CashBob-OSGi
5. Naimportujte všechny projekty
6. Ve složce CashBobServer-1.0 se nachází předkompilovaná serverová část applikace. Spusťte ji.
7. V Eclipse opět zvolte import `CashBob OSGi Client.launch`, který je umístěn v projektu CashBob
8. Spusťte aplikaci pomocí spouštěcí konfigurace přes Run >Run configurations…

Příloha C

Obsah CD

```
+- abstract.txt  Abstrakt práce
+- BP-Text Text práce včetně zdrojů
+- CashBob-OSGi Implementační část práce
+- CashBobServer-1.0 Předkompilovaná serverová část aplikace
+- Eclipse Vývojové prostředí Eclipse
+- readme.txt Obsah CD
```