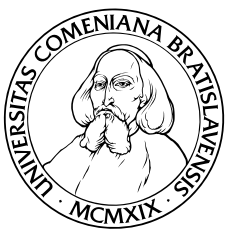


Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Modulárne architektúry aplikácií a ich využitie

(Bakalárska práca)

Michal Vician

Vedúci: RNDr. Andrej Lúčny, PhD.

Bratislava, 2009

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

.....

Ďakujem týmto všetkým, ktorí ma podporovali a pomáhali mi vytvoriť túto prácu, konkrétne RNDr. Andrejovi Lúčnemu, PhD. za jeho odborné vedenie, pripomienky a za čas a trpezlivosť, ktorú mi pri vypracovávaní tejto práce venoval.

Michal Vician

Obsah

Kapitola 1

Úvod

1.1 Motivácia a cieľ práce

V dnešnej dobe sa pri vývoji aplikácií kladie stále väčší dôraz na architektúru týchto aplikácií. Dôkazom je počet frameworkov, ktoré sú dnes dostupné, a ktoré zasahujú do mnohých aspektov vývoja softvéru prostredníctvom API, ktoré poskytujú. Väčšina týchto moderných frameworkov pozná pojem komponent, pretože si to vyžaduje spôsob, akým sú dnes z veľkej časti aplikácie vyvíjané. Vývojári totiž niekedy strávia väčšinu času spájaním už odladených a dobre fungujúcich komponentov do jedného fungujúceho celku. Je to ľahko pochopiteľné. Veď prečo by mali vývojári zakaždým znovu objavovať koleso? Háčik je však v tom, že správa komponentov, ich distribúcia do systému, ako aj zabezpečenie ich vzájomnej komunikácie, nie je triviálna záležitosť. V prípade modulárnych architektúr sa navyše na programátorov kladú nové požiadavky, ktoré súvisia so životným cyklom týchto komponentov.

Cieľom tejto práce je preskúmať aké výhody a nevýhody má použitie dynamickej modulárnej softvérovej architektúry pri vývoji aplikácií v porovnaní s architekúrami, ktoré definujú vzťahy medzi komponentami iba staticky. Zároveň sa budem zaoberať niektorými návrhovými vzormi (angl. *design pattern*) a popíšem, ktoré sú pre takúto architektúru vhodné a ktoré nie. Cieľom je takisto ukázať, ako možno vyvinúť aplikáciu s touto architektúrou v prostredí jazyka Java. Všetky tieto poznatky budem demonštrovať na malej ukážkovej aplikácií AudioTT, ktorá využíva modulárny framework *OSGi*.

Kapitola 2

Modulárne architektúry

2.1 Definícia a základné vlastnosti

Modulárna architektúra je konkrétny prípad *softvérovej architektúry*. Existuje viacero definícií pojmu *softvérová architektúra*. Do dnešného dňa však nie je žiadna z nich akceptovaná ako presná definícia tohoto pojmu [4]. Z tohoto dôvodu zadefinujem tieto pojmy tak, ako vystupujú v kontexte tejto bakalárskej práce.

Architektúra softvérového systému

Je to štruktúra (množina štruktúr), ktorá zostavuje komponenty systému do jedného celku.

Architektúrou softvéru ako konceptom sa ako prvý začal zaoberať Edsger Dijkstra (1968) a neskôr David Parnas (1970) [4]. Obaja títo vedci začali zdôrazňovať, že záleží na štruktúre softvéru a neoplatí sa programovať tak, aby program iba plnil účel (vracal korektné výstupy ap).

Architektúru softvéru si netreba mýliť so sofvérovým dizajnom. *Softvérový dizajn* je proces tvorby systému, pri ktorom sa rozhodujeme, ktoré prostriedky a akým spôsobom budú použité pri samotnom vývoji.

Modulárna architektúra softvérového systému

Je to štruktúra, ktorá zostavuje komponenty systému do jedného celku, avšak väzby medzi týmito komponentami sú voľné (loosely-coupled) a flexibilné. Pod pojmom voľné väzby sa myslí, že sú špecifikované rozhraním. Pod pojmom flexibilné zase, že je možné jednotlivé komponenty rôzne kombinovať. Komponenty modulárnej architektúry sa nazývajú *moduly*.

Výhodou systému s takouto architektúrou je omnoho jednoduchšia údržba. V tejto bakalárskej práci však túto definíciu ešte rozšírim o jednu novú podmienku - dynamickosť. Dynamickosť znamená, že po pridaní alebo odstránení ľubovoľného modulu v ľubovoľnom čase systém nemôže spadnúť (Runtime error) ale iba zmeniť svoju funkcionalitu. Takáto architektúra nám potom umožňuje vyvinúť systém, ktorý je možné aktualizovať počas jeho behu (nie je potrebný reštart celého systému).

Systémy s takouto architektúrou sú väčšinou postavené na podpornom frameworku, ktorý zabezpečuje komunikáciu medzi modulmi a popisuje ich životný cyklus. Životný cyklus modulu musí byť frameworkom zadefinovaný kvôli požiadavke pridávať a odstraňovať moduly v ľubovoľnom čase.

V tejto bakalárskej práci pracujem s modulárnou architektúrou OSGi, ktorá definuje framework, na ktorom je možné vyvinúť aplikáciu s týmito vlastnosťami:

- Aplikácia je zložená z modulov.
- Väzby medzi modulmi sú voľné.
- Do systému je možné moduly pridať alebo z neho odobrať v ľubovoľnom čase (dynamickosť).

2.2 Výhody a nevýhody

Flexibilita

Modulárny softvér je dobre konfigurovateľný a flexibilný. Znamená to, že rôznou konfiguráciou modulov systému je možné poskytnúť daný systém v rôznych variáciách (s rôznou funkcionalitou). V praxi je dnes táto vlastnosť žiadaná veľmi často. Napr. rôzne variácie toho istého softvéru sú predávané pod rôznymi licenciami.

Framework

Pri vývoji modulárneho systému si asi nebude každý písať podporný framework sám, ale použije už existujúci. Použiť existujúci framework je nesporne výhoda už iba v tom, koľko času by trvalo vyvinúť nový. Treba si však uvedomiť, že vývoj softvéru je na zvolenom frameworku úplne závislý. Ak sa počas vývoja zistí, že framework neposkytuje požadovanú vlastnosť, alebo že je v istých prípadoch zle zaimplementovaný ap., je to veľký problém.

Zložitosť

Či bude výsledný systém zložitejší ak bude modulárny je otázne kvôli uhlu pohľadu, akým sa na zložitosť pozrieme. Každopádne pri dobrom návrhu modulov a vzťahov medzi nimi prináša so sebou modularita jednoduchšiu údržbu a spomínanú flexibilitu.

Jedným aspektom je teda zložitosť týkajúca sa samotného návrhu modulov - ktorý modul bude mať v systéme akú úlohu a ako budú tieto moduly preväzbené. Tento návrh už sám o sebe nie je jednoduchou úlohou.

Ďalším aspektom je zložitosť týkajúca sa vývoja jednotlivých modulov. Predstavme si moduly ako funkčné jednotky. V monolitickom systému by tieto jednotky boli prepojené statickými väzbami. Tie by sa vytvorili pri kompilácií alebo až počas štartu systému aplikácie. Napríklad *Spring* je framework, ktorý umožňuje závislosti medzi komponentami

definovať v deklaratívne v XML súbore. Väzby medzi jednotlivými komponentami potom *Spring* vytvorí podľa konfiguračných súborov až pri spustení aplikácie ([9]).

V modulárnom systéme však každá funkčná jednotka musí rátať s tým, že jej ľubovolná väzba/závislosť môže v ľubovoľnom čase zmiznúť ale sa aj objaviť. Každý modul teda musí okrem svojej základnej funkcionality obsahovať ešte aj logiku, ktorá reaguje na udalosť straty ľubovolnej závislosti modulu ako aj na udalosť poskytnutia takejto závislosti.

Navyše, spustenie alebo zastavenie modulu môže byť voči funkcionalite poskytovanej modulom asynchrónna operácia. Treba teda myslieť aj na to, že modul môže stratiť svoju závislosť práve v čase, keď vykonáva operáciu, na ktorej realizáciu danú závislosť potrebuje. Alebo môže modul dostať pokyn na svoje zastavenie práve v čase, keď vykonáva dajakú operáciu. Od modulu sa očakáva, že je zaimplementovaný tak, aby danú operáciu čo najskôr a bezpečne ukončil. Pri implementovaní modulov je preto nutné pripraviť sa na prácu vo viacvláknovom (multithreadovom) prostredí.

Náchylnosť na chyby

Zo zložitosti pri implementácií modulov vyplýva, že budú náchylnejšie na chyby ako keby boli moduly preväzbené staticky. Dobré frameworky však poskytujú štandardné nástroje resp. API, pomocou ktorého je možné moduly implementovať relatívne jednoducho a bezpečne.

Kapitola 3

Modularita v prostredí jazyka Java

Momentálne existujú dva korporátne podporované frameworky, ktoré umožňujú v jazyku Java vyvíjať aplikácie s dynamickou modulárnou architektúrou. Sú nimi *NetBeans platform* a *OSGi*. Tieto frameworky sa líšia len minimálne. Obidva implementújú základné črty dynamického modulárneho frameworku ako sú modul alebo servis registrov. Rozdiel je iba v ich API a terminológií. Napríklad *service registry* v OSGi (register servisov) je v princípe to isté ako *Lookup* v Netbeans platform.

V dnešnej dobe zažíva najväčší boom zrejme práve technológia OSGi, pretože ju k sebe integrujú iné frameworky ako napríklad *Spring*. Zároveň začína byť táto technológia vo veľkom intergrovaná aj do aplikačných servrov (*Websphere*, *JBoss*, *Spring DM Server*, ...). V niektorých prípadoch sú na tejto technológií servre rovno postavené (*Jetty*). Dôvodom integrovania technológie OSGi do aplikačných servrov je jej dobrá špecifikácia a možnosť vývoja aplikácií, ktoré musia bežať v režime 24/7. Pri dobrom návrhu takejto aplikácie je totiž možné vykonať jej aktualizáciu bez reštartu aplikačného servra.

3.1 NetBeans platform

NetBeans je prvé integrované vývojové prostredie (IDE) pre jazyk Java napísané v jazyku Java. Toto vývojové prostredia začali vyvíjať študenti z Českej republiky ako školský projekt už v roku 1996. Od počiatku vývoja sa snažili o to, aby bolo toto vývojové prostredie pluginovateľné. Neskôr si uvedomili, že ak sa im podarí oddeliť od IDE tú časť kódu, ktorá zabezpečuje modularitu ich vývojového prostredia, mohli by ju poskytovať ako framework pre tvorbu modulárnych aplikácií. Dnes je tento framework známy ako *NetBeans platform*.

NetBeans ako vývojové prostredie má k dispozícií vynikajúce nástroje na vývoj aplikácií postavených na NetBeans platform, ako aj nástroje na vývoj pluginov do samotného vývojového prostredia. Zároveň poskytuje v dnešnej dobe asi najlepší nástroj na tvorbu užívateľského rozhrania postaveného na knižnici *Swing*. Tento nástroj sa volá *Matisse*. Ak sa teda niekto rozhodne založiť svoju aplikáciu na NetBeans platform, nemusí sa báť nedostatku podporných nástrojov.

Nevýhodou však je, že NetBeans platform nie je štandardizovaný framework tak ako je to v prípade OSGi (viď. nasledujúca sekcia). V dôsledku toho nie je natoľko integrovaný do iných frameworkov. Otázkou potom je, do učenia ktorého frameworku sa skôr oplatí investovať energiu. Samozrejme, že to závisí ešte od viacerých faktorov a povahy projektu, ktorý sa ide vyvíjať. Integrovateľnosť s inými frameworkami však zanedbateľný faktor určite nie je.

3.2 OSGi

OSGi Service Platform je špecifikácia vyvíjaná *OSGi Alliance* [1]. Táto špecifikácia definuje modulárny framework pre platformu Java.

Základné charakteristiky OSGi frameworku

- Rozširuje pravidlá viditeľnosti jazyka Java (visibility rules - private, protected, public) o nový typ viditeľnosti, ktorý umožňuje definovať viditeľnosť celých balíčkov (Java package) medzi jednotlivými modulmi (viď. Bundle space v špecifikácií [1]).
- Menežuje závislosti medzi modulmi a ich verziovanie (versioning).
- Je dynamický - umožňuje moduly systému inštalovať, spúšťať, zastaviť, aktualizovať a o odinštalovať počas behu tohoto systému.
- Je servisne orientovaný - modul môže do frameworku hocikedy zaregistrovať servis (poskytnúť frameworku funkcionalitu), ktorý môže následne skonzumovať ľubovoľný iný modul.

Implementácie

Implementácií OSGi je pomerne veľa. V dnešnej dobe je však Alianciou OSGi certifikovaných iba týchto päť implementácií:

1. Makewave Knopflerfish Pro 2.0 (*www.makewave.com*)
2. ProSyst Software mBedded Server 6.0 (*www.prosyst.com*)
3. Eclipse Equinox 3.2 (*www.eclipse.org/equinox/*)
4. Samsung OSGi R4 Solution (*www.samsung.com*)
5. HitachiSoft SuperJ Engine Framework (*http://hitachisoft.jp/*)

Každá z nich má svoje výhody a nevýhody. Niektoré sú implementované tak, aby mohli bežať v zariadeniach s obmedzenými zdrojmi (embedded device). Príkladom je napríklad implementácia Concierge (*http://concierge.sourceforge.net/*). Concierge má veľkosť iba

približne 80kB, takže je možné ho využiť v zariadeniach akými sú mobilné telefóny, alebo roboty s malou dostupnou pamäťou.

Na účely tejto bakalárskej práce som si však vybral implementáciu Equinox. Dôvodom je, že na tejto implementácií je postavené samotné vývojové prostredie Eclipse. Equinox je preto veľmi dobre zdokumentovaným frameworkom. Prácu s ním navyše uľahčuje ešte aj to, že Eclipse poskytuje širokú škálu nástrojov určených na vývoj aplikácií na ňom založených. Bohužiaľ, ani tieto nástroje ešte nie sú dokonalé. Niektoré úkony sú pri vývoji takejto aplikácie preto komplikované aj v samotnom Eclipse.

3.2.1 Equinox

OSGi implementácia Equinox je obsiahnutá v jednom JAR súbore. Ako väčsina implementácií aj Equinox poskytuje konzolu, prostredníctvom ktorej je možné spravovať bežiacu inštanciu frameworku (inštalovať do systému nové moduly, spúšťať ich a podobne).

Zapnutie a vypnutie konzoli

Ak chceme mať k dispozícií konzolu, stačí Equinox spustiť z príkazového riadku ako bežný JAR súbor s dodatočným prepínačom `-console`.

```
miso$ java -jar org.eclipse.osgi_3.4.0.jar -console
osgi>
```

Konzola poskytuje zopár základných príkazov. Ich zoznam so stručným popisom na čo slúžia možno zobraziť zadaním príkazu `?` alebo `help`.

```
osgi> ?
---Eclipse Runtime commands---
    diag - Displays unsatisfied constraints for the specified bundle(s).
    enableBundle - enable the specified bundle(s)
    disableBundle - disable the specified bundle(s)
    disabledBundles - list disabled bundles in the system
---Controlling the OSGi framework---
    launch - start the OSGi Framework
...
```

Aplikácia sa vypína príkazom:

```
osgi> exit
```

Spustenie a zastavenie modulu

Modul v OSGi je JAR súbor s určitými vlastnosťami (viď. 3.2.2). Pred spustením modulu je potrebné ho nainštalovať. Inštalácia je proces, pri ktorom sa overí či je daný JAR súbor valídnym OSGi modulom a pri ktorom sa tento JAR súbor fyzicky skopíruje do configuračného adresára Equinoxu.

Riadenie životného cyklu modulu prostredníctvom konzoli budem demonštrovať na aplikácií AudioTT. Po spustení aplikácie môžeme v konzole zobraziť zoznam modulov, ktorými je aplikácia momentálne tvorená. Na zobrazenie tohoto zoznamu slúžia príkazy `status` a `ss` (*short status*).

```
osgi> ss

Framework is launched.

id  State      Bundle
0   ACTIVE     org.eclipse.osgi_3.4.3.R34x_v20081215-1030
1   ACTIVE     org.eclipse.equinox.util_1.0.0.v20080414
2   ACTIVE     org.audiott.ui_1.0.0
...
```

Pri prvom štarte poskytuje aplikácia iba základnú funkcionalitu. Aby sa aplikácia AudioTT dala využiť na prezeranie a editáciu napríklad metadát MP3 súborov, je potrebné nainštalovať a spustiť modul `org.vician.meta.mp3`.

```
osgi> install file:|||home/miso/projects/audiott/org.vician.meta.mp3.jar
Bundle id is 8

osgi> start 8
```

Po naištalovaní modulu sa tomuto modulu priradí identifikátor (číslo), ktorým potom možno identifikovať modul pri práci s inými príkazmi. Modul sa v tomto prípade naštaruje príkazom `start 8`. Ak bol modul úspešne naštartovaný tak sa nachádza v stave `ACTIVE`. Overiť si to možno opäť výpisom príkazu `ss`.

Počas svojho štartu modul zaregistroval do registra servisov servis (viď. 3.2.3) umožňujúci aplikácií čítať metadáta MP3 súborov. Rozšírila sa tak funkcionalita programu bez toho, aby sme ho reštartovali. Zastaviť modul možno rovnako jednoducho ako ho spustiť:

```
osgi> stop 8
```

Po zastavení modulu sa modul nachádza v stave `RESOLVED`. Tento stav značí, že modul nie je aktvívny a nie sú už viac k dispozícií ani servisy, ktoré zaregistroval pri svojom štarte. Zároveň však stav modulu `RESOLVED` značí, že všetky servisy a moduly, na ktorých je závislí sú vo frameworku k dispozícií, a preto je tento modul pripravený na svoje opätovné spustenie. Po zastavení modulu sa preto aplikácia AudioTT nachádza v počiatočnom stave - tz. že neumožňuje editovať metadáta MP3 súborov.

Veľmi dôležitým príkazom je ešte príkaz `update`, ktorý slúži na aktualizáciu modulu. Príkaz `update` v podstate daný modul preinštaluje a pokúsi sa ho uviesť do rovnakého stavu, v akom bol pred aktualizáciou. Znamená to, že ak sa aktualizuje modul, ktorý sa nachádza napríklad v stave `ACTIVE`, framework sa po jeho preinštalovaní pokúsi modul do tohoto stavu automaticky dostať. Príkaz `update` zlyhá na výnimke `FileNotFoundException` ak sa aktuaklizovaný modulu fyzicky nenáchádza na rovnakom mieste z akého bol prvotne nainštalovaný.

3.2.2 Moduly

Pre pochopenie ako OSGi definuje modul je nutné vedieť, čo je to JAR súbor a jeho manifest [5]. Modul je totiž z pohľadu OSGi taký JAR súbor, ktorý obsahuje dodatočné meta informácie (názov modulu, verzia modulu a podobne). Tieto informácie sa nachádzajú v samotnom manifeste JAR súboru. V niektorých prípadoch však formát manifestu nepostačuje a viac sa hodí napríklad formát XML. XML súbor je potom v manifeste zadefinovaný ako jeho rozšírenie (extension). Z toho je jasne vidno, že manifest je ústredným bodom konfigurácie modulu a že je z neho možné vyčítať všetky vlastnosti modulu v prostredí OSGi frameworku.

OSGi a viditeľnosť zdrojov medzi modulmi

Každý Java programátor by mal dobre poznať pojem *class path*. Je to zoznam priečinkov a JAR súborov, v ktorých sa nachádzajú Java triedy a iné zdroje, ktoré môže program dynamicky načítať počas svojho behu prostredníctvom tzv. *class loadera*. V prípade, že sa program snaží pristúpiť k triede, ktorá sa nenachádza v zozname *class path*, *class loader* vyvolá výnimku `ClassNotFoundException`.

Ako už bolo spomenuté v sekcií o základných vlastnostiach frameworku OSGi (3.2), OSGi rozširuje pravidlá viditeľnosti jazyka Java. OSGi framework bežiaci v jednej virtuálnej mašine (JVM) môže obsahovať viacero bežiacich modulov. Tieto moduly však v implicitnom stave nevedia navzájom pristupovať k svojim zdrojom. Treba si uvedomiť, že moduly sú JAR súbory. Keby tieto JAR súbory boli spustené v jednej virtuálnej mašine bez frameworku OSGi, k svojim zdrojom by mohli navzájom pistupovať. Otázkou teda je, akým spôsobom OSGi kontroluje viditeľnosť zdrojov v jednotlivých moduloch a ako možno túto viditeľnosť konfigurovať.

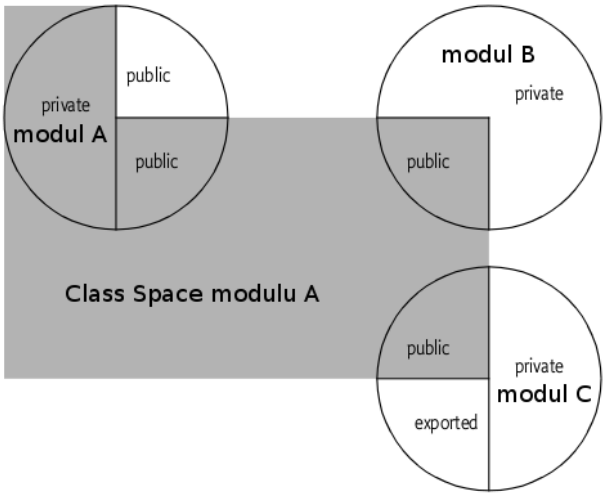
Odpoveďou na túto otázku je tzv. *class space*.

Class space a Class loader

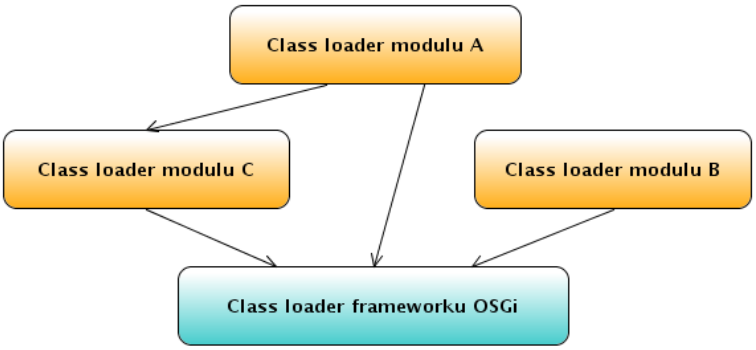
V prostredí OSGi je vytvorený jeden class loader pre každý modul. Prístup k zdrojom je teda plne pod kontrolov tohoto špeciálneho class loadera. Class space je definovaný ako množina tried a zdrojov, ktoré je schopný tento špeciálny class loader načítať. Každý modul má teda svoj vlastný *class space*.

Ako som už spomínal, implicitne moduly svoje zdroje medzi sebou nezdieľajú. Ak chce modul zdieľať dajaký zdroj, môže tak urobiť na úrovni balíčkov (Java package). Slúžia na to klauzuly v manifeste JAR súboru.

- *Export-Package: some.java.package* umožní použiť zdroje zadefinované v balíčku *some.java.package* z iného modulu
- *Import-Package: some.java.package* deklaruje použitie zdrojov exportovaných iným modulov v danom module
- *Require-Bundle: meno.modulu* vyžaduje prítomnosť modulu s daným identifikátorom



Obrázok 3.1: Ilustrácia *class space* špecifického modulu.



Obrázok 3.2: Príklad *class loader delegation modelu*. V tomto príklade modul A nevie pristúpiť k zdrojom modulu B.

Klauzúl definujúcich vlastnosti a závislosti modulov je omnoho viac. Ich zoznam možno nájsť v špecifikácií ([1]). Tie čo som vymenoval však patria k základným, čo sa týka viditeľnosti zdrojov medzi modulmi. Aby *class loadery* jednotlivých modulov vedeli pristupovať k zdrojom tak, ako sú zadeklarované v manifeste, OSGi framework medzi nimi vztvára hierarchiu, ktorá sa v nazýva *class loader delegation model*. V tomto modeli ide v princípe o tom, že ak nie je class loader daného modulu schopný načítať dajaký konkrétny zdroj, požiada o toto načítanie class loader toho modulu, ktorého zdroje importuje (import je zadefinovaný v manifeste modulu) (3.2).

Spustenie a zastavenie modulu

Na spustenie modulu cez konzolové rozhranie OSGi slúži príkaz `start` (3.2.1). Modul môže, ale aj nemusí počas štartu vykonať isté inicializačné operácie. Ak sa vyžaduje explicitná inicializácia modulu, treba tak urobiť v špeciálnom objekte, ktorý sa nazýva aktivátor mo-

dulu. Tento objekt je vytvorený frameworkom pred naštartovaním modulu. Jeho inštancia je vytvorená z triedy implementujúcej rozhranie `org.osgi.framework.BundleActivator`. Aby ju OSGi framework vedel v module násť, musí byť zadefinovaná v manifeste JAR súboru nasledovne:

```
Bundle-Activator: some.package.ActivatorClassName
```

Rozhranie `BundleActivator` definuje dve metódy. Sú nimi metóda `start(BundleContext)` a metóda `stop(BundleContext)`. Z ich názvov je zjavné, že prvá sa volá pri štarte a druhá pred zastavením modulu.

Čas štartovania modulu je definovaný ako čas vykonania metódy `start(BundleContext)`. Tento čas by mal byť čo najkratší kvôli tomu, aby nebránil štartu iných modulov. Môžu totiž nastať prípady, keď sa aktualizuje modul na ktorom je závislých veľa iných modulov. Aktualizácia takéhoto modulu môže potom spôsobiť (ale aj nemusí) lavínu reštartov modulov, ktoré sú na ňom závislé. Aby sa metóda `start(BundleContext)` vykonala čo najrýchlejšie, je potrebné vykonať v nej len nevyhnutné operácie. Takouto operáciou je napríklad registrácia servisu (3.2.3). Iné operácie, ktoré sú dlhotrvajúce je potom vhodné nechať vykonať asynchrónne - v novom vlákne.

Treba si však uvedomiť, že môže prísť požiadavka modul vypnúť aj v čase, keď inicializácia modulu bežiaca vo zvláštnom vlákne ešte neskončila. Kvôli týmto situáciám je nutné, aby bol modul implementovaný tak, aby vedel tento asynchrónny proces kedykoľvek ukončiť. Znamená to minimálne to, že aktivátor modulu si musí pamätať zoznam vlákien, ktoré spustil. Takisto musí byť schopný ich prerušiť (korektne ukončiť) v prípade potreby.

Na programátora sa preto kladú veľké nároky na znalosť práce vo viacvláknovom prostredí. Aby sa nemusel každý programátor s touto situáciou vysporiadať sám, je zvykom vytvoriť akéhosi manažéra vlákien bežiacich v module. Takéto pomocné implementácie však zatiaľ nie sú súčasťou žiadnej implementácie OSGi.

3.2.3 Servisy

Servis je v OSGi definovaný nasledovne: *"Servis je Java objekt (POJO), ktorý je zaregistrovaný v registry servisov pod jedným alebo viacerými rozhraniami spoločne s dodatočnými atribútmi. Tento objekt môže byť v registri servisov vyhľadaný a použitý inými modulmi"*.

POJO je vo svete jazyka Java často používaná skratka pre anglický výraz *Plain Old Java Object*. Pomenúvajú sa ním bežné Java objekty a to väčšinou v kontexte myšlienky - čím jednoduchší dizaj, tým lepšie.

Vytvorenie a registrácia servisu

Na ilustráciu využijem servis z aplikácie AudioTT, ktorý umožňuje načítanie metadát MP3 súboru. Rozhranie tohoto servisu má iba jednu metódu.

```
public interface MetaReader {
    Object[] read(File file, MetaId[] ids) throws IOException;
}
```

Implementáciu tohoto rozhrania možno zaregistrovať ako servis priamo v aktivátore modulu. Servis sa tak stane prístupným hneď po naštartovaní modulu.

```
public class Activator implements BundleActivator {
    private ServiceReference ref;
    public void start(BundleContext context) throws Exception {
        MetaReader reader = new Id3v1Reader();
        Dictionary<String, String> properties = new Hashtable<String, String>();
        properties.put(METADATA_FORMAT, "id3v1");
        ref = context.registerService(MetaReader.class.getName(), reader,
            properties);
    }
    public void stop(BundleContext context) throws Exception {
        ref.unregister();
    }
}
```

Rozhranie je zaregistrované ešte s dodatočným parametrom, ktorý definuje formát metadát, ktorý je schopný daný servis čítať. Navyše netreba zabudnúť servis odregistrovať pri zastavení modulu.

Takáto explicitná registrácia servisov má isté nevýhody. Konkrétne pri servise `MetaReader` z aplikácie AudioTT sa môže stať, že tento servis nebude nikdy využitý. Čo ak užívateľ nemá žiadne MP3 súbory, ale všetky súbory čo si prezerá sú vo formáte WMA? Kvôli takýmto situáciám by bolo dobré servis zaregistrovať až keď si ho niekto explicitne vypýta. Nielen že by sa tým ušetrila pamäť, ale urýchlil by sa tým aj čas inicializácie modulu, pretože servis by bol nainicializovaný a zaregistrovaný až keď je to potrebné. Takéto správanie sa dá zabezpečiť prostredníctvom tzv. *deklaratívnych servisov* (3.2.4).

Vyhľadanie a použitie servisu

Servis možno vyhľadať viacerými spôsobmi. Spôsob, ktorý využíva priamo API poskytované frameworkom OSGi umožňuje mať proces vyhľadávania pod kontrolov. Nevýhodou však je, že takýto kód je zbytočne ukecaný a navyše často aj chybový. Nasledujúci kód vyhľadá servis `MetaReader`, ktorý je schopný čítať metadáta vo formáte Id3v1. Na prvý pohľad sa môže zdať tento kód bezchybný. Avšak nie je, pretože neráta s tým, že servis `MetaReader` môže byť už odregistrovaný v čase, keď ho voláme.

```
// ziskanie servisu
ServiceReference[] references = context.getServiceReferences(
      MetaReader.class.getName(), "(metadataFormat=Id3v1)");
if (references != null) {
    // zavolanie prveho dostupneho servisu
    serviceRef = references[0];
    MetaReader reader = (MetaReader) bundleContext.getService(serviceRef);
}
// ak uz servis dany modul nevyuziva, je vhodne o tom notifikovat framework
```

```
bundleContext.ungetService(serviceRef);
```

Kvôli zložitosti získavania servisu prostredníctvom API poskytovaného OSGi frameworkom je k dispozícií takzvaný `ServiceTracker`. Táto trieda je tvorí akúsi nadstavbu OSGi, pretože bola definovaná až dodatočne (v prvých verziách špecifikácie absentovala). `ServiceTracker` má v každom čase prehľad o všetkých servisoch daného typu, ktoré sú práve k dispozícií. Oproti predošlému kódu, je použitie triedy `ServiceTracker` jednoduchšie a zároveň nie je chybové.

```
// vytvorenie a spustenie intancie ServiceTracker
ServiceTracker metaReaderTracker = new ServiceTracker(bundleContext,
       MetaReader.class.getName(), "(metadataFormat=Id3v1)");
metaReaderTracker.open();
// zavolanie servisu
((MetaReaderTracker)metaReaderTracker.getService()).read(...);
```

Samozrejme, že volanie `getService()` môže vrátiť `null`. Volanie servisnej metódy bez explicitnej kontroly, či požadovaný servis existuje však často býva v poriadku, pretože systém musí byť pripravné na výnimky typu `RuntimeException` spôsobené z tohoto dôvodu.

3.2.4 Deklaratívne servisy

Deklaratívne servisy (angl. *declarative services*) sú súčasťou dodatku k špecifikácií OSGi [2]. V princípe ide o model, ktorý umožňuje definovať servisy deklaratíve.

Explicitná registrácia servisu pri štarte aplikácie má tú nevýhodu, že v prípade ak daný servis nie je využitý iným modulom, zaberá tento servis zbytočne pamäť. Takzvaný *component model* definovaný v [2] túto situáciu rieši tak, že servisy ktoré modul exportuje sú zadefinované v XML súbore. XML súbor je v tomto prípade použitý z toho dôvodu, že je v ňom možné ľahšie vyjadriť vlastnosti servisu. Konkrétne sa v ňom ľahšie popisujú závislosti servisu na iných servisoch. Cesty k všetkým XML súborom, ktoré deklarujú servisy poskytované daným modulom, sú zadefinované v manifeste JAR súboru, aby ich vedel framework nájsť.

Zaujímavý je spôsob, akým OSGi framework implementuje vyhľadanie XML súborov definujúcich servisy exportované modulmi. Ešte zaujímaveší je však spôsob, akým je môžné poskytnúť frameworku konkrétny servis až v čase, keď si daný servis niekto vypýta.

Každý modul môže byť zaregistrovaný v OSGi frameworku ako tzv. *BundleListener*. Inak povedané, modul môže požadovať od frameworku, aby bol notifikovaný o spustení alebo zastavení každého iného modulu. Tento prístup je využívaný v *extender patterne* (4.3) a takisto aj v takzvanom *cm* module. Modul *cm* (názov vznikol ako skratka od termínu *component model*) sa pozrie do manifestu každého modulu, ktorý sa spustí. V prípade, že v manifeste nájde odkaz na XML súbor definujúci servisy deklarované práve štartovaným modulom, pokúsi sa modul *cm* zabezpečiť týmto servisom najprv ich závislosti. Ak táto operácia prebehne v poriadku, vytvorí pre servisy proxy objekty. Tieto proxy objekty

zaberajú málo pamäte a sú rýchlo inicializované. Modul *cm* ich do registra servisov zaregistruje namiesto reálnych servisov. Keď si niekto vypýta daný servis, proxy objekt sa už postará o vytvorenie skutočného servisu a zamení svoju referenciu v registry servisov za referenciu na skutočný servis.

Kapitola 4

Návrhové vzory modulárnych aplikácií

Návrhový vzor (anglicky *design pattern*) je termín používaný na označenie všeobecného riešenia problémov, vyskytujúcich sa pri návrhu programov. Predstavuje teda akúsi šablónu, ktorá môže byť použitá v rôznych situáciách. Algoritmy za návrhové vzory považované nie sú, pretože riešia konkrétne problémy ale nie problémy samotného návrhu architektúry programu.

Každý návrhový vzor má isté vlastnosti, podľa ktorých vieme určiť či je vhodné ho v danej situácií použiť alebo nie. Keďže táto bakalárska práca je o modulárnych architektúrach, budem sa v tejto kapitole zaoberať návrhovými vzormi vhodnými pre takúto architektúru ako aj vzormi ktoré sú v praxi často používané avšak pre túto architektúru vhodné nie sú.

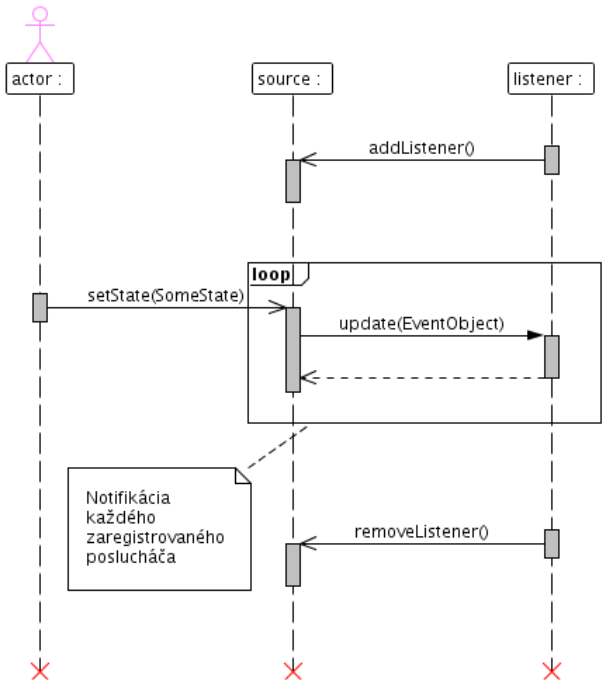
Kvôli stručnosti budem niekedy pojem návrhový vzor pomenúvať jeho skráteným anglickým názvom - *pattern*.

4.1 Listener pattern

Listener pattern sa hodí pre prípady, keď objekty (poslucháči) chcú byť informované o udalosti ktorej zdrojom je iný objekt (zdroj) [6]. Pri použití tohoto návrhového vzoru rozlišujeme týchto aktérov:

1. *Zdroj udalosti* (*event source*) - objekt generujúci udalosť
2. *Udalosť* (*event object*) - objekt reprezentujúci udalosť, ktorý je nositeľom informácií o nej (napr. pri kliknití myšou obsahuje čas, kedy ku kliknutiu došlo alebo súradnice myši)
3. *Poslucháč udalosti* (*event listener*) - objekt, ktorý je informovaný o vzniku udalosti

Výhodou tohoto patternu je to, že poslucháčov reagujúcich na danú udalosť môže byť ľubovoľne veľa. Zároveň zdroj udalosti nevie nič o implementácií svojich poslucháčov. Musí si však pamätať zoznam ich referencií, aby ich mohol vo vhodnej chvíli notifikovať,

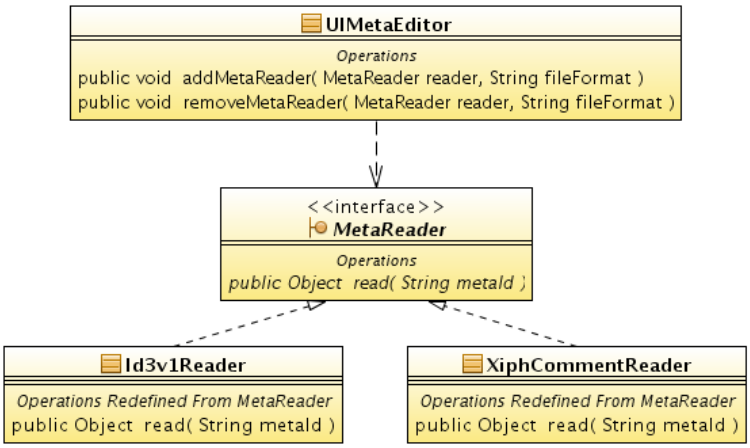


Obrázok 4.1: Sekvenčný diagram listener patternu v systéme so statickými závislosťami

že k udalosti došlo. Zdroj zároveň musí poskytnúť možnosť registrácie a odregistrovania konkrétneho poslucháča.

Typické použitie listener patternu

Kvôli spomínaným vlastnostiam je listener pattern veľmi často využívaný v knižniciach pre tvorbu užívateľských rozhraní, ako sú napríklad *Qt* alebo *Swing*. Aby som bol konkrétnejší, typickým príkladom je tlačítko, ktoré má reagovať na udalosť svojho stlačenia. V knižnici *Swing* je toto tlačítko reprezentované triedou `JButton`. Tlačítko je zdrojom udalosti `ActionEvent`, ktorá sa vytvorí po jeho stlačení. Trieda `JButton` poskytuje zároveň možnosť registrácie a odregistrovania ľubovoľného poslucháča prostredníctvom metód `addActionListener` a `removeActionListener`. Každý poslucháč pritom musí implementovať rozhranie `ActionListener`. Ako vidno na obrázku 4.1, poslucháč sa najprv zaregistruje. Neskôr, keď užívateľ stlačí tlačítko (zmení jeho stav), inštancia triedy `JButton`, ktorá tlačítko reprezentuje, notifikuje o tejto udalosti všetkých zaregistrovaných poslucháčov zavolaním metódy `actionPerformed()` rozhrania `ActionListener`. Poslucháči potom môžu na túto udalosť zareagovať ľubovoľnou akciou - spustiť dajaký výpočet a podobne.



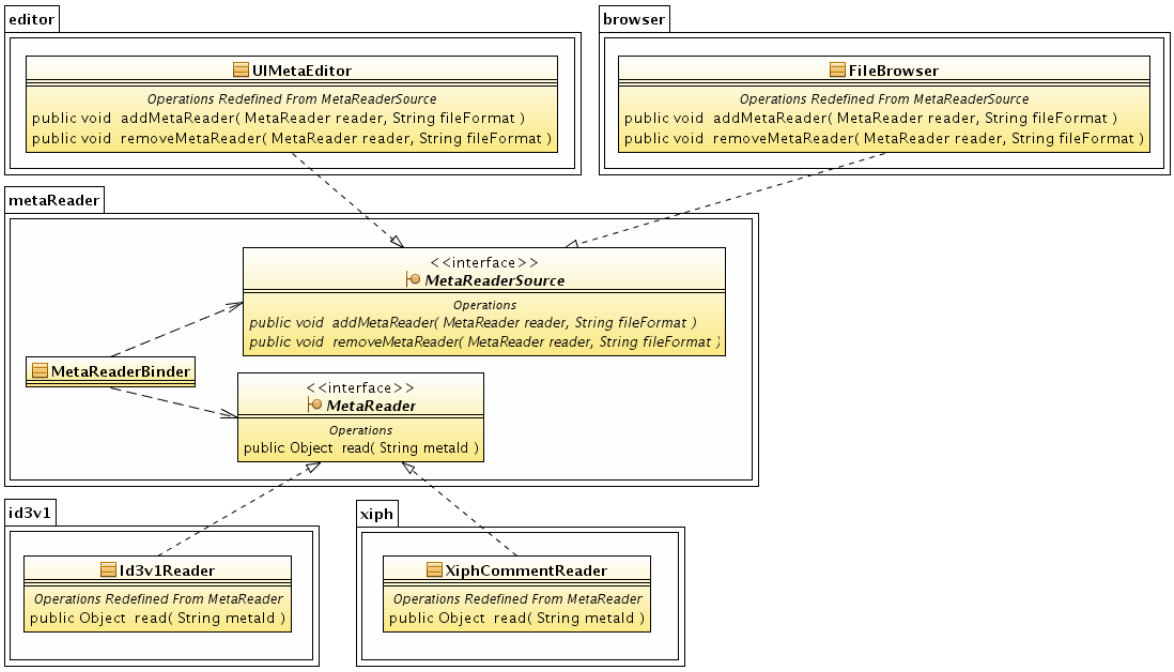
Obrázok 4.2: Class diagram tried aplikácie AudioTT zabezpečujúcich načítanie a zobrazenie meta dát.

Použitie listener patternu v aplikácií AudioTT

V nasledovnom príklade chcem ukázať, ako by som listener pattern využil v aplikácií AudioTT ak by závislosti medzi jej modulmi boli statické. Neskôr popíšem prečo je takýto prístup nevhodné použiť v dynamických modulárných systémoch.

AudioTT umožňuje zobrazovať metadáta rôznych audio súborov v jednej tabuľke. Tieto metadáta sú kvôli rôznym formátom súborov načítavané rôznymi algoritmami. Jednotlivé algoritmy by mohli byť implementované ako listenery, ktoré by boli volané po udalosti stlačenia tlačítka na načítanie meta dát. Tento prístup by sa oproti štandardnému listener patternu popísanému v predchádzajúcej sekcií líšil iba mierne. Poslucháči by sa totiž registrovali pre udalosť požadujúcu načítanie meta dát aj s istým parametrom. V tomto prípade je tým parametrom spomínaný formát súboru. Na druhej strane by zdroj udalosti (tabuľka) nenotifikoval všetkých poslucháčov o tom, že chce načítať metadáta. Notifikoval by iba tých poslucháčov, ktorý sa zaregistrovali pre čítanie daného formátu súboru. Tento dodatočný parameter by teda plnil funkciu akéhosi filtra pri notifikácií poslucháčov.

Obrázok 4.2 znázorňuje vzťahy medzi jednotlivými triedami v AudioTT. `UIMetaEditor` reprezentuje užívateľské rozhranie, prostredníctvom ktorého je možné meta dáta načítať a editovať. V kontexte listener patternu je to zdroj (event source) udalosti načítania resp. zápisu meta dát. `MetaReader` je rozhranie, ktoré musí implementovať každý algoritmus, ktorý načítava meta dáta. V triede `UIMetaEditor` sa tieto implementácie registrujú prostredníctvom metódy `addMetaReader`, ktorá prijíma okrem poslucháča navyše aj spomínaný parameter definujúci formát súboru, pre ktorý sa poslucháč chce zaregistrovať.



Obrázok 4.3: Class diagram tried aplikácie AudioTT zabezpečujúcich načítanie a zobrazenie meta dát v dynamickom modulárnom systéme s použitím listener patternu.

Nevýhody listener patternu v aplikáciach s dynamickou modulárnou architektúrou

V scenári z predošlej sekcie sa na prvý pohľad javí použitie listener patternu vhodné. Treba si však uvedomiť, že v dynamickej modulárnej architektúre môže ľubovoľný modul v ľubovoľnom čase zmiznúť, ako sa aj objaviť. Otázkou teda zostáva, ako zabezpečiť registráciu a odregistrovanie poslucháčov v takomto dynamickom prostredí. Navyše ešte môžeme požadovať, aby existovalo viacero zdrojov tej istej udalosti.

Uvediem konkrétny príklad. V aplikácií AudioTT sa aktualizuje modul reprezentujúci prehliadač súborov. Táto aktualizácia do prehliadača pridala funkciu zobrazovania náhľadu obrázku, uloženého ako meta informácia v audio súbore (niektoré audio súbory zvyknú mať v sebe uložený obrázok albumu, v ktorom bola pieseň vydaná). Znamená to, že aj samotný prehliadač musí mať k dispozícií zoznam implementácií rozhrania `MetaReader`, aby bol schopný obrázky zo súborov načítavať.

Jedno riešenie implementácie listener patternu v takomto prostredí je popísané v [3]. Samozrejme, že by sa dali nájsť ešte aj iné. Napríklad by sme mohli vytvoriť objekt, ktorý by si pamätal zoznam všetkých poslucháčov ako aj zoznam všetkých zdrojov udalostí. Na obrázku 4.3 je tento objekt reprezentovaný ako `MetaReaderBinder`. Tento objekt by reagoval na udalosť spustenia resp. zastavenia ľubovoľného modulu tak, že by sa presvedčil, či niektorá z tried tohoto modulu implementuje rozhranie `MetaReader` resp. `MetaReaderSource`.

Ak by našiel triedu `MetaReaderSource`, pridal by si ju do svojho interného zoznamu a zavolal by metódu `addMetaReader` pre všetkých poslucháčov, ktorých už má vo svojej mape. Ak by našiel triedu `MetaReader`, pridal by si ju do svojej internej mapy a zavolal by metódu `addMetaReader` pre všetky zdroje, ktoré má vo svojom internom zozname. Obdobným spôsobom by `MetaReaderBinder` odregistrovával poslucháčov v prípade zastavenia/odinštalovania mudulu obsahujúceho zdroj alebo poslucháča.

V prostredí *OSGi* by implementácia triedy `MetaReaderBinder` mohla vyzerať nasledovne.

```
package org.audiott.meta.reader;

import java.util.*;
import org.osgi.framework.*;
...

public class MetaReaderBinder implements BundleListener {
    private List<MetaReaderSource> metaReaderSources =
            new ArrayList<MetaReaderSource>();
    private Map<String, List<MetaReader>> metaReaders =
            new TreeMap<String, List<MetaReader>>();

    @Override
    public void bundleChanged(BundleEvent event) {
        Bundle budle = event.getBundle();
        switch (event.getType()) {
        case BundleEvent.STARTED:
            List<MetaReaderSource> sources = findSources(bundle);
            if (!sources.isEmpty()) {
                // zaregistrovanie vetkch listenerov pre nov zdroje
                for (MetaReaderSource source : sources) {
                    Set<String> fileFormats = metaReaders.keySet();
                    for (String format : fileFormats)
                        for (MetaReader reader : metaReaders.get(format))
                            source.addMetaReader(reader, format);
                }
            }
            Map<String, MetaReader> listeners = findListeners(bundle);
            if (!map.isEmpty()) {
                // zaregistrovanie novch posluchov pre exist. zdroje
            }
            break;

        case BundleEvent.STOPPED:
            fireAction(CONTRACT_ACTION, event.getBundle());
            if (!findSources(bundle).isEmpty())
                // odregistrovanie zdrojov nachadz. sa v zastavenom module ...
```

```
            if (!findListeners(bundle).isEmpty())
                // odregistrovanie listenerov nach. sa v zastevenom module ...
            break;
        }
    }
    ...
}
```

V porovnaní s listener patternom tak ako je popísaný na začiatku kapitoly by sme teda museli pre každé rozhranie listenera vytvoriť navyše ešte jedno rozhranie pre zdroj udalostí a triedu, ktorá by tieto objekty prepájala. Výhodou by bolo jedine to, že poslucháči by nemuseli implementovať logiku svojej registrácie a odregistrovania v zdrojoch udalostí. Poslucháči a zdroje by však zrejme museli byť singletony, inak by bolo problematické na tieto objekty získať referencie.

Ako je z predchádzajúcich príkladov vidno, nech sa na listener pattern pozrieme akokoľvek, implementovať princíp registrovania a odregistrovávania poslucháčov v zdrojoch udalostí je v aplikácií s dynamickou modúlárnou architektúrou nevýhodné. Je totiž potrebé zaimplementovať množstvo logiky navyše, ktorá rieši spomínanú registráciu.

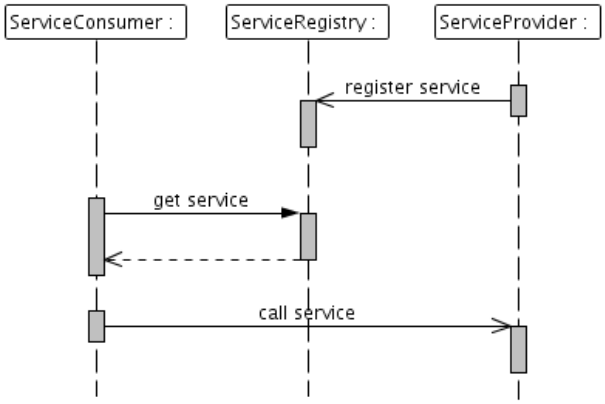
4.2 Whiteboard pattern

Ako sa ukázalo, pre modulárne prostredie je *listener* pattern nevyhovujúci. Návrhový vzor *whiteboard* [3] je jeho veľmi efektívnou náhradou.

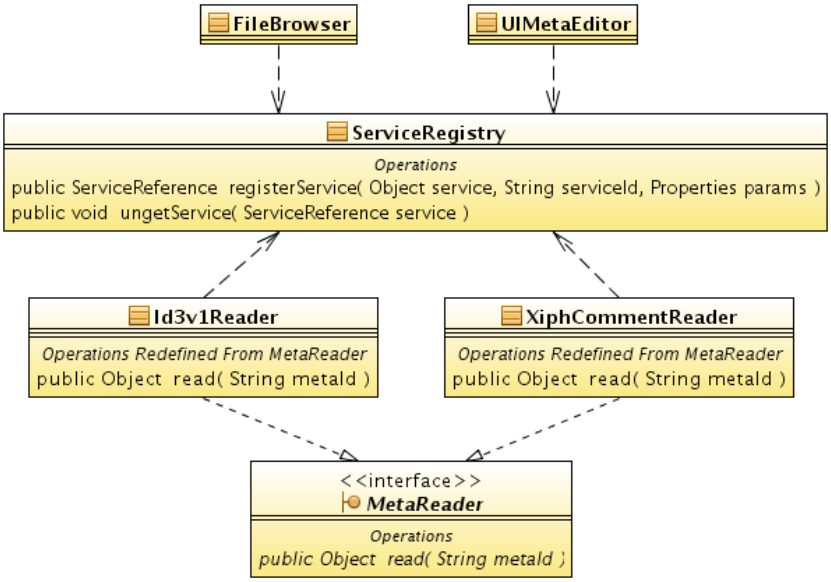
Whiteboard pattern sa oproti *listener patternu* líši hlavne tým, že využíva iba jeden objekt, ktorý slúží na registráciu všetkých typov poslucháčov. Tento objekt sa nazýva *register servisov* (anglicky *service registry*). (Pozn.: Aby som používal terminológiu z *OSGi*, ďalej budem poslucháčov pomenúvať už iba ako servisi.) Register servisov *OSGi* navyše umožňuje získať konkrétny servis v ňom zaregistrovaný na základe názvu rozhrania, ktoré servis implementuje, a na základe dodatočného zoznamu parametrov, ktoré boli servisu priradené pri jeho registrácií v registry.

Postup registrácie a získania servisu je znázornený na obrázku 4.4. Obrázok 4.5 znázoňuje závislosti medzi triedami zo sekcie 4.1 pri použití whiteboard patternu. Hlavné znaky a výhody tohto prístupu oproti listener patternu sú potom zjavné:

- Register servisov je zdieľaný. Každý v ňom môže zaregistrovať servis alebo si dajaký vypýtať.
- Miesto toho, aby logiku registrácie a odregistrovania poslucháčov obsahoval každý zdroj udalosti, je táto logika zaimplementovaná iba raz a to v registri servisov.
- Zdroje udalostí nemusia implementovať žiadne rozhranie.



Obrázok 4.4: Sekvenčný diagram whiteboard patternu.



Obrázok 4.5: Class diagram tried znázorňujúci použitie whiteboard patternu v aplikácií AudioTT.

Implementácia whiteboard patternu v OSGi

Framework *OSGi* už obsahuje servis registrov so všetkou potrebnou funkcionalitou, takže podstatne zjednodušuje implementáciu.

Registrácia servisu MetaReader Ak chceme, aby modul obsahujúci triedu `Id3v1Reader` zaregistroval tento servis hneď po svojom štarte a odregistroval ho po jeho zastavení, musíme to urobiť v metódach reprezentujúcich životný cyklus modulu - čiže v "aktivátore" modulu, alebo ešte inak, v triede implementujúcej rozhranie `BundleActivator`. Inštanciu triedy `Id3v1Reader` zaregistrujeme pod rozhraním, ktoré implementuje, a s dodatočným parametrom, ktorý bude určovať, že je schopný čítať iba meta dáta vo formáte *ID3v1* [12]. Obdobným spôsobom by sme zaregistrovali aj reader pre iný formát meta dát.

```
package org.audiott.meta.id3v1;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
public class Activator implements BundleActivator {

    private ServiceRegistration serviceRegistration;

    @Override
    public void start(BundleContext context) throws Exception {
        Hashtable properties = new Hashtable();
        properties.put(Constants.SUPPORTED_METADATA, "Id3v1");
        Object service = new Id3v1Reader();
        serviceRegistration = context.registerService(
                MetaReader.class.getName(), service, properties);
    }

    @Override
    public void stop(BundleContext context) throws Exception {
        serviceRegistration.unregister();
    }

}
```

Implementácia tried UIMetaEditor a FileBrowser V *OSGi* je aj táto implementácia opäť dostatočne jednoduchá kvôli API, ktoré nám *OSGi* poskytuje. Chceme, aby sa moduly správali tak, že si vypýtajú dané servisy bezprostredne pred tým ako ich budú potrebovať. Takýto prístup je dobré používať všade, kde je to možné, pretože umožňuje šetriť pamäť až do chvíle, kým si servis niekto explicitne nevyžiada (viď. 3.2.4).

Servisy poskytujúce funkcionalitu čítania meta dát získame z registra servisov prostredníctvom názvu rozhrania, ktoré musia tieto servisy implementovať, a dodatočného parametra, ktorý špecifikuje formát metadát, ktoré chceme načítať. Tento dodatočný parameter je

definovaný textovým reťazcom vo formáte, ktorý definuje špecifikácia LDAP (tzv. LDAP Filter [10]).

```
package org.audiott.ui;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class UIMetaEditor implements BundleActivator {

    private BundleContext context;
    public void start(BundleContext context) throws Exception {
        this.context = context
    }
    public void stop(BundleContext context) throws Exception {}

    ...

    private void readMetaData() {
        Object[] services = context.getServices(
                MetaReader.class.getName(), "(supportedMetadata=Id3v1)");
        for (Object s : services) {
            MetaReader reader = (MetaReader) s;
            ...
            String author = (String) reader.read(file, Constants.AUTHOR);
            String album = (String) reader.read(file, Constants.ALBUM);
            String title = (String) reader.read(file, Constants.TITLE);
            ...
    }
    ...
```

Obdobným spôsobom je zimplementovaná aj trieda `FileBrowser`.

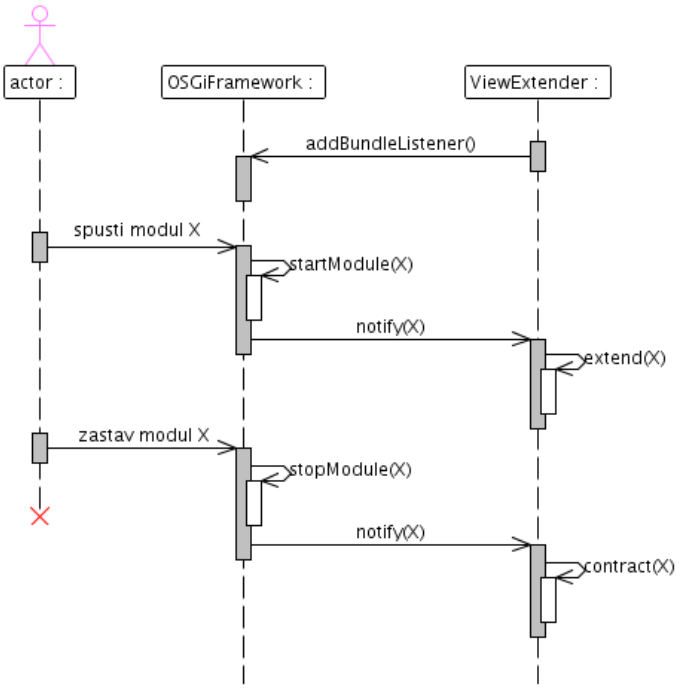
Záver

Listener pattern nie je vhodný pre implementáciu v aplikáciách s modulárnou architektúrou, kvôli komplikovanej infraštruktúre, zabezpečujúcej registráciu a odregistrovanie poslucháčov (listenerov). Whiteboard pattern tento problém rieši jednoducho, zavedením jedného objektu, ktorý slúži ako globálny register servisov.

4.3 Extender pattern

Extender pattern je veľmi efektívny a ľahko zrozumiteľný návrhový vzor, ktorý je vhodné použiť v situáciách, keď chceme do systému pridať servis alebo komponent dynamicky, pričom vlastnosti tohoto komponentu sú zadefinované deklaratívne. Pri tomto návrhovom vzore rozlišujeme dva druhy aktérov:

- *extender* - komponent, ktorý zabezpečuje rozšírenie systému o extension



Obrázok 4.6: Sekvenčný diagram znázorňujúci interakciu medzi aktérmi extender patternu.

- *extension* - komponent, o ktorý chceme systém rozšíriť

Kardinalita medzi týmito aktérmi je one-to-many (1..*). Jeden extender totiž slúži na rozšírenie systému o jeden alebo o viacero komponentov (extensions).

Vysvetlenie patternu na príklade z aplikácie AudioTT

AudioTT zobrazuje informácie v paneloch. Tieto panely sa nachádzajú v rôznych častiach užívateľského rozhrania. Pomocou extender patternu je možné naprogramovať panel a určiť jeho umiestnenie v užívateľskom rozhraní deklaratívne a pritom zachovať dynamický charekter modulu (panel sa môže za behu aplikácie aktivovať ako aj deaktivovať).

V OSGi frameworku je zaregistrovaný modul `org.audiott.ui`, ktorý obsahuje triedu `ViewExtender`. Táto trieda reaguje na každé spustenie ľubovoľného modulu frameworku tým, že preskúma jeho manifest. Ak v ňom nájde deklaráciu hovoriacu o tom, že modul obsahuje panel užívateľského rozhrania, extender vytvorí inštanciu tohoto panelu a rozšíri ním už existujúce užívateľské rozhranie aplikácie. Aby extender vedel, kde má panel zobraziť, obsahuje manifest ešte informáciu o tejto polohe.

Pomocná trieda ExtenderHelper

Všetky implementácie extender patternu majú spoločné to, že musia preskúmať manifest štartujúcich modulov. Avšak spôsob akým extender daný komponent (extension) do

systému začlení sa už v jednotlivých implementáciách líši.

V aplikácií AudioTT postavenej na frameworku OSGi, je táto spoločná črta extenderov zaimplementovaná v triede `ExtenderHelper`. Konkrétne extendery potom `ExtenderHelper` využívajú takým spôsobom, že sa v ňom zaregistrujú ako poslucháči, ktorí chcú byť notifikovaní o udalosti spustenia/zastavenia modulu obsahujúceho konkrétnu deklaráciu v manifeste práve spúšťaného/zastavovaného modulu (Pozn.: použitie listener patternu je v tomto prípade vhodné, pretože vzťah medzi helperom a extendermi je *1..** a zároveň je `ExtenderHelper` vždy k dispozícií, pretože tvorí jadro aplikácie).

```
package org.vician.utils.osgi;

import ...

public class ExtenderHelper implements BundleListener {

    protected void activate(ComponentContext cmpContext) {
        // ExtenderHelper chce by notifikovan o spusten/zastaven modulu
        // bez ohadu na obsah jeho manifestu.
        cmpContext.getBundleContext().addBundleListener(this);
    }

    protected void deactivate(ComponentContext cmpContext) {
        cmpContext.getBundleContext().removeBundleListener(this);
    }

    public void addListener(String key, Extender e) {
        // Registrcia extendera, ktor chce by notifikovan o
        // tarte/zastaven modulu v prpade prtomnosti deklarcie
        // "key" v manifeste modulu.
    }

    @Override
    public void bundleChanged(BundleEvent event) {
        Dictionary bundleHeaders = bundle.getHeaders();
        for (String key : registeredListeners.keySet()) {
            // Preskmanie manifestu a zistenie, i je pre dan
            // deklarcie zaregistrovan dajak extender.
            ...
            if (isRegistered(extender, key)) {
              switch (event.getType()) {
              case BundleEvent.STARTED:
                fireAction(EXTEND_ACTION, key, event.getBundle());
              break;
              case BundleEvent.STOPPED:
                fireAction(CONTRACT_ACTION, key, event.getBundle());
              break;
```

```
            }
        }
    }
}
```

Extender užívateľského rozhrania - ViewExtender

`ViewExtender` je implementovaný pomocou triedy `ExtenderHelper` jednoduchým spôsobom. Stačí, že sa zaregistruje v triede `ExtenderHelper` tak, aby bol notifikovaný o spustení a zastavení tých modulov, ktoré majú v manifeste deklaráciu s kľúčom "AudioTT-View". `ExtenderHelper` vykonáva notifikáciu volaním metód `extend(...)` alebo `contract(...)`. Parametre týchto metód posiela obsahujú okrem kľúča deklarácie manifestu aj jej hodnotu. `ViewExtender` očakáva, že bude táto hodnota vo formáte *position:class*, kde *position* je pozícia, na ktorej má byť panel zobrazený (*LEFT*, *RIGHT*, ...) a *class* je meno triedy, reprezentujúcej tento panel.

```
package org.audiott.ui;
import ...
public class ViewExtender implements Extender {

    protected void setExtenderHelper(ExtenderHelper helper) {
        helper.addListener("AudioTT-View", this);
    }

    public void extend(String key, String value, Bundle bundle) { ... }
    public void contract(String key, String value, Bundle bundle) { ... }
    ...
}
```

Vytvorenie inštancie UI panelu Tento pattern má síce výhody v tom, že umožňuje deklaratívne popísať určité vlastnosti pri zachovaní dynamického charakteru modulu, avšak môže byť trochu zradný hlavne kvôli procesu vytvárania inštancie konkrétnej triedy. V prípade triedy `ViewExtender` potrebujeme vytvoriť inštanciu triedy zadeklarovanej spomínaným reťazcom *position:class*.

Vo frameworku OSGi je problémom pri vytvorení tejto inštancie to, že OSGi rozširuje pravidlá viditeľnosti jazyka Java (3.2). Inštanciu preto nie je možné vytvoriť klasickým spôsobom jazyka Java `getClass().forName(menoTriedy).newInstance()`. Takéto volanie by vyvolalo výnimku `ClassNotFoundException`, pretože class loader modulu obsahujúceho `ViewExtender` nevie načítať triedy iných modulov (panel užívateľského rozhrania, ktorého inštanciu chceme vytvoriť sa nachádza v inom module). Preto treba použiť metódu `bundle.loadClass(menoTriedy).newInstance()`. Táto metóda zabezpečí, aby sa trieda načítala class loaderom, ktorý poskytuje modul reprezentovaný objektom `bundle`.

Využitie extender patternu v iných systémoch, jeho výhody a nevýhody

Jedna nesporná výhoda extender patternu je v tom, že umožňuje deklaratívne zadefinovať "parametre" komponentu, o ktorý systém rozširujeme. V niektorých prípadoch však môže byť tento deklaratívny prístup obmedzujúci. Programátorovi navyše neumožňuje robiť príliš veľké zásahy do systému.

Každý extender beží s istými právami voči hosťovskému systému. Preto si pri jeho programovaní treba dávať pozor na to, aké operácie robí extender v záujme komponentu, o ktorý chceme systém rozšíriť. V istých prípadoch by totiž mohlo dôjsť k tomu, že by takýto komponent robil zásahy do systému prostredníctvom extendera i keď samotný komponent na takéto zásahy nemá žiadne oprávnenie.

Na druhej strane, to že extender má väčšie práva ako komponent môže byť výhodou. Príkladom je napríklad situácia, kedy by extender vytvoril tabuľky v databáze pre komponent, ktorý by tieto práva nemal. Samozrejme, že komponent by musel mať práva na select a iné operácie súvisiace s dotazovaním, aby vedel s databázou pracovať.

Kapitola 5

Záver

Najväčším prínosom dynamickej modulárnej architektúry je zrejme to, že umožňuje vyvíjať aplikácie, ktoré nie je potrebné kvôli ich aktúalizácií reštartovať celé. Stači reštartovať iba moduly, ktorých sa aktualizácia týka. V najhoršom prípade je však potrebné reštartovať ešte aj tie moduly, ktoré sú na aktualizovaných moduloch závislé. Moduly možno rovnako počas behu aplikácie pridávať a odoberať a tým meniť funkcionalitu poskytovanú aplikáciou počas jej behu.

V prostredí jazyka Java existuje viacero frameworkov, ktoré umožňujú takéto aplikácie vyvinúť. Framework *OSGi* má oproti ostatným tú výhodu, že je štandardizovaný a je k dispozícií viacero jeho implementácií. Navyše framework OSGi rozširuje pravidlá viditeľnosti jazyka Java na balíčky (Java package) prostredníctvom zavedenia špeciálneho class loadera pre každý modul. To núti programátorov k trochu lepším návykom, čo sa týka vývoja modulov a návrhu internej hierarchie tried v rámci spomínaných balíčkov. Programátori si totiž musia dávať pozor na to, ktoré balíčky nechajú viditeľnými aj pre iné moduly, a ktoré sú určené iba pre interné operácie v rámci modulu. Výsledkom potom je, že pri dobrom návrhu modulov a ich závislostí je úrdžba aplikácie, v porovnaní s veľkými monolitickými aplikáciami, omnoho jednoduchšia.

Hlavnou nevýhodou aplikácií založených na modulárnej architektúre je zvýšená zložitosť ich vývoja. Treba totiž počítať s faktom, že sa môže ľubovoľný modul v ľubovoľnom čase spustiť ako aj zastaviť. Požiadavka na zastavenie modulu môže prísť asynchrónne voči práve vykonávanej operácií modulu. Z implementačného hľadiska je teda takáto aplikácia viacvláknová. Na programátorov sa preto kladie požiadavka dobrej znalosti vývoja viacvláknových aplikácií.

V tejto bakalárskej práci som sa okrem analýzy modulárnych architektúr a spôsobom vývoj aplikácií na nich založených zaoberal aj tromi návrhovými vzormi. Zistil som, že na účely propagácie udalostí v systéme je v rámci modulárnej architektúry - vzhľadom na dynamickosť modulov - štandardný návrhový vzor *listener* nevhodný. Ako jeho náhrada slúži návrhový vzor *whiteboard*. Veľmi elegantným sa ukázal byť v modulárnom prostredí návrhový vzor *extender*. Ten umožňuje deklaratívne popísať funkcionalitu, ktorú so sebou modul do systému prináša, pri zachovaní dynamických vlastností tohoto modulu.

Spomínané návrhové vzory som využil v aplikácií AudioTT napísanej v jazyku Java a založenej na frameworku OSGi.

V rámci tejto bakalárskej práce nie sú popísané všetky aspekty modulárnych architektúr. Potenciál takejto architektúry je zjavne omnoho väčší. Ako príklad uvediem spôsob, akým sa dajú vyvíjať distribuované aplikácie. Pri použití väčšiny nástrojov umožňujúcich vývoj distribuovaných aplikácií (napríklad *CORBA* [11]) je nutné už vopred poznať rozdelenie aplikácie na časti, ktoré budú bežať na rôznych počítačoch. Pri vývoji aplikácie s dynamickou modulárnou architektúrou to však vopred vedieť netreba. Dôvod je taký, že jednotlivé moduly medzi sebou komunikujú prostredníctvom servisov bez ohľadu na to, či bežia všetky na jednom počítači alebo na viacerých. Táto vlastnosť umožňuje vykonať rozhodnutie o distribuovaní jednotlivých modulov v čase, keď je celá aplikácia v podstate už vyvinutá.

Literatúra

[1] The OSGi Alliance: *OSGi Service Platform, Core Specification, Release 4*, 2007.
http://www.osgi.org/Download/File?url=/download/r4v41/r4.core.pdf

[2] The OSGi Alliance: *OSGi Service Platform, Service Compendium, Release 4*, 2007.
http://www.osgi.org/Download/File?url=/download/r4v41/r4.cmpn.pdf

[3] The OSGi Alliance: *Listeners Considered Harmful: The "Whiteboard" Pattern*, 2004.
http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf

[4] Wikipedia: *Software architecture*
http://en.wikipedia.org/wiki/Software_architecture

[5] Sun Microsystems: *JAR File Specification*, San Francisco, 2005.
http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html

[6] Mark Grand: *Patterns in Java*, John Wiley and Sons, 2002.

[7] Brian Goetz - Tim Peierls - Joshua Bloch - Doug Lea - Joseph Bowbeer - David Holmes: *Java concurrency in practice*, Addison-Wesley, 2006.

[8] Bruce Eckel, Chuck Allison: *Thinking in Java*, Prentice Hall, Michigan, 2000.

[9] Spring Dynamic Modules for OSGi Service Platforms
http://www.springsource.org/osgi

[10] By Tim Howes - Mark Smith - Gordon S. Good: *Understanding and deploying LDAP directory services*, Addison-Wesley, 2003, s. 74.

[11] By Gerald Brose - Andreas Vogel - Keith Duddy: *Java Programming with CORBA*, John Wiley & Sons, 2001.

[12] Martin Nillson: *Id3v2.3.0 informal standard*, 1999.
http://www.id3.org/id3v2.3.0

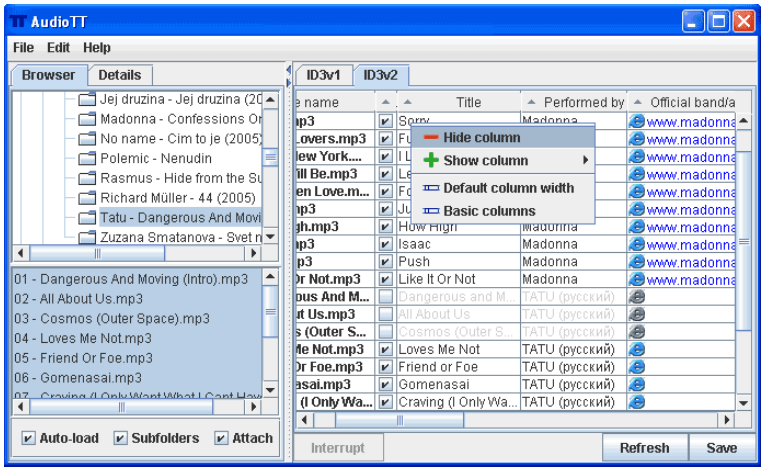
Dodatok A

Aplikácia AudioTT

AudioTT (Audio Tagging Tool) je aplikácia, ktorú som pôvodne vytvoril ako môj ročníkový projekt.

Funkcionalita aplikácie AudioTT umožňuje prezerať a editovať metadáta audio súborov formátu MP3. Z pohľadu užívateľského rozhrania sú užívateľom zvolené MP3 súbory načítané do tabuľky, v ktorej stĺpcoch sú zobrazované jednotlivé metadáta (titul, autor piesne, názov albumu, ...).

Architektúra aplikácie Pôvodná verzia AudioTT je vytvorená klasickým spôsobom, čiže ako monolitická aplikácia. V tejto bakalárskej práci sme sa zaoberali tým, ako ju možno prebudovať na modulárnu aplikáciu založenú na frameworku OSGi. Využili sme pritom návrhové vzory popísané v kapitole 4. Sústredili sme sa pri tom na vyskladnenie funkcionality umožňujúcej načítavanie a zápis metadát do modulov. V bakalárskej práci sa na jednotlivé moduly odvolávam ako na súčasť demoštračnej aplikácie.



Obrázok A.1: Hlavná obrazovka aplikácie AudioTT.