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

Diplomová práce

Aspektově orientovaná rozšíření komponentového modelu

Plzeň, 2012 Jakub Truneček

Originální zadání

Prohlášení

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

V Plzni dne 17. května 2012

. . . .

Jakub Truneček

Abstrakt

Aspektově orientované programování umožňuje vyjmutí protínajících potřeb do samostatných celků za použití aspektů a návrhový vzor depedendency injection poskytuje způsob jak řešit vzájemné závislosti jednotlivých tříd mezi sebou.

Cílem práce je rozšířit komponentový aplikační rámec CoSi o podporu aspektově orientovaného programování a dále implementovat deklarativní formu registrace a vyhledávání služeb pomocí technik představených v dependency injection.

První polovina práce předkládá teoretický úvod do všech dotčených oblastí a v druhé části práce popisuje implementaci jednotlivých rozšíření společně se zhodnocením jejich přínosu a funkčnosti.

Abstract

Aspect-oriented programming allows to remove crosscuting concerns to individual units using aspects. Depedendency injection design pattern provides a way to address interdependence of various classes among themselves.

The goal of this thesis is to extend the CoSi component framework by Aspect-oriented programming support and implement the declarative form of registration and lookup of services using the techniques presented in dependency injection.

The first half of the thesis presents a theoretical introduction to all affected areas and the second part describes the implementation of each extension together with an evaluation of their contribution to the system and their functionality.

Poděkování

Touto formou bych rád poděkoval panu Ing. Přemyslu Bradovi, MSc., Ph.D. za jeho věčné připomínky v průběhu realizace práce a za jeho čas, který práci věnoval.

Obsah

1 Úvod **1**

2 Komponentové modely **2**
2.1 Znovupoužitelnost kódu 2
2.2 Úvod do komponent 3
2.3 Komponentové modely a frameworky 6

3 Aspektově orientované programování **19**
3.1 Motivace a základní konstrukty 20
3.2 Možnosti realizace v Javě 22
3.3 Implementace pro Javu 24
3.4 AOP v komponentových modelech 28

4 Dependency Injection **30**
4.1 Motivace 30
4.2 Formy DI 33
4.3 Service Locator 34
4.4 Vztah pojmu Inversion of Control k DI 35
4.5 Implementace pro Javu 36
4.6 DI v komponentových modelech 38

5 Obecné úpravy CoSi **40**
5.1 Popis obecných úprav 40
5.2 Implementace obecných úprav 47
5.3 Zhodnocení implementace obecných úprav 57

6 Rozšíření CoSi o Dependency Injection **63**
6.1 Formulace zadání 63
6.2 Analýza aktuálního stavu 64
6.3 Návrh rozšíření 66

6.4 Implementace rozšíření 70
6.5 Zhodnocení funkčnosti rozšíření 73

7 Rozšíření CoSi o AOP **78**
7.1 Formulace zadání 78
7.2 Návrh rozšíření 79
7.3 Implementace rozšíření 85
7.4 Zhodnocení funkčnosti AOP 89

8 Závěr **93**
8.1 Návrhy na další rozšíření 94

Seznam použitých zkratek **96**

Seznam obrázků **97**

Literatura **98**

Kapitola 1

Úvod

Tato práce si klade za úkol rozšířit komponentový model CoSi, který je vyvíjen na katedře informatiky a výpočetní techniky, o prvky aspektově orientovaného programování a formu dependency injection na úrovni služeb.

Komponentové modely se v posledních 15 letech stávají čím dál tím více užívaným přístupem při návrhu a implementaci aplikací, což je dáno zejména strmým navýšením poptávky po softwarovém vybavení a stále většímu tlaku na snížení nákladů na vývoj a údržbu kódu. Teoretický úvod do komponentových modelů a motivaci k jejich vzniku předkládá druhá kapitola nazvaná *Komponentové modely*.

Aspektově orientované programování je koncept, který nabízí programátorovi možnost rozdělit program do logicky souvislých a znovupoužitelných celků i tam, kde klasický objektově orientovaný přístup nestačí, tedy zejména v takzvaných protínajících potřebách. Principy a pojmy používanými v AOP se zabývá třetí kapitola s názvem *Aspektově orientované programování.*

Návrhový vzor Dependency Injection (DI) poskytuje návrhově čisté a ověřené řešení, jak řešit závislosti mezi třídami. Čtvrtá kapitola, nazvaná *Dependency Injection*, poskytuje vysvětlení pojmů, vymezení problému a popis forem DI, kterými je tento problém řešen.

Obecným cílem práce je přidat do komponentového frameworku CoSi rozšíření, která jednak ulehčí práci při psaní komponent a v druhé řadě přidají nové možnosti pro zvýšení modularity celého systému. Na základě teoretických informací z prvních třech kapitol je v kapitolách *Obecné úpravy CoSi, Rozšíření CoSi o Dependency Injection* a *Rozšíření CoSi o AOP* navrženo a implementováno několik rozšíření. Všechna jsou navrhována tak, aby maximální možnou mírou zapadala do současného konceptu CoSi a aby zároveň využívala ověřených postupů, které jsou zejména čerpány z referenčního komponentového modelu OSGi.

Kapitola 2

Komponentové modely

Tato kapitola poskytuje úvod do komponentových technologií a modelů, které jsou nezbytné pro další části práce. V úvodní sekci kapitola pojednává o potřebách, které vedly k zavedení softwarové disciplíny založené na komponentách (*Component Based Software Engineering - CBSE*). Další část kapitoly pak pokrývá základní teoretické pojmy a definice vyskytující se v tomto odvětví. A konečně poslední část dává nahlédnout do konkrétních komponentových modelů včetně cílového modelu a aplikačního rámce CoSi[1].

2.1 Znovupoužitelnost kódu

Téměř ve všech technických odvětvích je běžné, že v průběhu vývoje produktu se znovu-používají již existující komponenty či systémy [19]. Kupříkladu společnost zabývající se výrobou osobních automobilů využívá již hotových celků jako jsou motory, převodové skříně či brzdové systémy.

Tuto myšlenku lze v celku snadno převézt do světa software. Vývojový proces založený na komponentách spočívá ve vývoji komponent a jejich následnému skládání do výsledného produktu. Přestože přínosy tohoto přístupu jsou známy již mnoho let (McIlroy, 1968), větší nárůst jeho užívání je zaznamenatelný až v posledních 15 letech.

Tradičně se software vyvíjel na míru požadavkům s hlavním zaměřením na dodržení termínů dodání a výsledné ceny. Nehledělo se příliš na to, jestli a jak bude možné vyvíjený software dále rozšiřovat, popřípadě zda-li bude možné již vyvinutý software, byť jen z části, použít v dalších, třeba i podobných projektech. Přechod z tohoto tradičního přístupu vývoje na vývoj založený na znovupoužitelnosti můžeme přičítat enormnímu zvyšování poptávky po softwarovém vybavení do rozličných odvětví. Navyšování poptávky logicky vedlo k požadavkům na snižování nákladů, a to

[1]Zkratka anglického výrazu *Components Simplified*

jak na vývoj tak i na údržbu takového vybavení.

Odvětví znovupoužitelnosti kódu se dá rozdělit do třech podskupin [19]:

1. **Znovupoužitelnost na úrovni celých aplikací** - aplikace jsou již vyvíjeny s tímto ohledem a dají se úpravou konfigurace, popřípadě lehkými úpravami kódu, použít pro více zákazníků. Klasickým příkladem jsou například CMS[2] řešení či vysoce konfigurovatelné internetové obchody. Tento přístup není v kontextu této práce důležitým a není již v dalším textu zmiňován.

2. **Znovupoužitelnost na úrovni tříd, funkcí a knihoven** - tento přístup je asi nejtradičnějším a dá se říci, že je aplikován nejdéle. Jednotlivé třídy či více tříd pohromadě (balíčky, knihovny) mají obecnou funkčnost a mohou být využívány v různých systémech. Tento přístup k znovupoužitelnosti kódu krátce popisuje podsekce *OOP a znovupoužitelnost* na této straně.

3. **Znovupoužitelnost na úrovni komponent** - komponenty jsou samostatné jednotky nasazení, deklarují své závislosti a poskytované funkčnosti. Celý zbytek kapitoly se zabývá tímto přístupem.

2.1.1 OOP a znovupoužitelnost

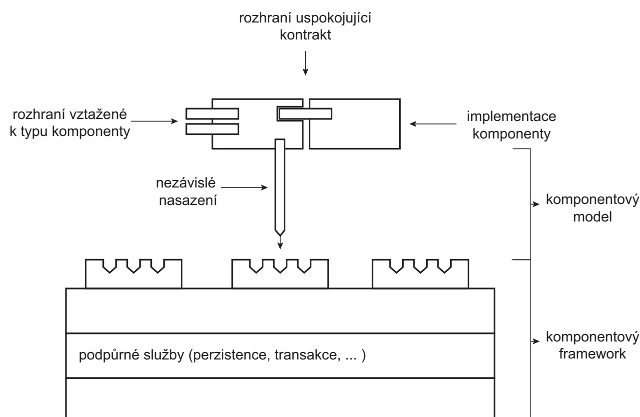
Zavedení objektově orientovaného programování si kladlo jako jeden ze svých hlavních cílů poskytnout možnost znovupoužitelnosti jako rychlejší, snadnější, systematičtější a pevně integrovaný proces do běžného vývoje [23]. Nicméně praxe užití OOP[3] ukázala, že objekt, potažmo třída, jakožto hlavní prvek modularity, je až příliš konkrétní. Jeho implementace nemůže být vždy jako čistý black-box[4], nehledě na to, že při sestavování výsledného systému je vždy potřeba (linkování a kompilace). Přesto je OOP dnes vnímáno jako jeden z hlavních přístupů pro zvýšení znovupoužitelnosti kódu a je prakticky jeden z nejrozšířenějších úzů v softwarovém světě vůbec.

Výše uvedené „zklamání" z objektově orientovaného programování však dalo za vznik softwarové disciplíny založené na komponentách, kde je samozřejmě OOP využíváno a je jeho nedílnou součástí.

2.2 Úvod do komponent

Na softwarové inženýrství založené na komponentách lze nahlížet jako na dodržování určitého *návrhového vzoru*. Tento návrhový vzor můžeme pojmenovat jako *komponen-*

[2]*Content management system* je aplikace pro správu obsahu, česky se používá název *redakční systém*
[3]Zkratka z anglického výrazu *Object Oriented Programming* - Objektově orientované programování
[4]*Black-box* - výraz se užívá pro skrytí implementace určité funkčnosti před okolním světem.



Obrázek 2.1: Komponentově založený návrhový vzor [převzato z 5]

tově založený a jako takový ho zobrazuje obrázek 2.1.

Komponentový model, jak ho popisuje Bachman [1], je v tomto návrhovém vzoru vnímám jako sada rozhraní a kontraktů. Tyto kontrakty a rozhraní definují různé typy komponent, které mohou v komponentovém modelu figurovat. Komponentový framework definuje běhové prostředí pro komponenty a poskytuje podpůrné služby pro komponenty. Může se jednat o služby svázané s konkrétním komponentovým modelem či dokonce typem komponenty, příkladem těchto služeb může být řízení životního cyklu komponent, perzistenční služba, transakční služba a další. Aby mohly být jednotlivé komponenty nezávisle zavedeny do systému, musí implementovat rozhraní, které určí jejich typ. Jedna komponenta může implementovat i více rozhraní.

Další text této kapitoly definuje jednotlivé pojmy figurující v CBSE[5], nicméně stále ještě nepadla zmínka o tom, co je motivem k jeho použití. Přínosy se mohou zdát celkem zjevné, přesto jsou pro úplnost uvedeny v následujícím seznamu tak, jak je definuje Bachman [1] a doplňuje autor této práce:

- **Nezávislá rozšíření.** Komponenty jsou jednotky rozšířitelnosti (modularity) a komponentový model přesně definuje, jak mají být tato rozšíření (komponenty) tvořena. Zároveň pomocí rozhraní a kontraktů je zaručeno, že komponenta jasně deklaruje svou funkčnost, závislosti a přínos do systému, tudíž mohou být vyvíjeny a nasazovány do systému nezávisle na ostatních komponentách.

- **Snížený časový nárok na vývoj systému a tím i nákladové stránky.** Tento přínos je zjevný ze znovupoužitelnosti kódu. Sestavení nového systému z již hotových částí je dramaticky rychlejší, než psaní systémů stále od začátku.

- **Vylepšení kvality výsledných systémů.** Dá se předpokládat, že užitím kompo-

[5]Component Based Software Engineering - zkratka pro softwarové inženýrství založená na komponentách. Tato zkratka se často vyskytuje společně se zkratkou CBD (Component Based Development)

nent, které jsou již v produkčním prostředí několikrát použity, povede k vyšší kvalitě software.

Je však potřeba uvést i rizika, která jsou s CBSE spojena. Ta formuloval Liška ve své diplomové práci [12], v mírně upravené verzi zní:

- Není vždy zřejmé, co je možné znovu-použít a co ne. Požadavky na komponenty nejsou vždy známé předem a jsou často nepřesně nebo nedostatečně formulovány.
- Pro splnění znovupoužitelnosti musí být komponenty dostatečně obecné, rozšiřitelné a přizpůsobitelné, což může vést ke komplikovanější použitelnosti a k zvýšeným nárokům na výpočetní výkon.
- Úprava komponent může být drahá. Zejména pokud se jedná o komponenty dodávané třetí stranou.

2.2.1 Komponenta

I přesto, že máme poměrně ucelenou představu, co to komponenta je, neexistuje žádná obecně uznávaná definice. To je způsobeno zejména tím, že CBSE je ve srovnání s epochou vývoje software poměrně mladou disciplínou a praktické využití komponent doposud neposkytlo tolik zkušeností, aby mohly být přetaveny v jednotnou definici.

Pohled na komponenty se také poněkud liší v akademickém světě, kde je komponenta vnímána jako malý funkční celek, který je předmětem kompozice a má striktně skrytou implementaci . Oproti tomu v praktickém prostředí se připouští i rozsáhlé komponenty a zároveň není vždy pedantsky dodržováno pravidlo čistého black-box modelu [5].

I přes tyto potíže lze v literatuře nalézt hned několik definicí, jednu z nich formuloval Bachman [1]. Komponenta je:

- skrytá/neveřejná implementace funkcionality,
- je používána třetí stranou (*third-party*),
- odpovídá komponentovému modelu.

2.2.2 Rozhraní

Rozhraní je pro komponenty v podstatě ovladač, Crnkovic [5] je nazývá přístupovým bodem ke komponentě, potažmo ke službám, které komponenta poskytuje. Všichni

klienti komponenty, kteří chtějí využívat její služby, musí přistupovat přes toto rozhraní. Pokud komponenta poskytuje více přístupových bodů, respektive poskytuje více druhů služeb, měly by být jako takové rozděleny do více rozhraní.

Důležitou vlastností rozhraní je, že neposkytují implementaci. Poskytují pouze popis funkčnosti, kterou komponenta poskytuje a to jen na úrovni výčtu operací a jejich argumentů. Tato vlastnost umožňuje [5]:

- nahrazení implementace funkčnosti bez nutnosti kompilovat celý systém (nezmění se rozhraní),
- přidávání dalších rozhraní opět bez nutnosti kompilace systému.

Popis funkčnosti pomocí rozhraní však nemusí být vždy dostačující a nemůže zcela popsat chování komponenty v systému [5]. Tento nedostatek se řeší pomocí *mimo funkčních charakteristik*, které umožňují deklarativní cestou specifikovat další vlastnosti komponenty a dále poskytují mechanismus, jak na tento popis reagovat při běhu systému.

2.2.3 Komponentový model a framework

Pojmy komponentový model a komponentový framework jsou leckdy považovány za synonyma, nicméně v CBSE je zaveden konsensus, že *komponentový model* specifikuje standardy a konvence směrem k vývojářům komponent, oproti tomu *komponentový framework* je v podstatě implementace komponentového modelu. Poskytuje běhové prostředí a podpůrné služby tak, aby byl splněn (vynucen) komponentový model.

2.3 Komponentové modely a frameworky

V počátcích CBD probíhal návrh i vývoj aplikačních rámců, poskytujících nezbytnou infrastrukturu pro komponenty, bez jakýchkoliv obecně uznávaných specifikací a norem. To logicky vedlo k velké variabilitě frameworků, které byly často specifické pro produkty dodávané konkrétní firmou.

S postupem času a rozvojem této disciplíny se začalo pracovat na „standardizaci" komponentových modelů. V roce 1997 vznikla například specifikace EJB ve společnosti IBM, kterou později (1999) převzala společnost Sun Microsystems a k dnešnímu dni je k dispozici již 3. verze této specifikace - EJB3. V roce 2000 vyšla první specifikace OSGi[6] pod názvem OSGi Service Gateway Specification, která se zaměřovala zejména na zařízení s omezenou pamětí a výpočetním výkonem (například soudobé mobilní

[6] *Open Services Gateway Initiative*

telefony, Set Top boxy či modemy). Později, konkrétně v roce 2001, pro zjevný úspěch první verze, vyšla druhá specifikace s novým pojmenováním „OSGi Service Platform" a společně se změnou názvu rozšířila i pole působnosti z embedded zařízení na plošné použití všude tam, kde se používá Java. Dnes OSGi Alliance[7] poskytuje již 4. verzi specifikace.

Následující textu se zaměřuje na výše uvedené komponentové modely a krátce je popisuje. Poslední podsekce této kapitoly popisuje komponentový model CoSi.

2.3.1 EJB

Tato podsekce čerpá zejména z knihy [4], která je výborným zdrojem informací o EJB3. Je předkládán pouze zkrácený úvod do problematiky. Další informace jsou k dohledání v uvedené knize.

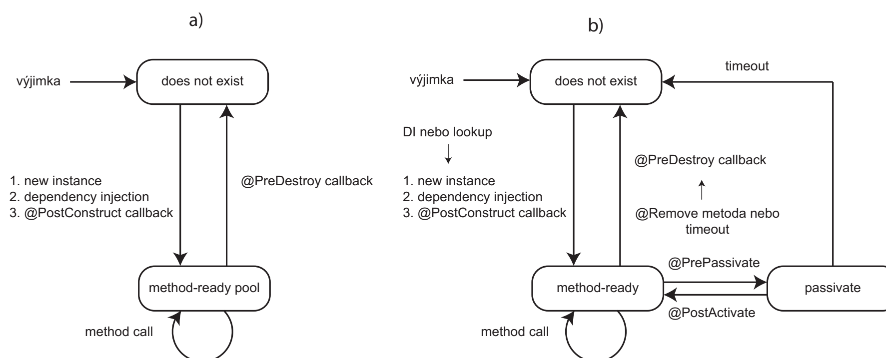
Specifikace EJB, tedy *Enterprise Java Beans*, spadá pod platformu Java EE[8]. Jave EE (někdy označovaná jako J2EE - Java 2 Enterprise Edition) je platforma postavená nad Java SE (Java Standard Edition), poskytuje API a běhové prostředí pro vývoj a provoz velkých, vícevrstvých, škálovatelných, spolehlivých a bezpečných síťových aplikací [4]. Svými vlastnostmi určena zejména pro podnikové systémy. Java EE se skládá z následujících vrstev:

- klientská vrstva
- webová vrstva - v rámci této vrstvy poskytuje například tyto technologie: Java EE servlety, JavaServer Faces (JSF), JavaServer Pages (JSP),
- business vrstva - poskytované technologie jsou například: Enterprise Java Beans (EJB), Java Persistence API entity
- informační vrstva - tato vrstva zahrnuje přístup k různým zdrojům, například databázím, poskytuje například tyto technologie: Java Database Connectivity API (JDBC), Java Persistence API (JPA)

EJB jsou serverové komponenty umožňující tvorbu modulární business vrstvy pro podnikové aplikace. EJB si klade za cíl oddělit obchodní logiku od ostatních vrstev (prezentační, perzistenční). Enterprise beany vycházejí z vlastností klasických Java Beans, což jsou dle definice klasické Java třídy, které dodržují určité konvence a mají za úkol sdružit více objektů do jednoho přes *settery* a *gettery.* Specifikace definuje dva základní typy bean:

[7] OSGi bylo založeno v 99 roce s cílem vytvořit otevřené specifikace pro poskytování služeb v distribuovaném prostředí.

[8] *Java Enterprise Edition*



Obrázek 2.2: Životní cyklus a) stateless beany, b) statefull beany

- session beans, dále dělené na
  - stateless session beans,
  - stateful session beana,
- message driven beans.

EJB ve verzi 3 dále definuje i pojem Entity, což je datová obálka předepsaná specifikací JPA (v EJB 2 jsou EntityBean). Ke komponentám (beanům) se přistupuje pomocí rozhraní, které musí každá beana implementovat, tato rozhraní mohou být bud' lokální (*local*) nebo vzdálená (*remote*). Takové dělení umožňuje definovat úroveň přístupnosti k těmto komponentám. Metody bean implementující lokální rozhraní jsou přístupné pouze lokálně, tedy z kontextu EJB kontejneru. Oproti tomu metody bean implementující předpis vzdáleného rohraní jsou přes RMI přístupné prakticky odkudkoliv.

EJB kontejner představuje virtuální prostor v aplikačním serveru, který umožňuje nasazování a běh bean. Spravuje mimo jiné i jejich životní cyklus.

Životní cyklus stateless session beany

Bezstavové beany (stateless) neuchovávají svůj stav mezi obsluhou jednotlivých požadavků. Pro obsloužení požadavku je klientovi přidělena jedna instance beany, nad kterou může volat operace definované jejím rozhraním. Tato instance je v průběhu jednoho požadavku přidělena výhradě jemu, tudíž je bezpečná vůči paralelním hazardům. Bezstavové beany umožňují používání vnitřních atributů, ale není zaručeno, že při dalším požadavku budou data v nich uložená nezměněna (odsud pochází název *bezstavové*). Vzhledem k této vlastnosti není potřeba aby EJB kontejner vytvářel nové instance pro každý požadavek, proto se tyto beany sdružují v poolu, ze kterého jsou dle požadavků odebírány a po dokončení požadavku opět vraceny zpět.

Životní cyklus bezstavové beany je zachycen diagramem na obrázku 2.2 a). Při spouštění si EJB kontejner dle svého nastavení vytvoří sadu instancí od každé bezstavové beany. Vytváření instance probíhá tak, že kontejner nejdříve vytvoří instanci bezparametrickým konstruktorem, poté do beany vloží všechny její závislosti a následně zavolá metodu anotovanou `@PostConstruct`, pokud existuje - to umožňuje inicializovat vše potřebné knihovny (například otevřít JDBC připojení). Ve chvíli, kdy je instance beany ve stavu *method-ready* může být vyzvednuta z poolu a obsluhovat klientské požadavky. Pokud se kontejner rozhodne zmenšit pool (například z důvodu nedostatku paměti) nebo je ukončován, volají se případné metody s anotací `@PreDestroy`, které slouží jako inverzní operace k `@PostConstruct` metodám (například uzavření JDBC spojení). V případě, že bezstavový bean během svých operací vyhodí výjimku, je odstraněn z poolu bez volání metody `@PreDestroy`.

Životní cyklus stateful session beany

Stavové beny (stateful) jsou oproti bezstavovým komplikovanější. Jsou to objekty, které si uchovávají svůj stav mezi jednotlivými požadavky klienta v rámci sezení (session). Pro zajištění takové funkčnosti musí kontejner pro každé unikátní sezení klienta vytvářet novou instanci beany, která si zachová svůj vnitřní stav po dobu celého sezení. Tento přístup je náchylný na paměťovou náročnost, proto se kontejner může na základě vnitřní logiky rozhodnout, že beanu uloží na určené úložiště (pasivuje) a v případě potřeby ji opět obnoví (aktivuje). Z tohoto důvodu je nutné, aby stavové beany byly serializovatelné. Stavové beany se označují anotací `@Stateful`.

Životní cyklus tohoto typu beany je zachycen na obrázku 2.2 b). Kontejner zakládá instanci beany v případě, že je požadována (lookup) nebo je předmětem závislosti jiné beany. Proces založení nové instance je obdobný jako u bezstavové beany. Nejprve se zavolá bezparametrický konstruktor, poté se vloží závislosti a nakonec se zavolá (existuje-li) metoda anotovaná `@PostConstruct`. Beana může zakončit svou existenci vyhozením výjimky (není volána metoda `@PreDestroy`), zavoláním metody uvozené anotací `@Remove` nebo vypršením timeoutu. V případě uvolňování zdrojů nastává takzvaná pasivace, kterou předchází případné volání metody anotované `@PrePassivate`, při aktivaci (inverze k pasivaci) se v případě její existence volá metoda anotovaná `@PostActivate`.

Ukázky deklarací EJB

Deklarace veřejného a lokálního rozhraní

```
1  @javax.ejb.Remote
2  public interface RemoteService { /*...*/ }
```

```
3 @javax.ejb.Local
4 public interface LocalService { /*...*/ }
```

Deklarace stavového beany s lokálním a vzdáleným rozhraním:

```
1 @javax.ejb.Stateful
2 class Service implements LocalService, RemoteService {
3   // implementace metod rozhrani
4 }
```

Deklarace bezstavového beany se stejnými rozhraními:

```
1 @javax.ejb.Stateless
2 class Service implements LocalService, RemoteService {
3   // implementace metod rozhrani
4 }
```

2.3.2 OSGi Service Platform

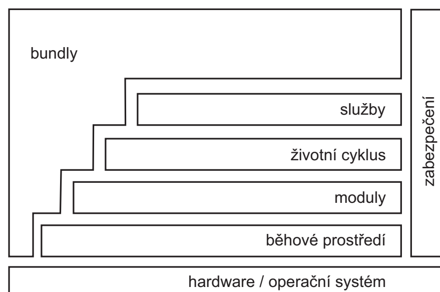
Tato sekce čerpá zejména z internetových stránek OSGi Alliance [17] a ze specifikace OSGi Service Platform Core specification [16]. Poskytuje pouze zkrácený přehled této architektury, více informací je možné nalézt v uvedených zdrojích [14, 16, 17].

OSGi Service Platform je komponentový model specifikovaný organizací OSGi Alliance. Jedná se o neziskovou organizaci založenou roku 1999. Mezi členy této organizace patří vedoucí poskytovatelé obsahu a služeb, sítí a infrastruktur, dodavatelé podnikových systémů, softwaroví vývojáři, výrobci spotřební elektroniky a výzkumné instituce [14]. Historie první verze specifikace sahá do roku 99, kdy byla vydána. Původně specifikace cílila zejména na zařízení s omezenou pamětí a výpočetním výkonem, další verze se už zaměřovaly na celou šířku záběru platformálně nezávislého jazyka Java. Původní záběr se však dodnes odráží v kompaktnosti specifikace (implementace Knopflerfish má pouhých 250KB [24]). OSGi Alliance poskytuje specifikace, referenční implementace, testovací soupravy a certifikace pro implementace jejich specifikací.

OSGi Service Platform je konfigurovatelnou platformou vybudovanou nad JVM[9]. OSGi Framework (v této sekci již jen Framework) tvoří jádro této platformy. Poskytuje univerzální, bezpečný a řízený Java aplikační rámec, který umožňuje nasazování rozšiřitelných aplikací zvaných *bundles*. Framework umožňuje stahování, instalaci bundlů a jejich odebírání v okamžiku, kdy již nejsou potřeba. Zároveň řídí proces této instalace, odinstalace či aktualizace v běhovém prostředí dynamickým a škálovatelným způsobem. Aby toho byl schopen, spravuje statické i dynamické závislosti mezi bundles a mezi službami. Funkcionalita Frameworku se dá rozdělit do několika vrstev [16]:

- vrstva zabezpečení,

[9] Java Virtual Machine



Obrázek 2.3: vrstvy OSGi frameworku, zdroj [16]

- vrstva modulů,
- vrstva životního cyklu,
- vrstva služeb.

Schématické zobrazení těchto vrstev zachycuje obrázek 2.3.

Běhové prostředí

Je libovolná konfigurace JRE. OSGi cílí i na zařízení s omezeným výpočetním výkonem, tudíž mezi platné běhové prostředí patří například i různé profily Java ME jako: MIDP[10] či CDC[11].

Vrstva zabezpečení

Vrstva je založená na Java Security Architecture a její použití není povinné. Jak je vidět z obrázku 2.3, jde napříč všemi dalšími vrstvami a poskytuje mechanismus, jak zabezpečit interakci s Frameworkem, popřípadě jak definovat vlastní zabezpečení na úrovni služeb.

Autentizace bundlu se provádí na bázi lokality nebo certifikace. Zevrubné pojednání o této vrstvě lze najít ve 2 kapitole specifikace [16].

Vrstva modulů

Vrstva modulů definuje modulární model pro běhové prostředí Frameworku, tedy pro Javu. Řeší některé z nedostatků modelu nasazení v Javě a definuje přísná pravidla pro sdílení (potažmo skrývání) Java balíčků (packages) mezi bundly.

[10]Mobile Information Device Profile
[11]The Connected Device Configuration

Bundle Ve Frameworku je jedinou nasaditelnou komponentou *bundle*, čímž se stává základní jednotkou modularity. Bundle je nasazován v podobě souboru typu Java ARchive (JAR), což je klasický ZIP archív, který obsahuje aplikace a jejich zdroje. Bundly sdílejí stejnou koncovku jako klasické java aplikace (.jar), nicméně existuje speciální MIME typ rezervovaný pro OSGi bundly, tento MIME typ je:
`application/vnd.osgi.bundle`.
Dle specifikace [16] je bundle JAR soubor, který obsahuje:

- Zdroje nezbytné pro poskytování nějaké funkcionality. Tyto zdroje mohou být soubory tříd Java, nebo také další data jako například HTML soubory, nápověda, ikony, obrázky a tak dále. JAR soubor bundlu může také obsahovat další JAR soubory, tento mechanismus však platí jen pro dvě úrovně zanoření, není tedy rekurzivní.

- Soubor s *manifestem* popisující obsah JAR souboru a poskytující informace o bundlu. Tento soubor používá hlavičky ke specifikaci informací, které Framework potřebuje k instalaci a aktivaci bundlu. Manifest například definuje závislosti bundlu, exporty bundlu či aktivační třídu bundlu.

- Dokumentaci v OSGI-OPT složce nebo v jejích podsložkách. Tato dokumentace je volitelná.

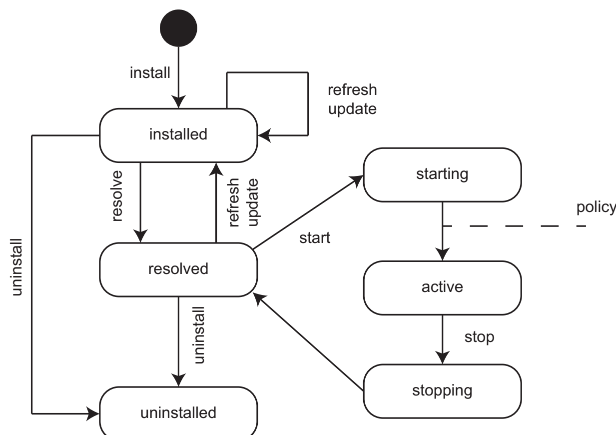
Hlavičky metadat bundlu Framework definuje poměrně velké množství typů hlaviček, které vývojář může použít pro definici informací o bundlu. Kompletní popis hlaviček je k dohledání v specifikaci OSGi Service Platform v. 4 [16].

Architektura zavádění tříd v bundlu Mnoho bundlů může sdílet jediný virtuální stroj (VM). V tomto VM bundly mohou skrývat své Java balíky a třídy před ostatními bundly, stejně jako je mohou s ostatními bundly sdílet. Klíčový mechanizmus pro skrývání a sdílení Java balíků je Java zavaděč tříd (Java class loader), který načítá třídy a zdroje z podmnožiny delegovaných class loaderů dalších bundlů. Každý bundle má svůj vlastní class loader, který delegačním přístupem může načítat třídy a zdroje z jiných balíků.

Resolving a wiring (spojování závislostí bundlu) je obšírně popsán ve specifikaci [16] v kapitole 3.

Vrstva životního cyklu

Vrstva životního cyklu poskytuje aplikační rozhraní (API) pro řízení a pro přístup k operacím nad životním cyklem Frameworku a bundlů. Následující text se okrajově



Obrázek 2.4: Životní cyklus bundlu v OSGi, převzato z [16]

zabývá životním cyklem bundlů, další a kompletní informace lze hledat ve specifikaci [16] konkrétně pak v kapitole 4.

Bundle se při běhu Frameworku nachází vždy v jednom z definovaných stavů. Stavový diagram životního cyklu bundlu zobrazuje obrázek 2.6. Následuje popis jednotlivých stavů bundlu:

- INSTALLED - Je úspěšně nainstalován do systému a všechny jeho náležitosti byly splněny (metadata a povinné hlavičky)
- RESOLVED - Všechny závislosti, které potřebuje jsou k dispozici. Tento stav značí, že je bundle připraven k nastartování. V tomto stavu se může vyskytovat i pokud byl v minulosti zastaven.
- STARTING - Bundle právě startuje, je volána metoda `start` na `BundleActivator`u a prozatím nebylo volání dokončeno. Pokud je nastaven na *lazy* aktivaci, zůstane v tomto stavu dokud není aktivován.
- ACTIVE - Bundle byl úspěšně aktivován a běží. To znamená, že metoda `start` `BundleActivator`u byla zavolána a její volání skončilo.
- STOPPING - Bundle se právě ukončuje. Metoda `stop` u `BundleActivator`u byla zavolána a čeká se na její dokončení.
- UNINSTALLED - Bundle byl odinstalován. Jedná se o konečný stav, to znamená, že již nemůže přejít do žádného jiného stavu.

Vrstva služeb

Registr služeb je důležitou částí Frameworku jelikož umožňuje bundlům interagovat mezi sebou a to, v duchu celého Frameworku, velmi dynamicky. Skrze tento re-

gistr služeb Framework umožňuje zaregistrovat jeden nebo více přístupových bodů k bundlům skrze služby. Každá služba je registrována přes jedno nebo více rozhraní, které implementuje.

Zajímavou funkčností registru služeb je možnost vyhledávání služeb přes atributy, společně s kterými byl zaregistrován, pomocí LDAP[12]filtrů.

2.3.3 CoSi

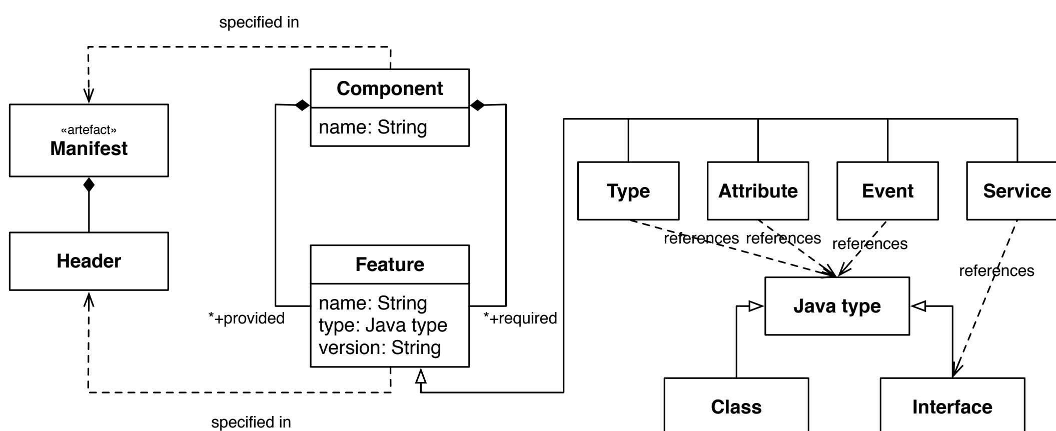
Cílem této práce je rozšířit komponentový aplikační rámec CoSi (dále jen Framework) o podporu Aspektově orientovaného programování a také o podporu deklarativního vstřikování závislostí na úrovni služeb. Na základě návrhu je potřeba rozhodnout, zdali se pro taková rozšíření bude měnit i komponentový model CoSi (specifikaci) či nikoliv. Jedním z nezbytných podkladů pro toto rozhodnutí je dobrá znalost aktuálního stavu specifikace (aktuálně se nachází ve verzi 2.0, [3]). Tato sekce podává krátký úvod do komponentového modelu CoSi. Konkrétní části specifikace, které budou předmětem případných úprav, bude práce zmiňovat v implementační části.

Komponentový model a Framework CoSi byl navržen a implementován roku 2007 na Západočeské univerzitě v Plzni na katedře informatiky a výpočetní techniky. Jeho zjevnou předlouhou je výše zmíněný komponentový model OSGi.

Specifikace komponentového modelu CoSi si hned ve svém úvodu stanovuje základní požadavky:

- **Plochý komponentový model** - kompozice komponent není ve Frameworku povolena. Všechny komponenty komunikují na stejné úrovni a mají stejný přístup ke kontejneru (vice versa).
- **Čistý black-box model** - vše co komponenta poskytuje svému okolí je specifikováno mimo implementaci komponenty.
- **Možnost poskytovat více služeb se stejným rozhraním** - Framework musí umožnit registraci více služeb stejného typu a zároveň musí poskytnout mechanismus, jak je odlišit.
- **Jeden proces** - kompletní běhové prostředí, ve kterém běží jak, Framework tak komponenty, je spuštěno v rámci jednoho procesu a v rámci jednoho VM. Ve Frameworku neexistuje distribuované volání.
- **Kombinace Java a Groovy** - Framework je napsán v programovacím jazyce Java. Komponenty mohou být psané jak v programovacím jazyce Java, tak i v Groovy.

[12]Lightweight Directory Access Protocol



Obrázek 2.5: Meta model CoSi (bez mimo-funkčních charakteristik), převzato z [3]

Komponentový model CoSi

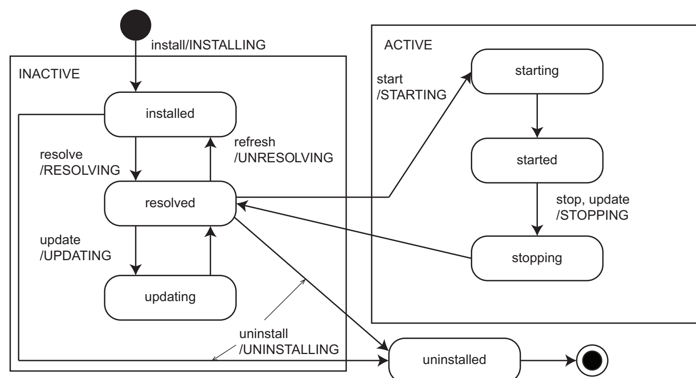
Komponentový model CoSi obsahuje entity, které jsou zobrazené na obrázku 2.5. Komponenta obsahuje název, poskytovatele a jeho verzi (na obrázku je zobrazen pouze název), tyto vlastnosti jednoznačně identifikují komponentu. Dále meta model definuje čtyři základní typy vlastností (v názvosloví CoSi *features*), které mohou být komponentou vyžadovány potažmo poskytovány. Každá vlastnost je identifikována jménem (kde je to potřeba), typem, popřípadě dalšími atributy (verze například).

Služba Služba je implementace funkcionality, funkcionalita je definována rozhraním. Poskytované služby jsou registrovány kontejnerem do registru služeb. Registr služeb poskytuje mechanismus pro vyhledávání a vázání služeb na komponenty, které službu vyžadují.

Typ Typ odkazuje na třídu či rozhraní. Poskytované typy jsou exportovány jejich výčtem v manifestu komponenty a importovány jsou pomocí delegačního zavaděče tříd.

Událost Události poskytují mechanismus, jak mezi sebou mohou komponenty komunikovat. Události jsou identifikovány jménem a typem, což umožňuje komponentám deklarovat o jaký typ události (potažmo o jak pojmenovanou zprávu) mají zájem popřípadě jaký typ události evokují.

Atribut Atributy definují typy hodnot, které mohou být komponentou nastavovány nebo čteny. Atributy jsou zpřístupněny pomocí registru atributů. Atribut je pár klíč-hodnota (*String*, *Object*).



Obrázek 2.6: Životní cyklus bundlu v CoSi, převzato z [3]

Bundle Základní a jedinou jednotkou modularity je, stejně jako u OSGi, JAR archív, který dodržuje definovanou strukturu. V názvosloví CoSi se komponenta (tedy i JAR archív) nazývá *bundle* (dále v textu již bude užito pouze tento název). Struktura a vlastnosti bundlu jsou následující:

- Bundle je obyčejný JAR archív, který obsahuje *manifest* deklarující nezbytné informace pro Framework. Adresářová struktura uvnitř JAR archívu obsahuje několik povinných adresářů*: bin, lib* a *META-INF.*
- Manifest bundlu deklaruje veškeré závislosti na výše uvedených vlastnostech, stejně tak deklaruje veškeré vlastnosti, které do systému poskytuje.
- Manifest deklaruje aktivační třídu bundlu, kterou je zajištěno spuštění a vypnutí bundlu.

Meta model nově od verze specifikace 2 umožňuje definovat i mimo-funkční charakteristiky.

Aktivátor bundlu V CoSi je každý bundle považován za spustitelný, tudíž je deklarace aktivátoru bundlu povinná.

Životní cyklus bundlu

Životní cyklus bundlu v CoSi je rozšířený životní cyklus bundlu z OSGi. Životní cyklus obsahuje několik stavů, ve kterých se bundle může vyskytovat, a také několik událostí, které jsou běhovým kontejnerem vystřelovány při přechodu mezi jednotlivými stavy. Celý proces názorně zobrazuje obrázek 2.6. Rozeznáváme následující stavy:

- INSTALLED - bundle byl úspěšně nainstalován. Znamená to, že po formální stránce splňuje všechny náležitosti požadované kontejnerem.

- RESOLVED - bundle je úspěšně vyhodnocen. To znamená, že všechny jeho závislosti jsou uspokojeny. Tento stav je synonymem ke stavu STOPPED v OSGi. Do tohoto stavu se bundle přesouvá také po zastavení.
- UPDATING - bundle je právě aktualizován. Jedná se rozšiřující stav životního cyklu.
- STARTING - bundle právě startuje. Byla zavolána metoda `start` na rozhraní `BundleControl` a čeká se na její návratovou hodnotu.
- STARTED - bundle byl úspěšně spuštěn a je v běhu.
- STOPPING - bundle právě zastavuje svůj běh. Byla zavolána metoda `stop` na rozhraní `BundleControl` a čeká se na její návrat.
- UNINSTALLED - bundle je odebrán ze systému. Jedná se o konečný stav, tudíž bundle již nemůže přejít do žádného jiného stavu.

Běhový kontejner

Běhový kontejner je v CoSi, mimo jiné, zodpovědný za:

- načítání bundlů a údržbu jejich životního cyklu,
- registraci služeb,
- poskytování informací o bundlech (tj. o jejich stavu, závislostech, atd.),
- poskytování služeb (i systémových).

Poskytuje nezbytnou infrastrukturu pro komunikaci a běh komponent. Pokud kupříkladu bundle A poskytuje atributy, typy, služby nebo zprávy a v systému existuje bundle B, které tyto vlastnosti legálně importuje, poskytne kontejner veškeré mechanismy, které zajistí import těchto závislostí.

Architektura zavádění tříd

Architektura zavádění tříd je obdobná jako u OSGi, přesto je poněkud zjednodušená. Ono zjednodušení spočívá zejména v tom, že CoSi předpisuje běh bundlů i kontejneru pouze a jen v jednom virtuálním stroji a v jednom vlákně. Každý bundle má svůj vlastní zavaděč tříd, který delegačním mechanismem umožňuje načítání tříd a typů, které bundle požaduje. Zároveň tento zavaděč umožňuje načítání tříd a zdrojových kódů napsaných v dynamickém skriptovacím jazyce Groovy. Kompletní popis architektury zavádění tříd je k dispozici ve specifikaci CoSi [3].

Rozdíly oproti OSGi

Výše uvedený popis komponentového modelu CoSi napovídá, že vychází ze specifikace OSGi. Přesto se CoSi od OSGi liší a to zejména proto, že CoSi je mnohem přímočařejší model. Pokud bychom na jeden konec pomyslné osy položili klasický kód psaný například v Javě, kde všechny závislosti a propojení jsou definovány čistě staticky na úrovni programového kódu. A na druhou stranu této osy potom umístili naprosto dynamický komponentový model OSGi, tak CoSi je přibližně uprostřed mezi těmito body. Z tohoto úhlu pohledu logicky plynou následující rozdíly:

- *Exportování a importování typů (balíků)* - v CoSi se musejí deklarovat exportované a importované typy výčtem jednotlivých položek. Oproti tomu v OSGi se exportují pouze balíky. Tento rozdíl plyne zejména z jednodušší architektury načítání tříd. Ve verzi 2 CoSi umožňuje import i export celých balíků, přesto však upozorňuje na to, že takový export/import může způsobit nestabilitu běžících bundlů.

- *Bezpečnost* - CoSi si klade jako z jednu prvních podmínek udržet tento komponentový model maximálně jednoduchý, proto v něm není žádná podpora zabezpečení.

- *Groovy* - CoSi, díky své architektuře načítání tříd, umožňuje načítání zdrojových kódů (tříd) psaných v dynamickém skriptovacím jazyce Groovy.

- *Nativní knihovny a kódy* - CoSi (oproti OSGi) neumožňuje práci s nativním knihovnami (.dll, .so). Tento rozdíl opět plyne z předpokladu jednoduchosti CoSi.

- *Běhové prostředí* - CoSi neumožňuje definovat na úrovni bundlu požadované běhové prostředí. Každý bundle běží v takovém prostředí, ve kterém je spuštěn kontejner.

- *Soubor s manifestem a struktura JAR archívu* - Struktura archívu je téměř totožná, nicméně kontejner klade požadavky na povinné adresáře *bin* a *lib.* Umístění a podoba manifestu bundlu je shodná jako u OSGi, jednotlivé hlavičky jsou však odlišné. Popis jednotlivých hlaviček je uveden ve specifikaci [3].

- *Perzistentní stav kontejneru* - CoSi neuchová perzistentní stav kontejneru po jeho zastavení.

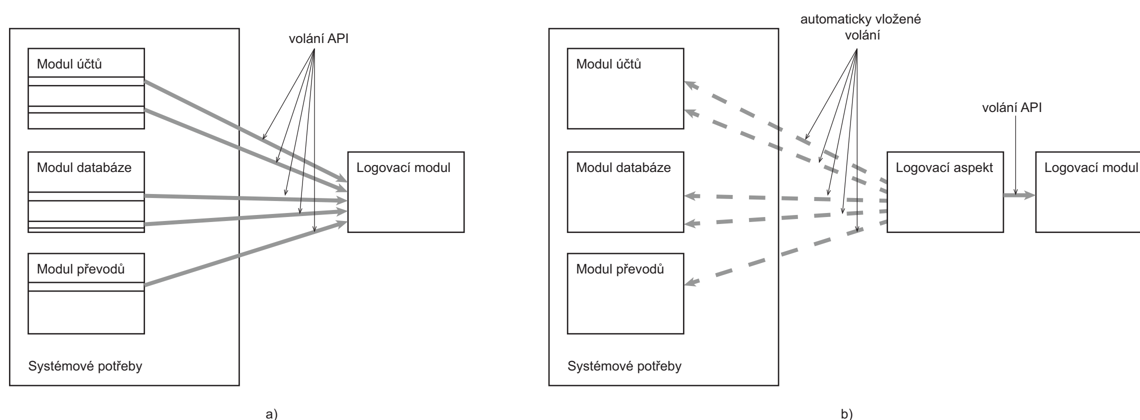
- *Povinnost implementovat aktivátor bundlu* - CoSi předpokládá, že každá komponenta je spustitelná, tudíž implementace aktivátoru je v CoSi povinná.

Kapitola 3

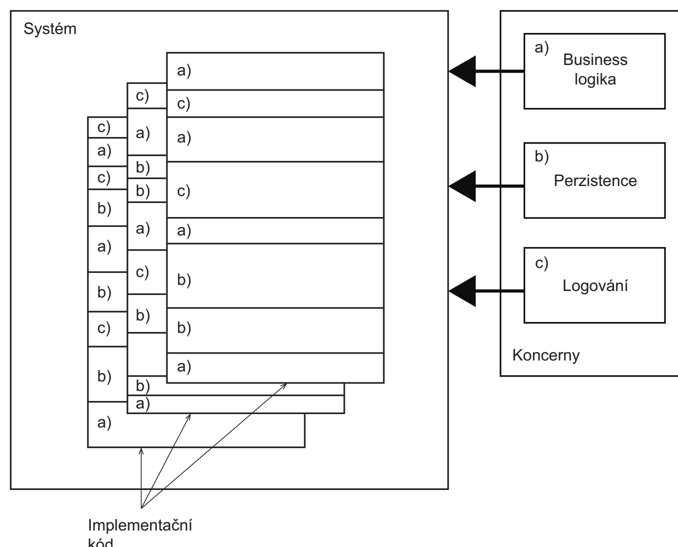
Aspektově orientované programování

Ve všech aplikacích, i přes jejich odlišnou výslednou funkci existují ,1stejné či velmi podobné funkcionality. Jsou jimi například systém logování, transakce, persistence a další. Právě tyto funkčnosti, které se stále opakují a jsou předmětem každého softwarového celku, se za použití tradičních přístupů nedají vyčlenit tak, aby se nevyskytovaly napříč celou aplikací. Aspektově orientované programování (dále jen) poskytuje metodologický přístup, který umožňuje tyto funkcionality vyčlenit do samostatných celků.

Následující kapitola poskytuje úvod do AOP, podává výklad základních termínů a ve své poslední části představuje ukázkové implementace pro programovací jazyk Java.



Obrázek 3.1: a) Systém bez AOP b) systém s AOP (převzato z [11]).



Obrázek 3.2: Kompozice výsledného systému z koncernů, převzato z [11] .

3.1 Motivace a základní konstrukty

Jaký problém tedy vlastně AOP řeší? Obrázek 3.1 a) ukazuje příklad bankovní aplikace obsahující 3 systémové potřeby[13] a 1 protínající potřebu. Užití protínajících potřeb se provádí voláním API v potřebných místech. Tento přístup porušuje myšlenku modularity a také poučku DRY[14] tím, že nutí vývojáře do metod vkládat stále dokola kód, který s danou metodou má jen pramálo společného (v tomto případě logování). AOP, jak ukazuje ilustrace 3.1 b), zavádí novou jednotku modularity - *aspekt*. Pomocí tohoto aspektu je opakující se kód odstraněn a tím je zároveň odstraněno i porušení modularity, neboť volání API se přesouvá do aspektu.

Aby byla protínající potřeba vložena na své místo, zavádí AOP několik pojmů, které vysvětluje následující text.

3.1.1 Systémové a protínající potřeby

Jedná se o potřeby aplikace, které je třeba řešit v zájmu splnění celkového cíle softwarového sytému. Softwarový systém je sousledné navázání jednotlivých potřeb. Například bankovní systém je množina následujících potřeb: management zákazníků a účtů, výpočty, mezibankovní transakce, bankomatové transakce a další. Potřeby se dělí na dva základní typy a to na:

- *systémové potřeby* - to jsou ty potřeby, které jsou spojeny s výslednou funkcí sys-

[13] vysvětlení pojmu *potřeba* a jejich dělení popisuje podsekce *Systémové a protínající potřeby* na této straně

[14] don't repeat yourself

tému a jsou pro ni zároveň specifickými. Zároveň se jedná o potřeby, které mají jasně vymezitelné pole působnosti a pomocí OOP přístupu je lze vyčlenit do samostatných celků. Příklady těchto potřeb jsou uvedeny ve výše nastíněném bankovním systému.

- *protínající potřeby* - jsou ty, které se zpravidla opakují ve všech aplikacích a hlavně protínají všechny ostatní potřeby. Jedná se například o bezpečnost aplikace, logování, transakční zpracování. Tyto potřeby a jejich protínání ukazuje obrázek 3.2.

3.1.2 Model přípojných bodů

Model přípojných bodů (anglicky Join Point Model - JPM) se skládá z několika pojmů:

- **Pokyn (*advice*)** - pokyn je programový kód, který má být vpleten do určitých přípojných bodů. Zároveň umožňuje volbu kdy se má vykonat, tedy jestli před, po nebo okolo přípojného bodu.
- **Přípojný bod (*joinpoint*)** - přípojné body jsou místa v běhu programu, do kterých mohou být vpleteny pokyny. Aby bylo možné tyto body použít, musejí být adresovatelné a srozumitelné běžnému programátorovi. Zároveň by neměly být ovlivněny změnami v programu, to znamená, že by neměly být závislé na struktuře programu. Klasickým přípojným bodem je volání funkce či metody, nebo vytváření nové instance objektu a podobně.
- **Řez programem** (***pointcut***) - řezy programem vybírají určité přípojné body a jsou v podstatě předpisem, kam vplést určité pokyny. Pro zápis řezů programem se většinou užívá speciálního jazyka.
- **Vplétání (*weaving*)** - vplétání je proces, při kterém se vkládají pokyny do přípojných bodů specifikovaných řezem programu.
- **Aspekt (*aspect*)** - aspekt je jednotkou modularity v AOP. Jedná se o obálku pro pokyny a řezy programem. Aspekty by měly disponovat všemi prostředky rodičovského programovacího jazyka jako například: dědičnost, viditelnost vlastností a další.

Tato, dnes již standardní, terminologie vzešla z termínů používaných v AspectJ. To je zapříčiněno dominantním postavením AspectJ na trhu AOP - termíny jako *joinpoint, pointcut, advice, weawing* či *aspect* se poprvé objevily právě v této knihovně. V současné době není ustálen překlad těchto termínů do českého jazyka, pro tuto práci byly použity překlady navržené v článku *„Aspektová terminologie"* [2].

3.2 Možnosti realizace v Javě

Již bylo popsáno co AOP řeší a také jaké termíny k tomu používá, otázkou nadále zůstává jak toho dosáhnout. Základní myšlenkou aspektově orientovaného programování je na určitá místa v běhu programu doplnit chování, které není specifické pro daný modul. Je tedy nutné do hotového kódu deklarativní cestou vkládat volání jiných částí aplikace - aspektů. Z pohledu implementace jde o dosti netriviální záležitost. Pro demonstraci je uveden následující kód:

```
1  class AccountService ...
2    // Metodu je potreba volat v transakci
3    public void transfer(Account from, Account to, Float amount)
4    throws Exception {
5      // Do logu je potreba vlozit, ze zaciname prevod
6      if (from.ballance >= amount) {
7        to.ballance += amount;
8        from.ballance -= amount;
9      } else {
10       // Vlozit do logu, ze se nepovedlo
11       throw new Exception("Malo␣penez␣na␣ucte");
12     }
13     // vlozit do logu, ze prevod byl dokoncen
14   }
```

Z komentářů v ukázce je patrné, že potřebujeme zajistit užití logování a transakcí. AOP má za úkol deklarativní cestou vyprodukovat následující kód (myšleno co do funkce, nikoliv podoby):

```
1  class AccountService ...
2    private Logger log;
3    private Database db;
4
5    public void transfer(Account from, Account to, Float amount)
6      throws Exception {
7      db.startTransaction();
8      log.info("Transfer␣begins");
9      if (from.ballance >= amount) {
10       to.ballance += amount;
11       from.ballance -= amount;
12     } else {
13       log.error("Tranfer␣ends␣with␣exception");
14       db.transactionRollback();
15       throw new Exception("Low␣ballance");
16     }
17     db.transactionCommit();
18     log.info("Transfer␣was␣ok");
19   }
```

Je zjevné, že kód uvedený v druhé ukázce přibližně odpovídá tomu, který by byl programátorem napsán bez užití AOP, je také zřejmé, jak se do kompetence metody na převod peněz implementuje i to, co s faktickým úkonem nemá nic společného a kód tak ztrácí na eleganci i přehlednosti. AOP musí tedy na základě deklarací provést

úpravy kódu a daná volání doplnit. Tomuto úkonu se říká *vplétání* (*weaving*), tento proces je realizován na základě informací, které poskytuje model přípojných bodů. Je možno ho implementovat třemi základními způsoby: při kompilaci, při načítání tříd do běhového prostředí a nebo při běhu aplikace.

3.2.1 Při kompilaci

Tento způsob vplétání je vložen do procesu kompilace kódu. Nejprve se provede vyhodnocení modelu přípojných bodů. V tomto vyhodnocení se načtou všechny řezy programem a zkompilují se pokyny do klasických java tříd. Poté se provede kompilace všech ostatních tříd. Konkrétně v Javě nám po kompilaci vznikne tak zvaný bytecode. Tento bytecode je následně procházen a jsou vyhodnocovány všechny přípojné body oproti všem načteným řezům programem. Pokud je právě vyhodnocovaný přípojný bod vybrán libovolným řezem, provede se vložení kódu, který volá daný pokyn z příslušného aspektu. Je nutné si uvědomit, že tento kód již musí být vložen již na úrovni bytecodu, proto se pracuje s knihovnami, které umožňují upravovat již zkompilované třídy.

Tento způsob je nejvýhodnější vzhledem k výkonu výsledné aplikace. Veškerá režie spojená s rozhodováním, zda aplikovat pokyn a s tím spojené zpracování modelu přípojných bodů se vykonává pouze jednou a to při kompilaci. To je však vykoupeno omezenou dynamičností. Není možné doplnit libovolné aspekty za běhu, není možné zkompilovat kód, pokud nemáme k dispozici všechny aspekty. V kontextu s komponentovými technologiemi je tento způsob sám o sobě v podstatě nepoužitelný, protože bychom mohli využívat aspekty pouze v jedné komponentě a zároveň by musely být k dané komponentě připojeny (při-kompilovány natvrdo).

3.2.2 Při načítání tříd

Jedná se o dynamický způsob vplétání, které je prováděno v okamžiku, kdy je daná třída zaváděna do běhového prostředí. V Javě se o zavádění tříd stará *zavaděč tříd* (*class loader*). Ten může být upraven tak, že po načtení třídy provede stejný proces jako je proveden v procesu vplétání při kompilaci. Na základě informací z modelu přípojných bodů provede úpravu bytecodu, aby volání jednotlivých pokynů byla vložena na své místo.

V porovnání s předchozím způsobem je tento méně výhodný z pohledu výkonové zátěže, nicméně poskytuje vyšší dynamičnost. Z pohledu komponentových technologií je poněkud problematický. Za prvé mají komponenty většinou vlastní zavaděče tříd a tento proces by se buď musel vložit přímo do jádra komponentového frameworku,

nebo by se musel navrhnout mechanismus, který by umožňoval úpravu bytecodu těsně před jeho zavedením do běhového prostředí. Za druhé je problém v tom, že jak jsou průběžně komponenty zaváděny do kontejneru, mohou si s sebou přinést i nové aspekty. Tyto aspekty je poté třeba vplést do již zavedených tříd. Bylo by potřeba vypnout a znovu zapnout (restartovat) všechny komponenty, aby jejich zavaděče tříd provedly nové vplétání.

3.2.3 Při běhu

Ze všech předchozích způsobů implementací vplétání je tento nejdynamičtější. Jeho funkce je založena na registru aspektů, proxy třídách a kontejneru, který spravuje všechny objekty podléhající procesu vplétání. Do kontejneru se registrují všechny instance objektů, při této registraci kontejner vytvoří proxy třídu, která naslouchá voláním všech metod cílového objektu. Při volání metody se vytvoří přípojný bod, který se konfrontuje s registrem aspektů (respektive s řezy programem v aspektech), pokud je tento přípojný bod vybrán alespoň jedním řezem, aplikují se pokyny, které jsou na něj navázány.

Je však patrné, že nespornou nevýhodou je vyhodnocování řezů programem při každém volání metody, což logicky vede k výkonovým nárokům. Tyto nároky se samozřejmě mohou projevit na výkonu výsledné komponentové aplikace. Na druhou stranu je tento přístup velmi dynamický a hodí se právě na komponentové systémy.

3.3 Implementace pro Javu

V následujícím textu budou popsány dvě nejznámější implementace AOP pro programovací jazyk java: AspectJ a Spring AOP.

AspectJ je aktuálně lídr na trhu AOP a i proto je Spring AOP svým způsobem reimplementace AspectJ, kde se využívá vplétání kódu za běhu (pomocí proxy tříd). K zápisu řezů programem pak používá AspectJ i SpringAOP stejný jazyk.

3.3.1 AspectJ

ApectJ je aspektově orientované rozšíření pro programovací jazyk Java. Kompilátor AspectJ produkuje soubory tříd, které jsou konformní k Java specifikaci byte kódu, což znamená, že mohou běžet na libovolném Java virtuálním stroji (JVM). Celé AspectJ včetně jeho konstruktů je postavené na jazyce Java. Čerpá tak veškeré benefity tohoto jazyka (dědičnost, datové typy, atp.) a zároveň má pro případné uživatele poměrně strmou učící křivku.

AspectJ obsahuje dvě základní části: specifikaci jazyka a implementaci jazyka AspectJ. Specifikace jazyka definuje jazyk, ve kterém jsou aspekty psány. Jak již bylo řečeno, většinou se jedná běžné Java konstrukty, které jsou doplněny o některé nové. Implementace jazyka pak poskytuje nástroje na výsledné vplétání kódu, ladění a také nástroje pro integraci AspectJ do několika integrovaných vývojových prostředí (IDE) [11].

Konstrukty AspectJ

ApectJ zavádí do programovacího jazyka Java nové konstrukty pro specifikaci pravidel vplétání. Veškeré konstrukty a rozšíření jsou navržena tak, aby běžný Java programátor neměl potíže s jejich používáním. V následujících odstavcích budou tyto konstrukty zběžně popsány na ukázkách kódu, pro bližší pochopení konstruktů je doporučena kniha [11], ze které je i čerpán následující popis, nebo přímo dokumentace k AspectJ [7].

Řezy programem Řezy programem (*pointcuts*) se v AspectJ definují speciálním výrazem, který je následně překládán a interpretován jako logický výraz. Následující ukázka kódu přibližuje syntax tohoto výrazu:

```
1 execution(void Account.credit(float))
```

Konkrétně tento řez programem vybírá všechny body spojení, které jsou voláním metody
`credit()` s jedním argumentem typu `float` na třídě `Account`.

Pokyn Pokyny (*advices*) se v AspectJ podobají obyčejným Java metodám. Jejich hlavička je však poněkud odlišná. Nejprve se deklaruje místo vložení pokynu (*before, after, around*) a následně, za dvojtečkou, řez programu. Dále již následuje samotné tělo pokynu. Ukázka pokynu:

```
1 before() : execution(void Account.credit(float)) {
2   System.out.println("Credit␣method␣will␣be␣performed");
3 }
```

Aspekt Aspekt (*aspect*) v AspectJ se logikcy podobá třídě - logicky proto, že se jedná o ekvivalent pro OOP (AOP - aspekt, OOP - objekt). Jeho předpis znázorňuje následující ukázka:

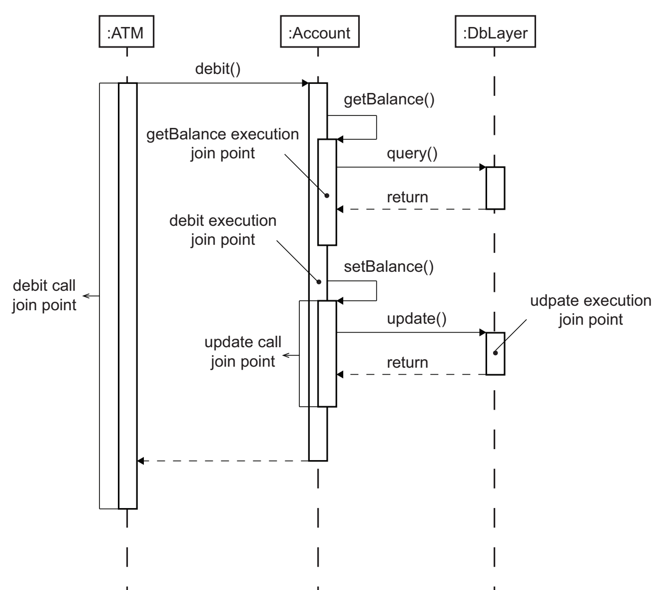
```
1 public aspect Example {
2   pointcut creditExecution() : execution(void Account.credit(float));
3
4   before() : creditExecution() {
5     System.out.println("Credit␣method␣will␣be␣performed");
```

```
6        }
7    }
```

V AspectJ může aspekt obsahovat vše, co může obsahovat normální Java třída, tedy vnitřní proměnné, metody i zanořené třídy.

Další konstrukty AspectJ navíc k uvedeným konstruktům definuje *introduction* a *compiler-time declaration*. Tyto konstrukty a jejich význam jsou v kontextu této práce nerelevantní, proto již nejsou zmiňovány. Více informací k těmto konstruktům lze nalézt přímo v dokumentaci AspectJ [7] popřípadě v knize [11].

Typy přípojných bodů



Obrázek 3.3: Sekvenční diagram znázorňující nejběžnější join pointy, převzato z [11].

V běhu programu může existovat nespočetné množství adresovatelných bodů jako *volání metody*, *přiřazení dat do proměnné*, *větvení programu*, *for*, *while*, *do/wile smyčka*, atd. V AspectJ však tyto přípojné body nejsou zpřístupněny všechny, na obrázku 3.3 jsou pomocí sekvenčního diagramu znázorněny dva nejčastější body, které jsou v následujícím textu letmo popsány. Je nutné uvést, že se nejedná o všechny, které AspectJ poskytuje. Pro kompletní výčet typů přípojných bodů je nutné nahlédnout do dokumentace k AspectJ [7].

Z diagramu 3.3 lze rozpoznat dva typy přípojných bodů: *call* a *execution*. Rozdíl mezi nimi je taktéž patrný z diagramu a týká se zejména kontextu, ze kterého je přípojný bod dosažen. Bod typu *call* se sice váže k třídě, která předepisuje danou metodu,

ale evokován je z kontextu volajícího kódu. Oproti tomu *execution* bod je evokován z kontextu volané metody.

3.3.2 Spring AOP

Spring AOP vychází z AspectJ. Nicméně se od AspectJ dramaticky liší v technice vplétání kódu. Zatímco AspectJ je rozšířením programovacího jazyka Java s vlastním kompilátorem, Spring AOP je celé postaveno na dynamickém vplétání při běhu programu. To je umožněno zejména díky Spring IoC[15] kontejneru, nad kterým je možno aplikovat AOP techniky (a to pouze nad ním).

Spring AOP operuje nad stejnými termíny jako AspectJ - jmenovitě: *aspekt*, *přípojný bod*, *řez programem* a *pokyn*. Navíc představuje dva pojmy, které plynou z podstaty dynamického vplétání:

- **Target object -** tento pojem představuje objekt, jenž byl upraven (propleten s) jedním či více aspekty. Vzhledem k principu funkce dynamického proplétání za běhu programu, bude tento objekt vždy schován za proxy objekt.

- **AOP proxy -** objekt, jenž je vytvořen Spring frameworkem za účelem skrytí původního objektů a odchytávání volání metod původního objektu. Tento přístup byl již představen na straně 24.

Princip vytváření proxy objektů může fungovat jen za předpokladu, že objekty budou instancovány nebo vraceny kontejnerem, nelze očekávat, že při vytvoření vlastní instance pomocí klíčového slova `new`, budou na tento objekt aplikovány aspekty, i přesto že řezy programem danou třídu popisují.

Model přípojných bodů je oproti AspectJ omezen a disponuje pouze jedním typem přípojných bodů, tím je typ *execution*. Toto omezení je taktéž logickým důsledkem zvolené techniky vplétání. Dále jsou i rozdílné způsoby definice jednotlivých prvků AOP. Protože Spring AOP není postaveno na vlastní kompilaci, nemůže definovat vlastní konstrukty, proto používá pro Javu nativních anotací. Následující fragment kódu ukazuje definici aspektu:

```
1  @Aspect
2  public Example {
3    @Pointcut("execution(void Account.credit(float))")
4    private void creditExecution();
5
6    @Before("creditExecution()")
7    public void creditExecution() {
8      System.out.println("Credit method will be performed");
```

[15] Inversion of Control

```
9       }
10  }
```

Pokud by vývojář neměl k dispozici Javu verze 5, kde byly anotace poprvé představeny, Spring AOP umožňuje deklarovat všechny tyto konstrukty na úrovni XML. Bližší informace k formátu definičního XML lze dohledat v dokumentaci [20].

3.4 AOP v komponentových modelech

V komponentových modelech, které jsou specifické svým dynamickým přidáváním komponent za běhu, je realizace AOP tímto faktem poněkud komplikována. Nelze totiž provést vplétání pouze jednou, protože nově nasazené komponenty mohou obsahovat další aspekty, které musejí být zohledněny a vpleteny na místa, které specifikují jejich řezy programem.

Pro implementační část práce je přínosné prozkoumat, jak je tento požadavek řešen ve známých komponentových modelech. Protože se CoSi velmi podobá OSGi, následující dvě podsekce popisují implementaci AOP v komponentovém modelu OSGi.

Prvně uvedená implementace - Equinox Aspects - je založena AspectJ a mechanismu, který se nazývá *hook*. Druhá implementace - Spring AOP v Spring DM - vychází z rozšíření OSGi o Spring kontejner, čímž je umožněno používat i AOP.

3.4.1 Equinox Aspects

Equinox Aspects je rozšíření OSGi frameorku Equinox, které je distribuované jako běžný bundle a umožňuje používání aspektově orientovaného programování v ostatních bundlech, které jsou do frameworku nasazeny. Toto rozšíření je postaveno na mechanismu, který se ve specifikaci OSGi nazývá hook. Ten umožňuje tomuto ovlivňovat načítané třídy na úrovni výsledného bytecode.

Celý proces lze zjednodušeně popsat následovně:

1. Při instalaci nového bundlu rozšíření zkontroluje, zda-li tento bundle obsahuje aspekty.

2. Pokud ano, posbírá veškeré nutné informace pro vplétání.

3. Provede tichý restart všech spuštěných bundlů

4. Jakmile jakýkoliv bundle požádá zavaděč tříd o třídu, je pomocí hooku spuštěn kód, který na základě posbíraných informací provede propletení a výsledek vrátí.

Zápis aspektů, jejich vplétání a všechny ostatní úlohy spojené s AOP přebírá AspectJ, který je tímto rozšířením obaleno. Rozšíření je děleno do několika částí a je stavěno jako adaptér pro libovolné knihovny, tudíž je možné AspectJ nahradit i jinou knihovnou.

Další informace o tomto rozšíření lze nalézt přímo v dokumentaci [6].

3.4.2 Spring DM a Spring AOP

Spring DM[16] zavádí do OSGi možnost využít sílu Spring IoC kontejneru. Ten mimo jiné poskytuje *AOP*.

Spring AOP může v komponentových modelech fungovat bez úprav, protože, jak již bylo uvedeno výše, funguje na bázi dynamického vplétání za běhu. Pro jednotlivé objekty se vytvářejí proxy objekty, které obstarávají vplétání. Tato vlastnost zaručí, že není potřeba bundly restartovat v okamžiku, kdy je zaveden nový bundle obsahující aspekty, protože všechny reference, které jsou vlastněny ostatními bundly, odkazují právě na proxy objekty. Je tedy pouze nutné posbírat nutné informace z nově nasazeného bundlu a nějakým způsobem je předat proxy objektu, který je zohlední.

Dokumentace k Spring DM je k dispozici v použité literatuře [21].

[16] Spring Dynamic Modules

Kapitola 4

Dependency Injection

Dependency Injection (dále jen DI) se řadí mezi návrhové vzory a stejně jako u ostatních návrhových vzorů, je jeho úkolem poskytnout efektivní, čisté a ozkoušené řešení častých *problémů* vyskytujících se při návrhu aplikací.

Zcela na úvod kapitola popisuje onen problém, který vedl ke vzniku tohoto návrhového vzoru a pokračuje vysvětlením, jednotlivých forem DI. Dále je zmíněno alternativní řešení problému, návrhový vzor *Service Locator*, a je popsán rozdíl tohoto řešení oproti DI. V závěru kapitoly jsou uvedeny ukázky frameworků, které DI implementují.

Text této části ve velké míře čerpá z článku Martina Fowlera [8], který je výborným zdrojem teoretických i praktických informací k DI.

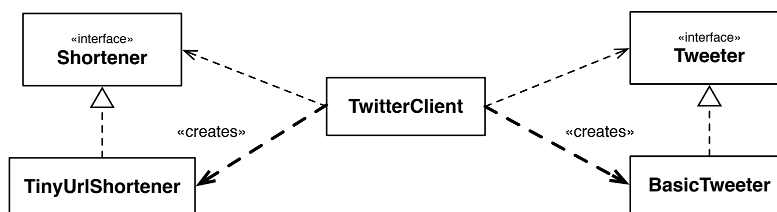
4.1 Motivace

Pro uvození do problematiky je vhodné uvést jednoduchý příklad nastiňující problém, který vzniká při návrhu objektově orientovaných aplikací. Příklad implementuje jednoduchého klienta pro službu Twitter[17]. Jeho stěžejní funkce, odeslání tweetu[18], je implementována jednou metodou:

```
1  class TwitterClient...
2      public void tweet(String text) {
3          if (text.length() > 140) {
4              text = shortener.shorten(text);
5          }
6          tweeter.send(text);
7      }
```

Tato implementace je velmi naivní a slouží pouze jako ukázková. Nejprve se zkontroluje, zda je text delší než 140 znaků, pokud ano použije se objekt `shortener` pro jeho zkrácení a poté se pomocí objektu `tweeter` provede samotné odeslání tweetu.

[17] Twitter je sociální internetová aplikace, je funkčnost se dá připodobnit k mikroblogu
[18] Tweet je příspěvek vložený do služby Twitter



Obrázek 4.1: Závislosti při vytváření instancí uvnitř objektu.

Stěžejním bodem v kontextu této kapitoly jsou objekty `tweeter` a `shortener`, respektive to, kde se vzaly instance těchto objektů uvnitř objektu `TwitterClient` a proč v návrhu figurují.

Je nežádoucí, aby objekt `TwitterClient` v sobě obsahoval implementace pro jednotlivé metodiky zkracování textu, či transportní záležitosti starající se o fyzické odeslání příspěvku. Oddělení této funkcionality do samostatných celků se provede pomocí dvou rozhraní:

```
1  interface Tweeter {
2    public void send(String tweet);
3  }
4
5  interface Shortener {
6    public String shorten(String text);
7  }
```

Nyní je vše správně odděleno, nicméně stále je potřeba v nějakém okamžiku do objektu `TwitterClient` vložit konkrétní implementace obou rozhraní. Doplněním konstruktoru objektu `TwitterClient` lze tyto konkrétní instance vytvořit:

```
1  class TwitterClient...
2      private Tweeter tweeter;
3      private Shortener shortener;
4      public TwitterClient() {
5          tweeter = new BasicTweeter("apiKey");
6          shortener = new TinyUrlShortener();
7      }
```

`BasicTweeter` odesílá příspěvky pomocí HTTP API poskytovaného službou Twitter, k čemuž je vyžadován tajný klíč. A `TinyUrlShortener` zkracuje text tak, že nejprve zkrátí všechny URL odkazy a pokud to nepomůže, ořízne text na maximální délku 140ti znaků.

Zásadní problém v tomto přístupu jsou vzniklé závislosti mezi třídami zobrazené na obrázku 4.1. Třída `TwitterClient` je závislá jak na obou rozhraních, tak i na konkrétních implementacích, což například znemožňuje snadno vyměnit tyto implementace za jiné i přesto, že jsou použita rozhraní. Správné by bylo, aby třída `TwitterClient` závisela pouze na rozhraních.

4.1.1 Definice pojmů

V kontextu DI a této kapitoly se vyskytují tři pojmy - *dependency*, *injection* a *assembler.* Tato podsekce všechny pojmy nadefinuje a poskytne překlady do českého jazyka, kterých se bude, společně s původními anglickými, ve zbytku kapitoly využívat.

Závislost (*dependency)*

V kontextu DI je závislostí myšlena závislost mezi třídami či rozhraními.

Pokud například třída pojmenovaná `ClassA` ve svém těle používá třídu `ClassB` a rozhraní `InterfaceA` říkáme, že třída `ClassA` je *závislá* na třídě `ClassB` a rozhraní `InterfaceA`.

Vkládání (*injection*)

Vkládání závislostí je proces, kterým jsou do závislých tříd vloženy instance objektů, které tyto závislosti implementují.

Závisí-li třída `ClassA` na rozhraní `InterfaceA` a třída `ClassB` toto rozhraní implementuje, pak je *vkládání* proces, při kterém je do objektu `ClassA` dopravena instance třídy `ClassB`.

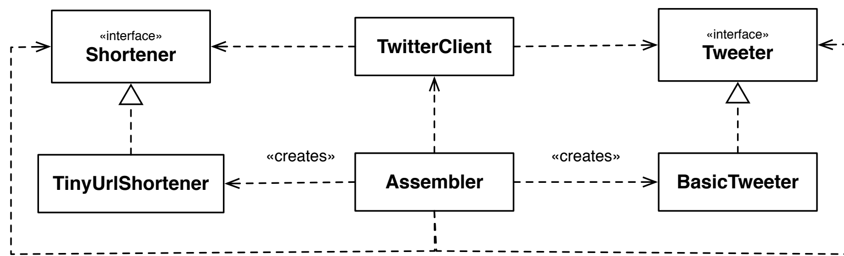
Konfigurátor závislostí (*assembler*)

Konfigurátor závislostí provádí samotné vkládání závislostí. A to ať už na základě informací načtených z konfiguračního souboru, pomocí programového API nebo staticky pomocí ručně psaného kódu.

Závisí-li třída `ClassA` na rozhraní `InterfaceA` a třída `ClassB` společně s třídou `ClassC` toto rozhraní implementují, pak *konfigurátor závislostí* rozhodne, pro kterou z tříd `ClassB` a `ClassC` se vyrobí instance a tu poté vloží do objektu `ClassA`.

4.1.2 Definice problému

DI řeší problém na tvrdo vytvořených závislostí ve třídách (vizte zvýrazněné šipky v obrázku 4.1), které vyžadují implementaci určitého rozhraní. Takové závislosti v podstatě eliminují výhody, které poskytuje použití rozhraní, zhoršují údržbu kódu, znovupoužitelnost kódu a dokonce znemožňují řízené testování kódu pomocí jednotkových testů.



Obrázek 4.2: Závislosti při užití DI.

4.2 Formy DI

Základní myšlenkou DI je mít oddělené objekty od konkrétních implementací rozhraní pomocí konfigurátoru závislostí. Ten přebírá řízení ve tvorbě konkrétních instancí a v jejich vkládání do závisejících tříd. Tato myšlenka je zobrazena obrázkem 4.2.

Z obrázku je zjevné, že nyní již třída `TwitterClient` závisí pouze na rozhraních `Shortener` a `Tweeter`, pomocí konfigurátoru závislostí je umožněno libovolně měnit konkrétní implementace těchto rozhraní. Tyto implementace jsou poté konfigurátorem vytvořeny a vloženy do objektu třídy `TwitterClient`.

V DI se vyskytují tři hlavní formy, jakými jsou instance do závislých objektů konfigurátorem vkládány - pomocí argumentů konstruktorů (*Constructor Injection*), pomocí metod nazývaných settery (*Setter Injection*) nebo programovou reflexí přímo do vnitřních proměnných (*Field Injection*).

4.2.1 Constructor Injection

Třída deklaruje všechny svoje závislosti pomocí argumentů konstruktoru:

```
1  class TwitterClient...
2      private Tweeter tweeter;
3      private Shortener shortener;
4      public TwitterClient(Tweeter tweeter, Shortener shortener) {
5          this.tweeter = tweeter;
6          this.shortener = shortener;
7      }
```

Konfigurátor závislostí pak musí postupovat tak, že nejprve vytvoří instance objektů implementující požadovaná rozhraní a následně vytváří instanci třídy `TwitterClient` konstruktorem, kterému předává připravené instance.

Tato forma DI se hodí na mandatorní závislosti, neboť nelze vytvořit instanci závislé třídy pokud nejsou k dispozici její závislosti.

4.2.2 Setter Injection

Třída deklaruje všechny svoje závislosti pomocí metod:

```
1 class TwitterClient...
2     private Tweeter tweeter;
3     private Shortener shortener;
4     public void setShortener(Shortener shortener) {
5         this.shortener = shortener;
6     }
7     public void setTweeter(Tweeter tweeter) {
8         this.tweeter = tweeter;
9     }
```

Konfigurátor závislostí si připraví objekty tříd implementující obě rozhraní, poté vytvoří instanci třídy `TwitterClient` a následně zavolá obě metody a poskytne jim připravené instance.

Oproti předchozí formě je tato vhodnější na nepovinné závislosti, tedy takové, bez kterých objekt stále může plnit svojí funkci.

4.2.3 Field Injection

Závislosti jsou vloženy přímo do vnitřních proměnných objektu:

```
1 class TwitterClient...
2     private Tweeter tweeter;
3     private Shortener shortener;
```

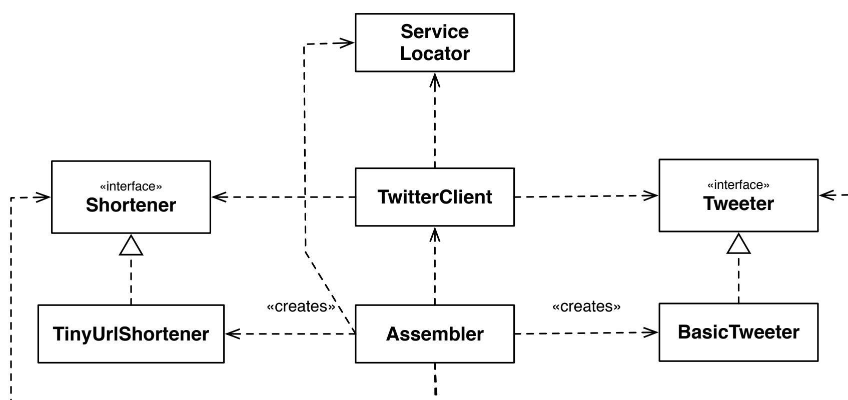
Konfigurátor opět připraví požadované objekty, poté instanci třídy `TwitterClient` a následně vloží objekty přímo do vnitřních proměnných. Problém nastává pokud jsou, jako v případě ukázky, tyto vnitřní proměnné definovány jako `private` nebo `protected`. Konfigurátor v tomto případě musí vkládat vytvořené objekty pomocí programové reflexe.

Obecně je tento způsob nedoporučovaný. Třída totiž nedává explicitně najevo, na čem je závislá a plnění závislostí pomocí reflexe je poněkud netransparentní.

4.3 Service Locator

Service Locator je stejně jako DI návrhový vzor a řeší naprosto stejný problém. Zásadním rozdílem mezi oběma vzory je ve formě, kterou do závislých objektů dopravuje jejich závislosti. Zatímco DI je založeno na vložení závislostí konfigurátorem závislostí, Service Locator funguje jako registr objektů, ze kterých si své závislosti vyzvedne sám závislý objekt. Jak se změní hierarchie aplikace při použití tohoto vzoru ukazuje obrázek 4.3.

Následující ukázka kódu dokresluje tento návrhový vzor:



Obrázek 4.3: Závislosti při užití vzoru Service Locator.

```
public interface ServiceLocator...
    public Object locateService(String name);
    public void saveService(String name, Object service);

class TwitterClient...
    private Tweeter tweeter;
    private Shortener shortener;
    public TwitterClient(ServiceLocator locator) {
        this.tweeter = (Tweeter) locator.locateService("tweeter");
        this.shortener = (Shortener) locator.locateService("shortener");
    }
```

Pojednání o výhodách a nevýhodách jednotlivých přístupů včetně dalších informací lze dohledat v článku [8].

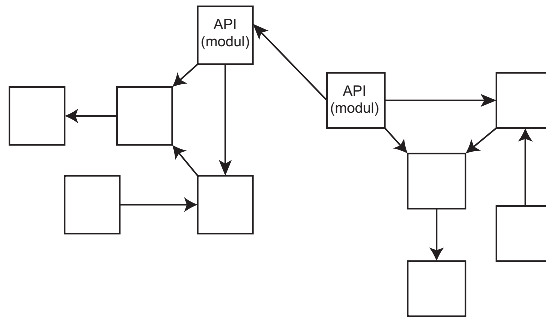
4.4 Vztah pojmu Inversion of Control k DI

Pojmy DI a *Inversion of Control* (dále jen IoC) se téměř výhradně vyskytují společně, přesto však panují rozdílné názory na jejich popis a vzájemný vztah. Tato práce se ztotožňuje s výkladem, který uvádí Fowler [8].

IoC je obecným pojmem, který vyjadřuje inverzi (převrácení) v řízení běhu aplikace. Je tedy podstatné ve spojení s tímto pojmem uvádět, o jakou inverzi se konkrétně jedná. Pokud se IoC vyskytuje ve spojení s pojmem závislost, je zpravidla zamýšleno převrácení způsobu vkládání závislostí ze závisejících tříd do konfigurátorů závislostí.

Tato přílišná obecnost pojmu IoC při užívání v souvislosti se závislostmi vedla k tomu, že vývojáři shledali tento pojem příliš zmatečným a ustálili se na novém konkrétnějším pojmu *Dependency Injection*.

Tudíž se pojmy IoC a DI (v kontextu závislostí) dají považovat za synonyma s tím, že správnější je v tomto případě užívat pojem DI.



Obrázek 4.4: Ilustrace modularity za použití Guice frameworku, převzato z [18].

4.5 Implementace pro Javu

V předchozím textu je uvedeno na čem zakládá, jak funguje a co nám přináší DI. Stále je však potřeba zajistit dodání potřebných instancí objektů, které jsou jinými objekty vyžadovány. To samozřejmě lze dělat ručně, kdy se vytvoří všechny instance objektů propojí se mezi sebou do funkčního celku. Tento proces však může být poněkud komplikovaný a náchylný na chyby.

Na trhu se nachází několik povedených nástrojů, které se o sestavení závislostí postarají sami na základě poskytnutých imformací. V následujícím textu jsou okrajově popsány dva frameworky, které poskytují DI. První z nich je framework vyvinutý pod záštitou společnosti Google - *Google Guice*, druhý je součástí Spring frameworku a nazývá se *Spring IoC kontejner*.

4.5.1 Google Guice

Google Guice je open source framework vyvíjený ve společnosti Google. Guice umožňuje implementačním třídám deklarovat svoje závislosti pomocí anotace `@Inject` na úrovni konstruktoru, metody nebo vnitřní proměnné. Aplikační celky jsou skládány do takzvaných modulů, ve kterých je provedena programovaná deklarace informací pro rozřešení závislostí. Soustavu takovýchto modulů ukazuje obrázek 4.4.

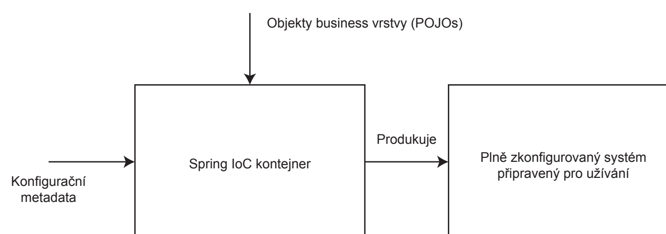
Následuje ukázka kódu s užitím Guice, která zajistí nastavení implementačních tříd pro jednotlivá rozhraní. Použitím *injectoru* na řádku 21 dojde k propojení všech závislostí.

```
1  public class TweetClient ...
2    private final Shortener shortener;
3    private final Tweeter tweeter;
4    @Inject
5    public TweetClient(Shortener shortener, Tweeter tweeter) {
6      this.shorterner = shortener;
7      this.tweeter = tweeter;
8    }
```

```
9
10 public class TweetModule extends AbstractModule {
11   protected void configure() {
12     bind(Shortener.class).to(TinyUrlShortener.class);
13     bind(Tweeter.class).to(SmsTweeter.class);
14   }
15 }
16
17 ...
18 public static void main(String[] args) {
19    Injector injector = Guice.createInjector(new TweetModule());
20    TweetClient client = injector.getInstance(TweetClient.class);
21    TweetClient.setVisible(true);
22 }
```

Guice poskytuje rozmanité možnosti, jak deklarovat (konfigurovat) závislosti a implementační třídy, ty však již jdou za rámec této práce. Velmi zajímavým zdrojem informací pro tento framework je přednáška v rámci Google IO konference [18] a kompletním zdrojem informací je pak dokumentace k samotnému frameworku [9].

4.5.2 Spring IoC kontejner



Obrázek 4.5: Funkce Spring IoC kontejneru, převzato z [22].

Spring IoC kontejner je samostatně použitelná součást Spring frameworku. Základní jednotkou pro tvorbu aplikací ve Springu jsou objekty nazývající se Srping Beany. Tyto beany jsou vytvářeny, sestavovány a spravovány právě Spring IoC kontejnerem. Vazby mezi jednotlivými beany jsou definovány pomocí metadat. Funkčnost IoC kontejneru zobrazuje obrázek 4.5.

Konfigurace metadat se ve Springu realizuje buďto pomocí XML souborů nebo Java 5 anotacemi přímo v Java třídách, následující fragment kódu ukazuje užití Spring IoC kontejneru v ukázkovém scénáři:

```
1 public class TweetClient ...
2     private final Shortener shortener;
3     private final Tweeter tweeter;
4
5     public TweetClient(Shortener shortener, Tweeter tweeter) {
```

```
6           this.shorterner = shortener;
7           this.tweeter = tweeter;
8       }
9
10  <beans>
11      <bean id="tweetClient" class="org.example.TweetClient">
12          <property name="shortener" ref="shortener"/>
13          <property name="tweeter" ref="tweeter"/>
14      </bean>
15      <bean id="tweeter" class="org.example.SmsTweeter" />
16      <bean id="shortener" class="org.example.TinyUrlShortener" />
17  </beans>
18
19  ...
20  public static void main(String[] args) {
21      ApplicationContext context =
22          new ClassPathXmlApplicationContext(new String[] {"services.xml"});
23      TweetClient client = context.getBean("tweetClient", TweetClient.class);
24      TweetClient.setVisible(true);
25  }
```

Je zjevné, že funkčnost Spring IoC kontejneru je obdobná jako funkčnost frameworku Guice. Spring IoC taktéž poskytuje rozmanité možnost konfigurace metadat a díky možnosti definovat metadata v XML souborech je umožněno používání POJO[19] objektů. Podrobné informace ke kontejneru lze nalézt v dokumentaci [22].

4.6 DI v komponentových modelech

DI na úrovni komponentových modulů nalézá své uplatnění zejména v kontextu služeb. Uvedené komponentové modely (EJB, OSGi i CoSi) poskytují mechanismy pro registraci a vyzvedávání služeb mezi jednotlivými komponentami. Deklarované závislosti komponenty jsou de-facto závislosti jejích služeb na službách ostatních komponent. Registry služeb se tedy zdají být ideálním místem pro využití DI.

EJB3 disponují nativním mechanismem pro řízení a vkládání závislostí založeném na Java 5 anotacích (podporována je i deklarace na úrovni XML). Využívá se konkrétně mechanismu JNDI vyhledávání a vkládání závislostí na všech úrovních (*constructor injection, setter injection a field injection*). Následující ukázka znázorňuje vložení závislosti do vnitřní proměnné objektu:

```
1  @Resource(name="jdbc/MyDataSource")
2  private DataSource dataSource;
```

OSGi samo o sobě podporu DI neposkytuje. Nicméně v komunitě OSGi vývojářů se vyskytuje hned několik rozšíření, která tuto podporu doplňují.

[19]Klasické Java třídy bez závislosti na použitém kontejneru či frameworku.

4.6.1 Declarative Services

Jedná se o jedno z nejstarších a nejméně rozsáhlých rozšíření, které je součástí specifikace OSGi (konkrétně Compendium Services V4).

Tato extenze je založena na XML deklaraci metadat, na které odkazuje hlavička v manifestu. Událostmi frameworku je zajištěno, že je rozšíření upozorněno na registraci nových bundlů a díky nové informaci v jejich metadatech se provede automatické vytvoření instancí služeb a navázání závislostí mezi nimi. Více informací o tomto rozšíření lze dohledat ve specifikaci [15].

4.6.2 Spring Dynamic Modules

Spring Dynamic Modules pro OSGi service platformu umožňuje běh aplikací založených na Spring frameworku v OSGi. Tím je do OSGi dodána i kompletní funkčnost Spring IoC kontejneru, která byla okrajově popsána v sekci *Spring IoC kontejner*. OSGi specifikace ve verzi 4.2 představuje specifikaci *Blueprint Service Container,* čímž Spring DM povyšuje na jeden ze standardních způsobů, jak dosáhnout DI na úrovni služeb v OSGi. Více informací k *Blueprint Service Container* je k dohledání ve specifikaci [15].

4.6.3 Peaberry

Peaberry projekt je rozšiřující knihovna pro Google Guice, která umožňuje DI na úrovni služeb v OSGi. V rámci knihovny je připravena integrace do OSGi formou bundlu a zároveň je k dispozici zásuvný model pro implementaci podpory libovolného jiného frameworku pro DI. Peaberry je nasaditelné na libovolnou implementaci OSGi specifikace V4 - například Euquinox, Felix a další.

S použitím Peaberry je do OSGi zavedena funkčnost popsaná v sekci *Google Guice,* další informace lze dohledat na webové stránce projektu [10].

Kapitola 5

Obecné úpravy CoSi

Aby bylo možné doplnit do CoSi požadovaná rozšíření, bylo nejprve třeba navrhnout a implementovat obecné úpravy. Tyto úpravy později poskytnou nezbytnou funkcionalitu pro jednotlivá rozšíření tak, aby byla do systému přidána co nejšetrněji.

Hierarchicky byla tato kapitola předřazena i přesto, že při návrhu obou rozšíření vznikaly tyto úpravy souběžně. Pokud bylo třeba pro AOP či DI nějakým způsobem upravit chování jádra, bylo vždy postupováno tak, že byly hledány podobné funkcionality v ostatních komponentových modelech (zejména v OSGi) a následně pak, v duchu těchto inspirací, byla navržena a implementována úprava CoSi.

Následující text obsahuje výčet všech větších úprav, které v systému proběhly. Tento výčet je rozdělen v rámci tří sekcí. První se věnuje popisu a návrhu jednotlivých úprav v kontextu s původním stavem CoSi. Druhá popisuje fyzickou implementaci úprav - tedy jaké třídy byly změněny, jaké přístupy byly použity, atp. Poslední, třetí sekce, zhodnocuje funkční stránku provedených úprav - pro tento účel byla navržena jednoduchá testovací aplikace. Na této aplikaci budou postupně testovány všechny úpravy i všechna rozšíření (AOP, DI), což bude demonstrovat nejen funkčnost, ale i přínos těchto rozšíření.

5.1 Popis obecných úprav

Stěžejní částí celého CoSi je bezesporu mechanismus služeb, který zprostředkovává komunikaci, výměnu funkčností a potažmo celou modularitu mezi jednotlivými bundly. Cílová rozšíření se více či méně zaměřují svou přidanou hodnotou právě směrem k práci se službami a tak i obecné úpravy přidávají funkcionalitu zejména v této oblasti.

5.1.1 Registr služeb

CoSi neobsahuje separátní registr služeb jako takový. Úlohu správce registrovaných služeb přebírá aplikační kontext, který je ve frameworku implementován třídou `ApplicationContext`. Mimo evidování služeb, se také stará o evidenci informací o exportérech a importérech všech druhů vlastností (pojem vlastnost definuje metamodel CoSi, který je popsán v sekci *Komponentový model CoSi* na straně 15). Vzhledem k tomu, že převážná část návrhových a implementačních prací bude probíhat nad životním cyklem služeb, je na místě, aby byla správa služeb vyjmuta mimo aplikační kontext s tím, že tento kontext bude poskytovat referenci na nově implementovaný *Registr Služeb*.

Pro návrh rozhraní registru je nejprve třeba zmapovat původní API, které poskytuje aplikační kontext. To zobrazuje následující kód:

```
1  public class ApplicationContext...
2    public void addNewService(ServiceReference P_newService);
3    public void removeService(Object P_service);
4    public ServiceReference getService(String clazz);
5    public ServiceReference getNamedService(String clazz, String name);
6    public ArrayList<ServiceReference> getServices(String clazz,
7        HashMap<String, String> P_properties);
8    public Object getService(ServiceReference P_service);
9    public void removeBundleServices(Bundle P_bundle);
10   public Collection<ServiceReference> getInstalledServicesForBundle(int Pi_id);
```

Z výše uvedených metod bylo navrženo nové API pro registr služeb. Při návrhu byl kladen důraz na dodržení zpětné kompatibility a také na opravu některých nešvarů. Zejména se jednalo o úpravu návratových hodnot z konkrétních implementací na obecná rozhraní a o změnu nestandardních konvencí v zápisu kódu na obecně uznávaný standard. Výsledkem je následující předpis:

```
1  public class ServiceRegistry...
2      public void registerService(ServiceReference service);
3      public void unregisterService(Object servive);
4      public void unregisterService(ServiceReference service);
5
6      public ServiceReference getService(String className);
7      public ServiceReference getNamedService(String className, String name);
8      public Object getService(ServiceReference service);
9      public <T> T getService(Class<T> cls);
10
11     public Collection<ServiceReference> getServices(String className,
12             Map<String, String> properties);
13     public Collection<ServiceReference> getRegisterdServices();
14     public Collection<ServiceReference> getServices(Bundle bundle);
```

Nové aplikační rozhraní více méně koreluje s původním, není tedy nutné zabíhat do detailů popisováním funkčností jednotlivých metod. Za zmínku však stojí dvě nově přidané metody.

První byla přidána čistě z logiky věci, jedná se o metodu `unregisterService(ServiceReference)`. Její význam je prostý, má za úkol zkrátit cestu k odebrání služby za pomocí její reference. Původně bylo potřeba nejprve pomocí reference objekt služby vyzvednout a ten poté odebrat, nyní lze služba odebrat přímo na základě této reference.

Druhá metoda - `getService(Class<T>)` - je zajímavější. V kódu frameworku se často vyskytuje kód pro přetypování služby z třídy `Object` na její konkrétní rozhraní. Přitom už pro vyzvednutí služby je nutné napsat název jejího rozhraní a poté ještě přetypovat, což se zdá jako zbytečná práce navíc. Tato metoda má za úkol tuto práci odstranit. Místo názvu rozhraní ve formě řetězce přijímá objekt třídy `Class`, z něho zjistí jméno rozhraní, vyhledá příslušnou službu a automaticky ji přetypuje. Teoreticky může nastat případ (prakticky by ale neměl), kdy přetypování selže, v tom případě metoda nevyhodí výjimku, ale vrátí `null`.

5.1.2 Životní cyklus služeb

Framework poskytuje nástroje, jak naslouchat změnám v životním cyklu bundlů a nástroje, jak na ně reagovat. Neposkytuje však obdobný mechanismus u služeb, přitom má své uplatnění.

Vysvětlení, nebo příklad, takového uplatnění se nepodává snadno. Nejprve je nutné si uvědomit, jak nakládá framework se závislostmi uvedenými v meta informacích bundlu. Jednoduše je všechny projde a zjistí, jestli jsou jinými bundly uspokojeny. To vše se ale děje pouze na deklarativní úrovni a dále se již nekontroluje, jestli službu, kterou bundle deklaroval, opravdu zaregistroval. To vede k tomu, že bundle závislý na určité službě, byl do systému zaveden, ale při snaze vyzvednout službu neuspěl. Nyní nemá možnost, jak na případnou registraci kýžené služby vyčkat a končí chybou. Možnost naslouchání životnímu cyklu služeb tuto možnost, mimo jiná uplatnění, poskytne.

Služby v CoSi mohou prakticky nabývat pouze dvou stavů, jedním stavem je *REGISTERED* a druhým stavem je *UNREGISTERING.*

- REGISTERED - v tomto stavu se nachází služba v okamžiku, kdy byla legálně zaregistrována do systému. Legálně znamená, že proběhly všechny kontroly na restrikce spojené s meta-modelem CoSi.

- UNREGISTERING - do tohoto stavu služba přechází v okamžiku, kdy byla zavolána metoda na odstranění služby z registru služeb. Jedná se o konečný stav, ze kterého již služba nepřechází do jiného.

Registr služeb se tedy dále rozšíří o řízení životního cyklu spravovaných služeb. Při přechodech mezi jednotlivými stavy bude generovat události, přičemž umožní libovolné přihlášení posluchačů těchto událostí. Implementace tohoto mechanismu bude vycházet z již stávajícího modelu, který se vyskytuje u bundlů a assemblies[20]. V aktuální verzi frameworku je implementován manažer posluchačů třídou `ListenerManager`, kterou bude třeba rozšířit o nový typ události, konkrétně o `ServiceEvent`. Dále je nutné rozšířit rozhraní `BundleContext` o následující metody:

```
1  public void addServiceListener(ServiceListener serviceListener);
2  public void addServiceListener(ServiceListener serviceListener, String filter);
3  public void removeServiceListener(ServiceListener serviceListener);
```

Zatímco význam metod na prvním a třetím řádku je zjevný a nepotřebuje další komentář, metoda na třetím řádku obsahuje nový prvek v CoSi frameworku a tím je LDAP filtr pro identifikaci služeb dle definovaných atributů.

5.1.3 LDAP filtr přes atributy služeb

Aktuální verze CoSi aplikačního rámce umožňuje nastavovat službám při jejich registraci libovolné množství atributů (klíč-hodnota). Tato možnost má svoje opodstatnění v reálné potřebě registrovat více než jednu službu pod právě jedním rozhraním. Díky těmto atributům společně s pojmenováním dané služby (což je de-facto také atribut `name=MyUsefulService`) je pak možné vyžádat si konkrétní implementaci obecné služby.

Pro názornost práce uvádí ukázkovou aplikaci implementující cizojazyčný slovník s následujícím triviálním rozhraním, jeho dvěma triviálními implementacemi a registrací do kontejneru:

```
1  interface TranslateService {
2    public String translate(String word);
3  }
4  // Implementace sluzby
5  public class TranslateService1 implements TranslateService {
6    public String translate(String word) {
7      if (word.equals("Ahoj")) {
8        return "Hello";
9      }
10   }
11 }
12 public class TranslateService2 implements TranslateService {
13   public String translate(String word) {
14     if (word.equals("Ahoj")) {
15       return "Bon␣jour";
```

[20]Assembly je v pojetí CoSi skupina bundlů. Slouží pouze jako jednotka nasazení a neumožňuje hierarchické zanořování.

```
16        }
17      }
18   }
19   // Registrace sluzeb do kontejneru
20   Map<String, String> attrs = new HashMap<String, String>(2);
21   attrs.put("From", "Czech");
22   attrs.put("To", "English");
23   bundleContext.registerService(TranslateService.class.getName(),
24       new TranslateService1(), attrs);
25
26   attrs = new HashMap<String, String>(2);
27   attrs.put("From", "Czech");
28   attrs.put("To", "French");
29   bundleContext.registerService(TranslateService.class.getName(),
30         new TranslateService2(), attrs);
```

Další bundle, řekněme implementace uživatelského rozhraní slovníku, bude potřebovat vyhledat:

1. Všechny slovníky překládající z českého jazyka.
2. Všechny slovníky překládající z českého jazyka do jazyka začínajícího na písmeno „F".

Zatímco první požadavek je v aktuálním návrhu CoSi frameworku splnitelný pomocí metody `getService(...)`, druhý požadavek splnit nelze.

Právě pro podporu variabilní selekce služeb bude implementován LDAP filtr. Výše je uvedena metoda pro výběr na úrovni interceptorů životního cyklu, dále je přidána do rozhraní `BundleContext` další metoda pro aplikaci filtrů při vyhledávání služeb v registru:

```
1  public ServiceReference[] getServiceReferences(String clazz,
2      String filter);
```

Následují ukázky užití tohoto filtru:

```
1   // Splneni druheho pozadavku
2   getServiceReferences("TranslateService", "(&(From=Czech)(To=F*))");
3
4   // notifikovat pouze o zmenach daneho typu sluzby
5   addServiceListener(this, "(objectClass=TranslateService)")
6
7   // dalsi priklad demonstrujici operatory LDAP
8   // Tento filtr vybira vsechny sluzby typu TranslateService, ktere zaroven
9   // nemaji atribut From nastaven na Czech
10  addServiceListener(this, "(&(objectClass=TranslateService)(!(From=Czech)))")
```

Syntaxe LDAP filtru je poměrně známá, v případě potřeby je možné ji dohledat v dokumentaci společnosti Microsoft [13].

5.1.4 Hooky na úrovni služeb

Návrh životního cyklu služeb umožňuje naslouchat registracím/od-registracím služeb do/z registru služeb, neumožňuje však měnit podobu služeb (například změnou reference či jinak). Takové vlastnosti bude třeba při implementaci prvků AOP. Volba mechanismu hooků je výsledkem zkoumání komponentových modelů obsahujících podporu aspektově orientovaného programování. Konkrétně OSGi Service Platform specifikace [16] představuje termín hook a specifikuje jeho funkci a přínos. Uvedeny jsou tři druhy hooků:

- *hook na úrovni událostí služeb* - předchází doručení událostí služeb jejich posluchačům, hooku je povoleno odebírat události pro specifické bundly, čímž je umožněno efektivně maskovat události před bundly.
- *hook na úrovni vyhledávání služeb* - předchází metodám určeným pro vyhledání služby, výsledek vyhledání může být hookem změněn odebráním určitých referencí na službu, čímž je umožněno efektivní skrývání specifických služeb před specifickými bundly.
- *hook na úrovni posluchačů událostí služeb* - tento hook nemá v kontextu práce opodstatnění a je uveden pouze pro úplnost.

Návrh mechanismu hooků do CoSi je silně inspirován touto specifikací, přičemž navíc přidává typ čtvrtý:

- *hook na úrovni registrace služeb* - předchází registraci služby, reference na registrovanou službu může být hookem změněna, což umožňuje efektivní obalování služeb proxy třídami.

Z výše uvedeného vyplývá, že hooky umožňují standardizovanou a efektivní cestou měnit chování registru služeb. Registrují se stejně jako klasické služby, musí však implementovat speciální rozhraní, které dodává framework. Na základě těchto rozhraní jsou identifikovány registrem služeb, který k nim umožňuje přístup. V registru služeb je proto přidána následující metoda:

```
1 public <T> Set<ServiceReference> getHooks(Class<T> hookClass);
```

Framework předepisuje tři rozhraní (dle výše uvedených typů hooků):

```
1 package cz.zcu.kiv.cosi.core.hooks.service;
2 // ...
3 public interface EventHook {
4   public void event(ServiceEvent event, Collection<Bundle> bundles);
5 }
6 // ...
```

```
7  public interface FindHook {
8    public void find(BundleContext context, String name,
9        Collection<ServiceReference> references);
10 }
11 // ...
12 public interface RegisterHook {
13   public ServiceReference register(BundleContext bundle,
14       ServiceReference serviceReference, Object service);
15 }
```

Hook na úrovni událostí služeb

Pro zachytávání událostí služeb, které jsou doručovány bundlům, je potřeba zaregistrovat `EventHook` jako běžnou službu do registru služeb. Framework poté musí posílat události služeb všem zaregistrovaným hookům. Zároveň bundle, který chce registrovat hook musí tento záměr patřičně deklarovat. Tato deklarace se provádí stejně jako klasická deklarace služby, tedy v manifest souboru pomocí hlavičky `Provide-Services`, kde poskytovaná služba je odvozena od rohraní `EventHook`. Pořadí volání hooků je určeno pořadím, ve kterém byly hooky zaregistrovány. Hooky jsou volány *poté,* co je vygenerována událost služby, ale *před tím*, co jsou tyto události doručovány bundlům, které zaregistrovaly posluchače. Hooku je předán seznam všech bundlů, kterým má být událost doručena a to formou seznamu, ze kterého lze pouze odebírat položky. To umožňuje hooku na úrovni událostí služeb skrývat určité události pro určité bundly.

Hook na úrovni událostí obdrží všechny typy událostí, tedy událost přechodu na stav REGISTERED i UNREGISTERING. Rozhraní EventHook má pouze jednu metodu:

- `event(ServiceEvent, Collection)` - Je volána v okamžiku, kdy byla vygenerována událost týkající se služeb. Metodě je předána vygenerovaná událost a seznam bundlů, které mají být událostí notifikovány. Tato kolekce umožňuje odebírání položek.

Hlavním účelem tohoto typu hooku je možnost maskování určitých událostí služeb pro určité bundly.

Hook na úrovni vyhledávání služeb

Hook na úrovni vyhledávání služeb je volán v okamžiku, kdy se cílový bundle pokouší vyhledat a vyzvednout službu z registru služeb za pomocí jedné z metod k tomu určených. Zaregistrovanému hooku je umožněno procházet výslednou kolekci referencí služeb, a zároveň také z této kolekce odebírat položky. Tím může hook skrývat libovolné reference služeb pro libovolné bundly.

Rozhraní pro tento typ služeb (FindHook) definuje opět pouze jednu metodu:

- `find(BundleContext, String, Collection)` - Je volána ve chvíli, kdy jsou vyhledány všechny reference na služby, které odpovídají původnímu volání bundlu. První argument obsahuje kontext bundlu, který o službu požádal, druhý argument je název rozhraní, od kterého je služba odvozena a konečně třetí argument obsahuje kolekci referencí služeb, které odpovídají původnímu volání bundlu. Tato kolekce umožňuje odebírání položek.

Tento hook je do systému přidán, aby umožňoval efektivním způsobem omezit viditelnost určitých služeb pro určité bundly.

Hook na úrovni registrace služeb Hook na úrovni registrace služeb je volán ihned potom, co je libovolným bundlem volána metoda určená pro registraci služby. Registrovanému hooku tohoto typu je umožněno změnit výsledný ukazatel na objekt, který službu implementuje. Rozhraní tohoto hooku předpisuje, stejně jako všechny předchozí hooky, pouze jednu metodu:

- `register(BundleContext, ServiceReference, Object)` - Je volána po registraci libovolné služby do kontejneru. První argument metody je kontext bundlu, který registraci provedl, druhý argument je reference na službu, která byla vytvořena kontejnerem. A konečně poslední argument je objekt, který reprezentuje registrovanou službu. Této metodě je umožněno vracet novou referenci na službu.

Smyslem tohoto hooku je možnost úpravy registrované služby, což umožňuje například skrýt registrovanou službu za proxy třídu. Tento hook není specifikován v OSGi a je přidán právě pro podporu AOP aspektů, které budou diskutovány v pozdější kapitole.

5.2 Implementace obecných úprav

Následující sekce se věnuje fyzickým změnám v programovém kódu jádra CoSi. Snaží se minimalizovat ukázky programového kód jen na důležitá místa v implementaci. V rámci každého bodu uvádí *seznam změň*, který je určen pro rychlou orientaci při implementaci dalších rozšíření v rámci této práce.

5.2.1 Drobné úpravy

Během implementace hlavních rozšíření vznikly drobné úpravy jádra, které nebyly původně navrženy. Tyto úpravy však ovlivňují chování frameworku, proto je nutné alespoň krátce uvést jejich popis.

Deprecated hlavička manifestu

Specifikace uvádí dva typy hlaviček pro definici poskytovaných i vyžadovaných služeb mající stejný význam a dále uvádí, že původní hlavička `*-Interfaces` je *deprecated*. Při studiu aktuálních zdrojových kódů bylo zjištěno, že v případě kdy vývojář použije v deklaraci manifest souboru obě tyto hlavičky, což není explicitně zakázáno, framework upřednostní hlavičku `*-Services`. Toto chování je patrné z následujícího fragmentu programového kódu:

```
1 public List<RequiringTypeHeaderEntry> getRequireInterfaces() {
2   if (getHeaderValue(REQUIRE_SERVICES) == null)
3     return (List<RequiringTypeHeaderEntry>) getHeaderValue(REQUIRE_INTERFACES);
4   else
5     return (List<RequiringTypeHeaderEntry>) getHeaderValue(REQUIRE_SERVICES);
6 }
7 // ... analogicky dalsi metoda
```

Takové chování, bez explicitní poznámky ve specifikace, je netransparentní a proto bylo v rámci práce upraveno změnou uvedených metod a rozšířením procesu parsování metadat:

```
1  private void parseManifest() {
2      ...
3      if (getHeaderValue(PROVIDE_INTERFACES) != null) {
4          if (getHeaderValue(PROVIDE_SERVICES) == null) {
5              headers.add(new ManifestRequiringHeader(PROVIDE_SERVICES, ""));
6          }
7          List<ProvidingTypeHeaderEntry> toMerge =
8                  (List<ProvidingTypeHeaderEntry>) getHeaderValue(PROVIDE_INTERFACES);
9          ((List<ProvidingTypeHeaderEntry>) getHeaderValue(PROVIDE_SERVICES))
10                 .addAll(toMerge);
11         removeHeader(PROVIDE_INTERFACES);
12     }
13     if (getHeaderValue(REQUIRE_INTERFACES) != null) {
14         if (getHeaderValue(REQUIRE_SERVICES) == null) {
15             headers.add(new ManifestProvidingHeader(REQUIRE_SERVICES, ""));
16         }
17         List<RequiringTypeHeaderEntry> toMerge =
18                 (List<RequiringTypeHeaderEntry>) getHeaderValue(REQUIRE_INTERFACES);
19         ((List<RequiringTypeHeaderEntry>) getHeaderValue(REQUIRE_SERVICES))
20                 .addAll(toMerge);
21         removeHeader(REQUIRE_INTERFACES);
22     }
23 }
24 public List<RequiringTypeHeaderEntry> getRequireInterfaces() {
```

```
25      return (List<RequiringTypeHeaderEntry>) getHeaderValue(REQUIRE_SERVICES);
26  }
27  public List<ProvidingTypeHeaderEntry> getProvideInterfaces() {
28      return (List<ProvidingTypeHeaderEntry>) getHeaderValue(PROVIDE_SERVICES);
29  }
```

Tato změna provádí případné spojení obou kolekcí do jedné, což je transparentnější chování. Taktéž bylo přidáno vypisování varování, které upozorňuje na použití již *deprecated* hlavičky `*-Interfaces` (lze nalézt ve třídě `BundleMetada` na řádcích 144 a 152).

Kontrola vyžadovaných služeb

Specifikace definuje, že v systému nesmí být spuštěn bundle, který nemá uspokojené všechny závislosti. Tato podmínka je ve frameworku realizována na přechodu mezi stavy INSTALLED a RESOLVED (vizte obrázek na straně 16) tak, že se nejprve načtou a rozparsují veškerá metadata uvedená v manifest souboru a poté se kontrolují všechny uvedené závislosti. To se provádí postupným procházením všech ostatních bundlů nacházejících se ve stavu STARTED, přičemž je zjišťováno, zda-li požadovanou závislost poskytují. Pokud je takový bundle nalezen, pokračuje se dalšími závislostmi až k té poslední.

Ve specifikace ale není zmíněno, je-li možné na úrovni bundlu poskytovat službu (či jiné vlastnosti), kterou sám bundle chce konzumovat. Tato vlastnost by našla například uplatnění v okamžiku, kdy si bundle implementuje výchozí chování služby pro případ, že se v systému nebude vyskytovat žádná jiná implementace.

V rámci diplomové práce byla tato vlastnost do-implementována na úrovni resolvingu bundlu následující úpravou (přidány řádky 9-13):

```
1  for (RequiringTypeHeaderEntry entry : F_requiredInterfaces) {
2    if (entry.isOptional()) {
3      // no checks are done on optional entries
4      continue;
5    }
6    boolean F_hasExporter = application.existsValidInterfaceExporter(entry);
7    if (!F_hasExporter) {
8      // Bundle can be also exporter!
9      for (ProvidingTypeHeaderEntry F_providedInterfaceEntry : F_providedInterfaces) {
10       F_providedInterfaceEntry.getValue().equals(entry.getValue());
11       F_hasExporter = true;
12     }
13     if (!F_hasExporter) {
14       String Fs_interfaceClassName = entry.getValue();
15       _printResolutionFailed("There␣is␣no␣provider␣for␣required␣interface:␣"
16         + Fs_interfaceClassName + "␣" + entry.getNameAttribute());
17       state = INSTALLED;
18       return false;
19     }
```

```
20    }
21  }
```

Framework dále neumožňuje zastavit bundle poskytující vlastnosti, které importuje jiný bundle. Výše uvedenou úpravou by bylo znemožněno zastavit bundle který zároveň exportuje i importuje jeden typ služby. Bylo tedy nutné změnit i logiku v přechodu mezi stavy STARTED a STOPPING (přidány řádky 7-11):

```
1  boolean Fb_consumerExist = false;
2  for (TypeProviderIdent F_ident : F_providedInterfaces) {
3    Collection<TypeConsumerIdent> F_consumers =
4      application.getInterfaceConsumers(F_ident.getProvidedType());
5    if (F_consumers != null && F_consumers.size() > 0) {
6      boolean cont = true;
7      if (F_consumers.size() == 1
8        && F_consumers.iterator().next()
9          .getConsumingBundle().equals((Bundle) this)) {
10       cont = false;
11     }
12     if (cont) {
13       Fb_consumerExist = true;
14       break;
15     }
16   }
17 }
18 if (Fb_consumerExist) {
19   System.out.println("This␣bundle's␣services␣are␣used␣"
20     + "by␣other␣bundles␣and␣cannot␣be␣stopped.");
21   return false;
22 }
```

Obě změny probíhaly v třídě `BundleImpl`.

Odstranění povinného aktivátoru bundlu

Ve specifikaci CoSi je uvedeno, že každý bundl je považován za spustitelný a z toho důvodu musí povinně poskytovat aktivátor v podobě implementace rozhraní `BundleControl` a deklarace patřičné hlavičky v manifest souboru. Mohou však existovat bundly, které například poskytují jen předpisy služeb (rozhraní), které jsou poté napříč ostatními bundly používány.

Původní stav CoSi byl takový, že při parsování manifest souboru se neprováděla kontrola na přítomnost této hlavičky:

```
1  public class BundleMetadata ...
2      private ArrayList<String> _getMandatoryHeaders() {
3          ArrayList<String> F_ret = new ArrayList<String>();
4          ...
5          F_ret.add(BUNDLE_NAME);
6          //F_ret.add(CONTROL_CLASS);
7          ...
8          return F_ret;
```

```
9       }
```

Pokud však hlavička nebyla uvedena, byla vyprodukována chyba při resolvingu bundlu. Tudíž byl framework v jakémsi nekonzistentním mezistavu. Úpravou metody zodpovědné za resolving bundlu byl tento mezistav dotažen do podoby, kdy není potřeba definovat aktivátor. Pokud aktivátor není v manifestu uveden, je vytvořen prázdný systémový aktivátor. Změnu ukazuje následující výňatek kódu:

```
1  public class BundleImpl...
2      protected boolean doResolveBundle() {
3          ...
4          String F_bundleActivator = manifest.getControlClass();
5          try {
6              if (F_bundleActivator == null) {
7                  // Activator is not mandatory, empty implementation instead
8                  bundleActivatorClass = bundleControl = new DefaultBundleControl();
9              } else {
10                 bundleActivatorClass =
11                     bundleClassloader.loadClass(F_bundleActivator).newInstance();
12                 bundleControl = (BundleControl) bundleActivatorClass;
13             }
14         } catch (Throwable e) {
15             ...
16         }
17         ...
```

5.2.2 Registr služeb

Nejprve byl implementován celý registr služeb, který je dán rozhraním uvedeným v prvotním návrhu společně s metodou pro přístup k hookům. Implementační třída je umístěna v balíku `cz.zcu.kiv.cosi.container` a jmenuje se `ServiceRegistry`. Po implementaci bylo potřeba tento registr zasadit do systému tak, aby nebyla ovlivněna zpětná kompatibilita již vyprodukovaných bundlů.

Z třídy `ApplicationContext` byly vyjmuty všechny metody, které souvisely se službami a byla přidána jedna nová:

```
1  public class ApplicationContext...
2      private ServiceRegistry serviceRegistry;
3
4      private ApplicationContext() {
5          ...
6          serviceRegistry = new ServiceRegistry();
7      }
8      public ServiceRegistry getServiceRegistry() {
9          return serviceRegistry;
10     }
```

Následně byly refaktorovány všechny třídy, které používaly ApplicationContext k přístupu ke službám.

Seznam změn

Změněny byly následující třídy:

```
1 cz.zcu.kiv.cosi.container.BundleContextImpl
2 cz.zcu.kiv.cosi.container.BundleImpl
3 cz.zcu.kiv.cosi.container.bundles.systemservice.impl.SystemServiceImpl
```

Přidána byla následující třída:

```
1 cz.zcu.kiv.cosi.container.ServiceRegistry
```

5.2.3 Životní cyklus služeb

Pro proces doručování událostí ve změnách životního cyklu služeb bylo využito stávající implementace pro bundly. Původní systém se skládá z třídy zodpovědné za registraci posluchačů, tříd které slouží jako datové obálky pro informace spojené s generovanou událostí a z abstraktních tříd, které musely implementovat posluchači událostí.

Standardně není zvykem definovat abstrakci pro posluchače pomocí abstraktních tříd. Jelikož v Javě není umožněna vícenásobná dědičnost, je tímto přístupem znemožněno vepsat více posluchačů do jedné třídy, nebo do třídy, která má již svého předka. Abstraktní třídy předepisující metody pro posluchače tedy byly refaktorizovány na rozhraní a společně s tím byla provedena i nezbytná úprava implementovaných posluchačů a to jak na úrovni frameworku, tak na úrovni již implementovaných bundlů.

V rámci životního cyklu služeb byly přidány dvě rozhraní pro posluchače:

- `ServiceListener` - pokud bundle zaregistruje tento posluchač, jsou mu doručovány všechny události služeb, které deklaroval jako svoje závislosti.
- `AllServiceListener` - tento posluchač slouží k pozorování událostí nad všemi službami v systému, tedy i nad těmi, které nejsou přímou závislostí bundlu, který posluchač zaregistroval.

Seznam změn

Změněny byly následující třídy:

```
1 cz.zcu.kiv.cosi.container.listener.BundleUpdateListenerImpl
2 cz.zcu.kiv.cosi.container.listener.ListenerManager
3 cz.zcu.kiv.cosi.core.listener.BundleEvent
```

Přidány byla následující třídy a rozhraní:

```
1 cz.zcu.kiv.cosi.core.listener.ServiceEvent
2
3 // rozhrani
4 cz.zcu.kiv.cosi.core.listener.AllServiceListener
```

```
5 cz.zcu.kiv.cosi.core.listener.ServiceListener
6 cz.zcu.kiv.cosi.core.listener.BundleListener
7 cz.zcu.kiv.cosi.core.listener.SynchronousBundleListener
8 cz.zcu.kiv.cosi.core.listener.VetoableBundleListener
```

Odebrány byly následující třídy:

```
1 cz.zcu.kiv.cosi.core.listener.BundleListener
2 cz.zcu.kiv.cosi.core.listener.SynchronousBundleListener
3 cz.zcu.kiv.cosi.core.listener.VetoableBundleListener
```

5.2.4 LDAP filtr přes atributy služeb

V první řadě bylo nutné mít k dispozici interpret pro LDAP filtry, ten byl převzat a upraven z OSGi implementace Equinox a jeho implementace je umístěna ve třídě `FilterImpl` v balíku `cz.zcu.kiv.cosi.container`. Následně bylo potřeba vložit fyzické vyhodnocování filtrů v místech, které určují metody umožňující použití filtrů.

První metoda umožňuje filtrování příchozích událostí životního cyklu služeb a vyskytuje se ve třídě `ListenerManager`, při zavolání této metody se provede pouze rozparsování filtru a uložení do informační datové struktury. Filtr je aplikován až ve chvíli, kdy má být doručena událost:

```
1  public class ListenerManager  ...
2      public synchronized void serviceChanged(final ServiceEvent serviceEvent) {
3          Collection<Bundle> bundles = new ArrayList<Bundle>();
4          for (ServiceListenerInfo sli : serviceListeners) {
5              // Notifikace patri bud jen bundlu, ktery daou sluzbu deklaruje
6              // jako zavislost, nebo zaregistroval AllServiceListener
7              boolean notify = sli.getListner() instanceof AllServiceListener
8                  || canAccessService(sli.getBundle(), serviceEvent.getServiceReference());
9
10             // Aplikace filtru, pokud existuje a zaroven pokud
11             // ma byt bundle notifikovan
12             if (notify && sli.getFilter() != null) {
13                 notify = sli.getFilter().match(serviceEvent.getServiceReference());
14             }
15
16             // Vsechny podminky splneny, pridat tento bundle
17             // mezi notifikovane
18             if (notify) {
19                 bundles.add(sli.bundle);
20             }
21
22             // Samotna notifikace
23             for (ServiceListenerInfo sli : serviceListeners) {
24                 if (bundles.contains(sli.getBundle())) {
25                     sli.listner.serviceChanged(serviceEvent);
26                 }
27             }
28         }
```

```
29      }
```

Druhá metoda umožňuje filtrování při výběru služby z kontextu bundlu, cílem úpravy je tedy třída `BundleContextImpl`:

```
1  public class BundleContextImpl...
2      public ServiceReference[] getServiceReferences(String clazz, String filter) {
3          // Vyzvednou se vsechny reference
4          Collection<ServiceReference> srs  = application
5              .getServiceRegistry().getServices(clazz, new HashMap<String, String>());
6          Collection<ServiceReference> result  = new HashSet<ServiceReference>();
7          Filter f = FilterImpl.newInstance(filter);
8          // Do vysledku se vlozi pouze ty, ktere vyhovuji filtru
9          for (ServiceReference sr : srs) {
10             if (f.match(sr)) {
11                 result.add(sr);
12             }
13         }
14         return result.toArray(new ServiceReference[result.size()]);
15     }
```

Důležitou vlastností filtru je, že se mezi atributy výchoze přidává i název rozhraní, se kterým byla služba zaregistrována. Název klíče pro tento atribut je `objectClass`. Tato vlastnost umožňuje filtrování i dle názvu rozhraní.

Seznam změn

Změněny byly následující třídy a rozhraní:

```
1  cz.zcu.kiv.cosi.container.listener.ListenerManager
2  cz.zcu.kiv.cosi.container.BundleContextImpl
3
4  // rozhrani
5  cz.zcu.kiv.cosi.core.BundleContext
```

Přidány byly následující třída a rozhraní:

```
1  cz.zcu.kiv.cosi.container.FilterImpl
2  cz.zcu.kiv.cosi.core.InvalidSyntaxException
3
4  // rozhrani
5  cz.zcu.kiv.cosi.core.Filter;
```

5.2.5 Hooky na úrovni služeb

Implementace, stejně jako návrh, vychází ze specifikace hooků v OSGi. Jak již bylo uvedeno, hooky se registrují jako normální služby. Registr služeb při registraci jakékoliv služby kontroluje registrační rozhraní, zda-li se nejedná o hook. Pokud tomu tak je, je tato služba uložena do speciální kolekce mimo ostatní služby. Z této kolekce pak lze hooky vyzvednou metodou `getHooks(Class)`, kterou obsahuje registr služeb.

Vyzvedávání hooků provádějí tři místa v celém frameworku, jsou to přesně ta místa, které popisují jednotlivé typy hooků.

Hook na úrovni událostí služeb

Tento hook je vykonáván těsně před tím, než mají být doručeny události, proto je změna umístěna ve třídě ListenerManager:

```
1  public class BundleContextImpl  ...
2      public synchronized void serviceChanged(final ServiceEvent serviceEvent) {
3          Collection<Bundle> bundles = new ArrayList<Bundle>();
4          for (ServiceListenerInfo sli : serviceListeners) {
5              ...
6              if (bundles.size() > 0) {
7                      // Volani vsech hooku
8                      ServiceRegistry registry = ApplicationContext
9                              .getInstance().getServiceRegistry();
10                     Collection shrinkableBundles = new ShrinkableCollection(bundles);
11                     Set<ServiceReference> hooks = registry.getHooks(EventHook.class)
12                     for (ServiceReference hookReference : hooks) {
13                             EventHook hook =
14                                     (EventHook) registry.getService(hookReference);
15                             hook.event(serviceEvent, shrinkableBundles);
16                     }
17                     // Notifikace tech bundlu, ktere prosly hooky
18                     for (ServiceListenerInfo sli : serviceListeners) {
19                             if (bundles.contains(sli.getBundle())) {
20                                     sli.listner.serviceChanged(serviceEvent);
21                             }
22                     }
23             }
24         }
25     }
```

Za povšimnutí stojí třída `ShrinkableCollection`, jde o implementaci rozhraní `Collection`, která umožňuje pouze jedinou změnu nad svými daty - odebírání. Hook, který chce maskovat události před určitým bundlem ho tak má možnost z kolekce odebrat. Pokud kolekci vyprázdní celou, zamaskuje kompletně celou událost.

Hook na úrovni vyhledávání služeb

Hook tohoto typu je volán v okamžiku, kdy je v registru služeb vyhledávána služba. Bundle nikdy neoperuje nad registrem služeb přímo, nýbrž pomocí svého kontextu. Z tohoto důvodu je hook na úrovni vyhledávání služeb zasazen do třídy implementující tento kontext - třída `BundleContectImpl`. Byla v ní provedena následující úprava:

```
1  public class BundleContextImpl ...
2      public Object getService(String Ps_clazz,
3              HashMap<String, String> P_properties) {
```

```
4           Object result = null;
5           ...
6
7           Collection<ServiceReference> services = application
8                   .getServiceRegistry().getServices(Ps_clazz, P_properties);
9
10          // Aplikace hooku - nacteni typu FindHook
11          Collection shrinkableService = new ShrinkableCollection(services);
12          for (ServiceReference sr : application.getServiceRegistry()
13                  .getHooks(FindHook.class)) {
14              FindHook hook = (FindHook) application.getServiceRegistry().getService(sr);
15              hook.find(this, Ps_clazz, shrinkableService);
16          }
17          ...
18          return result;
19      }
```

Hooku je umožněno odebírat položky z kolekce (ukazatele na služby) opět díky `ShrinkableCollection`, čímž mohou maskovat implementace služeb od určitého bundlu či určitých typů.

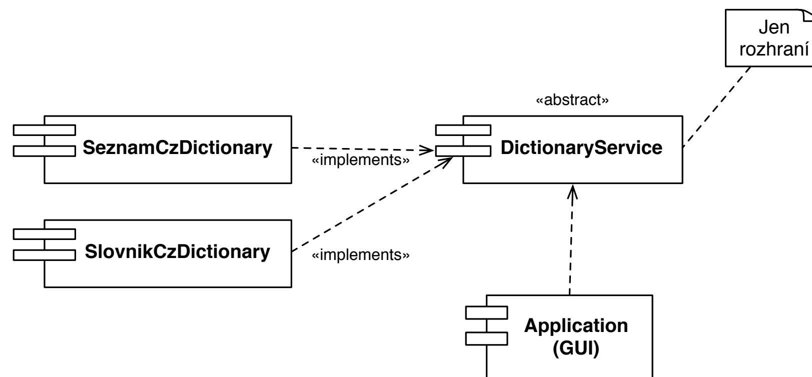
Kombinací obou uvedených hooků lze dosáhnout jednoduchého maskování služby za proxy, postup je následující:

Hook na úrovni registrace služeb

Tento hook je aplikován v okamžiku, kdy je libovolným bundlem registrována služba. Stejně jako u vyzvedávání služby nekomunikuje bundle s registrem služeb přímo ale zprostředkovaně. Zprostředkovatelem je taktéž kontext daného bundlu, tudíž i zavedení tohoto typu hooku bylo provedeno ve třídě `BundleContectImpl`:

```
1   public class BundleContextImpl  ...
2       public void registerService(String Ps_clazz, Object P_service,
3               HashMap<String, String> P_properties) {
4           ...
5
6           ServiceReference newServiceHolder = new
7                   ServiceReferenceImpl(Ps_clazz, P_service, bundle,
8                           P_properties, F_providedInterface);
9
10          // Hook implementation
11          for (ServiceReference sr : application.getServiceRegistry().
12                      getHooks(RegisterHook.class)) {
13                  RegisterHook hook = (RegisterHook) application
14                          .getServiceRegistry().getService(sr);
15                  newServiceHolder = hook.register(this, newServiceHolder,
16                          ((ServiceReferenceImpl)newServiceHolder).getService());
17          }
18          application.getServiceRegistry().registerService(newServiceHolder);
19      }
```

Jak je patrné z kódu, tento hook může změnit ukazatel na libovolnou službu a tím například obalit do proxy třídy, právě tohoto mechanismu bude využito v AOP rozší-



Obrázek 5.1: Rozložení ukázkové aplikace do bundlů.

ření.

Seznam změn

Změněny byly následující třídy:

```
1 cz.zcu.kiv.cosi.container.listener.ListenerManager
2 cz.zcu.kiv.cosi.container.BundleContextImpl
3 cz.zcu.kiv.cosi.container.ServiceRegistry
```

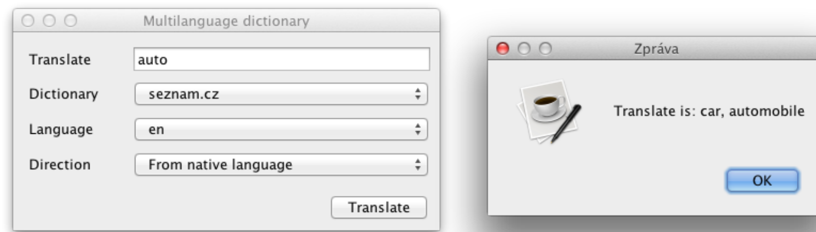
Přidány byly následující rozhraní:

```
1 cz.zcu.kiv.cosi.core.hooks.service.FindHook
2 cz.zcu.kiv.cosi.core.hooks.service.LstenerHook
3 cz.zcu.kiv.cosi.core.hooks.service.RgisterHook
```

5.3 Zhodnocení implementace obecných úprav

Při implementaci veškerých úprav a rozšíření byly prováděny průběžné testy, přesto je však vhodné výsledky demonstrovat uceleně v podobě funkční aplikace. Následující podsekce obsahuje popis jednoduché aplikace, která ověří veškeré zásahy do jádra a později bude rozšiřována tak, aby prokázala funkce obou implementovaných rozšíření.

Po obecném popisu pokračuje práce popisem míst této aplikace, která jsou v kontextu úprav jádra kritická a prokazuje, že veškeré změny provádějí svou funkci tak, jak bylo navrženo.



Obrázek 5.2: Ukázka grafického rozhraní aplikace.

5.3.1 Ukázková aplikace - multijazyčný slovník

Jedná se o jednoduchou aplikaci, která bude sloužit k překládání český slov do několika jazyků. Překlady bude čerpat z webových slovníků a je navržena tak, aby bylo možno tyto webové slovníky doplňovat implementací jednoduchého bundlu. Diagram na obrázku 5.1 ukazuje, jak bude aplikace rozložena do bundlů.

Bundle `DictinoryService` předepisuje základní rozhraní pro implementaci slovníku, rozhraní má následující podobu:

```
1  package cz.zcu.kiv.cosi.bundles.dictionary.services;
2
3  public interface DictionaryService {
4      public String getNativeLanguage();
5      public String[] getLanguages() throws Exception;
6      public String[] getTranslates(String language,
7              String word, boolean toNative) throws Exception;
8  }
```

- `getNativeLanguage()` vrací jaký je nativní jazyk slovníku v podobě dvoupísmenné zkratky.
- `getLanguages()` vrací všechny jazyky, do kterých slovník dokáže překládat.
- `getTranslates(String, String, Boolean)` vrací všechny překlady slova, které je dáno druhým argumentem metody. První argument určuje jazyk, který slovník umí překládat (pokud je zadán jazyk, který slovník nezná, je vyhozena výjimka). A třetí argument definuje, jakým směrem se bude překládat - při hodnotě `true` se bude překládat z jazyka určeného prvním argumentem do nativního jazyka, při hodnotě `false` se bude překládat opačným směrem.

Bundle `Application` implementuje jednoduché uživatelské grafické rozhraní a zároveň deklaruje závislost na rozhraní `DictionaryService`. Grafická podoba aplikace je zobrazena obrázkem 5.2 a manifest tohoto bundlu vypadá následovně:

```
Cosi-Version: 1.0
Bundle-Name: CoSi - Dictionary application NODS
Bundle-Version: 1.0.0
Bundle-Classpath: .;lib/swing-layout-1.0.3.jar
Bundle-Provider: Jakub Trunecek
Require-Services: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService;
        optional=true
Control-Class: cz.zcu.kiv.cosi.bundles.dictionary.app.Activator
```

Zbylé dva bundly jsou implementace slovníků - jeden čerpá data ze slovníku `slovnik.seznam.cz` a druhý z slovníku `slovnik.cz`. Zajímavé je, jak jsou data z těchto slovníků stahována. Při každém volání metody `getTranslates()` se sestaví URL adresa, na které jsou k dispozici překlady. Z této URL adresy se stáhne vygenerovaný HTML kód a pomocí knihovny Jsoup se provede rozparsování za použití CSS3 selektorů. Ukázka metody poskytující překlady ze slovníku `slovnik.seznam.cz`:

```
public class DictionaryServiceImpl implements DictionaryService...
    public String[] getTranslates(String language, String word,
            boolean toNative) throws Exception {
        if (!hasLanguage(language)) {
            throw new IllegalArgumentException("No␣language␣" + language);
        }
        String url = "http://beta.slovnik.seznam.cz/%s/word/?q=%s";
        String dir = String.format("%s-%s",
            toNative ? language : "cz",
            toNative ? "cz" : language);
        Collection<String> results = new ArrayList<String>();
        Document doc = Jsoup.connect(String.format(url, dir, word)).get();
        for (Element a : doc.select("div#fastMeanings").get(0).select("a")) {
            results.add(a.text());
        }
        return results.toArray(new String[results.size()]);
    }
```

Pro budoucí možnost porovnání je uvedena i podoba manifest souboru jednoho z těchto bundlů:

```
Cosi-Version: 1.0
Bundle-Name: CoSi - Dictionary service based on Seznam.cz
Bundle-Version: 1.0.0
Provide-Services: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService
Bundle-Classpath: .;lib/jsoup-1.6.2.jar
Bundle-Provider: Jakub Trunecek
Control-Class: cz.zcu.kiv.cosi.bundles.dictionary.seznamcz.Activator
Bundle-Description: Dictionary service implementation based on Seznam.cz
Require-Types: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService
```

5.3.2 Funkčnost jednotlivých úprav

Ukázková aplikace v uvedené podobě dokazuje funkčnost všech implementovaných úprav vyjma upravené kontroly vyžadovaných služeb a hooků. Funkčnost nové kont-

roly vyžadovaných služeb byla úspěšně otestovaná samostatně a vzhledem k malému rozsahu této úpravy není její test uveden v textu této práce. Hooky jsou ověřeny až implementací AOP rozšíření, protože jejich chování je velmi specifické a jejich použití se týká spíše obecných rozšíření (jako právě AOP) než-li konkrétních implementací.

V následujícím textu bude vysvětleno, ve kterých částech aplikace a jak jsou ověřeny výše uvedené úpravy.

Deprecated hlavička manifestu

Funkce je ověřena dvěma systémovými bundly, jedním je `MessageBundle` a druhým pak `ExtrafuncRegistryBundle`. Tyto bundly používají ještě starou verzi hlavičky, což je snadno ověřitelné nahlédnutím do jejich manifest souborů:

```
1 Bundle-Name: MessageBundle
2 ...
3 Provide-Interfaces: cz.zcu.kiv.cosi.core.messageservice.MessageService
```

Při spuštění kontejneru CoSi je na tento fakt upozorněno hlášením:

```
1 Starting the system...
2 Initializing container
3
4 [Warning] Header Provide-Interfaces is deprecated, use Provide-Services instead
5 [Warning] Header Provide-Interfaces is deprecated, use Provide-Services instead
```

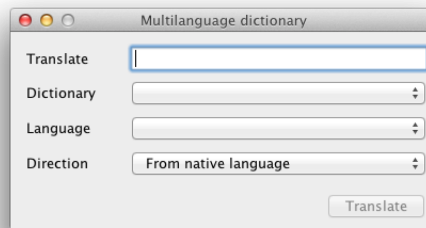
Odstranění povinného aktivátoru bundlu

V aplikaci figuruje bundle, který poskytuje pouze rozhraní a tudíž nemá žádný aktivátor. Jedná se o `DictionaryService` bundle, který má tento manifest:

```
1 Bundle-Name: CoSi - Dictionary services
2 Provide-Types: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService
```

Z ukázky je zjevné, že bundle neobsahuje žádný aktivátor, přesto je bundle do systému zaveden a spuštěn:

```
1 System shell
2 >ps
3 Id  State   Name
4 ------------------------------------------------
5 0 Started   systembundle (1.0.0)
6 1 Started   messagebundle (1.0.0)
7 2 Started   extrafuncregistrybundle (1.0.0)
8 3 Started   simpleshell.jar (1.0.1)
9 4 Started   cz.zcu.kiv.cosi.bundles.dictionary-services-1.0.0.jar (1.0.0)
```



Obrázek 5.3: Neaktivní grafické rozhraní.

Nový registr služeb

Registr služeb se používá téměř ve všech bundlech. Fakt, že jsou služby k dispozici a funguje jejich registrace i vyzvedávání dokazuje, že nový registr služeb je plně funkční. Zpětná kompatibilita je dokázána funkčností všech systémových bundlů, které byly napsány před novým registrem služeb a které taktéž používají metody pro práci se službami.

Životní cyklus služeb

Životního cyklu služeb a jeho událostí bylo využito v grafickém rozhraní aplikace. Grafické rozhraní pro svou činnost vyžaduje služby implementující rozhraní `DictionaryService`. Tato závislost je volitelná, aplikace tudíž může být spuštěna i v případě, že žádné implementace nejsou k dispozici. V tomto případě je uživatelské rozhraní neaktivní, to ilustruje obrázek 5.3 a tento fragment kódu:

```
1  public class Gui extends javax.swing.JFrame...
2      private List<DictionaryService> services = new ArrayList<DictionaryService>();
3
4      public void addService(DictionaryService service, String name) {
5          int index = services.indexOf(service);
6          if (index == -1) {
7              services.add(service);
8              translateButton.setEnabled(true);
9              dictionaries.addItem(name);
10         }
11     }
12
13     public void removeService(DictionaryService service) {
14         int index = services.indexOf(service);
15         if (index >= 0) {
16             services.remove(index);
17             dictionaries.removeItemAt(index);
18             if (services.size() == 0) {
19                 translateButton.setEnabled(false);
20             }
```

```
21            }
22        }
23
24        ...
```

Dále je aktivátorem zaregistrován posluchač, který v případě registrace služby požadovaného typu aktivuje rozhraní a přidá ji do seznamu použitelných služeb:

```
1  public class Activator implements BundleControl, ServiceListener...
2      public void start(BundleContext P_context) throws Exception {
3          ...
4          context.addServiceListener(this,
5                  "(objectClass=" + DictionaryService.class.getName() + ")");
6      }
7
8      public void serviceChanged(ServiceEvent event) {
9          DictionaryService service = (DictionaryService) context
10             .getService(event.getServiceReference());
11         if (event.getType() == ServiceEvent.REGISTERED) {
12             gui.addService(service,
13                     event.getServiceReference().getProperties().get("name"));
14         } else {
15             gui.removeService(service);
16         }
17     }
```

LDAP filtr přes atributy služeb

LDAP filtr se v aplikaci používá při registraci posluchače. Má za úkol zajistit, aby byl notifikován pouze o událostech týkajících se služby, která implementuje rozhraní `DictionaryService`. To jednak zajistí, aby posluchač nebyl volán zbytečně. A v druhé řadě to také umožní opomenout testování, zad-li se jedná o požadovanou službu a lze ji rovnou přetypovat:

```
1  public class Activator implements BundleControl, ServiceListener...
2      public void serviceChanged(ServiceEvent event) {
3          DictionaryService service = (DictionaryService) context
4              .getService(event.getServiceReference());
5              ...
6      }
```

Kapitola 6

Rozšíření CoSi o Dependency Injection

Motivaci pro přidání tohoto rozšíření lze hledat zejména ve snaze ulehčit práci vývojáře bundlu, který v aktuální verzi musí deklarovat všechny své závislosti na službách a stejně tak deklarovat všechny poskytované služby. Následně, i přes tuto explicitní deklaraci, musí jednotlivé služby vyzvedávat respektive registrovat manuálně. Cílem tohoto řešení bude možnost vše realizovat pouze deklarativně s pomocí vkládání závislostí na úrovni metod.

V úvodu budou přesně formulovány požadavky na toto rozšíření, dále bude předložena analýza aktuálního stavu, která prozkoumá a popíše současné procesy pracující na úrovni služeb. Následně, ze zjištěných skutečností, bude práce specifikovat výslednou podobu rozšíření, včetně způsobů integrace do aktuální podoby frameworku. A v poslední části kapitoly bude demonstrovat funkčnost a přínos rozšíření úpravou ukázkové aplikace.

Implementované rozšíření bylo, po vzoru OSGi, pojmenováno jako ***Deklarativní služby***.

6.1 Formulace zadání

V CoSi musí každý bundle, který chce pracovat se službami, obsahovat aktivátor, ve kterém provádí registrace a vyzvedávání služeb z běhového kontejneru. Úkolem rozšíření bude tyto registrace a vyzvedávání služeb přemístit z aktivátor do pouhé deklarace. Rozšíření *Deklarativní služby* je formulováno následujícími požadavky:

1. *musí* být navrženo a implementováno jako běžný CoSi bundle,
2. *musí* umožnit deklarativní cestou registraci a vyzvedávání služeb z kontejneru,
3. *musí* dodržet zpětnou kompatibilitu umožňující klasické registrování a vyzvedávání služeb,

4. *by mělo* omezit deklarace na nezbytné minimum,
5. *by mělo* umožnit definovat mandatorní a nemandatorní závislosti,
6. *by mělo* umožnit dynamické vkládání vícenásobných závislostí.

6.2 Analýza aktuálního stavu

Nejprve je nutné prozkoumat veškeré možnosti registrace a vyzvedávání služeb v aktuální verzi CoSi frameworku.

6.2.1 Registrace služby

Proces registrace služby začíná již v deklaraci *manifest* souboru, kde musí bundle explicitně deklarovat jakou službou bude přispívat do systému. Ze specifikace CoSi [3] lze vyčíst, že tato deklarace se realizuje pomocí hlavičky `Provide-Services`[21].

Hodnotu této hlavičky tvoří libovolný počet služeb. Deklarace služby se skládá ze jména rozhraní, které služba implementuje, společně s volitelnými parametry *verze* a *jméno*. Následuje ukázka deklarace poskytování dvou služeb:

```
Provide-Services: com.example.ServiceInterface;name=CoolService;version=1.0.0,
  com.example.AnotherService;name=LessCoolService
```

Dalším krokem je vložení instance její implementace do registru služeb. To se se nejčastěji provádí v těle metody `start()` aktivátoru bundlu voláním metody `registerService()` na aktuálním kontextu. Framework umožňuje přidat k registrované službě libovolný počet atributů (klíč-hodnota), které mohou například sloužit k lepší identifikace konkrétní služby. Ukázka tohoto kroku by mohla vypadat následovně:

```java
public void start(BundleContext context) {
  HashMap<String, String> attrs = new HashMap<String, String>();
  attrs.put("Key", "Value");
  context.registerService(ServiceInterface.class.getName(),
      new ServiceImplementation(), attrs);
}
```

6.2.2 Vyzvednutí služby

U vyzvednutí služby, stejně jako u registrace, je nejprve nutné deklarovat tento záměr v *manifest* souboru. K tomu slouží hlavička `Requires-Service`, která popisuje

[21]Deprecated hlavička `Provide-Interfaces` a `Require-Interfaces` již v práci nebude zmiňována.

všechny služby, které bundle požaduje. Ke každé požadované službě lze dále uvést následující vlastnosti: *jméno, rozsah verzí* požadované služby, *poskytovatele* služby a *rozsah verzí* bundlu jenž službu poskytuje. Ukázka použití této hlavičky vypadá takto:

```
1 Require-Services: com.example.ServiceInterface;name=CoolService;
2   versionRange=(1.0.0,2.0.0];bundleProvider=com.acme,
3   com.example.AnotherService;name=LessCoolService
```

Fyzické vyzvednutí služby z registru služeb se zpravidla provádí při startování bundlu. Konkrétně opět v těle metody `start()` aktivátoru bundlu. Kontext bundlu poskytuje několik metod pro získání služby:

- `Object getService(String)` vrací instanci služby, která byla registrována pod názvem rozhraní uvedeným v prvním argumentu. Pokud je v systému zaregistrováno více služeb pod stejným rozhraní vrátí tu, která byla zaregistrována jako první. Pokud v systému neexistuje žádná taková služba, vrátí `null`.

- `Object getService(String, Map<String, String>)` vrací instanci služby, která byla registrována pod názvem rozhraní uvedeným v prvním argumentu společně s atributy uvedenými v druhém argumentu. Existuje-li v systému více takových služeb, vrací první nalezenou, neexistuje-li žádná, vrací `null`.

- `Object getService(ServiceReference)` vrací instanci služby podle její reference uvedené v prvním argumentu. Reference služby je jednoznačná, tudíž lze najít pouze jednu nebo žádnou službu.

- `Object getService(String, String)` vrací instanci služby, která byla registrována pod názvem rozhraní uvedeným v prvním argumentu společně s atributy, které odpovídají LDAP filtru zadaným druhým argumentem. Existuje-li v systému více takových služeb, vrací první nalezenou, neexistuje-li žádná, vrací `null`. Tato metoda byla přidána v rámci této práce (vizte sekci *5.1.3*).

Použití jedné z těchto metod v aktivátoru bundlu ukazuje následující útržek kódu:

```
1 private ServiceInterface service;
2 public void start(BundleContext context) {
3   service = (ServiceInterface) context.getService(ServiceInterface.class.getName());
4 }
```

Díky novému životnímu cyklu služeb je ve frameworku nově umožněno vyzvednout službu v obsluze události. Tento přístup je vhodné použít v kombinaci s nepovinnou závislostí. Následuje ukázka:

```
1 public class Activator implements BundleControl, ServiceListener...
2
3     public void start(BundleContext P_context) {
4         context.addServiceListener(this,
```

```
5               "(objectClass=" + DictionaryService.class.getName() + ")");
6       }
7
8       public void serviceChanged(ServiceEvent event) {
9           if (event.getType() == ServiceEvent.REGISTERED) {
10              ServiceInterface service =
11                      ((DictionaryService)) context.getService(event.getServiceReference());
12              this.service = service;
13          } else {
14              this.service = null;
15          }
16      }
17
18  }
```

6.3 Návrh rozšíření

Následující návrh se skládá z několika částí. První je definice meta-modelu, který zajistí obálku pro informace o bundlu - tyto informace lze rozdělit na dvě skupiny

1. Popis požadovaných a poskytovaných služeb. Tato část je převzata z klasického způsobu deklarace vlastností, které popisuje specifikace CoSi.

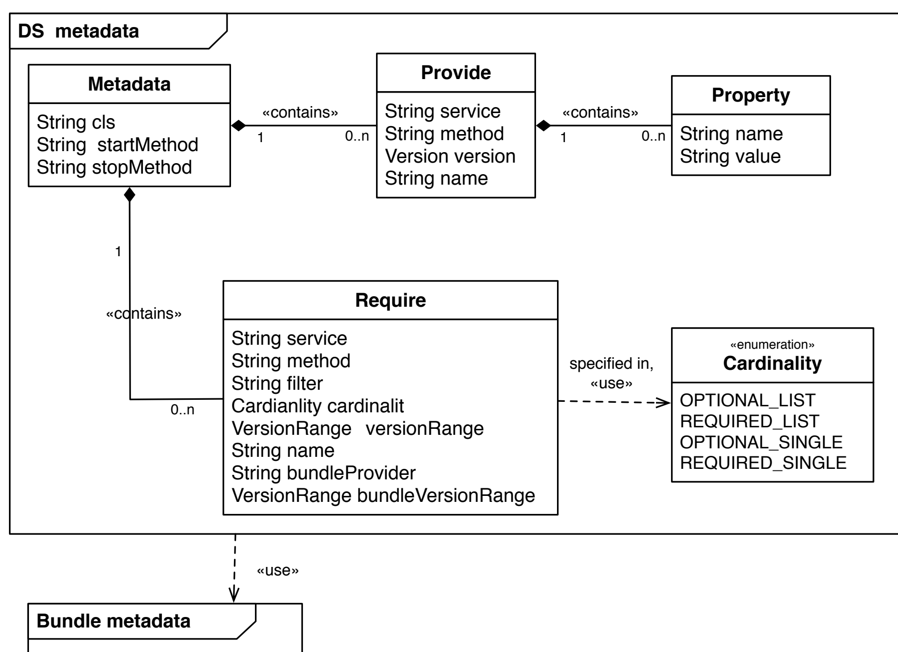
2. Popis jak provést dependency injection požadovaných služeb a jak získat instance poskytovaných služeb.

Další částí je popis chování rozšíření, tedy to, jak s informacemi nabytými z meta-modelu bude rožíření nakládat. Poslední část této sekce bude obsahovat výčet požadovaných funkcí jádra, které jsou nezbytné pro implementaci rozšíření.

6.3.1 Meta-model rozšíření

Deklarace služeb k registraci a k vyzvednutí služeb ze systémového kontejneru je zajištěna pomocí formátu XML nebo YAML, který je přidán jako soubor v deklarujícím bundlu. Skutečnost, že tato deklarace existuje, uvádí hlavička v manifest souboru.

Pokud rozšíření detekuje chybu při načítání metadat z libovolného formátu, musí vyhodit výjimku, která zajistí, že daný bundle nebude zaveden do systému. Výjimka musí být vyhozena v jakémkoliv případě: tedy pokud je například XML formát nevalidní, nebo nesplňuje definiční předpis. U YAML formátu se může jednat taktéž o nevalidní formát nebo o nesplnění mandatorních položek.



Obrázek 6.1: Návrh meta-modelu deklarativních služeb.

Definice v manifestu

Rozšíření představuje novou hlavičku v manifest souboru, která slouží jednak pro poukázání na fakt, že bundle chce využívat funkcí poskytovaných tímto rozšířením. A v druhé řadě slouží k identifikace zdroje který poskytuje popis metadat potřebných pro toto rozšíření. Hlavička má následující podobu:

```
1  Declarative-Services: META-INF/services.xml
```

V navrhované verzi rozšíření bude k dispozici načítání metadat z formátu XML a YAML, nicméně mechanismus načítání musí být navržen tak, aby bylo v budoucnu možné doplnit další formáty deklarace metadat. Aktuálně se bude identifikovat formát pomocí koncovky zdrojového souboru: U *.xml se použije XML skener a analogicky u *.yml a *.yaml pak YAML skener. Bližší informace o tom, jak přidat další formáty budou uvedeny v implementační sekci.

Navržený meta-model

Za pomocí analýzy aktuálního stavu frameworku byl sestaven meta-model, který pokrývá veškeré deklarativní možnosti na úrovni manfiest souboru. Dále meta-model pokrývá i možnosti, které jsou dány metodami pro registraci služby (atributy) a pro vyhledání služby (atributy a filtry). Tento meta-model zobrazuje obrázek 6.1.

- Hlavním prvkem celého meta-modelu je entita pojmenovaná `Metadata`, ta po-

mocí atributu `cls` obsahuje odkaz na třídu, která bude obsahovat všechny metody, použité pro dependency injection. Rozšíření svým chováním ruší potřebu implementovat aktivátor bundlu, čímž by vývojář přišel o možnost využití `start()` a `stop()` metody například pro registraci posluchačů. Entita `Metadata` proto umožňuje specifikaci názvů metod, které lze pro tyto účely využít - jedná se o atributy `startMethod` a `stopMethod`.

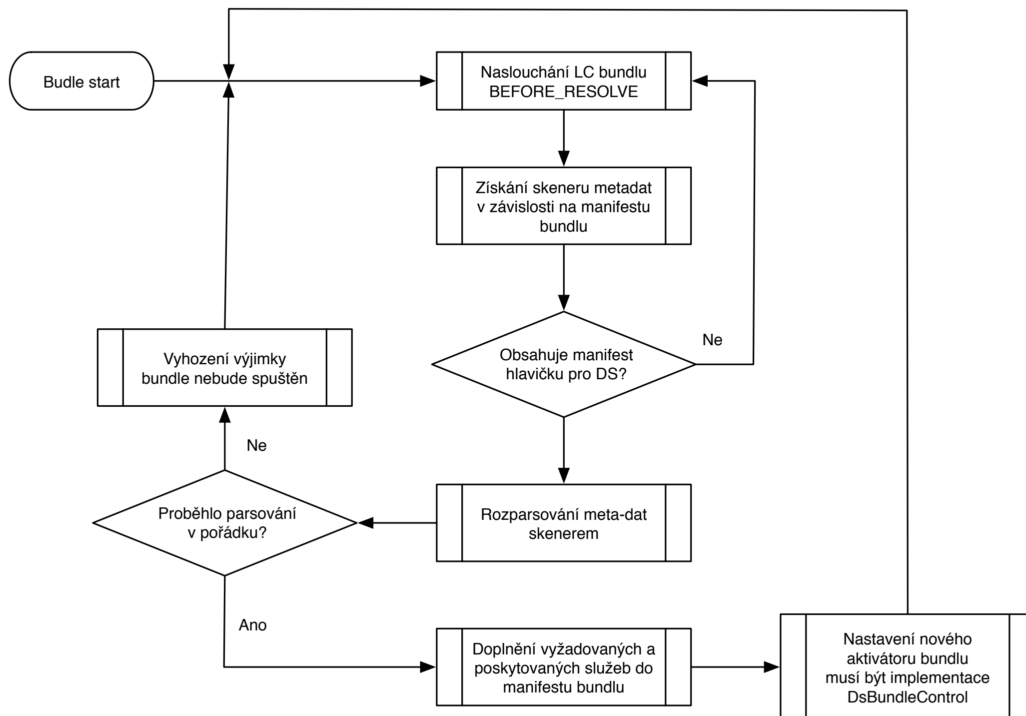
- `Metadata` entita v sobě sdružuje kolekci entit `Provide`. Entity `Provide` popisují, jaké služby jsou do systému bundlem registrovány. Atributy, které tato entita obsahuje, vzešly z analýzy aktuálního procesu registrace služby. `Service` atribut je určen pro název rozhraní, které služba implementuje. Atribut `version` je volitelný a reprezentuje verzi služby. Volitelným atributem je i `name`, který v sobě nese informaci o názvu služby - význam této informace byl již diskutován výše. Dalším atributem je `method`, tento atribut popisuje, jaká metoda objektu (který je definován v entitě `Metadata` atributem `cls`) vrací instanci služby pro registraci.

- Pomocí metody `registerService()` lze k službě přidat i libovolný počet klíč-hodnota položek. Proto entita `Provide` v sobě obsahuje kolekci entit `Property`. Tato entita obsahuje pouze dva atributy `name` a `value`, jejichž význam je naprosto zjevný.

- `Metadata` entita taktéž obsahuje kolekci entit `Require`. Tato entita má za úkol deklarovat všechny služby, které bundle vyžaduje a z toho jsou i odvozeny její atributy. `Service` atribut určuje o jaký typ služby jde (o jaké rozhraní), atribut `filter` umožňuje aplikovat na výběr služby nově představený LDAP filtr, atribut `cardinality` nastavuje jednak povinnost této závislosti a v druhé řadě to, jestli je umožněno vložit i více služeb odpovídajících požadavkům. Další atributy jako `versionRange`, `name`, `bundleProvider`, `bundleVersionRange` jsou odvozeny od specifikace CoSi, kde je taktéž popsán jejich význam [3].

6.3.2 Popis chování rozšíření

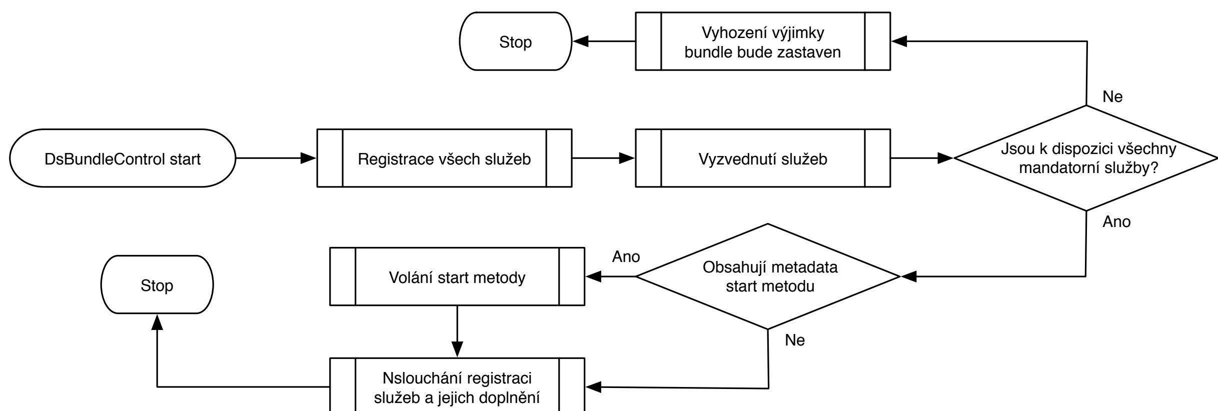
Rozšíření bude implementováno běžným CoSi bundlem. Tento bundle při spuštění zaregistruje posluchač, ve kterém bude provádět kontrolu všech bundlů, které přecházejí ze stavu REGISTERED do stavu RESOLVED. Během této kontroly bude zjišťovat, jestli bundle ve svém manifestu obsahuje hlavičku naznačující použití deklarativních služeb. Pokud ano, z obsahu této hlavičky připraví skener, který načte a rozparsuje meta-data bundlu. Z rozparsovaných dat doplní manifest bundlu tak, aby bylo později možné vyzvedávat a registrovat deklarované služby. Tento algoritmus popisuje



Obrázek 6.2: Vývojový diagram zpracování registrovaných bundlů.

vývojový diagram na obrázku 6.2.

Každý bundle, který chce využívat služeb deklarativních služeb, musí implementovat třídu, která bude sloužit jako centrální místo pro vkládání závislostí a vyzvedávání objektů poskytovaných služeb. Tato třída musí být odvozena od abstraktní třídy `DsBundleControl`. V této třídě se na základě načtených metadat provede automatická registrace služeb a poté i vyzvednutí požadovaných služeb. Tento proces ukazuje vývojový diagram na obrázku 6.3.



Obrázek 6.3: Vývojový diagram logiky v abstraktní třídě `DsBundleControl`.

6.3.3 Požadované funkce jádra

Tato část práce uvádí seznam nezbytných funkcí, které musí jádro CoSi poskytovat, aby bylo možno navržené rozšíření implementovat.

Možnost naslouchat životnímu cyklu bundlů

Rozšíření musí být schopno reagovat na události v životním cyklu bundlů. Důležité jsou události na úrovni přechodu mezi stavy INSTALLED a RESOLVED, kdy se bude provádět načítání metadat.

Možnost upravit metadata bundlu

Aby vývojář nemusel uvádět redundantní informace ohledně poskytovaných a vyžadovaných služeb v meta-datech deklarativních služeb a zároveň v manifestu bundlu, bude nutné měnit meta-data (manifest) bundlu.

Možnost naslouchat životnímu cyklu služeb

Deklarativní služby umožňují pomocí kardinality nastavit některé závislosti na službách jako volitelné a dále i jako vícenásobné (myšleno ve významu, že bundle umí pracovat s více službami stejného typu). Proto je nutné naslouchat službám a v průběhu jejich registrace je vkládat do bundlů, u kterých nebyly tyto služby dostupné v okamžiku jejich spuštění.

6.4 Implementace rozšíření

Rozšíření je implementováno jako samostatný bundle, který má následující manifest:

```
1  Cosi-Version: 2.0
2  Bundle-Name: CoSi - Declarative Services extension
3  Bundle-Version: 1.0.0
4  Bundle-Name: CoSi - Declarative Services extension
5  Bundle-Provider: Jakub Trunecek
6  Provide-Services: cz.zcu.kiv.cosi.bundles.ds.RegistryService
7  Provide-Types: cz.zcu.kiv.cosi.bundles.ds.impl.DsBundleControl
8  Bundle-Classpath: .;lib/jyaml-1.3.jar
9  Control-Class: cz.zcu.kiv.cosi.bundles.ds.impl.Activator
```

Odchytávání registrovaných bundlů se provádí v aktivátoru:

```
1  public class Activator implements BundleControl, VetoableBundleListener...
2      public void start(BundleContext context) throws Exception {
3          context.addBundleListener(this);
4          ...
5      }
```

```
6
7       public void bundleChanged(BundleEvent event) throws BundleListenerException {
8           try {
9               if (event.getType() == BundleEvent.BEFORE_RESOLVE) {
10                  Scanner scanner = ScannerFactory.getScaner(event.getBundle());
11                  if (scanner == null) {
12                      // Tento bundle nepouziva DS
13                      return;
14                  }
15                  // Nacteni metadat
16                  Metadata metadata = scanner.scan();
17
18                  // Vlozeni metadat do manifestu
19                  ...
20              }
21          } catch (Exception ex) {
22              throw new BundleListenerException("Error during scanning bundle", event, ex);
23          }
24          ...
25      }
```

Z pohledu implementace je zajímavé, jak je navrženo načítání metadat a jak ho lze případně rozšířit. Zo popisuje následující podsekce. Další zajímavou částí je samotné vkládání závislostí, které je obsaženo v další podsekci.

6.4.1 Načítání metadat

Načítání metadat je realizováno pomocí skeneru, který je předepsán následujícím rozhraním:

```
1 public interface Scanner {
2     public Metadata scan() throws ScannerException;
3 }
```

Aktuální verze rozšíření obsahuje dvě implementace tohoto rozhraní:

- `cz.zcu.kiv.cosi.bundles.ds.impl.scanner.XmlScanner`
- `cz.zcu.kiv.cosi.bundles.ds.impl.scanner.YamlScanner`

První implementace načítá meta-data z XML pomocí nativního XML parseru v balíku `javax.xml`. Druhá implementace je určena pro načítán meta-dat z YAML souboru. K tomu byla použita knihovna jyaml. Jednotlivé implementace nejsou příliš zajímavé a jsou k dohledávání přímo ve zdrojových kódech. Důležité však je, jak rozhodnout, který skener použít. K tomu slouží třída `ScannerFactory`:

```
1 public final class ScannerFactory {
2     public static final String MANIFEST_PROPERTY = "Declarative-Services";
3
4     private ScannerFactory() {
5     }
```

```
6
7      public static Scanner getScaner(Bundle bundle) {
8          String line = bundle.getBundleMetadata().getManifestLine(MANIFEST_PROPERTY);
9          if (line == null) {
10             return null;
11         }
12         // Check the extension
13         String[] lineItems = line.split("\\.");
14         String extension = lineItems[lineItems.length - 1];
15         if (extension.equalsIgnoreCase("xml")) {
16             return createXmlScanner(line, bundle);
17         } else if (extension.equalsIgnoreCase("yml")
18                 || extension.equalsIgnoreCase("yaml")) {
19             return createYamlScanner(line, bundle);
20         } else {
21             return null;
22         }
23     }
24
25     private static Scanner createXmlScanner(String path, Bundle bundle) {
26         return new XmlScanner(bundle.getResource(path));
27     }
28
29     private static Scanner createYamlScanner(String path, Bundle bundle) {
30         return new YamlScanner(bundle.getResource(path));
31     }
32 }
```

Pokud by v budoucnu bylo potřeba přidat další formát, stačí aby byl implementován příslušný skener a byla upravena třída `ScannerFactory` ve smyslu tohoto skeneru.

6.4.2 Realizace vkládání závislostí a registrace služeb

Registrace a vyhledávání služeb musí být prováděno z kontextu cílového bundlu nikoliv z bundlu deklarativních služeb. Aby toho bylo dosaženo, je potřeba, aby bundle používající deklarativní služby implementoval třídu, která je odvozena od abstraktní třídy `DsBunldeControl`. Tato třída přepisuje metody start() a stop() na finální a v nich provádí samotné registrace, od-registrace a vyhledávání služeb. To vše provádí na základě načtených meta-dat, které se mezi bundly přenášejí pomocí služby deklarativních služeb. Následující kód uvádí implementaci této abstraktní třídy na úrovni hlaviček metod:

```
1  public abstract class DsBundleControl implements BundleControl...
2      private RegistryService service;
3      private BundleContext context;
4      private Metadata metadata;
5      private Collection<Object> registeredServices = new ArrayList<Object>();
6
7      public final void start(BundleContext context) throws Exception {
8          service = (RegistryService) context
```

```
9                 .getService(RegistryService.class.getName());
10            this.context = context;
11            metadata = service.getMetadata(context.getBundle());
12            start();
13        }
14
15        public final void stop(BundleContext context) throws Exception {
16            stop();
17        }
18
19        private void start() throws Exception {
20            // validace metadat
21            validateMetadata();
22            // registrace poskytovanych sluzeb
23            registerProvides();
24            // vlozeni zavislosti
25            injectRequires();
26            // volani start metody existuje-li
27            callStart();
28        }
29
30        private void stop() throws Exception {
31            // volani stop metody existuje-li
32            callStop();
33            // odregistrace vsech zaregistrovanych sluzeb
34            unregisterProvides();
35        }
36    }
```

6.5 Zhodnocení funkčnosti rozšíření

Funkčnost rozšíření byla ověřena úpravou ukázkové aplikace, tuto úpravu popisuje první část této sekce. Další podsekce názorně ukazuje přínos rozšíření porovnáním původního a nového aktivátoru GUI bundlu ukázkové aplikace. Poslední podsekce hodnotí, zda byly splněny všechny požadavky kladené na rozšíření v úvodu této kapitoly.

6.5.1 Úpravy ukázkové aplikace

V ukázkové aplikaci byly vybrány dva bundly, u kterých se provedla reimplementace na deklarativní služby. Tato reimplementace je poměrně triviální, spočívá ve změně aktivátoru, manifest souboru a v přidání nového souboru s deklaracemi pro rozšíření.

První byl přepracován bundl poskytující překladový slovník s pomocí služby `slovnik.seznam.cz`. Aktivátor byl přepsán následovně:

```
1  public class Activator extends DsBundleControl {
2      public DictionaryService getDictionaryService() {
3          return new DictionaryServiceImpl();
```

```
4       }
5   }
```

Důležitou změnou je, že aktivátor již dále není implementací rozhraní `BundleControl`, ale implementací abstraktní třídy `DsBundleControl`. Aby mohla být v bundlu tato třída použita, je třeba vyžadovat její typ v manifest souboru, ten je upraven do následující podoby:

```
1  Cosi-Version: 1.0
2  Bundle-Name: CoSi - Dictionary service based on Seznam.cz DS
3  Bundle-Version: 2.0.0
4  Bundle-Classpath: .;lib/jsoup-1.6.2.jar
5  Bundle-Provider: Jakub Trunecek
6  Declarative-Services: META-INF/services.xml
7  Require-Types:
8      cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService;
9      cz.zcu.kiv.cosi.bundles.ds.impl.DsBundleControl
```

Je zjevné, že odpadla deklarace závislostí (`Require-Services`) a deklarace poskytovaných služeb (`Provide-Services`), na místo toho byla přidána hlavička pro deklarativní služby (`Declarative-Services`). Popis meta-dat pro deklarativní služby bylo v tomto případě realizováno pomocí XML a vypadá následovně:

```
1  <!DOCTYPE bundle PUBLIC "COSI//DC" "...">
2  <bundle
3      class="cz.zcu.kiv.cosi.bundles.dictionary.seznamcz.Activator">
4          <provides
5              service="cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService"
6              version="1.0.0"
7              method="getDictionaryService">
8                  <property name="nativeLanguage" value="cs" />
9                  <property name="name" value="seznam.cz" />
10         </provides>
11 </bundle>
```

Po spuštění takto upraveného bundlu se vše chová jako před úpravou, důkazem je výpis požadovaných a poskytovaných služeb do systému:

```
1  Starting the system...
2  Initializing container
3
4  [Warning] Header Provide-Interfaces is deprecated, use Provide-Services instead
5  [Warning] Header Provide-Interfaces is deprecated, use Provide-Services instead
6  Type 'help' for help
7  System shell
8  >interfaces 9
9  Provided interfaces of bundle cz.zcu.kiv...seznamcz-2.0.0.jar (id 9)
10 --------------------------------------------------------
11 cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService
12
13 >interfaces 9 -r
14 Consumed interfaces by bundle cz.zcu.kiv...seznamcz-2.0.0.jar (id 9)
15 --------------------------------------------------------
16 cz.zcu.kiv.cosi.bundles.ds.RegistryService
```

Dalším upraveným bundlem bylo samotné grafické rozhraní bundlu, na této úpravě lze demonstrovat přínos tohoto rozšíření a je popisováno v následující podsekci.

6.5.2 Viditelný přínos

Nejprve je potřeba uvést podobu původního aktivátoru bundlu pro grafické rozhraní aplikace:

```
public class Activator implements BundleControl,
        ServiceListener {
    private Gui gui;
    private BundleContext context;
    public void start(BundleContext P_context) throws Exception {
        context = P_context;
        gui = new Gui();
        ServiceReference[] services = context
            .getServiceReferences(DictionaryService.class.getName(),
                new HashMap<String, String>());
        for (ServiceReference s : services) {
            gui.addService((DictionaryService) context.getService(s),
                s.getProperties().get("name"));
        }
        gui.setVisible(true);
        context.addServiceListener(this,
            "(objectClass=" + DictionaryService.class.getName() + ")");
    }
    public void stop(BundleContext P_context) throws Exception {
        gui.dispose();
    }
    public void serviceChanged(ServiceEvent event) {
        if (event.getType() == ServiceEvent.REGISTERED) {
            gui.addService((DictionaryService) context
                        .getService(event.getServiceReference()),
                    event.getServiceReference().getProperties().get("name"));
        } else {
            gui.removeService((DictionaryService)
                context.getService(event.getServiceReference()));
        }
    }

}
```

a také jeho manifest:

```
Bundle-Name: CoSi - Dictionary application NODS
Bundle-Version: 1.0.0
Bundle-Classpath: .;lib/swing-layout-1.0.3.jar
Bundle-Provider: Jakub Trunecek
Require-Services:
    cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService;optional=true
Control-Class: cz.zcu.kiv.cosi.bundles.dictionary.app.Activator
Require-Types: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService
```

Je patrné, že vývojář bundlu musí nejprve deklarovat všechny závislosti v manifestu a poté si v aktivátoru služby vyzvednout. Navíc si pomocí životního cyklu služeb musí pohlídat případné do-registrace služeb. Nyní práce ukazuje změnu aktivátoru s použitím deklarativních služeb:

```
1  public class Activator
2          extends DsBundleControl implements ServiceListener {
3      private Gui gui;
4      private BundleContext context;
5
6      public void doStart(BundleContext P_context) {
7          gui = new Gui();
8          gui.setVisible(true);
9      }
10     public void doStop(BundleContext P_context) {
11         gui.dispose();
12     }
13     public void addDictionaryService(DictionaryService service, ServiceReference sr) {
14         gui.addService(service, sr.getProperties().get("name"));
15     }
16 }
```

Aby takto aplikace fungovala, je potřeba dodat deklaraci meta-dat (pro ukázku použit YAML):

```
1  class: cz.zcu.kiv.cosi.bundles.dictionary.app.Activator
2  startMethod: doStart
3  stopMethod: doStop
4  requires:
5    - service: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService
6      method: addDictionaryService
7      cardinality: 0..n
```

A nakonec je potřeba změnit manifest bundlu:

```
1  Bundle-Name: CoSi - Dictionary application DS
2  Bundle-Version: 2.0.0
3  Bundle-Classpath: .;lib/swing-layout-1.0.3.jar
4  Bundle-Provider: Jakub Trunecek
5  Declarative-Services: META-INF/services.yml
6  Require-Types: cz.zcu.kiv.cosi.bundles.dictionary.services.DictionaryService;
7      cz.zcu.kiv.cosi.bundles.ds.impl.DsBundleControl
```

Je tedy zjevné, že použitím rozšíření je velmi markantně zjednodušen aktivátor bundlu a dále již není potřeba v manifestu bundlu uvádět závislosti, ale pouze odkaz na meta-data pro rozšíření. Až v těchto meta-datech jsou uvedeny závislosti a poskytované služby, které jsou rozšířením doplněny do manifestu bundlu. Další výhodou je uvedená kardinalita, které se postará i o do-registrované služby a není potřeba registrovat vlastní posluchač.

6.5.3 Splnění požadavků

Rozšíření *musí být* navrženo a implementováno jako běžný CoSi bundle. Tento požadavek byl splněn. Použití deklarativních služeb je dobrovolné a jeho volba je čistě v kompetenci vývojáře bundlu.

Rozšíření *musí* umožnit deklarativní cestou registraci a vyzvedávání služeb z kontejneru. Pro vývojáře jsou aktuálně k dispozici dva formáty pro zápis deklarací vyžadovaných služeb. V kombinaci s novým aktivátorem je zajištěno vložení požadovaných služeb i registrace poskytovaných služeb čistě na základě deklarace.

Rozšíření *musí* dodržet zpětnou kompatibilitu umožňující klasické registrování a vyzvedávání služeb. Vzhledem k tomu, že deklarativní služby jsou implementovány jako klasický bundl, není ohrožena zpětná kompatibilita. Použití rozšíření je dobrovolné a lze i nadále použít klasický způsob práce se službami.

Rozšíření *by mělo* omezit deklarace na nezbytné minimum. Tento požadavek byl splněn tím, že není potřeba redundantně deklarovat závislosti a registrace služeb v matedatech rozšíření a v manifestu bundlu. Rozšíření se na základě meta-dat automaticky postará o doplnění manifestu za běhu.

Rozšíření *by mělo* umožnit definovat mandatorní a nemandatorní závislosti. Pomocí kardinality lze nastavit, zda je vyžadovaná služba povinná či nikoliv. Pokud v okamžiku spouštění bundlu nejsou k dispozici všechny jeho povinné závislosti, není tento bundle spuštěn.

Rozšíření *by mělo* umožnit dynamické vkládání vícenásobných závislostí. Pomocí nového životního cyklu služeb a jeho událostí je do rozšíření za-implementována podpora pro vícenásobné závislosti. Na úrovni meta-dat je tato funkčnost vystavena pomocí kardinality, která umožňuje deklarovat nejen povinnost závislosti, ale i její vícenásobnost.

Shrnutí

Rozšíření splňuje veškeré požadavky, které na něj byly kladeny. Zároveň zjevně zjednodušuje práci se službami a tím splňuje hlavní motivaci pro jeho implementaci.

Kapitola 7

Rozšíření CoSi o AOP

Jak již bylo uvedeno v teoretické části práce, existují takzvané *protínající potřeby*, které nelze oddělit běžnou aplikací objektově orientovaného přístupu. V komponentových modelech platí tento problém dvojnásob. Není možné do bundlu, který je vyvinut třetí stranou, jednoduše doplnit například transakční zpracování metod nebo logování pomocí knihovny, kterou bude využívat celá aplikace. Proto je namístě navrhnout a implementovat základní podporu AOP prvků do frameworku CoSi a umožnit tak standardizovaný způsob pro oddělení těchto protínajících potřeb například do samostatných bundlů.

Následující kapitola v úvodu formuluje jasné požadavky na rozšíření a dále se věnuje jeho návrhu. V návrhu je stěžejní volba metody vplétání, která pak předurčuje další rozhodnutí. Mezi ně patří například návrh typů přípojných bodů, možnosti řezů programem atd. V další části práce jsou předvedeny zajímavé implementační části práce jako například implementace speciálního jazyka pro zápis řezů programem. V poslední části kapitoly se zhodnocuje funkčnost implementovaného rozšíření opět pomocí ukázkové aplikace.

7.1 Formulace zadání

Rozšíření aplikačního rámce CoSi o prvky AOP formulují následující požadavky:

1. *musí* být navrženo a implementováno minimálně jako systémový CoSi bundl,
2. *musí* definovat a implementovat kompletní model přípojných bodů na úrovni vykonávání metod nad registrem služeb,
3. *musí* definovat a implementovat kompletní model řezů programem (*pointcut*),
4. *musí* definovat a implementovat intuitivní způsob zápisu řezů programem,

5. *musí* umožnit seskupovat řezy programem a pokyny do aspektů,

6. *musí* umožnit registraci/od-registraci aspektů do systému a tím zajistit jejich vplétání/odstranění,

7. *by mělo* umožnit budoucí vývoj zápisu řezů programem.

7.2 Návrh rozšíření

V první řadě je nutné zvolit způsob vplétání pokynů, který bude nejlépe odpovídat možnostem CoSi. Dále, na základě této volby, bude možné specifikovat další stěžejní body spojené s aspektově orientovaným programováním - zejména model přípojných bodů a formu zápisu řezů programem.

7.2.1 Volba způsobu vplétání

Pro výběr volby vplétání jsou k dispozici tři varianty definované v teoretické části práce, jsou jimi:

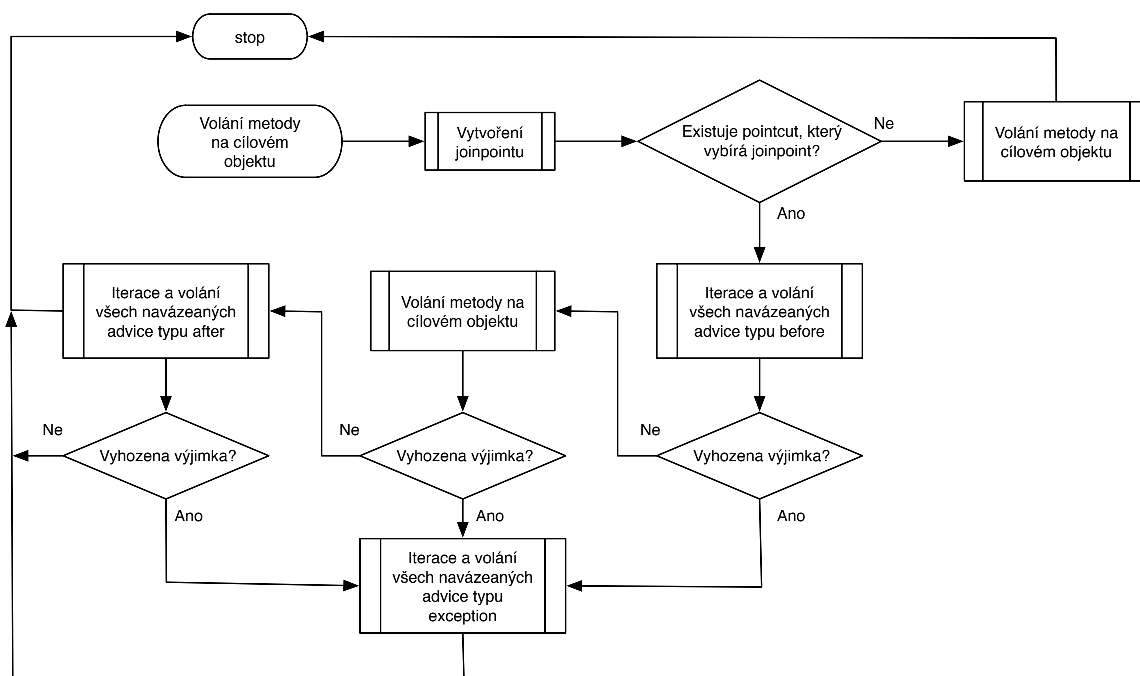
1. vplétání při kompilaci,

2. vplétání při načítání tříd,

3. vplétání při běhu.

První možnost - *vplétání při kompilaci* - je z pohledu komponentových systému problematická. Komponenty jsou zkompilované balíky, které se skládají do výsledných aplikačních celků, a není tedy možné do nich žádným způsobem vplést pokyny při kompilaci. Tento způsob vplétání, i přesto že je v otázce výkonu aplikace nejlepším, použít nelze.

Druhá možnost - *vplétání při načítání tříd* - je na tom lépe. Komponentový model CoSi obsahuje vlastní implementaci zavaděče tříd, který by bylo možné upravit tak, aby před načtením třídy provedl vpletení pokynů. Prvním problémem je, že zavaděč tříd by musel vplétání realizovat na úrovni byte kódu, to je řešitelné pomocí externích Java knihoven. Druhým, kritičtějším problémem, je fakt, že bundle obsahující aspekty může být do systému zaveden kdykoliv, tedy i v okamžiku, kdy již ostatní bundly mají načtené třídy, které právě nainstalovaný bundle může měnit svými pokyny. V takovém případě by bylo potřeba tyto bundly nějakou formou restartovat, aby znovu požádaly o načtení tříd. To by v CoSi nebylo možné řešit jinak, než kompletní změnou politiky služeb a ve výsledku i kompletní změnou specifikace. To je nepřípustné a proto je nutné vyloučit i tuto možnost.



Obrázek 7.1: Vývojový diagram popisující fungování proxy třídy.

Poslední možnost - *vplétání při běhu* - klade jistá omezení. Nelze vplétat pokyny do objektů, které jsou instanciovány klíčovým slovem *new*, jinými slovy musí být vplétání realizováno za pomocí kontejneru, ze kterého se instance získávají (jedná se o formu návrhového vzoru *Service Locator*). Dále tato forma klade největší nároky na výkon a umožňuje použití pouze přípojných bodů na úrovni volání metody. Na druhou stranu se však téměř ideálně hodí pro nároky AOP rozšíření. Vplétání bude realizováno pouze nad službami v registru služeb a metoda je dostatečně snadná na implementaci. Metoda vplétání tedy byla stanovena na ***vplétání při běhu za pomoci proxy tříd.***

Proxy třídy

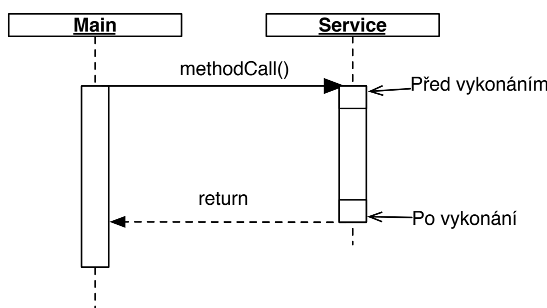
Proxy třída je obecný výraz pro třídu, která nějakým způsobem překrývá chování původní třídy. Jinými slovy proxy objekt naslouchá voláním metod a ty pak případně deleguje je na skrytý objekt. Před nebo po této delegaci může provést různé úpravy argumentů, návratové hodnoty, nebo dokonce může úplně vynechat volání původní metody. Java obsahuje nativní podporu proxy tříd v balíku `java.lang.reflect`, které bude využito pro vplétání. Nejprve je nutné implementovat rozhraní `InvocationHandler` a následně vytvořit objekt proxy třídy pomocí metody `Proxy.newProxyInstance()`. Použití tohoto mechanismu demonstruje následující ukázka:

```
public interface Foo {
    Object bar(Object obj) throws BazException;
}
public class FooImpl implements Foo..
    Object bar(Object obj) throws BazException..


public class DebugProxy implements InvocationHandler {
    private Object obj;
    public static Object newInstance(Object obj) {
        return java.lang.reflect.Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new DebugProxy(obj));
    }
    private DebugProxy(Object obj) {
        this.obj = obj;
    }
    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        Object result;
            try {
                System.out.println("before␣method␣" + m.getName());
                result = m.invoke(obj, args);
            } catch (InvocationTargetException e) {
                throw e.getTargetException();
            } catch (Exception e) {
                throw new RuntimeException("unexpected␣invocation␣exception:␣" +
                    e.getMessage());
            } finally {
                System.out.println("after␣method␣" + m.getName());
            }
            return result;
    }
}

...
Foo foo = (Foo) DebugProxy.newInstance(new FooImpl());
foo.bar(null);
```

V AOP rozšíření se metoda `invoke()` v proxy třídě bude chovat podle zobrazeného vývojového diagramu na obrázku 7.1. Nejprve vytvoří přípojný bod a ten zkonfrontuje se všemi evidovanými řezy programem. Pokud nenajde žádný, který by daný přípojný bod identifikoval pokračuje na obyčejné volání původní metody a končí svou činnost. Pokud však takový řez existuje, projdou se všechny pokyny, které jsou na něj navázány a mají být provedeny před voláním metody a provedou se. V případě že při těchto voláních nastane výjimka, projdou se pokyny „typu výjimka" vykonají se a tím je volání skončeno. Pokud výjimka nenastane, pokračuje se obdobně na volání původní metody a následně na volání pokynů, které se mají vykonat po volání původní metody.



Obrázek 7.2: Volání pokynu před a po vykonání metody.

Aby vše mohlo fungovat musí framework CoSi umožnit vytváření proxy tříd při registraci služby a zároveň poskytnout možnost zaregistrovat místo původní služby právě vytvořený proxy objekt. K tomu bude sloužit v této práci nově představený koncept *hooku na úrovni registrace služby.*

7.2.2 Přípojné body

Přípojné body teoreticky mohou být jakýkoliv bod ve vykonávání programu, například cyklus, podmínka, vytvoření instance, volání metody či vykonávání metody. Zvolený způsob vplétání a zadání omezují možné přípojné body na jeden - přípojný bod na úrovni vykonávání metody. Jedná se o bod, ve kterém se běh programu nachází v okamžiku, kdy bylo provedeno volání libovolné metody nad libovolným objektem. Pokyny v tomto případě lze aplikovat před, po a okolo tohoto přípojného bodu. Sekvenční diagram na obrázku 7.2 ukazuje příklad volání pokynu před a po přípojném bodu.

Přípojné body v rozšíření pro CoSi jsou dány následujícími rozhraními:

```
1  public interface JoinPoint {
2      Object[] getArguments();
3      Method getMethod();
4      Class getTarget();
5  }
6
7  public interface BeforeJoinPoint extends JoinPoint {
8      void setArguments(Object[] arguments);
9      public void setResultAndAvoidInvocation(Object object);
10     public Object getResult();
11 }
12
13 public interface AfterJoinPoint extends JoinPoint {
14     public Object getResult();
15     public void setResult(Object object);
16 }
17
18 public interface ExceptionJoinPoint extends JoinPoint {
```

```
19      public Throwable getThrowable();
20      public void setThrowable(Throwable t);
21  }
```

Z těchto rozhraní vyplývají i jejich možnosti. Obecný přípojný bod se vždy týká konkrétní třídy (`getTarget()`), konkrétní metody (`getMehotd()`) a argumentů, se kterými probíhá volání (`getArguments()`).

Odvozený přípojný bod před vykonáním pokynu navíc umožňuje změnit argumenty metody (`setArguments()`), také umožňuje zastavit následující vykonávání původní metody nastavením návratové hodnoty (`setResultAndAvoidInvocation()`) a samozřejmě obsahuje přístupovou metodu pro takto nastavenou hodnotu (`getResult()`).

Přípojný bod po vykonávání pokynu je rozšířen o možnost získání návratové hodnoty (`getResult()`) a její změnu (`setResult()`).

Speciální přípojný bod používaný v okamžiku výskytu výjimky umožňuje přístup k vyhozené výjimce (`getThrowable()`) a zároveň umožňuje její změnu (`setThrowable()`).

7.2.3 Řezy programem a jejich zápis

Řez programem má za úkol identifikovat určitý přípojný bod, jeho rozhraní je tedy jasné:

```
1  public interface Pointcut {
2      public boolean match(JoinPoint point);
3  }
```

Aby nebylo nutné jednotlivé řezy ručně „programovat" implementací tohoto rozhraní, je navržen jednoduchý jazyk pro popis řezů programem, který je dán následující gramatikou:

```
1  pointcut    : IDENTIFIER "(" args ")"
2              | pointcut "||" pointcut
3              | pointcut "&&" pointcut
4              | "!" pointcut
5              | "(" pointcut ")"
6              ;
7
8  args        : PATTERN moreargs
9              |
10             ;
11
12 moreargs    : "," nextarg
13             |
14             ;
15
16 nextarg     : PATTERN moreargs
17             ;
```

```
18
19 PATTERN     = [\[\]a-zA-Z0-9_\.\*;]+
20 IDENTIFIER  = [a-zA-Z][a-zA-Z0-9_]+
```

Tato gramatika nám umožňuje například následující zápis:

```
1 interface(*.dictionary.services.DictionaryService) && return(*String[])
```

Kde identifikátory `interface` a `return` jsou implementace typizovaných řezů programem, které budou představeny v implementační části.

Při implementaci jazyka je nutné počítat s možností rozšíření formou nových před-implementovaných řezů, tak aby nebylo třeba přepisovat vymyšlené řešení, ale aby stačilo jednoduše dopsat typizovaný řez a přidat ho do nějaké statické deklarace poskytovaných řezů.

7.2.4 Zápis aspektu

Zápis aspektu by měl být co nejintuitivnější a pokud možno co nejpodobnější běžným konstruktům jazyka Java. Vzhledem k tomu, že nebude implementován vlastní kompilátor, je nasnadě k zápisu aspektů použít klasické Java třídy s anotacemi. Následuje ukázka aspektu:

```
1  @Aspect
2  public class LoggingAspect {
3      private final Log logger;
4
5      public LoggingAspect(Log logger) {
6          this.log = log;
7      }
8
9      @Pointcut("interface(*)␣&&␣method(*)")
10     public void allMethodsPointCut() {}
11
12     @Before("allMethodsPointCut")
13     public void logAdviceBefore(BeforeJoinPoint jp) {
14         log.debug("Entering␣method␣" + jp.getMethod().getName());
15     }
16
17     @After("allMethodsPointCut")
18     public void cacheAdviceAfter(AfterJoinPoint jp) {
19         log.debug("Leaving␣method␣" + jp.getMethod().getName());
20     }
21 }
```

Bude třeba implementovat parser, který anotace převede na meta-informace, které budou poskytnuty proxy třídám.

7.2.5 Registrace aspektů

Libovolné bundly budou mít možnost registrovat vlastní aspekty. Je nutné připravit službu, která umožní tuto registraci. Služba má následující rozhraní:

```
1 public interface AopService {
2     public void registerAspect(Object aspect) throws AopException;
3     public void unregisterAspect(Object name) throws AopException;
4 }
```

Na rozdíl služeb není potřeba deklarovat registrované aspekty v manifestu bundlu.

7.3 Implementace rozšíření

Implementace AOP byla provedena poměrně velkým objemem kódu, proto v rámci práce budou uvedeny pouze zajímavé části této implementace Další implementační detaily lze dohledat přímo ve zdrojových kódech na přiloženém CD.

V první podsekci je uvedena struktura a implementace různých typů řezů programem a následně i implementace jazyka pro jejich zápis. Další podsekce ukazuje implementaci vplétání kódu za pomocí hook mechanismu. A poslední podsekce popisuje, jak a proč byla implementována možnost vypnutí podpory AOP.

7.3.1 Model řezů programem a jazyk pro jejich zápis

Nejprve byl implementován jednoduchý hierarchický model primitivních řezů programem:

- `cz.zcu.kiv.cosi.container.aop.pointcut.And`
- `cz.zcu.kiv.cosi.container.aop.pointcut.Or`
- `cz.zcu.kiv.cosi.container.aop.pointcut.Not`

První dvě třídy implementují logické operátory mezi dvěma řezy, například `And` je implementován takto:

```
1 public class And implements Pointcut {
2     protected Pointcut left;
3     protected Pointcut right;
4
5     public And(Pointcut left, Pointcut right) {
6         this.left = left;
7         this.right = right;
8     }
9
10    public boolean match(JoinPoint point) {
11        return left.match(point) && right.match(point);
```

```
12        }
13    }
```

Dále byly implementovány jednoduché typizované řezy, které byly pojmenovány jako funkce (balík `cz.zcu.kiv.cosi.container.aop.pointcut.functions`):

- `ArgsFunction` - umožňuje vybrat metodu na základě typů jejích argumentů.
- `ClassFunction` - umožňuje vybrat objekt na základě názvu třídy.
- `InterfaceFunction` - umožňuje vybrat objekt na základě implementovaného rozhraní.
- `MethodFunction` - umožňuje vybrat metodu na základě jejího názvu.
- `ReturnFunction` - umožňuje vybrat metodu na základě návratového typu.
- `ThrowsFunction` - umožňuje vybrat metodu na základě typu výjimky, kterou vyhazuje.

Pro názornost je uvedena ukázková implementace třídy `ClassFunction`:

```
1   public class ClassFunction implements Function {
2       protected String pattern;
3       private Pattern regPattern;
4
5       public void addArgument(String pattern) {
6           this.pattern = pattern;
7       }
8
9       protected Pattern getPattern() {
10          if (this.regPattern == null) {
11              String pattern = "^" + this.pattern
12                  .replace(".", "\\.")
13                  .replace("[", "\\[")
14                  .replace("*", "[a-zA-Z0-9_\\.]+") + "$";
15              this.regPattern = Pattern.compile(pattern);
16          }
17          return this.regPattern;
18      }
19     public boolean match(JoinPoint point) {
20         return getPattern()
21             .matcher(point.getTarget().getName())
22             .matches();
23      }
24  }
```

Dále byl připraven jazyk pro zápis řezů programem, pro jeho vývoj byl použit nástroj JFlex pro vygenerování lexikálního analyzátoru a nástroj Byacc pro syntaktický analyzátor a pro vytvoření objektové reprezentace výrazu. Výsledkem je, že z výrazu:

```
1   interface(*.DictionaryService) && return(*String[])
```

je vytvořena následující hierarchie objektů:

```
cz.zcu.kiv.cosi.container.aop.pointcut.And{
    left = cz.zcu...functions.InterfaceFunction{
        pattern = *.DictionaryService
    },
    right = cz.zcu...functions.ReturnFunction{
        pattern = *String[]
    }
}
```

Předpisy pro nástroje JFlex a Byacc jsou součástí přiložených programových kódů na CD.

7.3.2 Vplétání pokynů

Vplétání pokynů je implementováno pomocí nativního mechanismu proxy tříd (třída `ProxyClass` v balíku `cz.zcu.kiv.cosi.container.aop`):

```java
public class ProxyClass implements InvocationHandler...

    public static Object create(ClassLoader classLoader,
            AspectRegistry registry, Object from) {
        ...
        // vytvori instanci pomoci Proxy.newProxyInstance metody
    }

    public Object invoke(Object o, Method method, Object[] os) throws Throwable {
        JoinPoint joinPoint = new JoinPointImpl(target.getClass(), method, os);
        Advice[] advices = registry.getAdvices(joinPoint);
        // Vykonani pokynu pred
        Collection<BeforeAdvice> before = getAdvices(advices, BeforeAdvice.class);
        BeforeJoinPoint bjp = new BeforeJoinPointImpl(joinPoint);
        for (BeforeAdvice a : before) {
                a.executeBefore(bjp);
        }

        Object res = bjp.getResult();
        try {
            // Volani maskovane metody
            if (res == null) {
                res = method.invoke(target, bjp.getArguments());
            }
        } catch (Throwable e) {
            // Exception pokyn
            Collection<ExceptionAdvice> exception =
                getAdvices(advices, ExceptionAdvice.class);
            ExceptionJoinPoint ejp = new ExceptionJoinPointImpl(joinPoint, e);
            for (ExceptionAdvice a : exception) {
                a.execute(ejp);
            }

            if (ejp.getThrowable() != null) {
                throw  ejp.getThrowable();
            }
```

```
37          }
38
39          // Vykonani pokynu po
40          Collection<AfterAdvice> after = getAdvices(advices, AfterAdvice.class);
41          AfterJoinPoint ajp = new AfterJoinPointImpl(joinPoint, res);
42          for (AfterAdvice a : after) {
43              a.executeAfter(ajp);
44          }
45          return ajp.getResult();
46      }
47
48      private static <T extends Advice> Collection<T>
49              getAdvices(Advice[] advices, Class<T> cls) {
50         ...
51         // vybere z kolekce jen polozky urciteho typu
52      }
53
54  }
```

Všechny služby jsou skryty za instancí této třídy a při každém volání metody je prováděn kód v metodě `invoke()`, který postupně provádí volání všech navázaných pokynů.

Stále je však nutné zajistit vytváření proxy objektu v okamžiku registrace, k tomu slouží implementace hooku na úrovni registrace služeb, který je spojen s aktivátorem systémového bundlu pro AOP:

```
1  package cz.zcu.kiv.cosi.container.bundles.aopservice.impl;
2
3  public class Activator implements BundleControl, RegisterHook {
4
5      private AspectRegistry registry;
6      private AopService service;
7      public Activator() {
8          registry = new AspectRegistryImpl();
9          service = new AopServiceImpl(registry);
10     }
11
12     public void start(BundleContext context) throws Exception {
13         context.registerService(RegisterHook.class.getName(), this);
14         context.registerService(AopService.class.getName(), service);
15     }
16
17     public void stop(BundleContext P_context) throws Exception {
18         context.unregisterService(this);
19         context.unregisterService(service);
20     }
21
22     public ServiceReference register(BundleContext bundle,
23             ServiceReference serviceReference, Object service) {
24         ClassLoader classLoader =
25             (ClassLoader)((BundleImpl) bundle
26                 .getBundle()).getBundleClassLoader();
27
28         // Vsechny sluzby jsou skryty za proxy
```

```
29          Object serviceProxy = ProxyClass.create(classLoader, registry, service);
30          ServiceReference sr = new ServiceReferenceImpl(serviceReference.getClazz(),
31              serviceProxy, serviceReference.getProvidingBundle(),
32              serviceReference.getProperties(),
33              serviceReference.getServiceMetadata());
34          return sr;
35      }
36
37  }
```

Stěžejní část se nachází v metodě `register()`, kde je prováděno vytváření proxy třídy při každé registraci nové služby.

7.3.3 Možnost vypnutí AOP

Protože AOP bundle vytváří pro každou službu proxy objekt a při každém volání metody na službách je navíc tento volána metoda `invoke()` tohoto objektu, je do frameworku zanesena permanentní výkonová zátěž. V případě, kdy nejsou použity prvky AOP, je tato zátěž zbytečná. Bylo tedy umožněno AOP rozšíření vypnout konfigurací kontejneru, která se provádí v souboru `cosi.config`:

```
1  # ...
2  AOP_ENABLED=false
```

7.4 Zhodnocení funkčnosti AOP

Stejně jako u předchozího rozšíření byla funkce prokázána úpravou ukázkové aplikace, o které pojednává první část této sekce. Na závěr je opět zhodnoceno splnění formulovaných požadavků.

7.4.1 Úprava ukázkové aplikace

Ukázková aplikace stahuje data z internetových stránek jednotlivých slovníků. Toto stahování může nastávat poměrně často a bylo by vhodné ukládat mezivýsledky do vyrovnávací paměti (cache). Do aplikace tedy byla přidána implementace cachování výsledků pomocí aspektu. Byl přidán bundl, který má následující podobu manifest souboru (bundle taktéž využívá deklarativních služeb):

```
1  Bundle-Name: CoSi - Dictionary service caching based on AOP
2  Bundle-Provider: Jakub Trunecek
3  Bundle-Version: 1.0.0
4  Declarative-Services: META-INF/services.yml
5  Require-Types: cz.zcu.kiv.cosi.bundles.ds.impl.DsBundleControl
6  Archiver-Version: Plexus Archiver
7
```

```
//services.yml
class: cz.zcu.kiv.cosi.bundles.dictionary.caching.Activator
stopMethod: doStop
requires:
  - service: cz.zcu.kiv.cosi.core.aop.service.AopService
    method: setAopService
    cardinality: 1..1
```

Aktivátor bundlu je velmi jednoduchý, pouze registruje implementovaný aspekt:

```java
public class Activator extends DsBundleControl {
    AopService service;
    CachingAspect aspect;
    public void setAopService(AopService service) {
        this.service = service;
        aspect = new CachingAspect(new MemoryCacheStorage());
        service.registerAspect(aspect);
    }
    public void doStop(BundleContext context)  {
        service.unregisterAspect(aspect);
    }
}
```

Na závěr byl implementován jednoduchý aspekt, který realizuje samotné ukládání do vyrovnávací paměti:

```java
@Aspect
public class CachingAspect {
    private final CacheStorage storage;
    public CachingAspect(CacheStorage storage) {
        this.storage = storage;
    }

    @Pointcut("interface(*.dictionary.services.DictionaryService)␣&&␣return(String[])")
    public void dictionaryServiceArrayReturningMethods() {}

    @Before("dictionaryServiceArrayReturningMethods")
    public void cacheAdviceBefore(BeforeJoinPoint jp) {
        if (storage.get(createKey(jp)) != null) {
            System.out.println("-␣value␣loaded␣from␣cache");
            jp.setResultAndAvoidInvocation(storage.get(createKey(jp)));
        } else {
            System.out.println("-␣value␣is␣not␣in␣chace");
        }
    }

    @After("dictionaryServiceArrayReturningMethods")
    public void cacheAdviceAfter(AfterJoinPoint jp) {
        if (storage.get(createKey(jp)) == null) {
            System.out.println("-␣value␣was␣cached!");
            storage.save(createKey(jp), jp.getResult());
        }
    }

    private String createKey(JoinPoint point) {
        ...
        // vytvari unikatni klic pro join point
```

```
32        }
33
34  }
```

Uvedený řez programu vybírá všechny volání metod nad rozhraním `DictionaryService` vracející pole typu `String` a aplikuje na ně pokyn před a po. Pokyn vykonávaný před voláním kontroluje, zdali výsledek již není obsažen ve vyrovnávací paměti. Pokud ano nastaví ho jako návratovou hodnotu a tím zamezí samotnému volání původní metody. Pokyn, který se vykonává po volání metody, kontroluje, zda se v paměti nachází hodnota pro dané volání. Pokud tomu tak není, znamená to, že hodnota ještě nebyla uložena a proto ji uloží.

Funkčnost úpravy a tím i AOP je ověřena výstupem aplikace při zadání stejného slova k přeložení dvakrát po sobě:

```
1  Starting the system...
2  Initializing container
3
4  [Warning] Header Provide-Interfaces is deprecated, use Provide-Services instead
5  [Warning] Header Provide-Interfaces is deprecated, use Provide-Services instead
6  Type 'help' for help
7  System shell
8  >
9  - value is not in chace
10 - value was cached!
11 - value loaded from cache
```

7.4.2 Splnění požadavků

Rozšíření *musí* být navrženo a implementováno minimálně jako systémový CoSi bundle. Rozšíření nebylo možné implementovat jako běžný bundle, protože je potřeba, aby AOP bundle byl zaveden úplně jako první a mohl tak vplétat pokyny i do ostatních systémových bundlů. AOP tedy bylo implementováno jako systémový bundle a tím byl tento požadavek splněn.

Rozšíření *musí* definovat a implementovat kompletní model přípojných bodů na úrovni vykonávání metod nad registrem služeb. Vplétání je realizováno použitím techniky proxy tříd, kterými jsou maskovány objekty registru služeb. Rozšíření implementuje přípojné body při vykonávání metod, jak bylo požadováno, a dále umožňuje vykonávání pokynů před, okolo a po daném typu přípojného bodu. Zároveň i poskytuje pokyn, který je možno aplikovat v případě vyhození výjimky.

Rozšíření *musí* definovat a implementovat kompletní model řezů programem (*pointcut*). Model řezů programem je navržen a implementován jako hierarchie triviálních

řezů, které je možné vyskládat do libovolně složité kombinace. Tím je poskytnut velmi flexibilní a pro účely AOP rozšíření kompletní model řezů programem.

Rozšíření *musí* definovat a implementovat intuitivní způsob zápisu řezů programem. Pro vyskládání jednoduchých řezů programem do složitých celků byl navržen a implementován jazyk, který je svou podobou dostatečně výmluvný a intuitivní.

Rozšíření *musí* umožnit seskupovat řezy programem a pokyny do aspektů. Tento požadavek je realizován použitím Java 5 anotací, které umožňují vývojáři jednoduše napsat aspekt obsahující pokyny a navázat je na řezy programem.

Rozšíření *musí* umožnit registraci/od-registraci aspektů do systému a tím zajistit jejích vplétání/odstranění. Možnost registrace a od-registrace aspektů je umožněna bundlům pomocí služby, kterou AOP bundle poskytuje. Každý bundle, který chce AOP použít, musí deklarovat závislost na této službě a po startu ji vyzvednout a zaregistrovat do ní svoje aspekty.

Rozšíření *by mělo* umožnit budoucí vývoj zápisu řezů programem. Jazyk pro zápis řezů programem je navržen tak, že se jednoduše dají doplňovat libovolné „funkce”, které pak pomocí třídy `FunctionFactory` mohou být pojmenovány a následně mohou být použity ve výrazech bez nutnosti úpravy lexikální či syntaktické analýzy jazyka.

Kapitola 8

Závěr

Jedním z hlavních cílů práce bylo navrhnout a implementovat rozšíření komponentového rámce CoSi o dependency injection. Toto rozšíření vychází z ověřené specifikace podobné funkcionality v OSGi nazvané *declarative services*. Pro CoSi je implementováno samostatným bundlem, bylo nazváno *deklarativní služby* a jeho hlavním přínosem je markantní zjednodušení práce se službami - tedy zjednodušení registrace a vyhledávání služeb. Celý proces návrhu a implementace včetně ověření funkčnosti a názorné ukázky přínosu tohoto rozšíření popisuje kapitola 6.

Druhým hlavním cílem bylo navrhnout a zakomponovat možnost použití prvků přístupu AOP. Teoretický základ pro toto rozšíření, na jehož základě byl vybrán nejvhodnější způsob implementace, je předložen v kapitole 3, na základě kterého byl vybrán nejvhodnější způsob implementace. Proces návrhu a implementace popisuje kapitola 7. Během implementace byl navrhnut jazyk pro zápis řezů programem a celý model přípojných bodů. Dále byla na ukázkové aplikaci ověřena funkčnost a použitým scénářem byl nastíněn směr, pro který je technika AOP vhodná.

V průběhu návrhu stěžejních částí práce byly objeveny jisté nesrovnalosti v aktuální implementaci a také byly identifikovány potřebné funkčnosti, které CoSi neposkytovalo. Kapitola 5 popisuje veškeré úpravy, které proběhly mimo hlavní rámec práce, přesto však byly potřebné ať už z pohledu celkové konzistence systému nebo z důvodu potřeby pro implementovaná rozšíření.

Celkově práce splnila vytyčené cíle, aplikační rámec nyní obsahuje nový životní cyklus služeb, nový registr služeb, mechanismus hooků, použitelný LDAP filr pro identifikaci služeb, deklarativní způsob vyhledávání a registrace služeb a možnost používat AOP techniky. Zároveň práce poskytla nezbytný teoretický úvod do všech dotčených problematik, kterými byly znovupoužitelnost kódu, komponentové modely, dependency injection a aspektově orientované programování.

8.1 Návrhy na další rozšíření

Díky nutnosti analyzovat a podstatně doplnit velkou část implementace jádra frameworku získal autor práce značný přehled o dalších úpravách, které by rámec mohly dále vylepšit.

V první řadě by bylo vhodné sjednotit programovací konvence, které jsou aktuálně trochu různorodé. Takové sjednocení by velmi prospělo výsledné čitelnosti kódu a snazší údržbě.

Náhodou bylo při realizaci této práce zjištěno, že pokud je bundle implementován pomocí skriptovacího jazyka Groovy a poskytuje typy, nelze je načíst a použít v bundlech, které jsou implementovány v Javě. Tento problém je nejspíše jen obtížně řešitelný, možná by ale bylo vhodné analyzovat, zda se přeci jen tento problém nedá odstranit.

Dalším směrem, kterým by se evoluce komponentového modelu CoSi mohla vydat, je například chytřejší práce se stromem závislostí mezi bundly tak, aby framework byl schopen bundly startovat a vypínat pořadí, ve kterém by byly uspokojeny přítomné závislosti - *dependency resolving.*

Dále byly konzultovány návrhy na změnu specifikace ve směru dynamičnosti instalace, spouštění a vypínání bundlů spolu s registrací služeb. Navrhované změny by však až příliš měnily základní myšlenku CoSi a bylo od nich z tohoto důvodu upuštěno.

Seznam použitých zkratek

AOP	Aspect oriented programing
API	Application Programming Interface
CBD	Component Based Development
CBSE	Component Based Software Engineering
CMS	Content Management System
CoSi	Components Simplified
DI	Dependency Injection
EJB	Enterprise Java Beans
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JAR	Java Archive
Java EE	Java Enterprise Edition
Java ME	Java Micro Edition
Java SE	Java Standard Edition
JDBC	Java Database Connectivity
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JSF	Java Server Faces

JSP	Java Server Pages
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
OOP	Object Oriented Programing
OSGi	Open Services Gateway Initiative
POJO	Plain Old Java Object
VM	Virtual Machine
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Seznam obrázků

2.1 Komponentově založený návrhový vzor [převzato z 5] 4
2.2 Životní cyklus a) stateless beany, b) statefull beany 8
2.3 vrstvy OSGi frameworku, zdroj [16] 11
2.4 Životní cyklus bundlu v OSGi, převzato z [16] 13
2.5 Meta model CoSi (bez mimo-funkčních charakteristik), převzato z [3] . 15
2.6 Životní cyklus bundlu v CoSi, převzato z [3] 16

3.1 a) Systém bez AOP b) systém s AOP (převzato z [11]). 19
3.2 Kompozice výsledného systému z koncernů, převzato z [11] 20
3.3 Sekvenční diagram znázorňující nejběžnější join pointy, převzato z [11]. 26

4.1 Závislosti při vytváření instancí uvnitř objektu. 31
4.2 Závislosti při užití DI. 33
4.3 Závislosti při užití vzoru Service Locator. 35
4.4 Ilustrace modularity za použití Guice frameworku, převzato z [18]. . . . 36
4.5 Funkce Spring IoC kontejneru, převzato z [22]. 37

5.1 Rozložení ukázkové aplikace do bundlů. 57
5.2 Ukázka grafického rozhraní aplikace. 58
5.3 Neaktivní grafické rozhraní. 61

6.1 Návrh meta-modelu deklarativních služeb. 67
6.2 Vývojový diagram zpracování registrovaných bundlů. 69
6.3 Vývojový diagram logiky v abstraktní třídě `DsBundleControl`. 69

7.1 Vývojový diagram popisující fungování proxy třídy. 80
7.2 Volání pokynu před a po vykonání metody. 82

Literatura

[1] BACHMAN, F. Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, 2000.

[2] BRADA, P. Aspektová terminologie. Dostupné z: `http://blogspot.zcu.cz/clanky/89/aspektova-terminologie`. [online], [cit. 2012-05-08]].

[3] BRADA, P. – WAJTR, B. – LIšKA, V. The CoSi Component Model. Dce/tr-2008-07, University of West Bohemia, Department of Computer Science and Engineering, 2008.

[4] BURKE, B. – MONSON-HAEFEL, R. *Enterprise JavaBeans 3.0*. Sebastopol, CA : O'Reilly, 2006. ISBN 9780596009786 059600978X.

[5] CRNKOVIC, I. *Building reliable component-based software*. Boston : Artech House, 2002. ISBN 978-1580533270.

[6] ECLIPSE. Equinox Aspects – Quick-start guide, . Dostupné z: `http://www.eclipse.org/equinox/incubator/aspects/equinox-aspects-quick-start.php`. [online], [cit. 2012-04-29].

[7] ECLIPSE. AspectJ Documentation, . Dostupné z: `http://www.eclipse.org/aspectj/docs.php`. [online], [cit. 2012-04-28].

[8] FOWLER, M. Inversion of Control Containers and the Dependency Injection pattern. Dostupné z: `http://martinfowler.com/articles/injection.html`. [online], [cit. 2012-05-01].

[9] GOOGLE. Google Guice - User's guide, . Dostupné z: `http://code.google.com/p/google-guice/wiki/Motivation`. [online], [cit. 2012-04-30].

[10] GOOGLE. Peaberry - dynamic service extension for Google-Guice, . Dostupné z: `http://code.google.com/p/peaberry/`. [online], [cit. 2012-04-30].

[11] LADDAD, R. *AspectJ in action : practical aspect-oriented programming*. Greenwich : Manning, 2003. ISBN 1930110936 9781930110939.

[12] LIšKA, V. Pokročilé aspekty komponentových modelů. Master's thesis, Západočeská univerzita v Plzni, 2009.

[13] MICROSOFT. LDAP Query Basics. Dostupné z: `http://technet.microsoft.com/en-us/library/aa996205(v=exchg.65).aspx`. [online], [cit. 2012-05-01].

[14] OSGI. About OSGi Alliance, . Dostupné z: `http://www.osgi.org/About/HomePage`. [online], [cit. 2012-04-20].

[15] OSGI. *OSGi Service Platform Compendium Specification*. Release 4. The OSGi Alliance, 2009.

[16] OSGI. *OSGi Service Platform Core Specification*. Release 4, Version 4.2. The OSGi Alliance, 2009.

[17] OSGI. OSGi technology, . Dostupné z: `http://www.osgi.org/About/Technology`. [online], [cit. 2012-04-22].

[18] PRASANNA, D. R. Google I/O 2009 - Big Modular Java with Guice. Dostupné z: `http://www.youtube.com/watch?v=hBVJbzAagfs`. [online], [cit. 2012-04-30].

[19] SOMMERVILLE, I. *Software engineering*. 8. vydání. Harlow, England; New York : Addison-Wesley, 2007. ISBN 7-111-19770-4.

[20] SPRING. Aspect Oriented Programming with Spring, . Dostupné z: `http://static.springsource.org/spring/docs/2.5.x/reference/aop.html`. [online], [cit. 2012-04-28].

[21] SPRING. Spring Dynamic Modules Reference Guide, . Dostupné z: `http://static.springsource.org/osgi/docs/1.1.3/reference/html/`. [online], [cit. 2012-04-29].

[22] SPRING. The IoC container, . Dostupné z: `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html`. [online], [cit. 2012-04-30].

[23] WIKIPEDIA. Code reuse - Wikipedia, The Free Encyclopedia, 2012. Dostupné z: `http://en.wikipedia.org/w/index.php?title=Code_reuse&oldid=487984995`. [online], [cit. 2012-04-20].

[24] WIKIPEDIA. OSGi Service Platform. Dostupné z: http://cs.wikipedia.org/wiki/OSGi_Service_Platform. [online], [cit. 2012-04-20].