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

Diplomová práce

Optimalizace a rozšíření porovnávání OSGi komponent

Plzeň 2012 Zuzana Burešová

Prohlášení

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

V Plzni dne 16. května 2012

Zuzana Burešová

Abstract

The comparator of Java types and OSGi components, which was developed on the Department of Computer Science and Engineering of the University of West Bohemia, can check compatibility of OSGi components and prevent errors from running incompatible components in a component framework. There are many application opportunities of this functionality, but currently only a few superstructural tools are ready for real usage. The goal of this diploma thesis is to spread possibility of real usage of the comparator by two means. The first targets improving comparator usability on performance-limited devices by optimizations of its memory consumption and computation time. Secondly, the work aims at enhancing functionality of the superstructural tools to provide more benefits for users. The implemented enhancement allows manual component difference analysis by providing graphical presentation of the differences.

Obsah

1 Úvod **1**
- 1.1 Výchozí stav . . . 1
- 1.2 Cíl projektu . . . 2
- 1.3 Stručný přehled . . . 2

2 Komponenty **4**
- 2.1 Koncept . . . 4
  - 2.1.1 Úvod . . . 4
  - 2.1.2 Komponenta . . . 4
  - 2.1.3 Komponentový model . . . 5
  - 2.1.4 Komponentový framework . . . 6
  - 2.1.5 Problémy . . . 6
- 2.2 Technologie OSGi . . . 6
  - 2.2.1 OSGi specifikace . . . 6
  - 2.2.2 OSGi komponenta . . . 7
  - 2.2.3 OSGi framework . . . 8
  - 2.2.4 Životní cyklus OSGi komponenty . . . 9
  - 2.2.5 Aktivátor OSGi komponenty . . . 10
  - 2.2.6 Interakce mezi OSGi komponentami . . . 10
- 2.3 Sémantické verzování a resolving v OSGi . . . 10
  - 2.3.1 Resolving . . . 10
  - 2.3.2 Sémantické verzování . . . 11
  - 2.3.3 Problémy resolvingu a sémantického verzování . . . 11
  - 2.3.4 Další problémy resolvingu . . . 12

3 Nahraditelnost komponent **13**
- 3.1 Použití . . . 13
- 3.2 Výsledky a směr výzkumu na katedře . . . 13
  - 3.2.1 Použití komparátoru při resolvingu . . . 13
  - 3.2.2 Podpora pro bezpečnou nahraditelnost komponent . . . 14
  - 3.2.3 Podpora pro správné sémantické verzování . . . 14
  - 3.2.4 Další náměty pro použití komparátoru . . . 15
- 3.3 Teorie nahraditelnosti komponent . . . 15
  - 3.3.1 Relace subtypu mezi komponentami . . . 15
  - 3.3.2 Jak lze relaci subtypu zjistit . . . 16
  - 3.3.3 Metody ověření nahraditelnosti . . . 16
- 3.4 Komparátory Java tříd a OSGi komponent . . . 16
  - 3.4.1 Projekty pro získání reprezentace . . . 17

3.4.2 Projekty komparátorů získané reprezentace 17
3.4.3 Závislosti mezi projekty 18
3.4.4 Další projekty 18
3.5 Architektura komparátorů 19
3.5.1 Reprezentace Java typů 19
3.5.2 Architektura načítání reprezentace 19
3.5.3 Architektura komparátorů 19
3.5.4 Agregace výsledku porovnání 20
3.6 Problémy a náměty pro další práci 21
3.6.1 Generické typy 21
3.6.2 Porovnávání tříd 21
3.6.3 Ověřování kompatibility dvou komponent 21
3.6.4 Další nedostatky 22
3.7 Implementované opravy a vylepšení 22
3.7.1 Úpravy hierarchie Java typů 23
3.7.2 Pomoc při thread-safety analýze 23
3.7.3 Úpravy struktury výsledků porovnání 23
3.7.4 Vylepšení algoritmu porovnání tříd 23
3.8 Technologie a vývojové nástroje 24
3.8.1 Java 24
3.8.2 OSGi 24
3.8.3 Apache Maven 24
3.8.4 Assembla 25
3.8.5 Hudson Continuous Integration 25
3.9 Způsob práce v týmu 25

4 Využití komparátorů na malých zařízeních 27
4.1 Malá zařízení 27
4.1.1 Android 27
4.1.2 Využití OSGi na platformě Android 27
4.1.3 Využití komparátorů na platformě Android 28
4.2 Předchozí práce 28
4.3 Úkol této práce 29
4.3.1 Metodika 29
4.4 Výkonnostní testy 29
4.4.1 Účel výkonnostních testů 29
4.4.2 Vytváření výkonnostních testů 30
4.5 Analýza kódu 31
4.5.1 Nástroje použité pro analýzu komparátorů 31
4.6 Optimalizace 32
4.6.1 Měření efektu optimalizací 32
4.6.2 Úroveň optimalizace 33
4.7 Výkonnost Java aplikací 33
4.7.1 Doba výpočtu 33
4.7.2 Spotřeba paměti 34
4.7.3 Garbage kolektor 35
4.8 Automatizované metody měření výkonnostních parametrů Java aplikací 37
4.8.1 Metody měření doby výpočtu 37
4.8.2 Metody měření spotřeby paměti 38

4.8.3 Jak ovládat garbage kolektor . . . 41
4.9 Implementace měření výkonnostních parametrů komparátoru . . 42
4.9.1 Účel implementace . . . 42
4.9.2 Výstup testovací aplikace . . . 43
4.9.3 Architektura . . . 43
4.9.4 Popis některých metod důležitých rozhraní . . . 44
4.9.5 Implementace objektů typu Tester . . . 44
4.9.6 Implementace objektů typu PerfTestCase . . . 45
4.9.7 Implementace objektů typu ResultsGatherer . . . 45
4.9.8 Implementace řídících objektů . . . 46
4.9.9 Pomocné třídy a knihovny . . . 46
4.9.10 Integrace do Hudsonu . . . 47
4.9.11 Ověření přesnosti měření . . . 47
4.10 Provedené optimalizace . . . 47
4.10.1 Oprava logování . . . 48
4.10.2 Optimalizace algoritmu porovnávání tříd . . . 48
4.10.3 Optimalizace algoritmu porovnání seznamu metod . . . 48
4.10.4 Celkový efekt optimalizací . . . 49
4.11 Zhodnocení výsledků . . . 50

5 Využití komparátorů při analýze rozdílů 51
5.1 Cíle a přínosy . . . 51
5.1.1 Grafické zobrazení rozdílů . . . 51
5.1.2 Další možné použití výstupu . . . 51
5.2 Projekt OBVS . . . 52
5.2.1 Architektura . . . 52
5.2.2 Způsob integrace zobrazení rozdílů . . . 53
5.3 Technologie . . . 54
5.3.1 HTML . . . 54
5.3.2 CSS . . . 54
5.3.3 Javascript . . . 54
5.3.4 JQuery . . . 55
5.3.5 Treeview . . . 56
5.4 Projekt prezentace rozdílů . . . 56
5.4.1 Výstupy . . . 56
5.4.2 Implementace . . . 58
5.4.3 Možnosti rozšíření . . . 59
5.5 Přínosy a další využití . . . 61

6 Závěr 62
6.1 Shrnutí práce . . . 62
6.2 Přínosy . . . 62
6.2.1 Přínosy pro komparátor . . . 62
6.2.2 Osobní přínosy . . . 63
6.3 Náměty pro další práci . . . 63
6.3.1 Optimalizace načítání reprezentace . . . 63
6.3.2 Nové nástroje . . . 63

1 Úvod

1.1 Výchozí stav

Koncept rozdělení aplikace na více komponent, které fungují nezávisle a komunikují vzájemně podle daných pravidel, se v posledních desetiletích velmi rozšířil. Důvodem je především vyšší rychlost vývoje aplikace složené z komponent. Komponenty, podobně jako objektově orientovaný přístup, přispívají k lepší organizaci kódu a zapouzdření jednotlivých funkčností. Obzvlášt' pro velké aplikace je tato další úroveň členění výhodou, díky které se kód stává čitelnější a udržovatelnější.

Největšího urychlení vývoje aplikace lze dosáhnout použitím již hotových částí kódu. Při volbě komponentového přístupu lze využít hotový komponentový framework, který zajišt'uje komponentám běhové prostředí a poskytuje mnoho užitečných služeb. K dispozici je také velké množství hotových komponent, které je možné do aplikace snadno začlenit. Nově naprogramované komponenty bude možné znovu použít v budoucnu.

V dnešní době jsou nároky na kvalitu aplikací velmi vysoké, proto je nedílnou součástí vývojového procesu testování. Díky komponentovému přístupu lze testování výrazně urychlit a zjednodušit. Není nutné při každé změně testovat celou aplikaci, ale pouze upravenou komponentu. Po otestování komponenty samotné je ale nutné ověřit také její kompatibilitu s ostatními komponentami v systému. Pokud by do systému byla vložena nekompatibilní komponenta, hrozí pád celé aplikace. Proto je riskantní nahradit komponentu za běhu systému, ačkoliv to některé komponentové frameworky umožňují.

Na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni probíhá vývoj nástrojů, které dokážou kompatibilitu komponent automaticky ověřit. Výzkum je zaměřen především na platformu OSGi (Open Services Gateway initiative). Ověření probíhá na základě porovnání kompletního rozhraní komponenty oproti rozhraní vyžadovanému, nebo oproti rozhraní nahrazované komponenty. Díky tomu lze dosáhnout vyšší spolehlivosti, než jaké je dosaženo testováním. Další výhodou je úspora času na vytváření testů. Tato metoda je také vhodná k zajištění bezpečného nahrazení komponenty za běhu systému.

Nástroje vyvíjené v rámci výzkumu na katedře lze rozdělit na komparátor typů jazyka Java (JaCC - Java Class Compatibility checker) a komparátor OSGi komponent (OBCC - OSGi Bundle Compatibility Checking toolset), který je nadstavbou nástroje JaCC. Kromě vývoje těchto dvou základních nástrojů, které se používají ve formě knihoven, se další výzkumné projekty zaměřují také na jejich reálné využití. Příkladem je nástroj pro automatické verzování OSGi komponent nebo integrace kontroly kompatibility do frameworku Apache Felix.

1.2 Cíl projektu

Cílem projektu je rozšířit možnosti reálného využití komparátorů. K tomu byly zvoleny dva směry. Prvním je využití komparátoru na výkonově omezených (např. mobilních) zařízeních. Druhý směr se zaměřuje na rozšíření funkčnosti aplikačních nástrojů, díky kterému se zvýší přínosy a rozsah možného použití nástrojů pro uživatele.

Složitost aplikací pro malá (mobilní) zařízení stále roste, proto se i zde dobře uplatňuje komponentový přístup. Výkonnost těchto zařízení je ale nižší než výkonnost běžných počítačů. Aplikace pro malá zařízení proto musí hospodařit úsporně s pamětí, které bývá na takových zařízeních málo, a pro zajištění dostatečně rychlé odezvy je potřeba optimalizovat také čas výpočtu. Prvním úkolem této práce je změřit výkonnost komparátorů, analyzovat možnosti optimalizací a tyto optimalizace implementovat.

Druhá část práce se zabývá rozšířením nástroje pro automatické verzování OSGi komponent o grafické zobrazení rozdílů mezi původní a nově verzovanou (upravenou) komponentou. V současné době nemusí být uživateli jasné, z jakého důvodu je nutné konkrétní část verze zvýšit. Po implementaci rozšíření budou důvody jasně patrné a snadno dohledatelné. Uživatel bude moci na první pohled zhruba posoudit i kvantitu změn. Zobrazení rozdílů bude dále možné využít i pro další účely, například jako alternativa k nástroji SVN diff. Kromě grafického výstupu užitečného pro člověka bude rozšíření obsahovat také strojově čitelný výstup ve formátu XML.

1.3 Stručný přehled

V první kapitole byly vysvětleny důvody a cíl práce.

Druhá kapitola shrnuje principy komponentového přístupu v programování a detailněji se zaměřuje na platformu OSGi. V závěru kapitoly jsou vysvětleny příčiny některých problémů, které snižují přínosy komponentového přístupu.

Třetí kapitola se zabývá výzkumem, který výše popsané problémy řeší. Konkrétně jsou popsány projekty komparátorů vyvíjené na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. Stručně představena je také teorie nahraditelnosti komponent, ze které projekty vychází.

Ve čtvrté kapitole je popsána metodika výkonnostní analýzy a testování Java aplikací. Na základě této metodiky byla vytvořena testovací aplikace, která velmi přesně měří důležité výkonnostní charakteristiky Java aplikace. Metodika a testovací aplikace byly využity během optimalizací komparátorů, které jsou popsány v závěru kapitoly.

Pátá kapitola popisuje rozšíření nástroje pro automatické verzování OSGi komponent o grafické zobrazení rozdílů mezi komponentami.

V závěru jsou shrnuty dosažené výsledky a přínosy práce. Dále jsou uvedeny také náměty pro další práci.

2 Komponenty

2.1 Koncept

2.1.1 Úvod

Jak již bylo řečeno v úvodu práce, použití komponent při vývoji software přináší mnoho výhod. V této kapitole bude koncept komponent (Component-based software engineering, viz [11]) vysvětlen podrobněji a budou rozebrány také jeho problémy.

Komponentový přístup je založen na myšlence, že tvorba aplikace znamená skládání aplikace z jednotlivých nezávislých komponent, které jsou buď již hotové, nebo jejichž vývoj může probíhat odděleně. Ve skutečnosti ale není možné zajistit stoprocentní nezávislost komponent. Komponenty musí vzájemně spolupracovat, částečné nezávislosti se dociluje tím, že jejich interakce jsou omezené. Komponenty mohou mezi sebou komunikovat pouze v rámci pravidel daných komponentovým modelem a prostřednictvím svého rozhraní.

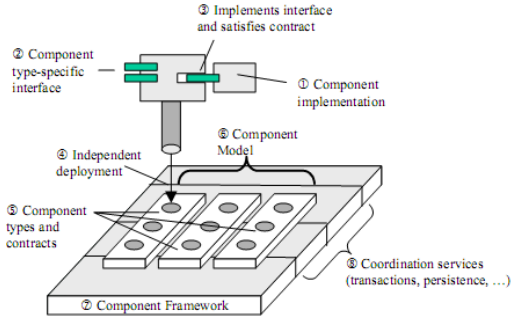
Existuje mnoho komponentových modelů pro různé programovací jazyky. Komponentový model standardizuje tvorbu komponent a díky tomu je možné používat komponenty různých výrobců. Pro výběr konkrétního modelu je důležité, aby byl model dostatečně oblíbený a tím pádem bylo k dispozici velké množství hotových komponent. Použitím hotových komponent se uspoří velké množství času při vývoji a testování software. S komponentovým modelem získáme také kvalitní základ architektury aplikace. Implementací modelu je komponentový framework. Pro některé modely je na výběr i více frameworků (viz [20]). Programátor se tedy nemusí zabývat běhovým prostředím a řízením komponent, ale zaměří se pouze na implementaci konkrétní komponenty.

2.1.2 Komponenta

Každá komponenta by měla implementovat určitou ucelenou funkčnost. Podle konceptu potom změna funkčnosti aplikace znamená přidání nebo nahrazení některé komponenty. Takto vytvářené komponenty lze také snadno nezávisle testovat, protože díky ucelené funkčnosti jsou interakce s okolím minimalizovány.

Na obrázku 2.1 je znázorněn vztah komponenty, jejího rozhraní, komponentového modelu a frameworku. Komponenta musí splňovat požadavky komponentového modelu. Díky tomu může komponentový framework komponenty ovládat a zprostředkovávat jejich interakce. Tyto interakce se realizují převážně jako služby. Komponenta může poskytovat služby ostatním komponentám a může také nějaké služby vyžadovat.

Z pohledu frameworku a ostatních komponent je komponenta popsána svým



Obrázek 2.1: Vztah komponenty, jejího rozhraní, komponentového modelu a frameworku. Obrázek byl převzat z [11]

rozhraním a případně mimofunkčními charakteristikami. Komponenty, které ji využívají, teoreticky neznají implementační detaily a nesmí se na ně spoléhat. Díky tomu je komponenta nahraditelná. Ostatní komponenty jsou závislé pouze na jejím rozhraní.

Komponenta může implementovat více různých rozhraní. Pro svou funkčnost obvykle potřebuje služby jiné komponenty, která implementuje potřebné rozhraní. Propojení komponent probíhá pouze na základě požadovaného rozhraní, klientská komponenta by neměla spoléhat na propojení s nějakou konkrétní komponentou.

Čím lépe jsou splněny popsané principy komponentového přístupu, tím více jsou vytvořené komponenty znovupoužitelné.

2.1.3 Komponentový model

Komponentový model definuje typy komponent a možnosti jejich vzájemných interakcí a interakcí s frameworkem. Specifikace modelu obsahuje také způsob zápisu rozhraní. To může být popsáno pomocí programovacího jazyka, speciálního jazyka pro definici rozhraní (IDL – Interface Definition Language) nebo v jiném formátu (např. manifest v OSGi).

Model určuje, co je to komponenta a jak má být připravena ke spuštění ve frameworku. Komponentou může být třída, objekt, balík nebo jiná část kódu aplikace. Popis podoby komponenty musí být dostatečně přesný, aby byl framework schopen komponentu spustit a dále s ní pracovat.

Pro vzájemnou interakci mezi komponentami definuje model způsob, jak komponenta může nabídnout služby ostatním komponentám a naopak jak může získat služby od ostatních komponent. Pro interakci komponenty s frameworkem model specifikuje způsob spuštění a následného řízení komponenty frameworkem a způsob volání služeb frameworku komponentou.

2.1.4 Komponentový framework

Komponentový framework je běhové prostředí pro komponenty. Obvykle představuje kostru aplikace a iniciuje jejich spuštění po svém startu. Framework implementuje konkrétní komponentový model a vyžaduje splnění modelu jednotlivými komponentami. Pokud by komponenta nesplňovala model, framework by s ní neuměl pracovat.

Pro komponenty je framework něco jako operační systém pro aplikace. Podobně jako operační systém, framework spouští některé komponenty hned po svém startu, jiné na příkaz uživatele. Framework řídí životní cyklus komponent, poskytuje mechanismus pro jejich interakce a spravuje sdílené zdroje. Komponenty mohou volat služby frameworku, podobně jako aplikace volají služby operačního systému.

2.1.5 Problémy

Existence mnoha komponentových modelů omezuje znovupoužitelnost hotových komponent. Komponentu lze použít pouze v aplikaci založené na stejném modelu, pro jaký byla vytvořena. Rozdrobenost komponent mezi jednotlivé modely ztěžuje hledání vhodné komponenty.

Dalším problémem je riziko nekompatibility po nahrazení komponenty. Nekompatibilita může vést k neočekávané funkčnosti nebo až k pádu aplikace. Aby se předešlo problémům, je nutné po nahrazení komponenty celou aplikaci testovat pomocí integračních testů. Pokud mají být tyto testy spolehlivé, zabere jejich vytvoření příliš mnoho času a úspora času díky komponentovému přístupu se snižuje.

Některé modely se problém snaží řešit pomocí systému verzí. Například v modelech DCE, CORBA a OSGi je použito sémantické verzování ve formátu: *major.minor.micro*. Tento systém je ale nespolehlivý, protože správné určení poskytované i vyžadované verze je ponecháno na programátorovi. Z těchto důvodů je nahrazení komponenty bez následných testů nebo do konce za běhu systému riskantní.

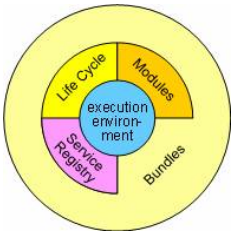
2.2 Technologie OSGi

2.2.1 OSGi specifikace

Technologie OSGi (Open Service Gateway initiative) je komponentový model pro jazyk Java. O její vývoj a propagaci se stará skupina s názvem OSGi Alliance [9], která vznikla v roce 1999. Skupina poskytuje nejen specifikaci modelu, ale také referenční implementaci.

Jádrem OSGi specifikace je OSGi framework. Jeho části jsou zobrazeny na

obrázku 2.2.



Obrázek 2.2: Schéma OSGi frameworku. Obrázek byl převzat z [9]

- *Execution environment:* specifikace běhového prostředí.
  OSGi lze použít na libovolné konfiguraci Java 2.
- *Modules:* specifikace načítání tříd (class loading).
  Při běžném načítání tříd v Javě existuje pouze jediný prostor tříd (classpath). V OSGi má každá komponenta svůj vlastní prostor tříd. Díky tomu je například možné spustit v jedné aplikaci najednou dvě třídy se shodným názvem. Oddělením prostorů tříd se zajišťuje také izolovanost komponent a jejich bezpečnost.
- *Life Cycle:* specifikace životního cyklu komponent.
  Komponenty v OSGi lze za běhu systému instalovat, spustit, zastavit, obnovit a odinstalovat. Pomocí API frameworku může tyto operace provádět některá z komponent.
- *Service Registry:* specifikace interakcí komponent pomocí služeb.
  Komponenty mohou do systému registrovat nabízené služby, po čase je mohou opět odregistrovat. V registru služeb mohou také vyhledávat služby ostatních komponent. V OSGi je služba reprezentována Java třídou.

2.2.2 OSGi komponenta

Komponenta se v terminologii OSGi nazývá bundle (viz [40]). Je to obyčejný Java archiv (JAR), který musí navíc obsahovat manifest s popisem komponenty ve formě několika speciálních hlaviček. Manifest komponenty je umístěn v JAR archivu na obvyklé cestě: META-INF/MANIFEST.MF. Obsah souboru se skládá z jednotlivých řádek ve formátu „klíč: hodnota“. V následujícím seznamu uvádím některé typické hlavičky OSGi manifestu:

- *Bundle-SymbolicName*
  Tato hlavička je povinná a udává název komponenty, který se používá jako identifikátor komponenty ve frameworku. Jako hodnota se obvykle používá reverzní doménová notace, jako pro Java balíky.

- *Bundle-Version*
  Tato hlavička označuje verzi komponenty. Pokud není specifikována, použije se defaultní hodnota *0.0.0* .
- *Bundle-Activator*
  Plně kvalifikované jméno třídy, která implementuje rozhraní BundleActivator. Aktivátor komponenty je volán frameworkem při startu a zastavení komponenty. Jeho implementace není povinná.
- *Export-Package*
  Seznam exportovaných balíků. Tyto balíky jsou viditelné pro ostatní komponenty, které si balíky importují. Jména balíků se oddělují čárkou a za jménem lze uvést i verzi balíku. Ta se uvádí za středník ve formátu „version=verze“.
- *Import-Package*
  Seznam importovaných balíků. Tyto balíky komponenta potřebuje pro svůj běh. Framework se pokusí balíky importovat z jiných komponent. Pokud se to nepovede, komponenta nemůže být spuštěna. Formát zápisu je stejný jako pro hlavičku Export-Package, ale kromě určité verze balíku lze uvést i přípustný rozsah verzí (více v [40]).
- *Require-Bundle*
  Tuto hlavičku lze použít k importu všech exportovaných balíků konkrétní komponenty. Zároveň se vytváří závislost na této konkrétní komponentě (závislost na implementaci). Její používání není doporučeno, protože se tím porušují základní principy komponentového přístupu.

Příklad jednoduchého manifestu:

```
Bundle-Name: Example bundle
Bundle-SymbolicName: cz.zcu.kiv.obcc.exampleBundle
Bundle-Version: 1.0.0.R123
Bundle-Activator: cz.zcu.kiv.obcc.exampleBundle.Activator
Export-Package: cz.zcu.kiv.obcc.exampleBundle
Import-Package: cz.zcu.kiv.obcc, org.slf4j.impl;\
        version="[1.3.1, 1.4.0)"
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Z uvedeného manifestu lze vyčíst, že název komponenty čitelný pro člověka je „Example bundle“, ve frameworku je bundle identifikován názvem „cz.zcu.kiv.obcc.exampleBundle“ a verzí „1.0.0.R123“. Komponenta obsahuje aktivátor a exportuje balík „cz.zcu.kiv.obcc.exampleBundle“. Ke svému fungování vyžaduje balík „cz.zcu.kiv.obcc“ v libovolné verzi a balík „org.slf4j.impl“ ve verzi větší nebo rovné 1.3.1 a menší než 1.4.0. Komponenta může běžet pouze v prostředí JavaSE-1.6.

2.2.3 OSGi framework

Pro technologii OSGi existuje mnoho různých implementací komponentového frameworku. Některé z nich splňují OSGi specifikaci kompletně, jiné pouze z

části.

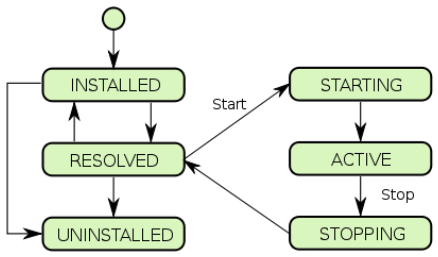
Příklady OSGi frameworků:

- *Apache Felix* [10]
  Open Source projekt, který se snaží implementovat OSGi specifikaci verze 4. Některé části specifikace ještě implementované nejsou.
- *Eclipse Equinox* [22]
  Také Open Source projekt, který je integrovaný s vývojovým prostředím Eclipse. Splňuje všechny požadavky OSGi specifikace verze 4 a vlastní certifikát OSGi Alliance.

Framework je typicky možné ovládat pomocí konzole (Felix, Equinox), některé frameworky poskytují i grafické rozhraní (Knopflerfish [41]).

2.2.4 Životní cyklus OSGi komponenty

Po startu aplikace se nejprve rozběhne framework, který následně spustí jednotlivé komponenty. Proces spouštění komponent není triviální, komponenta prochází postupně fázemi installed, resolved, starting a active. Životní cyklus OSGi komponenty zobrazuje obrázek 2.3.



Obrázek 2.3: Životní cyklus OSGi komponenty. Obrázek byl převzat z Wikipedie.

Ve fázi installed je komponenta načtena ve frameworku z disku. Ještě ale nemůže pracovat, protože nemá k dispozici požadované zdroje, definované například pomocí hlavičky manifestu Require-Package. Framework musí následně provést resolving, což je zjednodušeně řečeno proces provázání importů a exportů. Pokud proběhne resolving úspěšně, dostane se komponenta do fáze resolved. Nyní již má k dispozici vše, co potřebuje pro svou činnost. Framework ji může začít spouštět. Tím se komponenta dostává do fáze starting, ve které je

spuštěna startovací metoda jejího aktivátoru (viz kapitola 2.2.5). Plně spuštěná komponenta se dostane do fáze active.

Komponenta může být také zastavena nebo dokonce odinstalována frameworkem nebo jinou komponentou. Během fáze stopping je zavolána ukončovací metoda aktivátoru a komponenta přejde zpět do fáze resolved. Odtud lze komponentu odinstalovat ze systému (uninstalled).

2.2.5 Aktivátor OSGi komponenty

OSGi komponenta může obsahovat aktivátor, což je třída implementující rozhraní *org.osgi.framework.BundleActivator*. Toto rozhraní nařizuje implementaci metod start a stop, které jsou zavolány frameworkem při startu a zastavování komponenty. Parametrem metod je objekt třídy *org.osgi.framework.BundleContext*, který umožňuje interakci komponenty s frameworkem.

Plně kvalifikovaný název třídy aktivátoru musí být zapsána v manifestu v hlavičce Bundle-Activator.

2.2.6 Interakce mezi OSGi komponentami

Komponenty mohou spolupracovat pomocí sdílení Java balíků, které je realizováno zejména pomocí hlaviček manifestu Import-Package a Export-Package. Třídy z importovaných balíků může komponenta libovolně používat.

Volnější způsob spolupráce je realizován pomocí služeb. Komponenta může nabízet různé služby ostatním komponentám. Tyto služby jsou představovány Java objektem a do frameworku jsou registrovány obvykle pod svým rozhraním, případně i třídou. Ostatní komponenty můžou službu pomocí tohoto rozhraní vyhledat. Registrace a získání služeb může probíhat buď programově, například v metodě start aktivátoru komponenty, nebo pomocí XML. Doporučeným způsobem je deklarace služeb v XML. Komponenta může také reagovat na události registrace a deregistrace určité služby.

Díky tomu, že je služba reprezentována objektem Java třídy, nezpůsobuje její využívání téměř žádné režijní náklady.

2.3 Sémantické verzování a resolving v OSGi

2.3.1 Resolving

Resolving je proces párování zdrojů požadovaných a poskytovaných jednotlivými komponentami v systému. Je potřeba ho provést před spuštěním komponent, aby komponenty měli k dispozici veškeré zdroje, které potřebují pro svou

činnost. Resolving je úkolem frameworku, který vyhledává pro každý požadavek nějaký vyhovující zdroj podle daných pravidel.

V OSGi se typicky provádí provázání importovaných a exportovaných balíků. Požadovaný balík je vyhledáván podle názvu a verze, pokud je uvedena, případně podle dalších atributů. Při uvedení konkrétního čísla verze vyhovuje balík s touto verzí nebo vyšší. Verzi lze omezit také na určitý rozsah.

2.3.2 Sémantické verzování

Pravidla pro verzování OSGi komponent [8] jsou dána specifikací. Verzování se nazývá sémantické, protože verze obsahuje několik složek s daným významem a důležitostí. Její formát je: „major.minor.micro.qualifier“. První tři složky jsou čísla, poslední složka může obsahovat i písmena. Důležitost složek klesá zleva doprava a libovolný počet složek zprava lze vynechat. Význam jednotlivých složek je následující:

- *major*: složka s nejvyšší důležitostí. Její navýšení se provádí vždy, když dojde k porušení zpětné kompatibility (například smazáním nebo změnou metody, která je součástí API). Balíky s různou verzí ve složce major jsou zcela nekompatibilní.
- *minor*: složka se střední důležitostí. Navyšuje se při změnách API, které jsou zpětně kompatibilní (například přidání nové metody do API). Vyžadovaný balík je kompatibilní s poskytovaným balíkem, pokud je major složka verze stejná a minor složka poskytovaného balíku vyšší nebo rovna vyžadovanému.
- *micro*: složka s nejnižší důležitostí. Její navýšení probíhá při každém vydání komponenty, kdy nebylo změněno API, ale pouze vnitřní implementace. Balíky s verzemi lišícími se až ve složce micro by měly být kompatibilní.
- *qualifier*: složka, která se nepoužívá k porovnání verzí, ale pouze k dalšímu upřesnění verze, například číslem SVN revize. Na kompatibilitu balíků by neměla mít žádný vliv.

Pokud dojde k navýšení některé složky, všechny méně důležité složky kromě složky qualifier se vynulují.

2.3.3 Problémy resolvingu a sémantického verzování

Framework vybírá balíky k provázání zejména podle názvu a verze. Tyto informace uvádí programátor v manifestu importující a exportující komponenty. Je tedy zodpovědností programátora, aby určil verze správně. Problémem je, že programátor obvykle mění verze pouze na základě pocitů a domněnek, nemusí si uvědomit dopad některých změn kódu na případnou změnu v API.

Důsledkem jsou časté chyby sémantických verzí komponent a balíků. Tyto chyby mohou způsobit nekompatibilitu komponent, které byly propojeny při resolvingu. To může vést k neočekávané funkčnosti nebo až k pádu celé aplikace.

2.3.4 Další problémy resolvingu

Framework musí někdy řešit složité situace, které mají více možných řešení. Programátor si nemusí vždy uvědomit, jaké problémy mohou tyto situace způsobit. Každý framework může navíc tyto situace řešit různě. Jedná se například o případ, kdy pro uspokojení importu je k dispozici více komponent s odpovídajícím exportem. Jiné problémy mohou nastat, pokud je implementace balíku rozdělena mezi více komponent.

Další nepříjemné efekty mohou vznikat po aktualizaci nebo odinstalování komponenty v běžícím frameworku. Pokud byly ostatní komponenty na aktualizovanou komponentu navázány, tyto vazby se nepřeruší okamžitě. Ve frameworku může proto nějakou dobu existovat stará i nová verze komponenty zároveň. Staré vazby budou využívat starou implementaci, nově vzniklé vazby budou používat novou implementaci. To může způsobit problémy kompatibility. Dalším důsledkem je, že aktualizace komponenty se neprojeví ihned změnou funkčnosti. Aktualizace nebo rozpojení starých vazeb se provede až po aktualizaci celého frameworku.

3 Nahraditelnost komponent

3.1 Použití

Nahrazovat komponentu v komponentové aplikaci je potřeba z různých důvodů. Často se například komponenta nahrazuje za účelem její aktualizace na novější verzi, nebo lze nahrazením upravit funkčnost aplikace.

V některých případech může být žádoucí nahradit komponentu bez přerušení běhu aplikace. Takový případ je serverová aplikace, která musí fungovat nepřetržitě. Aktualizace takové aplikace za běhu přispěje k dodržení limitů na odstávky. Je ovšem potřeba, aby bylo nahrazení bezpečné, tj. aby v systému nevznikla nekompatibilita a nedošlo k pádu aplikace.

Ověření, že nahrazení komponenty za jinou nezpůsobí problémy nebo dokonce pád aplikace, se nazývá nahraditelnost komponent. Komponenta je nahraditelná za jinou komponentu, pokud je nová komponenta kompatibilní s ostatními komponentami v systému.

3.2 Výsledky a směr výzkumu na katedře

Nahraditelností komponent se na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni zabývá tým pod vedením Ing. MSc. Přemysla Brady, Ph.D. Práce probíhají v rámci projektů GAČR (Methods and models for consistency verification of advanced component-based applications, Methods of development and verification of component-based applications using natural language specifications).

V rámci výzkumu byl vytvořen komparátor typů jazyka Java (JaCC - Java Class Compatibility checker) a komparátor OSGi komponent (OBCC - OSGi Bundle Compatibility Checking toolset), který je nadstavbou nástroje JaCC. Komponenty jsou porovnávány do hloubky až na úroveň metod a členských polí Java tříd.

3.2.1 Použití komparátoru při resolvingu

Komparátor lze využít při resolvingu v komponentovém frameworku. Komparátor ověří kompatibilitu komponent, mezi kterými se má vytvořit vazba. Pokud jsou tyto komponenty nekompatibilní, provázání se vůbec neprovede a aplikaci nebude možné spustit. Tím se chyba odhalí okamžitě a předejde se neočekávaným problémům za běhu aplikace.

Další výhodou je možnost použití komparátoru při výběru vhodné exportující komponenty, pokud základní požadavky (jméno, verze, další atributy) spl-

ňuje komponent více. Výběr je pomocí komparátoru omezen pouze na kompatibilní komponenty.

Komparátor byl již zkušebně integrován do frameworku Apache Felix v rámci projektu Felix resolver integration (viz [18]). Dalším námětem je rozšíření frameworku o zobrazení důvodů, proč nelze komponenty provázat a spustit. Pro tento účel bude možné použít grafické zobrazení rozdílů mezi komponentami, které je součástí této práce (kapitola 5).

3.2.2 Podpora pro bezpečnou nahraditelnost komponent

Další možností využití komparátoru je kontrola kompatibility komponent ještě před jejich zavedením do frameworku. Součástí komponentové aplikace může být komponenta, která umožňuje rozšiřování a aktualizaci aplikace za běhu. Tato komponenta může fungovat buď automaticky, nebo může být ovládána uživatelem. Před nasazením vybrané komponenty do frameworku lze využít komparátor ke kontrole kompatibility s ostatními komponentami ve frameworku.

Kompatibilitu je možné kontrolovat dokonce ještě před stažením vybrané komponenty z internetu. K tomuto účelu je připravováno úložiště komponent, které pomocí komparátoru vytváří a ukládá informace o kompatibilitě jednotlivých komponent (viz [32]).

3.2.3 Podpora pro správné sémantické verzování

Dalším směrem pro použití komparátoru je zajištění správného sémantického verzování jednotlivých komponent při jejich vydání. Komponentový framework následně spoléhá standardně jen na čísla verzí. Pokud by byly správně verzovány všechny komponenty v systému, problémy s kompatibilitou by neměly nastat.

Aby se předešlo chybám, lze ponechat určení verze nástroji pro automatické verzování. Webový nástroj pro automatické verzování je k dispozici na adrese *http://osgi.kiv.zcu.cz/obvs/index.html*. Uživatel do formuláře zadá dva soubory, komponentu v původní verzi a upravenou komponentu k overzování. Nástroj použije komparátor k porovnání komponent a podle výsledku určí verzi podle sémantických pravidel pro novou komponentu.

Nástroj pro automatické verzování lze používat také jako plugin do mavenu (viz [35]).

Kromě automatického verzování lze podpořit volbu správné verze také pomocí grafického zobrazení rozdílů mezi původní a upravenou komponentou, kterým se zabývá kapitola 5 této práce. Grafická podoba rozdílů pomůže člověku ke správnému určení verze nové komponenty.

3.2.4 Další náměty pro použití komparátoru

Komparátor by bylo možné dále využít k vyhledávání vyhovující komponenty na internetu nebo v konkrétním úložišti. Podobné řešení je prezentováno v [42]. Pomocí dotazovací komponenty je vyhledávána jiná komponenta, která je s dotazovací komponentou kompatibilní (nebo téměř kompatibilní).

V současné době by bylo možné komparátor využít pouze k vyhledání zcela kompatibilních komponent. Námětem do budoucna je rozšíření výsledku porovnání komponent o kvantifikaci rozdílů. Díky tomu by bylo možné vyhledané komponenty řadit podle míry nekompatibility.

3.3 Teorie nahraditelnosti komponent

Aby bylo možné určit, zda je komponenta nahraditelná za jinou komponentu, je potřeba definovat požadavky nahraditelnosti. Obecně lze říci, že komponenta je nahraditelná za jinou, pokud toto nahrazení nezpůsobí nekompatibilitu. Způsobů, jakými to lze ověřit, je více. Komparátory vyvinuté na katedře ověřují nahraditelnost bez nutnosti spuštění a testování komponent. Následující text popisuje postupy ověřování nahraditelnosti implementované v komparátorech JaCC a OBCC.

3.3.1 Relace subtypu mezi komponentami

Komponentu lze reprezentovat pomocí elementů jejího rozhraní. Ty lze rozdělit na elementy importované a exportované. Rozhraní OSGi komponenty je tvořeno informacemi obsaženými v manifestu, popisech služeb ve formátu XML a Java třídami, které jsou dosažitelné pro ostatní komponenty prostřednictvím exportovaných balíků. Jednotlivé třídy je potřeba reprezentovat až na úroveň signatur jejich polí a metod.

Když máme k dispozici reprezentace dvou komponent, můžeme tyto reprezentace mezi sebou porovnávat. Účelem porovnání je zjistit vzájemný vztah komponent. Ty mohou být buď zcela nekompatibilní, nebo může být některá komponenta subtypem (viz [16]) druhé komponenty. Relaci subtypu si lze představit jako relaci předek – potomek v dědičnosti Java tříd, kde potomek je subtypem (nebo specializací) svého předka.

Podobně jako princip polymorfismu, kde lze nahradit objekt typovaný na předka libovolným potomkem, lze komponentu A nahradit komponentou B, pokud je komponenta B subtypem komponenty A.

3.3.2 Jak lze relaci subtypu zjistit

Zjištění relace subtypu mezi komponentami je obtížné, proto je nutné rozdělit komponentu na menší části, u kterých se zjištění této relace zjednodušuje. Komponenty můžeme rozložit na dostatečně jednoduché části, aby bylo určení relace triviální (primitivní Java typy, metody. . .). Relaci subtypu snadno určíme u všech těchto částí. Celkový výsledek potom postupně skládáme z výsledků dílčích.

3.3.3 Metody ověření nahraditelnosti

Nahraditelnost komponenty A komponentou B lze ověřit pomocí zjištění relace subtypu mezi komponentami A a B. Pokud je komponenta B subtypem komponenty A, lze komponentu A komponentou B nahradit.

Tato základní metoda je použitelná bez znalosti dodatečných informací o prostředí (kontextu) komponenty, ve kterém má být nasazena. Pokud tyto informace máme k dispozici, lze je použít k lepšímu rozhodnutí o nahraditelnosti komponenty za těchto konkrétních podmínek.

Tato druhá metoda se nazývá kontextová nahraditelnost [15]. Místo toho, aby se nově přidávaná komponenta porovnávala s původní komponentou, porovnává se nově přidávaná komponenta se svým komplementem získaným z kontextu.

Kontextem jsou myšleny ostatní komponenty ve frameworku. Z pohledu nahraditelnosti jsou z kontextu zajímavé pouze komponenty s nějakou vazbou na původní komponentu, nebo s potenciální vazbou na novou komponentu.

Komplement komponenty je část komponenty, která je skutečně využívána kontextem. Komplement je proto neúplná reprezentace komponenty. Protože se komplement získává z kontextu, není nahraditelnost závislá na původní nahrazované komponentě. Ta dokonce nemusí vůbec existovat a kontextovou nahraditelnost lze využít čistě pro účely zjištění kompatibility exportující a importující komponenty.

Pokud je komponenta subtypem svého komplementu získaného z kontextu, je s kontextem kompatibilní (původní komponenta je nahraditelná touto komponentou v tomto konkrétním kontextu).

3.4 Komparátory Java tříd a OSGi komponent

V této kapitole budou podrobněji popsány jednotlivé projekty komparátorů i dalších nástrojů, které je využívají. Projekty představují výchozí stav pro tuto práci. Během práce jsem se ale také zapojila do vývoje projektů. Implementovala jsem některá vylepšení a opravy chyb.

3.4.1 Projekty pro získání reprezentace

Úkolem následujících projektů je získání reprezentace Java typů nebo OSGi komponent. Reprezentaci je možné získávat různými způsoby.

- *javatypes* [40]
  Projekt obsahuje zejména datové třídy pro uchování načtené reprezentace Java typů. Kromě toho obsahuje také načítání této reprezentace z bytekódu a reflexe, ale tato implementace je nyní již zastaralá a nově se používá implementace v projektu javatypes-loader.
- *javatypes-loader*
  Projekt implementuje v současné době načítání reprezentace Java typů z bytekódu. V budoucnu by měl obsahovat také načítání z reflexe.
- *bundle-types* [40]
  Projekt obsahuje datové třídy pro uchování reprezentace elementů OSGi komponent.
- *bundle-loader* [40]
  V projektu je implementováno načítání reprezentace elementů OSGi komponent.
- *bundle-parser*
  Projekt načítá reprezentaci Java typů i elementů OSGi komponent z kontextu použitím načítání z bytekódu. V budoucnu by měl být nahrazen novou implementací načítání reprezentace v projektech javatypes-loader a bundle-loader.
- *bundle-context-loader*
  Projekt načítá reprezentaci Java typů a elementů OSGi komponent z kontextu pomocí reflexe.
- *types-cmp*
  Všechny předchozí projekty závisí na projektu types-cmp, který obsahuje základní rozhraní.

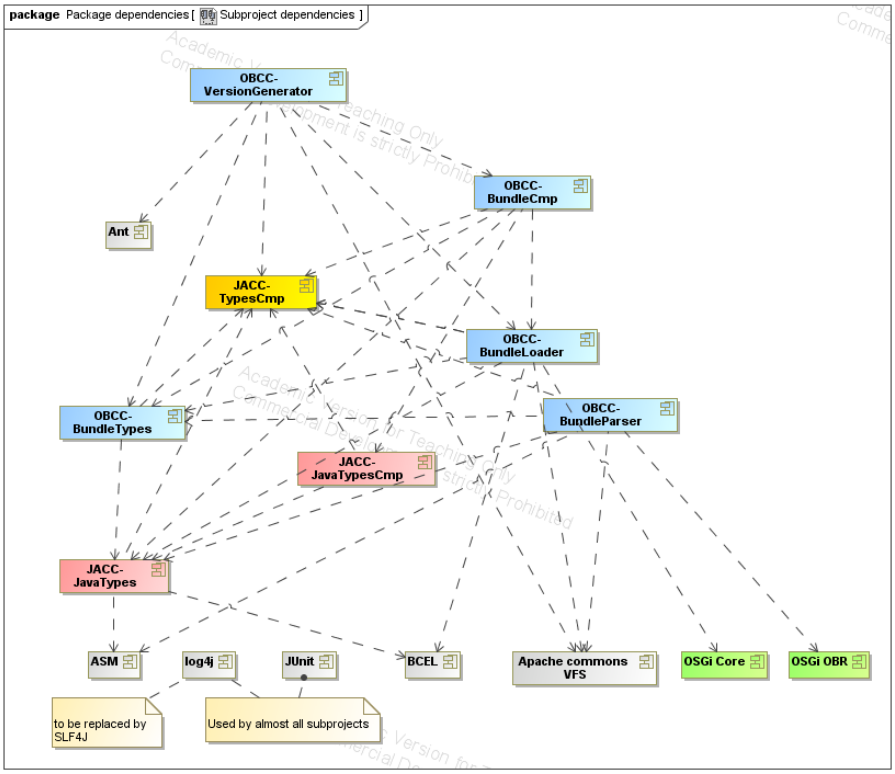
3.4.2 Projekty komparátorů získané reprezentace

Následující projekty porovnávají reprezentaci Java tříd nebo elementů OSGi komponent a zjišťují relaci subtypu mezi porovnávanými komponentami.

- *javatypes-cmp*
  Komparátor reprezentací Java typů.
- *bundle-cmp*
  Komparátor reprezentací elementů OSGi komponent.

3.4.3 Závislosti mezi projekty

Projekty lze rozdělit na část JaCC (types-cmp, javatypes, javatypes-loader, javatypes-cmp) a OBCC (bundle-types, bundle-loader, bundle-parser, bundle-context-loader, bundle-cmp). Zjednodušeně lze říci, že projekty OBCC závisí na projektech JaCC a projekty komparátorů závisí na projektech obsahujících reprezentaci komponent. Podrobněji zobrazuje závislosti mezi projekty diagram 3.1.



Obrázek 3.1: Závislosti mezi projekty. Obrázek z dokumentace projektu OBCC.

3.4.4 Další projekty

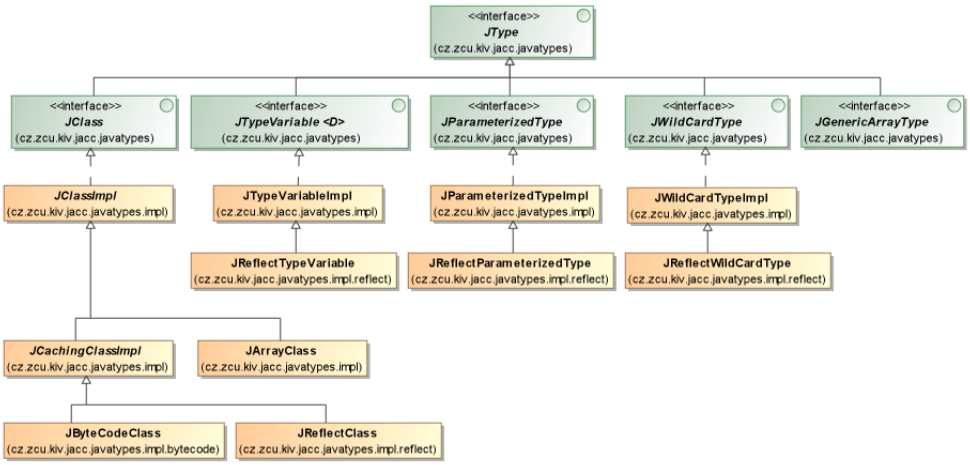
Následující projekty využívají komparátory k vytvoření reálně použitelné funkcionality.

- *OBVS (OSGi Bundle Versioning Service)* [14]
  Webové rozhraní pro automatické verzování OSGi komponent.
- *maven-bundle-version-plugin*
  Plugin do Mavenu pro automatické verzování OSGi komponent.

3.5 Architektura komparátorů

3.5.1 Reprezentace Java typů

Datová struktura načtené reprezentace Java typů je inspirována rozhraním Java reflexe (balík *java.lang.reflection*). Oproti rozhraní reflexe se struktura trochu liší a je zjednodušena. Diagram 3.2 zobrazuje hierarchii Java typů implementovanou v projektu JaCC.



Obrázek 3.2: Hierarchie Java typů. Obrázek z dokumentace projektu OBCC.

3.5.2 Architektura načítání reprezentace

Architektura načítání reprezentace není jednotná. Je to způsobeno tím, že na jednotlivých projektech pracovalo velké množství lidí a nebyla stanovena jednotná pravidla. V současné době se pracuje na sjednocení architektury vytvořením nového projektu javatypes-loader, který v budoucnosti nahradí (spolu s projektem bundle-loader) původní implementace načítání reprezentace, které jsou roztroušené mezi různými projekty.

3.5.3 Architektura komparátorů

Projekty komparátorů lze rozdělit na třídy provádějící porovnání a datové třídy pro uchování výsledku porovnání.

Třídy provádějící porovnání částečně kopírují hierarchii reprezentace. Pro každý reprezentovaný typ existuje třída, která dokáže objekty tohoto typu porovnat. Třídy typicky obsahují jedinou veřejnou metodu compare, jejíž parametry jsou porovnávané typy a stav porovnávání. Metoda vrací výsledek porov-

nání. Instance těchto tříd nemají stav, a proto je lze používat opakovaně a ve více vláknech. Instance se získávají pomocí tovární třídy (viz [25]).

Stav porovnání je pro všechny třídy provádějící porovnání společný a je uložen v objektu typu ComparatorState, který je předáván v metodách compare. Stav porovnání obsahuje historii porovnaných typů, různé parametry porovnání a tovární třídy pro získávání komparátorů jednotlivých typů.

3.5.4 Agregace výsledku porovnání

Každá třída provádějící porovnání má svůj specifický algoritmus a k sestavení výsledku používá výsledky získané od komparátorů nižších úrovní. Výsledek porovnání se skládá z agregovaného rozdílu, který může nabývat pouze několika hodnot a vyjadřuje relaci subtypu. Druhou částí je množina stejně strukturovaných výsledků získaných od komparátorů nižších úrovní, ze kterých je možné dohledat důvody pro souhrnný výsledek.

Agregovaný rozdíl může nabývat těchto hodnot:

- UNK: neznámý výsledek
- NON: shoda
- INS: vložení (element je přítomný pouze v nové komponentě)
- DEL: smazání (element je přítomný pouze v původní komponentě)
- SPE: specializace (element nové komponenty je subtypem odpovídajícího elementu v původní komponentě)
- GEN: generalizace (element původní komponenty je subtypem odpovídajícího elementu v nové komponentě)
- MUT: mutace (mezi elemnety neplatí relace subtypu v žádném směru)

Během porovnání nejdříve vznikají pouze výsledky NON, INS nebo DEL. Ostatní výsledky vznikají až jejich agregací. Agregace rozdílů se řídí podle priorit. Pořadí výsledků podle priorit od nejdůležitějších po nejméně důležité: UNK, MUT, INS, SPE, DEL, GEN, NON. Pokud je v agregovaných výsledcích zastoupen rozdíl INS nebo SPE zároveň s rozdílem DEL nebo GEN, výsledkem je vždy MUT.

Agregovat je obvykle potřeba získaná množina rozdílů z komparátorů na nižší úrovni. V některých případech je ale nutné provést nejdříve inverzi rozdílu, protože pro získání relace subtypu je potřeba získat opačnou relaci subtypu na nižší úrovni. To se projevuje například u parametrů metod. Pokud je klient zvyklý volat v původní komponentě metodu s parametrem A, musí komponenta, která je subtypem původní komponenty, toto volání také umožnit. Proto formální parametr volané metody v původní komponentě musí být subtypem parametru v nové komponentě (případně může být samozřejmě shodný).

3.6 Problémy a náměty pro další práci

3.6.1 Generické typy

Během své práce na projektech jsem objevila několik nedostatků nebo chyb implementace. Prvním a nejzásadnějším je problém s generikami v komparátoru. Implementace porovnávání generických typů není dotažená a algoritmus často končí v nekonečné smyčce. Důvodem je, že komparátor nepočítá s možností cyklických typů, jako například:

```
class A <E extends List<A>>
```

Komparátor je připraven pouze na cykly obyčejných tříd, které řeší zavedením historie porovnávaných tříd. Před samotným porovnáním tříd je pomocí historie ověřeno, že porovnávaná dvojice ještě nebyla porovnávána, nebo není porovnávána právě teď díky rekurzi. Dvojice je ihned vložena do historie a poté se provede porovnání. Stejné řešení je potřeba zavést pro generické typy.

Reprezentace generických typů je popsána v dokumentu [2].

3.6.2 Porovnávání tříd

Druhým nedostatkem, který zhoršuje funkcionalitu, je absence mechanismu, kterému v rozhraní Java reflexe odpovídá metoda isAssignableFrom. Metoda zjišťuje, zda je možné objekt přetypovat na danou třídu nebo rozhraní. Implementace v komparátorech by znamenala hlavně nutnost načítat předky a rozhraní tříd. Poté by bylo možné upravit algoritmus porovnávání tříd, aby se při rozdílném názvu nevygenerovala automaticky mutace. Pokud by se jedna z porovnávaných tříd shodovala s některým předkem třídy druhé, výsledek porovnání by se mohl zlepšit na generalizaci nebo specializaci.

3.6.3 Ověřování kompatibility dvou komponent

Třetí problém je spíše koncepčního typu. Týká se ověřování kompatibility exportující a importující komponenty. Uvedu nejdříve problematický příklad kódu v jednotlivých komponentách:

```
Komponenta 1:
interface MujInterface {
        void metoda(Parametr p);
}

Komponenta 2:
class MojeImplementace implements MujInterface {
        void metoda(Parametr p) {
                p.toString();
        }
```

```
}

Komponenta 3:
class Parametr {
        String toString() {
                return "parametr";
        }
}
```

Třídu Parametr z komponenty 3 importují obě zbylé komponenty. Komponenta 2 importuje rozhraní MujInterface z komponenty 1. Problém nastává při ověření kompatibility komponent 1 a 2.

Porovnání bude probíhat na základě načtené reprezentace komponenty 1 standardním způsobem a načteného komplementu komponenty 1 z kontextu, tj. z komponenty 2. První reprezentace bude obsahovat třídu Parametr, která je prázdná (nejsou dostupné zdroje pro načtení její implementace). Druhá reprezentace bude obsahovat také třídu Parametr, ale nyní včetně její metody toString(), protože se jedná o načítání z kontextu (načtou se všechny metody, které jsou použité).

Výsledkem porovnání bude, že komponenta načtená z kontextu je subtypem komponenty načtené standardním způsobem, protože obsahuje navíc metodu Parametr.toString(). To znamená, že komponenty 1 a 2 budou vyhodnoceny jako nekompatibilní. Komponenta 2 používá metodu, která v komponentě 1 chybí.

Řešením je zavedení ignorovaných balíků během porovnávání. Pravděpodobnost, že bude implementace balíku rozprostřena do více komponent, je malá. Mezi ignorované balíky lze proto zahrnout všechny importované balíky obou komponent, které zároveň nejsou exportované některou z nich. Výsledek porovnání tříd z těchto balíků by měl být vždy NON, protože při importu stejné komponenty do obou komponent budou tyto třídy skutečně stejné. Porovnání by se na této úrovni mělo zastavit, aby nebylo možné se dostat na jiné balíky importovaných komponent.

3.6.4 Další nedostatky

Dále jsem zjistila, že v komparátorech chybí kompletně implementace anotací, implementace načítání reprezentace pomocí reflexe je ve velmi nedotaženém stavu a chybí implementace porovnávání výjimek.

3.7 Implementované opravy a vylepšení

Následující opravy a vylepšení komparátorů jsem implementovala nad rámec prací popsaných v kapitolách 4 a 5, které jsou jádrem této diplomové práce. U

jednotlivých úprav uvádím také čísla požadavků v systému správy změn (viz kapitola 3.8.4), kde lze dohledat jejich detailní popis.

3.7.1 Úpravy hierarchie Java typů

Během seznamování se s kódem komparátoru jsem objevila některé nelogičnosti v hierarchii tříd reprezentace Java typů. Dále jsem upravila implementaci metod toString, která původně způsobovala v některých případech nekonečné cyklení. Detailní popis provedených změn lze dohledat v nástroji pro správu požadavků pod těmito čísly požadavků:

JaCC - #105, #89, #122, #89, #123, #125

3.7.2 Pomoc při thread-safety analýze

Bc. Martin Štěpánek prováděl v rámci oborového projektu thread-safety analýzu, neboli analýzu bezpečnosti komparátorů při běhu ve více vláknech najednou. Tento úkol svým rozsahem překračoval rozsah oborového projektu, proto jsem se na práci také podílela.

Prováděla jsem jak samotnou analýzu části projektů, tak implementaci některých následných úprav. Výsledky analýzy lze dohledat v dokumentu [4]. Popis mnou implementovaných úprav je v nástroji pro správu požadavků pod těmito čísly požadavků:

JaCC - #114, #42, #119

OBCC - #291, #256, #319

3.7.3 Úpravy struktury výsledků porovnání

Strukturu výsledků porovnání jsem vylepšila o referenci na agregovaný výsledek a začala jsem pracovat také na zjednodušení struktury. Požadavky lze dohledat pod těmito čísly:

JaCC - #120, #158

3.7.4 Vylepšení algoritmu porovnání tříd

Vylepšila jsem algoritmus porovnání tříd tím, že jsem sjednotila způsob detekce rekurze s ukládáním a používáním výsledků již porovnaných tříd. Protože měla tato úprava vliv na výkonnost komparátoru, je tato změna blíže popsána v části provedených optimalizací v kapitole 4.10.2. Požadavky jsou dohledatelné pod těmito čísly:

JaCC - #113, #114

3.8 Technologie a vývojové nástroje

3.8.1 Java

Všechny projekty jsou psané v programovacím jazyce Java a používají standardní JDK verze 1.6.

Na projektech je potřeba dodržovat domluvené konvence. Kód i komentáře by se měly psát anglicky a v kódování UTF-8. Každá třída a důležité metody by měly být komentovány Javadoc komentářem. K logování by se měla využívat knihovna SLF4J.

Každý zdrojový soubor by měl obsahovat hlavičku s informacemi o licenci. Ve vývojovém prostředí Eclipse lze nastavit automatické vkládání této hlavičky v nabídce Window – Preferences – Java – Code Style – Code Templates – Comments – Files. Kromě nastavení textu je potřeba také zaškrtnout pod textem automatické vkládání kommentářů.

Další konvence jsou dány pomocí pravidel pro automatické formátování, které lze v Eclipse nastavit v nabídce Window – Preferences – Java – Code Style – Formatter. Pravidla lze stáhnout z [3].

V projektech by měla být dodržována pravidla práce s výjimkami popsaná v [1].

3.8.2 OSGi

Projekty komparátorů lze použít buď jako knihovny, nebo také jako OSGi komponenty.

3.8.3 Apache Maven

Maven [21] je nástroj pro automatický build. Je alternativou k nástrojům Ant nebo Make. Na rozdíl od nich se nezaměřuje na popis jak se co má udělat, ale pouze popis požadovaného výsledku.

Konfigurace build procesu je umístěna v XML souboru pom.xml. Ten obsahuje typicky identifikaci projektu, závislosti a případně seznam pluginů, které přidávají další akce (například testování). Identifikace projektu je rozdělena na část groupId (identifikace skupiny projektů), artifactId (identifikace projektu) a version (verze).

Projekty a knihovny, na kterých projekt závisi (dependencies), se identifikují

také pomocí tagů groupId, artifactId a version. Pomocí tagu scope lze určit, zda bude knihovna vložena do výsledného JAR souboru (compiled), nebo zda bude pouze přidávána na classpath při spuštění (provided).

Projekt musí mít předepsanou strukturu (pokud není pomocí konfigurace změněna). Zdrojové soubory jsou v adresáři src/main/java a ostatní zdroje v src/main/resources. Pro testy jsou podobně vymezeny adresáře src/test/java a src/test/resources. Build se spouští bud' z příkazové řádky, nebo pomocí grafického rozhraní Eclipse (plugin M2Eclipse). Obvykle se používá příkaz:

```
mvn install
```

Tento příkaz zkompiluje zdrojové soubory, zabalí přeložené třídy a zdroje do JAR archivu, spustí automatické testy a JAR archiv uloží do lokálního úložiště. Případné závislosti se snaží stahovat z veřejných úložišť na internetu.

V projektech JaCC a OBCC se používá takzvaný rodičovský POM (parent), který je umístěný v trunk/pom. V jsou definované společné části jednotlivých POM souborů.

Pro build všech projektů najednou lze použít takzvaný reaktor POM, který je umístěný přímo v adresáři trunk. Nejdříve je nutné spustit mvn install pro JaCC, následně pro OBCC.

3.8.4 Assembla

Assembla [28] je nástroj, který integruje verzovací systém SVN se správou požadavků. Commit do SVN lze jednoduše provázat s příslušným požadavkem pomocí uvedení komentáře „Re #123“ (příklad pro číslo požadavku 123). SVN lze používat pomocí libovolného klienta, například TortoiseSVN [43].

Projekt JaCC je na adrese *http://www.assembla.com/spaces/jacc*, projekt OBCC na *http://www.assembla.com/spaces/obcc*.

3.8.5 Hudson Continuous Integration

Hudson [37] je nástroj pro plynulou integraci. Pravidelně zjišťuje, zda v SVN nepřibyla změna, a pokud ano, spustí build projektu a testy.

Hudson pro projekty JaCC a OBCC je k dispozici na adrese *http://students.kiv.zcu.cz:8080/*.

3.9 Způsob práce v týmu

Na projektech JaCC a OBCC pracovalo v průběhu této práce několik lidí současně. Bylo nutné synchronizovat některé změny (refactoring) s optimalizacemi

a rozšířeními.

Aby mohla spolupráce v týmu dobře fungovat, je nutné svou práci dostatečně dokumentovat. To znamená zejména vytváření záznamů v nástroji pro správu požadavků a propojování každého commitu do SVN s požadavkem.

Před nahráním provedených změn do SVN je nutné ověřit, že jsou projekty kompilovatelné pomocí Mavenu. Procházet by měly také automatické testy (v současné době prochází jen některé).

V průběhu práce se celý tým pracující na projektech JaCC a OBCC pravidelně scházel každých čtrnáct dnů. Forma schůzek byla převzata z metodiky Scrum.

V metodice Scrum se tato schůzka nazývá daily standup a opakuje se každý den. Členové týmu by měli celou dobu stát (aby se schůzka neprodlužovala) a každý podává zprávu o třech bodech:

- co jsem dělal(a)
- co budu dělat
- co mi brání v práci

Schůzku vede Scrum-master, který následně řeší překážky členů týmu. Schůzka by měla být dlouhá zhruba patnáct minut a neměly by se na ní řešit problémy. Měla by sloužit pouze ke vzájemnému informování o tom, kdo na čem pracuje.

Tyto schůzky částečně předchází provádění zbytečné nebo opakované práce v důsledku nedostatečné komunikace mezi členy týmu.

4 Využití komparátorů na malých zařízeních

4.1 Malá zařízení

Používání malých (výkonově omezených) zařízení, jako jsou chytré telefony, PDA nebo tablety, je stále více oblíbené. Do budoucna lze očekávat podporu (téměř) standardní Javy na většině mobilních zařízeních. Již nyní lze používat většinu aplikací pro standardní Javu na platformě Android.

4.1.1 Android

Android [26] je open-source platforma pro mobilní zařízení (chytré telefony, navigace, tablety). Skládá se z operačního systému, ovladačů pro konkrétní zařízení, aplikačního frameworku a běžných uživatelských aplikací. Operační systém je založen na jádru systému Linux.

Aplikační framework poskytuje rozhraní pro uživatelské aplikace v Javě. Java na Androidu se mírně liší od standardní Javy. Virtuálním strojem je Dalvik virtual machine. První rozdíl oproti standardní Javě je nutnost překladu class souborů do speciálního formátu (dex). Mezi další rozdíly patří, že Dalvik nepodporuje některé speciální knihovny, například JAXB nebo DOM pro práci s XML.

Programátor aplikací má k dispozici Android SDK – soubor knihoven a nástrojů pro vývoj aplikací. Součástí SDK je také velmi dobře použitelný simulátor telefonu. Aplikace jsou psané jako komponenty aplikačního frameworku a v názvosloví Androidu se jim říká aktivity. Aktivity mezi sebou komunikují pomocí zpráv s názvem „intent“.

Android poskytuje vlastní prostředky pro tvorbu uživatelského rozhraní. Obvykle je grafické rozhraní konfigurováno pomocí grafického editoru a je přidruženo ke každé aktivitě.

4.1.2 Využití OSGi na platformě Android

Ačkoliv Android poskytuje svůj vlastní komponentový (aplikační) framework, může být OSGi framework na Androidu také užitečný. Existuje mnoho OSGi aplikací, které by se mohly bez větších úprav spustit na platformě Android.

Pokusy přenést OSGi framework na Android se již objevují, příkladem je projekt EZdroid [33], který se zaměřuje na použití frameworku Apache Felix na platformě Android.

Výhodou OSGi komponent je nižší režie na komunikaci, než jakou mají komponenty Androidu. Komunikace v OSGi probíhá pomocí volání metod. Může to být také jednodušší pro programátora než posílání zpráv a vazba mezi komponentami je těsnější.

Na druhou stranu, použití OSGi na Androidu má i své nevýhody. Například nelze jednoduše vytvářet uživatelské rozhraní v OSGi komponentách pomocí grafického editoru. Komponenty mohou rozhraní vytvářet jedině programově, nebo musí být rozhraní součástí frameworku samotného (aktivity, ve které je framework spouštěn).

4.1.3 Využití komparátorů na platformě Android

Při používání OSGi na Androidu nebo jiných platformách pro malá zařízení vzniká také požadavek na možnost bezpečného nahrazení nebo aktualizaci komponent. Ověření nahraditelnosti ale přináší nové režijní náklady na čas běhu aplikace i spotřebu paměti.

Malá zařízení jsou mnohem méně výkonná než běžné počítače a mají velmi omezené paměťové zdroje. Příliš velká režie může proto snadno způsobit nepoužitelnost takového systému. Aby bylo možné na malých zařízeních komparátory používat, je potřeba, aby jejich paměťové a časové nároky byly přijatelně nízké.

4.2 Předchozí práce

V rámci oborového projektu (viz [17]) byl framework Apache Felix upraven pro běh na platformě Android. Byla využita také integrace komparátorů do procesu resolvingu (typová kontrola při resolvingu). Práce měla za úkol udělat jednoduchý výkonnostní test komparátorů na platformě Android. K vyhodnocení výsledků byly porovnávány časy a paměť spotřebovaná při resolvingu bez typové kontroly a s typovou kontrolou.

Výsledkem práce je možnost spuštění komparátorů na platformě Android a orientační výsledky výkonnostních testů. Ukázalo se, že paměťové a časové nároky typové kontroly nejsou ideální. Při testech reálné aplikace došlo několikrát k vyčerpání paměti. Relativní nárůst paměťových i časových nároků se lišil pro různé velikosti komponent. Časová náročnost resolvingu se s přidáním typové kontroly zvýšila v rozmezí 9 až 45 procent, paměťová náročnost se zvýšila o 15 až 64 procent. Pro uživatele to znamená zvýšení doby běhu v řádu desítek sekund.

4.3 Úkol této práce

Úkolem této práce je zvýšit použitelnost komparátorů na malých zařízeních snížením jejich nároků na systémové zdroje. Komparátory by měli úsporně hospodařit s pamětí a vytvářet malé datové struktury. Práce se zaměří také na kontrolu a případnou optimalizaci algoritmů pro snížení času provádění typové kontroly.

4.3.1 Metodika

Před samotnou optimalizací je nutné vytvořit mechanismus měření aktuální výkonnosti (časové i paměťové). K tomuto účelu byla vytvořena testovací aplikace, která produkuje přehledný výstup obsahující aktuální výsledky i zlepšení oproti výsledkům předchozím.

Následně je možné analyzovat kód aplikace za účelem nalezení problematických částí kódu a provedení jejich optimalizace. Analýza komparátorů probíhala manuálně i za použití programových nástrojů (profiler, heap analyser).

Efekt optimalizací lze ověřit a vyhodnotit pomocí testovací aplikace pro jednotlivé změny i pro všechny provedené optimalizace. Testovací aplikace byla nasazena do procesu plynulé integrace (continuous integration), aby mohla být výkonnost sledována také při budoucích změnách.

4.4 Výkonnostní testy

Pomocí výkonnostních testů [34] lze ověřit, jak se testovaná aplikace chová při určité zátěži. Důležitými hledisky k posouzení výkonnosti aplikace jsou obvykle konzumace paměti, odezva nebo čas výpočtu. U serverových aplikací se často měří také průchodnost – počet zpracovaných požadavků za jednotku času. Výkonnostní testy lze využít také k měření škálovatelnosti a spolehlivosti aplikace.

4.4.1 Účel výkonnostních testů

Účel výkonnostních testů může být následující:

- *Ověření, že systém splňuje daná výkonnostní kritéria.*
  Specifikace požadavků na systém může obsahovat kromě funkčních požadavků také požadavky na výkonnost, například maximální dobu odezvy na konkrétní akce uživatele nebo průchodnost určitého typu požadavků. Výkonnostní požadavky mohou být dané také konfigurací cílového zařízení, například omezením paměti.

- *Ověření, že systém zvládne reálnou zátěž.*
  Běžné funkční testy testují aplikaci při minimální zátěži. Chování aplikace při velké zátěži může být jiné. Pokud zátěž vzroste nad určitou mez, systém nemusí být schopen některé požadavky plnit. Proto se používají testy, které simulují předpokládanou reálnou zátěž. Sleduje se, zda systém i při této zátěži splňuje požadované nebo rozumné limity odezvy a je schopen poskytovat kompletní funkcionalitu. Dále je možné zátěž postupně zvyšovat a hledat maximální zátěž, kterou je systém schopen zvládnout.

- *Porovnání 2 systémů a určení, který má lepší výkonnost.*
  V některých případech jsou komponenty systému nebo celé aplikace vybírány podle výkonnosti. Srovnání výkonnosti vlastní aplikace s konkurenčními systémy lze využít k propagaci.

- *Měření aktuální výkonnosti a porovnání s předchozími verzemi.*
  Měření a porovnání různých verzí jedné aplikace lze využít k vyhodnocení efektu optimalizací nebo sledování efektu jednotlivých změn na výkonnost. Tento účel vyžaduje přesnější metody měření, protože změny výkonnosti mohou být malé.

- *Vyhledání kritických částí systému (bottleneck).*
  Tento typ testů je zaměřen na vyhledání míst v kódu, kde výpočet tráví nejvíce času. Tyto místa je vhodné následně optimalizovat. K vyhledání kritických míst se obvykle používají nástroje zvané profilery.

- *Monitorování systému.*
  Monitorování se používá u již nasazených systémů, proto je požadována nízká režie prováděných měření. Monitorováním lze odhalit potenciální problémy vzniklé například zvětšením dat nad únosnou mez nebo zvýšením zatížení v důsledku vyšší oblíbenosti systému. Na tyto problémy lze včas reagovat.

4.4.2 Vytváření výkonnostních testů

Implementaci výkonnostních testů nejvíce ovlivňuje jejich účel. Ten rozhoduje o tom, jak velkou režii mohou testy mít, zda je lze opakovat na stejných datech a jaké technologie lze při implementaci využít. Účel ovlivňuje také metriky, které je vhodné měřit.

Technologie měření, možnost opakovatelnosti testů a přípustná režie mají zásadní dopad na přesnost měření. Pro některé účely je vyžadována vysoká přesnost měření (porovnání původní a optimalizované verze systému), pro jiné plně postačují orientační hodnoty (monitorování systému, měření při reálné zátěži).

Aby měly výsledky testů co největší hodnotu, měly by být testy prováděny v prostředí, které se co nejvíce blíží reálnému. Také testovací data by měla odpovídat realitě svou velikostí i různorodostí.

4.5 Analýza kódu

Pomocí výkonnostních testů lze zjistit výkonnostní problémy aplikace. Následně je nutné provést analýzu kódu, která odhalí příčiny problémů a detekuje místa v kódu, která jsou vhodná k optimalizaci.

Analýzu lze provádět buď manuálně, nebo s použitím nějakého nástroje. Pomocí manuální analýzy lze odhalit zejména neefektivní algoritmy nebo nevhodné datové struktury.

Základním nástrojem pro analýzu je profiler. Funkčnost profilerů obvykle zahrnuje měření času výpočtu v jednotlivých metodách a počet jejich zavolání. Čas strávený v metodě lze rozdělit na čas včetně dalších zavolaných metod a čas bez nich. Časy jsou obvykle zkresleny kvůli režii jejich měření. Pro účely analýzy ale toto zkreslení není podstatné.

Pomocí profileru lze odhalit metody, které spotřebují nejvíce času výpočtu. Platí zde Paretovo pravidlo, že 80 procent času je spotřebováno ve 20 procentech metod. Na těchto 20 procent nejnáročnějších metod má smysl se následně zaměřit při manuální analýze.

Dalšími užitečnými nástroji jsou analyzátory paměti (heap analyser). S jejich pomocí lze odhalit objekty, které zabírají nejvíce paměti, nebo třídy s podezřele velkým počtem instancí. Na základě výsledků se lze zaměřit na optimalizaci datových struktur a uvolňování nepotřebných objektů.

Analyzátory paměti obvykle umožňují zachytit a dále analyzovat stav paměti v jednom okamžiku. Je proto důležité tento okamžik vhodně zvolit, aby získané výsledky měly dobrou informační hodnotu.

Dále jsou k dispozici nástroje pro statickou analýzu kódu. Ty dokážou odhalit problémy způsobené špatnou kvalitou kódu.

4.5.1 Nástroje použité pro analýzu komparátorů

- *Eclipse TPTP*
  Eclipse TPTP [24] je profiler založený na vývojovém prostředí Eclipse. Jeho uživatelské rozhraní je velmi přívětivé. Nástroj nabízí velké množství pohledů na naměřená data, ze kterých jsem využila hlavně přehled časově nejnáročnějších balíků, tříd a metod. Po rozbalení detailu metody se zobrazí všechny volané metody opět seřazené podle spotřeby času.

  Nevýhodou nástroje je výrazná režie při měření. Výpočet, který běžně trval několik stovek milisekund, trval s použitím tohoto nástroje 14 sekund. Delší výpočty nebylo možné vůbec analyzovat, protože nástroj výpočet nezvládl a bylo nutné ho restartovat.

  Celkově se nástroj ukázal být velmi užitečný a pomohl mi rychle detekovat problém vhodný k optimalizaci (viz kapitola 4.10.1).

- *Eclipse MAT*

Eclipse MAT (Memory Analyzer) [23] je analyzátor paměti, který je také postaven na vývojovém nástroji Eclipse. Jeho uživatelské rozhraní je poměrně nepřehledné. Nástroj nabízí široké spektrum pohledů na data včetně možnosti vyhledávání konkrétních objektů. Já jsem využila zejména přehled největších konzumentů paměti seskupených podle balíků.

Nástroj analyzuje soubor vygenerovaný virtuálním strojem, který zachycuje stav paměti v jednom okamžiku výpočtu. Soubor lze vygenerovat programově použitím tohoto kódu:

```
MBeanServer server =
   ManagementFactory.getPlatformMBeanServer();
HotSpotDiagnosticMXBean hotspotMBean =
   ManagementFactory.newPlatformMXBeanProxy(
      server,
      "com.sun.management:type=HotSpotDiagnostic",
      HotSpotDiagnosticMXBean.class);
hotspotMBean.dumpHeap("heapdump.hprof", true);
```

Kód je specifický pro Oracle HotSpot VM [38] a neobsahuje ošetření výjimek. Stav paměti je uložen do souboru „heapdump.hprof“.

Také pomocí tohoto nástroje se mi podařilo detekovat významný problém v kódu, viz kapitola 4.10.3.

4.6 Optimalizace

Optimalizace jsou úpravy systému za účelem zlepšení jeho výkonu a snížení nároků na systémové zdroje (paměť, čas). Provádí se na základě předchozí analýzy kódu. Pro vyhodnocení efektu optimalizací je potřeba mít výkonnostní testy, které měří výkonnost jednotlivých verzí systému s dostatečnou přesností.

4.6.1 Měření efektu optimalizací

Pomocí výkonnostních testů lze srovnávat jednotlivé verze systému a sledovat vliv každé změny na výkonnost. K vyhodnocení jednotlivých změn lze srovnat výkonnost aktuální verze s verzí předchozí. Díky testování systému po každé změně lze odhalit změny, které způsobily snížení výkonnosti systému a tyto změny odstranit.

Dílčí změny mají obvykle malý efekt na výkonnost. Celkový efekt optimalizací se proto vyhodnocuje souhrnně pro sérii změn. K vyhodnocení slouží srovnání výkonnosti aktuální verze s výkonností počáteční (bázové) verze systému.

Po dosažení potřebného zvýšení výkonnosti nebo po uzavření optimalizačního cyklu je obvykle počáteční verze posunuta na verzi aktuální. Optimalizace potom mohou pokračovat v dalším cyklu. Během optimalizačního cyklu je potřeba eliminovat změny funkčnosti systému, protože ty mohou mít podstatný

vliv na statistiky výkonnosti. Efekt optimalizací by potom nebylo možné vyhodnotit. Počáteční verze pro měření výkonnosti musí být vždy posunuta za změny funkčnosti.

4.6.2 Úroveň optimalizace

Optimalizace lze provádět na různé úrovni:

- *design*
  Optimalizace na úrovni designu aplikace zahrnují volbu efektivních algoritmů a vhodných datových struktur. Na této úrovni lze obvykle dosáhnout největšího zlepšení výkonnosti.
- *programový kód*
  Kvalita programového kódu může také ovlivnit výkonnost. Lze ji ovlivnit volbou vhodných konstrukcí programovacího jazyka. Často je ovšem potřeba udělat kompromis mezi výkonností a čitelností kódu. Dnešní překladače navíc používají velmi efektivní metody optimalizace, proto optimalizace na této úrovni není příliš efektivní.
- *kompilace*
  Dnešní překladače dokážou výkonnost aplikace výrazně zvýšit. Optimalizace na této úrovni může znamenat změnu překladače nebo nastavení jeho optimalizačních parametrů. Výsledný rozdíl výkonnosti může být zásadní.
- *běh*
  Interpretované jazyky, jako například Java, provádějí kompilaci za běhu pomocí tzv. JIT (just-in-time) překladače. Výhodou těchto překladačů je možnost využití aktuálního vstupu k volbě vhodných optimalizací. Na druhou stranu není možné dosáhnout výkonnosti nativních aplikací kvůli režii s počáteční interpretací a následným překladem. Na této úrovni lze optimalizovat aplikaci volbou lepšího virtuálního stroje nebo nastavením parametrů JIT překladače.

4.7 Výkonnost Java aplikací

Hlavními výkonnostními ukazateli jsou doba výpočtu a spotřeba paměti. V této kapitole se zaměřím blíže na jejich specifika v Java aplikacích.

4.7.1 Doba výpočtu

Doba výpočtu je obecně ovlivněna mnoha faktory, jako je například výkonnost hardware, režie operačního systému nebo zatížení systému ostatními aplikacemi. Z těchto důvodů je její přesné měření vždy obtížné. Operační systémy proto

poskytují prostředky k měření procesorového času spotřebovaného konkrétním procesem, který některé tyto vlivy alespoň částečně eliminuje.

Pro aplikaci v Javě je ale získání informace o spotřebovaném procesorovém čase obtížně dosažitelná. Lze ji získat pomocí propojení s nativní knihovnou, což s sebou nese další režii. Oproti nativním aplikacím je tento čas zkreslen režií virtuálního stroje.

K dalšímu zkreslení doby běhu v Javě dochází kvůli činnosti garbage kolektoru (viz kapitola 4.7.3). Ten se spouští z pohledu aplikace nepředvídatelně a během jeho činnosti je činnost aplikace obvykle zastavena.

Velký vliv na dobu výpočtu má také JIT kompilátor, který provádí kompilaci do strojového kódu během výpočtu. Na počátku výpočtu shromažďuje kompilátor statistiky, na základě kterých může provést optimalizace kódu. Potom je postupně prováděna kompilace, která se projevuje postupným zrychlováním opakovaných částí kódu. Po určité době kompilátor přestává pracovat a rychlost opakovaných částí výpočtu se ustálí.

Možnostmí měření doby výpočtu v Javě se podrobně zabývá kapitola 4.8.1.

4.7.2 Spotřeba paměti

V programovacích jazycích nižších úrovní, jako je například C++, se o alokaci a uvolňování paměti stará programátor. Z pohledu aplikace je proto její spotřeba deterministická a opakovatelná. Naproti tomu v Javě se o uvolňování paměti stará garbage kolektor (viz kapitola 4.7.3). Ten funguje podle svých algoritmů a z pohledu aplikace zcela nedeterministicky. Opakované provedení stejného algoritmu na stejných datech může proto mít různou spotřebu paměti.

Chování garbage kolektoru je dokonce ovlivněno i dostupnou pamětí. Pokud je paměti málo, bude se kolektor spouštět častěji a maximální obsazená paměť bude nižší. Je proto potřeba ještě definovat, co je myšleno „spotřebou paměti“ v Javě.

Kromě paměti obsazené kvůli činnosti samotné aplikace je další paměť obsazena v důsledku činnosti virtuálního stroje. Tento druh paměti by bylo možné měřit z externí aplikace pomocí prostředků operačního systému. Pokud se rozhodneme měřit paměť spotřebovanou pouze činností samotné aplikace, máme k dispozici prostředky virtuálního stroje k měření paměti alokované na haldě. Obtížně měřitelná je paměť na zásobníku.

Možnostmi měření spotřeby paměti v Java aplikaci se podrobně zabývá kapitola 4.8.2.

4.7.3 Garbage kolektor

Činnost garbage kolektoru je jeden z hlavních faktorů ovlivňujících výkonnost aplikací v Javě a její měření. Jeho úkolem je uvolňovat objekty, které už nejsou potřeba. Z pohledu garbage kolektoru to jsou ty objekty, které již nejsou dosažitelné pomocí referencí z běžícího programu.

Algoritmů pro identifikaci nedosažitelných objektů je více a reálně se používá několik druhů garbage kolektorů [30]. Nejvýrazněji lze algoritmy rozdělit na:

- *sériové*
  Sériové algoritmy vyžadují pozastavení aplikace během jejich provádění. Jsou vhodné pro většinu aplikací, proto se následující text bude zaměřovat především na ně.
- *paralelní*
  Paralelní algoritmy fungují paralelně s běžící aplikací. Odstraňují nedeterministická zpomalení aplikace, což může být užitečné u aplikací reálného času nebo aplikací citlivých na zpoždění. Jejich použití má ale i své nevýhody, jako je například nižší efektivita.

V nových verzích Javy se virtuální stroj snaží volit optimální garbage kolektor, velikost haldy, nastavení JIT překladač a další parametry podle typu počítače. Konfiguraci lze vynutit zadáním parametrů virtuálního stroje.

Algoritmy garbage kolektoru

Algoritmy garbage kolektoru řeší především problém nalezení nedosažitelných objektů. Primitivní algoritmus je velmi jednoduchý. Kolektor nejprve vytvoří seznam přímo dosažitelných objektů (objekty odkazované ze zásobníku). Potom pomocí procházení grafu tvořeného referencemi mezi objekty označí všechny navštívené objekty. Všechny ostatní objekty, které zůstaly neoznačené, jsou nedosažitelné a kolektor je může uvolnit.

Problémem tohoto primitivního algoritmu je jeho časová náročnost. Reálné algoritmy se snaží používat různé heuristické metody ke zvýšení rychlosti vyhledávání. Důsledkem heuristických metod je nalezení pouze některých nepotřebných objektů. Nepotřebné objekty mohou proto přežít i několik běhů kolektoru.

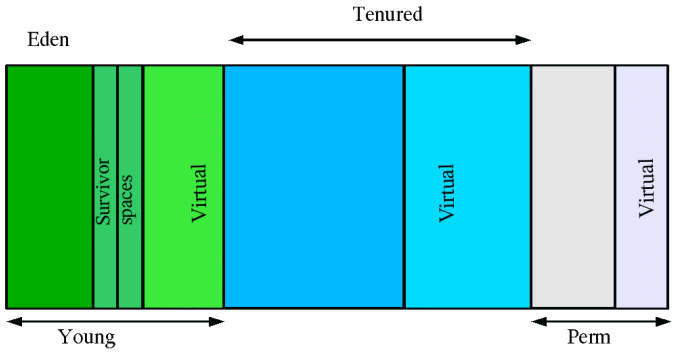
Základní heuristickou metodou je zavedení generací objektů. Tato metoda vychází z pozorování, že naprostá většina objektů přestává být potřebná velmi krátce po svém vzniku. Naopak objekty, které přežijí přes určitou časovou mez, zůstávají obvykle potřebné ještě velmi dlouho. Kolektor se proto zaměřuje zejména na vyhledávání mezi mladými objekty.

V praxi se používají tři generace objektů: mladá (young), starší (tenured) a permanentní (permanent) – obrázek 4.1 Kolektor vyhledává objekty po jednotlivých generacích, vždy když se daná generace naplní.

Nově alokované objekty přibývají do mladé generace. Po naplnění mladé generace je spuštěna minoritní kolekce (minor collection), která se zaměřuje pouze na mladou generaci. Obvykle je možné většinu objektů uvolnit, ty zbylé se přesunou do prostoru přeživších (součást mladé generace).

Prostory přeživších jsou v mladé generaci dva, jeden je vždy prázdný. Během provádění minoritní kolekce se přeživší objekty přesouvají do prázdného prostoru přeživších a původní obsazený prostor přeživších se vyprázdní. Po určitém množství přesunů objektu, nebo pokud je prostor přeživších zaplněn, je objekt poslán do starší generace.

Po naplnění starší generace doje k plné kolekci (full/major collection). Ta trvá déle a zabývá se objekty v mladé i starší generaci. Konkrétní algoritmus vyhledávání objektů může být pro každý typ kolekce jiný.



Obrázek 4.1: Generace objektů. Obrázek byl převzat z [30].

Nastavení garbage kolektoru

Výkonnost garbage kolektoru je velmi závislá na velikosti jednotlivých generací. Proto se v rámci činnosti kolektoru tyto velikosti dodatečně upravují, aby mohl kolektor fungovat optimálně. Čím je velikost mladé generace vyšší, tím je vyšší také efektivita kolektoru. Na druhou stranu se zároveň prodlužují pauzy aplikace při spuštění kolektoru. Velikost prostoru přeživších má na výkonnost kolektoru menší vliv.

Velikost generací a další konfiguraci chování garbage kolektoru lze upravovat pomocí parametrů virtuálního stroje. Některé užitečné parametry:

- `-verbose:gc`
  Vypisuje informace o každém spuštění garbage kolektoru.
- `-XX:+PrintGCDetails, -XX:+PrintGCTimeStamps`
  Rozšiřuje informativní výpisy o činnosti kolektoru o další informace.

- `-Xmx`
  Nastavuje maximální velikost haldy (reserved) a tím ovlivňuje maximální velikost generací.
- `-Xms`
  Nastavuje prvotní a zároveň minimální velikost haldy a tím ovlivňuje minimální velikost generací.
- `-XX:MinHeapFreeRation, -XX:MaxHeapFreeRatio`
  Definuje optimální aktuální velikost haldy na základě poměru volného a obsazeného prostoru. Při každém běhu kolektoru může být aktuální velikost haldy přizpůsobena v mezích maximální a minimální velikosti.
- `-XX:NewRatio`
  Poměr mladé a starší generace.
- `-XX:NewSize, -XX:MaxNewSize`
  Omezení velikosti mladé generace zdola a shora.
- `-XX:SurvivorRatio`
  Upravuje velikost prostoru pro přeživší.

Pokud je pomocí parametrů -Xmx a -Xms nastavena velikost haldy na shodnou (neměnnou) hodnotu, stane se chování garbage kolektoru lépe předvídatelné, protože se nebudou provádět změny aktuální velikosti haldy a generací.

Pokud je nastavena velikost mladé generace pomocí -XX:NewSize na příliš vysokou hodnotu, bude znemožněno provádění minoritní kolekce a každá kolekce bude plná.

Volání metody System.gc() naplánuje vždy plnou kolekci.

4.8 Automatizované metody měření výkonnostních parametrů Java aplikací

4.8.1 Metody měření doby výpočtu

Základem měření doby výpočtu je zjištění aktuálního času před výpočtem, zjištění aktuálního času po výpočtu a jejich odečtení. Jednotlivé metody se liší pouze ve způsobu, jakým je získán aktuální čas.

Metody získání aktuálního času

Nejvhodnější by bylo měřit dobu výpočtu pomocí procesorového času, tj. času běhu procesu na procesoru. Tento čas by měl dávat nejstálejší výsledky při opakování testu, protože není příliš ovlivněn ostatními procesy v systému. V Javě je ale problém tento čas získat. Použitím management API dostupným

v Javě lze sice tento čas získat, ale pouze s nízkým rozlišením. Měřením bylo zjištěno rozlišení (nejmenší interval) 15,6 milisekund. Kód získání času:

```
long time = ManagementFactory.getThreadMXBean()
        .getCurrentThreadCpuTime();
```

Druhou alternativou je měřit reálný čas, který lze v Javě získat buď v milisekundách, nebo v nanosekundách. Kód získání času v milisekundách:

```
long time = System.currentTimeMillis();
```

Rozlišení tohoto času je 1 milisekunda. Přesnost pro měření krátkých úseků je ale omezena kvůli občasnému provádění synchronizace.

Nejpřesnější je třetí způsob získávání času, v nanosekundách:

```
long time = System.nanoTime();
```

Rozlišení tohoto času je pouhých 513 nanosekund. Tento způsob je přímo určen pro měření krátkých časových intervalů.

Uváděná rozlišení jednotlivých časů se mohou na různých počítačích lišit. Procesorový čas nemusí být podporovaný vůbec.

Vybraná metoda získání aktuálního času

Vybraná metoda kombinuje výhody procesorového času a reálného času v nanosekundách. Základem je měření času v nanosekundách, které poskytuje výsledky s vysokým rozlišením. Výsledek je vždy porovnán s výsledkem získaným měřením procesorového času. Pokud se výsledky příliš liší (o nastavené procento), je výsledek ignorován a měření opakováno. Tím je částečně eliminován vliv ostatních procesů v systému.

Metodika měření

Čas bude měřen na delších fázích výpočtu, konkrétně u komparátorů se bude jednat o fázi načtení reprezentací a fázi porovnání. Přesnost měření může ovlivnit také spuštění garbage kolektoru, proto je nutné jeho spuštění během měření zabránit (popis implementace bude popsán v kapitole 4.8.3).

Před samotným měřením je nutné několikrát výpočet provést, aby se eliminoval vliv JIT kompilátoru. Druhou možností je JIT kompilaci vypnout pomocí přepínače virtuálního stroje `-Xint`.

4.8.2 Metody měření spotřeby paměti

Spotřeba paměti, podobně jako doba výpočtu, bude také měřena pro jednotlivé fáze výpočtu. Její měření je ale složitější, protože se (obecně) může během mě-

řené fáze nepředvídatelně měnit. Bude proto nutné blíže specifikovat, co je pod pojmem „spotřeba paměti“ myšleno.

Aktuálně obsazená paměť

O trochu jednodušším problémem je zjištění spotřeby paměti v jednom okamžiku, neboli zjištění aktuálně obsazené paměti. Metod k jejímu získání je několik.

Prvním způsobem je získání obsazené paměti prostřednictvím třídy Runtime:

```
long memory = Runtime.getRuntime().totalMemory()
        - Runtime.getRuntime().freeMemory();
```

Tento způsob není příliš přesný (nemá dostatečné rozlišení). Stejným problémem trpí i druhý způsob, kterým je použití management API Javy:

```
long memory = ManagementFactory.getMemoryMXBean()
        .getHeapMemoryUsage().getUsed();
```

Ukázalo se, že nejlepší metodou je získání obsazené paměti z informací garbage kolektoru dostupných přes management API:

```
com.sun.management.GarbageCollectorMXBean sunBean1 =
   (com.sun.management.GarbageCollectorMXBean)
   ManagementFactory.getGarbageCollectorMXBeans().get(1);
long memoryBeforeGC = 0;
long memoryAfterGC = 0;
GcInfo info = sunBean1.getLastGcInfo();
for(MemoryUsage e: info.getMemoryUsageBeforeGc().values()){
        memoryBeforeGC += e.getUsed();
}
for(MemoryUsage e: info.getMemoryUsageAfterGc().values()){
        memoryAfterGC += e.getUsed();
}
```

Tento kód funguje pouze pro virtuální stroj Oracle HotSpot VM [38]. Na začátku je získána bean třída s informacemi o garbage kolektoru provádějícího plnou kolekci. (V systému jsou dva kolektory, kolektor na indexu 0 obstarává minoritní kolekci.) Bean třída je přetypována na speciální rozhraní pro HotSpot VM kolektor. Díky tomu je možné získat informace o poslední kolekci (GcInfo). Informace obsahují obsazenou paměť pro jednotlivé generace a prostory před a po provedení kolekce.

Pro správnou funkci kódu je potřeba nastavit použití sériového garbage kolektoru. Kód získá hodnotu obsazené paměti v době před a po spuštění garbage kolektoru. Pro získání paměti v libovolném okamžiku výpočtu stačí vynutit spuštění kolektoru v tomto okamžiku (implementace je popsána v kapitole 4.8.3).

Další metoda používá úplně jiný přístup, než metody předchozí. Z popisovaných metod je tato metoda nejpřesnější, ale také nejnáročnější na implementaci,

proto nakonec nebyla v této práci použita. Základem je sledování vzniku a zániku jednotlivých objektů a udržování součtu jejich velikostí [39].

Velikost objektu lze například zjistit součtem velikostí všech jeho polí, které lze získat použitím reflexe. Přesnějšího výsledku lze docílit použitím instrumentace [19].

Sledování vzniku objektů lze implementovat pomocí aspektů, které mohou být navázány na volání konstruktoru. Sledování zániku objektů lze implementovat pomocí tzv. phantom referencí (popsáno v kapitole 4.8.3).

Definice spotřeby paměti výpočtu

Nyní je již vyjasněna metoda měření obsazené paměti v jednom okamžiku, stále je ale více možností definice spotřeby paměti celého výpočtu nebo jeho fáze. Nelze bez úprav využít jednoduchý algoritmus měření doby výpočtu a vypočítat spotřebu jen jako rozdíl obsazené paměti na začátku a na konci výpočtu.

Změny obsazené paměti jsou ovlivněny činností garbage kolektoru. I kdyby se jednalo o jazyk bez garbage kolektoru, pro dokonalý popis spotřeby paměti v časovém úseku je potřeba funkce, nikoliv jedna hodnota. Funkce je ale nepraktická pro účely vyhodnocení efektu optimalizací. Proto je potřeba tuto funkci převést na jednu nebo několik hodnot, které ponesou co nejdůležitější informace o spotřebě paměti.

Intuitivně je spotřeba paměti obvykle vnímána jako minimální paměť potřebná pro běh výpočtu. Pokud by se jednalo o program v jazyce bez garbage kolektoru, pak by bylo možné tuto spotřebu definovat jako maximum aktuálně alokované paměti během doby výpočtu. Správa paměti pomocí garbage kolektoru ale měření takové spotřeby téměř znemožňuje a pro praktické účely hodnota této informace klesá.

Pokud bychom se rozhodli ji přesto měřit, bylo by to možné jen s omezenou přesností. Přesnost je omezena hlavně z důvodu velké režie garbage kolektoru (nelze ho spouštět příliš často) a jeho nepředvídatelného chování (některé nedosažitelné objekty nemusí být při kolekci uvolněny). Přesnost by se teoreticky dala zvýšit nalezením vlastního mechanismu zjišťování nedosažitelných objektů.

Princip spočívá v dostatečně častém měření obsazené paměti (například na začátku a na konci každé metody) a zavolání garbage kolektoru vždy, když se spotřeba zvýší nad určitý práh. Po provedení kolekce se případně hodnota prahu zvýší na spotřebu paměti po kolekci plus nějaký krok, který určíme podle požadovaného rozlišení.

K častému měření paměti nelze využít metodu spojenou s garbage kolektorem. Pokud by nám stačilo nižší rozlišení, můžeme použít některou z prvních dvou popsaných metod pro měření aktuálně obsazené paměti. Pro zlepšení rozlišení by byla nejvhodnější metoda poslední, která využívá aspekty. Pomocí aspektů by bylo také možné zajistit spuštění měření na začátku a na konci každé

metody. Abychom měření co nejvíce urychlili, můžeme výpočet testovat několikrát a vždy zpřesnit krok a nastavit přesnější počáteční práh. Tím se sníží počet spuštění garbage kolektoru, což je časově náročná operace.

Metoda nebyla v této práci využita, zejména kvůli složitosti její implementace a díky použitelnosti následujících popsaných definic spotřeby paměti.

Druhou možností je definovat spotřebu paměti jako celkové množství alokované paměti v dané fázi výpočtu. I takto definovaná spotřeba má dobrou informační hodnotu, protože velikost alokované paměti ovlivňuje četnost spouštění garbage kolektoru, a tím je ovlivněna také doba výpočtu. Měření této spotřeby je velmi jednoduché za předpokladu, že před výpočtem je uvolněna veškerá paměť z předchozích výpočtů a během výpočtu nedojde ke spuštění garbage kolektoru. Jak toho docílit je popsáno v kapitole 4.8.3. Potom stačí jen změřit aktuální obsazenou paměť na začátku a na konci výpočtu a hodnoty odečíst.

Třetí definice bere spotřebu paměti výpočtu jako paměť obsazenou výpočtem vytvořenými datovými strukturami. Její měření je velmi podobné jako v předchozí definici, pouze je po výpočtu měřena paměť až po spuštění garbage kolektoru. Hodnoty spotřeby podle druhé a třetí definice jsou horním a dolním odhadem pro spotřebu podle první definice.

4.8.3 Jak ovládat garbage kolektor

Detekce spuštění garbage kolektoru

Spuštění garbage kolektoru lze sledovat díky informativním výpisům, které lze vynutit zadáním parametru virtuálního stroje (kapitola 4.7.3). Programově lze spuštění detekovat s využitím management API:

```
long count = 0;
for(GarbageCollectorMXBean bean:
        ManagementFactory.getGarbageCollectorMXBeans()) {
   count += bean.getCollectionCount();
}
```

Uvedený kód zjišťuje celkový počet spuštění garbage kolektoru od spuštění aplikace. Pokud se počet před výpočtem liší od počtu po provedení výpočtu, došlo během výpočtu ke spuštění garbage kolektoru.

Jak vynutit spuštění garbage kolektoru

Zavolání metody System.gc() naplánuje spuštění garbage kolektoru. Ten je spuštěn asynchronně. Na jeho spuštění je ale možné počkat, například pomocí phantom referencí.

Phantom reference [36] obsahuje referenci na libovolný objekt, která je ignorována garbage kolektorem. Pokud je objekt dosažitelný pouze pomocí phantom

referencí, může být uvolněn. Na událost uvolnění objektu lze dokonce reagovat, proto lze tyto reference využít k provedení úklidových operací.

Tato metoda je implementována v knihovně jlibs [7], která poskytuje jednoduché rozhraní pro synchronní zavolání garbage kolektoru:

```
jlibs.core.lang.RuntimeUtil.gc();
```

Metodu je třeba používat opatrně, protože čekání na garbage kolektor výrazně prodlužuje dobu výpočtu (testu).

Jak zabránit spuštění garbage kolektoru

Spolehlivě zabránit spuštění garbage kolektoru nelze, ale můžeme minimalizovat jeho pravděpodobnost a případné spuštění detekovat. Základem je zajištění dostatku volné paměti pro výpočet. Pomocí parametrů virtuálního stroje -Xms a -Xmx lze nastavit dostatečně vysokou hodnotu minimální a maximální velikosti paměti. Pomocí parametrů -XX:NewSize a -XX:MaxNewSize podobně nastavíme dostatečnou velikost mladé generace.

Před výpočtem uvolníme paměť synchronním zavoláním garbage kolektoru. Po provedení výpočtu zkontrolujeme, zda se během výpočtu garbage kolektor nespustil. Pokud ano, ignorujeme výsledek testu a test zopakujeme.

Jak donutit garbage kolektor uvolnit nepotřebnou paměť

Spuštění garbage kolektoru (včetně plné kolekce) negarantuje uvolnění veškeré nedosažitelné paměti. Experimentálně jsem nalezla způsob, jak uvolnit většinu paměti z předchozích výpočtů.

Metoda spočívá v zaplnění veškeré paměti až do vyhození chyby OutOfMemoryError a následném zavolání garbage kolektoru. Paměť nejprve plním bloky o velikosti 1 MB, poté i většími bloky. Fungování metody bylo ověřeno ziskem dostatečně přesných výsledků měření spotřeby paměti (kapitola 4.8.2).

4.9 Implementace měření výkonnostních parametrů komparátoru

4.9.1 Účel implementace

Pro vyhodnocení efektu optimalizací, které jsou úkolem této práce, jsem vytvořila aplikaci pro výkonnostní testy komparátoru. Efekt prováděných optimalizací je možné sledovat po každé změně kódu a celkový výsledek je získán z porovnání výkonnosti konečné a počáteční verze kódu. Efekt optimalizací lze vyčíst z přehledného reportu generovaného aplikací.

Testovací aplikaci jsem integrovala také do procesu plynulé integrace zajišťované aplikací Hudson. Spuštěním aplikace na zatíženém serveru se sice zkreslují výsledky doby výpočtu, ale přesnost měření spotřeby paměti zůstává zachována. Výkonnost komparátorů bude možné sledovat i v budoucnu po ukončení této práce.

4.9.2 Výstup testovací aplikace

Aplikace generuje dvě přehledné tabulky (viz kapitola 4.10.4) pro každou měřenou fázi výpočtu. Pro účely komparátorů byl výpočet rozdělen do dvou fází, na fázi načtení reprezentace („loader“) a fázi porovnání („cmp“).

Řádky tabulek obsahují naměřené výsledky pro různé metody měření. Aplikace měří dobu běhu výpočtu, celkovou alokovanou paměť a paměť obsazenou výslednými datovými strukturami.

První tabulka obsahuje absolutní naměřené hodnoty v jednotlivých verzích kódu. Druhá tabulka obsahuje pouze dva sloupce s daty. Význam prvního sloupce je zlepšení aktuální verze oproti verzi předchozí. Význam druhého sloupce je zlepšení aktuální verze oproti verzi počáteční.

Výsledky v tabulkách jsou vypočítány vždy ze sady testů pomocí jednoduchých statistických metod. Výsledky jednotlivých testů jsou dostupné ve formátu XML.

4.9.3 Architektura

Aplikace je spouštěna formou JUnit [12] testů. Tento způsob byl zvolen hlavně z důvodu snadnější integrace do Hudsonu. Proces testování navíc může selhat, když není možné docílit dostatečné přesnosti měření (např. kvůli špatné konfiguraci nebo zatížení počítače). V takovém případě je vyhozením výjimky docíleno selhání unit testu.

V procesu testování mají hlavní úlohu dva typy objektů: testovací případ (PerfTestCase) a tester (Tester). Testovací případ obsahuje implementaci testovaného výpočtu. Tester implementuje metodu měření, tj. získání hodnoty měřené veličiny před měřenou fází výpočtu a po ní. Tester dále rozhoduje o potřebném počtu opakování výpočtu každého testovacího případu a o úspěchu nebo neúspěchu měření.

Aplikace obsahuje několik metod měření, které jsou implementované jednotlivými testery. Pomocí řídícího objektu jsou spuštěny testovací případy postupně se všemi testery. Testovací případ informuje testera o počátku a konci jednotlivých fází výpočtu.

Výsledky testů jsou od všech testerů hromaděny v objektu typu ResultsGatherer, který zajišťuje jejich další zpracování a organizaci. Z naměřených hodnot je například vypočítán průměr a další statistické údaje.

4.9.4 Popis některých metod důležitých rozhraní

- `void Tester.startPhase(String phaseName)`
  Touto metodou oznamuje testovací případ testerovi, že začíná fáze výpočtu s názvem daným parametrem phaseName. Tester provede úklidové operace a zahájí měření.

- `void Tester.stopPhase(String message)`
  Touto metodou oznamuje testovací případ testerovi, že fáze výpočtu končí. Zpráva předaná parametrem upřesňuje okolnosti ukončení a je dohledatelná v XML výstupu aplikace. Tester získá výsledky měření.

- `void Tester.initAccuracyChecking()`
  Impuls od řídícího algoritmu, že má tester zahájit kontrolu přesnosti měření. Přesnost je měřena pro více jednotlivých měření najednou, protože jednotlivá měření dávají příliš malé vzorky pro kontrolu přesnosti.

- `boolean Tester.isAccuracyOk()`
  Řídící algoritmus touto metodou zjišťuje, zda měření proběhlo úspěšně (s dostatečnou přesností).

- `int Tester.getRepeatCount()`
  Řídící algoritmus touto metodou zjišťuje, kolikrát je potřeba každý testovací případ s použitím daného testera úspěšně opakovat. Různé testovací metody mohou vyžadovat různé množství pořízených dat pro dostatečnou přesnost měření.

- `void PerfTestCase.run(Tester tester)`
  Testovací případ v této metodě implementuje testovaný výpočet. Testera získaného jako parametr metody informuje o jednotlivých fázích výpočtu.

- `void ResultsGatherer.addResult(String tester, String phaseName, double startVal, double endVal, String message)`
  Metoda slouží k předání výsledku měření testera do objektu hromadícího výsledky. Parametry metody jsou název testera (testovací metody), název fáze výpočtu, počáteční naměřená hodnota, konečná naměřená hodnota, zpráva o způsobu ukončení výpočtu.

- `void ResultsGatherer.acceptResultsAndReset()`
  Metoda slouží ke schválení všech nových výsledků měření.

- `void ResultsGatherer.reset()`
  Metoda slouží k odstranění všech neschválených výsledků (z důvodu jejich nepřesnosti).

- `void ResultsGatherer.computeStatistics()`
  Výpočet statistických údajů z naměřených dat.

4.9.5 Implementace objektů typu Tester

- *NanoTester*
  NanoTester implementuje metodu pro měření doby výpočtu popsanou v

kapitole 4.8.1. Maximální možná odchylka reálného a procesorového času je parametrem konstruktoru. Díky tomu ji lze vhodně zvolit podle předpokládané doby výpočtu. Dalším parametrem konstruktoru je počet opakování testů, který ovlivňuje přesnost měření a dobu provádění testů. Konkrétní nastavení parametrů pro různé typy sad testovacích případů byly zvoleny na základě experimentů s přesností měření.

- *MemoryTester*
  MemoryTester implementuje dvě metody měření paměti popsané v kapitole 4.8.2, měření celkové alokované paměti a měření paměti obsazené výslednými datovými strukturami. Metody jsou velmi přesné dokonce bez nutnosti mnoha opakování testu. Na základě experimentů s přesností byl počet opakování zvolen na 20. Odchylka celkového měření je nižší než 0,1 procenta.

- *NullTester*
  NullTester je tester, který netestuje, ale slouží pouze k opakovanému provedení výpočtu před samotným měřením. Během této doby se provede JIT kompilace a výsledky následných měření budou ustálené.

4.9.6 Implementace objektů typu PerfTestCase

- *BigTestCase*
  Tento testovací případ je určen pro získání co nejpřesnějších výsledků za cenu dlouhé doby testování. V jednotlivých fázích se provádí najednou načtení a porovnání několika komponent, které zajišťují dostatečnou velikost testovacích dat.

- *MicroTestCase*
  Tento testovací případ provádí načtení a porovnání pouze jedné dvojice komponent. Je určen zejména pro testování bez JIT překladače, které trvá mnohem delší dobu.

4.9.7 Implementace objektů typu ResultsGatherer

- *ConsoleResultsGatherer*
  ConsoleResultsGatherer vypisuje výsledky testů do konzole. Je vytvořen pouze pro ladící účely.

- *BufferedResultsGatherer*
  BufferedResultsGatherer je obalovací objekt, který odstiňuje vnitřní ResultsGatherer od nutnosti schvalování výsledků. Neschválené výsledky jsou uchovávány v bufferu a po jejich schválení jsou přeposlány. BufferedResultsGatherer proto funguje jako dekorátor (návrhový vzor Decorator nebo Wrapper [25]).

- *StructureResultsGatherer*
  StructureResultsGatherer zajišťuje strukturování výsledků. Vytvořená struktura je vhodná pro serializaci do XML. V listech stromové struktury jsou

množiny souvisejících měření, ze kterých se počítají statistické údaje (implementováno třídou ResultSet).

Hodnoty jsou nejdříve filtrovány od neobvyklých hodnot (posuzováno podle vzdáleností od mediánu) a následně je ze zbylých hodnot vypočítán průměr.

4.9.8 Implementace řídících objektů

- *PerfTestsRunner*
  Řízení provádění testů má na starosti třída PerfTestsRunner. Před spuštěním testů je třída nakonfigurována pomocí sady testerů, sady testů a objektu pro shromažďování výsledků. Poté je spuštěno samotné testování.

  Testování probíhá postupně se všemi testery. Tester je spuštěn pro sérii testovacích případů a nakonec je vyhodnocena přesnost měření této série. Pokud nebylo dosaženo dostatečné přesnosti, musí být měření opakováno. Maximální počet souvislého opakování testu je nastaven na dostatečně vysokou hodnotu, aby za běžných podmínek nemohla být překročena. Pokud je aplikace špatně nakonfigurována, může dojít k úplné nefunkčnosti některého testeru a díky omezenému limitu nezůstane aplikace v nekonečné smyčce. Pro zvýšení pravděpodobnosti, že následná série bude úspěšná, zastaví se aplikace po chybě na určitou dobu závislou na počtu předchozích chyb.

  Pokud je série úspěšná, jsou získané výsledky akceptovány. V opačném případě se provede jejich odstranění. Každý tester udává potřebný počet úspěšných opakování série.

- *Konkrétní testovací scénáře*
  Konkrétní testovací scénář (JUnit test) sestaví množinu testů a množinu testerů. Před spuštěním testů zajistí také popis a identifikaci testování (SVN revizi, datum).

  Testovacích scénářů může probíhat více za sebou v různých procesech. Výsledky mezi scénáři se předávají pomocí serializace a deserializace objektu shromažďujícího výsledky.

  Pomocí třídy PerfTestsRunner je provedeno samotné testování. Následně jsou aktualizované výsledky serializovány a uloženy.

  Kromě popsaných testovacích scénářů systém obsahuje také dva scénáře pomocné, které zajišťují inicializaci před testováním a zpracování výsledků po testování. Inicializace spočívá ve smazání nebo přejmenování předchozího souboru s výsledky. Zpracování výsledků obsahuje výpočet statistických údajů a vygenerování HTML reportu.

4.9.9 Pomocné třídy a knihovny

- *SVNInfo*
  Třída zjišťuje aktuální revizi komparátoru v SVN. Ke svému fungování potřebuje mít možnost spustit příkaz svn na příkazové řádce: `svn info`

`http://svn.assembla.com/svn/jacc/trunk http://svn.assembla.com/svn/obcc/trunk -xml`.

Výstupem programu svn je XML s popisem aktuální verze projektů JaCC a OBCC. Toto XML je parsováno pomocí knihovny SAX.

- *Knihovna Xstream*
  Knihovna Xstream [5] slouží pro jednoduchou serializaci (a deserializaci) libovolných objektů do XML. Její použití je velmi jednoduché. Kód serializace:

```
XStream xstream = new XStream(new StaxDriver());
xstream.toXML(objectToSerialize,
        new FileOutputStream(storeFile));
```

  K serializaci se využívají informace o objektu získané z reflexe, proto nemusí serializovatelný objekt implementovat žádné rozhraní. Podobně jednoduchý je kód deserializace:

```
XStream xstream = new XStream(new StaxDriver());
MyObject myObject = (MyObject) xstream.fromXML(storeFile);
```

  Třída objektu musí samozřejmě odpovídat třídě serializovaného objektu.

4.9.10 Integrace do Hudsonu

Díky spouštění aplikace jako JUnit testy lze testování snadno spouštět pomocí dávkových skriptů a stejným způsobem také nakonfigurovat v Hudsonu. Aplikace má definované závislosti na všech testovaných projektech, proto bude testování spuštěno vždy při změně testovaného kódu.

Výsledky testů lze prohlížet přes webové rozhraní. Hudson umožňuje webové prohlížení souborů projektu. V kořenovém adresáři projektu je vygenerován soubor report.html s výsledky, který lze z webového rozhraní Hudsonu otevřít.

4.9.11 Ověření přesnosti měření

Přesnost měření byla ověřována několikanásobným spuštěním testů na stejné verzi kódu. Získané výsledky přesnosti jsou zobrazeny v tabulce 4.2 s výsledky měření.

Test optimalizovaná verze kódu (t2) byl spuštěn třikrát po sobě. Výsledky měření paměti se liší maximálně o 0,08 procenta, výsledky měření doby výpočtu maximálně o 0,7 procenta (bez JIT kompilace maximálně o 1,6 procenta).

4.10 Provedené optimalizace

Optimalizace provedené v rámci této práce byly zaměřeny pouze na část porovnání reprezentací OSGi komponent. Část načítání reprezentace nebylo možné

optimalizovat, protože souběžně probíhal vývoj její nové verze.

4.10.1 Oprava logování

Analýzou výkonnosti pomocí profileru byly odhaleny nápadně velké časové ztráty způsobené voláním metody toString() během porovnání metod. Zaměřila jsem se proto na způsob logování v komparátoru metod a odhalila jsem výrazné nedostatky. Logovací knihovně byl předáván vždy kompletně sestavený řetězec, který se tím pádem musel tvořit i při vypnutém logování.

Chybu jsem opravila využitím prostředků logovací knihovny pro formátování řetězce. Knihovně je předán formátovací řetězec a pole argumentů, nad kterými knihovna v případě potřeby (pouze při zapnutém logování) zavolá sama metodu toString().

Touto jednoduchou opravou se výsledky testů zlepšily na době výpočtu zhruba o 70% a na celkové alokované paměť téměř o 90%.

4.10.2 Optimalizace algoritmu porovnávání tříd

Manuální analýzou kódu jsem zjistila neefektivitu algoritmu porovnání tříd. Pro ukončení rekurze porovnání cyklické struktury tříd se používá historie tříd.

V původní verzi algoritmus používal zásobník právě porovnávaných tříd pro detekci a ukončení rekurze a kromě toho ještě samostatnou historii tříd pro ukládání výsledků porovnání tříd.

Ještě před zahájením měření efektu optimalizací (v rámci svých počátečních prací na projektech komparátorů) jsem tento algoritmus upravila, aby byla pro oba účely použita pouze jedna struktura – historie porovnaných tříd. Tato optimalizace není obsažena v naměřených výsledcích.

Následně jsem se pokusila optimalizaci ještě vylepšit změnou datové struktury, ale efekt byl záporný, proto jsem ponechala původní strukturu. Podařilo se mi ale snížit počet vyhledávání třídy v historii, což přineslo efekt na snížení testované doby výpočtu přibližně o 2%.

4.10.3 Optimalizace algoritmu porovnání seznamu metod

Během analýzy objektů v paměti jsem si všimla nesmyslně vysokého počtu výsledků porovnání metod. To mě dovedlo k závěru, že je potřeba optimalizovat algoritmus porovnání seznamu metod, ze kterého jsou výsledky porovnání metod vyžadovány.

Původní implementace algoritmu byla velmi nepřehledná a pravděpodobně i chybná. Do výsledné struktury relevantních výsledků porovnání ukládala ně-

které zbytečné výsledky. Algoritmus jsem proto kompletně předělala. Optimalizace se projevila v testech snížením doby výpočtu zhruba o 20% a poklesem velikosti vytvořených datových struktur o 75%.

Stručný popis nového algoritmu

V případě metod se nejprve vytvoří skupiny metod se stejným názvem a počtem parametrů. Každá skupina je rozdělena na dvě části reprezentující metody jednotlivých porovnávaných komponent. V rámci skupiny je potřeba porovnat každou metodu první komponenty s každou metodou druhé komponenty a sestavit celkový výsledek z výsledků dílčích. Dílčí výsledek je získán pro každou metodu skupiny tak, že je nalezen nejvýhodnější rozdíl od některé metody z opačné komponenty (v nejhorším případě se za rozdíl zvolí odstranění, protože metoda v opačné komponentě chybí). Dílčí výsledky pro metody druhé komponenty je potřeba obrátit.

Zjišťování, zda mají metody stejný název a počet parametrů, jsem upravila na použití komparátoru (ve smyslu rozhraní java.utils.Comparator) místo původní metody ověřující shodu. Algoritmus je společný pro metody i konstruktory, u kterých se nesmí porovnávat název, a je implementován v abstraktní třídě JMemberListComparator. Potomci této třídy inicializují komparátor podle svých potřeb. Díky nové možnosti řazení metod bylo možné zjednodušit a optimalizovat algoritmus vytváření skupin metod se stejným názvem a počtem parametrů.

4.10.4 Celkový efekt optimalizací

Celkový efekt optimalizací zachycují tabulky 4.2, 4.3 a 4.4. Na použitých testovacích datech byla doba výpočtu snížena přibližně o 86% (bez JIT kompilace o 82%), celková alokovaná paměť klesla o 94% a velikost výsledných datových struktur se snížila o 75%.

Naměřené hodnoty času jsou uvedeny v nanosekundách (měřeno na procesoru Core i7 2,8 GHZ) a hodnoty paměti v bytech. První sloupec hodnot představuje stav před optimalizací, další tři sloupce obsahují měření po optimalizaci.

	t1	t2	t2	t2
memoryAfterGC	5648819	1382040	1382112	1382000
memoryBeforeGC	558671387	31264887	31289283	31274556
time	314196111	35780523	35951433	36031711

Obrázek 4.2: Měření efektu optimalizací.

	t1	t2	t2	t2
time	4815033541	863386543	865446562	869594043

Obrázek 4.3: Měření efektu optimalizací bez JIT kompilace.

	from base	from previous
memoryAfterGC	75,53 %	0,01 %
memoryBeforeGC	94,40 %	0,05 %
time	88,53 %	-0,22 %

Obrázek 4.4: Měření efektu optimalizací - tabulka zlepšení posledního měření oproti měření prvnímu a předchozímu.

4.11 Zhodnocení výsledků

Provedením optimalizací se podařilo dosáhnout výrazného snížení doby běhu a spotřeby paměti porovnání reprezentací komponent. Použité nástroje pro analýzu kódu se ukázaly jako velmi přínosné, protože s jejich pomocí byly velmi rychle odhaleny problémy v kódu vhodné k optimalizaci.

Díky velmi přesnému měření efektu optimalizací pomocí vytvořené aplikace bylo možné posuzovat každou změnu kódu samostatně. Změny, které nevedly ke zvýšení výkonnosti, byly na základě špatných naměřených výsledků zamítnuty. Testovací aplikaci bude možné dále využít pro optimalizaci načítání reprezentace komponent.

5 Využití komparátorů při analýze rozdílů

5.1 Cíle a přínosy

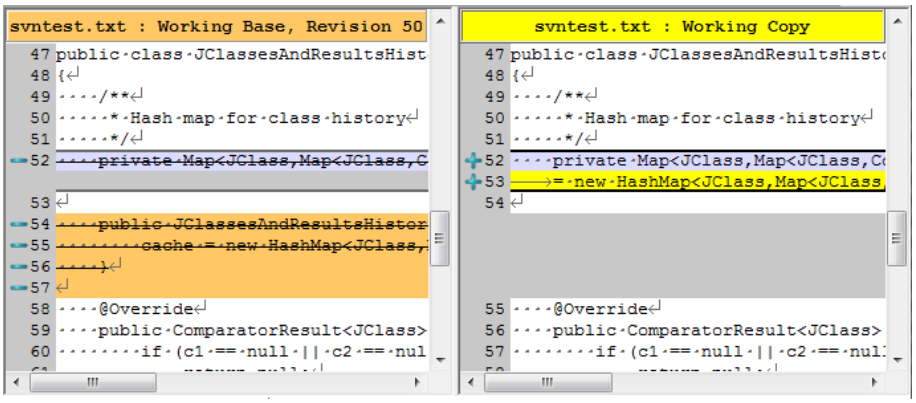
5.1.1 Grafické zobrazení rozdílů

Druhým cílem práce bylo rozšíření možností využití komparátoru. Komparátor je možné využít pro manuální analýzu rozdílu mezi komponentami. Za tímto účelem byl vytvořen grafický výstup, který přehledně zobrazuje jednotlivé rozdíly.

Grafický výstup byl integrován do nástroje pro automatické verzování (OBVS [14]). Uživatel verzovacího nástroje získá vysvětlení, z jakých důvodů nástroj určil konkrétní sémantickou verzi.

5.1.2 Další možné použití výstupu

Grafický výstup je možné dále využít k zobrazení rozdílů v kódu jako alternativa k SVN nástroji diff, který zobrazuje rozdíly dvou verzí souboru. Příklad výstupu SVN diff (TortoiseSVN) je na obrázku 5.1.



Obrázek 5.1: Náhled výstupu nástroje TortoiseSVN diff

Rozdíl mezi SVN diff a výstupem připraveným v této práci spočívá v principu porovnání kódu. SVN diff porovnává kód jako prostý text, bez jakékoliv znalosti jeho struktury. V textu se snaží vyhledávat podobné sekvence.

Výstup založený na komparátorech zobrazuje rozdíly získané porovnáním kódu se znalostí jeho struktury. Takové porovnání se nazývá porovnání založené

na modelu (model based). Komparátory vytvoří model obou komponent a tyto modely mezi sebou porovnají.

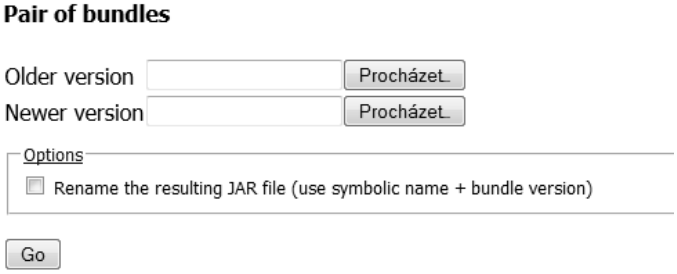
Získaný výstup odpovídá hloubce modelu. Informace, které nejsou součástí modelu (například kód uvnitř metod), nejsou porovnány. Výhodou je prezentace pouze podstatných informací, nevýhodou absence detailních informací.

Další výhodou porovnání založené na modelu je, že se dokáže vypořádat se zdánlivými změnami typu změna pořadí metod. Porovnání založené na textu tyto zdánlivé změny označuje jako změny. Naopak nevýhodou tohoto typu porovnání je možnost jeho použití pouze pro konkrétní programovací jazyk. Porovnání založené na textu je univerzální.

5.2 Projekt OBVS

Projekt OBVS (OSGi Bundle Versioning Service [14]) je nástroj pro automatické verzování OSGi komponent. Používá se prostřednictvím webového rozhraní. Uživatel prostřednictvím webového formuláře nahraje dvě OSGi komponenty, jednu v původní verzi, druhou upravenou a určenou k overzování. Nástroj pomocí komparátorů určí sémanticky správnou verzi pro upravenou komponentu. Overzovaná komponenta je uživateli nabídnuta ke stažení. Náhled formuláře pro zadání komponent je na obrázku 5.2.

Nástroj je nasazen na adrese *http://osgi.kiv.zcu.cz/obvs/index.html*.



Obrázek 5.2: Náhled rozhraní nástroje OBVS - formulář pro zadání komponent.

5.2.1 Architektura

Webové rozhraní je vytvořeno jako standardní J2EE aplikace, která používá servlety a JSP stránky. V současné době probíhá vývoj nového webového rozhraní, které bude založeno na technologii Spring. Pro své fungování využívá nástroj komparátory ve formě knihoven.

5.2.2 Způsob integrace zobrazení rozdílů

Pro technologie integrace je důležité, že současná i budoucí verze nástroje je psána v Javě a k zobrazování používá HTML, CSS a Javascript. Tyto technologie proto byly použity také k implementaci zobrazení rozdílů. Rozdíly se zobrazují na stránce zobrazené po overzování komponenty, odkud je možné overzovanou komponentu stáhnout.

Rozhraní

Grafické zobrazení rozdílů bylo implementováno jako samostatný projekt (kapitola 5.4) se závislostmi na komparátorech. Pro klientský program poskytuje velmi jednoduché rozhraní. Výstup je generován zavoláním jediné metody, které je předán parametr obsahující strukturu výsledku porovnání. Návratová hodnota je typu String a obsahuje HTML kód bez HTML hlaviček, aby ho bylo možné vložit bez úprav do webové stránky.

Tagy do HTML hlavičky je možné získat samostatně, opět jako String. Tyto tagy obsahují CSS styly a Javascript potřebný pro správné zobrazení výstupu. Tagy obsahují také odkazy na soubory stylů a skriptů. Do webových zdrojů proto byly vloženy potřebné obrázky a soubory s neměnnými CSS styly a Javascriptem.

Rizika nekompatibility

Vytvořený výstup by měl být integrován také do nové verze nástroje OBVS, proto byly k omezení rizika nekompatibility na úrovni technologií HTML, CSS a Javascriptu použity preventivní metody. Na úrovni HTML vzniká riziko konfliktu v hodnotách atributů id a name a v názvech tříd pro stylování. Riziko bylo omezeno volbou dostatečně specifických jmen a názvů tříd. Atributy id nejsou používány vůbec.

Na úrovni CSS hrozí zakrytí stylů. Buď mohou styly stránky ovlivnit styly zobrazení rozdílů, nebo naopak. Proto jsou použity dostatečně konkrétní selektory na obou stranách.

Na úrovni Javascriptu může dojít k přepsání globálních proměnných. V případě použití knihovny JQuery je obvykle jedinou globální proměnnou instance knihovny. Pokud by knihovna byla do stránky vložena dvakrát, je původní instance včetně nahraných pluginů ztracena. Řešením je uložení instance knihovny do jiné proměnné s dostatečně složitým názvem pomocí skriptu vloženého těsně za skript knihovny a pluginů. Vždy před použitím knihovny je instance zkopírována zpět do standardní proměnné.

5.3 Technologie

5.3.1 HTML

HTML (HyperText Markup Language) je značkovací jazyk pro text propojený odkazy. Je standardním jazykem pro vytváření webových stránek. V současné době je nejvíce rozšířena jeho verze 4, nejnovější verze 5 nemá zatím dostatečnou podporu mezi prohlížeči.

HTML má podobnou syntaxi jako XML, ale syntaktická pravidla jsou volnější (například není nutné psát ukončovací tagy). Použít lze omezenou sadu tagů se sémantickým (například) nebo prezentačním (například <b> nebo <font>) významem. Výborný tutorial technologie HTML je na [45].

Nástupcem HTML je XHTML (Extensible HyperText Markup Language), které vyžaduje dodržení syntaxe XML a odbourává použití prezentačních tagů. Stylování a způsob prezentace má být kompletně přesunuta na technologie CSS (kaskádové styly) a Javascript.

5.3.2 CSS

CSS (Cascading Style Sheets - kaskádové styly) je technologie pro stylování obsahu zapsaného ve formátu HTML, XHTML nebo XML. Hlavním úkolem je oddělení vzhledu dokumentu od jeho struktury a obsahu. V prohlížečích je běžně podporována jejich verze 2.

Styly se skládají z jednotlivých pravidel. Pravidlo obsahuje selektor a blok deklarací. Pomocí selektoru se určuje množina elementů, na které se pravidlo vztahuje. Blok deklarací popisuje formátování elementů.

Selektory mohou být velmi obecné (např. pro všechny odstavce) i naprosto konkrétní (např. podle id elementu). Čím je selektor více konkrétní, tím má vyšší prioritu. Další pravidla pro prioritu selektorů vychází z místa jejich deklarace nebo pořadí. Více se lze o CSS dozvědět na [44].

5.3.3 Javascript

Javascript je skriptovací jazyk, který má podobnou syntaxi jako C++ nebo Java. Poskytuje prostředky pro objektově orientované programování. Používá se zejména na webu, ale stále častěji také pro klasické aplikace. Příkladem je projekt Node.js [29], který používá Javascript na straně klienta i serveru.

Význam Javascriptu na webu spočívá v jeho interpretaci prohlížečem na straně klienta. To přináší různé výhody. Javascript umožňuje rychlou odezvu na akce uživatele, například kontrolu vyplněných hodnot nebo interaktivní nápovědu. Díky interpretaci u klienta nedochází tak často k požadavkům na server.

Tím se snižuje zatížení serveru i sítě.

Z pohledu programátora přináší Javascript zajímavé možnosti. Jedná se o slabě typovaný jazyk (weakly-typed), proměnné nemají pevně určený datový typ. Ke každému objektu lze přistupovat jako k asociativnímu poli. Objekty lze mimo jiné vytvářet pomocí zápisu v JSON (JavaScript Object Notation) formátu, který je velmi jednoduchý a dobře čitelný. V kombinaci s funkcí eval lze takto vytvořit objekt i z řetězce v JSON formátu, což se často používá pro komunikaci Javascriptu u klienta se serverem.

Kód v Javascriptu je organizován do funkcí. Globální kontext je brán jako funkce, která se zavolá během načítání stránky. Uvnitř je možné definovat další funkce, se kterými lze pracovat také jako s objekty. Zásadním rozdílem oproti jazykům, jako je například Java, je viditelnost proměnných. Java používá rozsah na úrovni bloků (block-level scoping), zatímco Javascript používá rozsah na úrovni funkcí (function-level scoping). To znamená, že proměnná je viditelná ze všech vnořených funkcí.

Podpora Javascriptu v prohlížečích je široká, ale v jednotlivých prohlížečích se může lišit rozhraní pro přístup k elementům stránky (DOM objektům). Proto je vhodné vždy používat knihovnu, která programátora od těchto rozdílů odstiňuje. Knihovna většinou navíc poskytuje mnoho užitečných funkcí. Příkladem takové knihovny je JQuery.

5.3.4 JQuery

JQuery [31] je lehká (lightweight) Javascriptová knihovna, která je podporována v naprosté většině prohlížečů. Její vývoj začal v roce 2006, proto je podporována i staršími prohlížeči, jako je například Internet Explorer verze 6. Knihovna se stala velmi oblíbenou a široce využívanou, proto lze očekávat její vývoj a podporu v nových verzích prohlížečů i do budoucna.

Knihovna je vydávána pod licencí GPL (GNU General Public License) nebo MIT (Massachusetts Institute of Technology). Lze ji proto svobodně používat v libovolném projektu.

Knihovna poskytuje podporu pro procházení a úpravy DOM (Document Object Model) objektů, reakce na události (akce uživatele), Ajax (technologie asynchronní komunikace se serverem), animace a mnoho dalšího. Další funkcionalitu lze přidat pomocí pluginů, kterých existuje velké množství.

Výhodou knihovny JQuery oproti jiným Javascriptovým knihovnám a frameworkům je malá velikost kódu, jednoduchost a snadné (intuitivní) použití, svobodná licence, velké množství funkcionality (částečně poskytované pluginy) a rozšířenost použití. Srovnání knihoven lze nalézt na Wikipedii [6].

5.3.5 Treeview

Treeview je plugin do knihovny JQuery pro zobrazování stromové struktury. Hlavní funkcionalitou je možnost rozbalování a zabalování (skrývání) jednotlivých větví struktury. Plugin dále zjednodušuje umístění ikonek k jednotlivým položkám struktury.

Struktura musí být tvořena vnořenými seznamy (tagy <ul>, <li>). Žádné další požadavky plugin nevyžaduje. Plugin zajišťuje základní stylování stromu, které lze jednoduše vylepšit pomocí vlastních stylů.

Treeview není jediný plugin pro vytvoření stromové struktury, ale jeho výhodou oproti ostatním je jeho velmi snadné použití, absence zbytečných funkcí a velmi jednoduchý kód.

5.4 Projekt prezentace rozdílů

5.4.1 Výstupy

Pro vytvoření grafického zobrazení rozdílů byl založen nový projekt s názvem obcc-presentation. Výstupy generované projektem budou mít širší uplatnění, než jen v rámci nástroje OBVS. Projekt generuje několik typů výstupu ve formátech HTML a XML.

Pro integraci do nástroje OBVS byl zvolen výstup s názvem htmlLimitedDepth (limitovaná hloubka). Alternativním HTML výstupem je výstup s názvem htmlFlat (plochý nebo lineární). Oba výstupy se snaží převést původní rekurzivní strukturu výsledků porovnání do lépe čitelné podoby.

Výstup htmlLimitedDepth

Struktura výsledků porovnání má charakter stromu získaného algoritmem DFS (Depth-first search). Taková struktura je pro člověka málo čitelná, protože zanoření větví stromu nerespektuje intuitivní zásady zanoření elementů OSGi komponent a Java typů. Přístup zvolený pro generování výstupu htmlLimitedDepth je snaha o transformaci struktury do stromu s úrovněmi Java balík – třída – metoda, na který je uživatel zvyklý například z vývojového prostředí.

Výstup je rozdělen na dva samostatné stromy. První zobrazuje rozdíly na úrovni OSGi elementů, druhý na úrovni Java typů. Některé úrovně původní struktury jsou vypuštěny. Strom elementů OSGi komponenty obsahuje úrovně: komponenta, OSGi balík nebo služba, Java balík, Java třída (s odkazem do příslušné větve ve stromu Java typů). Strom Java typů obsahuje úrovně: Java balík, třída, metody a členská pole, odkaz na typy parametrů a návratové hodnoty metod nebo typ pole. Aby byla orientace ve stromech uživateli maximálně

usnadněna, jsou použity pro jednotlivé větve ikony z vývojového prostředí Eclipse.

Rozdíly jsou zobrazeny na společném stromě pro obě komponenty pomocí rohových ikon. Tento systém je inspirován zobrazením rozdílů v aplikaci TortoiseSVN, konkrétně v nástroji diff v kombinaci se způsobem zobrazování změn v souborovém systému. Strom obsahuje všechny elementy, které jsou obsaženy alespoň v jedné z porovnávaných komponent. Pokud je element přítomný pouze v první komponentě, je ve stromu označen červenou ikonou ve tvaru křížku (element byl odebrán). Pokud je element přítomný pouze ve druhé komponentě, je označen modrou ikonou ve tvaru znaménka plus (element byl přidán). Pokud je výsledkem porovnání na některé úrovni mutace (MUT), je element označen červeným vykřičníkem. Ikony pro označování elementů byly převzaty z programu TortoiseSVN, ve kterém jsou použity pro označení změn v souborovém systému.

Větve stromu je možné zabalovat (skrývat) a rozbalovat. Uživatel má díky tomu jak souhrnný přehled o stavu jednotlivých balíků, tak si může rozbalit i detailní důvody rozdílů.

Výstup htmlFlat

Výstup htmlFlat převádí rekurzivní strukturu výsledků do struktury lineární (ploché). Jednotlivé výsledky jsou provázány se svým rodičem a potomky pomocí odkazů. Výstup byl inspirován profilovacím nástrojem pro Android Traceview [27].

Uživatel se pohybuje po jednotlivých uzlech stromu a má možnost se pomocí odkazu dostat na rodiče uzlu nebo jeho potomky. Zobrazeny jsou vždy jen detaily aktuálního uzlu. U ostatních uzlů je vidět typ elementu (pomocí Eclipse ikony), výsledek porovnání (pomocí TortoiseSVN ikony) a název. Všechny uzly jsou zobrazeny pod sebou bez definovaného pořadí a uživatel se může přepnout do libovolného uzlu.

Tento typ výstupu je určen spíše pro ladicí účely a k pochopení algoritmu porovnávání. Z těchto důvodů je struktura zachována v původní podobě.

XML výstupy

Pro oba popsané HTML výstupy existují také jejich alternativy ve formátu XML. Tyto výstupy jsou určeny pro další strojové zpracování, ale jsou také dobře čitelné pro člověka díky pečlivému odsazování a odřádkování.

5.4.2 Implementace

Architektura

Projekt je používán jako Java knihovna. Rozhraní pro použití knihovny bylo popsáno v kapitole 5.2.2. Popsané API obsahuje metody pro generování všech typů výstupu.

Výstupy jsou získávány ve formě řetězců, aby je bylo možné vložit do sestavovaného HTML nebo XML dokumentu jako jedna z jeho částí. HTML výstup se může odkazovat na další zdroje (obrázky, styly, skripty), jejichž správné umístění musí zajistit uživatel nebo klientská aplikace.

Univerzální vizitor

Výsledek porovnání je procházen pomocí jednotného mechanismu, který je založen na návrhovém vzoru vizitor (visitor pattern) [25]. Návrhový vzor vizitor je určen k průchodu strukturou a k provedení nějaké akce v každém uzlu. Objekty struktury musí obvykle implementovat rozhraní obsahující metodu visit (Visitor visitor), aby je vizitor mohl navštívit. Konkrétní implementace objektů musí tuto metodu implementovat a zavolat v ní správnou metodu vizitora pro své navštívení. Vizitor musí obsahovat metody pro navštívení každého typu objektu ve struktuře.

Vizitor implementovaný v tomto projektu nevyžaduje po objektech struktury implementaci žádného speciálního rozhraní. Ke svému fungování využívá reflexi. Podle typu objektu je schopen nalézt příslušnou metodu pro navštívení objektu sám. Díky tomu není ani vyžadována implementace metod pro všechny typy struktury konkrétním vizitorem.

Konkrétní vizitor dědí od třídy UniversalVisitor (implementace inspirována v [13]) a implementuje metody pro navštívení objektů potřebných typů. Metody musí splňovat jednotná pravidla pro pojmenování, aby byly nalezeny algoritmem vyhledávání metod, který je implementován ve třídě UniversalVisitor. Klient zavolá procházení struktury pomocí metody visit (Object object) implementované třídou UniversalVisitor. Tato metoda zavolá metodu pro třídu objektu, třídu jeho nejbližšího předka nebo nejbližší rozhraní.

Výhodou této implementace vizitora je možnost procházení libovolné struktury objektů. V tomoto projektu je toho využito k průchodu strukturou načtené reprezentace i strukturou výsledků porovnání. Konkrétní vizitor vytváří konkrétní výstup. Díky tomu jsou pohromadě části výstupu pro všechny typy objektů struktury.

Algoritmus tvorby výstupu htmlLimitedDepth

Výstup htmlLimitedDepth je tvořen pomocí několika instancí rekurzivních vizitorů (RecursiveResult2HTMLVisitor). První vizitor je spuštěn pro kořenový výsledek porovnání. Vizitor si sám určuje, zda se mají navštívit také potomci výsledku. Když je navštíven výsledek porovnání tříd, je vytvořen nový vizitor začínající na tomto výsledku. Původní vizitor už dále do hloubky nepokračuje.

Instance vizitorů jsou organizovány řídící třídou (ResultHTMLController). V té se postupně nahromadí vizitory pro každou porovnávanou třídu. Řídicí třída zorganizuje vizitory do jednotlivých balíků a spojí vygenerovaný výstup.

Klientská část

Z pohledu serveru je výstup obyčejný řetězec, který má být vložen do HTML stránky. Na klientské straně při interpretaci v prohlížeči je výstup formátován pomocí HTML, CSS a Javascriptu.

Vygenerované HTML obsahuje vnořené netříděné seznamy. Pomocí Javascriptu, konkrétně pluginu Treeview knihovny JQuery, jsou seznamy transformovány do stromu s rozbalovatelnými větvemi. Javascript je také použit k finálním úpravám struktury, které by bylo obtížné implementovat na úrovni Javy (vizitorů). Pomocí kaskádových stylů jsou uzlům struktury přiřazeny ikony.

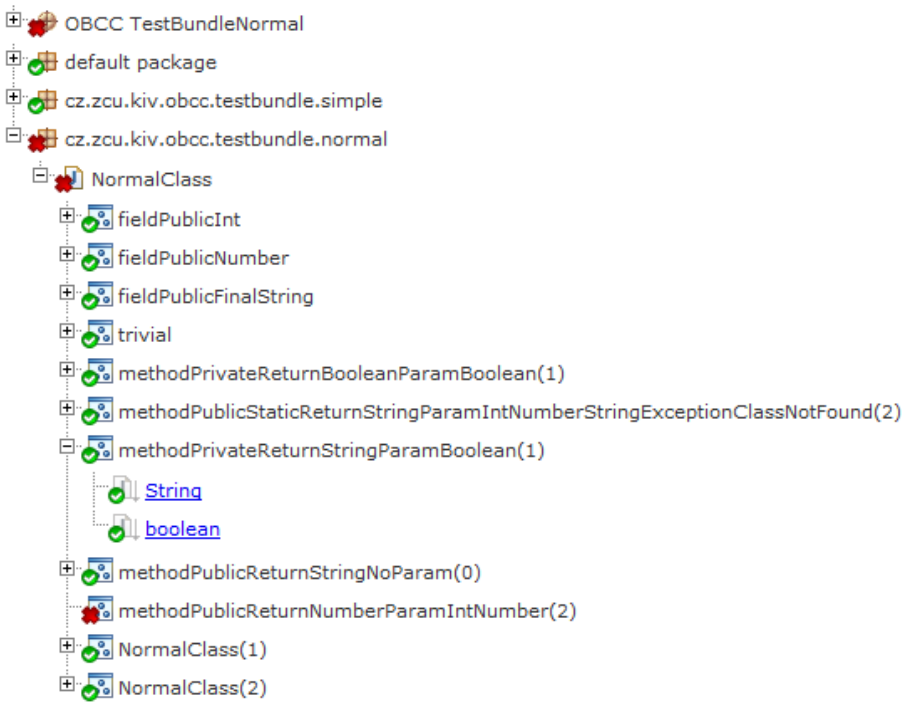
Výsledná grafická podoba

Obrázek 5.3 zobrazuje náhled výsledné podoby grafického výstupu htmlLimitedDepth. Z obrázku je patrné, že v nové verzi porovnávané komponenty byla smazána metoda methodPublicReturnNumberParamIntNumber třídy NormalClass z balíku cz.zcu.kiv.obcc.testbundle.normal.

5.4.3 Možnosti rozšíření

HTML výstup by bylo možné rozšířit o další informace o jednotlivých uzlech. Například by mohl být přidán odkaz na javadoc dokumentaci, pokud bude k dispozici.

Místo webového rozhraní nástroje OBVS by bylo možné vytvořit rozhraní ve formě klientské aplikace, která by posílala komponenty k overzování formou webových služeb. Tím by se usnadnilo použití nástroje pro uživatele. Grafický výstup by bylo možné zaslat do klientské aplikace spolu s overzovanou komponentou.



Obrázek 5.3: Náhled grafického výstupu htmlLimitedDepth.

5.5 Přínosy a další využití

Nástroj pro automatické verzování (OBVS) byl rozšířen o grafické zobrazení rozdílů mezi komponentami. Uživatel nyní kromě overzované komponenty získá také vysvětlení, proč byla určena konkrétní sémantická verze. Důvody pro nutnost změny verze lze v grafickém zobrazení velmi snadno vyhledat.

Grafický výstup by bylo možné dále využít i v jiných aplikacích, kde se využívají komparátory. Například by bylo možné zobrazovat důvody nekompatibility komponent při resolvingu nebo pokusu o nahrazení komponenty v komponentovém frameworku.

Dalším námětem je využití grafického výstupu při vyhledávání komponenty. Grafický výstup poskytuje uživateli orientační informaci o kvantitě rozdílů. Uživatel by se proto mohl lépe rozhodnout, kterou komponentu vybrat, aby byly úpravy nutné k obnovení kompatibility co nejmenší.

Grafický výstup by mohl být dále využíván jako alternativa k SVN nástrojům. Uživatel by pomocí něho dokázal snadno vyhledat všechny změny v rozhraní komponenty. Výstup by bylo možné také připojovat k informacím o vydané komponentě (release notes).

6 Závěr

6.1 Shrnutí práce

Seznámila jsem se s konceptem komponent obecně a s komponentovým modelem OSGi. Získané poznatky jsem shrnula v kapitole 2.

Prostudovala jsem projekty týkající se nahraditelnosti komponent vyvíjené na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. V kapitole 3 jsem zpracovala přehled projektů, vývojových nástrojů a způsobu práce na projektech. Popsala jsem také nedostatky a chybějící funkce komparátoru. Do práce na projektech jsem se sama zapojila. Kapitolu 3 lze využít také jako příručku pro další studenty, kteří na projektech začínají pracovat.

Na základě ruční analýzy kódu i měření výkonnosti pomocí několika nástrojů jsem analyzovala časovou a paměťovou náročnost implementace komparátoru a nalezla kritická místa v kódu. Tato místa jsem pomocí vhodných vylepšení optimalizovala. Pro zjištění efektu optimalizací jsem vytvořila aplikaci, která měří rozdíl výkonnosti aktuální verze oproti předchozím verzím.

Pro použitá testovací data klesla doba běhu optimalizovaného výpočtu porovnání zhruba o 88%, velikost vytvořených datových struktur o 75% a velikost celkové alokované paměti během výpočtu o 94%. Měření výkonnosti komparátorů jsem zapojila do procesu continuous integration, aby bylo možné sledovat vývoj výkonnosti i do budoucna.

Optimalizovanou implementaci komparátoru jsem využila k rozšíření nástroje pro automatické verzování OSGi komponent (OBVS). Po overzování komponenty jsou graficky zobrazeny rozdíly mezi původní a verzovanou komponentou. Díky tomu jsou na první pohled patrné důvody pro zvýšení sémantické verze komponenty.

6.2 Přínosy

6.2.1 Přínosy pro komparátor

Díky snížení paměťových a časových nároků komparátoru se zvýšila použitelnost komparátoru na výkonově omezených zařízeních. Výkonnost může být dále zlepšována a sledována díky jejímu automatickému měření. S použitím aplikace pro měření výkonnosti lze provést další optimalizace a snadno vyhodnotit jejich přínosy.

Vytvořený grafický výstup zobrazující rozdíly mezi komponentami má široké možnosti uplatnění. Lze ho například využít v komponentovém frameworku pro

vysvětlení důvodu nekompatibility. Jiná uplatnění dávají prostor pro další náměty na využití komparátorů.

6.2.2 Osobní přínosy

Díky této práci jsem se blíže seznámila s výhodami a možnostmi komponentového přístupu při vývoji aplikací a s technologií OSGi. Při práci na projektech komparátorů jsem poznala různá úskalí práce v týmu a nutnost dostatečné komunikace mezi členy týmu.

V části práce zabývající se optimalizací komparátoru jsem se detailněji seznámila s fungováním garbage kolektoru a dalších faktorů ovlivňujících výkonnost Java aplikace. Tyto znalosti mi mohou pomoci při řešení problémů v budoucnu. Vytvořenou aplikaci pro měření výkonnosti můžu využít také pro další projekty.

6.3 Náměty pro další práci

6.3.1 Optimalizace načítání reprezentace

Během této práce nebylo možné optimalizovat část komparátorů načítající reprezentaci OSGi komponent, protože souběžně probíhal vývoj její nové verze. Podle výsledků měření je nyní doba běhu i spotřeba paměti při načítání reprezentace výrazně vyšší, než při následném porovnání. V budoucnu je proto vhodné provést optimalizace také této části.

6.3.2 Nové nástroje

Námětem do budoucna je vytvoření aplikace pro vyhledávání komponenty na základě dotazovací komponenty. Systém by mohl porovnat potenciálně vhodné komponenty s dotazovací komponentou a seřadit je podle míry kompatibility. S tím souvisí nutnost rozšíření výsledku porovnání komponent o kvantifikaci rozdílů. Uživateli by mohly být také zobrazeny rozdíly rozhraní formou implementovaného grafického výstupu.

Dalším námětem je vytvoření aplikace, která by nahrazovala nebo doplňovala nástroj SVN diff při verzování OSGi komponent. Účelem aplikace by bylo zobrazení pouze takových rozdílů mezi verzemi OSGi komponenty, které mají vliv na její rozhraní. Uživatel by tak mohl lépe kontrolovat provedené změny a rozhodovat o vydání komponenty.

Seznam zkratek

API	Application programming interface
DOM	Document Object Model
GPL	GNU General Public License
HTML	HyperText Markup Language
IDL	Interface Definition Language
JaCC	Java Class Compatibility checker
JAR	Java archive
JDK	Java Development Kit
JIT	Just-in-time
JSON	JavaScript Object Notation
MIT	Massachusetts Institute of Technology
OBCC	OSGi Bundle Compatibility Checking toolset
OSGi	Open Services Gateway initiative
SDK	Software Development Kit
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language

Literatura

[1] *Exception Handling Rules*. Dokumentace projektu OBCC.
http://www.assembla.com/spaces/obcc/wiki/ExceptionHandlingRules.

[2] *JaCC representation tests*. Dokumentace projektu JaCC.
http://www.assembla.com/spaces/jacc/documents/bvV0jCtv4r4kq0eJe5cbLr/download/bvV0jCtv4r4kq0eJe5cbLr.

[3] *OBCC Eclipse code formatter*. Dokumentace projektu OBCC.
http://svn.assembla.com/svn/obcc/trunk/extras/src/main/resources/obcc/OBCC-eclipse-src-formatter.xml.

[4] *Thread-safety analysis results*. Dokumentace projektu JaCC.
http://subversion.assembla.com/svn/obcc/documentation/trunk/thread-safety-analysis-results.ods.

[5] *Xstream project*, 2011.
http://xstream.codehaus.org/.

[6] *Comparison of JavaScript frameworks*, 2012. Wikipedia
http://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks.

[7] *JLibs project*, 2012.
http://code.google.com/p/jlibs.

[8] OSGi Alliance. *Semantic Versioning, Technical Whitepaper*, 2010.
http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf.

[9] OSGi Alliance. *OSGi Alliance*, 2012.
http://www.osgi.org.

[10] Apache. *Apache Felix*, 2011.
http://felix.apache.org.

[11] Felix Bachmann et al. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[12] Kent Beck, Erich Gamma, and David Saff. *JUnit*, 2012.
http://www.junit.org/.

[13] Jeremy Blosser. *Reflect on the Visitor design pattern*, 2007. Java World, Infoworld, Inc.
http://www.javaworld.com/javaworld/javatips/jw-javatip98.html.

[14] Přemysl Brada. *Automated Semantic Versioning for OSGi Bundles*. OSGi Community Event 29th September 2010.

[15] Premysl Brada. Enhanced type-based component compatibility using deployment context information. *Electronic Notes on Theoretical Computer Science*, 279(2):17–31, December 2011. Proceedings of Formal Approaches to Software Component Applications (FESCA), satellite event of European Conference on Theory and Practice of Software (ETAPS).

[16] Premysl Brada and Lukas Valenta. Practical verification of component substitutability using subtype relation. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*, pages 38–45. IEEE Computer Society Press, 2006.

[17] Zuzana Burešová. *OBCC performance tests on Android*, 2011. Dokumentace projektu OBCC.
http://www.assembla.com/spaces/obcc/wiki/Performance_tests_on_Android.

[18] Zuzana Burešová, Veronika Dudová, and Karel Hovorka. *Felix resolver integration*, 2011.
http://www.assembla.com/spaces/obcc/wiki/Felix_resolver_integration.

[19] Neil Coffey. *Instrumentation: querying the memory usage of a Java object*, 2011.
http://www.javamex.com/tutorials/memory/instrumentation.shtml.

[20] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, September/October 2011.

[21] The Apache Software Foundation. *Apache Maven*, 2012.
http://maven.apache.org.

[22] The Eclipse Foundation. *Eclipse Equinox*, 2012.
http://www.eclipse.org/equinox/.

[23] The Eclipse Foundation. *Eclipse Memory Analyzer*, 2012.
http://www.eclipse.org/mat/.

[24] The Eclipse Foundation. *Eclipse Test and Performance Tools Platform Project*, 2012.
http://www.eclipse.org/tptp/.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[26] Google. *Android*, 2012.
https://developers.google.com/android.

[27] Google. *Profiling with Traceview and dmtracedump*, 2012. *http://developer.android.com/guide/developing/debugging/debugging-tracing.html*.

[28] Assembla Inc. *Assembla*, 2012. *http://www.assembla.com*.

[29] Joyent Inc. *Node.js*, 2012. *http://nodejs.org/*.

[30] Sun Microsystems Inc. *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*, 2003. *http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html*.

[31] The jQuery Foundation. *JQuery*, 2012. *http://jquery.com/*.

[32] Jiří Kučera. *Úložiště komponent podporující kontroly kompatibility.* Diplomová práce, Západočeská univerzita, 2011.

[33] Luminis. *EZdroid*, 2012. *http://www.ezdroid.com*.

[34] J.D. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. *Performance Testing Guidance for Web Applications.* Microsoft, 2007.

[35] Bohdan Mixánek. *Rozšírení nástroje pro verzování OSGi komponent.* Diplomová práce, Západočeská univerzita, 2011.

[36] Oracle. *Java SE 6 API Specification: Class PhantomReference*, 2011. *http://docs.oracle.com/javase/6/docs/api/java/lang/ref/PhantomReference.html*.

[37] Oracle. *Hudson*, 2012. *http://hudson-ci.org/*.

[38] Oracle. *Java SE HotSpot at a Glance*, 2012. *http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html*.

[39] David J. Pearce, Matthew Webster, Robert Berry, and Paul H.J. Kelly. *Profiling with AspectJ. Software Practise and Experience*, 1(1), 2005. *http://homepages.ecs.vuw.ac.nz/~djp/files/SPE05.pdf*.

[40] Jaroslav Plzák. *Získání reprezentace OSGi komponent.* Diplomová práce, Západočeská univerzita, 2009.

[41] The Knopflerfish Project. *Knopflerfish*, 2012. *http://www.knopflerfish.org*.

[42] Naiyana Tansalarak and Kajal Claypool. Finding a needle in the haystack: A technique for ranking matches between components. In *Proceedings of 8th International Symposium on Component-Based Software Engineering (CBSE 2005)*, volume 3489 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

[43] The TortoiseSVN Team. *TortoiseSVN*, 2012.
http://tortoisesvn.net/.

[44] W3schools. *CSS Tutorial*, 2012.
http://www.w3schools.com/html/.

[45] W3schools. *HTML Tutorial*, 2012.
http://www.w3schools.com/html/.