Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Diplomová práce

Získávání metrik o komponentách z distribučních balíků

Plzeň, 2014 Bc. Jan Šmajcl

Prohlášení

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

V Plzni dne 10. května 2014

Jan Šmajcl

Abstract

Obtaining component metrics from distribution packages

The aim of this work is the design and implementation of set of metrics for software components. Values of metrics should be obtained from distribution packages.

The theoretical part contains a brief introduction into measurement and software product metrics including the description of product metrics relevant to the goals of this work. This part introduces the CRCE component repository and describes the tools for bytecode analysis.

The practical part selects component metrics for implementation. Measurement of these metrics is implemented into CRCE Metrics plugin. This plugin is verified both in terms of correctness and computation time.

Abstrakt

Získávání metrik o komponentách z distribučních balíků

Cílem práce je navržení a implementace sady metrik pro softwarové komponenty. Hodnoty metrik mají být počítány na základě distribučních balíčků těchto komponent.

Teoretická část je zaměřena na oblast měření v oblasti software a produktových metrik. Součástí je i přehled a popis vybraných produktových metrik, které jsou relevantní k této práci. Dále je zde popsán koncept úložiště CRCE a nástroje pro analýzu bytecode.

Praktická část je tvořena výběrem komponentových metrik (mj. soudržnost, provázanost, složitost API, složitost metod), které jsou následně implementovány do úložiště CRCE v podobě pluginu. Součástí je i ověření funkčnosti pluginu z hlediska správnosti výpočtu a doby potřebné pro výpočet hodnot měřených metrik.

Poděkování

Velice děkuji doc. Ing. Přemyslu Bradovi, MSc., Ph.D. za ochotné a trpělivé vedení této práce.

Děkuji také své ženě, která byla trpělivá a zvládla to se mnou i v době, kdy jsem měl hlavu plnou učení.

Obsah

1 Úvod

Myšlenka vývoje softwaru pomocí komponent pochází z konce šedesátých let minulého století a dnes je již značně rozšířená. S komponentovým přístupem, stejně jako s jinými přístupy k vývoji softwaru, je spojena potřeba měření a výpočet hodnot odpovídajících metrik.

Na katedře informatiky na fakultě aplikovaných věd Západočeské univerzity v Plzni je prováděn výzkum v oblasti analýzy a vizualizace komponent. Referenční výzkum probíhá především na komponentách implementovaných v jazyce Java.

1.1 Motivace a cíl práce

Součástí výzkumu na katedře informatiky je i implementace speciálního komponentového úložiště CRCE. Jeho hlavní funkcionalitou je vedle ukládání komponent především verifikace komponentových aplikací z hlediska kompatibility a nahraditelnosti komponent.

V případě, že porovnáváme různé nahraditelné komponenty mezi sebou, je dobré mít představu i o jejich vlastnostech. K tomu mohou sloužit například produktové metriky jednotlivých komponent. Ty zachycují atributy jednotlivých komponent, které charakterizují z hlediska složitosti API a vnitřní struktury. Cílem práce je vytvoření pluginu do úložiště CRCE, který bude hodnoty metrik počítat.

Jednotlivé komponenty jsou do úložiště ukládány v podobě distribučních balíčků. Proto je nutné, aby bylo úložiště schopné vypočíst hodnoty metrik z těchto distribučních balíčků a ne na základě zdrojového kódu, ze kterého se často počítají. Zdrojový kód většinou již nebývá k dispozici.

1.2 Struktura práce

Tuto práci lze rozdělit na teoretickou a praktickou část. Teoretickou část tvoří druhá až čtvrtá kapitola. Ve druhé kapitole se nejprve zabývám problematikou měření obecně a pak speciálně v oblasti vývoje softwaru a s tím spojených metrik. Součástí této kapitoly je i popis několika existujících nástrojů, které jsou schopné počítat hodnoty metrik ze zdrojového kódu nebo bytecode. Ve třetí kapitole přináším určitý přehled produktových metrik, na které buď dále navazuji v praktické části, nebo slouží k pochopení problematiky a jsou základem pro implementaci jiných metrik. Ve čtvrté kapitole se pak zabývám projektem CRCE a nástroji pro analýzu bytecode.

Praktickou část tvoří pátá až sedmá kapitola. V páté kapitole je výběr a definice metrik, jejichž výpočet jsem v rámci této práce implementoval. Součástí této kapitoly je i popis vlivu bytecode na hodnoty metrik. V šesté kapitole popisuji samotnou implementaci plu-

ginu do úložiště CRCE, který provádí výpočet jednotlivých metrik. Součástí implementace byla i úprava WebUI, která je v této části také popsána. V sedmé kapitole se pak zabývám ověřením implementovaného pluginu. Ověření jsem prováděl z hlediska správnosti výpočtu a z hlediska doby výpočtu metrik.

1.3 Typografické úpravy

Pro jasné odlišení některých konstrukcí v textu používám následující typografické úpravy:

Citace – Citace jsou vyznačeny *kurzívou*.

Vzorce – Matematické vzorce jsou psány také *kurzívou*.

`Program` – Kód programu, názvy tříd a metod jsou psány `neproporcionálním písmem`.

2 Produktové softwarové metriky

Měření a metriky jsou nedílnou součástí řady inženýrských disciplín. S nástupem softwarového inženýrství se staly metriky součástí i tohoto odvětví. Vedle samotného pochopení důležitých pojmů, kterými měření a metriky jsou, je pro jejich aplikaci důležitý i software pro jejich získávání a to buď ze zdrojového kódu v době vývoje softwaru, nebo v případě, že již zdrojový kód není k dispozici, z bytecode (v případě Javy nebo v případě .NETu z CIL).

2.1 Co jsou to metriky

Aby bylo možné pochopit samotný pojem metriky, je nejdříve zapotřebí pochopit význam a vlastnosti měření a měření v obecné rovině.

2.1.1 Měření

V [Fen91] je měření definován takto:

„Měření je proces, při kterém jsou přiřazovány čísla nebo symboly atributům entit z reálného světa a to způsobem, aby tyto entity charakterizovaly dle jasně definovaných pravidel."

Měření je spojeno se zachycováním informací o atributech nějaké entity. Entitou je myšlen objekt, jako např. člověk, auto nebo distribuční balík Java aplikace. Atributem je vlastnost, která nás u dané entity zajímá. Příkladem může být výška u člověka, maximální dosažitelná rychlost u auta nebo počet požadovaných balíků u distribučního balíku Java aplikace.

Měření přiřazuje číslo nebo symbol atributům entit za účelem jejich charakteristiky. Předpokládá se, že když měříme např. výšku lidí, bude dané číslo tím větší, čím bude daný člověk vyšší. V závislosti na použitých jednotkách se může absolutní hodnota čísel lišit, ale v případě použití stejné jednotky bude vždy platit, že vyšší člověk bude charakterizován vyšší hodnotou výšky. Jednotlivé vlastnosti měření se mohou lišit v závislosti na úrovních a typech měření.

2.1.2 Úrovně měření

Dle [Kan02] je možné měření provádět na čtyrech různých úrovních: nominální, ordinální, intervalové a poměrové.

Nominální měření probíhá zařazováním entit do předem vymezených kategorií. Příkladem může být rozdělení lidí do skupin mužů a žen. Jiným příkladem může být rozdělení metodik vývoje softwaru na metodiku ad-hoc, vodopádovou, spirálovou, iterační a agilní.

Na nominální úrovni nemají jednotlivé kategorie vzájemné vztahy ve smyslu, že by jedna byla větší/lepší než jiná. Jediné použitelné matematické operace jsou tedy rovno (patří do dané kategorie) a nerovno (nepatří do dané kategorie).

Ordinální měření rozšiřuje nominální měření o operaci porovnávání. Příkladem může být rozdělení příjmu zaměstnance na vysoký, střední a nízký. Díky tomuto rozšíření je možné nejen řadit entity do skupin, ale navzájem tyto skupiny porovnávat. Navíc platí tranzitivní závislost, kde pokud skupiny A > B a B > C, pak A > C. Nedá se ale již říct, o kolik je jedna skupina větší/lepší než jiná. Pokud bychom např. jako hodnotící kritérium použili nějakou formu známek 1 až 5, neznamená to, že rozdíl mezi známkami 1 a 2 je stejný jako rozdíl mezi známkami 3 a 4 (nebo u příkladu příjmu, že rozdíl mezi vysokým a středním příjmem je stejný jako mezi středním a nízkým příjmem). Použitelnými matematickými operacemi jsou rovno, nerovno, větší a menší.

Intervalové měření udává, jak velké rozdíly jsou mezi jednotlivými měřenými entitami. Příkladem může být datum udávaný v rocích, kde můžeme říct, že rozdíl mezi roky 1800 a 1900 je 100 let stejně tak jako rozdíl mezi roky 1900 a 2000. Co již ale není možné říct, že například při teplotě 20 °C je dvakrát tepleji než při teplotě 10 °C. Použitelnými matematickými operacemi jsou rovno, nerovno, větší, menší, plus a mínus.

Poměrové měření má navíc oproti intervalovému měření absolutní nulu. Příkladem může být věk osoby od 0 do 100 let. Díky přítomnosti nuly je možné provádět s poměrovým měřením veškeré matematické operace včetně násobení a dělení. Použitelnými matematickými operacemi jsou tedy rovno, nerovno, větší, menší, plus, mínus, krát a děleno.

2.1.3 Některé typy měření

V [Kan02] je poukázáno na fakt, že bez ohledu na úroveň měření, je z naměřených dat zapotřebí získat odpovídající informaci. K sumarizaci naměřených dat a jejich porovnání může sloužit řada různých typů měření.

Než zde dané typy měření popíši, je třeba se ještě zastavit u problematiky českého názvosloví. Zatímco v angličtině jsou slova ratio, proportion a rate snadno rozlišitelná, v češtině se pro tato tři slova většinou používá slovo poměr. Proto se pokusím použít pro jednotlivé typy měření slova, která adekvátněji odrážejí význam slova poměr.

Poměr (ratio)

Poměr (ratio) získáme tak, že vydělíme dvě čísla stejného typu. Čitatel a jmenovatel udávají v tomto případě dva různé údaje, které se vzájemně vylučují. Příkladem může být poměr pohlaví, který je se vypočte podle následujícího vzorce:

$$PP = \frac{PM}{PZ}$$

kde

PM je počet mužů,

PZ je počet žen.

V tomto případě, pokud je výsledný poměr menší než 1, je v populaci více žen než mužů. V případě výsledku většího než 1 je tomu opačně. Poměry se používají i v oblasti softwarového vývoje a příkladem může být poměr počet testerů/počet vývojářů, který se v různých firmách liší.

Úměrnost (proportion)

Na rozdíl od poměru, v případě úměrnosti/úměry (proportion) je čitatel taktéž součástí jmenovatele:

$$p = \frac{a}{a+b}$$

Příkladem použití může být míra naplnění nádoby (nádoba naplněna z jedné poloviny).

Dalším rozdílem oproti poměru je i využitelnost úměrnosti. Zatímco poměr je nejlépe využitelný pro dvojici čísel, u úměrnosti tento problém nenastává. Pokud bychom měli čtyři různá čísla, která by dohromady dávala celek N:

$$a + b + c + d = N$$

pak zároveň platí:

$$\frac{a}{N} + \frac{b}{N} + \frac{c}{N} + \frac{d}{N} = 1$$

Procenta (percentage)

O procentech (percentage) hovoříme v případě, že úměrnost vyjádříme ve smyslu na 100 jednotek (jmenovatel je normalizovaný na 100). Pojmem procento (percent) je myšleno „na sto". Úměrnost p je tedy rovna 100p procent (100p%).

V praxi jsou procentní hodnoty používány ke zveřejňování různých výsledků. Mnohdy jsou ale nesprávně používána. První chybou, která se často objevuje, je zveřejňování údajů bez dodatečné informace, z jak velkého množství dat byl výsledek naměřený. Druhou chybou je měření na příliš malém vzorku (u softwaru např. na příliš malém množství softwarového produktu). Procentní hodnota vypočítaná z takového vzorku nebývá stabilní.

Míra (rate)

Na rozdíl od poměru, úměrnosti a procent, které poskytovaly pohled na danou entitu (nebo úkaz) ve fixním čase, koncepce míry (rate) je spojena se změnou dané entity. Míra vyjadřuje změnu jednoho množství (y) na jednotku jiného množství (x), na kterém je dané množství (y) závislé. Příkladem může být míra porodnosti v daném roce:

$$MP = \frac{PN}{PZ} * K$$

kde

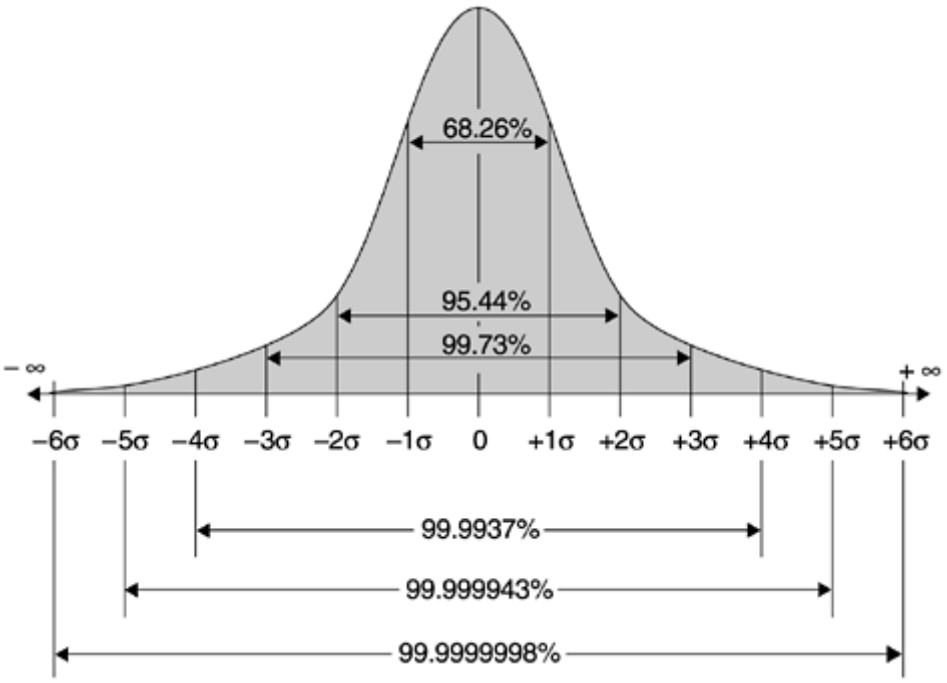
PN je počet narozených,

PZ je počet žen ve věku, kdy mohou mít děti,

K je konstanta, zpravidla 1000.

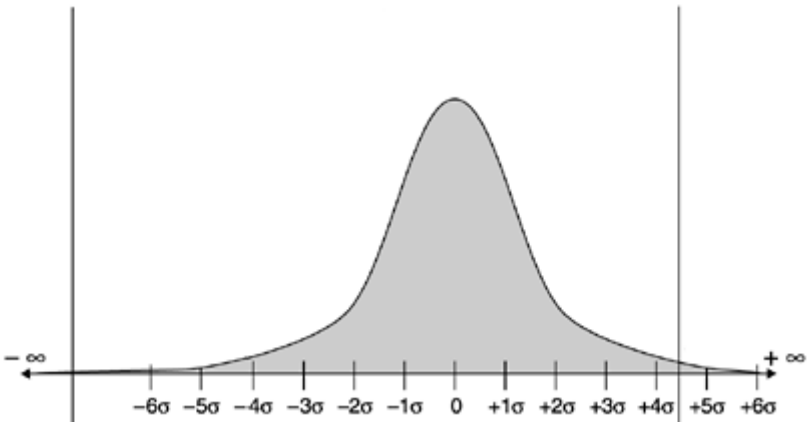
Six Sigma

Ukazatel six sigma představuje určitou úroveň kvality. V průmyslu byl zaveden ke konci osmdesátých let devatenáctého století společností Motorola. V dnešní době je six sigma považována za průmyslový standard. Sigma je symbol řecké abecedy a je používán pro standardní odchylku. Obrázek 2.1 ukazuje oblast, která je definována normálním rozdělením. Parametry rozdělení nemají vliv na procentní rozdělení oblastí, které vymezuje. Oblast pod křivkou definovanou plus mínus standardní odchylkou (sigma) od střední hodnoty představuje 68,26%. Oblast definovaná plus/minus šesti sigma je 9,9999998%.



Obrázek 2.1: Oblast definovaná normálním rozdělením (převzato z [Kan02])

Standard six sigma povoluje pouze 0,002 vadné součástky na 1 milion vyrobených. V praxi se většinou používá posunutá six sigma (obrázek 2.2), kde při posunu 1,5 sigma je kladené omezení pouze 3,4 vadné součástky na 1 milion vyrobených.



Obrázek 2.2: Posunutá Six Sigma (převzato z [Kan02])

2.1.4 Spolehlivost a validita

V případě, že máme definovaný způsob měření, jak spolehlivá je daná metoda měření? Dává daná metoda odpovídající výsledky s odpovídající kvalitou? Nejdůležitějšími kritérii jsou v tomto případě spolehlivost a validita.

Spolehlivost udává konzistentnost při opakování stejné metody měření na stejný subjekt. Pokud dosahujeme při opakovaném měření vysoké konzistentnosti nebo identický výsledek, hovoří se o vysokém stupni spolehlivosti. Měření libovolného subjektu může v sobě obsahovat chybu. Cílem je, aby bylo při měření dosaženo nejvyššího stupně spolehlivosti. Spolehlivost může být vyjádřena v podobě pomocí indexu variace:

$$IV = \frac{SO}{Pr}$$

kde

SO je standardní odchylka,

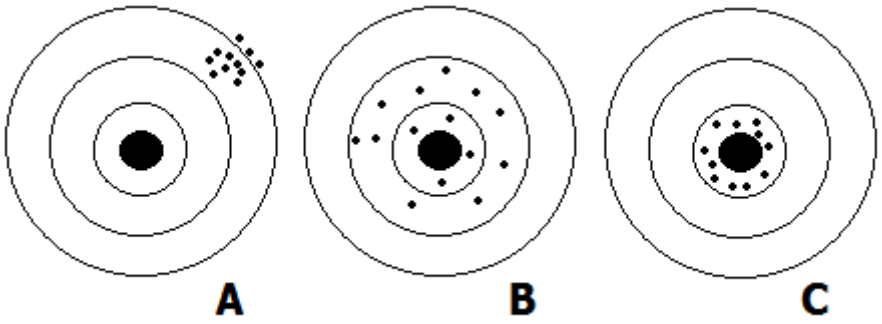
Pr průměr naměřených hodnot.

Čím je index variace nižší, tím měření vykazuje větší stupeň spolehlivosti.

Validita naproti tomu sleduje skutečnost, zda prováděné měření skutečně měří to, k čemu bylo určené. Jinak řečeno, reflektuje to, zda naměřené hodnoty odpovídají skutečnému záměru měření. V případech, kdy měření nezahrnuje žádný stupeň abstrakce, je pojem validita totožný s pojmem přesnost. Pokud sledujeme validitu v abstraktních oblastech, může to být značně obtížné. Například pokud bychom chtěli měřit počet věřících na základě toho, kolik lidí chodí do kostela, takové měření by neslo nízkou validitu, protože věřící může a nemusí chodit do kostela.

Je třeba zdůraznit, že validita a spolehlivost jsou dvě různé věci. To, že nějaké měření je spolehlivé neznamená, že je i validní a obráceně. Rozdíly se snaží zachytit obrázek 2.3. V části „A" je znázorněn případ, kdy jsou výsledky měření spolehlivé ale nejsou validní.

V části „B" je znázorněn případ, kdy jsou výsledky měření validní ale ne spolehlivé. A konečně v části „C" je znázorněn případ, kdy jsou výsledky validní i spolehlivé.



Obrázek 2.3: Spolehlivost a validita (předloha z [Kan02])

2.2 Měření v oblasti softwarového inženýrství

Na konferenci NATO Science Committee o softwarovém inženýrství [SE] bylo poukázáno na takzvanou „softwarovou krizi", která lze charakterizovat špatnou kvalitou softwaru, pozdními dodávkami a překračováním rozpočtů. Řešením této krize se má stát inženýrská disciplína zvaná softwarové inženýrství. Cílem je zavedení inženýrského přístupu do vývoje softwaru a to v oblastech řízení, oceňování, plánování modelování, analýzy, designu, implementace, testování a udržování.

Podobně jako jiné inženýrské disciplíny, i softwarové inženýrství potřebuje měření. Tom DeMarco v knize [DeM82] uvádí:

„Co není možné měřit, není možné řídit (ve smyslu managementu)."

Toto tvrzení je výstižné, ale ne dostačující. Jak je uvedeno v [Fen91], cílem měření není jen získání kontroly. Měření musí mít jasně definovaný cíl a musí být definovány entity a atributy, které budou měřeny. Ty jsou určeny cílem. Zatímco manažera může zajímat u dané komponenty např. cena implementace, inženýra může u stejné komponenty zajímat spíše časová náročnost provedení některé operace.

2.2.1 Softwarové metriky

Dle [Fen91] lze metriku charakterizovat takto:

„Metrika numericky charakterizuje „jednoduchý" atribut [části softwarového systému] jako délku kódu nebo počet nalezených chyb."

Softwarové metriky jsou používány v mnoha oblastech softwarového inženýrství. Mezi tyto oblasti patří např. odhadování cen a úsilí, měření produktivity, kontrola kvality, shro-

mažďování dat, měření kvality, modely spolehlivosti, vyhodnocování výkonu, algoritmická či výpočetní složitost, strukturální metriky a metriky složitosti.

2.2.2 Metriky procesní a produktové

Jednou možností dělení metrik, je dělení na metriky procesní a metriky produktové.

Procesní metriky se zaměřují na oblast procesu vývoje softwaru. Snahou je měřit atributy produktu a procesu relevantní k cíli vývojového procesu: Dodat včas za stanovenou cenu kvalitní produkt. Proto se procesní metriky vztahují k vlastnostem jako použitelnost či udržovatelnost. V rámci procesních metrik měříme atributy jako čas, cenu nebo úsilí. Dále s ohledem na kvalitu měříme počet chyb, selhání či změn. Vzhledem k tomu, že tato práce není o procesních metrikách, se jimi nebudu dále zabývat.

Produktové metriky mají za úkol zachycovat atributy konkrétního produktu, jako je např. třída nebo distribuční balíček. Snaží se charakterizovat tuto entitu. Příkladem může být počet tříd v distribučním balíčku (NOC) nebo McCabova cyclomatická složitost (CYCLO) jednotlivých metod.

2.2.3 Metriky přímé a odvozené

Dle [LMD06] lze metriky rozdělit do dvou skupin. Na metriky přímé, které jsou vypočítávány na základě zdrojového kódu (příp. bytecode) a na metriky vypočtené (odvozené), které se počítají z metrik přímých a mají často podobu různých poměrů.

Příkladem přímých metrik muže být metrika LOC (Lines of Code), jejíž hodnota je počet řádků zdrojového kódu. Jiným příkladem může být McCabova cyklomatická složitost (CYCLO), která počítá cyklomatickou složitost pro daný úsek kódu (např. pro určitou metodu či funkci).

Příkladem vypočtených metrik může být metrika strukturování tříd (NOM/NOC), která se vypočte jako počet všech metod (např. v distribučním balíku) děleno počtem tříd (ve stejném distribučním balíku).

2.2.4 Metriky rozhraní a metriky vnitřní

Metriky v oblasti komponentového vývoje lze rozdělit na metriky rozhraní a na metriky vnitřní (strukturální).

Metriky rozhraní se snaží zachytit vlastnosti zveřejněného rozhraní. Jak rozeberu dále, komponenty mají určitou část funkcionality zveřejněnou skrze veřejné API v podobě rozhraní nebo exportovaných package. Metriky rozhraní se snaží zachytit atributy těchto rozhraní či exportovaných package a to tak, aby každý takový prvek byl vhodným způsobem ohodnocený.

Metriky vnitřní naproti tomu vycházejí z vnitřní struktury komponenty. Základem je analýza struktury jednotlivých tříd (a jejich metod) a analýza jejich vzájemného propojení

(např. voláním metod). Příkladem může být měření vzájemné provázanosti tříd.

Výběrem konkrétních metrik pro implementaci se budu zabývat v části 5.1.

2.3 Nástroje pro získávání metrik ze zdrojového kódu

V době vývoje softwaru je důležité získání metrik ze zdrojového kódu. V této části popíši některé nástroje, se kterými jsem se setkal a prozkoumal jejich funkcionalitu. Nesnažím se zde zachytit všechny existující nástroje, ale popsat mou vyzkoušené nástroje. Z důvodu aplikace ve fázi vývoje mají všechny otestované nástroje podobu pluginu do vývojového prostředí. Veškeré nástroje, které v této části budu popisovat, počítají metriky na základě zdrojového kódu jazyka Java.

2.3.1 Eclipse Metrics plugin

První nástroj, pro získávání metrik, kterým jsem se zabýval, byl Eclipse Metrics plugin [EMp]. Pro mnoho vývojářů může být neocenitelná výhoda, kdy se metriky počítají průběžně s tvorbou zdrojového kódu. Díky tomu jsou neustále k dispozici aktuální hodnoty metrik.

Plugin se do Eclipse nainstaluje standardním způsobem přes Install New Software dialog. Po instalaci je zapotřebí udělat ještě dva kroky. Je zapotřebí pro každý projekt, ve kterém vývojář bude chtít metriky počítat, v jeho Properties metriky povolit. Po povolení jsou metriky přepočítávány po každém překladu kódu. Vzhledem k tomu, že u větších projektů může výpočet nějakou dobu trvat, není vždy vhodné počítat metriky pro každý projekt. Nyní je možné otevřít „Metrics View", kde jsou zobrazeny všechny vypočítané hodnoty metrik. Toto view má podobu rozevíracího se stromu, kde můžeme pro každou metriku vidět pro jeden prvek (v případě listu stromu) až po průměrnou hodnotu na úrovni projektu (pro kořen stromu).

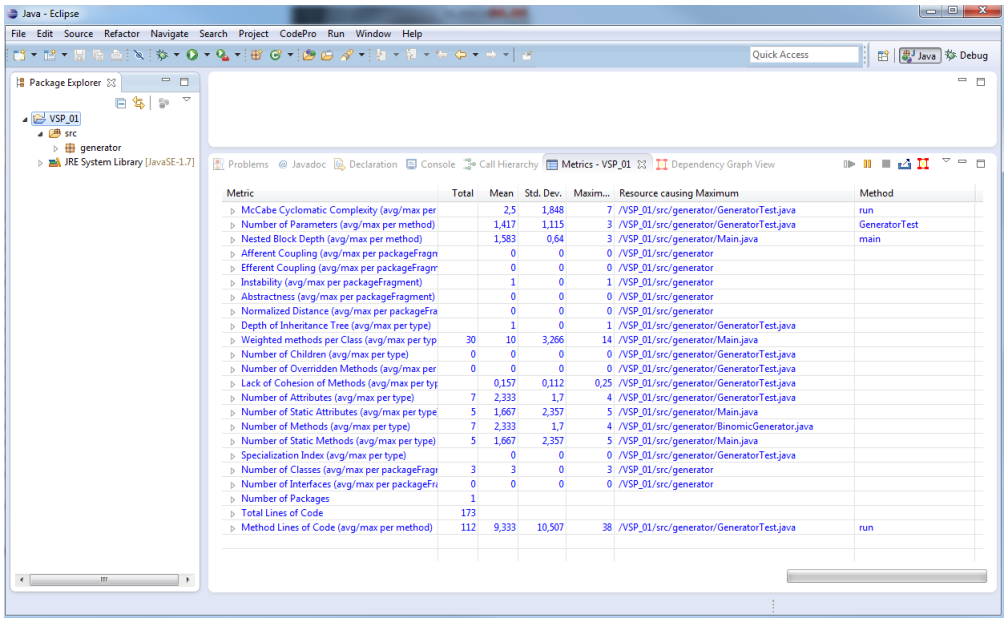
Plugin počítá řadu různých metrik. Počítá řadu běžných metrik jako jsou Number of Classes, Number of Children, Number of Interfaces, Depth of Inheritance Tree či Number of Methods spojených s objektovým návrhem. Nechybí ani metriky jako je Lines of Code či McCabe Cyclomatic Complexity. Plugin počítá i některé méně známé metriky jako jsou Afferent Coupling či Efferent Coupling které jsou popsány v [MaR02].

Vedle samotného výpočtu hodnot metrik umožňuje plugin analýzu závislostí balíčků, kterou je možné zobrazit v podobě grafu. U složitějších projektů může být tento graf značně nepřehledný. Proto plugin umožňuje i hledání nejkratší cesty.

Poslední nedílnou součástí pluginu, kterou bych rád zmínil, je export do Xml souboru. Často se může hodit pozdější analýza naměřených hodnot (ne jen prohlížení údajů). K tomuto účelu slouží export do Xml.

Jedinou zjevnou nevýhodou tohoto pluginu je fakt, že metriky nepočítá v případě, že nelze provést překlad projektu. V řadě případů je překlad prováděn jinými nástroji (např. maven) a není prováděn ve vývojovém prostředí. Pro takový typ projektů se tento nástroj

použít nedá.



Obrázek 2.4: Eclipse Metrics plugin

2.3.2 Metrics pluginy pro NetBeans

Vedle Eclipse je velmi oblíbené vývojové prostředí NetBeans. Proto jsem se snažil najít i odpovídající pluginy pro toto vývojové prostředí. Bohužel jsem nebyl tak úspěšný, jako v případě Eclipse. Povedlo se mi získat 2 různé pluginy: Simple Code Metrics plugin [SiCM] a SourceCodeMetrics plugin [SeCM].

Simple Code Metrics plugin je velice jednoduchý plugin. Po instalaci, která lze provést přes nastroj Plugins v NetBeans (po stažení pluginu z [SiCM]) se v prostředí NetBeans objeví ikona „SCM". Po stisknutí této ikony jsou vypočítány pro aktivní prvek ve stromu projektu metriky a to v podobě textového výpisu, který není nijak podrobný. Seznam podporovaných metrik není příliš dlouhý. Vedle běžných metrik jako Lines of Code, Number of Classes, Number of Methods či Cyclomatic Complexity stojí za zmínku ještě sada LCOM1 až LCOM4, kde každá metrika dává jiné hodnoty, ale nikde není odkaz na způsob, jak je ve skutečnosti počítána. Tato nedokumentovanost značně stěžuje využitelnost takového pluginu.

S oblastí dokumentace je na tom SourceCodeMetrics plugin ještě hůře. K pluginu není dokumentace vůbec žádná. Instalace se provádí stejným způsobem jako u předchozího pluginu (stažení z [SeCM]). Po instalaci se v kontextové nabídce nad projektem objeví možnost „Source Code Metrics". Po spuštění této volby je otevřeno okno s několika vypočtenými metrikami, které jsou spočítány pro projekt jako celek, pro jednotlivé balíčky a pro jednotlivé třídy. Bohužel seznam metrik, který je zde uveden jen v podobě zkratek, není příliš

vypovídající. Zkratky metrik jako „A", „C" nebo „D" bez jakékoliv dokumentace posouvají tento plugin za hranice použitelnosti.

Dle mého názoru ani jeden z těchto dvou pluginů není příliš vhodný k používání.

2.3.3 CodePro Analytix

Velice silným nástrojem, nejen pro oblast metrik, je plugin do Eclipse Google's CodePro Analytix [CPA]. CodePro Analytix obsahuje řadu nástrojů, které jsou určeny pro analýzu kódu a testování. Z hlediska této práce je zajímavý především nástroj pro výpočet metrik.

Plugin se instaluje do Eclipse standardním způsobem. Po instalaci se v kontextové nabídce objeví položka „CodePro Tools", která nabízí dvě možnosti pro aktivaci výpočtu metrik. První možností je „Compute Metrics", která odstartuje výpočet předdefinované sady metrik. Mezi těmito metrikami jsou metriky jako Number of Lines, Number of Methods nebo Number of Fields. Metrika Number of Classes se skrývá pod názvem Number of Types. Výstupem je dobře strukturovaný strom, který zobrazuje naměřené hodnoty a navíc je k dispozici vizualizace v podobě grafů, které odrážejí odpovídajícím způsobem jednotlivé naměřené hodnoty.

Druhou možností je aktivace výpočtu metrik přes volbu „Compute Metrics Using", která umožňuje výpočet předem nakonfigurované sady metrik. Výhodou tohoto řešení je, že vývojář může do výstupu zahrnout jen pro něho potřebné metriky.

Velice zajímavou vlastností může být i schopnost exportovat naměřená data do různých formátů výstupu, kde bych vedle Xml souboru zmínil například Html nebo prostý text.

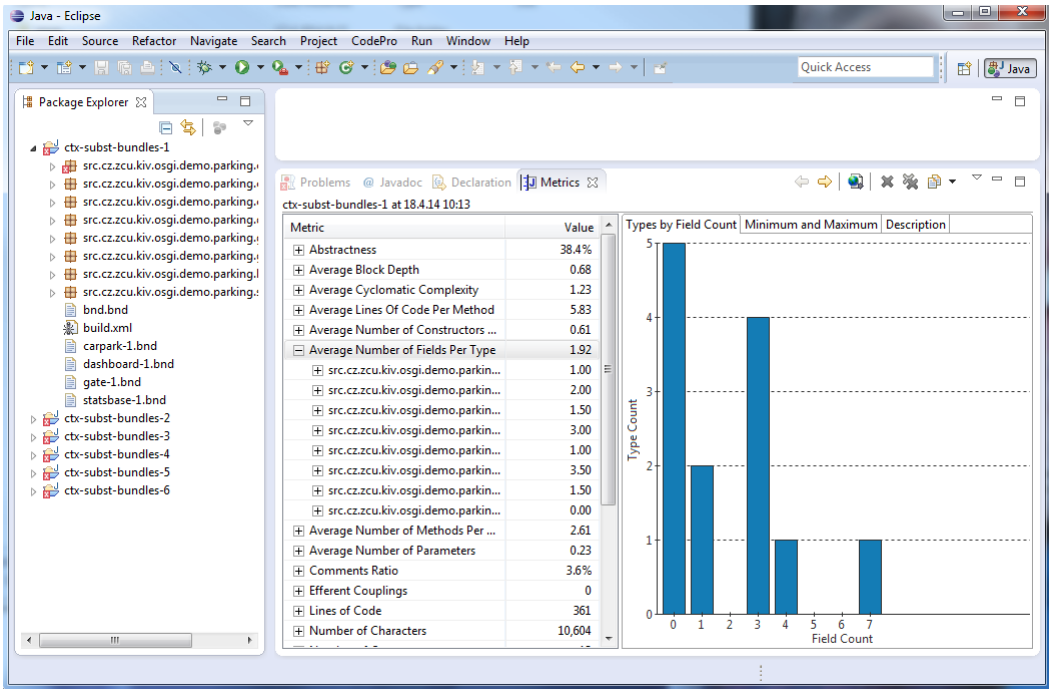
Pro srovnání s Eclipse Metrics pluginem, jedním z nástrojů CodePro Analytix je i velmi propracovaná analýza závislostí. V porovnání s ostatními testovanými nástroji je nástroj CodePro Analytix z hlediska funkcionality nejlepší volbou. Velkým kladem tohoto pluginu je kvalitní dokumentace a to včetně popisu jednotlivých metrik.

2.4 Nástroje pro získávání metrik z bytecode

Vedle získávání metrik ze zdrojového kódu může být zapotřebí získání odpovídajících hodnot z Java bytecode (např. z důvodu, že zdrojový kód není k dispozici). Hodnoty metrik lze získat z Java bytecode nebo, jak zde uvádím pro srovnání, i z .NET Common Intermediate Language (CIL). Stejně jako u nástrojů pro získávání metrik ze zdrojového kódu i zde uvádím nástroje, které jsem vyzkoušel.

2.4.1 Nástroj ckjm

Řada nástrojů pro výpočet metrik z Java bytecode má podobu jednoduché utility určené pro spouštění z příkazové řádky s velice omezenou funkcionalitou. Jako jednoho ze zástupců této skupiny bych uvedl nástroj ckjm ([CKJM]). Název je zkrácené Chidamber and Kemerer



Obrázek 2.5: CodePro Analytix

Java Metrics, což je sada metrik, které příslušný nástroj počítá.

Po stažení z [CKJM] není zapotřebí program nijak instalovat. Jedná se o v Javě psaný nástroj, který má jednoduchou funkcionalitu. Spouští se pomocí Java runtime java.exe a jako parametr se mu předává seznam class souborů, které má analyzovat. Pro každý class soubor vypočte 8 metrik. Vedle 6 metrik uvedených v [ChK94] jsou to ještě metriky Afferent Coupling a Number of Public Methods. Výstupem je vždy název třídy včetně package, do kterého přísluší, a řada 8 čísel. Údaje jsou bez popisu a proto je velice důležitá dokumentace, která je s nástrojem k dispozici.

Podle dokumentace by měl nástroj umět pracovat i s jar soubory, ze kterých si class soubory extrahuje, ale bohužel se mi při dodržení v dokumentaci popsaného postupu toto nepodařilo otestovat. Nástroj vypsal řadu neošetřených chyb bez rozumného vysvětlení, jak chybu opravit.

Výhodou nástroje může být možnost skládání spolu s dalšími programy pomocí kolony v příkazové řádce nebo, jak se uvádí v dokumentaci, využití s nástrojem Ant.

2.4.2 Komerční nástroj JArchitekt

Během studia nástrojů, které umožňují výpočet metrik na základě Java bytecode, jsem měl možnost otestovat komerční nástroj JArchitekt ([JAr]) a to ve 14-denní trial verzi. Nástroj je velice komplexní a je určen převážně pro fázi vývoje, kde napomáhá tvorbě kvalitního kódu, umožňuje porovnávání buildů nebo, pro tuto práci relevantní, výpočet

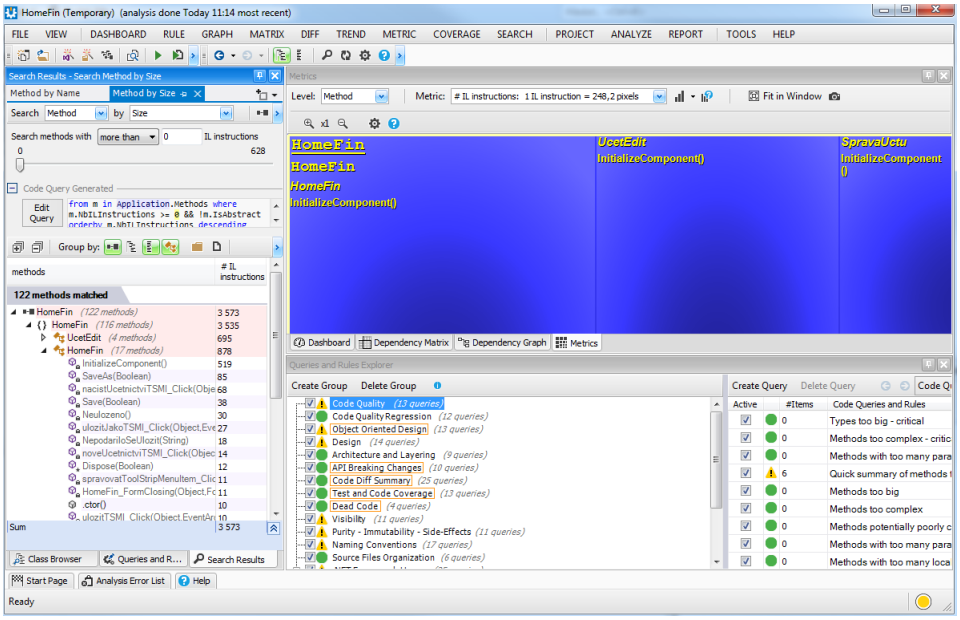
metrik.

Nástroj má propracované grafické rozhraní i verzi pro příkazovou řádku, která slouží pro integraci s dalšími nástroji. K dispozici je rozsáhlá dokumentace.

Metriky je možné počítat jak ze zdrojového kódu, tak z bytecode. Celkem je k dispozici 82 různých metrik, kde ale ne všechny metriky je možné počítat ze zdrojového kódu i z bytecode. Metriky je možné počítat na různých úrovních. Na úrovni projektu, balíčku, typu (třídy), metody nebo fieldu. Pro každou z těchto úrovní je k dispozici určitá sada metrik. Jako příklad metriky počítané jen z bytecode bych na úrovni metod uvedl výpočet počtu bytecode instrukcí. Jako příklad metriky dostupné pouze ze zdrojového kódu bych uvedl metriku Cyclomatické Complexity.

Nástroj má řadu způsobů zobrazení metrik, např. v podobě vizuálního dashboardu, formou grafů různých závislostí nebo jako tabulky vypočtených metrik. Na základě vypočtených metrik je možné si i zobrazovat největší či nejkomplexnější artefakty (např. metody) zkoumaného projektu.

Nástroj je velice komplexní a prozkoumání veškeré funkcionality by zabralo značné množství času. Vzhledem k tomu, že toto není těžištěm mé práce, se nebudu tímto nástrojem zabývat dále do hloubky.



Obrázek 2.6: NDepend

2.4.3 Komerční nástroj NDepend

V rámci průzkumu nástrojů jsem zkusil porovnat nástroje pro analýzu bytecode Javy s nástroji pro analýzou Common Intermediate Language (CIL), který je používán .NET

Frameworkem. Pro výpočet metrik ze zdrojového kódu se našla řada nástrojů včetně pluginů pro Visual Studio. Na rozdíl od nástrojů pro bytecode Javy, pro CIL byl výběr nástrojů velice limitovaný. Nakonec jsem mohl otestovat pouze komerční nástroj NDepend ([NDe]).

Po prvním spuštění je na první pohled patrná podobnost grafického rozhraní s nástrojem JArchitekt, který jsem popsal výše. Z dokumentace a webové prezentace není na první pohled patrné, zda se jedná o nástroj od stejného výrobce, nebo se jen výrobce inspiroval velice kvalitním nástrojem pro Javu.

Nástroj nabízí řadu pokročilých funkcí pro statickou analýzu jak zdrojového kódu (v podobě projektu Visual Studia), tak .NET assembly (CIL). Pro tuto práci podstatnou funkcionalitou je výpočet metrik. Ten se provádí identickým způsobem, jako u nástroje JArchitekt a má i odpovídající omezení (např. Cyclomatickou Complexitu je možné vypočítat jen ze zdrojového kódu).

3 Vybrané produktové metriky

V rámci historie vývoje softwaru se společně se změnami vývojových prostředků vyvíjela i oblast produktových metrik. Některé metriky jsou v oblasti vývoje softwaru používány celou řadu let (klasické metriky), některé byly vyvinuty s nástupem objektového přístupu (objektové metriky) a některé spojené s komponentovým přístupem jsou předmětem výzkumu posledních let (komponentové metriky).

V této části se snažím popsat metriky, ze kterých budu vycházet dále v práci a buď přímo souvisejí s měřením v oblasti vývoje pomocí komponent nebo jsou důležité pro pochopení dalších metrik.

3.1 Klasické metriky

Softwarové metriky vznikaly již v druhé polovině minulého století a odrážely v té době řešené potřeby. Přestože se od té doby značně změnily prostředky pro vývoj softwaru, tyto metriky jsou používány dodnes (někdy v mírně upravené podobě).

3.1.1 Lines of Code (LOC)

Jednou z nejznámějších metrik je metrika Lines of code (počet řádků kódu). Tato metrika se často používá pro určení velikosti projektu. Zpravidla je myšleno počet řádků zdrojového kódu (SLOC). Většinou se mluví o velikosti projektu v KLOC (1 KLOC = 1000 lines of code). Původně byla metrika určena pro programy psané v assebly language, kde platilo, že jeden řádek zdrojového kódu se rovná jedné instrukci. Jak se postupně vyvíjely vyšší programovací jazyky, jeden řádek zdrojového kódu odpovídal většímu množství strojových instrukcí. Následkem toho nebylo možné tento ukazatel používat k porovnání projektů v různých programovacích jazycích. Další problém byla specifika různých jazyků jako jsou začátek a konec bloku (otevírací a uzavírací závorky v Javě) a další variability psaní zdrojového kódu. Byl zaveden tedy ukazatel LLOC (logical lines of code), který se snaží odstranit rozdíly ve zvyklostech tím, že počítá počet generujících příkazů.

Rozdíl ukáži na příkladu. Mějme následující kód:

```
for (int i = 0; i < 100; i++) System.out.print("Ahoj");
```

Pro tento příklad platí, že LOC (SLOC) je rovno 1, zatímco LLOC je rovno 2. Pro srovnání kód přeformátujeme s použitím závorek (bez změny funkcionality):

```
for (int i = 0; i < 100; i++)
{
    System.out.print("Ahoj");
}
```

Rázem se dostaneme na 4 LOC (SLOC), zatímco LLOC zůstává stejně roven 2.

3.1.2 McCabova cyklomatická složitost (CYCLO)

Další velice známou metrikou je McCabova cyklomatická složitost. Tu v roce 1976 popsal T. J. McCabe v [McC76]. Navrhl metodu měření cyklomatické složitosti kódu, která měla sloužit především pro určení testovatelnosti programů. Do té doby byly snahy na udržitelnost a testovatelnost kódu směřovány na omezení délky programu (např. na 50 řádek kódu). McCabe poukázal na to, že 25 po sobě jdoucích nezávislých if-then podmínek může vést k 33,5 milionu nezávislých programových cest, ze kterých je možné testovat jen zlomek.

Cyklomatické číslo v klasické teorii grafů udává počet nezávislých kružnic v grafu. Při aplikaci na software, cyklomatickou složitostí se myslí počet lineárně nezávislých cest toku programu. V případě, že budeme mít program (nebo například metodu) s jedním vstupním a jedním výstupním bodem, cyklometrická složitost se vypočítá podle následujícího vzorce:

$$M = V(G) = e - n + 2 * p$$

kde

$V(G)$ je cyklomatické číslo grafu,

n je počet uzlů, kde uzlem grafu je neoddělitelná skupina příkazů (například tělo cyklu),

e je počet hran, kde orientace hrany odpovídá tomu, jak se budou jednotlivé skupiny příkazů po sobě provádět,

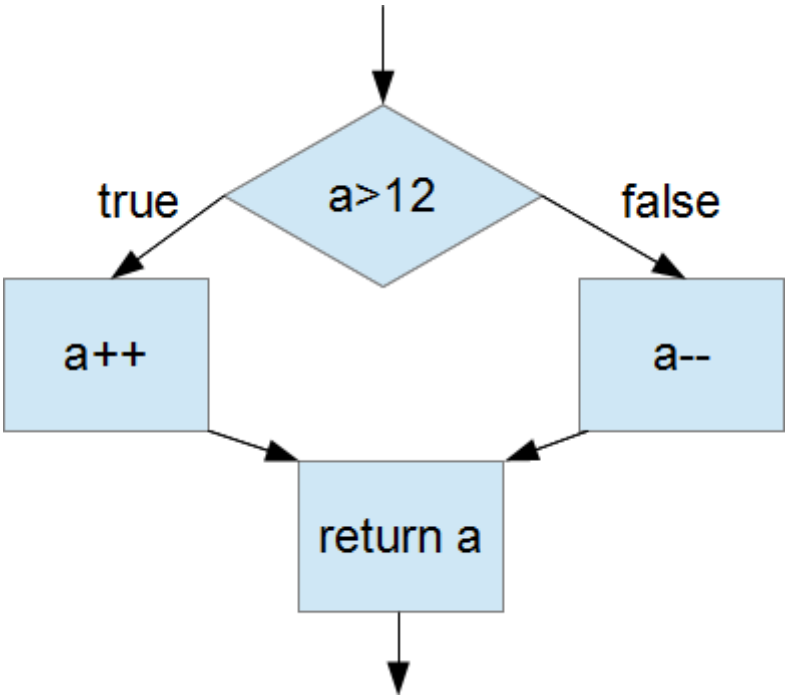
p je počet koncových uzlů nebo také počet napojených komponent.

Na cyklomatickou složitost se dá dívat také jako na počet binárních podmínek plus 1. Pokud nejsou všechny podmínky (rozhodnutí) binární, třícestná podmínka se počítá jako dvě binární a n-cestný příkaz select je započítáván jako $n-1$ binárných podmínek. Podobně podmínka iterace cyklu je považována za jedno binární rozhodnutí.

Výpočet ilustruji na následujícím příkladu:

```
if (a > 12)
{
    a++;
}
else
{
    a--;
}
return a;
```

Control flow diagram je znázorněn na obrázku 3.1. Graf má 4 uzly a 4 hrany. Cyklomatická složitost grafu je tedy $4 - 4 + 2 = 2$. Tento výsledek odpovídá i výpočtu metodou binárních podmínek, kde máme 1 binární podmínku (počet binárních podmínek $+1 = 2$).



Obrázek 3.1: Control flow diagram příkladu

3.2 Objektové metriky

S nástupem objektového přístupu v oblasti softwarového vývoje byla vyvinuta i řada odpovídajících metrik. Velké množství metrik je popsáno v [LMD06], ze které jsem čerpal v této části nebo například v [ChK94].

3.2.1 Počet operací (NOM)

Počet operací (NOM) (pro obecný programovací jazyk) nebo též počet metod (v případě Javy) udává počet všech operací/metod ve vyšetřovaném rozsahu (např. distribučním balíčku). Tato metrika se často používá v kombinaci s dalšími metrikami (např. počet tříd) pro výpočet příslušných poměrových vlastností. Java rozlišuje metody statické (volané na úrovni třídy) a instanční (volané nad konkrétními objekty). Každá metoda má ještě jeden ze čtyř modifikátorů viditelnosti (přístupu), které omezují jejich používání. Pro různé účely můžeme počítat určité skupiny metod (např. pouze veřejné).

3.2.2 Počet tříd (NOC) a počet balíčků (NOP)

Podobně jako můžeme počítat počet metod, lze počítat počet tříd a/nebo balíčků. V případě tříd v Javě je možné rozlišovat třídy s různými modifikátory viditelnosti (v případě vnější/hlavních tříd je výběr omezený). Dále Java umožňuje třídy vnořené, což je

třída uvnitř třídy. Ta může být deklarována navíc s modifikátory private nebo protected a může být statická (běžná třída nikoliv). Balíčky nemají žádné modifikátory, ale mají vliv na organizaci adresářové struktury.

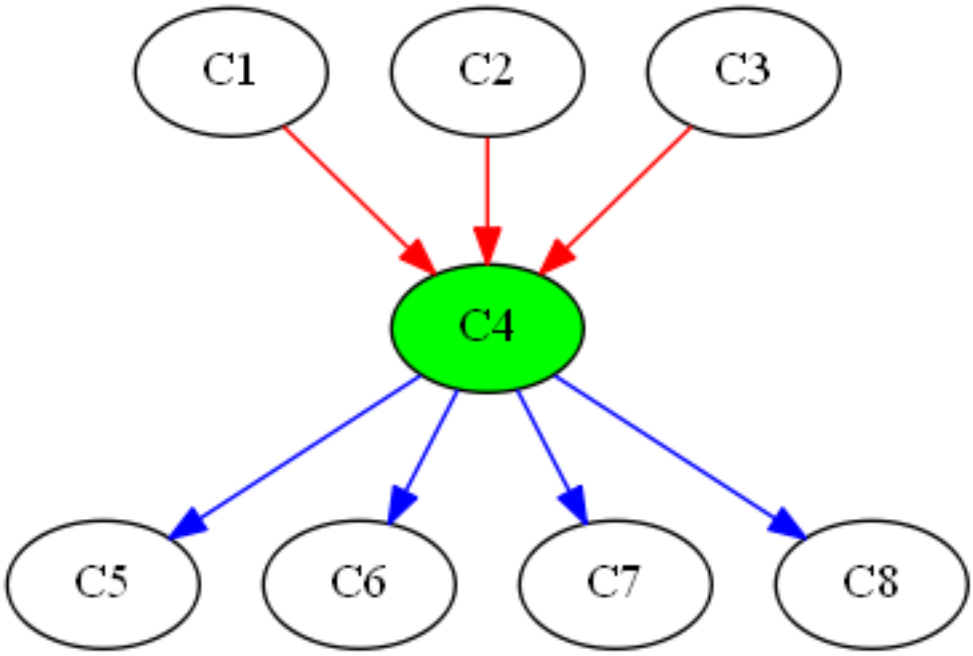
3.2.3 Počet volaných operací (CALLS)

Tato metrika počítá počet všech jednotlivých volaných metod. Každé volání se počítá jen jedenkrát za metodu, ve které je voláno. Pokud je například metoda `A` volána z metody `A1` a `A2`, je počítáno 2 volání. Pokud je metoda `A` volána dvakrát z metody `A1`, počítá jen jedno volání.

3.2.4 Počet volaných tříd (FAN-OUT)

Metrika FAN-OUT je definována jako počet modulů (tříd), které daný modul (třída) využívá. Je tedy definován jako počet referencí jedné třídy na ostatní třídy. Pokud třída využívá více metod z jedné třídy (nebo volá jednu metodu vícekrát), do FAN-OUT se započítává volaná třída (reference na objekt třídy) pouze jednou bez ohledu na četnost využití. V měřítku pro celý softwarový systém lze počítat součet FAN-OUT metrik jednotlivých tříd.

Na obrázku 3.2 je znázorněn příklad. Vyšetřovanou třídou je třída C_4. Hodnoty FAN-OUT této třídy je 4 a na obrázku je znázorněn modře.



Obrázek 3.2: FAN-IN/FAN-OUT příklad

3.2.5 Počet volajících tříd (FAN-IN)

Metrika FAN-IN je definována jako počet modulů (tříd), které daný modul (třídu) využívají. Je tedy definovaný jako počet tříd, které využívají referenci na vyšetřovanou třídu. Pokud některá třída volá více metod vyšetřované třídy nebo volá jednu metodu vyšetřované třídy vícekrát, do FAN-IN je započítávána pouze jednou. V měřítku pro celý softwarový systém lze počítat součet FAN-IN metrik jednotlivých tříd.

Na obrázku 3.2 je znázorněn příklad. Vyšetřovanou třídou je třída C_4. Hodnoty FAN-IN této třídy je 3 a na obrázku je znázorněn červeně.

S výpočtem této metriky může být za určitých okolností problém. Zatímco v případě FAN-OUT, kde vyšetřujeme odkazy na objekty odkazované ze třídy, kde odkazy jsou ve vyšetřované třídě přítomné, v případě metriky FAN-IN tuto informaci k dispozici mít nemusíme, protože v případě tvorby knihoven nebo komponent nemusí být známo, kdo bude v budoucnu vyšetřovanou třídu skutečně využívat.

3.2.6 Depth of Inheritance Tree (DIT)

V [ChK94] je definována metrika Depth of Inheritance Tree (Hloubka stromu dědičnosti). Tato metrika je definována pro jednotlivé třídy. DIT udává délku cesty od uzlu (dané třídy) ke kořeni stromu. V případě, že programovací jazyk, pro který provádíme výpočet metriky, umožňuje vícenásobnou dědičnost (např. C++), bere se v úvahu vždy maximální délka. Tato metrika se snaží podchytit fakt, kolik potencionálních tříd předků může ovlivnit danou třídu.

3.2.7 Number of Children (NOC)

Metrika Number of Children (počet přímo oddděděných tříd) je metrika uvedená v [ChK94]. Jedná se o metriku definovanou pro jednu třídu. Tato metrika udává počet tříd, které jsou přímo oddděděné od dané třídy.

Zde bych chtěl poznamenat, že zkratka „NOC", která je používána přímo v práci [ChK94], je taktéž v řadě prací používána pro metriku počtu tříd (Number of Classes). Proto je velice důležité, aby v literatuře, v nástrojích pro výpočet metrik či v kódu potencionální aplikace bylo na první pohled rozlišitelné, o kterou z těchto dvou metrik se jedná.

3.2.8 Průměrný počet oddděděných tříd (ANDC)

Tato metrika počítá průměrný počet přímých potomků třídy. Počítá se průměr ze všech tříd v projektu. Do výpočtu se nezahrnují knihovní třídy (třídy mimo projekt) a rozhraní. ANDC se počítá podle následujícího vzorce:

$$ANDC = \frac{\sum_{i=0}^{n}(NDCi*i)}{n}$$

kde

$NDCi$ je počet tříd s počtem přímých potomků i,

i je počet přímých potomků,

n je počet všech tříd.

3.2.9 Průměrná výška hierarchie (AHH)

Metrika se počítá jako průměrná výška stromu dědičnosti (HIT) ze všech kořenových tříd dědičnosti. AHH je pak průměr z maximálních délek cesty z kořenové třídy k jejím potomkům (i nepřímým). Třída se považuje za kořenovou, pokud nedědí od žádné třídy v rámci projektu. Rozhraní se podobně jako u ANDC vynechává. Nezávislé třídy (ty, od kterých není žádná třída odděděna) mají HIT 0.

$$AHH = \frac{\sum_{i=1}^{n}(HIT_i)}{n}$$

kde

n je počet kořenových tříd.

3.2.10 Weighted Methods Per Class (WMC)

Weighted Methods Per Class (vážený počet metod) je metrika popsaná v [ChK94]. Je definována podle následujícího vzorce:

$$WMC = \sum_{i=1}^{n} c_i$$

kde

c_i je složitost jednotlivých metod.

Výpočet WMC je možné provádět pro jednu konkrétní třídu nebo pro celý systém. Jako způsob určení složitosti metody může být použita např. McCabova cyklomatická složitost (popsaná v části 3.1.2). V [ChK94] nebyla uvedena žádná konkrétní metoda výpočtu složitosti. V případě, že budeme považovat složitost metod jako jednotnou (za c_i použijeme hodnotu 1), WMC bude roven počtu metod (NOM) a bude roven n.

3.2.11 Coupling between object classes (CBO)

V [ChK94] je definována metrika Coupling between object classes (provázanost mezi třídami objektů). Tato metrika je počítána pro jednu třídu. Udává počet tříd, se kterými je daná třída provázána. To jsou třídy, u kterých třída, pro kterou CBO vypočítáváme, volá alespoň jednu metodu nebo používá nějakou instanční proměnnou. V případě, že třída nevolá žádnou metodu vyjma svých vlastních a nevyužívá ani cizí instanční proměnné, její CBO je rovno 0.

3.2.12 Response For a Class (RFC)

Response For a Class (odezva třídy) je metrika uvedená v [ChK94]. Metrika je definována podle následujícího vzorce:

$$RFC = |RS|$$

kde

RS je množina odezev třídy.

RS může být vyjádřen následovně:

$$RS = \{M\} \cup_{all\ i} \{R_i\}$$

kde

$\{M\}$ je množina všech metod třídy,

$\{R_i\}$ je množina všech metod volaných metodou i.

Množina odezev třídy je množina všech metod, které mohou být potencionálně vyvolány tím, že je zaslána zpráva objektu třídy (volána metoda třídy). Tato metrika tedy měří i potencionální komunikaci mezi danou třídou a ostatními třídami.

3.2.13 Lack of Cohesion in Methods (LCOM)

Lack of Cohesion in Methods (nedostatek soudržnosti v metodách) je poslední metrika definovaná v [ChK94]. LCOM je metrika počítaná pro jednu třídu. Pro každou metodu třídy určíme množinu využívaných instančních proměnných $\{I_j\}$. Poté pro každou dvojici metod zjistíme, zda pro ně existuje průnik využívaných instančních proměnných ($I_i \cap I_j \neq \emptyset$) nebo neexistuje ($I_i \cap I_j = \emptyset$). Z toho vytvoříme množiny $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ a $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. V případě, že veškeré množiny $\{I_i\}$ jsou prázdné ($= \emptyset$), je $P = \emptyset$. Pro výpočet LCOM pak platí:

$$LCOM = |P| - |Q|, |P| > |Q|$$

$$LCOM = 0, |P| \leq |Q|$$

Tato metrika vychází ze stupně podobnosti metod. Podobností metod je v tomto případě myšlena společná množina (průnik) používaných instančních proměnných. LCOM je pak rozdíl velikosti množiny párů s nulovou podobností (P) mínus velikost množiny párů s nenulovou podobností. Čím větší je počet podobných metod (co do počtu využívaných instančních proměnných), tím je třída považována za soudržnější.

3.3 Komponentové metriky

S rozvojem komponentového přístupu v oblasti softwarového inženýrství byl kladen důraz i na rozvoj odpovídajících metrik. Řada odborných článků se úspěšně či méně úspěšně pokoušela naplnit potřeby v této oblasti. Zde popisuji metriky z [CKK01] a [GuS09].

3.3.1 Component Plain Complexity (CPC)

Component Plain Complexity (CPC) je metrika popsaná v [CKK01], která je určena k měření složitosti komponenty. Je vypočítávána na základě počtů tříd, abstraktních tříd a rozhraní a složitosti jednotlivých tříd a metod.

Výpočet je prováděn na základě následujícího vzorce:

$$CPC = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{n} MC_j$$

kde

$CmpC$ je vypočítán na základě počtu tříd, abstraktních tříd, rozhraní a metod,

$\sum_{i=1}^{m} CC_i$ je složitost každé třídy,

$\sum_{j=1}^{n} MC_j$ je složitost každé metody.

Koeficient $CmpC$ je vypočten na základě počtu tříd, abstraktních tříd, rozhraní a metod, přičemž jednotlivé třídy a metody mohou mít svou váhu:

$$CmpC = \sum_{i=1}^{m}(Count(C_i) * W(C_i)) + \sum_{j=1}^{n} Count(I_j) + \sum_{k=1}^{o}(Count(M_k) * W(M_k))$$

kde

$Count(C)$ je počet tříd v komponentě,

$W(C)$ je váha jednotlivých tříd,

$Count(I)$ je počet rozhraní definovaných v rámci komponenty,

$Count(M)$ je počet všech metod definovaných v rámci tříd v komponentě,

$W(M)$ je váha jednotlivých metod.

Složitost jednotlivých tříd (CC) je vypočítána na základě počtu atributů příslušné třídy. V případě komplexních atributů může mít takový atribut svou váhu:

$$CC = \sum_{i=1}^{m}(Count(SA_i)) + \sum_{j=1}^{n}(Count(CA_j) * W(CA_j))$$

kde

$Count(SA)$ je počet jednoduchých atributů,

$Count(CA)$ je počet komplexních atributů,

$W(CA)$ je váha jednotlivých komplexních atributů.

Složitost jednotlivých metod (MC) v rámci komponenty je počítána na základě počtu parametrů, kde komplexní parametry mohou mít svou váhu:

$$MC = \sum_{i=1}^{m}(Count(SP_i)) + \sum_{j=1}^{n}(Count(CP_j) * W(CP_j))$$

kde

$Count(SP)$ je počet jednoduchých parametrů,

$Count(CP)$ je počet komplexních parametrů,

$W(CP)$ je váha jednotlivých komplexních parametrů.

3.3.2 Component Static Complexity (CSC)

Vzhledem k tomu, že metrika CPC odráží pouze počet tříd, rozhraní, metod a parametrů deklarovaných v komponentě a nezachycuje vnitřní strukturu, je v [CKK01] definována metrika Component Static Complexity (CSC), která se soustředí na měření složitosti vnitřní struktury a to ze statického pohledu. Tato složitost je počítána na základě vztahů mezi jednotlivými třídami obsaženými v komponentě.

Výpočet je prováděn na základě následujícího vzorce:

$$CSC = \sum_{i=1}^{m}(Count(R_i) * W(R_i))$$

kde

$Count(R)$ je počet všech vztahů mezi jednotlivými třídami,

$W(R)$ je váha daného vztahu.

Jsou rozlišovány čtyři druhy relací a jejich priority (a odpovídající váhy) by měly být seřazeny následovně: závislost (dependency) < generalizace < agregace < kompozice. V případě, že se objevuje n-ární relace mezi třídami. Tato relace by měla být převedena na relaci binární.

3.3.3 Component Dynamic Complexity (CDC)

Zatímco metrika CSC se soustředila pouze na vnitřní strukturu komponenty, metrika Component Dynamic Complexity (CDC) definovaná v [CKK01] se snaží o dynamický pohled na komponentu. Proto se tato metrika soustředí na počítání zpráv, které jsou posílány mezi třídami v rámci komponenty.

Výpočet je prováděn na základě následujícího vzorce:

$$CDC = \sum_{i=1}^{m} DC(IM_i)$$

kde

$\sum_{i=1}^{m} DC(IM)$ je složitost všech metod rozhraní.

Dále platí:

$$DC(IM) = \sum_{i=1}^{n}(Count(Msg_i) * Freq(Msg_i) + MC(Msg_i))$$

kde

Msg je zpráva předávána mezi třídami,

$Freq(Msg)$ je frekvence předávaných zpráv,

$MC(Msg)$ je složitost jednotlivých předávaných zpráv, která je počítána na základě výpočtu MC popsaného u metriky CPC.

3.3.4 Component Cyclomatic Complexity (CCC)

Poslední metrikou definovanou v [CKK01] je metrika Component Cyclomatic Complexity (CCC). Na rozdíl od předchozích třech metrik (CPC, CSC, CDC), které je možné počítat již v době objektového návrhu komponenty, je metriku CCC možné počítat až z implementované podoby komponenty. Metrika CCC je rozšířená verze metriky CPC o výpočet cyklomatické složitosti jednotlivých metod.

Výpočet metriky je prováděn na základě následujícího vzorce:

$$CCC = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{n} MC_j + \sum_{k=1}^{o} CCM_k$$

kde

$CmpC$ je vypočítán na základě počtu tříd, abstraktních tříd, rozhraní a metod,

$\sum_{i=1}^{m} CC_i$ je složitost každé třídy,

$\sum_{j=1}^{n} MC_j$ je složitost každé metody.

$\sum_{k=1}^{o} CCM_k$ je cyklomatická složitost každé metody.

Výpočet $CmpC$, $\sum_{i=1}^{m} CC_i$ a $\sum_{j=1}^{n} MC_j$ probíhá stejným způsobem jako u metriky CPC. Výpočet cyklomatické složitosti jednotlivých metod je prováděn na základě následujícího vzorce:

$$\sum_{k=1}^{o} CCM_k = edges - nodes + 2$$

Tento výpočet vychází z metriky CYCLO popsané v části 3.1.2.

3.3.5 Provázanost (WTCoup)

V práci [GuS09] je navržena metoda pro výpočet metriky vážené provázanosti (coupling) softwarového systému (komponenty). Tato metrika se snaží vzít v úvahu kromě přímé provázanosti i provázanost nepřímou (tranzitivní). K tomu, aby autoři dosáhli tohoto cíle, se na objektově-orientovaný systém dívají jako na orientovaný graf.

Systém je tvořen množinou tříd $C \equiv \{C_1, C_2, .., C_m\}$. M_i je množina metod třídy C_i a V_i je množina instančních proměnných třídy C_i. $MV_{i,j}$ je pak množina metod a instančních proměnných ve třídě C_j volaných/používaných třídou C_i za podmínky, že $i \neq j$ ($MV_{i,i}$ je definovaný jako null). Hrana ze třídy C_i do C_j existuje pouze v případě, že $MV_{i,j}$ není null. Graf je orientovaný, protože $MV_{i,j}$ nemusí být rovno $MV_{j,i}$. MV_i je definován jako množina všech metod a instančních proměnných volaných/používaných třídou C_i:

$$MV_i \equiv \cup MV_{i,j}$$

Množství metod a proměnných v množině $MV_{i,j}$ odráží sílu přímé provázanosti třídy C_i s třídou C_j. Přímá provázanost je definován následovně:

$$CoupD(i,j) = \frac{|MV_{i,j}|}{|MV_i|+|M_i|+|V_i|}$$

kde

$|MV_{i,j}|$ je velikost množiny volaných/používaných metod a instančních proměnných ze třídy C_j třídou C_i,

$|MV_i|$ je množství všech volaných/používaných metod a instančních proměnných třídou C_i z ostatních tříd,

$|M_i|$ je počet metod třídy C_i,

$|V_i|$ je počet instančních proměnných třídy C_i.

Jmenovatel, jak je navržen, odráží veškerou funkcionalitu třídy C_i – funkcionalitu vlastní a funkcionalitu poskytovanou v rámci volání/používání metod a instančních proměnných ostatních tříd. $CoupD(i,j)$ tedy odráží podíl využité funkcionality třídy C_j na celkové funkcionalitě C_i. Bude tedy platit $0 \leq CoupD(i,j) < 1$. $CoupD(i,j)$ bude vždy menší než 1, protože i v případě využívání pouze funkcionality jediné jiné třídy bude mít třída alespoň 1 metodu, která tuto třídu využívá a jmenovatel tak bude mít větší hodnotu než bude hodnota čitatele.

V dalším kroku je zapotřebí vypočíst nepřímou provázanost mezi třídami. Předpokládejme, že je vypočtena přímá provázanost $CoupD(i,j)$ a $CoupD(j,k)$, které jsou nenulové a hodnotu provázanosti $CoupD(i,k)$ rovnu 0. Je vidět, že v tomto případě není žádná přímá provázanost mezi třídami C_i a C_k. Přesto mezi těmito třídami je provázanost nepřímá a tu získáme vynásobením $CoupD(i,j) * CoupD(j,k)$. Provázanost mezi třídami existuje, pokud existuje cesta z jedné třídy do druhé po hranách, pro které je $CoupD$ nenulový. Na základě této skutečnosti je definována nepřímá provázanost mezi třídami C_i a C_j:

$$CoupT(i,j,p) = \prod_{e_{s,t} \in p} CoupD(s,t) = \prod_{e_{s,t} \in p} \frac{|MV_{s,t}|}{|MV_s| + |M_s| + |V_s|}$$

kde

$e_{s,t}$ je hrana mezi vrcholy s a t.

Je dobré si uvědomit, že v rámci $CoupT$ je zahrnutý i $CoupD$, který odpovídá cestě délky 1. Tranzitivní provázanost bude mít ve většině případů v důsledku delší cesty nižší hodnotu. Mezi dvěma třídami může být více než jedna cesta, pro kterou je hodnota $CoupT$ nenulová. Autoři metriky zvolili přístup, ve kterém z těchto hodnot vybírají tu největší. Platí tedy:

$$Coup(i,j) = CoupT(i,j,p_{max}(i,j))$$

kde

$p_{max}(i,j) = arg\ max_{p \in \prod} CoupT(i,j,p)$,

$\prod$ je množina všech cest ze třídy C_i do třídy C_j.

Poté, co je vypočtena hodnota provázanosti pro všechny dvojice tříd v systému, je možné vypočíst celkovou provázanost softwarové systému. Celkovou provázanost systému je možné vypočíst dle následujícího vzorce:

$$WTCoup = \frac{\sum_{i,j=1}^{m} Coup(i,j)}{m^2 - m}$$

kde

m je počet všech tříd v systému.

3.3.6 Soudržnost (WTCoh)

Druhou významnou metrikou popsanou v práci [GuS09] je metrika pro výpočet vážené soudržnosti (cohesion) tříd softwarového systému (komponenty). Na soudržnost třídy je pohlíženo z hlediska podobnosti metod ve smyslu podobnosti množin používaných instančních proměnných.

Třída je tvořena množinou metod $M \equiv \{M_1, M_2, .., M_m\}$. Množina $V_i \equiv \{V_{i,1}, V_{i,2}, .., V_{i,n}\}$ je množina instančních proměnných používaných metodou M_i. Mezi metodami M_i a M_j existuje hrana, pokud je průnik $Vi \cap V_j \neq \emptyset$. Hrany grafu tak reflektují přímou podobnost metod, které používají alespoň jednu instanční proměnnou společně. Na rozdíl od grafu provázanosti, graf podobnosti je neorientovaný.

V dalším kroku je každé hraně přiděleno číslo odrážející sílu podobnosti těchto dvou metod (mezi kterými je příslušná hrana). Toto číslo reflektuje přímou podobnost a je vypočítáno na základě následujícího vzorce:

$$SimD(i,j) = \frac{|V_i \cap V_j|}{|V_i \cup V_j|}$$

kde

$i \neq j$ $(SimD(i,i) = 0)$.

Jmenovatelem je celkové množství všech instančních proměnných použitých těmito dvěma metodami a odráží celkovou funkcionalitu metod M_i a M_j ve smyslu přístupu k instančním proměnným. Bude tedy platit $0 \leq SimD(i,j) \leq 1$. Hodnotu 1 bude nabývat v případě, že obě metody používají stejnou množinu instančních proměnných.

Do výpočtu podobnosti je v dalším kroku zahrnuta nepřímá podobnost. Síla podobnosti dvou metod je vypočtena součinem (nenulových) hodnot $SimD$ hran cesty mezi těmito metodami. Tranzitivní podobnost je tedy vypočtena následovně:

$$SimT(i,j,p) = \prod_{e_{s,t} \in p} SimD(s,t) = \prod_{e_{s,t} \in p} \frac{|V_s \cap V_t|}{|V_s \cup V_t|}$$

kde

$e_{s,t}$ je hrana mezi vrcholy s a t.

Podobně jako u výpočtu provázanosti, i zde je použita hodnota s největším $SimT$ pro určení podobnosti dvou metod $Sim(i,j)$:

$$Sim(i,j) = Sim(i,j,p_{max}(i,j))$$

kde

$p_{max}(i,j) = arg\ max_{p \in \prod} SimT(i,j,p)$,

$\prod$ je množina všech cest z metody M_i do metody M_j.

Na základě takto vypočtených hodnot $Sim(i,j)$ je možné vypočíst soudružnost dané třídy $ClassCohT$:

$$ClassCohT = \frac{\sum_{i,j=1}^{m} Sim(i,j)}{m^2 - m}$$

kde

m je počet metod v dané třídě.

Nakonec, váženou tranzitivní soudržnost softwarového systému (komponenty) získáme sečtením hodnot soudržnosti jednotlivých tříd $ClassCohT$ a vydělením počtem tříd n:

$$WTCoh = \frac{\sum_{i=1}^{n} ClassCohT_i}{n}$$

4 Úložiště CRCE

S rozvojem Component-based Developmentu a s rostoucím výkonem mobilních zařízení je před vývojáře kladena řada nových výzev. Jednou z těchto výzev je vytvoření moderního úložiště komponent, které má oproti klasickému úložišti řadu nových funkcí. Jedno takové úložiště je vyvíjeno na katedře informatiky na fakultě aplikovaných věd na Západočeské univerzitě v Plzni.

4.1 Component-based Development

Jak je uvedeno v [Bos], Component-based Development (CBD) nebo též Component-based software engineering (CBSE) je paradigma vývoje softwaru, které pochází již z druhé poloviny minulého století. V posledních letech je však skloňováno i v souvislosti s vývojem softwaru pro mobilní zařízení, která donedávna nedisponovala odpovídajícím výkonem.

4.1.1 Nástup CBSE

Myšlenka vývoje pomocí komponent sahá již do roku 1968, kdy na konferenci softwarového inženýrství NATO představil Douglas McIlroy myšlenku, že software by měl být komponentizovaný – sestavovaný z prefabrikovaných komponent. Tato myšlenka je popsána v [MPSC]. První realizací myšlenky bylo zahrnutí pipe a filtrů do operačního systému Unix.

Postupně nabírala myšlenka vývoje na základě komponent různých implementací, jako byly inovace programovacího jazyka C Objective–C, System Object Model (SOM) firmy IBM nebo implementace firmy Microsoft v podobě OLE nebo COM. V současné době existuje celá řada komponentových modelů pro vývoj softwaru.

S rozšířením programovacího jazyku Java došlo k rozvoji CBD i na této platformě. Asi nejznámější komponentová technologie na této platformě se nazývá OSGi a je používána v této práci.

4.1.2 Component-based Software Engineering

Component-based software engineering je proces, který zahrnuje návrh a konstrukci systému skládajícího se z komponent. U těchto komponent se předpokládá jejich znovupoužitelnost. Tento přístup posunuje myšlenku vývoje softwaru od programování směrem ke skládání systému. Component-based software development je založený na myšlence vývoje softwarového systému prostřednictvím výběru vhodných předpřipravených komponent a jejich zapojení do definované architektury.

Jedním z cílů tohoto přístupu je snížení nákladů vývoje rozsáhlejších systémů. Systémy

jsou skládány z řady předpřipravených komponent a díky tomu je možné značně snížit dobu potřebnou k vývoji takového softwaru.

Dalším cílem je zvýšení kvality softwaru. Předpokladem je, že kvalitu softwaru je možné zvýšit zvýšením kvality jednotlivých komponent, kde samostatné komponenty jsou snáze testovatelné než systém jako celek. Tento předpoklad však může být naplněn jen částečně, protože kvalita celého systému nemusí být přímo úměrná kvalitě jednotlivých komponent. Analýzou identifikace kritických komponent z hlediska testování se zabývá např. [CCI].

Cílem by měla být i detekce chyb na úrovni systému. Předpokladem je, že testování jednotlivých komponent může zjednodušit hledání chyb. Některé chyby systému však mohou nastat v důsledku vzájemné komunikace komponent. Přestože jsou jednotlivé komponenty kvalitně otestované, identifikace a oprava chyb na úrovni interakcí může být značně náročná.

4.1.3 Softwarové komponenty

Component-based software development je založený na skládání systému z předpřipravených komponent. Je třeba objasnit, co to softwarová komponenta je. Komponentou se zpravidla myslí nezávislá, malá, část systému. Možný je ale i pohled na systém jako celek jako na komponentu. Komponenty nejsou jen entity, které se objevují v době vývoje. Je zapotřebí se na komponentu dívat i z hlediska běhu aplikace. Komponenty existují za běhu aplikace.

Komponenty poskytují služby. Komponenta je spustitelná jednotka, která může být složena z jednoho nebo více spustitelných objektů. Komponenty mají publikované rozhraní a veškerá interakce s komponentou se provádí prostřednictvím tohoto rozhraní. Komponenty mohou nabízet různé množství funkcionality a mohou pro svou práci vyžadovat funkcionalitu (rozhraní) jiných komponent.

4.2 CRCE

Běžně se v component-based developmentu využívá nějaké formy úložiště předpřipravených komponent pro jejich pozdější využití při sestavování aplikací. Takové úložiště nabízí možnosti ukládání, hledání a získávání komponent. Tento oddíl popisuje CRCE, experimentální úložiště vyvíjené v rámci výzkumu analýzy a kompatibility komponent na katedře informatiky na Západočeské univerzitě v Plzni.

4.2.1 Motivace

V tradičním pojetí úložiště probíhá následné sestavení aplikace (včetně ověření vhodnosti a kompatibility) na straně klientu. Tento proces je vhodný pro klasický component-based development, kde je nejprve sestaven systém jako celek, otestován a následně je jako celek i distribuován. Kompatibilita komponent je vyřešena v době vývoje (sestavení

výsledné aplikace) a v případě distribuce je již systém chápán jako celek. Tento přístup využívají různé frameworky jako např. OSGi.

S růstem výkonu menších zařízení, jako jsou mobilní telefony nebo tablety, nastupuje možnost využití komponentových frameworků i na těchto zařízeních. Díky tomuto trendu bude moci vývoj softwaru pro tyto zařízení čerpat z výhod component-based developmentu. To sebou přináší řadu výzev, které je zapotřebí řešit. Jak se aplikace vyvíjejí v čase, některé komponenty v systému se mění. S tím je spojený update systému. U těchto malých zařízení narážíme na některé problémy, jako je limitované síťové připojení, které značně komplikují update systému jako celku. Navíc v případě potřeby ověřování kompatibility jednotlivých komponent může být výkon systému značně nedostačující.

Cílem je, vzhledem k těmto omezením, minimalizace komunikace tím, že dojde k updatu jen těch komponent, u kterých došlo ke změně verze. Proto je zapotřebí ověření kompatibility přesunout pryč ze zařízení směrem do úložiště, ze kterého jsou komponenty získávány.

4.2.2 Základní koncept CRCE

Jak bylo naznačeno v předchozí části, je zapotřebí přesunout ověřovací mechanismus kompatibility komponent z klientských zařízení, které se potýkají s omezenými zdroji (síťové připojení, výkon, výdrž baterie) směrem do úložiště komponent, které je provozováno na stroji s dostatečnou výpočetní kapacitou. Toto úložiště ověřuje vzájemnou kompatibilitu komponent a tuto informaci poskytuje klientu. Klient může následně rozhodnout, které komponenty bude updatovat, aby nedošlo k porušení konzistence updatovaného systému.

Přístup popsaný v [BaB11] předpokládá postup, kde ve chvíli, kdy je do úložiště vložena nová komponenta (např. nová verze komponenty), je provedeno ověření kompatibility s ostatními komponentami, které jsou již v úložišti přítomné. Výsledek je uložen do metadat, které úložiště vytváří pro každou komponentu. Důležitý je fakt, že v tomto případě je za ověření vzájemné kompatibility odpovědné úložiště a nikoliv vývojář nebo tester.

Ve chvíli potřeby updatu komponent na cílové zařízení jsou k ověření nahraditelnosti komponent využívána metadata a nikoliv samotné komponenty. To značně snižuje dobu ověření kompatibility cílového systému v době updatu (porovnání viz [BaB11]).

Úložiště, které má tuto funkčnost, musí splnit několik bodů funkcionality. Jednak musí být schopné spravovat různé druhy objektů. Vedle samotných komponent, které mají zpravidla podobu distribučních balíčků, jsou to především konfigurační data jednotlivých komponent a dále artefakty, které vznikly v procesu verifikace (metadata).

Další funkčností úložiště musí být verifikace kontraktu, kde vedle přímočarého ověřování kompatibility datových typů a metod je zapotřebí stavového či algoritmického ověření sémantických a interakčních specifikací.

Na rozdíl od tradičních úložišť se musí lišit podoba samotných metadat. Jedná se o metadata, která udržují sémantické informace, která jsou výsledkem verifikace kompatibility. Tato metadata musí jednak odrážet různé verze téže komponenty a dále detailní informace

pro rozhodnutí kompatibility nebo nekompatibility. Z metadat musí být schopný klient určit konzistenci aplikace se složitostí $O(n)$.

Poslední funkcionalitou, které by moderní úložiště mělo splňovat, je možnost různých interakčních módů. V jednom módu by měl mít klient možnost si vyžádat metadata o uložených komponentách, na základě těchto metadat sám rozhodnout o vhodnosti updatu komponent a ten následně provést. V jiném módu by klient specifikoval úložišti současnou konfiguraci systému, který chce updatovat a úložiště by samo provedlo vyhodnocení vhodnosti updatu jednotlivých komponent.

4.2.3 Projekt CRCE

Na základě výše zmíněných požadavků je na Katedře informatiky vyvíjeno úložiště Component Repository supporting Compatibility Evaluation [CRCE].

CRCE je vyvíjeno pomocí technologie OSGi, která umožňuje modularizaci. Je vyvíjeno jako pluginový framework. Hlavními stavebními kameny jsou úložiště, přístup k metadatům (DAO) a úložiště výsledků. Všechny tyto části jsou přístupné jednak prostřednictvím definovaného API a dále prostřednictvím webového rozhraní. Využití OSGi v kombinaci s definovaným API umožňuje využití pluginů, které umožňují rozšiřování CRCE.

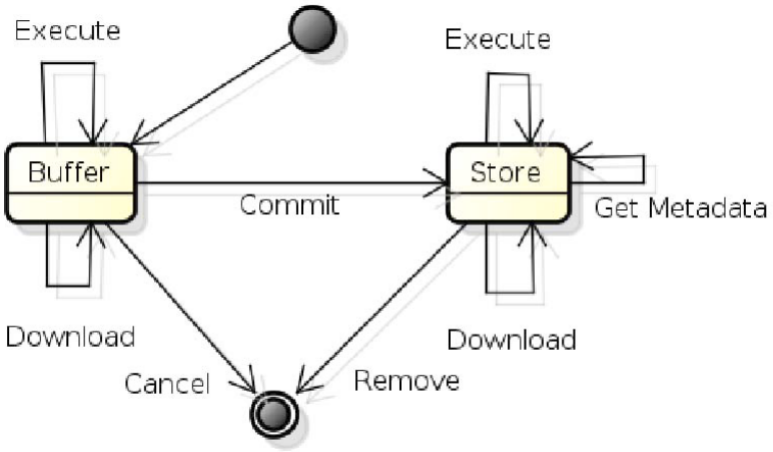
Úložiště představuje jádro pro ukládání komponent. V době psaní této práce mělo podobu souborového systému popsaného prostřednictvím XML souboru. Tento systém se inspiroval OSGi Bundle Repository (OBR). Ze strany klientů představuje CRCE standardní OBR rozhraní rozšířené o sadu potřebných funkcí k splnění požadované funkcionality.

4.2.4 Verifikace kompatibility

Problematikou ověřovaní kompatibility je heterogenost různých komponentových modelů, typů kontraktů a metod. Toto se snaží podchytit pluginová architektura CRCE. Každý model a podoba kontraktu je zpracováván pomocí sady pluginů, které jsou implementovány vždy pro daný specifický účel. Ty mají za úkol analyzovat příslušnou část komponenty a odpovídajícím způsobem aktualizovat metadata. Indexace a ukládání artefaktů je do úložiště integrováno prostřednictvím API modulů metadat a results.

K naplnění této funkcionality využívá CRCE životního cyklu, který je znázorněn na obrázku 4.1. Při nahrávání komponenty do úložiště je komponenta nejprve umístěna do bufferu, kde je ověřena její vnitřní konzistence a je provedena indexace a základní verifikace kompatibility. Tyto akce, u kterých se očekává nízká časová náročnost, jsou prováděny prostřednictvím řetězce pluginů.

V případě, že komponenta projde fází ověřování v bufferu bez problémů, je možné ji potvrdit k uložení do trvalého úložiště, odkud může být vyžádána klienty společně s metadaty. V době, kdy je komponenta uložena do trvalého úložiště, může nastat další sada ověření pomocí řetězce pluginů, přičemž v tomto případě již není kladeno omezení na rychlost těchto ověření. Výsledky těchto ověření jsou opět přidány do metadat.



Obrázek 4.1: Životní cyklus komponenty v rámci CRCE (převzato z [BaB11])

Indexační ověřovací pluginy jsou implementovány jako OSGi bundle a budou popsány v části 6.1.

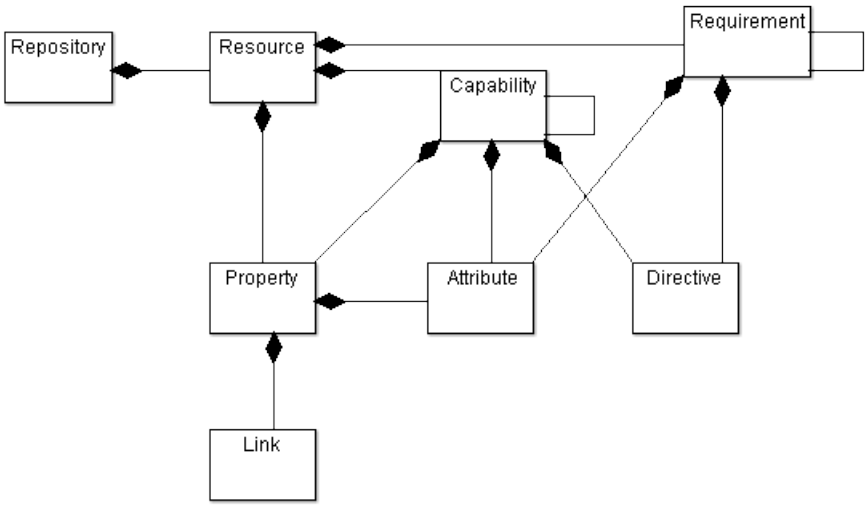
4.2.5 Struktura metadat

Výsledkem indexace a verifikace kompatibility komponent jsou metadata, které se vztahují buď ke komponentě jako celku nebo k její části. Tato metadata jsou základním kamenem k rychlému vyhodnocení konzistence prováděné na straně klientů.

Struktura metadat vychází z konceptu OBR. Hlavními prvky jsou *requirements* (požadavky) a *capabilities* (schopnosti). Metadata mají stromovou strukturu. Jejich kompozice je ukázána na obr 4.2 a popíši ji dále. Je třeba zdůraznit, že v době vzniku této práce (akademický rok 2013/2014), byl projekt CRCE portován z verze 1.0 do vývojové verze 2.0 a s tím byl spojený redesign podoby metadat. Proto je možné, že zde popisovaná podoba metadat nebude plně odpovídat výsledné podobě metadat v době, kdy bude vývoj na verzi 2.0 ukončen.

Kořenem struktury metadat je *repository* (úložiště jako celek). To sdružuje jednotlivé uzly *resource*, které představují metadata vždy k jedné komponentě. V tomto případě platí přímá úměra jedna komponenta jeden uzel *resource*. Každý *resource* má zpravidla několik *cababilities* a několik *requirements*. Může mít i sadu *properties*. Jednotlivé *capability* může sdružovat další *capabilities*, mít sadu *atributů*, *properties* a *directives* (směrnice např. omezující filtry). Podobně může *requirement* stromově sdružovat další *requirementy*, mít sadu *atributů* a *directives*. Jednotlivé *properties* mohou mít *atributy* a *linky*.

Metadata modul a jím definované API zprostředkovává zápis a čtení metadat k jednotlivým komponentám. Tyto volání jsou delegována na Metadata DAO, které představuje persistentní vrstvu. Díky tomuto API jsou provázána metadata s fyzicky uloženými komponentami.



Obrázek 4.2: Struktura CRCE metadat

4.3 Nástroje pro analýzu bytecode

Již řadu let nabývá programovací jazyk Java [Java] na popularitě. Za jeho nástupem, mimo jiné, stojí rychlejší a snadnější vývoj aplikací oproti nízkoúrovňovým jazykům (objektový přístup, dostupnost velkého množství knihoven) a přenositelnost vytvořených aplikací. Přenositelnosti je docíleno díky Java Virtual Machine [JVM], což je běhové prostředí, které je pro běh aplikací v Javě zapotřebí. JVM je k dispozici nejen na osobních počítačích či serverech pro Windows a Linux, ale i např. na mobilních zařízeních. Oproti klasickým interpretovaným jazykům se program napsaný v Javě překládá do bytecode, který je buď interpretován nebo dále překládán za běhu pomocí JVM. Díky tomu dosahuje Java většího výkonu, než klasické interpretované jazyky.

Java bytecode v sobě obsahuje celou řadu vysokoúrovňových informací, jako jsou struktury tříd, definice proměnných a pod. Díky tomu je možné bytecode (např. v podobě class souborů) analyzovat a případně upravovat. K analýze v podobě bytecode slouží knihovny, jako jsou např. BCEL či ASM. Řadu informací o jednotlivých třídách lze získat i za běhu programu a to pomocí Java Reflection.

4.3.1 Charakteristiky Java bytecode

Výstupem překladu jednotlivých tříd napsaných v jazyce Java jsou class soubory, které obsahují Java bytecode. Každý class soubor obsahuje bytecode jedné třídy. Vzhledem k tomu, že se aplikace často skládají z desítek až tisíců tříd, se jednotlivé class soubory společně s případnými dalšími soubory (např. metadaty) slučují do jar/war balíčků, které

představují komprimovaný archiv. JVM je s nimi schopná pracovat jako s celkem.

Každá třída (v podobě class souboru) obsahuje řadu informací. Vedle informací určených čistě pro vnitřní fungování JVM, jako je magická hodnota či major a minor verze (verze Javy), jsou to informace o třídě. Mezi tyto informace patří počet a seznam konstant (constant_pool – zde nejsou uvedeny konstanty definované v dané třídě, ale různé hodnoty jako např. řetězce, reference atp.), přístupová práva třídy, odkaz na tuto třídu (do constant_pool, kde je i její název), odkaz na třídu rodiče (v constant_pool) a odkazy na rozhraní, které třída implementuje (v constant_pool). Dále class soubor obsahuje počet instančních proměnných (fieldů) a jejich seznam (zde se nejedná o odkaz do constant_pool, ale informace jsou uloženy přímo v poli). Každý field má svá práva přístupu, název (v constant_pool), deskriptor (v constant_pool), počet a seznam atributů (fieldu). Mimo fieldy obsahuje třída i počet a seznam metod. O každé metodě jsou přítomny informace o právech přístupu, název (v constant_pool), deskriptor (v constant_pool), počet a seznam atributů (metody). Následuje počet a seznam atributů třídy jako takové.

Počínaje Java 5.0 byly do jazyka Java zahrnuty generika. Ty umožňují definování datových typů v podobě `List<String>`. Vzhledem k tomu, že se Java snaží o zpětnou kompatibilitu a tyto informace nejsou za běhu zapotřebí (JVM je nevyužívá), jsou tyto informace zahrnuty v podobě signatury a to v rámci atributů daného prvku (třídy, fieldu nebo metody).

Jako atributy je vedena celá řada další programových konstrukcí. Patří sem především seznam instrukcí a výjimek (v rámci code atributu), ale i řada dalších.

4.3.2 BCEL

The Byte Code Engineering Library (Apache Commons BCEL) [BCEL] je knihovna určená k analýze, vytváření a manipulaci s bytecode Java soubory (class).

Jednotlivé třídy jsou reprezentovány objekty skládajícími se z jednotlivých symbolických informací dané třídy, hlavně pak fieldy, metodami a jednotlivými instrukcemi. Tyto objekty mohou být načítány z existujících Java class souborů, měněny a následně znovu ukládány. Knihovna se dá využít i k tvorbě tříd za běhu.

Součástí BCEL je JustIce, což je Java verifier, který lze použít k analýze chyb v programech a poskytuje lepší informace, než samotná JVM.

Ač byl BCEL úspěšně nasazen v řadě projektů, v posledních letech přestal být vyvíjen. Proto některé projekty nahradily BCEL frameworkem ASM, který popíši dále.

4.3.3 ASM

ASM [ASM] je v Javě implementovaný framework pro analýzu či úpravu Java bytecode. Jeho hlavními oblastmi určení jsou analýza existujících programů, generování nového Java bytecode nebo transformace existujícího bytecode.

Framework se skládá ze dvou různých API: Core API a Tree API.

Core API

Prvním přístupem, který je možné k práci s Java bytecode využít, je použití Core API. Jeho hlavní funkcionalita je postavena okolo `Visitor` objektů. Oproti Tree API je Core API o něco rychlejší.

Jak Core API tak Tree API vychází ze základní podoby Java bytecode. Na Java bytecode je nahlíženo jako na strom, kde kořenem stromu je jedna třída a strom se skládá z jednotlivých částí reprezentující části dané třídy (např. fieldy). ASM není určeno přímo pro práci s distribučními balíky. Jednotlivé class-soubory je nutno nejprve načíst.

Core API je vystavěno na technice `Visitor` objektů. Základní rozhraní se jmenuje `ClassVisitor` a na její popis API je možné nahlédnou do [ASM]. V případě použití tohoto API je zapotřebí implementovat toto rozhraní. Jednoduché části třídy jsou zpracovávány pomocí jednotlivých `visitXxx` metod. Komplexnější části jsou pak zpracovávány pomocí dalších `Visitor` objektů, jako jsou `FieldVisitor`, `MethodVisitor` či `AnnotationVisitor`.

Pro manipulaci (volání jednotlivých `visitXxx` metod) je možné použít jednu ze tří základních nabízených komponent. Zaprvé je možné použít třídu `ClassReader`, která na základě načtené existující třídy postupně volá `visitXxx` metody v pořadí odpovídající načtené třídě. Na tuto třídu může být nahlíženo jako na generátor událostí (volání metod `visitXxx`). Druhou možností je použít třídu `ClassWriter`, která slouží ke generování kompilovaných tříd a jejich zápisu v podobě bytecode. Tato třída je konzumentem událostí. Posledním přístupem je delegování volání na jiný objekt rozhraní `ClassVisitor` a v takovém případě se jedná o filtr událostí.

Relativně složitějším prvkem v jednotlivých třídách jsou metody. Každá metoda je vedle svého deskriptoru, který popisuje název a typy návratových hodnot a parametrů, reprezentována posloupností bytecode instrukcí. Jednotlivými instrukcemi se v této části zabývat nebudu a je možné se s nimi seznámit například v [JVM]. Dalšími prvky, které se můžou v metodě objevit, jsou labely, try-catch bloky a od Javy verze 6.0 Frames.

Specifickým `Visitor` rozhraním je rozhraní `SignatureVisitor`. Se zavedením generických tříd v Javě 5.0 byly do bytecode zaneseny informace, které nejsou používány za běhu Java aplikace, ale které jsou dostupné prostřednictvím Reflection API. Tyto informace nebyly přidány z důvodu zpětné kompatibility do deskriptorů tříd, fieldů a metod, ale byly zavedeny jako signatury. Tyto signatury je zapotřebí zpracovávat nebo vytvářet pomocí implementace rozhraní `SignatureVisitor`. Vše probíhá podobně jako u jiných `Visitor` objektů voláním metod `visitXxx`.

Tree API

Jak název napovídá, prostřednictvím Tree API je bytecode jednotlivých tříd reprezentován v podobě stromu. Kořenem takového stromu je třída `ClassNode`. Ta má sadu fieldů (jako například název třídy, předka atp.), které třídu jednoznačně definují. Dále tato třída obsahuje řadu seznamů potomků, které reprezentují jednotlivé části třídy, kterými jsou vnitřní třídy, instanční proměnné, metody či anotace.

`FieldNode` představuje jednotlivé fieldy reprezentované třídy. Podobně jako samotná `ClassNode` má i `FieldNode` sadu fieldů, které ji příslušným způsobem definují. Vedle těchto vlastností může mít `FieldNode` ještě sadu atributů a anotací.

Metody jsou reprezentovány pomocí `MethodNode`. Metody mají základní vlastnosti jako název nebo description. Anotace se mohou u metody objevit buď u jednotlivých parametrů nebo u metody jako celku. Metoda může mít seznam try-catch bloků (`TryCatchBlockNode`). Může mít i lokální proměnné (`LocalVariableNode`). A v neposlední řadě je důležitý seznam instrukcí, jejichž nadtřídou je abstraktní třída `AbstractInsnNode`. Jednotlivé instrukce jsou reprezentovány nějakou konkrétní třídou od `AbstractInsnNode` odděděnou a v závislosti na typu mají různé vlastnosti.

Obdoba rozhraní `SignatureVisitor` v Tree API není a v případě potřeby složitého rozklíčování generických typů je zapotřebí použít tento `Visitor`.

Analýza metod

Součástí Tree API je podpora pro analýzu metod a to především analýzy toku dat (data flow) a analýzy toku řízení (control flow).

Analýza toku dat vychází z výpočtu na základě Frames. Stav může být reprezentován v různých podobách abstrakce. Například u referenční proměnné může být sledován fakt, zda obsahuje odkaz na objekt nějaké konkrétní třídy, nebo můžeme být sledována jen skutečnost, zda je nebo není reference `null`.

Analýza toku řízení se provádí na základě konstrukce control flow diagramu, kde jednotlivé instrukce představují uzly grafu a hrany mezi nimi reprezentují možnosti přechodu mezi instrukcemi (dosažitelnost).

Pro účely analýz je součástí ASM třída `Analyzer`. Použití pro výpočet McCabovy cyklomatické složitosti bude popsáno v části 6.3.4.

4.3.4 JaCC a OBCC

Pro účely ověřování kompatibility verzí jednotlivých komponent jsou na Katedře informatiky vyvíjeny dva projekty: Java Class Comparator (JaCC) a OSGi Bundle Compatibility Checking (OBCC). Se základy, na kterých oba projekty stojí, je možné se seznámit v [BrJ12].

JaCC

Projekt JaCC (viz [JaCC]) představuje Java knihovnu, která slouží k porovnávání kompatibility jednotlivých tříd. Knihovna umožňuje porovnávání tříd již načtených (pomocí Java reflection API), kompilovaných ale nenačtených (prostřednictvím analýzy bytecode), i tříd programově vytvořených (například v podobě stub objektů).

Protože knihovna využívá různé přístupy k Java třídám, definuje vlastní sadu `JavaTypes`

rozhraní, které reprezentují jednotlivé třídy a jejich části. Základním rozhraním je rozhraní `JType`. To může mít podobu rozhraní `JClass` (pro negenerické typy) nebo některé z `JParameterizedType`, `JWildCardType`, `JGenericArrayType` a `JTypeVariable` (pro generické typy).

Jednotlivé `JClass` objekty agregují objekty rozhraní `JMember`. Těmi můžou být objekty rozhraní `JField`, `JMethod` a `JContructor`, které intuitivně reprezentují fieldy, metody a konstruktor. Každý element může mít svůj objekt rozhraní `JModifier` (modifikátor přístupu) a u elementů, které mohou být v bytecode anotovány, je evidován jejich případný `JAnnotation` objekt.

Dle způsobu načítání jednotlivých tříd se používají odpovídající objekty `JTypeLoader` (`JClassLoader`). V případě použití Java Reflection je přístup přímočarý. Pro obdržení nenačtených přeložených tříd se používá framework ASM, který byl popsán dříve. Při použití programově vytvořených objektů jde o to vytvořit stub objekty, buď pro testovací účely, nebo například pro účely ověření kompatibility s knihovními třídami, které jsou nainstalované v systému a předpokládá se u nich, že se při updatu jednotlivých komponent nebudou měnit.

Samotné porovnávání pak probíhá na základě porovnání vždy dvou typů a to porovnáváním jednotlivých struktur. Mohou být zjištěny celkem čtyři různé výsledky: rovnocenné typy, specializace (např. přidání metody), generalizace (opak specializace – např. odstranění metody) nebo mutace (mezi třídami není žádná podobnost). Obecně lze říci, že v případě, že jsou typy rovnocenné, jsou typy nahraditelné. Dále je pro nahraditelnost přípustná specializace (obdoba dědičnosti, kde potomek může nahradit svého předka).

OBCC

OBCC (viz [OBCC]) je projekt, který slouží k ověřování kompatibility OSGi komponent. Vychází ze základního konceptu OSGi, kde každý bundle (distribuční balík) má určitou sadu poskytovaných (provide) a požadovaných (require) package a/nebo services.

Aby bylo možné porovnávat navzájem dva bundle, vytváří OBCC jejich reprezentaci (bundletypes). To je prováděno ve třech krocích: načtení informací z metadat, zpracování implementace na exportní (poskytované) straně a zpracování implementace na importní (požadované) straně.

V prvním kroku je zpracován bundle manifest, kde jsou uvedeny jednotlivé packages a další informace. Tyto informace jsou extrahovány z manifestu (textová forma) a mimo jiné jsou takto zjištěny názvy packages/services a jejich verze.

Ve druhém kroku je zapotřebí zpracovat Java bytecode poskytovaných packages/services. K tomu je využíván JaCC, který byl popsán výše. Vzhledem k tomu, že se jedná o v komponentě přítomný kód (komponenta jej poskytuje), načtení se děje postupným parsováním přítomných tříd. Prohledávání začíná od exportovaných tříd v rámci package/service a postupným rekurzivním sestupem jsou přidávány využívané přítomné třídy. V případě, že se objeví reference na typ, který není v komponentě přítomen, je buď vytvořen stub (v případě že typ je uvedený jako importovaný) nebo detekována chyba (nejedná-li se

o importovaný typ).

Situace na straně importu je značně komplikovanější. Zatímco na straně exportu bylo možné získat reprezentaci z bytecode, na straně importu tato informace chybí. Tento fakt ostatně plyne ze základní myšlenky komponentového vývoje. Jednotlivé importované typy je zapotřebí vytvořit (složit) na základě v komponentě používané funkcionality (komponentou využívané fieldy a metody). Např. pokud je někde v kódu volána metoda A třídy T a jinde volána metoda B třídy T, reprezentace třídy T bude obsahovat metody A a B.

K porovnávání kompatibility se pak využívá porovnávání typů popsaného výše.

5 Metriky pro softwarové komponenty

Před samotnou implementací bylo zapotřebí z množiny prozkoumaných metrik vybrat určitou podmnožinu metrik, které budou implementovány v rámci této práce a které by nejlépe komponentu, pro kterou jsou počítány, charakterizovaly. Vzhledem k rozsahu diplomové práce a s ohledem na zmiňovaný cíl, jsme při konzultaci s vedoucím diplomové práce vybrali několik metrik, kterými se budu zabývat v této kapitole.

5.1 Výběr a definice metrik

Před výběrem konkrétních metrik jsem provedl studium odborné literatury a vědeckých článků. Na základě tohoto studia vznikla kapitola 3, která odráží určitý potencionální předvýběr metrik, které by buď mohly být vhodné pro implementaci, nebo by na jejich základě mohly být skutečně implementované metriky postavené. Na základě tohoto předvýběru a s ohledem na specifika měření bytecode (popsané dále v 5.2) jsme vybrali s vedoucím diplomové práce dvě metriky rozhraní a čtyři metriky vnitřní, které jsem následně implementoval do úložiště CRCE v podobě pluginu.

Metriky rozhraní se snaží zachytit vlastnosti zveřejněného rozhraní. Měří tedy vlastnosti veřejného API komponenty a to bez ohledu na strukturu komponenty, která je za tímto rozhraním skryta. Každá komponenta má svou importovanou část (rozhraní či balíčky, které využívá) a svou exportovanou část (rozhraní a balíčky, které poskytuje).

První metrika, kterou jsme zvolili, má měřit složitost importované části a je jím metrika Number of Imports (viz 5.1.1). Druhá metrika má naopak za úkol měřit část exportovanou. Touto metrikou je API complexity (viz 5.1.2).

Metriky vnitřní vycházejí z vnitřní struktury komponenty. Oproti metrikám rozhraní, které měří pouze vnější podobu komponenty, vnitřní metriky mohou odrážet vnitřní strukturu komponenty na různých úrovních. Je možné zkoumat jednotlivé prvky, ze kterých je daná komponenta složena (metody, třídy, balíčky) zvlášť a hodnotu metriky komponenty následně počítat např. jako průměr hodnot jednotlivých částí. Možné je i sledovat vzájemnou propojenost jednotlivých částí.

Vzhledem k těmto možnostem je řada vnitřních metrik rozsáhlejší a pokrývá různé vlastnosti komponenty. První metrikou je metrika Ripple effect (viz 5.1.3), která se snaží podchytit celkové množství metod, které mohou být provedeny na základě volání metod API exportovaného balíčku. Složitost jednotlivých metod měří McCabova cyklomatická složitost (viz 5.1.4). Pro měření soudržnosti tříd jsme zvolily metriku WTCoh (viz 5.1.5). Poslední implementovanou metrikou je metrika pro měření provázanosti mezi třídami v podobě metriky WTCoup (viz 5.1.6).

5.1.1 Number of Imports (NOI)

Metrika Number of Imports jednoduchým způsobem charakterizuje složitost importované části. Na základě manifestu (soubor manifest.mf) je možné získat přehled všech importovaných balíčků a services. Skutečná struktura jednotlivých importovaných balíčků, jejich tříd a metod není přímo z vyšetřované komponenty známa. Na základě analýzy volání by se dala rekonstruovat používaná část importovaných tříd, ale neodpovídalo by to celkové podobě importovaného API (součástí importovaných balíčků a services jsou třídy a metody, které vyšetřovaná komponenta nevyužívá). Měření složitosti takto rekonstruované podoby importovaného API by nebylo odpovídající. Proto jsme zvolili pro charakteristiku importované části komponenty prostý součet všech importovaných balíčků a services. Tento údaj lze určit na základě analýzy manifestu a je relativně jednoduchý. Výpočet vypadá následovně:

$$NOI = |IP| + |IS|$$

kde

$|IP|$ je počet importovaných balíčků,

$|IS|$ je počet importovaných services.

5.1.2 API complexity

Celková podoba exportovaného rozhraní (balíčky, services) je známá. Pro měření složitosti exportovaného rozhraní lze použít řadu metrik. Pro tento účel se hodí např. metrika WMC (viz 3.2.10) nebo metrika CPC (viz 3.3.1). Nakonec byla pro implementaci zvolena metrika CPC.

Při výpočtu metriky CPC se používá řada vah jednotlivých tříd, metod, atributů tříd a parametrů metod. V rámci práce [CKK01] nejsou navrženy žádné konkrétní hodnoty ani postup, jak tyto váhy určit. Metrika byla v této práci navržena tak, aby mohla být použita již ve fázi návrhu softwaru (komponenty), kde se pravděpodobně předpokládá, že návrhář určí tyto váhy na základě zkušenosti či aktuální potřeby.

Jak je metrika navržena, je možné každé třídě, metodě či atributu přidělit individuální váhu. To umožňuje, v době softwarového návrhu, odpovídajícím způsobem zdůraznit důležitost jednotlivých prvků. Nevyhovuje to však účelu jednotného měření složitosti API, pro který je tato metrika využívána v této práci. Proto je pro jednotlivé prvky stejného typu (např. všechny třídy) použita stejná hodnota váhy. Tato hodnota je fixní. Celkem jsou použity čtyři konstanty – váha třídy, váha metody, váha složeného atributu (třídy) a váha složeného parametru (metody).

Metrika je počítána pro každý exportovaný balíček zvlášť. Pro exportované services počítána není (přesahuje rozsah této práce), ale počítala by se obdobným způsobem.

Výpočet metriky je prováděn následovně:

$$CPC = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{n} MC_j$$

kde

$CmpC$ je vypočítán na základě počtu tříd, abstraktních tříd, rozhraní a metod exportovaného balíčku,

$\sum_{i=1}^{m} CC_i$ je složitost každé třídy exportovaného balíčku,

$\sum_{j=1}^{n} MC_j$ je složitost každé metody tříd exportovaného balíčku.

5.1.3 Ripple effect

Metrika Ripple effect byla vytvořena pro účely této práce ve spolupráci s vedoucím práce. Vychází z metriky RFC (viz 3.2.12), ale rozšiřuje ji do podoby, aby daná metrika byla využitelná v komponentovém prostředí. Snahou je, aby metrika odrážela ripple effect, který způsobí volání metod některého exportovaného balíčku. Je tedy měřena pro každý exportovaný balíček zvlášť.

Při volání metod tříd exportovaného balíčku může daná metoda buď provést nějakou činnost sama, nebo může volat jiné metodu. Tato nově volaná metoda provede svou činnost sama nebo volá další metody. Při analýze jednotlivých volání lze takto postupným sestavováním množiny všech potencionálně vyvolaných metod vytvořit sadu všech metod, které volání metod tříd exportovaného balíčku může způsobit. Tyto metody lze rozdělit do dvou skupin: na metody tříd, které jsou v komponentě přítomné, a na metody tříd, které jsou buď knihovní nebo importované (z hlediska metriky je rozdíl mezi knihovní a importovanou třídou nepodstatný). Metod, které nejsou v komponentě přítomné, se do hodnoty metriky započítávají, ale není možné dále zkoumat jejich ripple effect. Metody, které jsou v komponentě přítomné, se do hodnoty metriky započítávají a zároveň lze rekurentně dále zkoumat metody, které volají.

V rámci volání je možné vedle implementované metody narazit také na volání metody bez implementace – abstraktní metody a metody rozhraní. Při exekuci dochází k volání metody nějaké konkrétní třídy, ale přesné určení dané třídy z bytecode jen stěží zjistitelné (zpětnou analýzou datového toku) a v některých případech by nebyla možná zjistit vůbec (při předávání implementace z vnějšku komponenty). Proto se v rámci této metriky nesnažím zjišťovat konkrétní implementaci takové metody a metodu pouze do hodnoty metriky započtu.

Výsledný Ripple effect balíčku se tedy vypočte následovně:

$$RE = |MI| + |MIA| + |ME|$$

kde

MI je množina všech tranzitivně vyvolaných (implementovaných) metod v rámci komponenty,

MIA je množina všech vyvolaných abstraktních metod v rámci komponenty,

ME je množina všech vyvolaných metod mimo komponentu.

5.1.4 McCabova cyklomatická složitost (CYCLO)

McCabova cyclomaticka složitost (viz 3.1.2) je klasická metrika používaná již od roku 1976. Původně byla navržena pro jazyky, které neměly výraznější strukturování. V současné době je zpravidla počítána pro jednotlivé metody.

Při výpočtu hodnoty metriky pro metodu je možné zvolit různé přístupy. Jak bylo popsáno dříve, uzlem grafu pro výpočet složitosti je zpravidla nedělitelný blok příkazů. Metodu, pro kterou je metrika počítána, je možné popsat blokovým grafem, na základě kterého je metrika počítána.

Co má velký vliv na vypočtenou metriku, je přístup ke koncovým uzlům metody (příkazům `return`). Metodu lze ukončit v různých místech metody. Je možné počítat všechny příkazy `return` jako jeden příkaz, nebo počítat každý příkaz `return` zvlášť. Základem rozdílného pohledu může být, zda se díváme na metodu v rámci kontextu s celou aplikací. Ač má metoda více `return` příkazů, po provedení tohoto příkazu je řízení vrácené vždy do stejného bodu. To však neplatí ve chvíli, kdy do výpočtu zahrneme i koncové body metody v podobě příkazu `throw`, který jazyk Java umožňuje. V takovém případě by pro výpočet hodnoty metriky byla zapotřebí analýza okolí volání dané metody, což nemusí být v řadě případů možné. V této práci je zvolen přístup, kde jsou všechny koncové uzly metody počítány jako jeden.

Další rozhodnutí, které je při volbě této metriky zapotřebí udělat, je volba, zda bude výpočet prováděn jen na konkrétních instančních metodách, zda bude prováděn i na statických metodách a zda bude prováděn na konstruktorech. Z principu nic nebrání tomu, aby byl prováděn na všech implementovaných metodách, což je i přístup, který byl zvolen v této práci. Jediné metody, na kterých není možné výpočet provést, jsou metody abstraktní a metody rozhraní, u kterých není hodnota metriky definována.

Pro jednotlivé třídy v komponentě se provádí výpočet průměru za danou třídu. V případě, že jsou ve třídě metody, pro které není hodnota metriky definována, jsou tyto metody z výpočtu vyřazeny. V případě, že by nastala situace, že třída/rozhraní nebude mít žádnou implementovanou metodu, bude hodnota metriky pro takovou třídu/rozhraní nedefinována.

Na úrovni komponenty jsou počítány celkem tři hodnoty metrik. První metrikou je výpočet průměru hodnot metrik jednotlivých tříd:

$$CYCLO_{avg} = \frac{\sum_{i=1}^{n} CYCLO_i}{n}$$

kde

$CYCLO_i$ je průměrná hodnota McCabovy cyklomatické složitosti jednotlivých metod dané třídy,

n je počet tříd, pro které je hodnota McCabovy cyklomatické složitosti definována.

Druhou metrikou je výpočet minima hodnot metrik jednotlivých tříd:

$$CYCLO_{min} = min(CYCLO_i)$$

kde

$CYCLO_i$ je průměrná hodnota McCabovy cyklomatické složitosti jednotlivých metod dané třídy.

Třetí metrikou je výpočet maxima hodnot metrik jednotlivých tříd:

$$CYCLO_{min} = max(CYCLO_i)$$

kde

$CYCLO_i$ je průměrná hodnota McCabovy cyklomatické složitosti jednotlivých metod dané třídy.

5.1.5 Metrika WTCoh

Pro výpočet soudržnosti tříd komponenty jsme vybrali metriku WTCoh (viz 3.3.6). Metrika vychází z výpočtu soudržnosti jednotlivých tříd na základě podobnosti metod (ve smyslu přístupu k instančním proměnným) a to se zahrnutím nepřímé (tranzitivní) podobnosti. Je vypočítána soudržnost každé třídy a následně je vypočten průměr za komponentu jako celek.

Metrika není počítána pro rozhraní a pro třídy bez implementovaných metod (čistě abstraktní třídy). Ze způsobu výpočtu soudržnosti jedné třídy ($ClassCohT$) je patrné, že není definována ani soudržnost třídy s pouze jednou implementovanou metodou.

Otázkou může být, zda do výpočtu zahrnovat statické fieldy a metody. Práce [GuS09] předpokládá, že se nezahrnují statické fieldy, ale metody explicitně nespecifikuje. Vzhledem k možnosti využívání statických fieldů (například statický logger) i na úrovni instančních metod jsem, po konzultaci s vedoucím práce, zahrnul do výpočtu statické metody i fieldy. V případě, že by se po delší zkušenosti používání této varianty metriky ukázalo, že to není příliš vhodné, je možné toto upravit.

Dále bylo zapotřebí vyřešit otázku algoritmu pro ohodnocení cesty pro nepřímou podobnost ($p_{max}(i,j) = arg\ max_{p \in \prod} SimT(i,j,p)$). Vzhledem k vlastnostem rozsahu hodnot přímé podobnosti $SimD$ bylo možné využít upravený Floyd–Warshallův algoritmus [Flo62]. Pro algoritmus je možné použít kritérium maximalizace tranzitivní podobnosti a místo součtu použít součin ohodnocení cest:

$$SimT_k(i,j) = max(SimT(i,j), SimT(i,k) * SimT(k,j))$$

kde

$SimT_k(i,j)$ je nejvyšší ohodnocení tranzitivní podobnosti při zahrnutí vrcholu k (cesta z i do j může a nemusí vést přes k),

$SimT(i,j)$ je tranzitivní podobnost před zahrnutím vrcholu k,

$SimT(i,k) * SimT(k,j)$ je ohodnocení tranzitivní podobnosti v případě, že cesta z i do j vede přes k,

k nabývá postupně hodnoty $1..m$, kde m je počet metod.

Tento algoritmus má složitost $O(n^3)$.

Výpočet $WTCoh$ je poté prováděn z hodnot jednotlivých $ClassCohT$ a to pouze tříd, pro které je hodnota definována:

$$WTCoh = \frac{\sum_{i=1}^{n} ClassCohT_i}{n}$$

5.1.6 Metrika WTCoup

Jako metriku pro výpočet provázanosti tříd komponenty byla zvolena metrika WTCoup (viz 3.3.5). Tato metrika zahrnuje vedle provázanosti přímé i provázanost nepřímou (tranzitivní). Metrika vychází z podílu využívané funkcionality jiné třídy na celkové funkcionalitě vyšetřované třídy.

Do výpočtu metriky nejsou zahrnovány rozhraní a třídy bez metod (anomální situace). Do výpočtu jsou zahrnovány abstraktní třídy. $WTCoup$ není definován pro komponenty, které nemají žádnou nebo mají pouze jednu třídu, kterou je možné do výpočtu zahrnout.

Stejně jako u metriky WTCoh, i u metriky WTCoup bylo zapotřebí rozhodnout, zda do funkcionality třídy započítávat statické fieldy a metody. I zde jsem je do výpočtu zahrnul. Třídy mohou vykazovat provázanost jak na instanční, tak na statické úrovni. Opět v případě, že by se při delším používání ukázalo, že tato volba nebyla vhodná, je možné toto rozhodnutí změnit.

Pro výpočet tranzitivní provázanosti jsem použil upravený Floyd–Warshallův algoritmus podobně jako u metriky WTCoh. Pro hodnoty přímé provázanosti zde platí stejná omezení jako pro hodnoty přímé podobnosti metod u metriky WTCoh.

Vedle provázanosti v rámci komponenty bylo vhodné připočíst funkcionalitu (a s tím spojený podíl na provázanosti) směřující ven z komponenty. Řada tříd využívá funkcionalitu z knihovních tříd nebo z importovaných komponent. Importované třídy funkcionalitu komponenty nevyužívají. Exportovaná funkcionalita není předmětem výpočtu provázanosti. Proto bylo k výpočtu provázanosti na základě práce [GuS09] zapotřebí připočíst i provázanost jednotlivých tříd k třídám mimo komponentu:

$$WTCoup = \frac{\sum_{i,j=1}^{m} Coup(i,j) + \sum_{i=1}^{m} CoupOut_i}{m^2 - m}$$

kde

$CoupOut_i$ je provázanost dané třídy ven z komponenty,

m je počet vyšetřovaných tříd v komponentě.

Jak jsem uvedl již u metriky WTCoh, Floyd–Warshallův algoritmus má složitost $O(n^3)$. Zatímco v případě výpočtu $ClassCohT$ velký problém s časovou náročností nebyl (třída má zpravidla jednotky až desítky metod), u větších komponent, které mohou mít tisíce tříd, může být výpočet značně pomalý.

5.2 Vliv bytecode na hodnoty metrik

Při seznamování se s bytecode Javy jsem narazil na řadu skutečností, které jsou významné pro výpočet metrik z Java bytecode. Významné jsou také v případě, že by hodnoty metrik vypočtené na základě bytecode byly porovnávány s hodnotami metrik vypočtených na základě zdrojového kódu. V této části popíši především ty, které jsem přímo řešil.

5.2.1 Lines of Code

Metrika Lines of Code (viz 3.1.1) se zpravidla počítá na základě zdrojového kódu. Otázka tedy je, zda by se dala počítat i na základě bytecode Javy. Na úrovni bytecode se nenacházejí žádné řádky komentáře a odpadá i problém s zvyklostmi zápisu příkazů jako např. ohraničení bloků. V tomto směru je bytecode jednotný a dobře měřitelný. Pokud by se měřil jen počet instrukcí, nebyl by problém.

Problém nastane ve chvíli, kdy bychom chtěli na úrovni bytecode měřit počet řádků zdrojového kódu, nebo výsledky měření na úrovni bytecode porovnávat s výsledky měření zdrojového kódu. Jak ukáži v další části, při překladu jsou doplňovány určité části kódu (např. konstruktor, pokud není uvedený) nebo jsou některé části vypuštěny (např. mrtvý kód). V důsledku toho není možné určit nějaký magický koeficient, kolik instrukcí bytecode odpovídá řádku zdrojového kódu. Ani pokus o rekonstrukci původního kódu na základě bytecode by nebyl přesný, protože řada různých konstrukcí (nejen z hlediska počtu řádků) je překládána na stejný výstup (podobu bytecode). Z těchto důvodů metrika Lines of Code nebyla zařazena pro implementaci.

5.2.2 Automaticky generované konstrukce

Standardní překladač Javy provádí při překladu řadu důležitých operací. Vedle samotného generování výstupního bytecode je to především doplnění implicitních konstrukcí (jako např. defaultní konstruktor). To může vést k určitým nesrovnalostem, pokud bychom porovnávali hodnoty metrik měřených na úrovni zdrojového kódu a na úrovni bytecode, pokud bychom na tyto vlastnosti překladače zapomněli.

Již zmiňovaný defaultní konstruktor je doplňovaný vždy, pokud jej programátor explicitně konstruktor neuvede. Dochází k tomu i v případě abstraktních tříd. Tělo tohoto konstruktoru volá konstruktor nadtřídy, kterou je často `java.lang.Object`. To má vliv na výpočet řady metrik, které konstruktory zahrnují. Například to může mít vliv na výpočet průměrné McCabovy cyklomatické složitosti (viz 5.1.4), kde složitost takto doplněného konstruktoru je vždy 1, nebo na výpočet Ripple effectu (viz 5.1.3), kde je podstatné volání konstruktoru nadtřídy.

Dalším zajímavým příkladem je překlad spojování řetězců (zřetězení). V kódu se často používá konstrukce v následující podobě:

```
System.out.print("Slozeny" + "text");
```

Ač tato konstrukce vypadá na první pohled jednoduše, při překladu dojde k doplnění funkcionality o použití objektu třídy `StringBuilder`, který se o zřetězení stará. Kód by pak (zjednodušeně) vypadal takto:

```
StringBuilder sb = new StringBuilder("Slozeny");
sb.append("text");
System.out.print(sb.toString());
```

Jak je z příkladu vidět, jednoduché zřetězení vede na využití explicitně neuvedené třídy a jejích metod, což bude mít vliv na výpočet některých metrik jako např. již zmiňovaný Ripple effect (5.1.3). V případě výpočtu metrik na úrovni bytecode toto není problém (kód je zde zahrnut), ale v případě výpočtu (a porovnání) metrik na úrovni zdrojového kódu je zapotřebí toto brát v úvahu.

Jiným příkladem doplnění při překladu může být vytvoření nebo nevytvoření statického konstruktoru. Na jeho vytvoření má vliv explicitní inicializace statických proměnných. Porovnejme následující dva případy:

```
public class A {
  private static A instance;

  ...
}

public class B {
  private static B instance = null;

  ...
}
```

V případě třídy `A` při překladu k doplnění statického konstruktoru nedojde. V případě třídy `B` je statický konstruktor doplněn a stará se o nastavení reference `instance` na `null`. Tím jen nepatrným rozdílem v zápisu zdrojového kódu vznikají při překladu rozdíly, které mohou vést k rozdílným hodnotám metrik (např. u již zmiňovaného Ripple effectu).

5.2.3 Generika v Javě

Počínaje Java 5.0 byly do jazyka Java zahrnuty generika. Ty umožňují definování datových typů na úrovni zdrojového kódu v podobě `List<String>`. Generika slouží k vytváření parametrizovaných datových typů, kterým následně určíme, s jakým konkrétním datovým typem skutečně pracují. Tato informace však slouží pouze k typové kontrole v době překladu a v době běhu se s ní nijak nepracuje. V bytecode je přítomná v podobě signatury dané metody, ale již ne na jiných místech. Např. v místě volání metody (různé instrukce `invokeXXX`) je uvedena jen zkrácená podoba deskriptoru metody bez parametrických informací. Tuto skutečnost je zapotřebí brát v úvahu především v případě vyšetřování řetězce volání metod jako se provádí u metriky Ripple effect. V takovém případě je zapotřebí provádět porovnávání shody metod na základě deskriptoru (a nikoliv signatury).

6 Implementace CRCE Metrics pluginu

Hlavním úkolem této práce v oblasti implementace bylo vytvoření CRCE Metrics pluginu. Bylo zapotřebí vytvořit jádro pluginu, který by se stal součástí komponentového úložiště CRCE, implementovat získávání potřebných dat z komponent a výpočet vybraných metrik. Nedílnou součástí této práce byla i úprava WebUI, aby dokázalo s naměřenými hodnotami pracovat a zobrazovat je.

V této kapitole postupně popíši jednotlivé implementační práce.

6.1 Pluginy CRCE

Úložiště CRCE [CRCE] je implementováno komponentovou technologií OSGi [OSGi]. Vedle základních stavebních kamenů, kterými jsou úložiště komponent, přístup k metadatům (DAO) a úložiště výsledků, zajišťuje funkcionalitu sada pluginů, které jsou vyvolávány v rámci životního cyklu resource v úložišti (viz obr. 4.1). V současné době jsou vyvíjené pluginy určeny především pro OSGi komponenty (včetně CRCE Metrics pluginu), ale principiálně toto úložiště není omezené pouze na komponenty této technologie.

6.1.1 OSGi bundle

Každý OSGi bundle má svůj aktivátor. Aktivátorem je třída (v případě CRCE Metrics pluginu je jím třída `Activator`), která se stará o životní cyklus samotné komponenty (pluginu). Ten je řízen metodami pro spuštění a zastavení komponenty. Důležitou událostí je především spuštění (vytvoření) komponenty. V této fázi se používá OSGi `DependencyManager`, který se stará o správné pořadí vytváření a spouštění jednotlivých services, které pluginy poskytují. V rámci vytváření je specifikována i řada rozhraní, které daná komponenta vyžaduje.

6.1.2 ActionHandler

Pluginy CRCE určené pro získávání metadat z komponent (obecně resource) ukládaných do úložiště implementují rozhraní `ActionHandler`. Toto rozhraní definuje sadu metod, které odpovídají událostem životního cyklu komponenty v CRCE. Daný plugin implementací konkrétní metody může reagovat na odpovídající událost. Jako příklad bych uvedl metody `onUploadToBuffer` nebo `afterPutToStore`. První zmíněná metoda se vyvolá ve chvíli nahrávání resource do bufferu, zatímco druhá metoda se provede ve chvíli, kdy je resource potvrzen k uložení do úložiště (po jejím nahrání).

Toto rozhraní implementuji třídou `MetricsIndexerActionHandler`. Tato třída používá pouze metodu `afterPutToStore`, která je provedena až poté, co je resource uložen do úložiště. Implementace metody má za úkol vyvolat asynchroní task, který provádí samotnou funkci pluginu. Funkcionalita pluginu se provádí na pozadí z důvodu časové náročnosti výpočtu některých metrik.

6.1.3 Výpočet na pozadí

Pro účely situací, kdy by měla funkcionalita některého pluginu zabrat nepřiměřeně velké množství času a způsobovat dlouhou odezvu na požadavky uživatele, jsou v CRCE zavedeny tasky (podrobně viz [Dan14]). K funkcionalitě asynchroního provádění událostí na pozadí slouží abstraktní třída `Task` a service `TaskRunnerService`. V rámci pluginu je zapotřebí implementovat třídu, která dědí od třídy `Task` a implementuje odpovídající metodu `run`. Objekt implementované třídy se pak předá service `TaskRunnerService`, která jej následně spustí společně s dalšími naplánovanými tasky. Samotný kód metody `run` se již provádí na pozadí a uživatel, který nahrává resource do úložiště, nemusí čekat na její skončení.

Mou implementací třídy `Task` je třída `MetricsIndexerTask`. Jejím prvním úkolem je zjištění, zda je resource ukládaný do úložiště OSGi komponenta. Jak již bylo zmíněno, do úložiště se dají ukládat i jiné objekty, než jsou OSGi komponenty. Informaci, zda se jedná o OSGi komponentu, zjišťuje jiný plugin CRCE. V případě, že se jedná o OSGi komponentu, tak třída `MetricsIndexerTask` tuto komponentu (v podobě jar souboru) načte a předá objektu třídy `MetricsIndexer`, který se stará o samotné řízení získávání metrik z komponenty. Po skončení výpočtu metrik metoda `run` ještě ověří validitu získaných metadat (pomocí instance třídy `MetadataValidator`) a vyvolá jejich uložení pomocí odpovídajícího DAO objektu (`ResourceDAO`).

6.2 Společné aspekty získávání dat pro metriky

Jednotlivé metriky komponent jsou počítány na základě rozboru bytecode získaného z jar souboru komponenty pomocí frameworku ASM popsaného v části 4.3.3. Vzhledem k tomu, že řada informací získávaných z komponenty slouží pro výpočet více jak jedné metriky, jsou před samotným výpočtem jednotlivých metrik získána společná data a podkladové metriky, které jsou následně využity jednotlivými třídami určenými pro výpočet metrik komponent.

6.2.1 Rozhraní pro podkladové metriky

Pro společná, z ASM získávaná, data a z nich vypočtené hodnoty podkladových metrik, jsem definoval řadu rozhraní. Rozhraní zohledňují strukturu celé komponenty a každé definuje určitou sadu, pro danou úroveň odpovídajících, metod. Pro fieldy je to rozhraní `FieldMetrics`, pro metody rozhraní `MethodMetrics`, pro třídy (a rozhraní) rozhraní

`ClassMetrics` a pro kolekci všech tříd (a rozhraní) rozhraní `ClassesMetrics`.

6.2.2 Implementace rozhraní pro společná data

Jednotlivá rozhraní popsaná v předchozí části mají své odpovídající implementace, které pomocí ASM frameworku získávají potřebné informace z jednotlivých tříd. Z v části 4.3.3 popsaných technologií využívám pro získávání dat především Tree API. Data tříd jsou získávána z `ClassNode`, data fieldů z `FieldNode` a data metod z `MethodNode`. Vzhledem k tomu, že Tree API neobsahuje odpovídající funkcionalitu pro získávání informací ze signatur, bylo zapotřebí ještě implementovat odpovídající `SignatureVisitor`, který implementuje třída `MethodMetricsSignatureVisitor`.

Rozhraní `FieldMetrics` je implementováno třemi třídami. První třídou je abstraktní třída `AbstractFieldMetrics`, která provádí společnou funkcionalitu pro druhé dvě třídy. Touto funkcionalitou je porovnávání (`equals`) a výpočet hash codu (`hashCode`). K implementaci dvou různých konkrétních podob rozhraní `FieldMetrics` vedla skutečnost, že některé metriky rozlišují, zda je používán field (u metod je to obdobné) třídy v rámci komponenty nebo tento field náleží třídě, která se v komponentě nenachází. Proto jsem implementoval dvojici tříd `ExternalFieldMetrics` (pro fieldy tříd mimo komponentu) a `FieldMetricsImpl` (pro fieldy tříd v komponentě).

Obdobně jako u rozhraní `FieldMetrics` je i u rozhraní `MethodMetrics` implementována trojice tříd. Vedle abstraktní třídy `AbstractMethodMetrics`, která se stará o porovnávání a výpočet hash code, to jsou třídy `ExternalMethodMetrics` (pro metody tříd mimo komponentu) a `MethodMetricsImpl` (pro metody tříd v rámci komponenty). V rámci získávání dat z bytecode se provádí analýza instrukcí dané metody (volání ostatních metod a používání fieldů). Provádí se také výpočet McCabovy Cyclomatické složitosti, kterým se budu zabývat v části 6.3.4.

Rozhraní `ClassMetrics` je implementováno třídou `ClassMetricsImpl`. Tato třída se stará o řízení parsování jednotlivých metod a fieldů a o získávání informací a podkladových metrik na úrovni třídy.

Rozhraní `ClassesMetrics` je implementováno třídou `ClassesMetricsImpl`. Jedná se o obalovou třídu seznamu (`List`) jednotlivých objektů `ClassMetrics` s dodatečnou funkcionalitou. Touto funkcionalitou je propojení vnitřních odkazů na konkrétní objekty rozhraní `FieldMetrics` a `MethodMetrics`. V době zpracování jednotlivých instrukcí metod jsou v rámci objektů `MethodMetricsImpl` veškeré volání metod a používání fieldů zaznamenány v podobě objektů tříd `ExternalFieldMetrics` resp. `ExternalMethodMetrics` a to bez ohledu na to, zda se jedná o fieldy a metody tříd mimo komponentu nebo v komponentě. Je to z důvodu, že v danou chvíli ještě není znám konečný seznam všech tříd komponenty (třídy jsou zpracovávány postupně). Pro rychlejší výpočet metrik je zapotřebí rozlišit, zda jsou fieldy a metody tříd v komponentě nebo mimo komponentu. K tomuto napomáhá nahrazení objektů tříd `ExternalFieldMetrics` a `ExternalMethodMetrics` odkazy na skutečné objekty v rámci `ClassMetricsImpl`. K nahrazení dočasných objektů odkazy na konkrétní objekty slouží metody `connectUsedOutClassFields` a `connectCalledMethods`, které musí být zavolány před prvním použitím seznamu jednotlivých `ClassMetrics`.

6.2.3 Společná rozhraní pro metriky komponent

Před samotnou implementací jednotlivých metrik komponent bylo zapotřebí unifikovat způsob, jak bude `MetricsIndexer` vypočtené hodnoty získávat. K tomuto účelu jsem vytvořil sadu rozhraní, které jednotlivé třídy pro výpočet metrik musí implementovat.

Výchozím rozhraním je rozhraní `Metrics`, které vyžaduje od každé metriky metodu pro inicializaci (`init` – v případě, že potřebuje provést nějakou přípravu před samotným výpočtem) a metody pro získání názvu metriky a návratového typu (některé metriky mohou vracet výsledek ve formě celého čísla, některé jako číslo desetinné).

Od rozhraní `Metrics` jsou odvozena rozhraní `ComponentMetrics`, `PackageMetrics` a `ServiceMetrics`. Ty jsou určené pro výpočet hodnot metrik na úrovni komponenty, na úrovni exportovaného balíčku a na úrovni exportované service. Předpokládá se, že pro některé metriky je možné provádět výpočet na více než jedné úrovni. Proto implementace výpočtu metriky může implementovat více než jedno rozhraní. V rámci této práce se rozhraní `ServiceMetrics` přímo nepoužívá a je připraveno pro budoucí použití.

6.2.4 Třída MetricsIndexer

Již několikrát zmíněná třída `MetricsIndexer` řídí získávání dat z jar souboru a následný výpočet metrik a ukládání vypočtených hodnot do metadat. V rámci metody `index` je načítán jar soubor a na základě jednotlivých parsovaných tříd pomocí ASM jsou vytvářeny jednotlivé `ClassMetrics` které tvoří seznam `ClassesMetrics`.

V metodě `computeAndSaveMetrics` jsou nejprve vytvořeny jednotlivé objekty pro výpočet metrik (v případě přidávání nových metrik je zapotřebí editovat tuto metodu), provedena jejich inicializace (metoda `init`) a následně jsou vypočteny hodnoty jednotlivých metrik (voláním metody `computeValue` resp. `computeValueForPackage`). Hodnoty jednotlivých metrik jsou uloženy do metadat metodou `addAtrributesToMetricsProperty`.

6.3 Implementace jednotlivých metrik

V rámci této práce jsem implementoval celkem šest metrik. Dvě metriky jsou počítány na úrovni exportovaných balíčků (API complexity a Ripple effect) a čtyři metriky jsou počítány na úrovni celé komponenty (Number of Imports, McCabova cyklomatická složitost, WTCoh a WTCoup). Definice jednotlivých metrik byla popsána v části 5.1.

6.3.1 Number of Imports

Třídou implementace metriky Number of Imports je třída `NumberOfImportsMetrics`. Metrika je počítána pro komponentu jako celek.

Metrika Number of Import počítá počet importovaných balíčků a services. Informace o importovaných balíčcích a services je obsažena v souboru manifest.mf, který je součástí jar

souboru komponenty. Jedním z možných přístupů by bylo např. parsování tohoto manifestu pomocí třídy `java.util.jar.Manifest`.

Vzhledem k tomu, že se o parsování manifestu v CRCE již stará jiný plugin a bylo by duplicitní tuto funkcionalitu implementovat opětovně, využívá se pro výpočet metriky informací v metadatech již obsažených. Hodnotou metriky je tedy počet položek seznamu requirement resource.

6.3.2 API complexity

Implementací metriky API complexity je třída `CpcMetrics`. Jako metrika byla zvolena metrika CPC s pevně stanovenými váhami. Metrika je počítána pro každý exportovaný balíček zvlášť.

Metrika využívá řadu předpočítaných hodnot ve fázi parsování dat ze tříd pomocí ASM (v jednotlivých třídách `ClassMetrics`). Z veřejných tříd exportovaného balíčku získává počty jednoduchých a složených fieldů, jednoduchých a složených parametrů metod, počty metod, tříd a rozhraní. Na základě těchto počtů a definovaných vah vypočítává hodnotu komplexity daného exportovaného balíčku.

Algoritmus lze zjednodušeně vyjádřit následujícím pseudokódem:

```
double computeAPIComplexity(packageName, classesMetrics)

  SET classCount to 0
  SET interfaceCount to 0
  SET methodCount to 0
  SET simpleTypeFieldCount to 0
  SET complexTypeFieldCount to 0
  SET simpleParametersCount to 0
  SET complexParametersCount to 0

  FOR each classMetrics in classesMetrics
    IF classMetrics is public AND classMetrics belong to packageName
      simpleTypeFieldCount ADD simpleTypeFieldCount of classMetrics
      complexTypeFieldCount ADD complexTypeFieldCount of classMetrics
      simpleParametersCount ADD simpleParametersCount of classMetrics
      complexParametersCount ADD complexParametersCount of classMetrics
      methodCount ADD methodCount of classMetrics

      IF classMetrics is interface
        INCREMENT interfaceCount
      ELSE
        INCREMENT classCount
      ENDIF
    ENDIF
  ENDFOR

  cmpC = classCount * CLASS_WEIGHT + interfaceCount
    + methodCount * METHOD_WEIGHT
```

```
  sumClassComplexity = simpleTypeFieldCount
    + COMPLEX_FIELD_WEIGHT * complexTypeFieldCount

  sumMethodComplexity = simpleParametersCount
    + complexParametersCount * COMPLEX_PARAMETER_WEIGHT

  cpc = cmpC + sumClassComplexity + sumMethodComplexity

RETURN cpc
```

6.3.3 Ripple effect

Metrika Ripple effect je metrika počítající všechny potencionálně vyvolané metody na základě volání veřejných metod tříd z exportovaného balíčku. Metrika se počítá pro každý exportovaný balíček zvlášť. Implementací této metriky je třída `RippleEffectMetrics`.

Pro algoritmus jsou zapotřebí dva seznamy metod (majících formu množiny). Prvním seznamem je seznam všech objevených metod (dosažitelných). To jsou metody, které hledáme – metody, které jsou potencionálně vyvolané (přímo nebo řetězově). Druhý seznam bude obsahovat metody s implementací (mohou volat jiné metody) a to pouze ty, které zatím nebyly prozkoumány.

V prvním kroku naplníme oba seznamy veřejnými metodami tříd exportovaného balíčku. Následně vezmeme první metodu z druhého seznamu (metody pro prozkoumání), metodu ze seznamu odstraníme. Získáme seznam touto metodou volaných metod a ty metody, které zatím neobsahuje první seznam, do něj přidáme. Pokud některá z těchto metod náleží třídě v rámci vyšetřované komponenty, přidáme metodu také do druhého seznamu. Skutečnost, že přidávaná metoda ještě nebyla v prvním seznamu, zaručuje, že nebude žádná metoda vyšetřována vícekrát. Metody tříd, které jsou mimo vyšetřovanou komponentu, dále prozkoumávat nebudeme a nejsou proto přidávány do druhého seznamu. Tento postup opakujeme až do chvíle, kdy je druhý seznam prázdný.

Poté, co jsou shromážděny veškeré potencionálně vyvolané metody, se spočte počet nalezených metod tříd v komponentě s implementací (ne-abstraktních), počet nalezených abstraktních metod tříd v komponentě a počet nalezených metod tříd mimo komponentu. Výsledkem metriky je součet všech těchto hodnot. Jednotlivé hodnoty jsou počítány zvlášť z důvodu možné budoucí změny způsobu výpočtu koncové hodnoty metriky.

Algoritmus lze zjednodušeně vyjádřit následujícím pseudokódem:

```
long computeRippleEffect(packageName, classesMetrics)

  FOR each classMetrics in classesMetrics
    IF classMetrics is public AND classMetrics belong to packageName
      ADD classMetrics into colletedMethods
      ADD classMetrics into methodsToVisit
    ENDIF
  ENDFOR
```

```
  WHILE methodsToVisit not empty
    investigatedMethod = GET first from methodsToVisit
    REMOVE first from methodsToVisit

    calledMethods = GET calledMethods of investigatedMethod
    FOR each method of calledMethods
      IF method is not in colletedMethods
        ADD method into colletedMethods
        IF method has calledMethods
          ADD method into methodsToVisit
        ENDIF
      ENDIF
    ENDFOR
  ENDWHILE

  SET internalNonAbstractMethodsCount to 0
  SET internalAbstractMethodsCount to 0
  SET externalMethodsCount to 0

  FOR each method in colletedMethods
    IF method is internal
      IF method is abstract
        INCREMENT internalAbstractMethodsCount
      ELSE
        INCREMENT internalNonAbstractMethodsCount
      ENDIF
    ELSE
      INCREMENT externalMethodsCount
    ENDIF
  ENDFOR

  rippleEffect = internalNonAbstractMethodsCount
  + internalAbstractMethodsCount
  + externalMethodsCount

RETURN rippleEffect
```

6.3.4 McCabova cyklomatická složitost (CYCLO)

Pro každou třídu, pro kterou má výpočet smysl (má alespoň jednu implementovanou metodu), je vypočítán průměr hodnot McCabovy cyklomatické složitosti jednotlivých metod a to v době parsování pomocí ASM. V rámci CRCE Metrics pluginu jsou počítány celkem tři hodnoty za celou komponentu – průměr z průměrů jednotlivých tříd, maximální hodnota průměrů jednotlivých tříd a minimální hodnota průměrů jednotlivých tříd. Jednotlivými třídami implementace těchto metrik jsou třídy `AverageCyclomaticComplexity`, `MinimumCyclomaticComplexity` a `MaximumCyclomaticComplexity`.

Před samotným výpočtem průměrné hodnoty za jednotlivé třídy je zapotřebí vypočíst hodnotu cyclomatické složitosti pro jednotlivé metody. To se děje v době parsování pomocí ASM. Pro výpočet je využita třída `Analyzer` z frameworku ASM. Je použit algoritmus

popsaný v [ASMG] s tím, že je použit upravený vzorec pro výpočet:

$$CYCLO = edges - nodes + 2 + endNodes - 1$$

Na úrovni třídy `ClassMetricsImpl` je vypočtena cyklomatická složitost za danou třídu. Výpočet hodnot na úrovni komponenty je počítán již zmiňovanými třemi třídami. V případě, že není pro některé komponenty hodnota metriky definována (např. obsahují pouze rozhraní), je výsledkem příslušné metriky hodnota `NaN`.

6.3.5 Metrika WTCoh

Implementací metriky WTCoh je třída `WTCohMetrics`. Metrika je počítána pro komponentu jako celek.

Metrika WTCoh se počítá na základě soudržnosti jednotlivých tříd. Nejprve je tedy zapotřebí postupně vypočíst hodnotu soudržnosti pro jednotlivé třídy, které mají implementovány alespoň dvě metody. Tato podmínka vychází z definice soudržnosti, která je počítána na základě podobnosti metod (ve smyslu přístupu k instančním proměnným). V rámci provádění algoritmu se procházejí všechny třídy v komponentě. Pro každou třídu je získán seznam implementovaných metod. V případě, že daná třída má alespoň dvě implementované metody, je pro ni počítána hodnota $ClassCohT$.

Nejprve je pro každou dvojici metod vypočítána přímá podobnost metod (ve smyslu přístupu k instančním proměnným). Tato podobnost je symetrická (platí $SimD(i,j) = SimD(j,i)$). Následuje výpočet nepřímé podobnosti modifikovaným Floyd–Warshallovo algoritmem, jak bylo popsáno v části 5.1.5. Nyní je možné sečíst jednotlivé vypočtené nepřímé podobnosti a vydělit počtem hran grafu ($m^2 - m$).

Hodnoty $ClassCohT$ za jednotlivé třídy jsou postupně sčítány a po provedení výpočtu pro všechny relevantní třídy v rámci komponenty je tento součet vydělen počtem relevantních tříd. V případě, že v komponentě není žádná relevantní třída (např. obsahuje pouze rozhraní), není hodnota této metriky definována a výsledkem je hodnota `NaN`.

Výpočet $ClassCohT$ pro jednotlivé třídy ukazuje následující algoritmus. Použité proměnné `unionVIVJ` a `intersectionVIVJ` představují sjednocení a průnik množin používaných metod.

```
double computeWTCoh(classesMetrics)

  SET classCohSum to 0
  SET classesCount to 0

  FOR each classMetrics in classesMetrics

    CLEAR methods
    FOR each method in methods of classMetrics
      IF method is not abstract
        ADD method into methods
```

```
      ENDIF
    ENDFOR

    FOR j 1 to methodCount - 1
      FOR i 0 to j - 1
        SET unionVIVJ as union of methods[i] and methods[j]
        SET intersectionVIVJ as intersection of methods[i] AND methods[j]
        simD = size of intersectionVIVJ / size of unionVIVJ

        sim[i][j] = simD
        sim[j][i] = simD
      ENDFOR
    ENDFOR

    FOR k 0 to methodCount - 1
      FOR i 0 to methodCount - 1
        FOR j 0 to methodCount - 1
          IF k != i AND k != j AND i != j
            SET sim[i][j] as max of sim[i][j] OR sim[i][k] * sim[k][j]
          ENDIF
        ENDFOR
      ENDFOR
    ENDFOR

    SET simSum to 0
    FOR j 1 to methodCount - 1
      FOR i 0 to j - 1
        ADD sim[i][j] into simSum
        ADD sim[j][i] into simSum
      ENDFOR
    ENDFOR
    classCohT = simSum / (methodsCount^2 - methodsCount)

    ADD classCohT into classCohSum
    INCREMENT classesCount
  ENDFOR

  wTCoh = classCohSum / classesCount

RETURN wTCoh
```

6.3.6 Metrika WTCoup

Metrika WTCoup je implementována třídou `WTCoupMetrics`. Metrika je počítána pro komponentu jako celek.

Metrika počítá provázanost mezi třídami na základě vzájemného volání metod. Prvním krokem výpočtu je vybrání pouze těch tříd, které mají implementovánu alespoň jednu metodu, čímž je zaručeno, že daná třída může (nemusí) využívat jiné třídy (nejedná se o rozhraní).

Následně je třeba vypočíst přímou provázanost mezi třídami. Pro výpočet přímé prová-

zanosti je zapotřebí nejprve určit celkovou funkcionalitu třídy, která je tvořena množinou fieldů a metod vlastní třídy a fieldů a metod všech ostatních tříd, které využívá. To představuje jmenovatele přímé provázanosti. Přímá provázanost je počítána vzhledem ke každé vyšetřované třídě v rámci komponenty. Vedle provázanosti s třídami v rámci komponenty, používají některé třídy metody a fieldy ze tříd, které nejsou součástí komponenty (knihovní třídy a třídy z importovaných komponent). Proto je pro každou třídu vypočtena i celková provázanost s třídami mimo komponentu, která bude později přičtena.

Ve chvíli, kdy je vypočtena přímá provázanost mezi jednotlivými třídami v komponentě, je možné přejít k dalšímu kroku, jímž je výpočet nepřímé provázanosti. K tomu, jako u metriky WTCoh, slouží upravený Floyd–Warshallův algoritmus. Poté, co je vypočtena nepřímá provázanost, následuje sumarizace celkové provázanosti komponenty. V této fázi jsou sčítány všechny hodnoty vzájemné nepřímé provázanosti a je připočtena provázanost se třídami mimo komponentu. Následně je tento součet vydělen počtem hran grafu ($m^2 - m$). V případě, že jsou v komponentě méně než dvě třídy s implementovanými metodami (např. pouze rozhraní), hodnota metriky není definována a výsledkem je hodnota `NaN`.

Výpočet lze zjednodušeně znázornit následujícím pseudokódem:

```
double computeWTCoup(classesMetrics)

  FOR each classMetrics in classesMetrics
    IF classMetrics has implemented method
      ADD classMetrics into investigatedClasses
    ENDIF
  ENDFOR

  FOR i 0 to investigatedClassesCount - 1
    SET classI to investigatedClasses[i]

    SET denominator to methodCount of classI
    ADD filedsCount of classI into denominator
    ADD allUsedOutClassMethods of classI into denominator
    ADD allUsedOutClassFieldsCount of classI into denominator

    FOR j 0 to investigatedClassesCount - 1
      SET numerator to count of usedFiledsOfClassJByClassI
      ADD count of usedMethodsOfClassJByClassI into numerator
      coup[i][j] = numerator / denominator
    ENDFOR

    SET numerator to count of usedFiledsOfClassOutOfJarByClassI
    ADD count of usedMethodsOfClassOutOfJarByClassI into numerator
    outJarCoup[i] = numerator / denominator

    FOR k 0 to investigatedClassesCount - 1
      FOR i 0 to investigatedClassesCount - 1
        FOR j 0 to investigatedClassesCount - 1
          IF k != i AND k != j AND i != j
            SET coup[i][j] as max of coup[i][j]
              OR coup[i][k] * coup[k][j]
```

```
          ENDIF
        ENDFOR
      ENDFOR
    ENDFOR
  ENDFOR

  SET coupSum to 0
  FOR j 1 to investigatedClassesCount - 1
    FOR i 0 to j - 1
      ADD coup[i][j] into coupSum
      ADD coup[j][i] into coupSum
    ENDFOR
  ENDFOR
  FOR i 1 to investigatedClassesCount - 1
    ADD outJarCoup[i][j] into coupSum
  ENDFOR

  wTCoup = coupSum / (investigatedClassesCount^2
    - investigatedClassesCount)

RETURN wTCoup
```

6.4 Úpravy WebUI

V souvislosti se začleněním CRCE Metrics pluginu do současné podoby CRCE bylo zapotřebí provést určité úpravy WebUI. Současné WebUI je převzato z verze 1.0 a není uzpůsobeno pro zobrazování nové podoby metadat. Součástí WebUI je třída `ResourceWrap`, která se stará o zabalení nových metadat tak, aby byly zobrazitelné v rámci staré verze webového rozhraní.

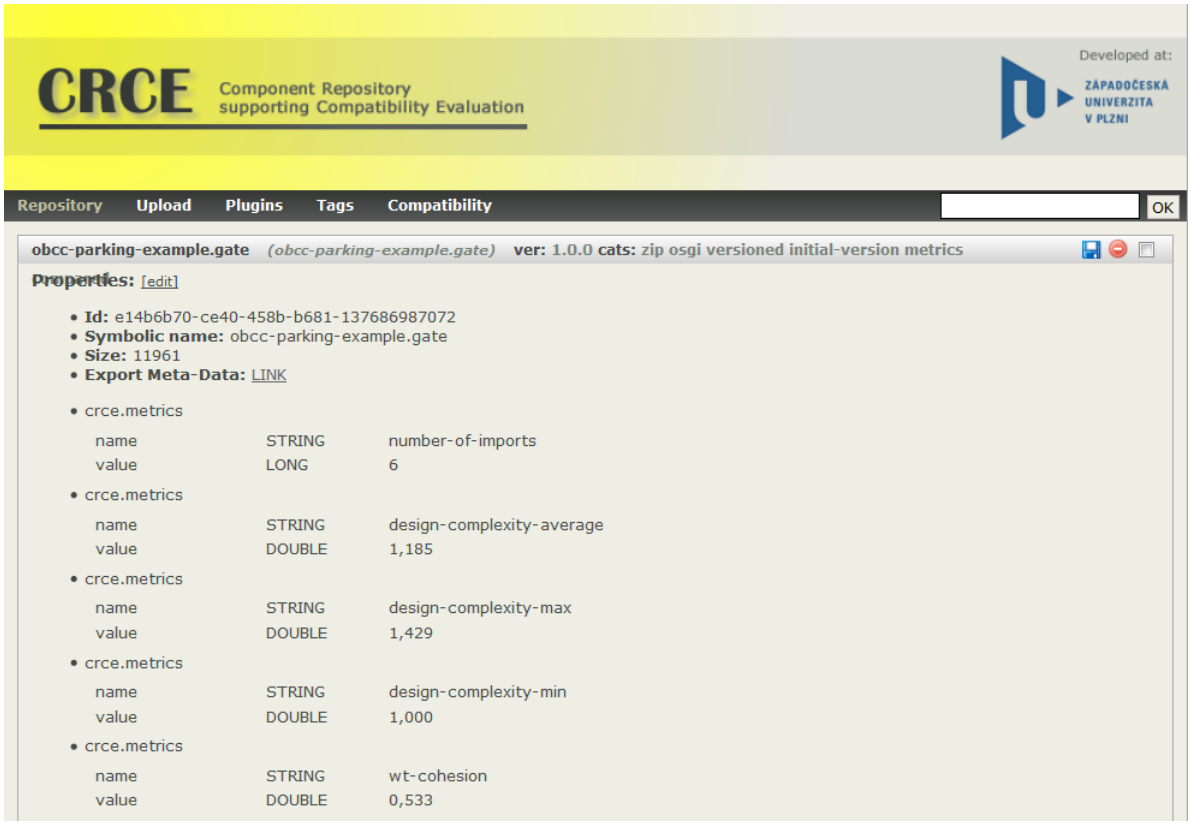
6.4.1 Property

V CRCE verzi 1.0 byly metadata značně omezená. Součástí těchto metadat byla položka property, které ale měla funkcionalitu odpovídající funkcionalitě atributů v současné podobě metadat. Dnešní podoba property v původních metadatech neexistovala. Proto je součástí WebUI wrapper, který obaluje atribut z nových metadat na původní property. Rozhraní wrapperu se nachází v balíčku `cz.zcu.kiv.crce.webui.internal.legacy` a jmenuje se `Property`. Při rozšiřování funkcionality WebUI bylo zapotřebí zavést i novou podobu property. Bylo zapotřebí vytvořit rozhraní, které by nové property obalovalo. Nebylo možné použít stejné pojmenování. Proto jsem zvolil název rozhraní `NewProperty`.

Dále bylo zapotřebí rozšířit rozhraní pro wrapper resource (`Resource`) a pro wrapper capability (`Capability`) o metodou, která bude sloužit k získávání jednotlivých `NewProperty`. Tou je metoda `getNewProperties`.

6.4.2 ResourceWrap

Implementace jednotlivých rozhraní wrapperů se nachází ve třídě `ResourceWrap` balíčku `cz.zcu.kiv.crce.webui.internal.custom`. Ten se stará o samotné mapování jednotlivých metadat v nové podobě do legacy wrapperů. Zde jsem provedl implementaci jednotlivých `getNewProperties` metod a celého rozhraní `NewProperty`.



Obrázek 6.1: CRCE WebUI

6.4.3 Úprava jsp šablony

Poslední změny bylo zapotřebí udělat v jsp šabloně webové stránky. Vzhledem k tomu, že jsou metriky počítány až ve chvíli, kdy je resource ukládán do úložiště a vzhledem k tomu, že bude webové rozhraní v nejbližší době přepracováno, aby odpovídalo nové podobě metadat, nebyla zapotřebí změna jiné šablony než úložiště (store.jsp).

Šablonu bylo zapotřebí změnit ve dvou místech. Zaprvé na úrovni daného resource bylo zapotřebí rozšířit zobrazované properties o položky `NewProperty`. Druhým místem změny byl výpis jednotlivých cababilities, kde bylo opět zapotřebí pro každou capability zobrazit hodnoty položky `NewProperty`.

7 Ověření funkčnosti CRCE Metrics pluginu

V rámci implementace CRCE Metrics pluginu bylo zapotřebí odpovídajícím způsobem ověřit jeho funkčnost. Nejprve bylo zapotřebí ověřit správnost výpočtu ve smyslu dodržení postupu výpočtu metrik. Další ověřovanou vlastností byla doba výpočtu metrik.

7.1 Ověření správnosti pluginu

Správnost pluginu jsem ověřoval na projektu Car Park Demo Application [CarP]. Projekt Car Park Demo Application je jednoduchá ukázková aplikace. Cílem aplikace je simulace systému pro parkování. Aplikace se skládá z několika OSGi bundle (komponent). V průběhu vývoje vzniklo celkem 6 verzí. V rámci ověření jsem použil verzi 1 a verzi 6. Pro ověření správnosti implementace výpočtu metrik byla zvolena z důvodu relativní jednoduchosti a dostupnosti zdrojových kódů.

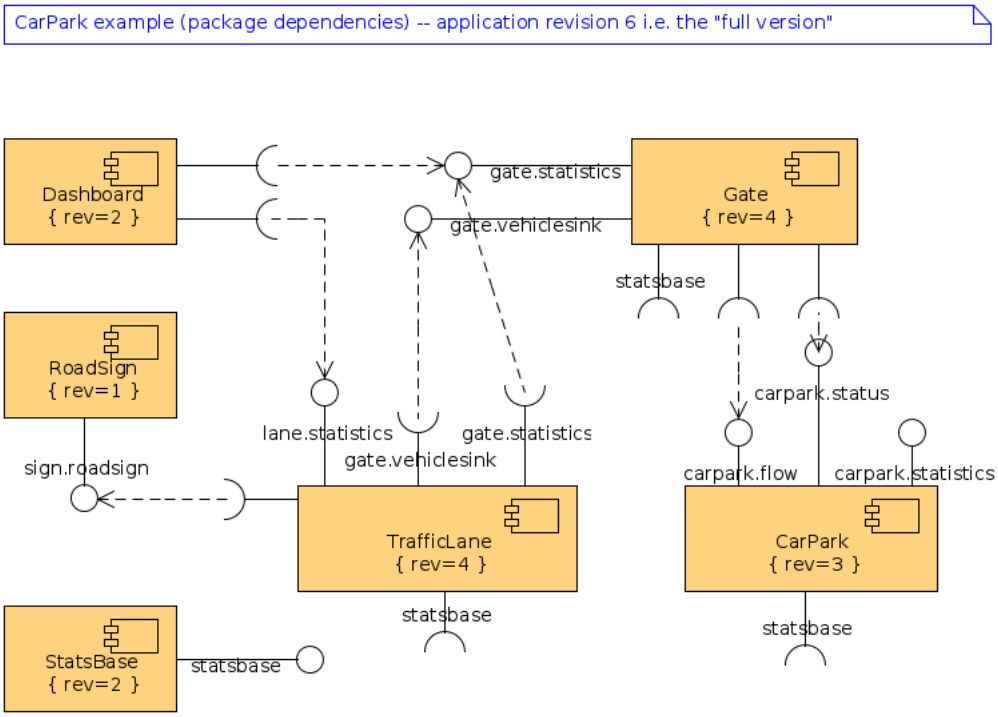
7.1.1 Podoba verze 1

Ve verzi 1 se skládá projekt celkem ze čtyř komponent. Těmito komponentami jsou `StatsBase`, `Gate`, `CarPark` a `Dashboard`. `StatsBase` je velice jednoduchá komponenta skládající se pouze z jednoho rozhraní, které je součástí jediného exportovaného balíčku. Komponenta `Gate` se skládá ze tří balíčků, kde dva jsou balíčky exportovanými. Tato komponenta je složena ze třech implementovaných tříd a dvou rozhraní. Komponenta `CarPark` se skládá ze tří balíčků, kde jsou opět dva exportované. Implementaci tvoří tři konkrétní třídy a dvě rozhraní. Poslední komponenta `Dashboard` se skládá ze dvou konkrétních tříd, které jsou součástí jediného balíčku, který již není exportovaný.

7.1.2 Podoba verze 6

Ve verzi 6 se projekt skládá ze šesti komponent. Oproti verzi 1 zde přibyly komponenty `TrafficLane` a `RoadSign`. Diagram závislostí jednotlivých komponent je vidět na obrázku 7.1. Komponenta StatsBase se v této verzi skládá vedle původního rozhraní také z jedné abstraktní třídy, která je taktéž součástí exportovaného balíčku. Z komponenty `Gate` byl vyčleněn jeden exportovaný balíček (skládající se z jednoho rozhraní a jedné konkrétní třídy) do komponenty `TrafficLane`. Implementace však byla rozšířena o jiný exportovaný balíček skládající se z jednoho rozhraní a jedné konkrétní třídy. Komponenta `TrafficLane` se skládá ze třech konkrétních tříd a jednoho rozhraní rozdělených do dvou balíčků. Exportovaný je balíček jeden. `RoadSign` se skládá ze dvou tříd a jednoho rozhraní rozdělených

do dvou balíčků s jedním z nich exportovaných. `CarPark` se v této verzi skládá ze čtyř konkrétních tříd a čtyř rozhraní rozdělených do čtyř balíčků. Exportované jsou balíčky tři. `Dashboard` se skládá ze dvou konkrétních tříd, které jsou součástí jediného balíčku, který není exportovaný.



Obrázek 7.1: Diagram projektu Car Park (převzato z [CarP])

7.1.3 Porovnání naměřených hodnot

Prostřednictvím CRCE Metrics pluginu byly naměřeny hodnoty metrik komponent obou verzí projektu. Hodnoty naměřených metrik za celou komponentu zachycují tabulky 7.1 pro verzi 1 a 7.2 pro verzi 6. Naměřené hodnoty pro jednotlivé balíčky zde neuvádím. Zájemce je může vyčíst z vygenerovaných metadat, které jsou součástí elektronické přílohy této práce.

Naměřené hodnoty lze vzájemně porovnávat a tak sledovat charakteristiky jednotlivých komponent, jak se v rámci vývoje vyvíjejí. Porovnávání verzí a dělání závěrů na základě změny hodnot metrik není předmětem této práce.

7.1.4 Ověření výpočtu metrik

Pro obě verze komponent bylo zapotřebí provést výpočet pomocí CRCE Metrics pluginu a zároveň manuálně na základě zdrojových kódů jednotlivých komponent. Při ručním

Balíček	NoI	CYCLO min	CYCLO max	CYCLO avg	WTCoh	WTCoup
`StatsBase`	0	NaN	NaN	NaN	NaN	NaN
`Gate`	6	1,000	1,429	1,185	0,533	0,285
`CarPark`	4	1,000	1,600	1,267	0,567	0,267
`Dashboard`	5	1,000	1,500	1,250	0,778	0,633

Tabulka 7.1: Metriky komponent Car Park Demo Application verze 1

Balíček	NoI	CYCLO min	CYCLO max	CYCLO avg	WTCoh	WTCoup
`StatsBase`	0	1,000	1,000	1,000	0,667	NaN
`Gate`	7	1,000	1,750	1,311	0,556	0,332
`CarPark`	6	1,000	1,600	1,289	0,474	0,191
`Dashboard`	4	1,000	1,500	1,250	0,778	0,652
`TrafficLane`	4	1,000	2,000	1,417	0,744	0,340
`RoadSign`	3	1,000	1,500	1,250	0,517	0,580

Tabulka 7.2: Metriky komponent Car Park Demo Application verze 6

výpočtu jsem musel zahrnout vlastnosti překladu do Java bytecode, které jsem popsal v části 5.2.2. U většiny metrik bylo ověření relativně jednoduché, ale v případě metrik WTCoh a WTCoup je výpočet značně náročný. Proto jsem jej provedl jen u několika náhodně vybraných komponent, abych ověřil správnost implementace algoritmu. Veškeré naměřené hodnoty z bytecode se shodují s hodnotami vypočtenými na základě zdrojového kódu.

7.2 Změření doby výpočtu metrik

Dobu výpočtu metrik jsem měřil u projektu CoCoME [CoCoME]. Projekt CoCoME (Common Component Modelling Example) je ukázková aplikace vybudovaná pomocí technologie OSGi. Projekt je vyvíjen napříč řadou univerzit. Tento projekt popisuje informační systém, který je možné vidět v supermarketu. Simuluje řadu činností, jako je čtení čárových kódů, funkci pokladny, zákazníka či pokladní. Projekt byl vybrán pro účely měření doby výpočtu metrik z důvodu velkého rozptylu velikostí jednotlivých komponent.

7.2.1 Předmět měření

V rámci měření jsou měřeny celkem tři různé časy. Prvním měřeným časem je doba parsování class souborů pomocí ASM a výpočet podkladových metrik. V této fázi dochází k načítání dat z jar souboru, který je uložený na disku. Proto na délku této fáze má vliv i doba diskových operací a práce se zip archivem (v podobě jar souboru). Druhou měřenou fází je fáze nahrazování dočasných objektů skutečnými referencemi me-

todami `connectUsedOutClassFields` a `connectCalledMethods` třídy `ClassesMetrics`, které probíhá v rámci volání metody `init` jednotlivých tříd pro výpočet metrik. Poslední měřenou fází je fáze výpočtu hodnot jednotlivých metrik a jejich přidání do metadat (doba uložení metadat do databáze již není měřena).

7.2.2 Složení aplikace

Projekt se skládá celkem ze 37 různých jar souborů. Z těchto 37 jar souborů je celkem 33 OSGi komponent. Zbývající soubory představují knihovní třídy. Jednotlivé jar soubory mají různou velikost. Od 515 bytu v případě nejmenší komponenty až po 4,06 MB v případě největší komponenty. Některé komponenty nejsou vytvořeny v rámci projektu CoCoME. Vedle komponent samotného projektu jsou součástí CoCoME například knihovny `org.apache.derby`, `org.hibernate` nebo `org.postgresql.jdbc`.

Projekt je relativně rozsáhlejší a popis jednotlivých komponent není pro účely měření podstatný. Zájemce se může s projektem podrobně seznámit na [CoCoME].

7.2.3 Hodnoty metrik

Vedle měření doby výpočtu metrik, kterou se budu zabývat dále, jsem pro všechny komponenty v projektu CoCoME naměřil i jejich hodnoty metrik. Vzhledem k rozsahu projektu zde uvádím pouze šest zástupců. Vedle největších komponent jsou zde tři zástupci menších komponent, které reprezentují komponenty podobné velikosti. Hodnoty naměřených metrik všech komponent lze vyčíst z metadat projektu CoCoME, které jsou součástí elektronické přílohy této práce.

Tabulka 7.3 zobrazuje velikosti jednotlivých komponent. Velikost je udávána dvěma hodnotami. První hodnotou je velikost jar souboru v kB resp. MB. Druhá hodnota udává počet class souborů – počet tříd a rozhraní komponenty. Velikost jednotlivých komponent může mít vliv na hodnoty některých metrik a má vliv na dobu jednotlivých fází. Součástí tabulky je i sloupec ID, který zde uměle zavádím, abych v případě potřeby v dalších tabulkách nemusel uvádět celý název komponenty.

Tabulka 7.4 ukazuje jednotlivé naměřené hodnoty metrik komponent. Je třeba zmínit hodnotu WTCoup u komponenty `com.springsource.org.hibernate` (ID 6), kde je v tabulce hodnota $0,000$. Hodnota je ve skutečnosti nenulová, ale vzhledem k počtu zobrazovaných desetinných míst byla na hodnotu nula zaokrouhlena.

7.2.4 Úpravy CRCE Metrics pluginu

Před samotným měřením doby výpočtu bylo zapotřebí rozšířit logovací schopnosti CRCE Metrics pluginu tak, aby ukládal jednotlivé naměřené časy pro pozdější zpracování do zvláštního souboru. Úpravy bylo zapotřebí provést ve zvláštní SVN branch, aby nedošlo k narušení běžného fungování pluginu.

ID	Balíček	Velikost souboru	Počet tříd
1	`cocome-osgiDS-dispatcher`	2,32 kB	2
2	`cocome-osgiDS-ScannerController-impl`	7,97 kB	5
3	`cocome-osgiDS-store-gui`	26 kB	17
4	`com.springsource.org.postgresql.jdbc4`	464 kB	189
5	`com.springsource.org.apache.derby`	2,42 MB	1438
6	`com.springsource.org.hibernate`	4,06 MB	2703

Tabulka 7.3: Velikost vybraných komponent CoCoMe

ID	NoI	CYCLO min	CYCLO max	CYCLO avg	WTCoh	WTCoup
1	4	NaN	NaN	NaN	NaN	NaN
2	9	1,000	1,286	1,057	0,616	0,173
3	7	1,000	4,222	1,928	0,467	0,047
4	6	1,000	11,333	2,046	0,221	0,004
5	20	1,000	21,400	2,225	0,228	0,001
6	46	1,000	11,333	1,588	0,210	0,000

Tabulka 7.4: Metriky vybraných komponent CoCoMe

CRCE používá pro logování `org.slf4j.Logger`. Výstup logování je definován v souboru lockback.xml. Zde lze nastavit více loggerů a logování do více souborů. Pro účely měření jsem nadefinoval speciální logger, který ukládá výstup do souboru crce-metrics.log. Logger je používán pouze ve třídě `MetricsIndexer`.

K měření času používám metodu `System.nanoTime`. Tato metoda slouží k nejpřesnějšímu měření ve smyslu uplynulého času. Není však nijak přímo vázaná na systémový čas (není možné ji použít k zjištění aktuálního času). Slouží k měření uplynulého času mezi jednotlivým voláním této metody. Proto je pro měření důležité v době testu neprovádět na testovacím počítači žádné jiné náročné operace, které by mohly mít vliv na výsledek měření (dobu měřených operací).

Vedle měřených časů je do logu ukládána také velikost jednotlivých komponent (v bytech) a počet class souborů (představující počet tříd a rozhraní komponenty).

7.2.5 Provedení měření

Jak jsem psal v předchozí části, měření bylo zapotřebí provádět na nezatíženém počítači. Aby nedošlo k ovlivnění doby výpočtu paralelním prováděním operací, každou komponentu jsem do úložiště nahrával zvlášť. Vždy jsem komponentu nahrával do prázdného úložiště. Pokus jsem prováděl pro každou komponentu pětkrát, abych bylo možné podchytit situace, kdy ovlivní dobu provádění některé měřené části neočekávaná událost jako např. práce garbage collectoru.

Měření bylo provedeno v rámci localhostu a to na počítači s procesorem Intel Core i7

Balíček	Time min	Time max	Time avg
`cocome-osgiDS-dispatcher`	0,190	0,200	0,195
`cocome-osgiDS-ScannerController-impl`	1,015	1,527	1,222
`cocome-osgiDS-store-gui`	3,538	3,724	3,612
`com.springsource.org.postgresql.jdbc4`	89,564	100,335	94,442
`com.springsource.org.apache.derby`	709,367	756,089	726,225
`com.springsource.org.hibernate`	793,086	1207,020	890,033

Tabulka 7.5: Doba (v $10^{-3}sec$) parsování pomocí ASM vybraných komponent CoCoMe

3,07 GHz, pamětí 8GB a Windows 7 x64.

7.2.6 Zpracování výsledků

Po naměření výsledků bylo zapotřebí data sumarizovat. Pro každou měřenou komponentu jsem pro každý měřený čas výpočtem určil průměrnou hodnotu, minimum hodnoty a maximum hodnoty. Naměřené časy jsem z jednotek nanosekund převedl na milisekundy ($10^{-3}sec$). Naměřené hodnoty v podobě Excel souboru jsou součástí elektronické přílohy této práce.

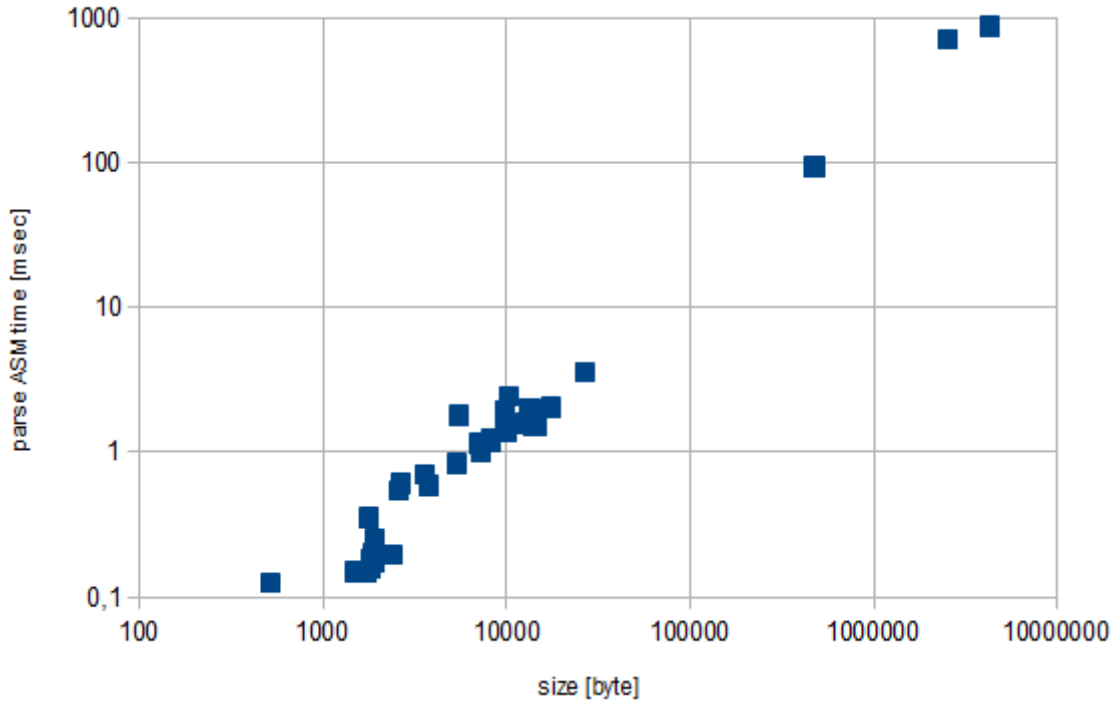
7.2.7 Výsledky měření

Naměřené hodnoty vybraných zástupců komponent jsou v tabulkách 7.5, 7.6, 7.7 a 7.8. Naměřené hodnoty všech komponent je možné získat z Excel souboru, který je elektronickou přílohou této práce. Tyto naměřené hodnoty jsem také zobrazil v podobě grafů 7.2, 7.3, 7.4 a 7.5. Všechny grafy používají logaritmicé měřítko.

Tabulka 7.5 ukazuje naměřené časy první fáze. Touto fází je načítání komponenty z disku a parsování jednotlivých class souborů pomocí ASM. V této fázi jsou počítány některé jednoduché metriky, které jsou podkladem pro výpočet metrik komponent. V tabulce jsou uvedeny tři časy – minimální čas, maximální čas a průměrný čas. Všechny zde uvedené časy jsou v milisekundách ($10^{-3}sec$). Odpovídající hodnoty znázorňuje graf 7.2. V grafu jsou znázorněny hodnoty všech měřených komponent projektu CoCoME.

Tabulka 7.6 ukazuje naměřené časy init fáze jednotlivých metrik. V této fází jsou nahrazovány dočasné objekty skutečnými referencemi metodami `connectUsedOutClassFields` a `connectCalledMethods` třídy `ClassesMetrics`. V tabulce jsou opět uvedeny tři časy – minimální čas, maximální čas a průměrný čas. Všechny zde uvedené časy jsou v milisekundách ($10^{-3}sec$). Odpovídající hodnoty znázorňuje graf 7.3. V grafu jsou opět znázorněny hodnoty všech měřených komponent projektu CoCoME.

Tabulka 7.7 ukazuje naměřené časy výpočtu jednotlivých metrik. V této fázi jsou počítány jednotlivé hodnoty a ukládány do metadat. Jako v předchozích případech tabulka obsahuje tři časy (minimální čas, maximální čas a průměrný čas) a uvedené časy jsou



Obrázek 7.2: Graf doby parsování pomocí ASM v závislosti na velikosti komponenty

v milisekundách ($10^{-3}sec$). Opět přikládám i odpovídající graf 7.4 pro všechny komponenty projektu CoCoME.

Poslední tabulkou je tabulka 7.8, která ukazuje celkovou dobu všech třech fází dohromady. Představuje tak celkovou dobu potřebnou pro získání všech počítaných metrik. Tabulka opět obsahuje tři časy (minimální čas, maximální čas a průměrný čas), které vznikly součtem dob jednotlivých fází (vždy odpovídajících danému pokusu). Uvedené časy jsou v milisekundách ($10^{-3}sec$). Odpovídající graf celkové doby získávání hodnot metrik je na obrázku 7.4.

7.2.8 Diskuse výsledků

Z naměřených hodnot je vidět, že velikost měřené komponenty má velký vliv na dobu měření. Od určité hranice velikosti komponenty, která se pohybuje někde v okolí 1 MB, začíná značně růst čas pro získávání hodnot metrik. Největší podíl na celkové době výpočtu takových komponent má samotná fáze výpočtu hodnot metrik komponenty a vliv přípravných fází je minimální. Pro výpočet velkých komponent je tedy výpočet metrik na pozadí (pomocí CRCE `Task`) nezbytností.

Na základě naměřených hodnot se však nabízí řada možných vylepšení, které by bylo v budoucnu možné realizovat. První zajímavou naměřenou hodnotou je doba první fáze i

Balíček	Time min	Time max	Time avg
cocome-osgiDS-dispatcher	0,007	0,011	0,009
cocome-osgiDS-ScannerController-impl	0,028	0,051	0,033
cocome-osgiDS-store-gui	0,167	0,184	0,171
com.springsource.org.postgresql.jdbc4	27,918	30,424	29,372
com.springsource.org.apache.derby	1299,221	1458,293	1359,072
com.springsource.org.hibernate	3035,040	3220,513	3097,037

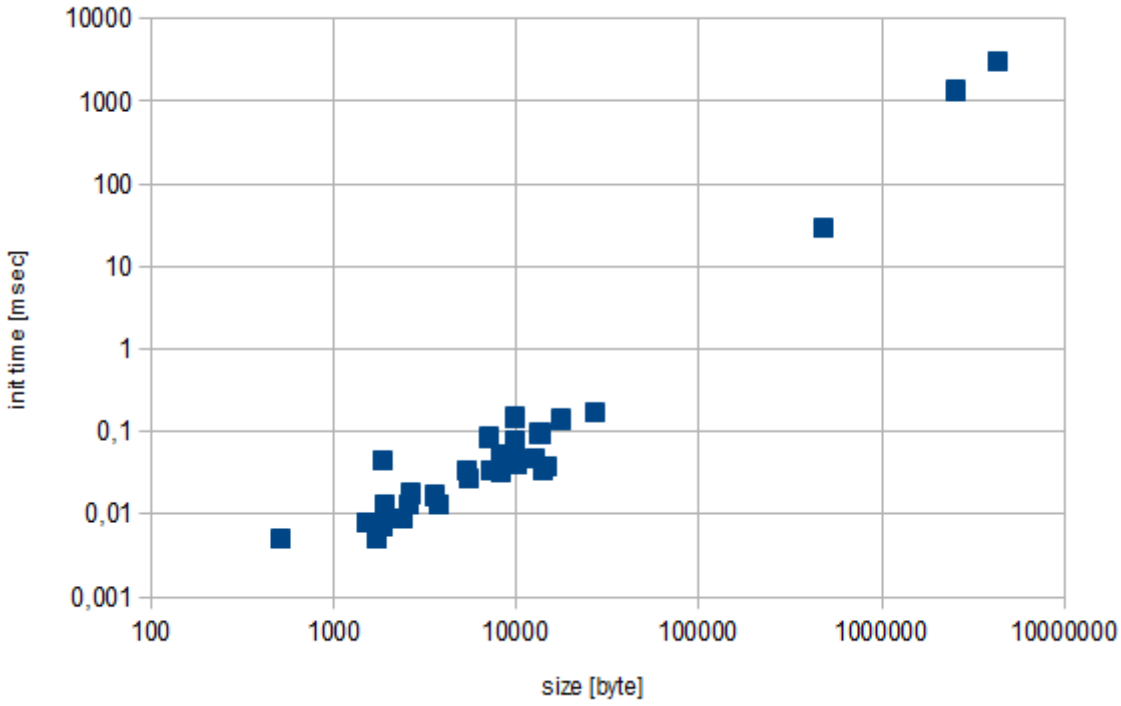
Tabulka 7.6: Doba (v $10^{-3}sec$) inicializace metrik vybraných komponent CoCoMe

Balíček	Time min	Time max	Time avg
cocome-osgiDS-dispatcher	0,174	0,510	0,303
cocome-osgiDS-ScannerController-impl	0,218	0,266	0,235
cocome-osgiDS-store-gui	0,646	0,683	0,668
com.springsource.org.postgresql.jdbc4	261,707	267,879	265,779
com.springsource.org.apache.derby	34803,923	35207,905	34947,135
com.springsource.org.hibernate	132129,856	135376,086	133589,996

Tabulka 7.7: Doba (v $10^{-3}sec$) výpočtu metrik vybraných komponent CoCoMe

Balíček	Time min	Time max	Time avg
cocome-osgiDS-dispatcher	0,382	0,718	0,506
cocome-osgiDS-ScannerController-impl	1,260	1,845	1,490
cocome-osgiDS-store-gui	4,354	4,559	4,452
com.springsource.org.postgresql.jdbc4	382,757	397,067	389,594
com.springsource.org.apache.derby	36865,367	37311,380	37032,432
com.springsource.org.hibernate	136013,357	139623,762	137577,066

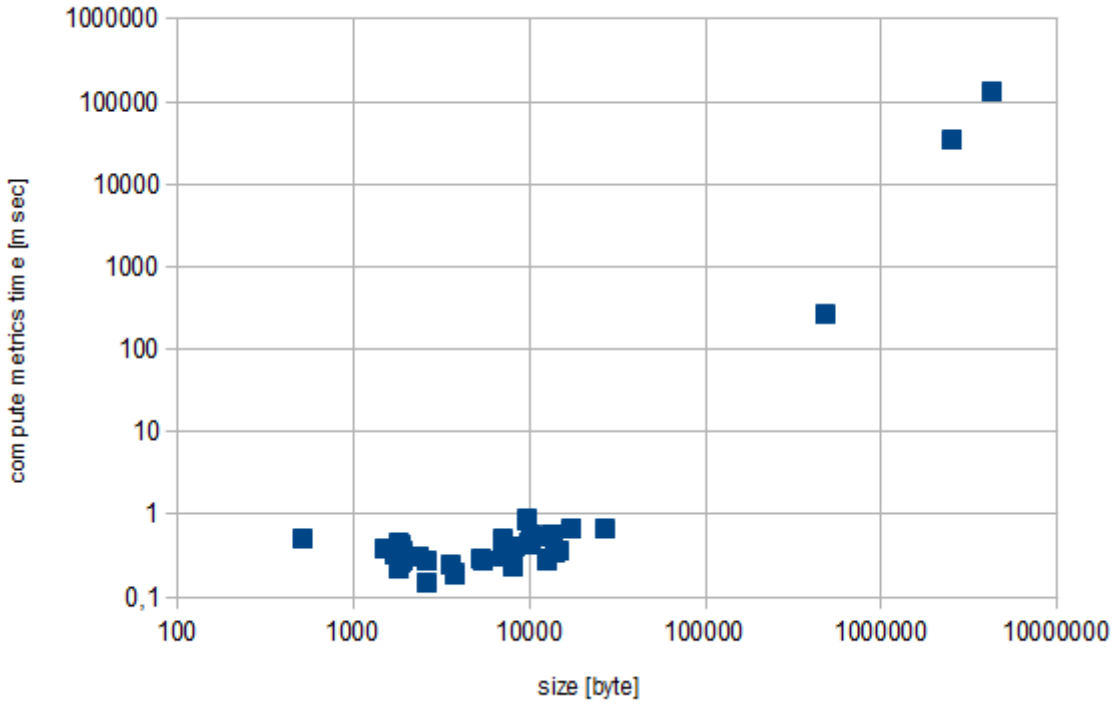
Tabulka 7.8: Celkové doba (v $10^{-3}sec$) výpočtu metrik vybraných komponent CoCoMe



Obrázek 7.3: Graf doby inicializace metrik v závislosti na velikosti komponenty

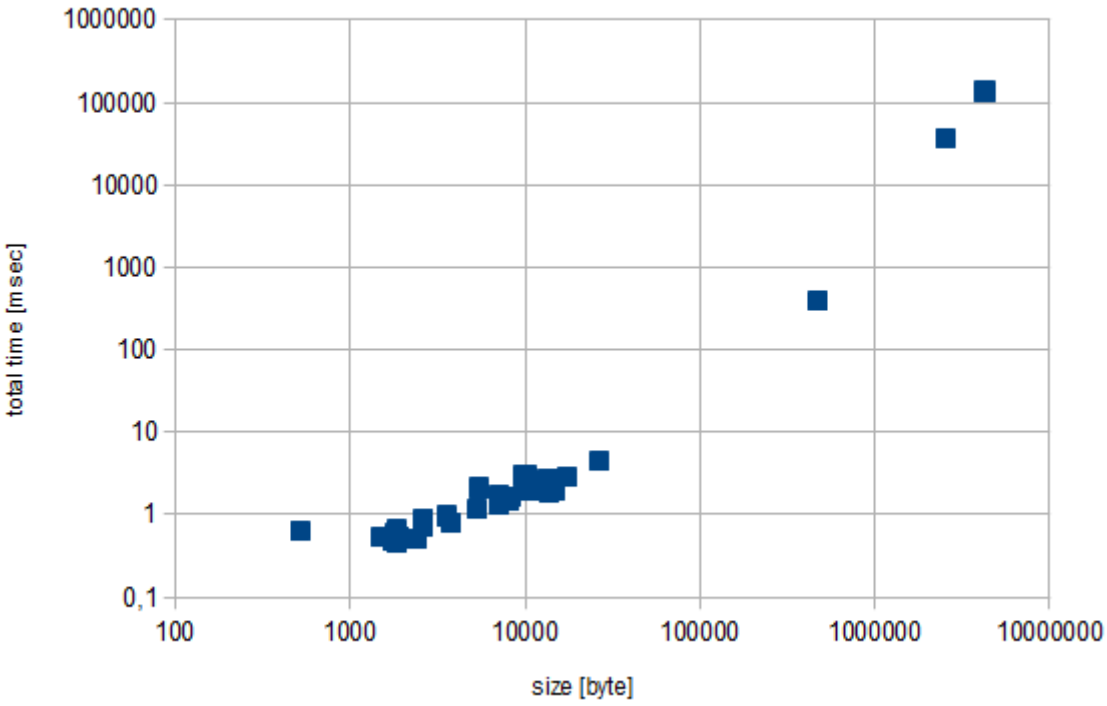
pro velké komponenty, kde v případě největší ověřované komponenty (velikost 4,06 MB) je průměrná doba této fáze menší než 1 sekunda. Dalo by se tedy uvažovat o rozdělení metrik do dvou skupin, kde by v první skupině byly metriky, u kterých by výpočet nebyl závislý na druhé fázi (inicializace) a doba třetí fáze (výpočtu hodnot metrik) by byla minimální. Do takové skupiny by mohla patřit např. metrika Number of Imports nebo McCabova cyklomatická složitost, u které jsou ve třetí fázi počítány pouze hodnoty minima, maxima a průměru za komponentu. Výpočet hodnot metrik této skupiny by bylo možné provádět již ve fázi nahrávání komponenty do bufferu (metoda `onUploadToBuffer`). Druhou skupinu by tvořily metriky, u kterých by výpočet zabral nepřiměřeně dlouhou dobu a jejichž hodnoty by byly získávány až v době ukládání komponenty do úložiště na pozadí, jako je tomu nyní. Pro přesné rozdělení metrik do těchto skupin by bylo zapotřebí provést měření výpočtu třetí fáze pro jednotlivé metriky.

Jinou možností řešení situace s dobou výpočtu metrik by byl test velikosti komponenty v době nahrávání do bufferu a poté při ukládání do úložiště. V případě, že by komponenta byla menší než nějaká konkrétní hodnota (např. menší než 1 MB), byl by výpočet metrik proveden již v době nahrávání do bufferu. Pokud by byla komponenta větší než stanovená hranice, výpočet by se provedl až ve chvíli ukládání do úložiště. Nevýhodou takového řešení by však byla nekonzistentnost a mohlo by to u některých uživatelů, kteří by nebyli s touto funkcionalitou obeznámeni, způsobit zmatení.



Obrázek 7.4: Graf doby výpočtu metrik v závislosti na velikosti komponenty

Dalším přínosem by bylo případné urychlení některých výpočtů. Toho by se dalo dosáhnout nasazením výkonnějších algoritmů pro výpočet metrik. Velkého urychlení by se mohlo dosáhnout např. při výpočtu složitosti metriky WTCoup, která vyšetřuje vzájemný vztah jednotlivých tříd (s implementovanými metodami). Algoritmus výpočtu této metriky, který jsem použil v této práci, má složitost $O(n^3)$. Nasazení rychlejšího algoritmu by tedy bylo velkým přínosem.



Obrázek 7.5: Graf celkové doby výpočtu v závislosti na velikosti komponenty

8 Závěr

Hlavním cílem této práce bylo prostudování produktových metrik software a možnosti jejich výpočtu z distribučních balíčků komponent. Na základě nastudovaných produktových metrik bylo třeba vybrat metriky pro komponenty a případně další vhodné metriky dodefinovat. Cílem pak byla implementace nástroje (v podobě CRCE pluginu), který hodnoty vybraných metrik počítá z distribučních balíčků komponent (OSGi).

V teoretické části jsem se zabýval měřením v obecné rovině a poté jsem se zaměřil na oblast softwaru a metrik. Prozkoumal jsem některé existující nástroje pro výpočet metrik. Z různých zdrojů (3 monografie, 4 odborné články) jsem shromáždil sadu produktových metrik, které pokrývají různé oblasti vývoje software (od klasických metrik až po metriky komponent). Prozkoumal jsem také funkcionalitu CRCE a metody pro analýzu bytecode.

Na základě shromážděných metrik jsme ve spolupráci s vedoucím práce definovali 6 různých metrik zaměřených specificky na black-box softwarové komponenty, které jsem v následující fázi implementoval v podobě pluginu CRCE. Metriky pokrývají rozhraní (API) komponent a strukturu komponent. CRCE Metrics plugin před touto prací neexistoval a proto jeho implementace zahrnovala vedle implementace výpočtu metrik také návrh pluginu, jeho začlenění do CRCE a následnou úpravu WebUI, aby byly naměřené hodnoty správně zobrazeny.

Funkcionalitu implementovaného pluginu jsem ověřil na vybrané sadě komponent. Ověřil jsem jeho správnost z hlediska výpočtu hodnot metrik tak, že jsem provedl výpočet metrik pro ověřované komponenty také na základě zdrojového kódu a hodnoty porovnal. Dále jsem provedl ověření pluginu měřením doby výpočtu metrik.

8.1 Co jsem se naučil

Když jsem si volil nabízené téma diplomové práce, očekával jsem, že v rámci této práce proniknu do problematiky produktových metrik a do problematiky bytecode. V době výběru této práce jsem měl o obou tématech jen mlhavé znalosti. Vzhledem k tomu, že bych se v příštích letech chtěl věnovat vývoji softwaru, považuji tato dvě témata za velice důležitá. Ač ve své praxi používám spíše jazyky C++ nebo C#, i pochopení bytecode Javy mi může pomoci být lepším vývojářem. Díky této práci se mé znalosti v obou oblastech značně rozšířily.

8.2 Možná vylepšení do budoucna

Jak jsem se zmiňoval v části 7.2.8, výpočet hodnot některých metrik u větších komponent může být značně časově náročný. S ohledem na tuto zkutečnost jsem v této části

navrhl řadu možných řešení. Jednou z navrhovaných řešení bylo rozdělení metrik do dvou skupin v závislosti na době měření s ohledem na velikost komponenty. Metriky, které lze i u velkých komponent měřit rychle, by byly počítány ve fázi nahrávání komponenty do bufferu. Metriky, které nelze měřit dostatečně rychle, by pak byly počítány na pozadí po nahrání do úložiště, jako je tomu nyní.

Jiným navrhovaným zlepšením je důraz na urychlení algoritmu výpočtu jednotlivých metrik. Především v případě metriky WTCoup by teoretická výměna algoritmu výpočtu tranzitivní provázanosti tříd mohla urychlit výpočet hodnoty metriky. Jinou možností je výpočet provázanosti tříd bez zahrnutí nepřímé (tranzitivní) provázanosti. V takovém případě by bylo možné použít např. verzi výpočtu metriky v podobě $WICoup$, která je popsána v [GuS09].

Zkratky

API	Application programming interface – rozhraní pro programování aplikací
ASM	Knihovna pro analýzu bytecode
CBD	Component-based Development – paradigma vývoje software pomocí komponent
CBSE	Component-based Software Engineering – paradigma vývoje software pomocí komponent
CIL	Common Intermediate Language – obdoba bytecode u .NET frameworku
COM	Component Object Model – komponentová technologie firmy Microsoft
CRCE	Component Repository supporting Compatibility Evaluation – komponentové úložiště vyvíjené na KIV ZČU
DAO	Data access object – objekt pro přístup k perzistentnímu úložišti
JaCC	Java class compatibility checker – knihovna pro porovnávání tříd v jazyce Java
JVM	Java virtual machine – program pro běh aplikací v bytecode
OBCC	OSGi Bundle Compatibility Checking toolset – sada nástrojů pro ověřování kompatibility OSGi komponent
OBR	OSGi Bundle Repository – specifikace pro úložiště komponent
OLE	Object Linking and Embedding – komponentová technologie firmy Microsoft
OSGi	Open Service Gateway initiative – komponentová technologie pro platformu Java
SOM	System Object Model – komponentová knihovna od firmy IBM
WebUI	Webové grafické rozhraní aplikace

Literatura

[ASM] **ASM website**
http://asm.ow2.org/index.html
online únor 2014

[ASMG] *Eric Bruneton* **ASM 4.0: A Java bytecode engineering library**
http://download.forge.objectweb.org/asm/asm4-guide.pdf
online duben 2014

[BaB11] *Bauml, J., Brada, P.* **Reconstruction of Type Information from Java Bytecode for Component Compatibility**
Electronic Notes in Theoretical Computer Science 264 (4),
2011

[BCEL] **BCEL website**
http://commons.apache.org/proper/commons-bcel/
online březen 2014

[Bos] *Debayan Bose* **Component Based Development**
http://arxiv.org/ftp/arxiv/papers/1011/1011.2163.pdf
online únor 2014

[BrJ12] *Brada, P., Jezek, K.* **Ensuring Component Application Consistency on Small Devices: A Repository-Based Approach.**
Proceedings of 38th Euromicro Conference on Software Engineering and Advanced Applications
IEEE Computer Society
2012

[CarP] *Brada, P.* **Car Park Demo Application project**
https://github.com/pbrada/obcc-parking-example/tree/pkg-deps-only
online duben 2014

[CCI] *Dr.D.Jeya Mala, M. Ramalakshmi Praba* **Critical components identification and verification for effective software test prioritization**
Proceedings of Third International Conference on Advanced Computing (ICoAC), 2011, Pages 181-186. ISBN 978-1-4673-0670-6. IEEE
2011

[CKJM] **CKJM website**
http://www.spinellis.gr/sw/ckjm/
online březen 2014

[CKK01] *Eun Sook Cho, Min Sun Kim, Soo Dong Kim* **Component metrics to measure component quality**
Proceedings of Eighth Asia-Pacific Software Engineering Conference, 2001. APSEC 2001. Pages 419-426. ISBN 0-7695-1408-1. IEEE.
2001

[CoCoME] **CoCoME website**
http://www.cocome.org/index.htm
online duben 2014

[CPA] **CodePro Analytix website**
https://developers.google.com/java-dev-tools/codepro/doc/
online březen 2014

[CRCE] **CRCE website**
https://www.assembla.com/spaces/crce/wiki
online únor 2014

[Dan14] *Daněk, J.* **Vytváření a zpřístupnění informací o kompatibilitě komponent**
Diplomová práce, ZČU [podána k obhajobě]
2014

[DeM82] *Tom DeMarco* **Controlling Software Projects: Management, Measurement, and Estimates**
Yourdon Press,
1982

[EMp] **Eclipse Metrics plugin website**
http://metrics2.sourceforge.net/
online březen 2014

[Fen91] *Norman E. Fenton* **Software metrics: A rigorous approach**
Chapman & Hall,
1991

[Flo62] *Robert W. Floyd* **Algorithm 97: Shortest path**
Communications of the ACM, 5(6): 345,
June 1962

[GuS09] *Gui Gui, Paul D. Scott* **Measuring Software Component Reusability by Coupling and Cohesion Metrics**
Journal of Computers, 4(9): 797-805,
September 2009

[ChK94] *Chidamber, S.R., Kemerer, C.F.* **A metrics suite for object oriented design**
Software Engineering, IEEE Transactions, 20(6): 476-493,
Jun 1994

[JaCC] **JaCC website**
https://www.assembla.com/spaces/jacc/wiki
online únor 2014

[JAr] **JArchitect website**
http://www.javadepend.com/
online březen 2014

[Java] **Java™ website**
http://www.java.com/en/
online duben 2014

[JVM] **The Java™ Virtual Machine Specification**
http://docs.oracle.com/javase/specs/jvms/se7/html/#type=article&q=
online srpen 2013

[Kan02] *Stephen H. Kan* **Metrics and Models in Software Quality Engineering (2nd Edition)**
Addison Wesley,
2002

[LMD06] *Michele Lanza, Radu Marinescu, S. Ducasse* **Object-Oriented Metrics in Practice**
Addison Wesley,
2006

[MaR02] *Robert C. Martin* **Agile Software Development, Principles, Patterns, and Practices**
Prentice Hall,
2002

[McC76] *T. J. McCabe* **A measure of complexity**
IEEE Transactions on Software Engineering, 2(4): 308–320,
December 1976

[MPSC] *Mass produced software components* **Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct., 1968: 79-87**
http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF
online únor 2014

[NDe] **NDepend website**
http://www.ndepend.com/
online březen 2014

[OBCC] **OBCC website**
https://www.assembla.com/spaces/obcc/wiki
online únor 2014

[OSGi] **OSGi website**
http://www.osgi.org/Main/HomePage
online duben 2014

[SE] **Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct., 1968**
http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF
online únor 2014

[SeCM] **SourceCodeMetrics NetBeans plugin website**
http://plugins.netbeans.org/plugin/42970/sourcecodemetrics
online březen 2014

[SiCM] **Simple Code Metrics NetBeans plugin website**
http://plugins.netbeans.org/PluginPortal/faces/PluginDetailPage.jsp?pluginid=9494
online březen 2014

Poznámka: Pokud u některých citací není údaj, nepodařilo se jej zjistit.

Seznam obrázků

Seznam tabulek