Západočeská Univerzita v Plzni
Fakulta aplikovaných věd

DIPLOMOVÁ PRÁCE

Antonín Slezáček

Statické ověření korektnosti vazeb aplikací třetích stran

Katedra informatiky

Vedoucí diplomové práce:	Ing. Kamil Ježek PhD.
Studijní program:	ININ
Studijní obor:	SWI

Plzeň 2013

Chtěl bych poděkovat všem, kteří ve mě věřili a byli mi stále oporou. Především chci poděkovat své mamince a sestře.

Holky jste báječné, nemohl bych si přát nikoho lepšího.

Dále bych chtěl poděkovat svému vedoucímu práce Ing. Kamilu Ježkovi PhD. Práce s ním byla úžasná, hlavně díky jeho chápavému přístupu a velmi rychlé odezvě na jakýkoli přednesený problém. Spolupráce si velmi vážím.

Díky Kamile.

Děkuji.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská Univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Statické ověření korektnosti vazeb aplikací třetích stran

Autor: Antonín Slezáček

Katedra: Katedra Informatiky

Vedoucí diplomové práce: Ing. Kamil Ježek PhD.

Abstrakt: Mezi jeden z nejrozšířenějších staticky typovaných porgramovacích jazyků patří bezesporu Java. Trendem moderního programovaní je využití komponent třetích stran. Závislosti a sestavení často obsarává Apache Maven. Problém je však v slabé definici závislosti. Často pak dochází k nekompatibilitám a pádům za běhu aplikace. Tato práce využívá statické analýzy byte kódu k odhalení nekompatibilit v systému. V podobě Maven plugin lze snadno zaintegrovat do vývojového procesu a preventivné předejí problémům spojených s nekompatibilitou komponent.

Klíčová slova: java, kompatibilita komponent, maven plugin

Title: Static Verification of Third-party Application Dependencies Correctness

Author: Antonín Slezáček

Department: Department of Computer Science and Engineering

Supervisor: Ing. Kamil Ježek PhD.

Abstract: One of most commonly used statically typed programming language is Java. Modern trend in programming is to leverage of using third party components. These dependencies and build cycle can be managed by Apache Maven. The problem lies in very weak definition of component dependency. Then incompatibilities comes to the place very often and as a result of that the application falls during the runtime. This work uses static analysis of the byte-code to detect incompatibilities in the system. As a Maven plugin it is easy to integrate this tool to the development process and with that prevent problems during runtime because of incompatibility of components.

Keywords: java, component compatibility, maven plugin

Obsah

Úvod

Představení

Doba, kdy se nadšenci předháněli v tom, kdo je schopen vyřešit problematiku na nejméně instrukcí s minimální paměťovou náročností, již dávno minula. Informační technologie v dnešní době zasahují téměř do všech oborů. Stal se z nich především nástroj pro podporu ostatních oborů a dovedností. Tato skutečnost pak udává trend jakým se IT ubírá.

Mezi priority, které jsou na IT kladeny, patří především cena na vývoj, časová náročnost, rozsah a komplexnost systému, schopnost reagovat na potřeby zákazníka, či měnící se potřeby trhu.

Dnes se vyvíjejí rozsáhlé komplexní systémy. Tyto systémy zasahují do všech částí, nalezneme je tedy, jak v podobě softwaru řídící výrobní linku, plánující výrobu, až po nástroje obchodní v podobě softwaru pro péči o zákazníky. Vezmeme-li v potaz jen cenu a časovou náročnost takového systému, pokud by byl vyvíjen pomocí assembleru, dojdeme k astronomickým číslům. Vysokoúrovňové jazyky odstiňují programátory od detailů a umožnují jim tak více pozornosti soustředit samotnému produktu. Avšak jazyk sám o sobě nestačil. Zde přicházejí k oblibě knihovny, řešící opakující se problémy. Programátor pak jen vhodně použije funkce knihoven a doplní o vlastní logiku. Dnešním trendem pak je komponentové programování, které namísto slepovacího kódu využívá kompozici.

Samozřejmě, že investice do takto rozsáhlých systémů je značná. Proto je velký důraz kladen na udržovatelnost systému, jeho rozšiřitelnost, konfigurovatelnost a testovatelnost.

Samozřejmý je i fakt, že vývoj systému probíhá po částech a odděleně. Každá součást systému je pak sama o sobě se vyvíjející část, která ale musí být schopná být začlněna do celého systému a správně s ostatními interagovat.

Na svět kompatibility interagujících částí je možné nahlédnout, jak z makro pohledu, kde spolu komunikují komponenty systému nebo dokonce systémy samotné. Nebo je zde možné zkoumat tento svět z druhé strany, z mikro pohledu, kde řešíme kompatibilitu na úrovni datových typů.

Shrnuli jsme si tedy jaké jsou trendy v moderním IT - komponentové programování, vysoko úrovňové jazyky. Tyto trendy zapříčinila potřeba dojít snadno, rychle a levně k cíli. Je dobré se zabývat kompatibilitou softwarových komponent, jelikož jejich vzájemná správná komunikace nutná k fungování celého systému. Pokud nebudou komponenty správné interagovat dojde k chybě za běhu, což muže vyústit až pádem celého systému.

Základem problému je absence mechanismu jasně definovat co komponenty

potřebují od ostatních komponent. Momentálně je možné jen specifikovat jaké komponenty program potřebuje. Definice vazby obsahuje pouze jméno závislé komponenty, v lepším případě ještě její verzi.

Vzrůstá tedy potřeba nástroje, který by kontroloval, zda spolu komponenty komunikují správně. Ideální by byla integrace tohoto nástroje již do vývoje. Tak bychom byli schopni ověřit korektnost vazeb mezi komponentami ještě ve fázi vývoje a předejít tak nepříjemným pádům systému za běhu.

Jakýkoli pád systému za běhu je velmi nepříjemný. Může způsobit například velké finanční ztráty.

O tom jaké problémy souvisí s tématem kompatibility jednotlivých součástí systému a jejich řesení pojednává kapitola 1.

Cíl práce

Tato práce si klade za cíl navrhnout a vyvinout nástroj, který odstraňuje absenci kontroly kompatibility u knihoven třetích stran a jejich závislostí.

1. Teoretická část

1.1 Problémy kompatibility softwarových komponent

Problematika kompatibility softwarových komponent je velmi široké téma. Tento text pojednává o specifících vysoko úrovňových, typových, překládaných jazyků, jakým je například Java.

Staticky typové jazyky se staly oblíbené především díky schopnosti odhalení a prevenci velkého množství chyb za běhu. Detekcí chyb během překladu nebo ještě v době psaní zdrojového kódu, je dosaženo rapidního snížení iterací testů vedoucích ke kýženému výsledku.

Během sémantické analýzy a překladu je možné odhalit chyby jako volání metod s nekompatibilní signaturou a přetypování. Chyby tohoto charakteru kontroluje překladač jazyka. Jak je vidět na obrázku 1.1, překladač má však omezené pole působnosti na kód, který je překládán a kód knihoven, který je z překládaného kódu volán. Není však již možné, aby překladač kontroloval kompatibilitu závislostí, které mohou mít využívané knihovny mezi sebou - červená šipka uvnitř překladačem kontrolované oblasti. Nebo ještě hůře, kompatibilitu závislostí těchto knihoven - šipky vně překladačem kontrolované oblasti.

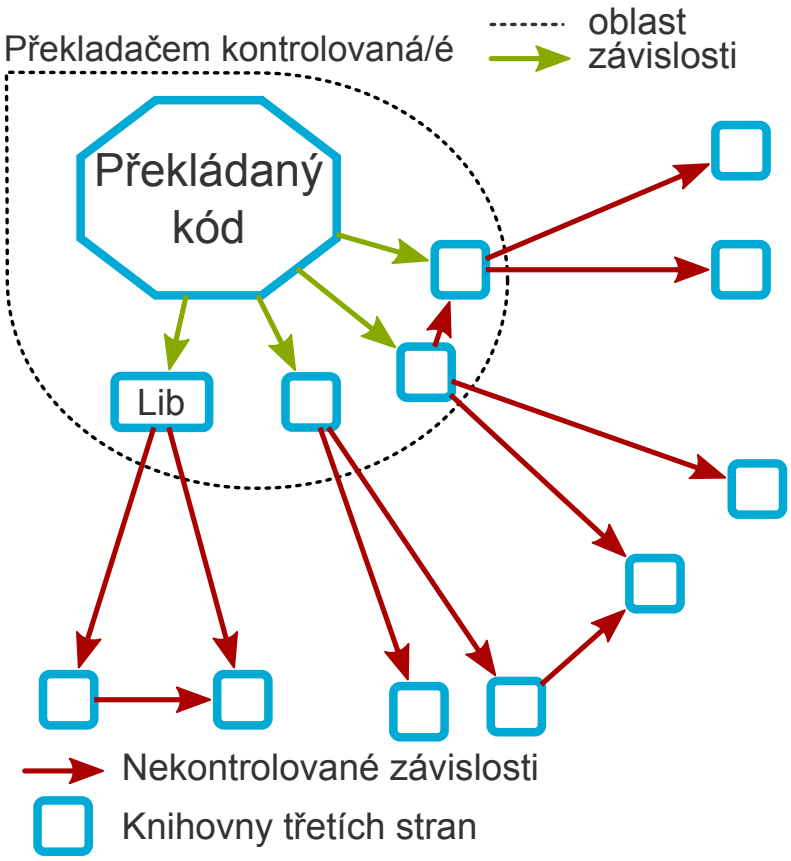
Základním problémem kontroly kompatibility je detailnost popisu vazby. Překladač má v tomto ohledu velmi dobré a podrobné informace o vazbě, jakým jsou signatury metody, informace o typech, návratových hodnotách atp. Je tedy schopen odhalit velké množství chyb. Naproti tomu však jsou nástroje, které si o takovém luxusu mohou nechat jen zdát. Tyto nástroje řeší například vazbu komponent nebo tříd za běhu programu. Často k rozlišení používají pouze jméno třídy či komponenty.

Dalším problémem je doba či fáze kontroly kompatibility. Překladač je schopen zkontrolovat kompatibilitu, s výše zmíněným omezením, pouze v době překladu. Pokud dojde k běhu programu s rozdílnými komponentami než byly dostupné za překladu, může dojít k chybám.

1.1.1 Statické systémy

Statický systém je během vývoje poskládán z jednotlivých částí a při běhu spustěn jako celek. Má-li být systém funkční, musí být jednotlivé části tohoto systému načteny na jedno místo, odkud budou dostupné. Takovému místu se v prostředí Javy říká `CLASSPATH`. `CLASSPATH` je seznam veškerých načtených a tedy dostupných tříd.

Jak jsme si již vysvětlili, jednotlivé knihovny se vyvíjejí i překládají odděleně,



Obrázek 1.1: Znázornění kontroly závislostí v i mimo kompetenci překladače

tudíž kontrola napříč komponentami není možná. Dalším aspektem je neustálý vývoj, tedy změna komponent. Vznikají tak různé verze té samé komponenty, které ale mohou být nezaměnitelné - jsou nekompatibilní.

Bohužel za běhu jediné co víme o třídě, na které je náš kód závislý, je jen její jméno! Přestože tedy je daná třída na `CLASSPATH` načtená, není zdaleka vyhráno. V případě, že se třída změnila a například již neobsahuje metodu, nebo se změnila signatura metody, kterou voláme, dojde za běhu k vyhození vyjímky `NoSuchMethodException`. K tomuto případu může dojít velmi snadno.

`CLASSPATH` u Javy Standard Edition je implementovaný jako plochý seznam s unikátními záznamy. Může klidně dojít k tomu, že přestože poskytujeme na `CLASSPATH` správnou verzi knihovny, některá knihovna, kterou využíváme, může mít závislost na té samé knihovně, avšak v jiné verzi. Pak mohou být připraveny k načtení 2 a více stejnojmenných tříd. Načtena však bude pouze ta třída náležící

balíčku, který je uveden k načtení jako první. Ostatní třídy stejného jména budou ignorovány.

Stejně tak může dojít k chybě, kdy rozdílné verze knihoven neobsahují třídy, které jsou využívány. Přestože je balík načtený na `CLASSPATH`, neobsahuje požadovanou třídu, je vyhozena vyjímka `ClassNotFoundException`.

1.1.2 Dynamické systémy

Dynamické systémy umožňují za běhu přidávat nebo odebírat jednotlivé komponenty. Jako příklad si vezmeme OSGi. OSGi umožňuje nasazení, odebrání, dokonce i výměnu komponenty za běhu systému. Vazby závislostí se tedy vytvářejí až v době nasazení komponenty do prostředí.

K tomuto úkolu se využívá tzv. `resolver`. Zodpovědností resolveru je vytvořit komponentě vazby s jejími závislostmi a opačně svázat tuto komponentu s komponentami, které jsou na ní závislé.

Resolveru k popisu vazby slouží metainformace uložené v komponentě v souboru `MANIFEST.MF`, odkud získá informaci `Bundle-Symbolic-Name`, která je jediná povinná, a `Bundle-Version`. Tyto informace slouží jako jednoznačné určení bundlu (komponenty) v běhovém prostředí OSGi. Zodpovědnost za kompatibilitu deklarovaných závislostí, jejím `Bundle-Symbolic-Name` a `Bundle-Version`, je přenesena na programátora.

Závislost v OSGi můžeme definovat mimo jiné na úrovni služeb, kde k identifikaci slouží rozhraní služby. Vazba služby a komponenty, která požádá o její registraci, probíhá již za běhu komponent. Opět bohužel poskytnuté informace o vazbě jsou jen jméno rozhraní služby. Nic neříkající o tom, co rozhraní poskytuje za metody a jaké jsou jejich signatury atp.

1.1.3 Shrnutí

Shrneme-li si opět prezentovaná fakta, dojdeme k závěru, že v disciplíně kontroly kompatibility se vyskytují minimálně dva faktory. A to kdy, v jaké fázi je vazba tvořena a jaké vazby je schopen nástroj zkontrolovat.

Jak z předchozího textu vyplývá překladač je jistě cenným nástrojem při prevenci chyb, zdaleka však není všespásným. S příchodem trendu, jakým je komponentové programování, vzniká bezesporu potřeba pro nástroj, který by byl schopen kontrolovat kompatibilitu vazeb u knihoven patřícího do tranzitivního uzávěru závislostí překládaného kódu či kontrolované komponenty a zároveň provádět tuto kontolu nejen při překladu, ale také při nasazení komponenty, či za jejícho běhu. Problémem však je, že v drtivém množství případů nevlastníme zdrojový kód využívaných knihoven. Což činí kontrolu kompatibility závislostí obtížnou, ne však nemožnou. OSGi například tento problém řeší metainformacemi uloženým

v souboru `MANIFEST.MF`, definicí jména, verze a zavedením sémantiky do verze. Takto je schopno OSGi rozhodnout, která verze splňuje kritéria kompatibility a která již ne. Bohužel se praxí ukázalo, že se sémantika verzování nedodržuje.

Půjdeme-li dále naskýtá se příležitost ke kontrole kompatibility komponent díky reverznímu inženýrství, které nám umožní znovu vytvoření signatur veřejných metod rozhraní, které by postačilo ke kontrole kompatibility.

1.2 Možné řešení problémů kompatibility

Hlavním úkolem je tedy odhalení veškerých závislostí propojující komponenty mezi sebou a to včetně transitivních a následné ověření kompatibility těchto vazeb. Jednou z možností jak přistoupit k řešení problému je statickou analýzou vytvořit reprezentaci vazeb závislostí jednotlivých volání a následně ověřit, zda vše co je volané je poskytované a zda je volání typově kompatibilní.

Tímto úkolem se zabývá nástroj vyvíjený na Zapadočeské univerzitě v Plzni nazvaný ve zkratce JaCC neboli Java Class Comparator [1]. Detailněji informace jsou uvedené v sekci 1.3.

1.2.1 Získání informací o vazbách

Jazyk Java nabízí hned dvě možnosti jak se tohoto úkolu zhostit.

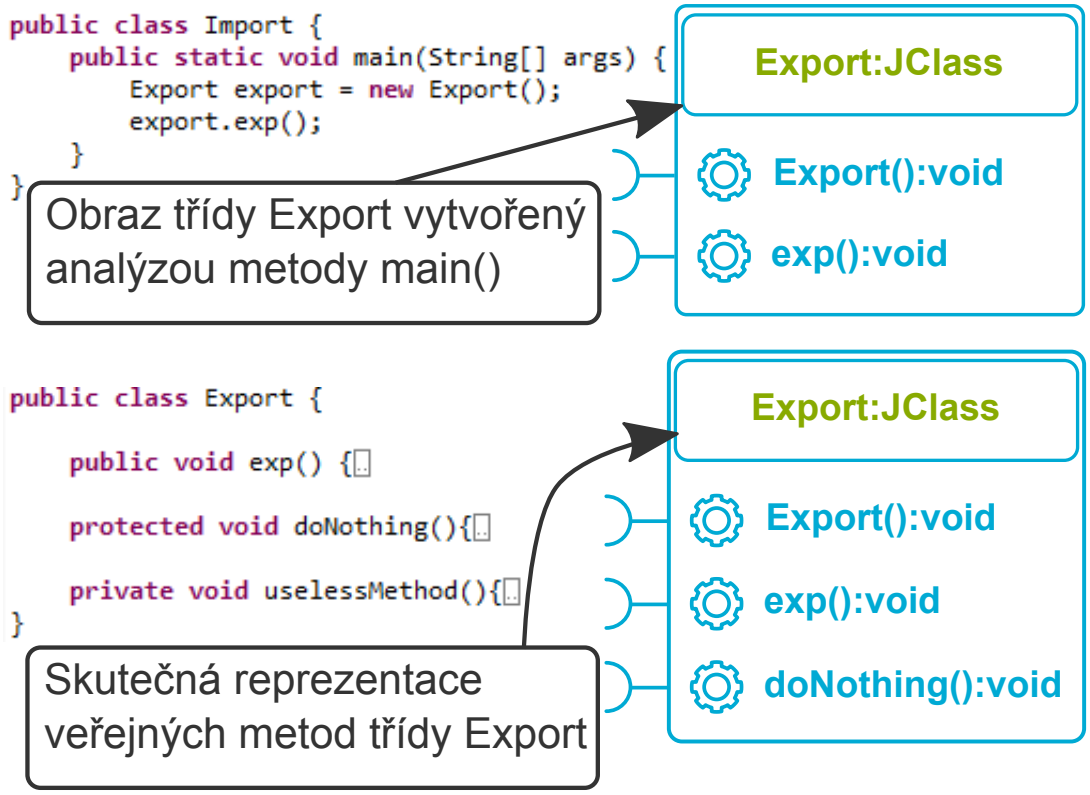
Prvním je Java Reflection API, které umožňuje mimo jiné, načtení informací jakými jsou jména metod, typy, počty a pořadí parametrů, návratové hodnoty, modifikátory přístupu, atributy a typy tříd. Avšak k tomu, aby bylo možné přistoupit k těmto informacím, je nutné, aby třída, o které chceme zjistit tyto informace, již byla načtena loaderem na `CLASSPATH`.

Druhým způsobem je analýza byte kódu. Překladače jakzyků jakým je např. Java nebo .NET jazyky, překládají zdrojový kód do platformě nezávislého kódu tzv. byte kódu. Tento byte kód je pak interpretován na konkrétní plaformě ve virtuálním stroji, který je již platformě závislý. Byte kód, jak již jméno napovídá, oplývá výhodami jak zdrojového kódu tak kódu binárního. Například u jazyka Java si byte kód uchovává strukturu originálního kódu, jména metod a hlavičky, zatím co jejich těla jsou přeložena do instrukcí pro virtuální stroj. Z byte kódu jsme tedy také schopni zjistit potřebné informace. Výhodou je, že není nutné načtení zkoumaného kódu na `CLASSPATH`.

Rekonstrukce informací o vazbách probíhá v principu pomocí statické analýzy. Bereme postupně třídu po třídě a procházíme těla jejích metod. Během průchodu tělem metody vytváříme obraz o tom, jaké metody a z jakých tříd jsou volané. Zároveň vytváříme obraz veřejných (public, protected) metod právě zkoumané

[1] http://www.assembla.com/spaces/jacc

třídy. Projdeme-li těla veškerých metod získáme obraz o tom, s jakými třídami tato třída interaguje a jaké metody na nich volá.



Obrázek 1.2: Znázornění tvorby informací o vazbách

1.2.2 Ověření kompatibility

Jakmile byly nalezeny veškeré vazby závislých komponent a byly o nich získány potřebné informace, můžeme přistoupit k ověření těchto vazeb.

Ke kontrole je potřeba mít seznam z okolí volaných metod náležících zkoumané třídě a seznam veřejných (public, protected) metod poskytovaných zkoumanou třídou. Položka seznamu zkoumané třídy pak obsahuje jméno metody a její signaturu tzn. návratový typ, počet a pořadí parametrů a jejich typ. Vazba je kompatibilní pokud se shoduje název metody, pořadí a počet parametrů a jsou kompatibilní typy všech parametrů a návratové hodnoty metody. Obecně typ je kompatibilní, pokud je shodný, nebo vyžadovaný typ je podtypem typu poskytnutého.

1.2.3 Vhodné fáze pro kontrolu kompatibility

Ke kontrole kompatibility závislostí jsou v základu vhodné 3 fáze.

První fází je již dříve zmiňovaný překlad. V této fázi je produkt ve stádiu vývoje a tak nehrozí žádné riziko při nalezení chyb v programu. Naopak nalezené chyby v této fázi se snadněji opravují, jelikož programátor má ještě kód „v“ hlavě, není třeba věnovat čas na porozumění kódu. Což má samozřejmě za následek, že chyby odstraněné v této fázi ušetří mnoho času a peněz, které by bylo potřeba vynaložit k nápravě chyb nalezených později.

Druhou fází jistě vhodnou pro kontrolu kompatibility je fáze nasazení (deploy). Pokud odhalíme chyby před nasazením do provozu, můžeme zabránit nepříjemnostem jako je nedostupnost služby, či její nesprávný běh. Pokud by například v dynamickém systému mělo dojít k výměně komponenty za komponentu s vyšší verzí (upgrade) a došlo k nalezení chyby, může systém výměnu zamítnout, ponechat tak komponentu stávající a nedošlo tak k žádnému problému. Odstranění nalezené chyby v této fázi je podstatně dražší, jelikož komponenta prošla například testováním a release procesem, které bude nutné zopakovat.

Třetí fáze je za běhu aplikace. Kompomenta je nasazená, jsou navázany její závislosti a dojde například k registraci služby, kterou komponenta poskytuje. Provedením kontroly kompatibility odhalíme problém v registraci služby a systém ji zamítne. Stále se relativně nic nestalo. Služba je sice nedostupná, ale pořád jde o lepší situaci než kdyby nekompatibilní služba způsobila pád aplikace.

Jistě by se našli ještě další fáze, které by se odvinuli od životního cyklu komponenty.

1.3 Související nástroje

1.3.1 JaCC

Název knihovny JaCC[2] je zkratkou pro Java Class Comparator. Jejím úkolem je zkonstruovat informace o typech a provést typové porovnání podle pravidel specifikace jazyku Java.

Knihovna je schopná pomocí rekonstrukce z byte kódu získat potřebné informace o vazbách mezi závislostmi. Tento úkol má na starost JaCC loader. Loaderu se jako vstup předá seznam Javovských balíčků JAR a loader pomocí nástroje ASM zrekonstruje jednotlivé třídy, provede statickou analýzu a jako výstup poskytne reprezentaci vazeb a informací o nich.

Další součást knihovny JaCC nazvaná comparator (komparátor) se stará o druhý úkol nutný při kontrole kompatibility a tím je ověření kompatibility vazeb.

[2] http://www.assembla.com/spaces/jacc

2. Návrh řešení

2.1 Požadavky na software

V předchozích částech tohoto dokumentu jsme se věnovali moderním trendům v programování a možným příčinám, které určují jejich směr. Zároveň jsme zaměřili pozornost na problémy, které s nimi nevyhnutelně souvisí. Především jsme se zabývali problematikou kompatibility veškerých vazeb mezi využívanými komponentami či knihovnami třetích stran. Přišli jsme na to, že dostupné nástroje v podobě překladače, jsou nedostačující, především v kontrole veškerých vazeb. Nastínili jsme i způsob, jak nedostatky při kontrole kompatibility všech vazeb mezi komponentami řešit. Jako vhodné řešení jsme zvolili rekonstrukci kódu pomocí reverzního inženýrství z byte kódu a nalezení vazeb statickou analýzou. Vzniká tedy potřeba mít nástroj, který by byl schopen využít této metody, vazby efektivně nalézt a ověřit jejich korektnost. Tento nástroj by pak informoval o nalezených chybách, za účelem jejich opravy.

Dostáváme se tak k definici konkrétních požadavků na takovýto nástroj. Následuje seznam požadavků na nástoj k ověření korektnosti vazeb mezi komponentami softwaru

1. Nalezení tranzitivního uzávěru závislých komponent systému.
2. Nalezení veškerých vazeb mezi komponentami.
3. Ověření kompatibility nalezených vazeb.
4. Identifikace chybějících závislostí.
5. Identifikace komponent obsahující konfliktní jména tříd.
6. Identifikace nepoužitých závislostí.
7. Poskytnutí informací o nalezených problémech.
8. Integrace nástroje do vývojového prostředí.

2.2 Návrh řešení jednotlivých požadavků

V této části textu se budeme věnovat detailům jednotlivých výše uvedených požadavků. Rozebereme je a zvolíme technologii k jejich realizaci.

2.2.1 Nalezení tranzitivního uzávěru závislých komponent systému

Abychom mohli program sestavit a podrobit analýze kompatibility, je nutné znát veškeré potřebné součásti. Tento úkol, ač na první pohled vypadá velmi jednoduše, se může ukázat jako velmi komplikovaná disciplína. Samozřejmě komponenty, které přímo využíváme v našem programu známe. Problém však nastává s informacemi o závislostech komponent, které využíváme.

Potřebujeme tedy vyřešit dva dílčí problémy a těmi jsou:

- získání informací o závislostech jednotlivých komponent,
- nalezení zdrojů, kde se tyto závislosti nacházejí.

Pro prostředí programovacího jazyku Java existuje několik nástrojů, které tyto problémy řeší. Mezi hlavní představitele patří Apache Maven [1], Apache Ivy [2] a Gradle [3]. Apache Maven řeší závislosti komponent deklarativní cestou, naproti tomu Apache Ivy přistupuje k této problematice spíše imperativně. Gradle je z výše zmiňovaných nejmladším, přesto však velmi nadějným nástrojem, který umožňuje oba přístupy.

Pro potřeby naší práce se budeme zabývat nástrojem Apache Maven. Tento nástroj byl vybrán pro svou stabilitu, zralost, především však pro jeho rozšířenost a velkou uživatelskou základnu.

Apache Maven řeší problém s informacemi o závislostech komponent pomocí souboru `pom.xml`, který obsahuje mimo jiné informace o všech závislostech dané komponenty. Informace o závislosti sestává ze tří údajů jednoznačně identifikujících komponentu.

Prvním z nich je `groupId`, které nese informaci o skupině komponent, do které daná komponenta patří. Zvykem je pojmenovávat `groupId` pomocí reverzního doménového jména např. `cz.slezacek.ccp3`, podobně jak je tomu například u pojmenovávání `package` v Javě.

Druhý identifikátor nese jméno `artifactId`, tento označuje jméno komponenty ve skupině. Zde je zvykem oddělovat jednotlivá slova pomocí pomlčky např. `compatibility-checking-plugin`.

Poslední identifikátor se nazývá `version`, nese informaci o vývojových stupních dané komponenty. Verzi je ve zvyku psát ve formátu:
`<major verze>.<minor verze>.<micro verze>-<kvalifikátor>`,
příkladem validní verze může být `1.2.1-SNAPSHOT` nebo jen `1.2.1`. Kde `major`, `minor` a `micro` jsou celá kladná čísla a `kvantifikátor` může obsahovat jakýkoli

[1] http://maven.apache.org/
[2] http://ant.apache.org/ivy/
[3] http://www.gradle.org/

řetězec. Platí pak pravidlo, že `1.2.1 > 1.2.1-SNAPSHOT`. Pozor porovnání těchto verzí je naopak v OSGi frameworku, kde naopak `1.2.1 < 1.2.1-SNAPSHOT`!

Příklad takového pom souboru můžeme vidět na výpisu 2.1. Výpis soboru byl zkrácen z důvodu stručnosti na definici dvou závislostí.

Na řádce číslo 1 pomocí tagu `<projekt>` definujeme nový Maven projekt. Tento projekt je identifikován pomocí tagů `<groupId>cz.slezacek.ccp3</groupId>` `<artifactId>pom-parent</artifactId><version>0.0.1</version>`. Sekcí uvozenou `<dependencyManagenemt>` definujeme část dokumentu týkající se závislostí projektu `pom-parent`. Na řádce číslo 13 pokračujeme výpisem seznamu závislostí tohoto projektu. Jak je vidět jednotlivé definice závislotí jsou uzavřené v `<dependency>` a sestávají z popisu závislosti pomocí `<groupId>`,`<artifactId>` a `<version>`.

Výpis 2.1: Soubor pom.xml definující závislosti projektu

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4      http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <modelVersion>4.0.0</modelVersion>
6
7      <groupId>cz.slezacek.ccp3</groupId>
8      <artifactId>pom-parent</artifactId>
9      <version>0.0.1</version>
10     <packaging>pom</packaging>
11
12     <dependencyManagement>
13         <dependencies>
14             <dependency>
15                 <groupId>cz.slezacek.ccp3</groupId>
16                 <artifactId>classpath-cache</artifactId>
17                 <version>0.0.1</version>
18                 <scope>compile</scope>
19             </dependency>
20             <dependency>
21                 <groupId>cz.slezacek.ccp3</groupId>
22                 <artifactId>analyzer</artifactId>
23                 <version>0.0.1</version>
24                 <scope>compile</scope>
25             </dependency>
26         </dependencies>
27     </dependencyManagement>
28 </project>
```

Z těchto informací tedy víme, že Maven projekt `cz.slezacek.ccp3.pom-parent-0.0.1`[4] má dvě závislosti a to `cz.slezacek.ccp3.classpath-cache-0.0.1` a `cz.slezacek.ccp3.analyzer-0.0.1`. Jelikož tyto závislosti jsou také Ma-

[4]Syntaxe pro zápis jména projektu <groupId>.<artifactId>-<version>

ven projekty, obsahují taktéž soubor `pom.xml`, kde jsou definovány jejich závislosti. Z těchto informací je Maven schopen zkonstruovat seznam veškerých závislostí projektu.

Nyní je nám známo, jak Maven získává a uchovává informace o závislostech jednotlivých komponent. Můžeme tedy přisoupit k druhé části a to nalezení zdrojů, kde se tyto závislosti nacházejí.

Jak jste si jistě mohli povšimnout, soubor `pom.xml` neobsahuje žádné informace o tom, kde dané komponenty hledat. Je tomu proto, že Maven k řešení tohoto problému používá centrální repozitář[5], kde se nachází nepřeberné množství komponent. Maven potom veškeré komponenty, na kterých je projekt závislý, postahuje do lokálního úložiště, které se obvykle nachází v adresáři `<user home>/.m2/repository/`.

Samozřejmé úložiště Mavenu nemůže obsahovat veškeré komponenty. Pokud se tak stane, Maven nabízí několik způsobů jak definovat úložiště vlastní. Jedním z nich je v souboru `settings.xml` obvykle se nacházejícím u distribuce Mavenu, nebo je možné ho „přepsat“ stejnojmenným souborem v `<user home>/.m2/`. Toto nastavení je pak platné pro veškeré projekty. Dalším způsobem je možné definovat úložiště přímo pro definovaný projekt. Definice se vkládá do souboru `pom.xml`, jak je ukázáno na výpisu 2.2. Pomocí tagu `<repositories>` uvozujeme seznam jednotlivých úložišť. V tagu `<repository>` je pak uvedené samotné úložiště. Úložiště je pak definováno jednoznačným identifikátorem `<id>`, dále je třeba adresa úložiště `<url>`. Měli bychom také k úložišti nastavit, zda je zde možné nalézt release verze nebo snapshot verze. Tag `<distributionManagement>` definuje úložiště, kam se budou komponenty publikovat. Detailnější informace lze nalézt v dokumentaci k Mavenu [6].

Maven pak postupuje při hledání závislostí následovně, nejprve se dotáže centrálního úložiště, zda obsahuje hledanou komponentu, pokud ne, dotazuje se postupně definovaných úložišť, až komponentu najde. Pokud ani tak komponentu nenajde skončí chybou.

Výpis 2.2: Soubor pom.xml definující úložiště komponent

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>cz.slezacek.ccp3</groupId>
    <artifactId>pom-parent</artifactId>
    <version>0.0.1</version>
    <packaging>pom</packaging>

```

[5] http://repo{1,2}.maven.org/, s webovým rozhraním na adrese http://search.maven.org/
[6] http://maven.apache.org/guides/, http://maven.apache.org/pom.html

```
...
<build>
    <extensions>
        <extension>
            <groupId>org.apache.maven.wagon</groupId>
            <artifactId>wagon-ssh</artifactId>
            <version>2.2</version>
        </extension>
    </extensions>
</build>

<distributionManagement>
    <repository>
        <id>ccp3</id>
        <name>ccp3 Internal Release Repository</name>
        <url>
        scp://eryx.zcu.cz/afs/zcu.cz/users/s/slezacek/public/repo
        </url>
        <layout>default</layout>
    </repository>
    <snapshotRepository>
        <id>ccp3_snapshot</id>
        <name>ccp3 Internal Snapshot Repository</name>
        <url>
        scp://eryx.zcu.cz/afs/zcu.cz/users/s/slezacek/public/repo
        </url>
        <layout>default</layout>
    </snapshotRepository>
</distributionManagement>

<repositories>
    <repository>
        <id>ccp3-public-release</id>
        <url>http://home.zcu.cz/~slezacek/repo</url>
        <releases>
            <enabled>true</enabled>
        </releases>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>ccp3-public-snapshot</id>
        <url>http://home.zcu.cz/~slezacek/repo</url>
        <releases>
            <enabled>false</enabled>
        </releases>
        <snapshots>
            <enabled>true</enabled>
```

```
60              </snapshots>
61          </repository>
62      </repositories>
63  </project>
```

2.2.2 Nalezení veškerých vazeb mezi komponentami

Teď když už víme jak nalézt veškeré potřebné komponenty a umíme je stáhnout, můžeme přistoupit k dalšímu kroku a tím je nalezení veškerých vazeb mezi komponentami. Vazba mezi komponentami vznikne, pokud třída z jedné komponenty volá metodu třídy, která náleží jiné komponentě.

Pokud bychom měli zdrojové kódy veškerých součástí, nebyl by takový problém, tyto vazby odhalit. Jednoduše (ve zjednodušeném pohledu) bychom analyzovali všechna těla metod. Zjistili bychom, co jednotlivé metody uvnitř svých těl volají. Jakmile bychom nalezli volání metody nenáležící třídě z právě zkoumané komponenty, nalezli jsme vazbu mezi komponentami.

Problém spočívá v tom, že v drtivé většině zdrojový kód nemáme. A není ani přítomen například ve fázi nasazení komponenty. V Javovském prostředí máme k dispozici jen přeložený kód v podobě byte kódu. Čelíme zde tedy dvěma problémům a to:

- jakým způsobem z byte kódu přečíst potřebné informace,
- jak nalézt všechny vazby.

Naštěstí v prvním problému nám velmi pomůže knihovna ASM[7]. ASM je víceúčelová knihovna pro manipulaci a analýzu byte kódu. Vyniká velmi snadnou použitelností a vysokou výkonností.

Právě tato knihovna je využita jako stavební kámen pro projekt JaCC[8]. Knihovna JaCC sestává z několika částí, z nichž právě část nazvaná `javatypes-loader` využívá knihovnu ASM. JaCC loader je schopen z byte kódu načíst reprezentaci tříd zkoumaných komponent. Jakmile máme třídy načtené, můžeme provést statickou analýzu. Statická analýza spočívá v průchodu všemi metodami tříd a analýze volaných metod unitř jejich těl. Takto loader defacto vytvoří pro každou třídu 2 reprezentace.

První reprezentace odpovídá načtené třídě a nese o ní informace o veškerých použitých typech, atributech, metodách, jejich jménech, signaturách, o importovaných třídách a o metodách, které jsou volané z jiných komponent. A reprezentaci druhou, která je obrazem vytvořeným z analýzy ostatních tříd. Reprezentace této

[7] http://asm.ow2.org/

[8] https://www.assembla.com/spaces/jacc

třídy nese informace o veškerých volaných metodách nad touto třídou z metod ostatních tříd. Jak je možné vidět na obrázku 1.2.

Loader je tedy schopen načíst jak, potřebné informace z byte kódu, tak nalézt veškeré vazby mezi třídami.

Tyto informace ukládá do vyrovnávací paměti. Data z této paměti jsou přístupná skrze API JaCC. Bohužel datová struktura nevyhovuje potřebám této práce, jelikož nepočítá s duplicitními záznamy. Této funkce je však zapotřebí z důvodu možnosti výskytu tříd s duplicitními jmény. Tím jest myšleno plně kvalifikované jméno třídy. Tento problém se reálně vyskytuje například při použití JPA knihoven a knihoven Hibernate.

Datová struktura počítající i s duplicitami je implementovaná jako součást této práce v části nazvané CCP3 `classpath-cache`.

2.2.3 Ověření kompatibility nalezených vazeb

V tomto bodu jsme schopni nalézt a obstarat veškeré závislosti projektu. Analyzovat tyto závislosti a nalézt u nich veškeré vazby. Známe tedy veškeré vazby, nic nám tedy nebrání v ověření koreknosti těchto vazeb. Víme, že každá třída je nyní v paměti reprezentována 2 krát. Jednou její reálnou reprezentací a podruhé jako soubor metod, které jsou z ní volány okolím.

Ověření korektnosti vazby spočívá v porovnání těchto dvou reprezentací. Zjištění, zda metody, které se objevují v reprezentaci volaných metod, jsou přítomné v reprezentaci reálné třídy a zda jsou shodné jejich jména a signatury.

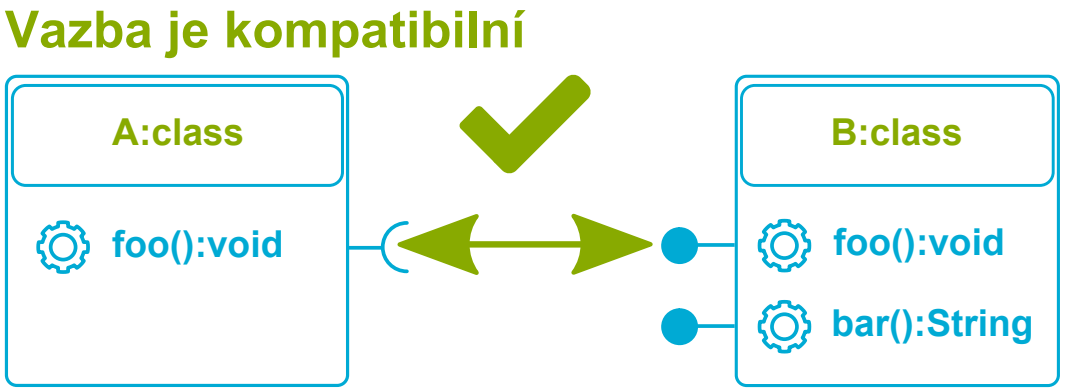
Disciplínou kontroly kompatibility vazeb se zabývá knihovna JaCC v části nazvané `javatypes-cmp`. Její součástí je i komparátor, který provádí kontrolu kompatibility. Tento projekt obsahuje potřebné nástroje pro kontrolu veškerých součástí jazyka Java, počínaje porovnáním primitivních datových typů, přes výčtové typy, metody, třídy, generiku a další.

Mějme tedy reprezentaci poskytovaných metod třídy A nazvaný A^{posk} a obraz z okolí volaných metod z třídy A nazvaný A^{volan}. Potom komparátoru jako vstup poskytneme A^{posk} a A^{volan} a jako výstup dostaneme výsledek jejich porovnání. Komparátor pracuje podle funkce $Diff : a \times b \rightarrow \{Non, Spec, Gen, Mut\}$. Kde výsledné hodnoty znamenají: *Non* – a a b jsou kompatibilní, *Spec* (Specialisation) – a je podtypem b, *Gen* (Generalisation) – a je rodičem b, *Mut* (Mutation) – a a b jsou rozdílné typy, nebo výsledek jejich vnitřních struktur je kombinací *Spec* a *Gen*. Pokud platí $a \in A^{posk}$ a $b \in A^{volan}$ pak $Diff(b, a) \in \{Non, Spec\}$ znamená kompatibilní vazbu. Pokud $Diff(b, a) \in \{Gen, Mut\}$ pak je vazba nekompatibilní.

V představě jaké možnosti mohou nastat nám napomohou obrázky 2.1, 2.2 a 2.3.

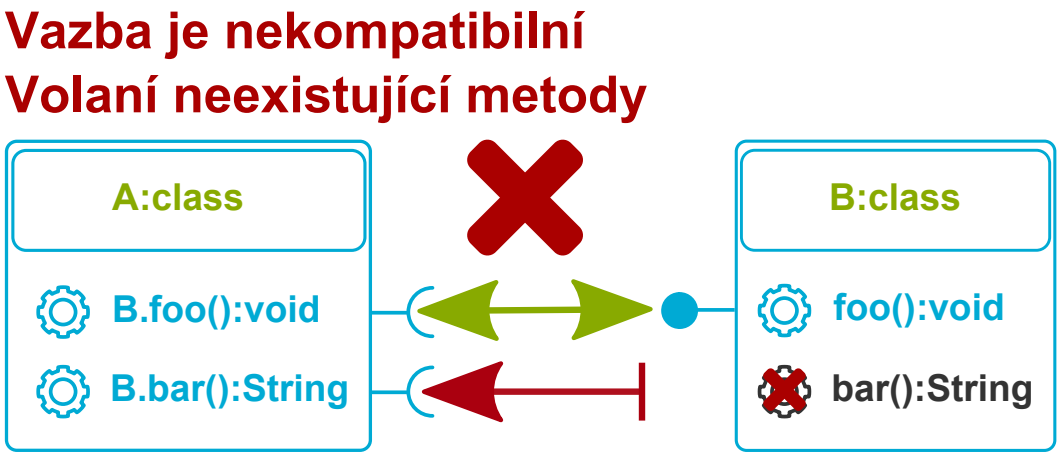
Jak se vidět na obrázku 2.1 ze třídy `A` se volá metoda `foo()` patřící třídě `B`. Jelikož třída `B` obsahuje metodu `foo()` a jejich signatury se shodují je vazba

kompatibilní.



Obrázek 2.1: Znázornění kompatibilní vazby

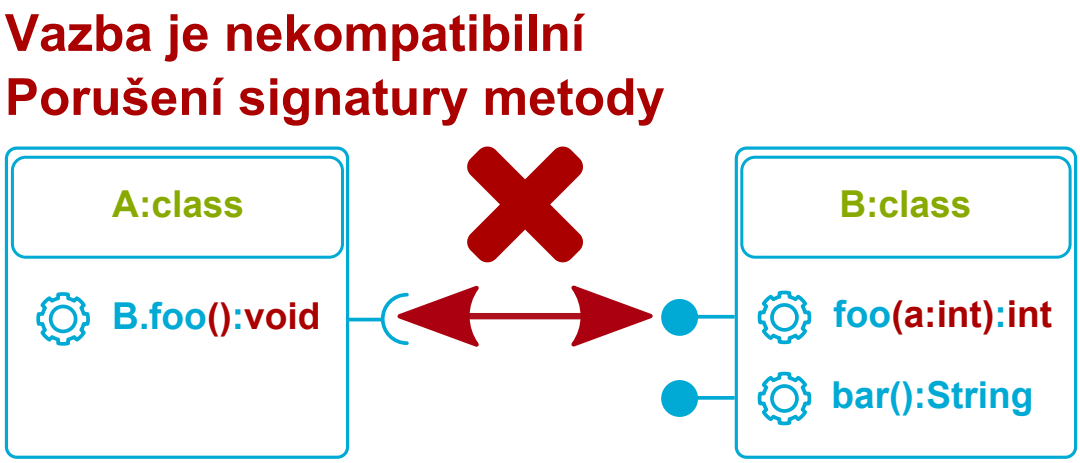
Na obrázku 2.2 můžeme vidět, že třída `A` opět volá metodu `foo()` a volá ještě metodu `bar()`, obě ze třídy `B`. Třída `B` však neobsahuje metodu `bar()`, třída `A` tedy volá neexistující metodu, vazba je nekompatibilní.



Obrázek 2.2: Znázornění nekompatibilní vazby chybějící volaná metoda

Poslední případ zobrazený na obrázku 2.3 popisuje situaci, kdy ze třídy `A` je volána metoda `foo():void`. Třída `B` však obsahuje metodu `foo(a:int):int`. Je zde vidět, jak jde porušit signaturu hned několika způsoby. Jednak nesprávným počtem parametrů, jejich neshodným typem, špatným pořadím paramerů a rozdílným

typem návratové hodnoty funkce. Vazba je tedy nekompatibilní. V podstatě by se tato situace klasifikovala jako volání chybějící metody ve třídě `B`, pro názornost byly tyto případy odděleny.



Obrázek 2.3: Znázornění nekompatibilní vazby porušení signatury metody

2.2.4 Identifikace chybějících závislostí

Identifikace chybějících závislostí je se znalostmi, které již máme, jednoduchou disciplínou. Přesto však velmi užitečnou. Problémy s chybějícími závislostmi jsou velmi časté. Často jde o chyby způsobené nepozorností, ať je však chyba zapříčiněná jakoukoliv banalitou, důsledky jsou stejně fatální. Problémem může být již třeba použití jiné verze komponenty, nebo dokonce záludnější chyba jiná verze JRE [9].

Přejděme tedy k problému detekce chybějících závislostí. Víme, že u každé třídy se pomocí JaCC loderu vytvoří 2 reprezentace. Tyto reprezentace se pak předávají JaCC komparátoru. Pokud však jde o chybějící závislost, nemáme jednu z reprezentací, konkrétně reprezentaci reálné třídy.

Řešení detekce chybějících závislostí je předmětem této práce. Konkrétně je za ni zodpovědná část nazvaná CCP3 `analyzer`. K detekci dochází při analýze tříd načtených do paměti.

[9] Java Runtime Environment

2.2.5 Identifikace komponent obsahující konfliktní jména tříd

Dalším dílčím úkolem je identifikovat komponenty obsahující stejnojmenné třídy (včetně jména balíku). K tomu, abychom hlouběji pochopili, proč představují stejnojmenné třídy problém, potřebujeme získat základní informace o class loaderech Javy. Pro jednoduchost vezmeme v úvahu pouze class loader Javy SE[10].

Class loader je nástroj, který načítá byte kód a reprezentuje ho v paměti JVM[11] tak, aby bylo možné načtené třídy využívat. Java SE má 3 základní class loadery:

- Bootstrap class loader,
- Extension class loader a
- Classpath class loader.

Tyto class loadery jsou zřetězeny, jak je zobrazeno na obrázku 2.4. Ať už se načítá jakákoli třída, je vždy dotazován System class loader, který deleguje dotaz na Extenstion class loader a ten zase na Bootstrap class loader. Bootstrap loader buď třídu najde a vratí Extension loaderu jeji referenci, nebo ji nenajde. V tom případě hledá třídu Extention class loader, situace je stejná, najde-li třídu, vrátí referenci, nebo postoupí dotaz System class loaderu. System classloader postupuje naprosto stejně, najde-li třídu, vrátí referenci, pokud ne, vyhodí vyjímku. (Případně je možné definovat uživatelské class loadery, ty se opět řetězí, ale až za System class loader.) Problém je však v implementaci samotné `CLASSPATH` Ta je implementována jako plochá hash tabulka.

Z toho vyplývá omezení, že pouze jedna třída se jménem a.b.c.X může být na `CLASSPATH` načtena. Když na `CLASSPATH` dáme dva balíčky jar obsahující každý třídu a.b.c.X, nastane problém. Načtena bude pouze třída z balíčku, který bude uveden dříve za parametrem příkazu `$ java -classpath`. Pak si již class loader „myslí“, že danou třídu již načetl a přeskočí ji.

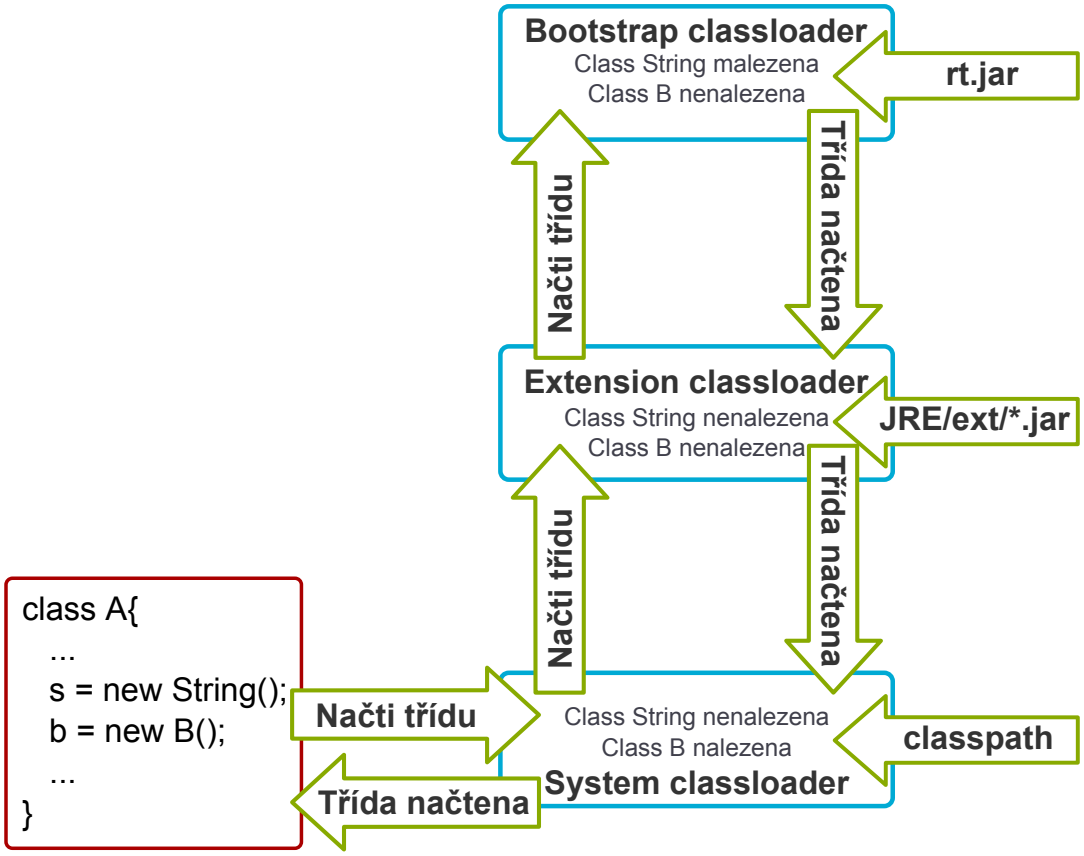
Nastane-li případ, že by náš program potřeboval jako závislost knihovnu a.b.c-2.0.0 a zároveň využíval knihovnu Y, která by jako závislost měla a.b.c-1.0.0, mohli bychom narazit na problém s nekompatibilitou.

Tato práce se snaží tento problém řešit preventivně tím, že o těchto problémech informuje. Nalezení tříd s konfliktními jmény je ve výsledku jednoduché, je jen nutné mít navrženou datovou strukturu tak, aby počítala s touto skutečností. Datová struktura pro uchování těchto reprezentací je obsažená v části této práce nazvané CCP3 `classpath-cache`.

[10] Standard Edition
[11] Java Virtual Machine

Je definována následovně `Hashtable<String, List<JClass>>`. Kde klíčem k hash tabulce je plně kvalifikované jméno třídy. Jako hodnota se přidávají reprezentace tříd v podobě JClass do seznamu. Detekce konflikních jmen tříd tedy spočívá v kontrole, zda seznam jako hodnota k danému klíči obsahuje více než 1 prvek.

Řešením problému detekce konfliktních jmen je implementováno v části práce nazvané CCP3 `analyzer`.



Obrázek 2.4: Znázornění načítání třídy clasloadery

2.2.6 Identifikace nepoužitých závislostí

Někdy pracujeme s projekty tak rozsáhlými, že je již velmi pracné zjistit, zda je daná komponenta opravdu využívána či nikoli.

Jelikož již umíme nalézt veškeré vazby mezi komponentami, stačí identifikovat komponenty, do kterých nevedou žádné vazby. Žádná třída z nich nevolá žádnou

metodu. Tyto komponenty pak představují nevyužívané komponenty.

Řešení problému identifikace nepoužitých závislostí je implementováno v části této práce nazvané CCP3 `analyzer`.

2.2.7 Poskytnutí informací o nalezených problémech

Program poskytuje informace ve formě textového výstupu. Formát tohoto výstupu byl přebrán z projektu APICC[12], který je vyvíjen na katedře informatiky na Zapadočeké univerzitě. Byl použit s laskavým svolením autora. Tento formát byl zvolen hlavně z důvodu dodržení konzistence výstupů z nástrojů využívající JaCC.

2.3 Použité algoritmy

V této části textu nám již jsou známé podrobnosti o požadavcích na tento software a jejich řešení. Můžeme se tedy pustit do popisu algoritmu, který je použit k celkové analýze.

2.3.1 Popis algoritmu analýzy

Algoritmus využívá datovou strukturu definovanou v projektu CCP3 `classpath-cache` rozhraním `cz.slezacek.ccp3.classpath_cache.entity.ClassWithOriginCache`.[13] Datová struktura dědí od rozhraní `java.util.Map<String, List<JClass>>` a pro zjednodušení vkládání dat do ní, definuje metody:

- `public abstract List<JClass> put(String key, JClass jclass)`,
- `public abstract List<JClass> put(String key, List<JClass> jclass)`.

V této datové struktuře jsou uložené, reprezentace veškerých poskytnutých tříd ke kontrole kompatibility. Jak již bylo zmíněno výše, struktura je organizovaná jako hash tabulka, kde

klíč hash tabulky
: představuje **plně kvalifikované jméno třídy**

hodnotu hash tabulky
: představuje **seznam reprezentací tříd** v podobě `JClass`.

[12] Application programming interface compatibility checker

[13] Název třídy pochází z rané fáze diplomové práce. V té době nástroj JaCC neuměl poskytnout informaci o tom z jakého balíku třída pochází. Tato datová struktura odstraňovala tento nedostatek. Po konzultaci bylo rozhodnuto převést funkčnost do knihovny JaCC. Přínos této třídy se tímto zúžil jen na schopnost ukládat více tříd se stejným jménem.

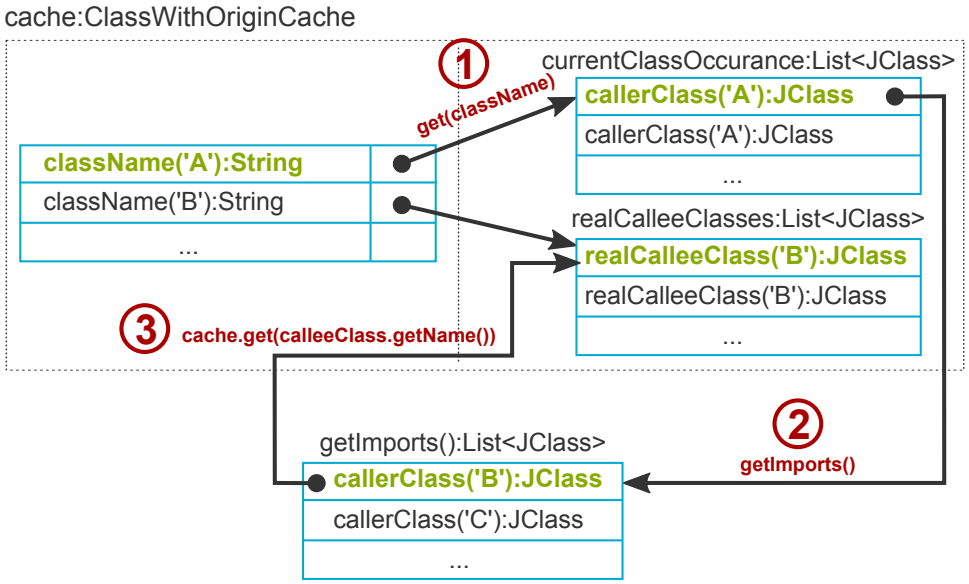
Veškeré operace analýzy operují nad daty z této datové struktury. Obrázek 2.5 znázorňuje algoritmus průchodu datavou strukturou a vyhledání dvojice reprezentací třídy, reprezentaci reálné třídy a reprezentaci volaných metod této třídy z okolí. Algoritmus spočívá v průchodu celou hash tabulkou.

A to tak, že vezmeme klíč hash tabulky najdeme pro něj hodnotu v podobě seznamu reprezentací tříd, tak jak je znázorněno na Obrázku 2.5 v kroku číslo 1. Dále procházíme seznam reprezentací tříd.

Jak znázorňuje Obrázek 2.5 v kroku číslo 2. Vezmeme si záznam ze seznamu, představující volající třídu. K této třídě získáme seznam tříd, které importuje. Procházíme seznam importovaných tříd.

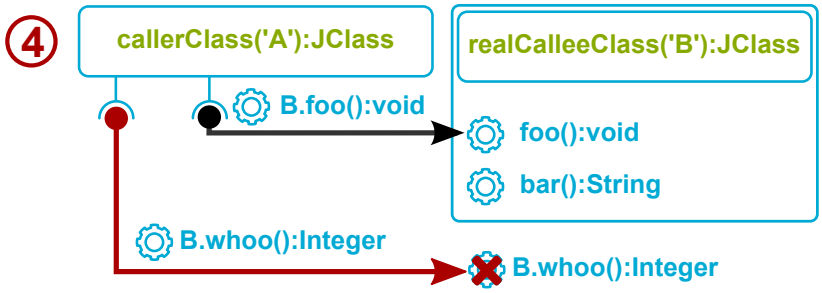
Jak je znázorněno na Obrázeku 2.5 v kroku číslo 3. Vezmeme si záznam ze seznamu importovaných tříd a podle jména třídy najdeme v hash tabulce seznam reprezentací této třídy.

Obrázek 2.6 zachycuje krok algoritmu číslo 4. V tomto kroku již máme dvojici reprezentací třídy ke kontrole kompatibility. Jak je znázorněno na Obrázku 2.6 provede se kontrola kompatibility.



Obrázek 2.5: Algoritmus analýzy

V příloze je pak možno najít Obrázek 1, na kterém je zachycen vývojový diagram celého algoritmu analýzy včetně detekce nekompatibilit, konfliktních jmen tříd, chybějících závislostí a nevyužitých závislostí.



Obrázek 2.6: Kontrola kompatibility

2.3.2 Rozdělení na části podle zodpovědnosti

Program je implementovám jako Apache Maven plugin a je rozdělen na 3 části.

Část nazvaná CCP3 `classpath-cache` obsahuje datavou strukturu vyrovnávací paměti, do které jsou načteny reprezentace tříd. Dále tato část obsahuje třídu schopnou načíst třídy z poskytnutého seznamu balíčků `*.jar` do výše zmíněné datové struktury. Načítání je provedeno využitím JaCC loaderu.

Další část je zodpovědná za analýzu dat uložených v datové struktuře. Tato část je nazvána CCP3 `analyzer`. Součástí analýzy je detekce nekompatibilních vazeb mezi komponentami, odhalení chybějících využívaných tříd, nalezení tříd mající konfliktní jména, nalezení závislostí, které nejsou využívány a uložení výsledků analýzy. Analyzer je schopen generovat graf, kde jako uzly jsou jednotlivé komponenty a hrany představují třídy, mezi kterými jsou vazby.

Poslední část představuje samotný Maven plugin, který je možné integrovat do vývojového nástroje. Nese název CCP3 `compatibility-checking-plugin`. Tento plugin plní úkol nalezení veškerých závislostí projektu a jejich předání `analyzeru`. Po provedení analýzy, je zodpovědný za informování o výsledcích analýzy.

2.4 Uživatelská dokumentace

Program CCP3 `compatibility-checking-plugin` je použitelný jako plugin Apache Maven. Slouží ke kontrole kompatibility vazeb mezi komponentami projektů.

Jeho použití je velmi snadné. Systém Maven využívá řadu fází životního cyklu sestavení. Následuje seznam s vysvětlením jednotlivých fázi:

validate
: ověří zda je projekt v pořádku a veškeré potřebé informace jsou dostupné

compile
: přeloží zdrojový kód projektu

test
: otestuje přeložený kód za použití vhodného testovacího frameworku

package
: vezme přeložený kód a zabalí jej do distribuovatelné podoby, jakým je například JAR

integration-test
: zpracuje a nasadí balíček, pokud je potřeba, do prostředí kde se spouští integrační testy

verify
: spustí jakékoli kontroly, že je balíček v pořádku a odpovídá kritérijím kvality

install
: nainstaluje balíček do lokálního úložiště, aby jej ostatní projekty mohly využít jako závislost

deploy
: nasazení do integračního nebo produkčního prostředí, zkopíruje finální balíček do vzdáleného úložiště pro sdílení s ostatními vývojáři a projekty

Plugin sám se váže na fázi `verify` Maven systému, ve které probíhají kontroly. Jakmile se kontrolovaný projekt dostane do této fáze, spustí se `compatibility-checking-plugin` a zkontroluje kompatibilitu vazeb. Jakmile proběhne kontrola plugin vypíše výsledky a je-li vše v pořádku pokračuje se dále. Pokud byly nalezeny chyby při kontrole, jsou vypsány a sestavování skončí chybou.

Jak tedy přidat tento plugin k projektu? Stačí jen přidat jeho spuštění do souboru `pom.xml`. Kód, který musíme přidat zachycuje výpis 2.3. Tento kód říká, budeme používat Maven plugin `cz.slezacek.ccp3.compatibility-checking-plugin-0.0.1`. Z tohoto pluginu zavolej `goal` označený jménem `check`.

V tagu `<configuration>` je pak konfigurace parametrů pro plugin. Parametry pluginu:

loadJavaHomeLib : `boolean`
: příznak zda má plugin načíst rt.jar[14]

excludeRtJarFromCheck : `boolean`
: příznak vyřadí třídy pocházející z rt.jar z veškerých kontrol[15]

[14]Plugin je připraven k načítání veškerých jarů z JRE. Jelikož je však načítání velkých jarů z JRE náročná operace, načítá plugin v této verzi jen rt.jar

[15]jelikož oprava chyb kompatibility v rt.jar nejsou v kompetenci programátora projektu

excludedJars : `Set<String>`
: seznam jmen souborů u nichž mají být vynechány veškeré kontroly. Stačí uvést část názvu. (<param>rt.jar</param><param>junit</param>)

generateGraphs : `boolean`
: příznak zapne / vypne generování grafů do souborů `incompatibilities-graph.graphml` a `jar-deps-unigraph.graphml`

Výpis 2.3: Kód nutný ke spuštění pluginu

```
<build>
    <plugins>
        <plugin>
            <groupId>cz.slezacek.ccp3</groupId>
            <artifactId>compatibility-checking-plugin</artifactId>
            <version>0.0.1</version>
            <executions>
                <execution>
                    <goals>
                        <goal>check</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <loadJavaHomeLib>true</loadJavaHomeLib>
                <excludeRtJarFromCheck>true</excludeRtJarFromCheck>
                <excludedJars></excludedJars>
                <generateGraph>false</generateGraph>
            </configuration>
        </plugin>
    </plugins>
</build>
```

2.4.1 Spuštění kontroly kompatibility

Spustit kontrolu kompatibility, pokud máme v `pom.xml` uvedený CCP3 plugin, je velmi snadné. Příkaz je zobrazen na výpisu 2.4.

Výpis 2.4: Příkaz pro spuštění kontroly kompatibility

```
$ mvn verify
```

Jelikož je plugin navázán na spuštění ve fázi `verify` a fáze `install` následuje až za touto fází, je stějně tak možné spustit plugin příkazem na výpise 2.5 nebo jakýmkoli s fází následující po `verify`.

Výpis 2.5: Příkaz pro spuštění kontroly kompatibility

```
$ mvn install
```

2.4.2 Výsledky poskytnuté pluginem

Výsledkem pluginu po provedení kontroly je výpis výsledků. Proběhlo-li vše v pořádku a nebyly nalezeny žádné nekompatibility ani jiné problémy, plugin vypíše hlášku zachycenou na výpisu 2.6.

Výpis 2.6: Výpis pluginu, nebyly nalezené problémy

```
[INFO] ----------------------------
[INFO] -- No unused imports found --
[INFO] ----------------------------
[INFO]
[INFO] ------------------------------------
[INFO] -- No conficting class names found --
[INFO] ------------------------------------
[INFO]
[INFO] ----------------------------
[INFO] -- No missing imports found --
[INFO] ----------------------------
[INFO]
[INFO] --------------------------------
[INFO] -- All bundles are compatible --
[INFO] --------------------------------
```

2.4.3 Nevyužité závislosti

Byla-li nalezena nějaká nevyužitá závislost, plugin o ní informuje tak, jak je zachyceno na výpisu 2.7. Jak je z výpisu zřejmé v tomto případě je v závislostech uveden `junit-4.8.1.jar`, ale nepoužívá se.

Výpis 2.7: Výpis pluginu s nalezenou nevyužitou závislostí

```
[WARNING] -------------------------
[WARNING] -- Unused imports found --
[WARNING] -------------------------
[WARNING]   junit-4.8.1.jar
[WARNING]
```

2.4.4 Konfliktní jména tříd

Pokud byly nalezené v projektu stejnojménné třídy z různých balíků, plugin o nich informuje v podobě zachycené na výpisu 2.8. K tomuto problému dochází velmi často. Velmi častým jevem je poskytování třídy se stejným jménem, dvěma různými balíky. Jak je z výpisu patrné, je možné, že je třída obsažena ve více než dvou balících.

Výpis 2.8: Výpis pluginu s nalezenými konfliktními jmény tříd

```
[ERROR] ---------------------------------
[ERROR] -- Conficting class names found --
[ERROR] ---------------------------------
[ERROR] Class name: org.apache.commons.logging.impl.
            WeakHashtable$Referenced
[ERROR] Found in:
[ERROR]     commons-logging-1.1.1.jar
[ERROR]     commons-logging-api-1.1.jar
[ERROR] ---------------------------------
[ERROR] Class name: javax.persistence.ElementCollection
[ERROR] Found in:
[ERROR]     javaee-web-api-6.0.jar
[ERROR]     hibernate-jpa-2.0-api-1.0.0.Final.jar
[ERROR] ---------------------------------
[ERROR] Class name: org.apache.commons.logging.impl.SimpleLog
[ERROR] Found in:
[ERROR]     commons-logging-1.1.1.jar
[ERROR]     commons-logging-api-1.1.jar
[ERROR]     jcl-over-slf4j-1.6.4.jar
[ERROR] ---------------------------------
```

2.4.5 Chybějící závislosti

Plugin odhaluje i chybějící závislosti. Pokud odhalí takovýto problém, informuje o něm výpisem zachyceným ve výpisu 2.9.

Výpis 2.9: Výpis pluginu s nalezenými chybějícími závislostmi

```
[ERROR] --------------------------
[ERROR] -- Missing imports found --
[ERROR] --------------------------
[ERROR] Missing classes:
[ERROR]     org.joda.time.DateTime
[ERROR]         Imported by:
[ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
        .format.datetime.joda.JodaTimeConverters$
        CalendarToReadableInstantConverter
[ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
        .format.datetime.joda.JodaTimeConverters$
        DateTimeToLongConverter
[ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
        .format.datetime.joda.JodaTimeConverters$
        DateTimeToDateMidnightConverter
[ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
        .format.datetime.joda.JodaTimeConverters$
        DateTimeToInstantConverter
[ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
```

```
20          .format.datetime.joda.JodaTimeConverters$
21          DateTimeToMutableDateTimeConverter
22  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
23          .format.datetime.joda.DateTimeParser
24  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
25          .format.datetime.joda.JodaTimeConverters$
26          DateTimeToLocalTimeConverter
27  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
28          .format.datetime.joda.
29          JodaDateTimeFormatAnnotationFormatterFactory
30  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
31          .format.datetime.joda.JodaTimeConverters$
32          DateTimeToCalendarConverter
33  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
34          .format.datetime.joda.JodaTimeConverters$
35          DateTimeToLocalDateConverter
36  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
37          .format.datetime.joda.JodaTimeConverters$
38          DateTimeToDateConverter
39  [ERROR]          spring-context-3.1.1.RELEASE.jar:org.springframework
40          .format.datetime.joda.JodaTimeConverters$
41          DateTimeToLocalDateTimeConverter
```

2.4.6 Nekompatibilní vazby

Pokud plugin nalezne v pojektu nekompatibilní vazby, informuje o nich v podobě zachycené na výpisu 2.10. Zde byla nalezená nekompatibilní vazba mezi balíčky `cglib-nodep-2.1.jar` a `spring-context-3.1.1.RELEASE.jar`. Problém nastal ve třídě `org.springframework.context.annotation.ConfigurationClass--Enhancer$DisposableBeanMethodInterceptor`, která volá metodu třídy `net.sf.--cglib.proxy.Enhancer`, která nese jméno `registerStaticCallbacks()` a má 2 parametry. Tato metoda nebyla ve třídě `net.sf.cglib.proxy.Enhancer` nalezena.

Výpis 2.10: Výpis pluginu s nalezenými nekompatibilními vazbami

```
1  [ERROR] ----------------------------
2  [ERROR] -- Incompatibilities found --
3  [ERROR] ----------------------------
4  [ERROR] Filename: cglib-nodep-2.1.jar
5  [ERROR]  Class: net.sf.cglib.proxy.Enhancer
6  [ERROR]   Methods: 9 elements x 43 elements
7  [ERROR]    Method: registerStaticCallbacks(2) (Not found)
8  [ERROR]    Imported by:
9  [ERROR]    Class: org.springframework.context.annotation.
10         ConfigurationClassEnhancer$DisposableBeanMethodInterceptor
11 [ERROR]    File: spring-context-3.1.1.RELEASE.jar
12 [ERROR]
```

2.5 Omezení a známé problémy nástroje

Nástroj s sebou nese jistá omezení.

První omezení vyplývá z metody, která je použita při hledání vazeb mezi komponentami. Statická analýza je schopná odhalit veškeré statické vazby. Bohužel nebude úspěšná u vazeb, které jsou vytvářeny za běhu dynamicky. Uvedeme si pár příkladů, u kterých bude metoda statické analýzy v podobě, jak ji implementuje JaCC neúspěšná. JaCC není schopné odhalit vazby vytvořené pomocí reflexe. Stejně tak bude neúspěsná, v případě vazeb vytvořených pomocí Expression Language, nebo vazeb vzniklých parsováním XML.

Druhým známým omezením je, že JaCC zatím nedokáže doplnit do reprezentace třídy metody, které třída dědí od svého rodiče, který náleží do jiného balíku. Z tohoto důvodu, je zatím velké množství nalezených chyb, falešným poplachem, způsobeným právě absencí této funkce. Na opravě tohoto nedostatku se již pracuje.

Dalším problémem je načítání závislostí ve tvaru `WAR` má JaCC loader problém se správným pojmenováním tříd. Jelikož je u WAR archivu jiné zanoření než u JAR archivu. Konkrétně u WAR archivu jsou třídy v adresáři `WEB-INF/classes/`. Tudíž například třída namísto jména `cz.hikersbook.Foo` dostane jméno `WEB-INF.classes.--cz.hikersbook.Foo`. Načež nedojde k jejímu nalezení v datové struktuře a je zařazena mezi chybějící závislosti.

Známým problémem u CCP3 je spuštění kontroly nad parent pom projektem, nebo nad reactor pom projektem. Reactor pom totiž obsahuje několik Maven projektů, které spolu tvoří celek. Cílem tedy bylo kontrolu spustit pouze jednou na konci, tedy při sestavování reactor pom projektu. Bohužel již zde byl problém v podpoře ze strany Mavenu. Tato funkcionalita je vyvíjena v době tvorby této práce. Existuje způsob (workaround) jak dosáhnout spustění jen jednou a to na konci. Bohužel při použití tohoto workaroundu, Maven sice poskytne seznam všech projektů, v podobě `List<MavenProject>`. Bohužel jednotlivé instance tohoto projektu vracejí seznam jejich závislostí prázdný. Přesné volání MavenProject.getArtifacts(), které by mělo podle dokumentace Mavenu, vracet tranzitivní uzávěr závislostí projektu, vrací prázdný seznam. Z tohoto důvodu, není možné spustit kontrolu nad všemi projekty, spustěním sestavení pouze na reactor pom projektu. Je však možné spustit kontrolu nad jednotlivými projekty. Definici CCP3 pluginu pak stačí uvést pouze do parent pomu.

3. Případová studie

K ověření funkčnosti CCP3 pluginu byla vybrána reálná aplikace nazvaná **hikersbook**. Aplikace plní funkci zápisníku výletů po zajímavých místech světa. Jedná se o webovou třívrstvou aplikaci v Java EE [1]. Datvovou vrstvu reprezentuje nosql databáze OrientDB, prezentační vrstva využívá šablonovací systém FreeMarker a JavaScriptovou knihovnu JQuery. Součástí aplice je také Spring IoC container.

Aplikaci tvoří přibližně 50 tříd a cca stejné množství knihoven třetích stran poskytující Spring, přístup k OrientDB databázi, logování, šablonovací systém a jiné. K sestavování aplikace je použit Maven.

3.1 Nalezené nekompatibility

Jak již bylo řečeno aplikace využívá Spring frameworku ve verzi 3.1.1. Tato verze je závislá na knihovně CGlib-nodep, kterou využívá pro svou bezchybnou funkci. Přestože Maven je schopen stáhnout tranzitivní uzávěr závislostí, Spring vývojáři nenadefinovali tuto knihovnu jako závislost, proto ji každá klientská aplikace musí explicitně uvést. V této aplikaci byla použita knihovna CGlib-nodep ve verzi 2.1. Což způsobovalo chybu za běhu zachycenou na výpisu 3.1.

Výpis 3.1: Výpis vyjímky

```
1 Caused by: java.lang.NoSuchMethodError:
2 net.sf.cglib.proxy.Enhancer.
3 registerStaticCallbacks(Ljava/lang/Class;[
4 Lnet/sf/cglib/proxy/Callback;)V
```

Po spustění kontroly CCP3 pluginem, byla mimo jiných tato chyba nalezena a vývojář tak na ni byl upozorněn. Výpisem 3.2 následuje zkrácený výpis chybové hlášky poskytnuté CCP3 pluginem.

Výpis 3.2: Výpis nalezených nekompatibilit

```
1  [ERROR] ----------------------------
2  [ERROR] -- Incompatibilities found --
3  [ERROR] ----------------------------
4  ...
5  [ERROR] Filename: cglib-nodep-2.1.jar
6  [ERROR]  Class: net.sf.cglib.proxy.Enhancer
7  [ERROR]   Methods: 9 elements x 43 elements
8  [ERROR]    Method: registerStaticCallbacks(2) (Not found)
9  [ERROR]    Imported by:
10 [ERROR]    Class: org.springframework.context.annotation.
```

[1] Java Enterprise Edition

```
11          ConfigurationClassEnhancer$DisposableBeanMethodInterceptor
12 [ERROR]    File: spring-context-3.1.1.RELEASE.jar
13 ...
```

Díky odhalení této chyby jsme předešli pádu aplikace za běhu systému. Chyby tohoto charakteru se velmi obtížně ladí.

3.2 Nalezené konfliktní třídy

Jak jsme se již zmiňovali v teoretické a praktické části této publikace, plugin je schopen odhalit problémové třídy. Tedy stejnojmenné třídy patřící různým balíkům. Jak můžeme vidět na výpisu 3.3 i tyto byly úspěšně nalezeny.

Výpis 3.3: Výpis nalezených tříd s konfliktními jmény

```
1  [ERROR] ---------------------------------
2  [ERROR] -- Conficting class names found --
3  [ERROR] ---------------------------------
4  [ERROR] Class name: javax.persistence.Access
5  [ERROR] Found in:
6  [ERROR]     javaee-web-api-6.0.jar
7  [ERROR]     hibernate-jpa-2.0-api-1.0.0.Final.jar
8  [ERROR] ---------------------------------
9  [ERROR] Class name: javax.persistence.ElementCollection
10 [ERROR] Found in:
11 [ERROR]     javaee-web-api-6.0.jar
12 [ERROR]     hibernate-jpa-2.0-api-1.0.0.Final.jar
13 [ERROR] ---------------------------------
14 ...
```

Nalezením těchto problémů můžeme předejít náhodně se objevujícím problémům s třídami stejného názvu, ale rozdílného API. Není jednoznačně definováno, která třída bude použita. Často to závisí na implementaci daného Java systému. Vzhledem k náhodnosti vzniku problému je extrémě obtížné tyto chyby odstranit!

3.3 Chybějící závislosti

Stějně tak se úspěšně podařilo nalézt chybějící závislosti aplikace. Nalezené výsledky jsou zobrazeny na zkráceném výpise 3.4.

Výpis 3.4: Výpis nalezených chybějících závislostí

```
1 [ERROR] --------------------------
2 [ERROR] -- Missing imports found --
3 [ERROR] --------------------------
4 [ERROR] Missing classes:
5 [ERROR]     edu.emory.mathcs.backport.java.util.concurrent.
```

```
6       ThreadFactory
7   [ERROR]          Imported by:
8   [ERROR]           spring-context-3.1.1.RELEASE.jar:org.
9       springframework.scheduling.backportconcurrent.
10      CustomizableThreadFactory
11  [ERROR]           spring-context-3.1.1.RELEASE.jar:org.
12      springframework.scheduling.backportconcurrent.
13      ThreadPoolTaskExecutor
14  [ERROR]     org.apache.commons.httpclient.methods.
15      EntityEnclosingMethod
16  [ERROR]          Imported by:
17  [ERROR]           spring-web-3.1.1.RELEASE.jar:org.
18      springframework.http.client.CommonsClientHttpRequest
19  [ERROR]     org.apache.xml.utils.PrefixResolver
20  [ERROR]          Imported by:
21  [ERROR]           freemarker-2.3.20.jar:freemarker.ext.
22              dom.XalanXPathSupport
23  [ERROR]           freemarker-2.3.20.jar:freemarker.ext.
24              dom.XalanXPathSupport$1
25  ...
```

V tomto případě plugin odhalil chybějící třídy balíků, jež jsou součástí aplikačního kontejneru a budou dostupné až za běhu.

Avšak odhalil velmi nepříjemnou chybu. Chyba se týká třídy `org.apache.xml.--utils.PrefixResolver`. Tato třída je obsažená v balíku `rt.jar` například ve verzi JRE 1.4.2. V testovacím prostředí však bylo nainstalováno JRE ve verzi 1.6.0_34, kde ale tato třída není. Záleží tedy i na tom na jaké verzi JRE je balík testován, popřípadě spušten.

K nalezeným chybějícím třídám je možné dohledat, v kterém balíku se nacházejí například pomocí služby `jarfinder.com`[2].

3.4 Nepoužívané závislosti

Aplikace je schopna nalézt i nepoužívané závislosti. V případě aplikace **hikersbook** nenalezla žádnou nevyužitou závislost. Jak je zachyceno na výpise 3.5. Toto se na první pohled zdálo jako chyba, jelikož aplikace hikersbook má ve své závislosti `junit-4.8.1`, který nepoužívá. Při důkladnějším zkoumání se však ukázalo, že tato komponenta je využívána ze třídy naležící balíku `rt.jar`.

Výpis 3.5: Výpis nalezených nepoužitých závislostí

```
1 [INFO] ----------------------------
2 [INFO] -- No unused imports found --
3 [INFO] ----------------------------
```

[2] http://www.jarfinder.com/

Pokud se `rt.jar` odstranil ze závislostí, došlo již ke správné detekci nevyužité závislosti. Jak ukazuje výpis 3.6.

Výpis 3.6: Výpis nalezených nepoužitých závislostí

```
[WARNING] ---------------------------
[WARNING] -- Unused imports found --
[WARNING] ---------------------------
[WARNING]   junit-4.8.1.jar
[WARNING]
```

Nalezením nevyužitých závislostí, můžeme předejít nepříjemnému faktu zbytečnémho nárůstu velikosti programu.

Závěr

V této publikaci byl nastíněn problém nedostatečné kontroly kompatibility u komponent třetích stran, jak ve fázi překladu, tak v ostatních fázích - nasazení a běhu. Pro řešení byla zvolena fáze překladu, jelikož kontrola v této fázi zásadně snižuje pravděpodobnost výskytu chyby v dalších fázích, odhalení chyby neznamená téměř žádné riziko a je v důsledku nejméně finančně náročné. Pro implementaci byla zvolena forma Maven pluginu, z důvodu nezávislosti na používaném vývojovém prostředí a možnost její integrace s nástroji Continuous Integration. Problém kompatibility vazeb komponent třetích stran je řešen za pomoci statické analýzy, kde dojde k nalezení veškerých vazeb mezi komponentami a ověření jejich kompatibility.

Bylo tedy dosaženo veškerých cílů práce. Prakticky se ověřilo řešení navrženého postupu a jeho implementace a to se ukázalo jako správné. Nástroj je schopen odhalit nekompatibility vazeb u aplikací třetích stran, navíc je schopen nalézt chybějící využívané součásti, upozornit na možné problémy vznikající při existenci stejnojmenných tříd a odhalit části aplikace, které nejsou využívané a jsou tudíž zbytečné. Nástroj funguje bez problémů. Aplikace je schopna ověřit kompatibilitu i u rozsáhlých aplikací v přijatelném čase, který nikterak zásadně nebrzdí sestavení aplikace. Je tedy možné ji používat při sestavování aplikace, který se při vývoji často opakuje.

Byl projeven zájem o zapojení aplikace do vývoje ze strany firmy Openmatics s.r.o, která se věnuje komponentovému vývoji v jazyce Java v oboru telematiky.

Další rozšíření práce vidím hlavně v integraci s vývojovými nástroji, jakými jsou například Eclipse, InteliJ IDEA, Netbeans atd. Dále její rozšíření i do dalších fází. Jako velmi užitečná by se jistě ukázala integrace s úložištěm obsahující informace o vazbách mezi jednotlivými verzemi komponent, což by zapříčinilo značné zrychlení celé kontroly, které by bylo oceněno hlavně při vývoji velmi rozsáhlých aplikací. Případně schopnost uložit samotné reprezentace tříd balíků, které se nemění tak, aby nedocházelo k jeji opětovnému načítání z byte kódu při každé kontrole. Tato funkce by byla obzvláště cenná při načítání velkých balíků jakým je třeba rt.jar. Načtení tohoto balíku trvá řádově několik minut.

Seznam použité literatury

[1] Kamil Ježek, Lukáš Holý, Přemek Brada, Antonín Slezáček

Software Components Compatibility Verification Based on Static Byte-Code Analysis

39th Euromicro Conference on Software Engineering and Advanced Applications

[2] Přemek Brada and Lukáš Valenta

Practical verification of component substitutability using subtype relation

Proceedings of the 32nd Euromicro conference on Software Engineering and Advanced Applications

IEEE Computer Society, 2006

[3] Kamil Jezek, Lukáš Holy, Přemek Brada

Supplying Compiler's Static Compatibility Checks by the Analysis of Third-party Libraries

17th European Conference on Software Maintenance and Reengineering IEEE Computer Society, 2013

[4] The Apache Software Foundation

Your First Mojo

`http://maven.apache.org/guides/plugin/guide-java-plugin-development.html`

[5] The Apache Software Foundation

Mojo API

`http://maven.apache.org/developers/mojo-api-specification.html`

[6] The Apache Software Foundation

Introduction to the Build Lifecycle

`http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html`

[7] Neil Bartlett

OSGi in Practice

`http://njbartlett.name/files/osgibook_preview_20091217.pdf`

Seznam použitých zkratek

JRE Java Runtime Environment

JVM Java Virtual Machine

Java SE Java Standard Edition

Java EE Java Enterprise Edition

APICC Aplication Programing Interface Compatibility Checker

JaCC Java Compatibility Checker

CCP3 Compatibility Checker Plugin 3 [sí-sí-pí-frí]

XML Extensible Markup Language

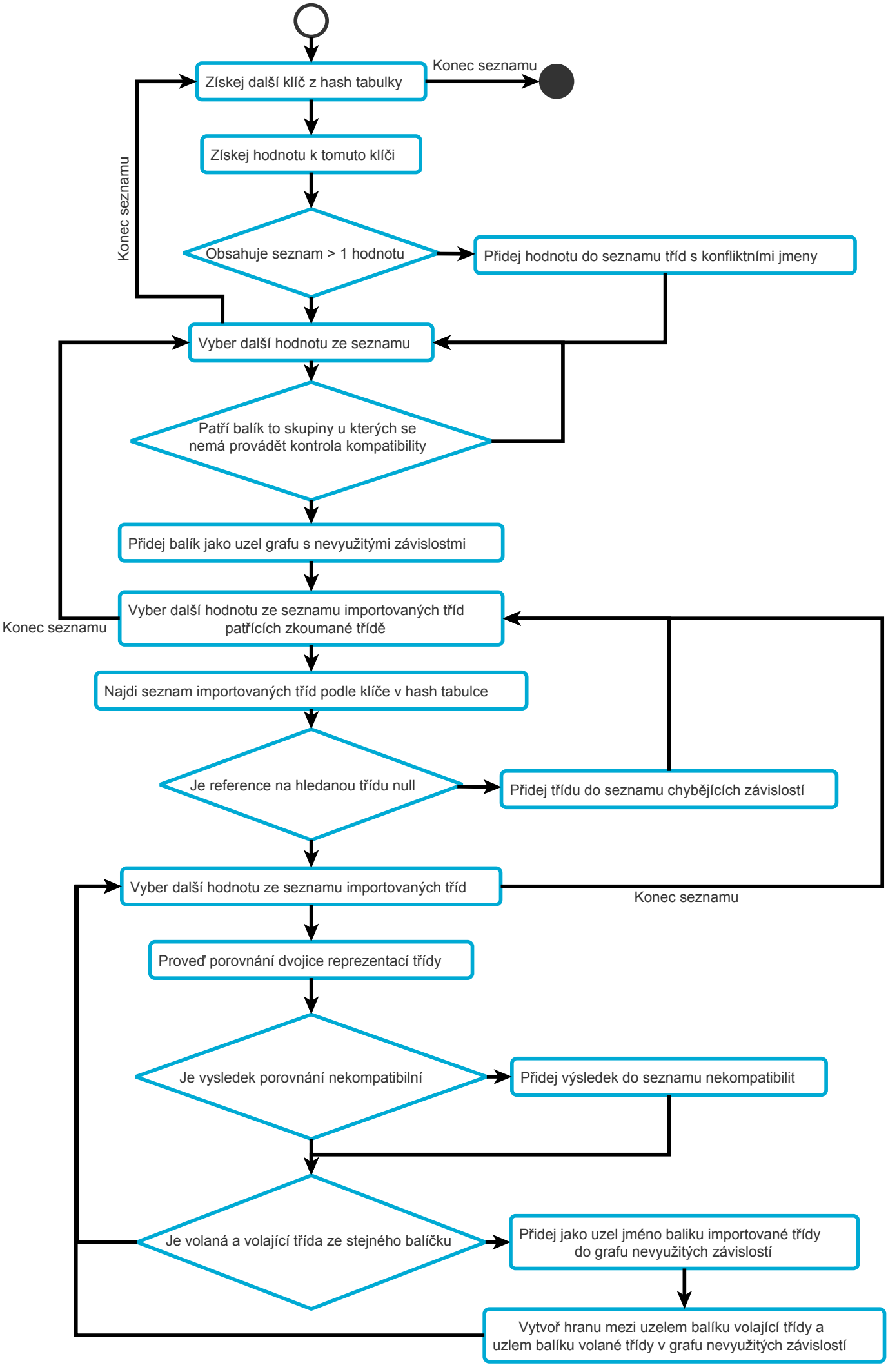
API Aplication Programming Interface

JPA Java Persistance API

jar, JAR Java Archive

WAR Web Archive

Přílohy



Obrázek 1: Vyvojový diagram algoritmu analýzy