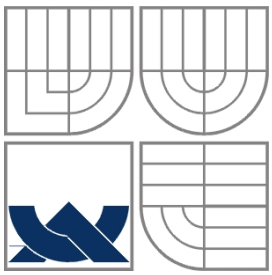
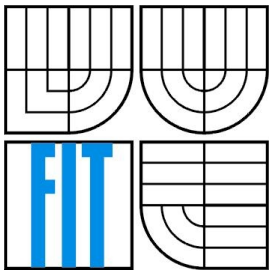


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

STATICKÁ DETEKCE ČASTÝCH CHYB JBOSS APLIKAČNÍHO SERVERU

STATIC DETECTION OF COMMON BUGS IN JBOSS APPLICATION SERVER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL VYVIAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZDENĚK LETKO

BRNO 2010

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů

Akademický rok 2009/2010

Zadání diplomové práce

Řešitel: **Vyvial Pavel, Bc.**
Obor: Inteligentní systémy
Téma: **Statická detekce častých chyb JBoss aplikačního serveru**
Static Detection of Common Bugs in JBoss Application Server
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se s aplikačním serverem JBoss.
2. Seznamte se s nástrojem pro statickou analýzu Java programů FindBugs.
3. V systému pro sledování chyb v JBoss aplikačním serveru vyberte vhodné chyby a odvoďte jejich vzory.
4. V nástroji FindBugs implementujte detektory, které tyto vzory vyhledávají.
5. Analyzujte výsledky implementovaných detektorů.
6. Zhodnoťte dosažené výsledky a diskutujte další možnosti rozšíření a využití.

Literatura:

- Marrs, T.: JBoss at work - a practical guide, O'Reilly, 2005.
- FindBugs [online]. Poslední změna: 2009-08-21. Dostupné na URL: http://findbugs.sourceforge.net/

Při obhajobě semestrální části diplomového projektu je požadováno:

- První tři body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Letko Zdeněk, Ing.**, UITS FIT VUT
Datum zadání: 21. září 2009
Datum odevzdání: 26. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Práce si klade za úkol poskytnout čtenáři popis statické analýzy prováděné prostřednictvím nástroje FindBugs nad aplikačním serverem JBoss od společnosti Red Hat. Na základě analýzy vybraných chyb byly vytvořeny vzory pro jejich detekci, které byly následně naimplementovány jako zásuvné moduly statického analyzátoru FindBugs (tzv. detektory). Vytvořené detektory byly otestovány na JBoss aplikačního serveru a výsledky jsou publikovány v závěru práce.

Abstract

First, a few bugs from a list of common bug were chosen and patterns describing these bugs were inferred. Then, detectors searching for such patterns were implemented as plug-ins to FindBugs static analyzer. Finally, detectors were used to detect bugs in development version of JBoss AS. Results are presented at the end of this paper.

Klíčová slova

formální verifikace, statická analýza, testování software, JBoss, FindBugs, chyby v programech, hledání chyb, detektor, Java

Keywords

formal verification, static analysis, software testing, JBoss, FindBugs, bugs, detector, Java

Citace

Pavel Vyvial: Statická detekce častých chyb JBoss aplikačního serveru, diplomová práce, Brno, FIT VUT v Brně, 2010

Statická detekce častých chyb JBoss aplikačního serveru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Zdeňka Letka.
Další informace mi poskytl Ing. Jiří Pechanec.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Vyvial
17. května 2010

Poděkování

Rád bych poděkoval své rodině, vedoucímu práce z VUT a Red Hatu, přátelům, kamarádům a známým za podporu při studiu.

Obsah

1 Úvod

Práce si klade za úkol prezentovat využití formální analýzy k hledání chyb v software. Konkrétním cílem je předvést *hledání vzorů častých chyb v JBoss aplikačním serveru* od firmy *Red Hat*. Toho je docíleno za pomoci *statické analýzy* prostřednictvím detektorů nástroje *FindBugs*. Práce se skládá z teoretické a praktické části. V teoretické části je k nalezení popis chybových vzorů, využitých technologií a analýz jako jsou např. formální analýza, statická analýza, Java EE, JBoss aplikační server, FindBugs, atd.. Praktická část práce se zabývá implementací detektorů často se vyskytujících chyb v analyzovaném software. Předtím než přijde na řadu teoretická část, dovolím si udělat krátkou odbočku a uvést motivaci, která by měla odpovědět na otázku: „Proč je důležité se věnovat detekci a následnému odstraňování chyb v programech?“

Člověk není neomylný. Každý, pokud se zrovna nejedná o přehnaného narcistu nebo sklerotika, když zapátrá ve své paměti, vzpomene si na nějakou chybu, kterou v životě udělal. Už staré latinské přísloví říká: „Errare humanum est. (Chybovat je lidské.)“ [1] Software v současné době tvoří převážně lidé, nepočítáme-li automatické generování kódu (např. v podobě genetického programování [31]). Dá se tedy očekávat, že nalezneme chyby i v tomto odvětví lidské činnosti. Ať už se programátoři snaží udělat software sebedokonalejší, čas od času se projeví jeden z Murphyho zákonů „Každý program obsahuje chybu.“ a objeví se chyby, které se nesmazatelně zapíší do historie informatiky jako například:

- 1962 – neúspěšná mise kosmické sondy (Mariner I space probe) [3]
- 1982 – výbuch plynovodu (Soviet gas pipeline) [4]
- 1996 – neúspěch mise Ariane 5 Flight 501(Ariane 5Flight 501) [5]
- 2003 – výpadek proudu v severovýchodní USA (Blackout in Northeast USA) [8]

První z těchto chyb se projevila až při ostrém provozu a stála USA cca. 19 miliónů dolarů, které by v případě dokonalé detekce chyb mohly být vynaloženy lépe. Dobré by bylo podotknout, že vývoj software, ať už je popisován vodopádovým nebo jiným modelem, iterativně prochází určitými etapami. Některé chyby se projeví při specifikaci požadavků, další při návrhu, jiné během vývoje, při testování nebo za ostrého provozu. Náklady (v člověko-hodinách) na opravu detekované chyby se pak zvyšují úměrně tomu, jak pozdě se na chybu přijde. Tabulka 1 ukazuje, že nárůst nákladů je mezi etapami zhruba trojnásobný. Tabulka 1 dále nepřímo ukazuje jednu z motivací, proč využívat *statickou analýzu* k hledání chyb. Tento druh detekce softwarových chyb totiž nepotřebuje mít celý funkční systém a navíc může být využíván už při samotném vývoji jednotlivých komponent, ze

kterých se konečný produkt skládá. Z toho vyplývají menší náklady na opravu dříve objevených chyb. Popis statické analýzy bude následovat v pozdějších kapitolách.

Etapa	Náklady (člověko-hodiny)
Specifikace	2
Návrh	5
Implementace	15
Akceptační testování	50
Údržba	150

Tabulka 1: Náklady na opravu chyb. [6]

National Institute of Standards and Technology (NIST) společně s Department of Commerce's v roce 2002 uveřejnily studii [32], ze které vyplývá, že softwarové chyby stojí ročně ekonomiku USA 59,5 miliard dolarů. Studie dále uvádí, že ne všechny chyby se dají odstranit. Na druhou stranu tyto instituce tvrdí, že více než třetina softwarových chyb, které stojí ročně USA 22,2 miliard dolarů mohou být odstraněny zlepšením testovací infrastruktury, která poskytne dřívější zjištění a odstranění softwarové chyby. To působí silnou motivaci k využívání technik použitelných již v brzkých etapách vývoje software, jako je např. statická analýza, na které je postavena tato práce.

Zbytek této práce je členěn následovně. Další kapitola představuje technologie využité v práci, jako je např. programovací jazyk Java. Následuje přehled chyb a možností jejich testování, kde najdeme popis využívaných technik k hledání chyb. Podkapitoly se dále zaměřují na statickou analýzu a statický analyzátor FindBugs. Práce pokračuje popisem JBoss aplikačního serveru (testovaného produktu) a jeho tracking systému. Pak přijde na řadu analýza často se vyskytujících chyb, pro které se v praktické části tvořily detektory. Další kapitola se věnuje experimentům a vše je na konci shrnuto v závěru.

2 Technologie, grafy, chyby a analýzy

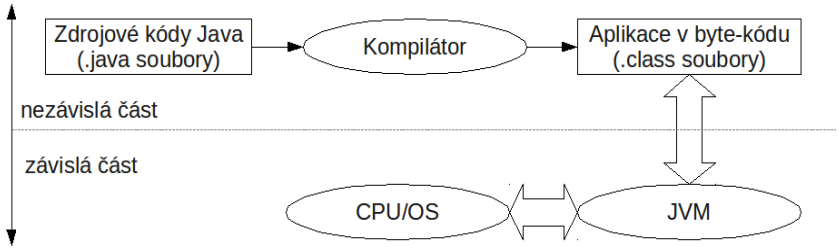
V úvodu byla uvedena motivace, z níž vyplývá ekonomická výhodnost programů bez chyb, k čemuž může dopomoci např. v práci využívaná statická analýza. První podkapitola představuje jazyk Java. Tento programovací jazyk je zde uveden proto, že jak nástroj k detekci chyb FindBugs, tak i analyzovaný produkt JBoss aplikační server jsou napsány v jazyce Java. Tato kapitola dále poskytuje přehled chyb vyskytujících se v software společně s technikami pro jejich detekci. Z technik využívaných k hledání chyb je podrobněji rozebrána statická analýza. Je to z důvodu, že je na této analýze postaven nástroj FindBugs, který figuruje v praktické části jako základní prvek spouštějící vytvořené detektory. V následující podkapitole přichází na řadu představení JBoss aplikačního serveru, pro který jsou detektory chyb tvořeny. Nakonec je zde ještě zmíněn JBoss Bug tracking systém, využívaný ke správě a hlášení chyb odkud byly vybrány chyby, pro které se v praktické části tvořily detektory.

2.1 Programovací jazyk Java

Java je lehce přenositelný objektově orientovaný programovací jazyk, původně vyvinutý Jamesem Goslingem v Sun Microsystems. Jedná se o *silně typovaný jazyk,* obsahující automatickou správu paměti (garbage collector) [43]. Java je univerzální a umožňuje tvorbu široké škály aplikací (konzole, GUI, servlet, mobilní aplikace, atd..). Objektová orientovanost jazyka nahlíží na aplikace jako na objekty, které mezi sebou komunikují voláním metod a zasíláním zpráv jednotlivým objektům. Java také podporuje tvorbu více-vláknových aplikací. Přenositelnost Java aplikací je zajištěna na binární úrovni a umožňuje tak aplikace napsané v Java přenášet na řadu platforem. Zdrojové kódy jazyka (soubory s koncovku .java) bývají kompilovány do *byte-kódu* (soubory s koncovkou .class) [44], který bývá interpretován pomocí *Java Virtual Machine* (JVM) – softwarové vrstvy sloužící jako uniformní rozhraní s platformou na které je JVM umístěna.

JVM obsahuje tyto tři hlavní paměti: 1. *Haldu* (Java Virtual Machine heap) – představující dynamickou paměť sdílenou skrz všechny JVM vlákna, kde bývají uloženy instance všech tříd. 2. *Zásobník* (Java virtual machine stack) – zajišťuje privátní paměť každého JVM vlákna a je vytvořen společně s vláknem. 3. *Method area* – je sdílená paměť skrz všechny JVM vlákna obsahující kód a statické proměnné. Najdeme zde např. Constant pool.

Některé platformy využívají kromě standardní kompilace i Just in Time kompilaci. [42] Problematiku kompilace a spouštění Java aplikací zachycuje Ilustrace 1.



Ilustrace 1: Kompilace a spouštění Java aplikace.

Ilustrace 1 ukazuje, že zdrojové soubory mohou být pomocí Java kompilátoru překompilovány na kterémkoliv stroji. Takto vzniklou aplikaci (v podobě byte-kódu) můžeme spustit na libovolném stroji s JVM (kompatibilním s verzí Javy, pro kterou byla aplikace napsána), na který bude přenesena. Z toho vyplývá, že Java programy jdou spustit na libovolném počítači s korektně běžící JVM a architektura stroje, na kterém byla prováděna kompilace nehraje v jejich spouštění roli.

JVM je přitom dostupná pro MS Windows, Unix/X, Mac OS X, mobilní zařízení, atd.. Java bývá považována za robustní, bezpečný programovací jazyk vykazující dobrou produktivitu. [40, 41]

Existují následující tři platformy Java [40, 41]:

- *Standard Edition (Java SE)* – umožňuje vývoj a rozmisťování Java aplikací na desktopech, serverech a zařízeních počítajících v reálném čase.
- *Micro Edition (Java ME)* – je jednou z nejrozšířenější aplikační platformou pro mobilní zařízení. Poskytuje robustní, flexibilní prostředí pro aplikace běžící na vestavěných zařízeních jako jsou mobilní telefony, PDA, TV set-top boxy a další. Platforma Java ME obsahuje flexibilní uživatelské rozhraní, robustní bezpečnostní model, širokou škálu vestavěných síťových protokolů, rozsáhlou podporu sítí a offline aplikací, které jdou dynamicky stáhnout.
- *Enterprise edition (Java EE)* – je průmyslový standard pro vývoj přenositelných, robustních, škálovatelných a bezpečných aplikací typu client-server. Java EE je postavena na základech Java SE. Poskytováním webových služeb, model komponent, managementu a API komunikace, vytváří průmyslový standard pro implementaci na enterprise službách orientované architektury (SOA) [61] a Web 2.0 [64] aplikací.

JVM je možné si stáhnout na stránkách společnosti Sun Microsystems i v těchto balících [40]:

- *Java Runtime Environment (JRE)* – obsahuje JVM a standardní knihovny potřebné ke spouštění Java aplikací.

- *Java Developers Kit (JDK, SDK)* – obsahuje JRE a několik nástrojů potřebných pro tvorbu Java programů. Jeho součástí je například:
  - javac – překladač zdrojových kódu do byte-kódu,
  - java – zavaděč Java aplikací schopný interpretovat byte-kód získaný z překladače javac,
  - jar – nástroj pro tvorbu jar balíčků z několika překompilovaných tříd,
  - jdb – ladící nástroj,
  - jps – nástroj poskytující informace o stavu Java procesů,
  - javap – zpětný překladač,
  - appletviewer – sloužící jako k zobrazení appletů bez potřeby webového prohlížeče,
  - javah – tvoří hlavičkové soubory pro nativní metody napsané v C,
  - javaws – Java Web Start spouštěč Java Network Launching Protocol (JNLP) aplikací,
  - extcheck – detekuje konflikty v jar balíčcích,
  - apt – anotační nástroj,
  - jhat – nástroj sloužící pro analyzování Java haldy,
  - jstack – nástroj sloužící pro analyzování Java zásobníku,
  - jstat – poskytující statistiky o JVM,
  - jstatd – jstat deamon,
  - jinfo – poskytuje informace z běžících Java procesů nebo crash dumpu (výpisu paměti),
  - jmap – slouží k výpisu paměti (např. sdílenou paměť objektů, haldu, core dump, atd.),
  - idlj – propojuje interface description language (IDL) s Java,
  - policytool – specifikuje oprávnění pro přístup ke kódu z různých zdrojů,
  - VisualVM – nástroj integrující některé příkazy příkazové řádky JDK nástrojů,
  - wsimport – vytváří přenositelné JAX-WS pro volání webových služeb.
  - jrunscript – příkazová řádka skriptovacího shellu Java.

Jak už bylo zmíněno JVM při svém běhu interpretuje Java byte-kód. Z důvodu, že i nástroj FindBugs popsaný v kapitole 2.6 provádí analýzy na základě Java byte-kódu, bude zde tento pojem stručně představen. Byte-kód je uskupení po sobě jdoucích instrukcí popisující funkci programu na strojové úrovni. Každá instrukce má velikost jeden byte a je reprezentována pomocí *opcode* (operation code) společně s (případně) využívanými parametry. Celkem existuje 256 různých opcode. Instrukce je možné rozdělit do následujících skupin: 1. načítající a ukládající, 2. aritmetické a logické, 5. provádějící typovou konverzi, 4. vytvářející a manipulující s objekty, 5. pracující s operand zásobníkem, 6. kontrolní a skokové 7. volající metody a navracející. Z těchto skupin můžete v Tabulce 2 nalézt instrukce využité při analýze v praktické části práce.

Název instrukce	Zjednodušený popis
instanceof	Rozhoduje, zda je objekt vyjmutý z operand zásobníku stejný jako typ určený parametry instrukce.
invokestatic	Volá statickou metodu určenou parametry instrukce.
invokeinterface	Volá metodu rozhraní určenou parametry instrukce.
invokevirtual	Volá instanci metodu určenou parametry instrukce.
invokespecial	Volá instanci metody určenou parametry instrukce. Touto metodou ovšem je speciální instrukce jako např. inicializace, atd.
if_acmpne	Porovná dvě reference z operand zásobníku na shodu. Pokud se neshodují, tak se v rámci byte-kódu skáče na pozici danou parametry instrukce.
ifnull	Porovná referenci ze zásobníku na null. Pokud je reference null, tak se v rámci byte-kódu skáče na pozici danou parametry instrukce.
ifnonnull	Porovná referenci ze zásobníku na null. Pokud reference není null, tak se v rámci byte-kódu skáče na pozici danou parametry instrukce.
aload_<n>	Načte n-tou referenci z lokálních proměnných a vloží ji na operand zásobník.
ireturn	Navrací int z volané metody volající metodě. Hodnota je vyjmuta ze zásobníku metody, kde byla nalezena ireturn instrukce a je vložena na zásobník od volající metody.
monitorenter	Značí vstup do synchronizované oblasti. Vlákno vykonávající monitorenter se pokusí získat monitor asociovaný s objektem z operand zásobníku. Pokud vlákno už monitor vlastní, tak inkrementuje počítadlo vstupů do synchronizované oblasti. V případě, že je monitor vlastněn jiným vláknem, tak vlákno čeká dokud není monitor uvolněn.
monitorexit	Značí odchod ze synchronizované oblasti. Vlákno vlastnící monitor dekrementuje počítadlo značící počet vstupů do synchronizované oblasti. Pokud je počítadlo na hodnotě 0, tak se monitor uvolní.
putfield	Nastaví místo v Constant Poolu identifikované parametrem instrukce.
putstatic	Nastaví místo v Constant Poolu identifikované parametrem instrukce.
goto	Skáče na pomocí parametrů určené místo v kódu.

Tabulka 2: Byte-kódové instrukce využívané detektory při analýze.

V práci se občas mluví o tzv. *invoke* instrukci. Pod tímto označením vystupují následující instrukce: invokeinterface, invokespecial, invokestatic a invokevirtual. Dále se v textu občas objevují *fieldInstruction*, pod které jsou zařazeny instrukce: putfield a putstatic. Z důvodu, že např. invoke instrukce volají metody, které je potřeba nějakým způsobem rozlišit byla vytvořena signatura metod popsaná v následující podkapitole.

2.1.1 Signatura Java metod

Signatura Java metod představuje jméno metody společně s počtem a typy jednotlivých parametrů metody. Signatura dále zahrnuje i návratový typ metody. Ve třídě je povoleno deklarovat dvě metody se stejným jménem, musí se ovšem lišit buď v počtu parametrů, jejich typem, pořadím nebo v návratovém typu metody. Třída nesmí deklarovat dvě metody se stejnou signaturou. Signatura metod bývá v práci využívána např. při zadávání parametrů detektorů.

Formát signatury musí dodržovat následující pravidla:

- Oddělovačem v cestě třídy je znak: „/“
- Mezi názvem třídy a názvem metody je oddělovací znak: „.“
- Název metody je ukončen znakem: „:“
- Za dvojtečkou následují závorky: „()“ do kterých se vepisují parametry metody.
- Datové typy a pole se zadávají pomocí zkratek uvedených v Tabulce 3.
- Při zadávání objektu se za písmenem „L“ specifikuje třída objektu.
- Jméno objektu je zakončeno středníkem.
- Za závorkou s parametry se stejným formátem jako parametry specifikuje návratový typ metody.

Datové typy	**Znak**	**Datové typy/pole**	**Znak**
boolean	Z	long	J
byte	B	object	L
char	C	short	S
double	D	void	V
float	F	array	[
int	I		

Tabulka 3: Speciální znaky využitelné v signatuře metod.

Jak mohou následující pravidla vypadat v praxi ukazuje tento příklad:
„java/lang/Class.forName:(Ljava/lang/String;ZLjava/lang/Classloader;)Ljava/lang/Class;“
představující metodu forName(java.lang.String, boolean, java.lang.Classloader) ze třídy java.lang.Class a návratovým typem java.lang.Class.

2.2 Využívané grafy

Tato kapitola je věnována popisu grafů. Konkrétně je zde představen *graf toku dat (control flow graph – CFG), graf volání (call graph – CG)* a *graf dědičnosti (inheritance graph – IG)*. Grafy jsou zde uvedeny z důvodu, že vytvořené detektory popisovány v kapitole 4 je využívají při analýze.

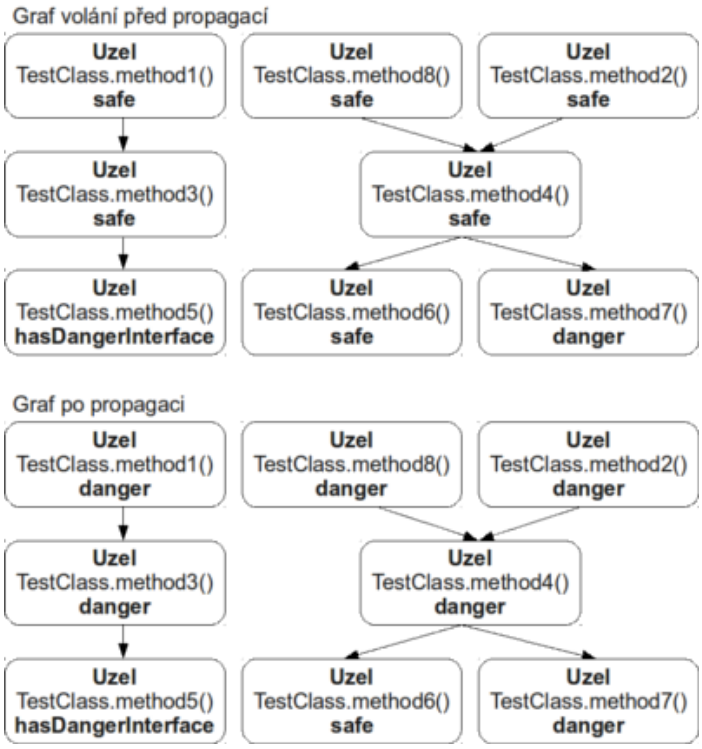
2.2.1 Graf toku dat

Graf toku dat (control flow graph – CFG) pomocí grafové notace reprezentuje všechny dostupné cesty v počítačovém programu. Cestou se přitom myslí všechny basic blocky navštívené za běhu programu. CFG je orientovaný graf definovaný jako uspořádaná dvojice (V, E), kde V je neprázdná množina uzlů a E množina některých uspořádaných dvojic prvků z množiny V reprezentující hrany. Uzly představují basic blocky analyzované metody. Orientované hrany spojují dva po sobě následující basic blocky a mají přiřazené označení z `edu.umd.cs.findbugs.ba.EdgeTypes`. Protože některé z těchto označení jsou zmiňovány v kapitole 4 zabývající se implementací, budou zde ty nejdůležitější uvedeny:

- IFCMP_EDGE – představuje hranu spojující uzel, ve kterém bylo nějaké porovnání (podmínka) s uzlem, kde se má pokračovat v případě pravdivě vyhodnocené podmínky.
- FALL_THROUGH_EDGE – představuje hranu spojující uzel, ve kterém bylo nějaké porovnání (podmínka) s uzlem, kde se má pokračovat v případě nepravdivého vyhodnocení podmínky. Toto označení bývá dále použito u hran spojující dva uzly v nevětvícím se kódu.
- SWITCH_EDGE – představuje hranu reprezentující explicitní skok u switche.
- SWITCH_DEFAULT_EDGE – představuje hranu reprezentující standardní skok u switche.
- GOTO_EDGE – hrana reprezentující skok pomocí goto instrukce.

2.2.2 Graf volání

Graf volání využívají některé detektory (*FindCourierUseOutOfInvoker*, *TestMethodGetPayLoad*, *TestSuddenAtributesChange*) popsané v kapitolách 4.2, 4.3 a 4.5. Graf volání je orientovaný graf definovaný jako uspořádaná dvojice (V, E), kde V je neprázdná množina uzlů a E množina některých uspořádaných dvojic prvků z V reprezentující hrany. Uzly představují metody analyzovaných tříd. Orientované hrany jsou `invoke` instrukce spojující zdrojový uzel (metodu, kde byla nalezena `invoke` instrukce) s cílovým uzlem (metodu uvedenou v parametru `invoke` instrukce jako cíl volání). Hrany spojují vždy dva uzly. Jeden uzel může přitom náležet do více hran. Uzly grafu mohou být různě označovány. Označení uzlu definuje informace (o bezpečnosti uzlu) vyčtené z byte-kódu metody korespondující k uzlu.

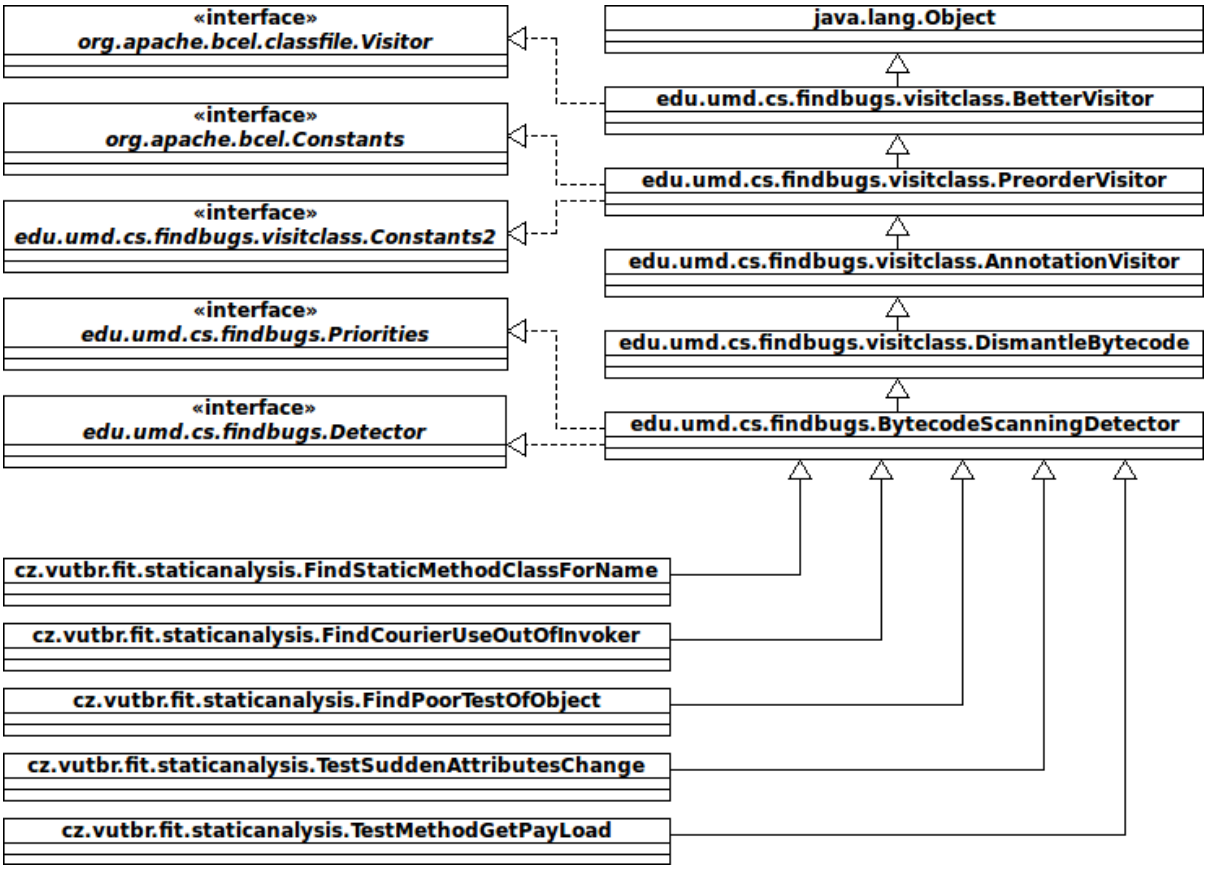


Ilustrace 2: Graf volání před a po propagaci.

Při práci s grafem volání se často využívá tzv. *propagace označení*. Jedná se o to, že podle označení cílového uzlu se přeznačují zdrojové uzly hran spojených s cílovým uzlem. Tato propagace se přitom provádí tranzitivně z nějaké podmnožiny uzlů do doby dokud je kam označení přes graf propagovat. Jak taková propagace může prakticky vypadat ukazuje Ilustrace 2. V horní části obrázku je graf před propagací označení hasDangerInterface a označení danger. V dolní části obrázku je graf po propagaci. Propagace označení přitom nemusí odpovídat nastavení označení zdrojového uzlu na označení cílového uzlu. Propagace může také měnit označení zdrojového uzlu na jiné označení než je označení cílového uzlu. Na Ilustraci 2 to dokazuje propagace označení uzlu patřící metodě TestClass.method5(). Jde vidět, že propagace označení této metody přiřadilo uzlu metody TestClass.method3() označení danger a ne označení hasDangerInterface. Tímto skončilo propagování označení hasDangerInterface. Propagace danger označení začala tím, že označení patřící uzlu od metody TestClass.method7() se propagovalo metodě TestClass.method4() odkud se danger označení propagovalo dále do metod TestClass.method8() a TestClass.method2(). Nakonec se ještě propagovalo označení z uzlu metody TestClass.method3() do uzlu metody TestClass.method1() a tímto propagace danger označení skončila, protože už nebylo kam dále propagovat. V detektorech často záleží, které označení se propaguje jako první. V případě, že by se totiž nejdříve propagovalo danger označení a až poté označení hasDangerInterface, tak by došlo k situaci, kdy by uzel patřící metodě TestClass.method1() neměl označení danger.

2.2.3 Graf dědičnosti

Graf dědičnosti využívají některé detektory (*TestMethodGetPayLoad*, *TestSuddenAtributesChange*) popsané v kapitolách 4.3 a 4.5. Graf dědičnosti je uspořádaná dvojice (V, E), kde V je nějaká neprázdná množina uzlů a E je množina hran. Uzly představují analyzované třídy nebo rozhraní. Hrany jsou uspořádané dvojice uzlů z množiny V vytvářeny mezi třídou dítěte (poděděnou třídou) a rodičovskou třídou (odkud se dědilo) nebo mezi třídou implementující nějaké rozhraní a rozhraním. Hrany spojují vždy dva uzly. Jeden uzel může figurovat ve více hranách. Uzel od každé analyzované třídy pokud se nejedná o uzel patřící `java.lang.Object` je v grafu dědičnosti spojen minimálně s jedním uzlem představující rodičovskou třídu odkud se dědilo. Pokud uzel figuruje ve více hranách jako dítě, tak je jasné, že implementuje nějaké rozhraní ,protože v jazyce Java není dovoleno dědit z více jak jedné třídy.



Ilustrace 3: Graf dědičnosti zachycující vztahy mezi detektory a FindBugs třídami.

Ilustrace 3 ukazuje příklad grafu dědičnosti zachycující vztah mezi implementovanými detektory a FindBugs třídami/rozhraními, které detektory podědily. Na obrázku jde vidět, že v grafu dědičnosti se vyskytují jak třídy, tak rozhraní. Vztahy zachycující odkud dědily rozhraní nebyly kvůli

zjednodušení do obrázku zahrnuty. Ilustrace dále demonstruje, že každá třída kromě třídy java.lang.Object musí odněkud dědit. Všechny detektory mají stejnou rodičovskou třídu BytecodeScanningDetector, která je poděděna z třídy DismantleBytecode. Třída BytecodeScanningDetector přímo implementuje rozhraní Priorities, Detector a nepřímo implementuje rozhraní Constants, Constants2 a Visitor. Přímou implementací/dědičností se zde myslí situace, kdy nejkratší cesta mezi uzlem třídy implementující/dědící rozhraní/třídu a uzlem rozhraní/rodiče obsahuje pouze jednu hranu. Pokud je na nejkratší cestě mezi uzly více jak jedna hrana, tak se jedná o nepřímou implementaci/dědičnost.

2.3 Chyby v programech

Tato práce se zabývá počítačovými chybami a jejich detekcí. Počítačová chyba (software bug) je označení využívané k popisu nekorektního chování počítačového programu. Chyba může přitom způsobit selhání programu, jeho špatnou funkčnost, nedokonalost, chybovost, nebo podávání neočekávaných, neúplných, či nekorektních výsledků.

Chyby lze rozdělit do následujících skupin [33]:

- **Koncepční chyby** – představují syntakticky správně napsaný kód, který ovšem programátor nebo softwarový architekt špatně navrhl, a tak dělá něco jiného, než co je očekáváno.
- **Matematické chyby** – sem patří např. nedovolené matematické operace, jako je dělení nulou, aritmetické přetečení, nebo podtečení a ztráta přesnosti způsobená zaokrouhlováním.
- **Logické chyby** – mezi tyto chyby patří např. nekonečné smyčky, nekonečná rekurze, dále sem můžeme zařadit tzv. chybu o jedničku (off-by-one error).
- **Syntaktické chyby** – pod které se dá zařadit např. nekorektní užívání klíčových slov. Jednoduché příklady těchto chyb bývají často odhaleny již překladačem při překladu.
- **Zdrojové chyby** – kde najdeme např. null pointer dereferenci (nullPointerException – NPE), užití neinicializované proměnné, chyba segmentace (segmentation fault), úniky zdrojů (resource leaks), přetečení zásobníku (buffer overflow), nebo nestřídmé využívání rekurze, které dokáže přivodit přetečení haldy (stack overflow).
- **Co-programming chyby** – pod které patří např. uváznutí (deadlock), časově závislá chyba nad daty (data race), chyby souběžnosti (concurrency bugs) a špatné využití zámků.
- **Týmové chyby** – mezi které patří nepropagované úpravy, nesprávné, nebo neaktuální komentáře, rozdíly mezi dokumentací a aktuálním produktem.

Důležité je si uvědomit, že vlastnosti jednotlivých chyb se mohou značně lišit. Některé chyby se dají „snadno“ detekovat automaticky. Mezi tyto druhy chyb patří např. syntaktické chyby, které je schopen nalézt překladač při kompilaci kódu. Pro jiné druhy chyb, jako jsou null pointer dereference, nebo nesprávné používání zámků je pro detekci potřeba sofistikovanějších analýz. Existují ovšem i takové chyby, které jsou postaveny na nerozhodnutelných problémech.

2.4 Techniky hledání chyb

Předchozí podkapitola se zabývala různými druhy chyb objevujících se v počítačových programech. K jejich hledání se používá spousta technik, některé z nich budou rozebrány v této podkapitole.

Před popisem technik jsou představeny hlášení, které mohou jednotlivé techniky produkovat:

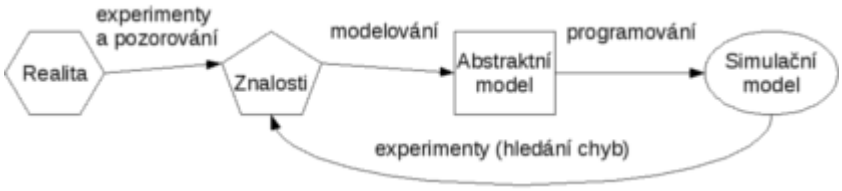
- *True positive* – hlášení s tímto označením tvrdí, že se v systému daný typ chyby vyskytuje. Toto tvrzení je pravdivé. Systém totiž danou chybu obsahuje.
- *False positive (false alarm)* – hlášení s tímto označením tvrdí, že se v systému daný typ chyby vyskytuje. Toto tvrzení je ovšem mylné. Systém totiž danou chybu neobsahuje.
- *False negative* – hlášení s tímto označením tvrdí o systému, že se v něm určitý druh chyby nevyskytuje. Toto tvrzení je mylné. Systém totiž danou chybu obsahuje.
- *True negative* – hlášení s tímto označením tvrdí, že se v systému daný typ chyby nevyskytuje. Toto tvrzení je pravdivé. Systém totiž danou chybu neobsahuje.

Techniky využívané k detekci chyb se dají rozdělit na formální a neformální. Formální techniky jsou na rozdíl od neformálních postaveny na matematickém základu. Formální metody díky toho dokážou kromě hledání chyb i matematicky dokázat, že systém určitou chybu neobsahuje.

Klasifikace technik využívaných ke hledání chyb může být následující:

- neformální techniky
  - testování
  - simulace
  - dynamická analýza
- formální techniky
  - dokazování vět (theorem proving)
  - model checking
  - statická analýza (static analysis)

Simulace [34] je technika založena na vytvoření abstraktního modelu, který odráží důležité vlastnosti modelovaného systému. Abstraktní model je převeden na simulační model. S takto vytvořeným simulačním modelem jsou následně prováděny simulace (experimenty). Ze simulací jsou získány nové výsledky, které se po zobecnění dají použít pro modelovaný systém. Ilustrace 4 zobrazuje celý proces získávání znalostí (hledání chyb) s využitím simulace.



Ilustrace 4: Simulace: Realita → Znalosti → Abstraktní model → Simulační model [34]

Výhoda tohoto druhu hledání chyb je v tom, že se nepracuje přímo s modelovaným systémem ale pouze s jeho modelem. Model může být o poznání jednodušší než modelovaný systém. Nevýhodou této techniky je velká náročnost vytváření abstraktního modelu z modelovaného systému (modelování). Jeho další nevýhodou je, že kvalita vytvořeného modelu zásadním způsobem ovlivňuje kvalitu výsledků získaných z experimentování s modelem. Mezi nástroje využívající tuto techniku se dá zařadit např. Quartus II. [34]

Testování [36, 39] je jednou z nejrozšířenějších technik k hledání chyb v programech. Je založeno na tom, že se vytvoří tzv. testovací případy (test cases), které definují výstup systému při použití patřičného vstupu. Pokud není patřičné odezvy testovaného systému dosaženo, tak se předpokládá chyba v programu nebo v testovacím případu. Výhodou testování je, že určité aplikace mohou testovat i patřičně proškolení laikové, kteří dostanou před-připravené test cases. Nevýhodou je, že takovéto testování nemůže oproti formálním technikám zaručit korektnost systému. Nástroje využívající testování ke hledání chyb je např. JUnit.

Dynamická analýza [9, 36] je termín využívaný v softwarovém inženýrství k popisu testování běžícího chování kódu. Technika stejně jako testování využívá spouštění kódu. Dynamická analýza automaticky shromažďuje informace vztahující se k analýze a prozkoumání neobvyklých běhů programu. Obvykle bývá postavena na vkládání kódu, který získává požadované informace (instrumentace), přímo do originálního kódu. Získané informace jsou analyzovány buď za běhu (on-the-fly) nebo po ukončení aplikace (post-mortem). Výhodou této techniky je schopnost nalézt i málo pravděpodobné chyby. Nevýhodou bývá produkování false negative a false positive hlášení. Mezi nástroje využívající tuto techniku patří např. BoundsChecker, Daikon, jTracert, Valgrind, atd.

Dokazování vět [9, 37] je přístup podobný matematickému dokazování. Dokazování teorémů začíná s axiomy, ze kterých se pomocí dedukčních pravidel získají nová fakta (vlastnosti systému). Nevýhodou této techniky je, že některé nástroje, které ho využívají, jsou schopny pouze poloautomatického dokazování korektnosti systému. Proto je při dokazování teorému zapotřebí expert, který daný důkaz vede. Využívány bývají typicky poloautomatické dokazovací prostředky (PVS, Isabel,Coq,ACL/2, atd.).

Model checking [9, 38] je založen na systematickém, explicitním, či symbolickém průchodu všemi stavy kontrolovaného systému, nebo jeho pokud možno automaticky vytvořené abstrakce za účelem kontroly, zda vyhovuje požadovaným vlastnostem. Model checking pracuje buď s reálným systémem, nebo s jeho modelem. Technika bývá využívána převážně pro systémy s konečným počtem stavů. Ověřované vlastnosti bývají obvykle popisovány temporálními logikami (LTL, CTL, CTL*, atd.). Mezi výhody model checkingu patří: relativně vysoký stupeň automatizace, snadnost použití, obecnost a poskytování protipříkladů (v případě, kdy je nalezena chyba, tak model checking poskytuje informaci jak se chyby docílilo). Tento způsob ověřování korektnosti systému sebou nese dvě nevýhody: časovou a paměťovou náročnost. Je to zapříčiněno tím, že reálný svět často požaduje nekonečný stavový prostor nebo má příliš velké požadavky na výkon počítače (důvodem je nedeterminismus a nutnost práce s velkým množstvím možných hodnot u různých datových typů – jeden třicetidvou-bitový integer může nabývat 2^{32} stavů). Model checking z tohoto důvodu bývá často vylepšován různými technikami pro redukci stavového prostoru, apod.. Mezi nástroje využívající model checking patří: Spin, SMV, RuleBase (IBM), Incisive Formal Analysis (Cadence), Blast, Magic Copper, JPF (NASA), SLAM/SDV (Microsoft), Magellan (Synopsys), atd.

Další technika využívána k detekci chyb je *statická analýza.* Bude zde podrobněji představena, protože na jejich základech pracuje nástroj FindBugs využívaný k praktickému řešení práce.

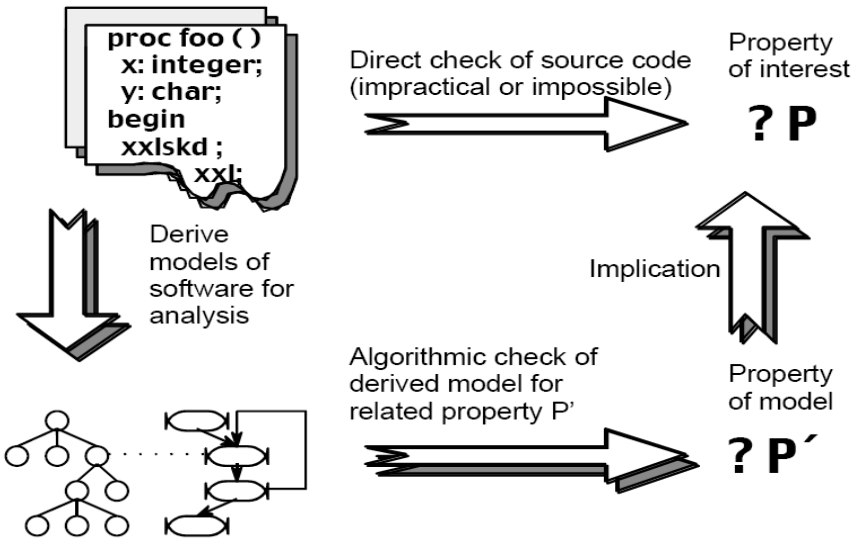
Statická analýza [2, 9] je založena na získávání znalostí ze zdrojového kódu aplikace bez jejího spouštění. Na základě zdrojového kódu odvozuje chování aplikace, ve které se snaží najít potenciální chyby. Statická analýza se naží obejít stavovou explozi, se kterou se potýká *model checking*. Některé druhy této metoda bohužel produkují velké množství false positive hlášení o chybách.

Rozlišujeme následující druhy statické analýzy: [12, 19]

- *Hledání chybových vzorů (bug pattern searching)* – jedná se o hledání předem definovaných vzorů v kódu, jejichž nalezení může zapříčinit chybové chování systému. Více informací bude zmíněno v kapitole 2.5.

- *Analýza toku dat (dataflow analysis)* – staticky simuluje aplikaci. Postupně prochází grafem toku dat a ohodnocuje jednotlivé místa fakty, nad kterými následně provádí analýzu. Jedná se o techniku bez spouštění kódu, která může automaticky odhalit potenciální problémy vznikající za běhu systému jako „úniky zdrojů (resource leaks), špatně inicializovaný objekt (NullPointerException), atd.“
- *Abstraktní interpretace (abstract interpretation)* – modeluje efekt každého příkazu jako stav abstraktního stroje, který nad-aproximovává chování systému.

Ilustrace 5 zobrazuje s čím je statická analýza schopna pracovat. Kus zdrojového kódu zobrazeného v levém horním rohu obrázku nám představuje verifikovaný systém. Programátor, který má systém verifikovat se nyní může rozhodnout pro dvě cesty. První cesta je přímé využití zdrojového kódu při verifikaci k získání požadovaných vlastností. Tato varianta je ovšem velice složitá a v praxi téměř nepoužitelná. Druhá cesta je poněkud delší, za to však v praxi použitelnější. Skládá se z vytvoření modelu na základě zdrojových kódu, získání požadovaných vlastností a následného odvození vlastností verifikovaného systému z vlastností modelu. [2, 9, 18]



Ilustrace 5: Analýza za pomocí modelu. Převzato z [18]

Pevná hranice mezi tím, co do statické analýzy patří a co ne, nebyla ustálena. Pokud budeme považovat za statickou analýzu opak dynamické analýzy a vyzdvihneme tedy její vlastnost, že nespouští zdrojový kód, dal by se za speciální druh statické analýzy považovat i model checking. Statická analýza je přitom využívána i v jiných oblastech (optimalizace, při vytváření kódu, atd..) než jen k ověřování korektnosti systému. [9]

Mezi výhody statické analýzy patří: [9, 12, 18]

- rychlost,
- možnost vytvářet nástroje s velkým stupněm automatizace,
- schopnost analyzovat obrovské systémy,
- nepotřebuje model prostředí (vstupy/výstupy, některé případně i knihovny, jiné moduly),
- ověřování malých částí rozsáhlých systémů,
- umožňuje hledat chyby už v brzkých fázích vývoje.

Mezi nevýhody statické analýzy patří: [9]

- některé analýzy produkují velké množství hlášení o chybách, které ve skutečnosti neexistují,
- jednotlivé statické analýzy jsou často vhodné jen pro řešení specifických úloh – pokud vznikne požadavek na verifikaci nového problému, je třeba vytvořit novou statickou analýzu pro jeho ověřování.

U výše zmíněných nevýhod by bylo dobré zmínit, že jdou trochu proti sobě. To znamená, že pokud se vývojář analyzátoru snaží odstranit např. problém velkého množství „false alarms“ zpřesňováním analýzy, začne narážet na podobné problémy jako u model checkingu a současně jim vytvořený analyzátor může být obtížnější, ne-li nemožné, využít při analýze jiného systému než, pro který byl analyzátor vytvořen a naopak.

K analýze JBoss Aplikačního serveru, pro který jsou v této práci vytvořeny detektory pro odhalení častých chyb, je využito statické analýzy prostřednictvím nástroje FindBugs, který bude podrobněji popsán v kapitole 2.6.

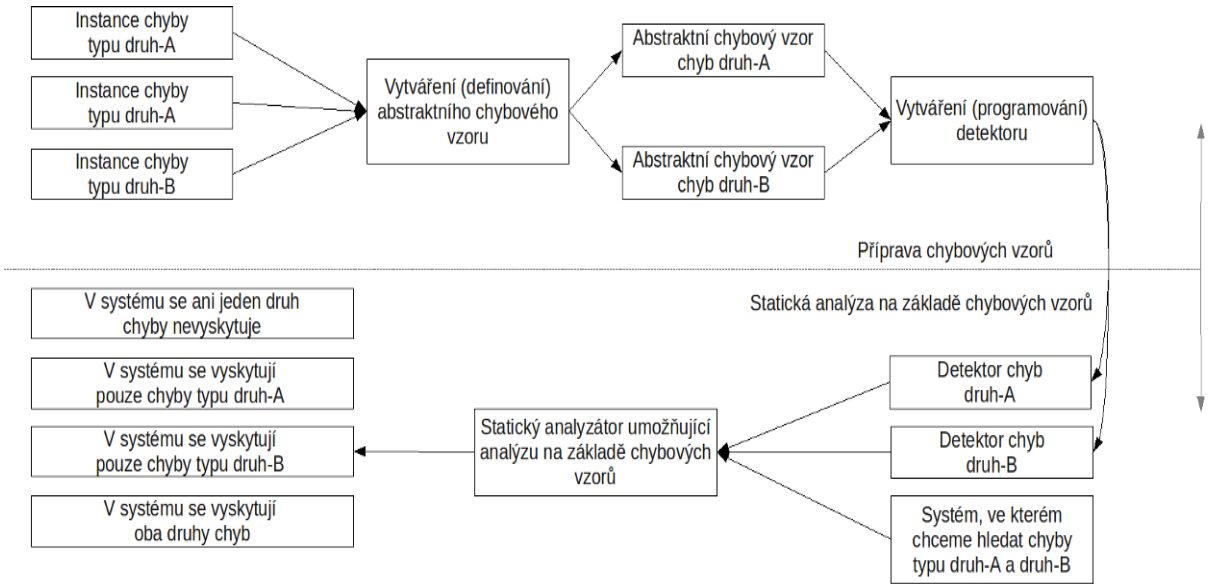
2.5 Statická analýza založená na hledání chybových vzorů

Statická analýza založená na hledání chybových vzorů staví na myšlence, že se počítačové chyby opakují. Když programátor dokáže identifikovat podstatu chyby a na jeho základě vytvoří chybový vzor, tak technikami jeho detekce dokáže vzorem popsané chyby hledat. Pomocí analýzy založené na hledání chybových vzorů můžeme absolutně garantovat, že patřičný systém neobsahuje chyby popsané vzory využitými při analýze. Jako vzor je přitom nejčastěji chápána určitým způsobem definovaná sekvence po sobě jdoucích příkazů. Obvykle bývají vzory popisovány pomocí regulárních výrazů, konečných automatů, gramatik, systémů různých omezení, atd. Vzor může popisovat buď korektní nebo nekorektní běh programu. Na vzorech založená analýza pak funguje tak, že se buď

snaží najít vzor definující chybu nebo porušení vzoru definujícího korektní běh. Ilustrace 6 znázorňuje tvorbu chybových vzorů a následnou statickou analýzu. [12, 19]

Níže uvedený obrázek 6 v horní části ukazuje vytváření chybových vzorů. Na začátku celého procesu jsou tři instance dvou druhů chyb (dva výskyty druh-A a jeden výskyt druh-B). U těchto chyb je definována jejich podstata a jsou tak vytvořeny abstraktní chybové vzory. Ty jsou následně pomocí programovacích technik převedeny na detektory schopné hledat chybové vzory.

Dolní část obrázku představuje statický analyzátor dostávající na vstup chybové vzory v podobě detektorů a systém, který chceme verifikovat. Výsledkem je buď odpověď představující bezchybný stav nebo odpověď značící nalezení jednoho nebo více druhů chyb.



Ilustrace 6: Vytváření chybových vzorů (nahoře) a statická analýza (dole).

Na internetu je k nalezení řada statických analyzátorů, které se zabývají ověřováním korektnosti Java programů prostřednictvím statické analýzy založené na hledání chybových vzorů. Zde je uvedeno několik zástupců společně s jejich stručnou charakteristikou (obsahující informaci, zda se jedná o placený software, jaký kód je nástroj schopen testovat, kolik obsahuje předdefinovaných chybových vzorů, zda umožňuje tvorbu vlastních vzorů, atd.):

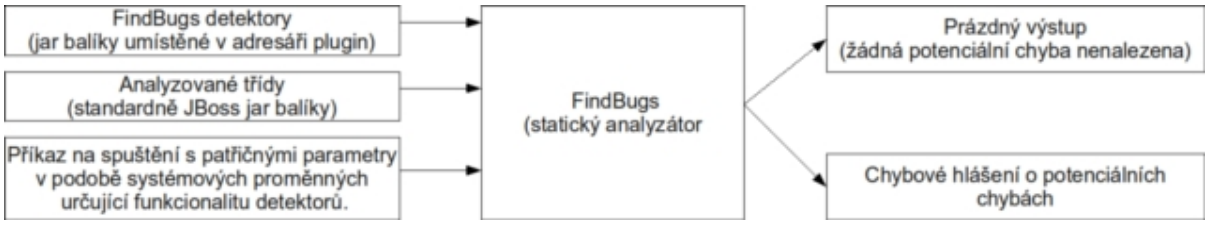
- *CheckStyle* [13] – Jedná se o open source nástroj. Umožňuje tvorbu vlastních chybových vzorů v programovacím jazyku Java. Zaměřen na problémy stylu – Java a EJB 2.x [49]. Obsahuje okolo 100 chybových vzorů, které se dají prostřednictvím tohoto nástroje hledat v software. Podporuje pouze pro Java kód.
- *FindBugs* [14] – Jedná se o open source nástroj. Má propracovanou správu reportů a historii chyb. Umožňuje tvorbu vlastních chybových vzorů v programovacím jazyku Java. Nástroj analyzuje byte-kód. Obsahuje okolo 300 hledaných chybových vzorů, které se dají

prostřednictvím tohoto nástroje hledat v software. Existuje pouze pro Java kód. Bude popsán v kapitole 2.6.

- *IntelliJ IDEA* [15] – Jedná se o komerční software. Obsahuje přes 700 chybových vzorů, které se dají prostřednictvím tohoto nástroje hledat v software. Umožňuje tvorbu vlastních chybových vzorů. Zvládá analýzu závislostí, Dependency Structure Matrix (DSM) [69].
- *PMD* [16] – Jedná se o open source nástroj. Umožňuje tvorbu vlastních chybových vzorů v Java a Xpath [70]. Obsahuje okolo 200 chybových vzorů v Java, JSP [58] a JSF [54], které se dají prostřednictvím tohoto nástroje hledat v software.
- *TPTP* [17] – Jedná se o open source nástroj. Integrován do Eclipse. Obsahuje více než 100 chybových vzorů, které se dají prostřednictvím tohoto nástroje hledat v software. Umožňuje tvorbu vlastních chybových vzorů. Je dostupný pouze pro Java kód.

2.6 Statický analyzátor FindBugs

V předchozí sekci byla představena statická analýza založená na hledání chybových vzorů a nyní bude uveden nástroj FindBugs, který ji využívá a byl použit k hledání chyb v JBoss aplikačním serveru. Autor práce už v minulosti pracoval s nástrojem v rámci projektu SHADOWS [2, 11].



Ilustrace 7: Základní idea použití detektorů.

FindBugs běží na platformě Java. Nástroj využívá statickou analýzu k hledání chyb v Java programech. Je to open source software distribuovaný pod „Lesser GNU Public License“ [71]. Nástroj provádí analýzu Java byte-kódu. FindBugs z výše uvedených metod statické analýzy využívá: hledání chybových vzorů a analýzu toku dat. Nástroj poskytuje přes 300 předdefinovaných chybových vzorů společně s možností si vytvořit chybový vzor vlastní. Vlastní chybový vzor je možné si napsat v jazyce Java a do FindBugs ho naimportovat jako plugin, což zobrazuje Ilustrace 7. Nástroj je možné integrovat do Eclipse a NetBeans. FindBugs prostřednictvím svého grafického uživatelského rozhraní (GUI) poskytuje dobrou správu reportů a historii chyb. Jedna z nevýhod pramenící z toho, že nástroj využívá statickou analýzu je ta, že nástroj občas produkuje false alarmy. FindBugs potřebuje ke svému provozu virtuální stroj kompatibilní se Sun's JRE (nebo JDK) verzi 1.5

nebo vyšší. FindBugs obsahuje plugin infrastrukturu umožňující zavádění nových (uživatelem vytvořených) detektorů. Analýza probíhá takto:

- Pomocí *Byte Code Engineering Library (BCEL)* [20] se vytvoří model aplikace/části aplikace. V tomto modelu se vyskytují jednotlivé komponenty (třídy, metody, Constant Pool, atd.) aplikace jako třídy BCEL.
- Model je dále obohacen o informace zahrnující: CFG, CallGraph a InheritanceGraph.
- Vytvoří se pořadí pluginů (detektorů) jak se budou postupně spouštět. Všechny detektory nejdřív analyzují jednu třídu a až poté se přestoupí k další. Detektory přitom můžou být vytvořeny v programovacím jazyce Java a od FindBugs verze 1.1 mohou být napsány v *ASM* [21] byte-kód frameworku.
- FindBugs postupně podle plánu spouští pluginy a výsledky ukládá do BugReporteru.
- Průběžně sesbíraná chybová hlášení se nakonec objeví v celkovém reportu, který může být za pomocí *dom4j* [22] vytvořen v XML formátu.

Chyby, které je FindBugs schopen hledat, tvůrci nástroje rozdělili do těchto skupin: [2, 15]

- *Korektnost (correctness)* – jedná se o části kódu, které programátor nejspíš neměl v úmyslu vytvořit (nekonečné smyčky, neinicializované ukazatele, atd.).
- *Multivláknová korektnost (multithreaded correctness)* – zde se dají zařadit chyby jako např. sdílení statického modelu, zamykání a odemykání ve všech cestách, nepodmíněného čekání.
- *Prohnané (dodge)* – představují chyby jako redundantní porovnání, zápis do statických polí (field) z instance metody, atd.
- *Špatný postup (bad practice)* – sem bylo zařazeno porovnání stringů, hashCode() a equals(), obecné zásady jmen.
- *Výkonnostní (performance)* – obsahují chyby jako new String(String), nepoužité pole (field), použití URL v množině nebo mapě.
- *Záludná poruchovost kódu (malicious code vulnerability)* – kde najdeme např. chybu, která hlásí, že nějaké konstantní pole není označeno klíčovým slovem „final“, atd..

Nástroj FindBugs je možno ovládat buď pomocí grafického uživatelského rozhraní (GUI) nebo pomocí příkazové řádky. Po spuštění analýzy můžete v GUI pracovat s výsledky, které jsou k dispozici a např. zvýrazňovat chyby v kódu (pokud je k dispozici). Tester provádějící kontrolu kódu má možnost si k jednotlivým analýzám psát komentáře nebo listovat ve zdrojovém kódu (pokud je k dispozici). Je zde možnost jednotlivé chybové hlášení označit patřičným označením (need further study, not a bug, mostly harmless, should fix, must fix, I will fix, bad analysis, unclassified, obsolete/unset; will not fix). Označování štítkem „not a bug“ je např. výhodné k odstraňování false

alarmů. Tester může také filtrovat zobrazené chyby na základě výše uvedených označení nebo chybových klasifikací, což zpřehledňuje výsledky analýzy.

Po uvedené charakteristice *FindBugs,* přichází na řadu představení detektorů, které se dají rozdělit do dvou hlavních kategorií: *založené na inspekci kódu (visitor-based)* a *založené na grafu toku řízení CFG-based (Control Flow Graph – CFG-based)*. *FindBugs* neklade žádné striktní omezení vyplývající z tohoto rozdělení. Jedná se o pouhou možnost využití dvou existujících způsobů tvorby detektorů. Obecně ale bývají *detektory založené na inspekci* považovány za jednodušší než *CFG detektory*, které využívají komplexnější analýzy (např. založené na frameworku *dataflow analýzy*).

Všechny detektory, které jsou součástí *FindBugs* jsou standardně k nalezení v balíčku *edu.umd.cs.findbugs.detect*. Nástroj umožňuje tvorbu vlastních detektorů bez nutné modifikace samotného *FindBugs*. Aby mohl být *vlastní detektor* využit k analýze musí být umístěn jako *plugin* (jar archív s patřičnými XML soubory) do *adresáře plugin* ve *FindBugs* instalaci.

Jeden z důležitých typů detektorů využívajících *inspekční paradigma* je stavový automat nad sekvencí instrukcí určité metody. Každé volání callback metody sawOpcode() zpřístupní individuální instrukci, která předstuvuje jeden znak na vstupu konečného automatu. Takovýto konečný automat pak dostává na vstupu jednotlivé instrukce a přechází mezi svými stavy. Pokud se dostane konečný automat popisující chybový vzor do určitého koncového stavu, může to značit pravděpodobné nalezení chyby, kterou automat popisuje a je potřeba vyvolat patřičné chybové hlášení. Podstata tohoto druhu detektoru je velice jednoduchá, i tak je ovšem schopna odhalit velikou část chyb. Velkou výhodou těchto detektorů je, že jsou o poznání rychlejší než detektory založené na *CFG paradigmatu*. Tato podstatná vlastnost pak umožňuje *detektorům založených na inspekci* hledat chyby i v obrovských balících, jako je např. rt.jar v řádu sekund. Jedním z rozšíření těchto detektorů, je např. třída OpcodeStack. Tato třída umožňuje spravovat informace o operand stacku jako je zaznamenávání instrukcí (patřících do analyzované metody), které ho za běhu navštíví. [23]

V následující části věnující se CFG detektorům budu využívat následující terminologii:[2, 23]

- Analysis objects – konečný produkt analýzy, která byla v minulosti ukončena.
- ClassContext – pokrývá známá fakta o třídě (zjištěná pomocí BCEL).
- Dataflow fact – stav dataflow analýzy v určité části analýzy.
- Detector – algoritmem popsaný hledaný vzor.

Detektory založené na *CFG* (Control Flow Graph) paradigmatu využívají *CFG* reprezentaci Java metod k vykonávání sofistikovanějších analýz, než jsou schopny provést na inspekci založené detektory. *CFG* detektory obvykle přímo implementují Detector interface.

```
for each method in the class do
      reguest a CFG for the method from the ClassContext
      request one or more analysis objects on the method from the
ClassContext
      for each location in the method do
            get the dataflow facts at the location
            inspect the dataflow facts
            if a dataflow facts indicates an error then
                  report warning
            end if
      end for
end for
```

Ilustrace 8: Pseudo-kód CFG detektoru. Převzato z [23]

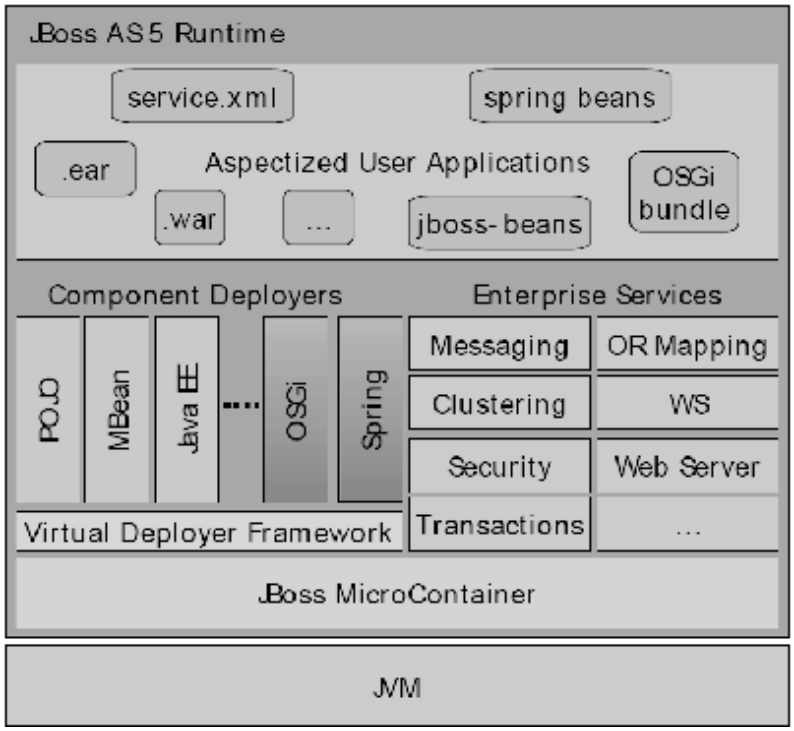
Ilustrace 8 nám ukazuje pseudo-kód metody visitClassContext(), která patří do detektorů založených na CFG analýze. Základní idea algoritmu je navštívit každou metodu analyzované třídy v kole dotazování určité informace z analysis objects. Detektor přitom iterativně prochází přes všechna místa v control flow grafu. Místem může být nazýván bod před nebo po určité instrukci. Detektor při analýze kontroluje v každém místě dataflow facts, aby objevil potenciální chybu.

CFG detektory jsou využívány k analýzám, které mohou být definovány jako „*forward or backward dataflow problem*“. Nevýhoda CFG detektorů je ta, že jsou o poznání pomalejší než detektory založené na inspekci. Je to z důvodu, že CFG detektory vytvářejí složitější reprezentaci tříd, metod a instrukcí, budují CFG a provádí dataflow analýzu, což v konečném důsledku vyžaduje více času a paměti. [2, 23]

2.7 JBoss aplikační server

V předchozích kapitole byl zmíněn nástroj využívaný k detekci chyb. Nyní bude představen software, ve kterém se budou chyby hledat. Podkapitola se tedy věnuje popisu JBoss aplikačního serveru, ve kterém se v praktické části práce budou pomocí statické analýzy hledat často se vyskytující chyby.

JBoss Aplikační server (JBossAS) je open source implementace sady Java EE služeb (jako např. EJB [49], JPA [53], JMS [57], JTS/JTA [59], JSP [58], JNDI [58], atd.) za pomoci využití na služby orientované (service-oriented) architektury. [45, 63] V podstatě se jedná o sestavu předkonfigurovaných profilů JBoss Enterprise Middleware komponent (které budou popsány níže).



Ilustrace 9: Architektura JBoss aplikačního serveru 5. Převzato z [27].

Ilustrace 9 ukazuje jak vypadá architektura JBossAS za běhu (runtime). Architektura JBossAS vychází z mikro-jádra (micro-kernel) architektury. Ta spočívá ve vytvoření mikro-jádra (viz. na ilustraci 9 JBoss MicroContainer, který bude popsán níže) s možností rozšiřování vlastností pomocí přidávání nových komponent (viz. na ilustraci 9 Component Deployers). MicroContainer má na starosti komunikaci s JVM a správu komunikace jednotlivých komponent mezi sebou. JBoss AS je schopen díky svému důmyslnému návrhu dynamické zprávy služeb (viz. na ilustraci 9 Enterprise Services) a aplikací (hot-deployment a redeployment komponent nad kompaktním jádrem – JBoss Server Spine) za běhu serveru (bez nutnosti restartu). Nasazování uživatelských aplikací je prováděno skrz překopírováni aplikace (v podobě jar, war, ejb, ear nebo xml viz. na ilustraci 9 Aspectized User Applications) do příslušného adresáře (server/jméno_konfigurace/deploy – popsáno bude níže). [24, 27]

Obnovování jednotlivých aplikací je řešena prostým přehráním aplikace a aktualizace se provede na základě časového údaje. Runtime komponenty jsou v obrázku propojeny dohromady pomocí Microcontainer se závislostmi skrz celý model komponent. Ilustrace 9 také představuje MicroContainer jako rozhraní mezi JBossAS a JVM. [24, 27]

Architektura JBossAS je v podstatě složená z mikro-jádra, JMX Mbean (technologie sloužící k správě a monitorování aplikací, systémových objektů, zařízení a servisně orientovaných služeb) serveru a sady připojitelných komponent (služeb). Mbeans dovoluje spouštět JBossAS v různých konfiguracích jako např. minimal, default, all, standard, web nebo jiné vlastní konfigurace. Díky přidávání/odebírání komponent do/z jednotlivých konfigurací má uživatel možnost si nadefinovat sadu služeb, která vyhovuje jeho potřebám, což v konečném důsledku ovlivňuje funkcionalitu a výkon. Základními konfigurace jsou [24, 27, 45]:

- *minimal* – Obsahuje minimální počet služeb pro spuštění JBoss. Spouští se logging služba, JNDI [58] server a URL deployment scanner. Konfigurace bývá používána při využití JMX/JBoss ke spouštění vlastních (uživatelských) služeb bez jiných Java EE technologií. Jedná se o pouhý server. Nezahrnuje např. EJB [49], podporu JMS [57] ani web kontejner, atd..
- *default* – Konfigurace obsahuje standardní služby využívané většinou Java EE aplikací. Nezahrnuje např. clustering službu [46], JAXR službu [63], IIOP službu [62], atd..
- *all* – Spouští všechny dostupné služby.

JBoss Aplikační server je napsán v programovacím jazykem Java, což z něj tvoří platformně nezávislou aplikaci běžící na operačních systémech podporujících Java. Tato serverová architektura je jednoduchá k použití a má vysokou flexibilitu. JBossAS má konkurenci: BEA Systems WebLogic Server a IBM WebSphere, která ovšem není tak rozšířená jak JBossAS. [74] JBossAS lze stáhnout na stránkách http://www.jboss.org/jbossas/ ve verzí 6.0.0.M3. JBossAS potřebuje ke svému běhu JDK 1.5 nebo JDK 1.6. Tato verze splňuje certifikaci Java EE5. [24, 30]

JBoss Microcontainer [45] (Dostupné z www <http://www.jboss.org/jbossmc>, citováno ke dni 17. 11. 2009) – jedná se o samostatný na aplikačním serveru nezávislý, lightweight kontejner pro správu (nasazení, konfigurace, life-cyklus) *POJOs (Plain Old Java Object) sloužící jako jádro systému*. *Microcontainer* integruje JBoss framework pro *Aspect Oriented Programming (JBoss AOP) [60]*. *JBoss Microcontainer* slouží jako náhrada za dříve využívaný JMX Microkernel [67]. *JBossAS 5* je navržen okolo rozšířeného konceptu *Virtual Deployment Framework (VDF)*, který bere orientovaný design mnoha dřívějších JBoss kontejnerů a aplikuje je do deployment vrstvy. VDF

umožňuje přizpůsobení modulů existujících komponent obsahujících *Java EE a JBoss Microcontainer* stejně jako zavedení jiných modulů jako *OSGi* [65] *a Spring* [66].

JBossAS se dále skládá z těchto komponent: [10, 24, 28, 29]

- *JBoss AOP* (Ke stažení dostupné z www <http://www.jboss.org/jbossaop/>, citováno ke dni *17. 11. 2009*) – je na Javu zaměřený framework používaný při programování zařízení integrovaných v *JBossAS* využívající nové paradigma na aspektech založeného programování (AOP) [60], které dovoluje organizovat software tak, jak to s tradičním objektovým návrhem nebylo možné.
- *ProfileService* (Ke stažení dostupné z www <*http://www.jboss.org/community/docs/DOC-11694*>*, citováno ke dni 17. 11. 2009)* – je zobecnění konfigurací JBossAS 4.x serveru, kde server konfigurace je kolekcí služeb a aplikací načtených z deploy adresáře pomocí služby – deployment scanner (MBean).
- *JBoss EJB3* (Ke stažení dostupné z www <http://www.jboss.org/jbossejb3>, citováno ke dni 17. 11. 2009) – představuje hlubokou opravu a zjednodušení Enterprise Java Beans (EJB) [49] specifikace. Cílem EJB 3.0 je zjednodušit vývoj a zaměřit se na psaní zřetelných starých Java objektů (POJOs) více než využívání EJB APIs.
- *JBoss Messaging* (Ke stažení dostupné z www <http://www.jboss.org/jbossmessaging>, citováno ke dni 17. 11. 2009) – je vysoce výkonný poskytovatel Java Messaging Service (JMS) [57] v JBoss Enterprise Middleware Stack (JEMS) vyskytující se v JBoss 5 jako standardní poskytovatel zpráv. Je to součastí páteřní infrastuktury JBoss ESB.
- *JBoss Cache* (Ke stažení dostupné z www <http://www.jboss.org/jbosscache>, citováno ke dni 17. 11. 2009) – je transakční, distribuovaná, in-memory cache využívaná mnoha službami. Poskytuje dvě funkce: tradiční stromovou strukturu založenou na cache uzlech a PojoCache, která jako in-memory pro transakční a replikační cache systém umožňuje uživatelům transparentní operace nad jednoduchými POJO bez aktivního uživatelského managementu s replikačními nebo persistenčními aspekty.
- *JBossWS* (Ke stažení dostupné z www <http://www.jboss.org/jbossws>, citováno ke dni 17. 11. 2009) – provádí integrační vrstvu pro webové služby třetích stran. Nabízí např. podporu JAX-WS (2.1) [61].
- *JBoss Transaction* (Ke stažení dostupné z www <http://www.jboss.org/jbosstm>, citováno ke dni 17. 11. 2009) – je standardní transakční manažer pro JBoss 5. JBoss Transactions se dá považovat za vůdce v distribuovaných transakcích schopného pracovat s JTA, JTS and Web Services standardy.

- *JBoss Web* (Ke stažení dostupné z www <http://www.jboss.org/jbossweb>, citováno ke dni 17. 11. 2009) – poskytuje na Tomcat [58] založený enterprise webový server vytvořený pro střední a velké aplikace. JBoss Web implementuje Servlet 2.5 [58] a JavaServer Pages 2.1 [58] specifikace, dále také zahrnuje spoustu doplňkových vlastností, které tímto tvoří stabilní základ pro vývoj a rozvoj webových aplikací společně s webovými službami.
- *JBoss Security* (Ke stažení dostupné z www <http://www.jboss.org/jbosssecurity>, citováno ke dni 17. 11. 2009) – zajišťuje bezpečnost. Nabízí knihovny zajišťující autentizaci, autorizaci, audit v běžných Java zařízeních a SPNego/Windows Desktop SSO (JBossNegotiation). Standardní model bezpečnosti je přitom postaven na bezpečnostním frameworku JAAS (Java Authentication and Authorization Service) umožňující práci v různých bezpečnostních infrastrukturách bez nutnosti změny implementace bezpečnostního správce. [51].
- *JBoss Hibernate* (Ke stažení dostupné z www <https://www.hibernate.org>, citováno ke dni 17. 11. 2009) – je výkonnou, na perzistentních objektech nebo relacích založenou dotazovací službou, která se dá označit za kritickou komponentu sady produktů JBoss Enterprise Middleware systému (JEMS). Hibernate dovoluje vytvořit stálé třídy následující objektově orientované paradigma obsahující asociace, dědění, polymorfismus, kompozice a kolekce, které uloží do databáze a zpřístupní je. Hibernate dále umožňuje pokládat dotazy prostřednictvím vlastního přenosného SQL rozšíření (HQL) stejně jako původního SQL.
- *JGroups* (Ke stažení dostupné z www <http://www.jgroups.org>, citováno ke dni 17. 11. 2009) – je soubor nástrojů pro spolehlivou multicast komunikaci. Může být použit k tvoření skupin procesů, jejichž členové si mohou navzájem posílat zprávy.
- *JBoss Remoting* (Ke stažení dostupné z www <http://www.jboss.org/jbossremoting>, citováno ke dni 17. 11. 2009) – přináší jednoduché API pro většinu síťových realizací a příbuzných služeb, které užívají výměnnou komunikaci přes síť. JBoss Remoting umožňuje vytváření synchronních a asynchronních vzdálených volání, jednocestné zasílaní zpráv, atd.
- *jBPM* (Ke stažení dostupné z www <http://www.jboss.org/jbossjbpm>, citováno ke dni 17. 11. 2009) – jedná se o rozšiřitelný, flexibilní process engine (workflow engine), který může běžet jako samostatný server nebo zapouzdřen v nějaké Java aplikaci. Podporuje spouštění procesů popsaných v jPDL, BPEL nebo Pageflow.

Mezi další technologie, které pod JBossAS najdeme můžeme zařadit: Failover (including sessions) [50], JACC (Java Authorization Contract for Containers) [52], JavaMail [53], JavaServer Faces (Mojarra) [54], JDBC [56], SAAJ (SOAP s API pro Java) [63], atd.. [24, 26, 45]

Na závěr této podkapitoly by bylo dobré podotknout, že *JBossAS* se dá považovat za hlavní komponentu nízkonákladové open source grid infrastruktury. [68, 30]

Po přečtení této podkapitoly by měl mít čtenář představu, v jakém software (JBossAS) se budou hledat chyby prostřednictvím nástroje FindBugs. Další podkapitola se bude zaobírat bug tracking systémem (JIRA), který vývojáři JBossAS využívají ke správě hlášení o chybách.

2.8 Bug tracking systém – JIRA

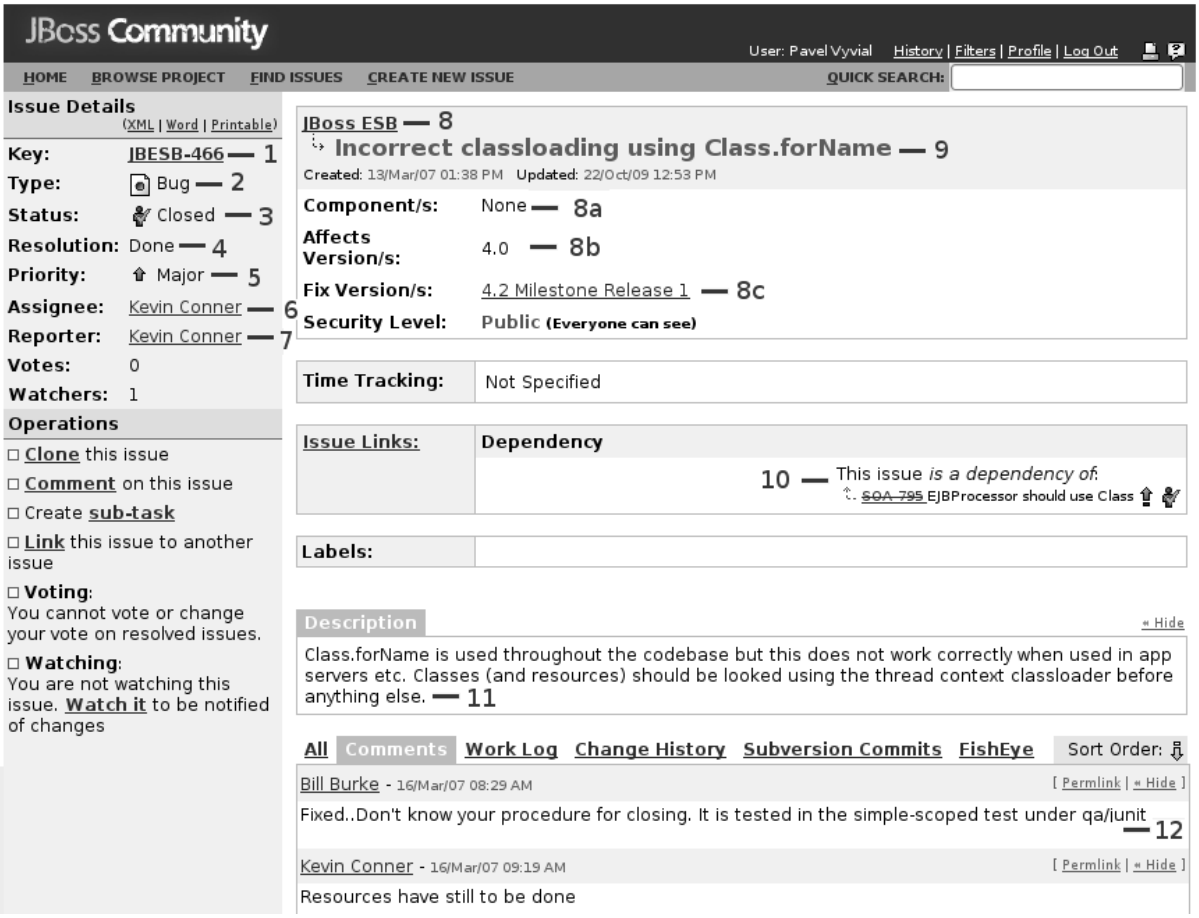
V této části bude představen bug tracking systém, který je využíván JBoss komunitou k evidenci softwarových chyb. Tento software se nazývá *JIRA* a byl vytvořen firmou Atlassian. Jedná se o nástroj napsaný v Java, využívaný k řízení a sledování úkolů a požadavků v projektu. *JIRA* se zaměřuje na dosažení očekávaného výkonu na projektu.

Bug tracking systém (issue tracking system) – je softwarová aplikace snažící se pomáhat testerům a programátorům sledovat informace o softwarových chybách v projektech. Bug tracking systémy se dají rozdělit na privátní a veřejné. Veřejné bug tracking systémy jsou typické pro open source projekty a umožňují kterémukoliv zaregistrovanému uživateli nahlásit a případně sledovat vývoj chyby. Privátní bug tracking systémy bývají typické u komerčních produktů a bývají pro veřejnost nedostupné. Nutno podotknout, že i u veřejných bug tracking systémů se můžou vyskytovat sekce pouze pro vývojáře, které obsahují např. hlášení o bezpečnostních chybách. Typicky se bug tracking systémy integrují s project management software. Bug tracking obvykle obsahuje databázi s informacemi o známých chybách. Informacemi se přitom myslí identifikace chyby, popis jak chybu vyvolat, místo kde se pravděpodobně nachází, priorita odstranění, identifikace člověka, který chybu nahlásil a člověka, který bude chybu opravovat.

Charakteristické vlastnosti a shrnutí hlavních přínosů tracking systému JIRA:

- Aktuální informace o projektu má tým dostupné přes webové rozhraní.
- Může sloužit jako průkazná historie projektové komunikace.
- Nabízí podporu projektové řízení.
- Obsahuje sledování a vyhodnocování kapacit.
- Podpora pro klientský servis a helpdesk.
- Poskytuje reporty, statistiky a historii.
- Úkoly podle priorit a termínu dokončení.
- Workflow management

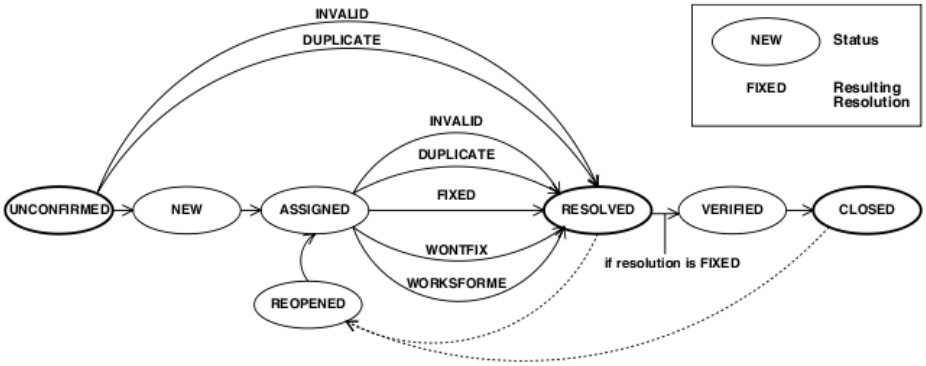
JBoss využívá *JIRA* (dostupnou z adresy https://jira.jboss.org/) ke správě chyb. Jak jsou jednotlivé chyby popsány, a co dokáže uživatel z webového rozhraní vyčíst nám ukazuje Ilustrace 10.



Ilustrace 10: Popis chyby JBESB-466 v JIRA.

Na obrázku je vidět chyba s označením JBESB-466 (v praktické části byl pro ni tvořen detektor), které představuje její unikátní identifikaci. Dále budu popisovat jednotlivé části GUI. 2-ka (v poli status) nám ukazuje na typ, který je v tomto případě „Bug“ značící, že se jedná o popis chyby. Obecně typ může nabývat jednu z těchto hodnot: „Bug, Improvement, New Feature, Task, Custom Issue“. 3-ka ukazující na status říká, jak daleko je řešení daného problému. Na obrázku je chyba označená za vyřešenou (viz. hodnota Closed). Status může nabývat těchto hodnot: „Open, In Progress, Resolved, Reopened, Closed“. Jak se status obvykle vyvíjí u chyb v JBossAS nám ukazuje Ilustrace 11. 4-ka ukazuje na Resolution, které značí vyjádření o stavu problematiky. 5-ka značí prioritu řešení chyby. V našem případě se jedná o chybu s označením „zásadní“ (Major). Priority mohou nabývat hodnot: „Blocker, Critical, Major, Minor, Trivial“. 6-ka identifikuje člověka, který dostal problém k vyřešení. 7-ka ukazuje na člověka, který chybu nahlásil. 8-ka ukazuje na projekt do kterého tato chyba spadá. 8A – identifikuje přímo komponentu a 8b – verzi ve které se chyba

vyskytla. 8c – říká ve které verzi byl problém vyřešen. 9-ka stručně charakterizuje zobrazený problém. 10-ka ukazuje na problém, který byl podobný. Na základě těchto závislostí se dá zjistit chyba, která má najevíce podobných problémů a určit tak chybu s nejčastějším výskytem. Tohoto principu bylo využito i v práci, ke získání seznamu nejčastěji se vyskytujících chyb. 11-ka značí popis chyby. Pod 12-kou jsou označeny komentáře lidí k danému problému. Celkově se jedná o přehledný popis softwarové chyby.



Ilustrace 11: Životní cyklus chyby v JBossAS. Převzato z [48]

Na Ilustraci 11 jde vidět životní cyklus chyby v bug tracking systému JIRA. Chyba je nahlášena do systému (unconfirmed). Tímto vzniklo chybové hlášení o chybě. Následně je chyba buď prohlášena za novou (new), nebo za duplikát (duplicate) již existující chyby, nebo za chybové hlášení o neexistující chybě (invalid – false alarm). V případě, že se jedná o duplikát nebo false alarm, tak se chyba považuje za vyřízenou (resolved). Následuje verifikace, že byla chyba vyřízena a pokud je vše v pořádku, tak se prohlásí chybové hlášení za uzavřené (closed) jinak se přechází k znovuotevření řešení chyby (reopened). V případě, že chyba nahlášena do systému je nová nebo znovuotevřená, přiřadí se ji vývojář, který ji má na starosti opravit (assigned). Po opravě se mění stav chybového hlášení na jeden z následujících stavů: invalid, duplicate, fixed, wontfix, worksforme. Invalid a duplicate byly již popsány. Fixed znamená, že chyba byla opravena. Wontfix znamená chybu, která nebude nikdy opravena. Worksforme značí, že všechny pokusy o reprodukci chyby skončili neúspěšně a na chybě se bude pracovat po dodání více informací. Následuje stav resolved, atd.

3 Hledané chyby

V této kapitole jsou popsány chybové vzory, které jsem odvodil na základě informací získaných z bug tracking systému a komunikace s Ing. Jiřím Pechancem. Pro tyto chybové vzory bude v kapitole 4 popsán návrh a implementace detektorů.

3.1 Použití zakázané statické metody

První ze zvolených často se vyskytujících chyb měla následující popis: V projektu JBoss ESB se nesmí v žádné třídě vyskytovat volání statické metody `Class.forName()`. Pokud je tomu jinak, tak je třeba nahlásit potencionální chybu. Chyba byla v JIRA nahlášená pod těmito ID: SOA-795, JBESB-536, JBESB-2044, JBESB-466.

Chybový vzor – detektor hlásí chybu když je splněn 1. – 3. bod:

1. Zjisti, zde analyzovaná třída má prefix „org.jboss“.
2. Zjisti, zda se dále za prefixem v cestě objevuje „esb“.
3. Zjisti zda metoda obsahuje byte-kódovou instrukci `invokestatic` volající metodu `Class.forName()`.

3.2 Využívání CourierFactory a interface DeliverOnlyCouriers a PickupOnlyCouriers

Druhá chyba se dá popsat následujícím způsobem: V projektu JBoss ESB je třeba odchytit použití `org.jboss.soa.esb.couriers.CourierFactory` v případě, že není voláno z *bezpečných_tříd*:

- `org.jboss.soa.esb.client.ServiceInvoker`,
- `org.jboss.soa.esb.listeners.message.Invoker`,
- `org.jboss.soa.esb.listeners.message.ActionProcessingPipeline`,
- `org.jboss.soa.esb.listeners.message.MessageAwareListener`,
- `org.jboss.internal.soa.esb.couriers.TwoWayCourierImpl`,
- `org.jboss.soa.esb.couriers.CourierCollection`,
- `org.jboss.soa.esb.couriers.CourierUtil`.

Zároveň by se nikde jinde nemělo objevit použití libovolné třídy, která implementuje jeden z těchto interface:

- `org.jboss.internal.soa.esb.couriers.DeliverOnlyCouriers`,
- `org.jboss.internal.soa.esb.couriers.PickupOnlyCouriers`.

Chybový vzor – detektor hlásí chybu když nastane jeden z následujících bodů:

1. Pokud se analyzuje třída, která nepatří do bezpečných_tříd a je nalezena invoke instrukce volající některou z metod třídy org.jboss.esb.couriers.CourierFactory.
2. Pokud se analyzuje třída, která nepatří do bezpečných_tříd a je nalezeno použití metod třídy jejíž interface je org.jboss.internal.soa.esb.couriers. DeliverOnlyCouriers.
3. Pokud se analyzuje třída, která nepatří do bezpečných_tříd a je nalezeno použití metod třídy jejíž interface je org.jboss.internal.soa.esb.couriers. PickupOnlyCouriers.

3.3 Testování getPayLoad(Message)

Třetí chyba se dá popsat následujícím způsobem: Je třeba odchytit všechny třídy, které přímo nebo nepřímo implementují interface org.jboss.soa.esb.actions.ActionPipelineProcessor. Každá takováto třída bude obsahovat nejméně jednu metodu, která bude public s návratovým typem org.jboss.soa.esb.message.Message (dále jen Message) a parametrem Message. V každé takovéto metodě (např. public Message process(Message m)) se bude hledat následující situace. V případě, že je volána nějaká metoda od objektu typu Message, který byl výše definované metodě předán jako parametr, musí mu předcházet volání metody getPayLoad(Message) na objektu třídy org.jboss.soa.esb.message.MessagePayloadProxy. Pokud toto volání metody getPayLoad(Message) volání nějaké metody od Message nepředchází, je třeba nahlásit chybu.

Chybový vzor pro tuto chybu se dá popsat tímto způsobem: V metodách s návratovým typem org.jboss.soa.esb.message.Message (dále jen Message), které se nachází ve třídě implementující interface org.jboss.soa.esb.actions.ActionPipelineProcessor je třeba si ukládat informaci o nalezení volání getPayLoad(Message) na objektu třídy org.jboss.soa.esb.message.MessagePayloadProxy. Pokud je pak v metodě nalezeno volání některé metody od objektu typu Message a volání getPayLoad(Message) mu nepředchází, jedná se o potenciální chybu.

3.4 Testování objektů

Čtvrtá chyba je v JIRA k nalezení pod ID: JBRULES-1435, JBRULES-1423, JBRULES-1429, JBRULES-142. Podstata problému výše uvedených chyb, jež mohou vést k NPE (nullPointerException) je v tom, že programátor špatně kontroluje shodnost dvou objektů. O dvou objektech se dá tvrdit, že jsou shodné v případě kdy nabývají oba „null“ (neinicializované) hodnoty nebo oba ukazují na tentýž objekt. Jak špatné testování objektů může vypadat ukazuje popis chyby JBRULES-1429 z verze 4.0.4 (ve verzi 4.0.3 se chyba nevyskytovala) metody org.drools.base.evaluators.ObjectFactory$ObjectEqualsComparator#equals začínající:

```
if ( arg0 == null ) {
        return arg1 == null;
}
if( arg1 != null && arg1 instanceof ShadowProxy ) {
        return arg1.equals( arg0 );
}
```

Problém se může v tomto případě vyskytnout, když arg0 je různý od null a arg1 nabývá null. Je to z důvodů, že výše uvedené testování patřící pod equals metodu nevrátí v tomto případě hodnotu false.

Správně by testování mělo vypadat např. takto:

```
if ( arg0 == null ) {
        return arg1 == null;
}
if (arg1 == null) {
        return false;
}
if (arg1 instanceof ShadowProxy ) {
        return arg1.equals( arg0 );
}
```

Chybový vzor – detektor nahlásí chybu když je splněno: 1. Analyzovaná metoda má v názvu slovo „equals“. 2. Analyzovaná metoda má dva parametry (arg0, arg1). 3. Narazí se na situaci, že by analyzovaná metoda testující shodu dvou objektů navracela nekorektní hodnotu nebo by se snažila volat `arg1.equals(arg0)` s argumentem, který má hodnotu null.

3.5 Detekce neočekávané změny atributů

Třídy implementující `org.jboss.soa.esb.actions.ActionPipelineProcessor` se v systému používají pouze v jedné instanci (singleton design pattern). Tato instance může být současně využívána více vlákny. Pátý detektor by měl identifikovat nesynchronizovanou změnu atributů třídy, která je provedena mimo konstruktor nebo metodu `initialise`. Změna atributů mimo konstruktor a metodu `initialise` je nebezpečná pokud není provedena v synchronizovaném bloku nebo na atributech typu `java.util.concurrent.*`. Pomocné metody, které jsou volány z `initialise` nebo konstruktoru mohou atributy měnit bez vyžadované synchronizace

Chybový vzor této chyby se dá popsat jako: Pokud se při prohledávání byte-kódu narazí na nesynchronizovanou změnu atributů třídy implementující `ActionPipelineProcessor`, je třeba nahlásit chybu.

4 Návrh a implementace detektorů

Kapitola popisuje návrhy a implementace detektorů chyb uvedených v předešlé kapitole. Každá podkapitola se věnuje jednomu konkrétnímu detektoru. V podkapitolách je k nalezení stručný popis detektoru a jeho návrh včetně diagramu tříd a implementace společně s diagramy aktivit popisující funkčnost daného detektoru.

Z důvodu, že všechny detektory pracují s třídami ClassContext a BugReporter, budou zde stručně představeny. ClassContext představuje jednotlivé analyzované třídy, které jsou detektoru předávány jádrem nástroje FindBugs prostřednictvím metody visitClassContext(). Druhou třídou využívanou při analýze je BugReporter sloužící k hlášení nalezených chyb.

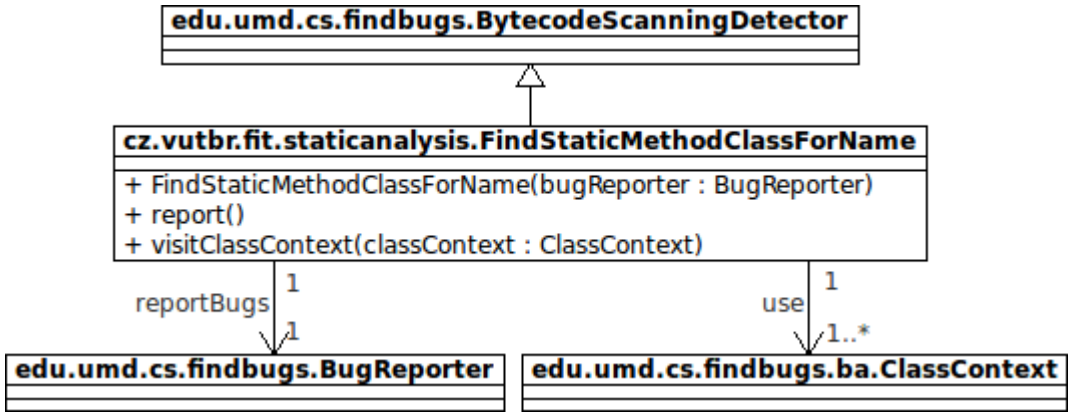
Proces analýzy všech detektorů začíná stejně a to načtením vstupních parametrů detektoru. Parametry se zadávají jako systémové proměnné Javy. V případě nenastavení vstupních parametrů, pracuje detektor ve standardním (implicitním) režimu. V opačném případě přizpůsobuje analýzu vstupním parametrům. Parametry, které je možné využít ke konfiguraci detektorů zobrazuje tabulka 21 nacházející se v přílohách pod Dodatkem B. Popis parametrů je k nalezení v implementační části jednotlivých detektorů. Jakmile jsou načteny parametry začíná proces samotné analýzy. Z balíku tříd, který byl vložen na vstup nástroje FindBugs se postupně analyzují všechny třídy. Analýza probíhá tak, že FindBugs postupně pro všechny třídy z analyzovaného balíku zavolá metodu visitClassContext() patřící do detektorů. Poté FindBugs zavolá metodu report() z detektorů. Nakonec se nahlásí chyby nasbírané v BugReporteru za běhu detektorů. Takto by se dala charakterizovat komunikace FindBugsu s detektory. Následují podkapitoly popisující návrh a implementaci jednotlivých detektorů.

4.1 Detekce použití zakázané statické metody

V této části bude popsán návrh a implementace 1. detektoru – *FindStaticMethodClassForName.* Byl použit pro detekci chyby popsané v kapitole 3.1. Detektor umožňuje hledat použití statické metody v třídách s patřičnou cestou (classpath). Detektor v implicitním (standardním) režimu hledá statické volání Class.forName() ve třídách patřících do projektu ESB. Detektor byl mimo to rozšířen, aby byl schopen v určených třídách hledat výskyt volání metody definované prostřednictvím systémových proměnných. Popis implicitní a explicitní funkčnosti detektoru bude popsán v části – 4.1.2 zabývající se implementací detektoru. Při implicitní detekci se postupuje tak, že se detektor zaměří na prohledávání zajímavých tříd (patřících do projektu ESB), ve kterých hledá instrukci `invokestatic` s parametrem `Class.forName()`. Pokud je taková instrukce nalezena, nahlásí se chyba. V příloze je v tabulce 16 k nalezení výsledky testu provedeného nad testovací třídou zobrazenou na Ilustraci 28.

4.1.1 Návrh

Detektor schopný hledat statické volání předem definované metody dědí svou funkčnost z třídy BytecodeScanningDetector. Tuto závislost nám ukazuje diagram tříd zobrazený na Ilustraci 12. Na obrázku dále vidíme, že celá funkčnost tohoto detektoru je zapouzdřena ve třídě FindStaticMethodClassForName. Tato třída k práci ještě využívá další dvě třídy (ClassContext, BugReporter zmíněné v kapitole 4).



Ilustrace 12: Diagram tříd 1. detektoru – FindStaticMethodClassForName.

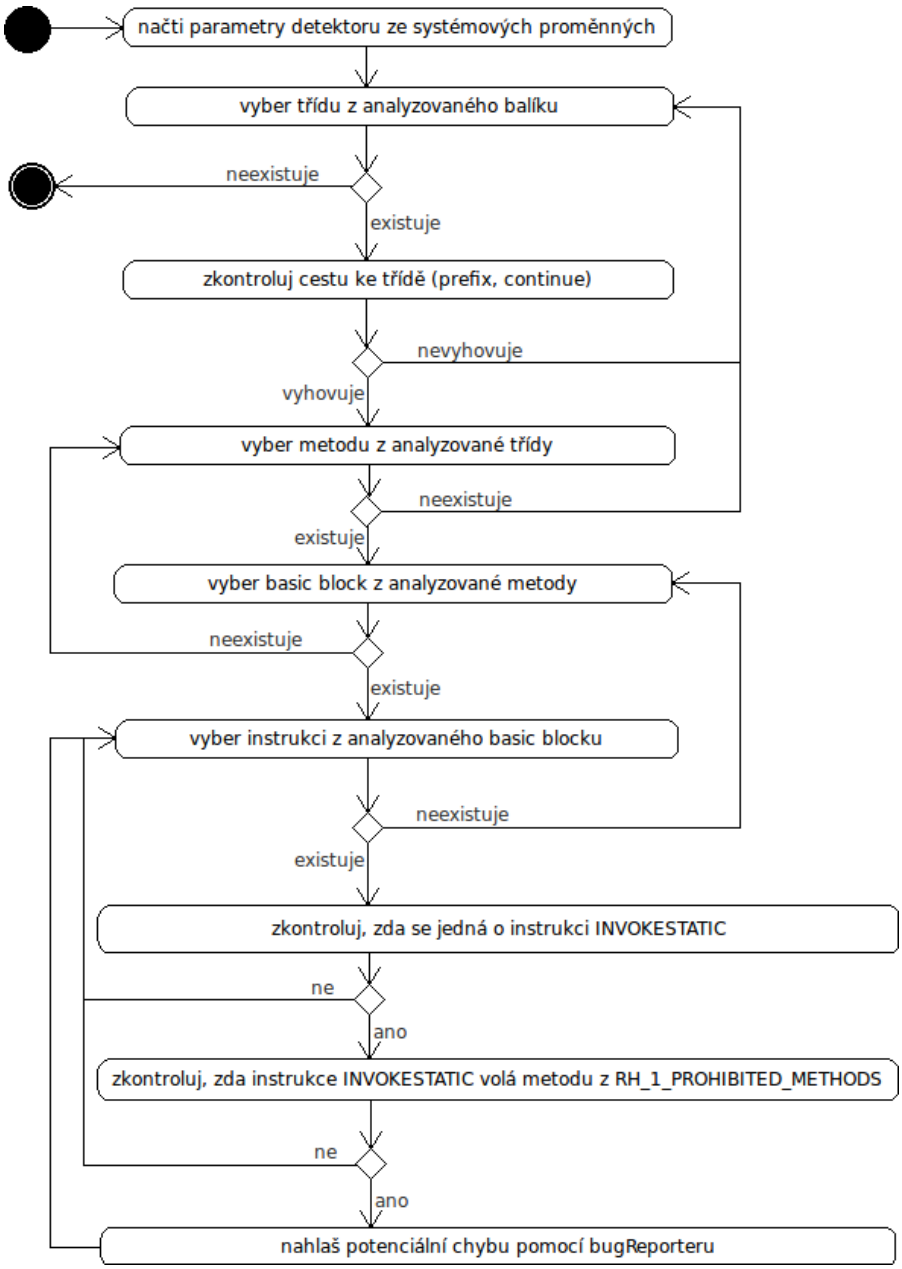
4.1.2 Implementace

Parametry, které je možné využít ke konfiguraci detektoru jsou následující:

- *RH_1_DEFINE_USE* – přepíná detektor mezi implicitním (neparametrizovaném)/explicitním (parametrizovaném) použitím. Explicitní použití detektoru je zaručeno nastavením tohoto parametru na hodnotu „1“. Implicitního použití detektoru lze docílit při zadání hodnoty „0“, nedefinováním nebo nekorektním definováním tohoto parametru.
- *RH_1_CLASSPATH_PREFIX* – představuje řetězec, kterým musí začínat cesta ke třídě ve které se má hledat nechtěné volání statické metody. Při implicitním použití detektoru nabývá hodnoty „*/org/jboss/*“. Když tento parametr není nastaven u explicitního použití, tak se bere jako by byl nastaven na prázdný řetězec. V tomto případě nejsou analyzované třídy filtrovány na základě prefixu cesty.
- *RH_1_CLASSPATH_CONTINUE* – představuje řetězec, který musí následovat někde za prefixem v cestě třídy. Při implicitním použití detektoru nabývá hodnoty „*/esb/*“, který filtruje hledání na třídy patřící do ESB projektu. Když tento parametr není nastaven u explicitního použití, tak se bere jako by byl nastaven na prázdný řetězec. V tomto případě nejsou analyzované třídy filtrovány na základě toho co následuje za prefixem.

- *RH_1_PROHIBITED_METHODS* – představuje množinu statických metod, které se mají hledat. Metody musí být definovány ve formátu (uvedeném v kapitole 2.1.1) využívaném k popisu signatury cílů byte-kódových `invoke` instrukcí a odděleny znakem „|“. Při standardním využití detektoru obsahuje tato množina identifikaci následujících metod:
  - *java/lang/Class.forName:(Ljava/lang/String;)Ljava/lang/Class;*
  - *java/lang/Class.forName:(Ljava/lang/String;ZLjava/lang/Classloader;)Ljava/lang/Class;*

Po načtení parametrů (viz kapitole 4) se přechází k analýze popsané Ilustrací 13.



Ilustrace 13: Diagram aktivit 1. detektoru.

Před prohledáváním metod analyzované třídy se zkontroluje, zda cesta třídy začíná RH_1_CLASSPATH_PREFIX, po kterém v cestě následuje RH_1_CLASSPATH_CONTINUE. V případě, že není shoda nalezena, tak se třída přeskakuje. V implicitním režimu detektoru, kde se RH_1_CLASSPATH_PREFIX=/org/jboss/ a RH_1_CLASSPATH_CONTINUE=/esb/ se bude pokračovat v analýze metod třídy: org.jboss.internal.soa.esb.couriers.Test a naopak třídy: org.jboss.internal.soa.couriers.testCases.Test2 nebo internal.soa.esb.couriers.Test3 budou přeskakovány, protože jejich cesta nevyhovuje definovaným parametrům. Od tříd, které tímto testem projdou jsou postupně prohledány jednotlivé metody. Při prohledávání se testují všechny instrukce od basic blocků patřících metodě na to zda se jedná o instrukci invokestatic. Pokud se nějaká taková instrukce najde, tak je zkontrolováno zda se její cíl rovná některé z nechtěných metod uložených v množině RH_1_PROHIBITED_METHODS. V případě nalezení shody je prostřednictvím BugReporteru nahlášena potenciální chyba.

Chybové hlášení o potenciální chybě nalezené 1. detektorem se hlásí na standardní výstup ve formátu, který může přiblížit následující příklad chybového hlášení: „H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.actions.EJBProcessor.process(Message) At EJBProcessor.java:[lines 96-121]“. Hlášení nám říká, že první detektor označený jako „H C PV“ nalezl chybu popsanou jako „Found Prohibited method call (Class.forName())“ v metodě EJBProcessor.process(Message) nachazející se v souboru EJBProcessor.java na řádcích 96-121.

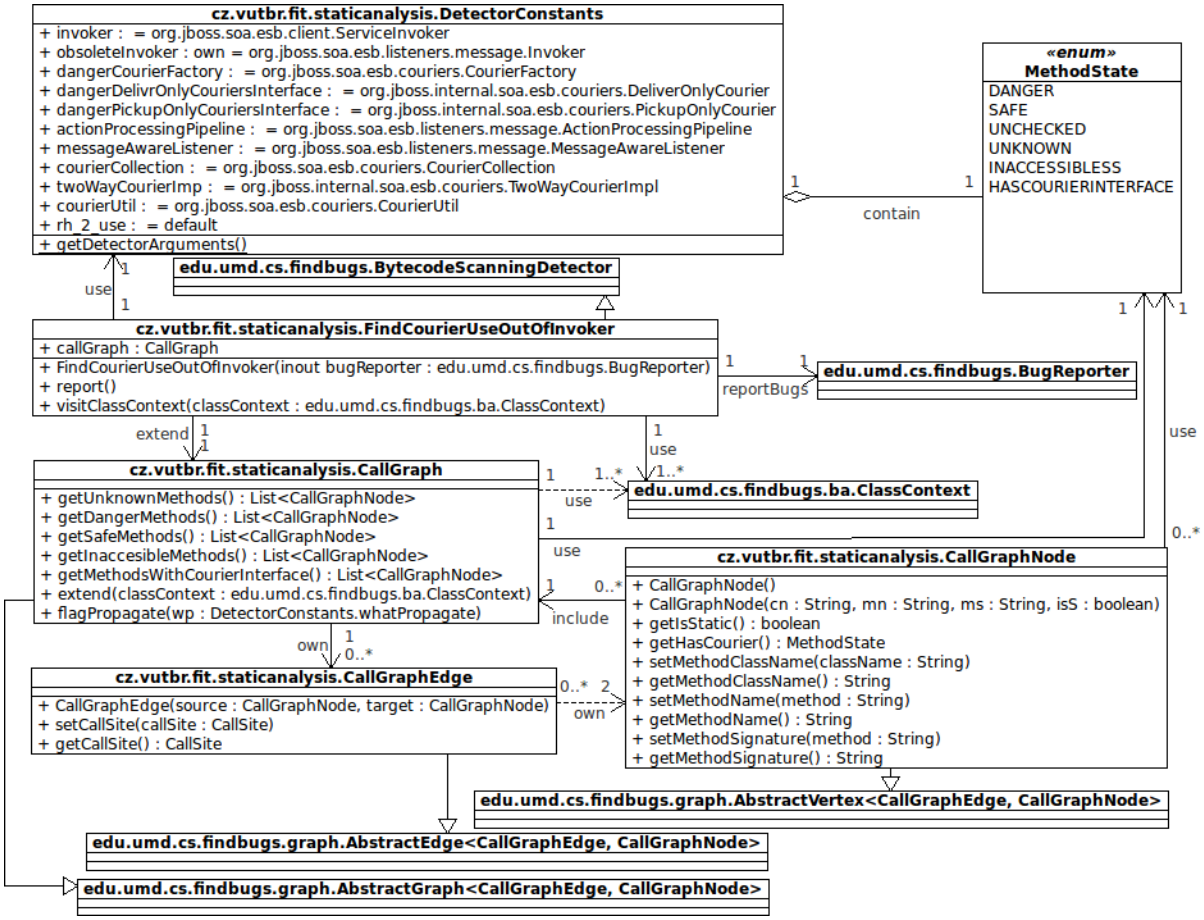
4.2 Detekce nekorektního využívání třídy a interface

V této části bude popsán návrh a implementace 2. detektoru – *FindCourierUseOutOfInvoker*. Byl použit pro detekci chyby popsané v kapitole 3.2. Detektor se snaží hledat použití nebezpečných tříd mimo třídy bezpečné. Mezi bezpečné třídy patří třídy určené výčtem. Nebezpečné třídy jsou určeny výčtem nebo tím, že implementují některé z nebezpečných rozhraní (určené výčtem). Ve standardním (implicitním) režimu je úkolem detektoru hledání použití (volání metod) třídy CourierFactory a tříd implementujících DeliverOnlyCouriers nebo PickupOnlyCouriers mimo bezpčné třídy uvedené v kapitole 3.2.

Detektor byl mimo to rozšířen, aby byl schopen najít uživatelem definované využití tříd mimo zadanou třídu. Popis implicitní (standardní) a explicitní (rozšířené) funkčnosti detektoru bude popsán v části – 4.2.2 zabývající se implementací detektoru. V tabulce 17 nacházející se v přílohách jsou k nalezení výsledky testů prováděných s detektorem.

4.2.1 Návrh

Detektor FindCourierUseOutOfInvoker je dědí svou funkcionalitu z třídy BytecodeScanningDetector. Tuto závislost nám ukazuje diagram tříd zobrazený na Ilustraci 14.

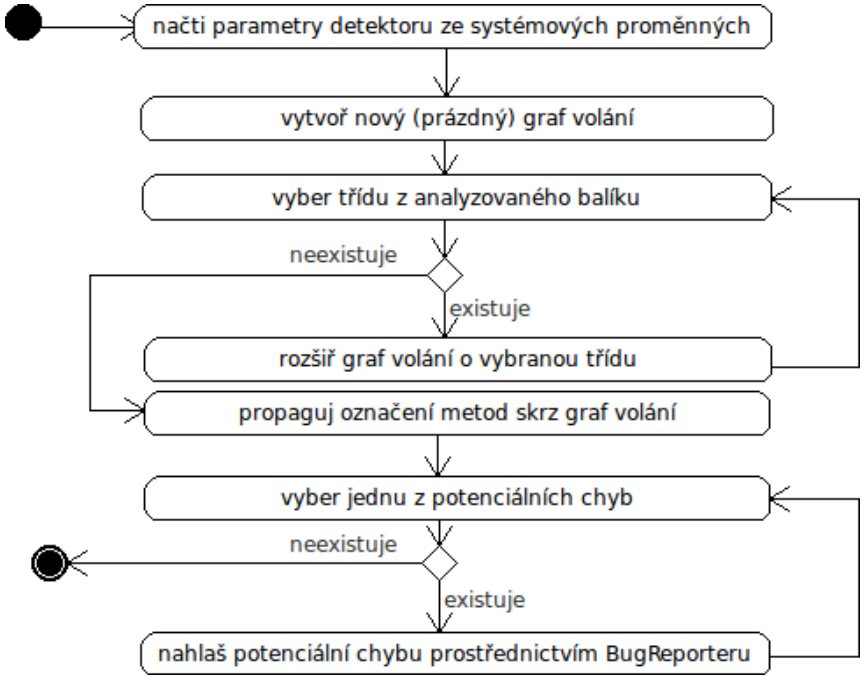


Ilustrace 14: Diagram tříd 2. detektoru – FindCourierUseOutOfInvoker.

Většina funkčnosti 2. detektoru je zapouzdřena ve třídách FindCourierUseOutOfInvoker a CallGraph. Třída FindCourierUseOutOfInvoker využívá k práci čtyři třídy. První a druhá třída (ClassContext a BugReporter) byly popsány v kapitole 4. Třetí třídou je DetectorConstants zajišťující načtení parametrů (metodou getDetectorArguments()) a zprostředkování veškerých konstant využívaných při analýze. Čtvrtou třídou je CallGraph, která na základě ClassContextu získaného (jako argument metody extend()) od FindCourierUseOutOfInvoker rozšiřuje graf volání. Ve třídě CallGraph současně probíhá i samotná detekce potenciálních chyb, které pak třída FindCourierUseOutOfInvoker (získá voláním metody getDangerMethods()) a prostřednictvím třídy BugReporter nechá zobrazit uživateli. Na obrázku jde vidět, že třída CallGraph je poděděna z AbstractGraph<CallGraphEdge, CallGraphNode> a pracuje s třídami CallGrapNode (poděděnou z AbstractVertex<CallGraphEdge, CallGraphNode>) a CallGraphEdge .

4.2.2 Implementace

Implementace se dá rozdělit do tří částí. První částí je práce třídy *FindCourierUseOutOfInvoker* zobrazená na Ilustraci 15. Druhá část je zobrazena na Ilustraci 15 jako „*rozšiř graf volání o vybranou třídu*“ funkčně ji má na starosti metoda *CallGraph.extend()* a podrobněji ji zobrazuje Ilustrace 17. Třetí část je zobrazena na Ilustraci 15 jako „propaguj označení metod skrz graf volání“ na starosti ji má metoda *CallGraph.flagPropagate()* a podrobněji ji zobrazuje Ilustrace 16.



Ilustrace 15: Diagram aktivit – FindCourierUseOutOfInvoker.

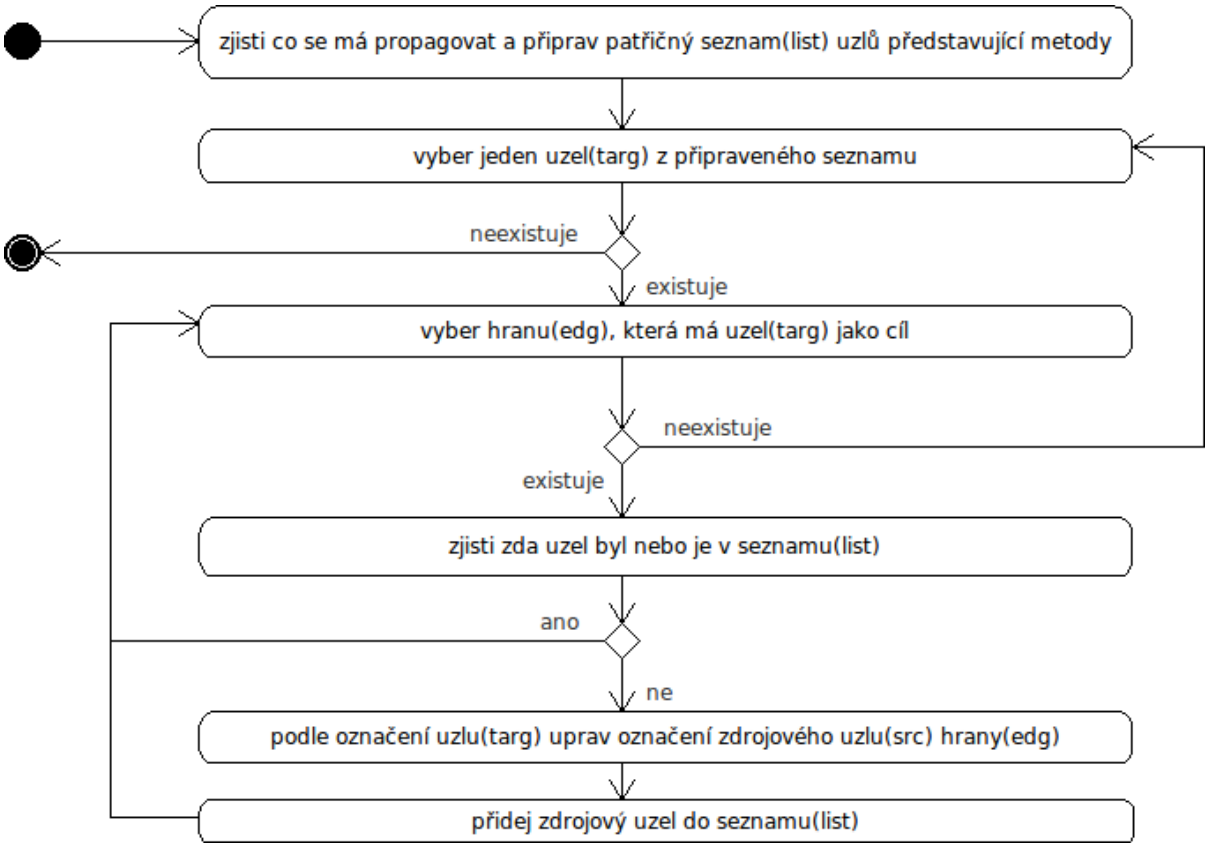
Parametry, které je možné využít ke konfiguraci detektoru jsou následující:

- *RH_2_DEFINE_USE* – přepíná detektor mezi implicitním (neparametrizovaném)/explicitním (parametrizovaném) použitím. Explicitní použití detektoru je zaručeno nastavením tohoto parametru na hodnotu „`1`“. Implicitního použití detektoru lze docílit při zadání hodnoty „`0`“, nedefinováním nebo nekorektním definováním tohoto parametru.
- *RH_2_SAFE_CLASSES* – představuje množinu tříd, ve kterých je dovoleno používat třídy, jejichž použití je jinde považováno za potencionální chybu. Třída se zadává spolu s její cestou oddělenou tečkami nebo je možné použít formát definovaný v kapitole 2.1.1. Jednotlivé třídy jsou od sebe odděleny znakem „|“. V implicitním režimu detektoru tato množina obsahuje následující třídy:
  - *org.jboss.soa.esb.client.ServiceInvoker,*
  - *org.jboss.soa.esb.listeners.message.Invoker,*

    - *org.jboss.soa.esb.listeners.message.ActionProcessingPipeline,*
    - *org.jboss.soa.esb.listeners.message.MessageAwareListener,*
    - *org.jboss.internal.soa.esb.couriers.TwoWayCourierImpl,*
    - *org.jboss.soa.esb.couriers.CourierCollection,*
    - *org.jboss.soa.esb.couriers.CourierUtil.*
- *RH_2_CATCH_CLASSES* – definuje množinu tříd jejíž použití se má odchytávat v případě, že se nejedná o použití v RH_2_SAFE_CLASSES. Při implicitním použití detektoru je tento parametr nastaven na „*org.jboss.soa.esb.couriers.CourierFactory*“. Třídy se zadávají v podobném formátu jako u předchozího parametru.
- *RH_2_CATCH_CLASSES_WITH_INTERFACES* – definuje množinu rozhraní. Pokud třída implementuje některé z těchto rozhraní, tak tato třída může být využita pouze ve třídách z *RH_2_SAFE_CLASSES*. V implicitním režimu detektoru tato množina obsahuje rozhraní:
    - *org.jboss.internal.soa.esb.couriers.DeliverOnlyCourier,*
    - *org.jboss.internal.soa.esb.couriers.PickupOnlyCourier.*

  Pokud není parametr nastaven v explicitním režimu, tak je nastaven na prázdnou množinu.

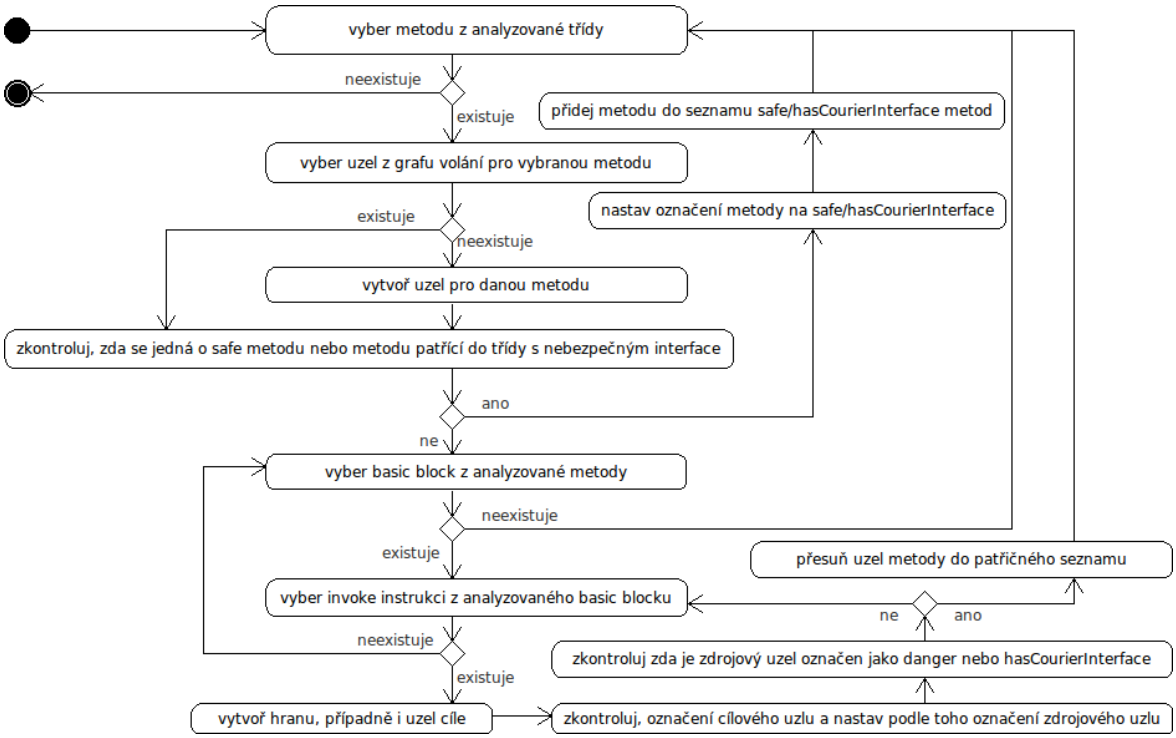
Po načtení parametrů se vytvoří a inicializuje nový (prázdný) CallGraph. Z balíku tříd, který byl předhozen na vstup nástroje FindBugs se postupně analyzují všechny třídy. Analýza pokračuje, dokud neprohledá všechny třídy. Prohledávání probíhá tak, že FindBugs pro každou třídu zavolá metoda visitClassContext(), která analýzu předá prostřednictvím metody extend() třídě CallGraph. Funkce této metody bude popsána v níže. Třída CallGraph v takto získaných třídách rozhodne, zda jsou její metody bezpečné. *Nebezpečné metody* si CallGraph uloží do seznamu List<CallGraphNode>. Jakmile jsou zanalyzovány všechny třídy (byly předány grafu volání prostřednictvím metody extend()) zavolá se metoda report(), která spustí propagaci označení metod (Ilustrace 16) pomocí grafu volání prostřednictvím metody flagPropagate(). Po propagaci si metoda report() vyžádá pomocí metody getDangerMethods() nebezpečné metody. V těchto metodách jsou pak s využitím třídy BugReporter nahlášeny potenciální chyby. Chybové hlášení o potenciální chybě nalezené 2. detektorem se hlásí na standardní výstup ve formátu, který může přiblížit následující příklad chybového hlášení: „H C PV2: Found prohibited method call (courier out of invoker) in testCases.TestRH2.firstMethod() At TestRH2.java:[lines 12-13]“, kde „H C PV2“ nám identifikuje hlášení 2. detektoru nasledované popisem a pozicí chyby.



Ilustrace 16: Diagram aktivit – propagace označení metod – CallGraph.flagPropagate().

Ilustrace 16 popisuje funkci metody CallGraph.flagPropagate(). Tato metoda má za úkol v CallGraphu propagovat označení cílových uzlů zdrojovým uzlům. Je využita v metodě FindCourierUseOutOfInvoker.report() k propagaci označení nebezpečných metod nebo metod patřících do tříd s nebezpečným rozhraním (RH_2_CATCH_CLASSES_WITH_INTERFACES). Metoda report() nejdřív propaguje označení (MethodState.HASCOURIERINTERFACE) patřící třídám s nebezpečným rozhraním a až poté propagaci označení nebezpečných metod (MethodState.DANGER). Popis propagace byl obecně zmíněn v kapitole 2.2.2.

Samotná propagace funguje tak, že podle propagovaného označení se načte patřičný seznam (*list*) uzlů metod, které toto označení již mají. Postupně se vybírají (každý jednou) jednotlivé uzly (*target*) ze seznamu (*list*). Pro target se následně zjistí všechny hrany, kde je cílovým uzlem. V každé hraně se zjistí zda zdrojový uzel (*source*) již byl nebo je v seznamu list. Pokud ano, tak se přechází k další hraně. V případě, že uzel source není v seznamu list, přeznačí se mu označení. Když se propaguje označení o nebezpečném rozhraní, tak zdrojový uzel získává označení, že je nebezpečný. Při propagaci označení nebezpečné metody získává zdrojový uzel stejné označení jako cílový uzel. Následuje přidání zdrojového uzlu do seznamu list a přechod k další hraně. Celá propagace končí, když už není označení kam propagovat.



Ilustrace 17: Diagram aktivit – rozšiřování grafu volání – CallGraph.extend().

Ilustrace 17 popisuje funkcionalitu bloku „rozšiř graf volání o vybranou třídu“ z Ilustrace 15. Metoda extend() dostane ClassContext reprezentující třídu, o kterou má být rozšířen graf volání. CallGraph je postupně rozšířen o všechny její metody následujícím způsobem. Zjistí se zda pro metodu třídy existuje uzel. Pokud neexistuje, tak se vytvoří a vloží se do seznamu bezpečných metod (safe-list). Dále se zkontroluje, zda se jedná o metodu patřící do RH_2_SAFE_CLASSES. Pokud ano, tak je automaticky tato metoda bezpečná, takže se značení uzlu metody přeznačí na SAFE. Uzel metody se vloží do safe-list. V opačném případě se testuje, zda metoda patří do třídy implementující nebo dědící jedno z nebezpečných rozhraní (RH_2_CATCH_CLASSES_WITH_INTERFACES). Pokud ano označí se uzel metody jako HASCOURIERINTERFACE a vloží se do seznamu s metodami patřícími tříd implementujících nebezpečné rozhraní (hasCourierInterface-list). V opačném případě se postupně prohledají všechny instrukce metody. Analýza se zaměřuje na invoke instrukce. Pokud je nějaká invoke instrukce nalezena a ještě pro ni neexistuje hrana tak se vytvoří. V případě, že by zatím neexistoval cílový uzel hrany, je také vytvořen. Následuje kontrola, zda označení cílového uzlu je DANGER. Pokud ano, tak je zdrojový uzel označen jako DANGER a přesune se do seznamu nebezpečných metod (danger-list). V opačném případě je testováno, zda cílový uzel má označení HASCOURIERINTERFACE. Pokud ano, tak je zdrojový uzel označen jako DANGER a přesunut do danger-list. Pokud cílový uzel nemá označení HASCOURIERINTERFACE ani DANGER, tak se pokračuje v hledání dalších invoke instrukcí a označení uzlu se nemění. Pokud

není nalezena žádná invoke instrukce volající uzel s označením HASCOURIERINTERFACE nebo DANGER, zůstane označení uzlu SAFE a může ho změnit už jen metoda CallGraph.flagPropagate().

4.3 Detekce nekorektní inicializace při práci s objekty

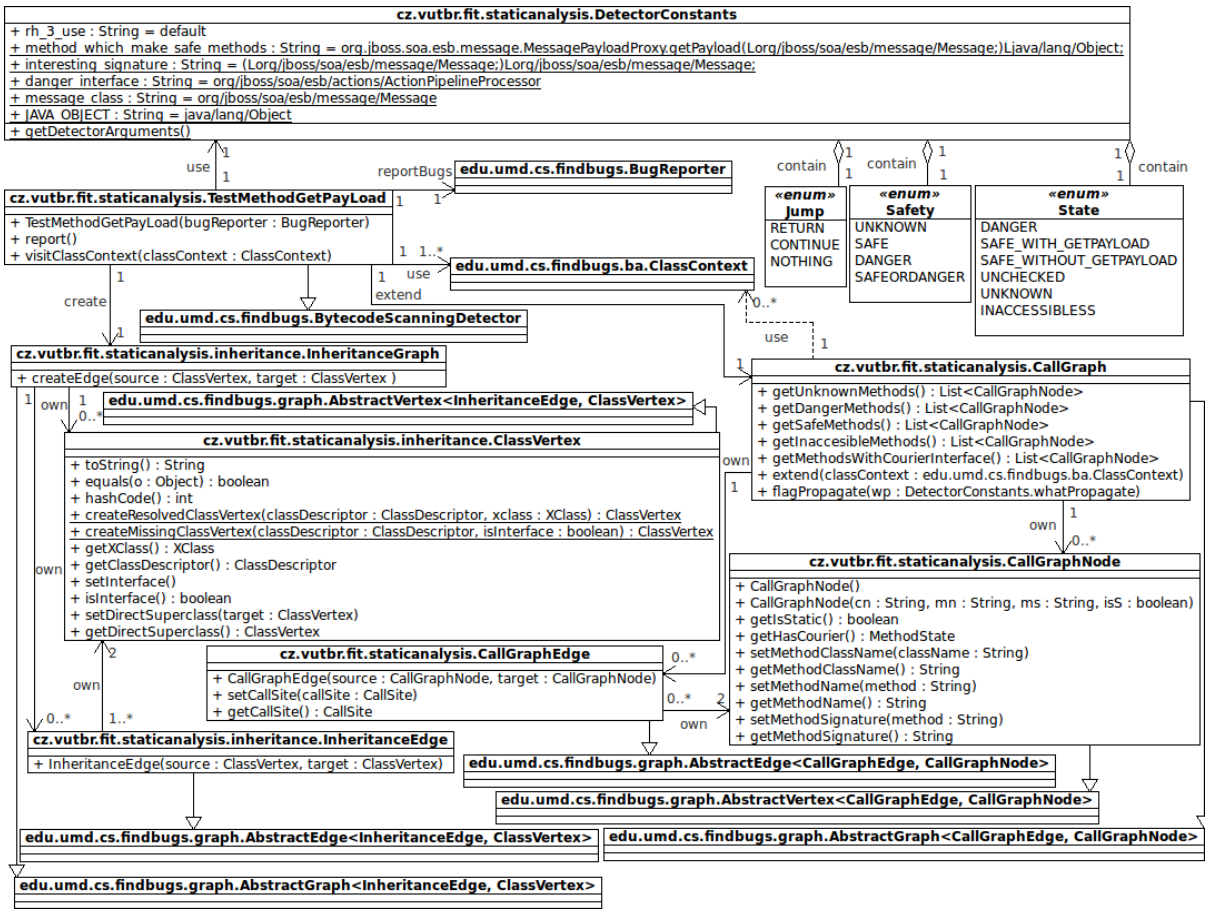
V této části bude popsán návrh a implementace 3. detektoru – *TestMethodGetPayLoad*. Detektor byl použit pro detekci chyby popsané v kapitole 3.3. Detektor se snaží v metodách s určitou signaturou patřící do třídy implementující definované rozhraní hledat situaci, kdy volání metod nad objektem vstupujícím jako argument do analyzované metody nepředchází volání inicializační metody na patřičné třídě. Ve standardním (implicitním) režimu je úkolem detektoru analýza metod s návratovým typem `Message`, argumentem typu `Message` patřících do třídy s interface `ActionPipelineProcessor`. Konkrétně se v takovýchto metodách hledá, zda před každým voláním metody typu `Message` existuje volání typu `getPayLoad(Message)` na objektu třídy *MessagePayloadProxy*. Pokud je nalezeno volání metody objektu `Message` bez předchozího volání `getPayLoad(Message)`, je to považováno za potencionální chybu, která je nahlášena pomocí BugReporteru.

Detektor byl mimo to rozšířen, aby si uživatel mohl zvolit metody, které mají být prohledány. Je to možné díky možnosti definovat si rozhraní třídy a signatury metod. Současně si lze i zvolit jaká metoda se má v metodách objevovat před použitím uživatelem definovaného objektu (argumentu metody). Nastavení detektoru je proveditelné skrz systémové proměnné. Popis implicitní a explicitní funkčnosti detektoru bude popsán v části – 4.3.2 zabývající se implementací detektoru. V tabulce 18 nacházející se v přílohách jsou k nalezení výsledky testů prováděných s detektorem.

4.3.1 Návrh

Detektor TestMethodGetPayLoad je dědí z třídy BytecodeScanningDetector. Tuto závislost nám ukazuje diagram tříd zobrazený na Ilustraci 18. Většina funkčnosti 3. detektoru je zapouzdřena ve třídách TestMethodGetPayLoad a CallGraph. Třída TestMethodGetPayLoad využívá k práci pět tříd. Jádro analýzy je v metodě report() třídy TestMethodGetPayLoad. Tato metoda je volána na konci analýzy, když je vytvořen graf volání a graf dědičnosti pro všechny analyzované třídy. První a druhá třída (ClassContext a BugReporter) byly popsány v kapitole 4. Třetí třídou je DetectorConstants zajišťující načtení parametrů (metodou getDetectorArguments()) a zprostředkování veškerých konstant využívaných při analýze. Čtvrtou třídou je CallGraph, která na základě ClassContextu získaného (jako argument metody extend()) od TestMethodGetPayLoad rozšiřuje graf volání. Třída

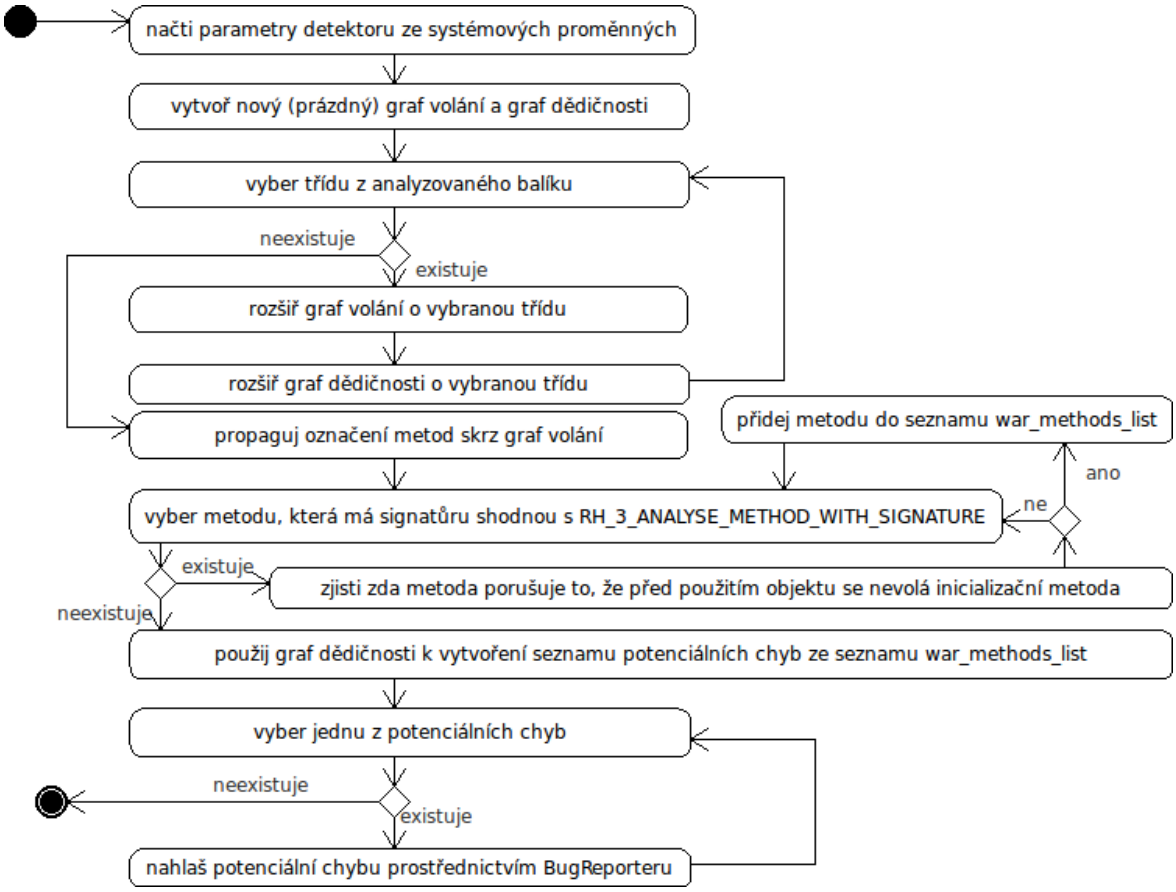
CallGraph vytváří graf volání. Při jeho tvorbě prochází byte-kódem jednotlivých metod a snaží se najít potenciálně nebezpečné metody. Potenciálně nebezpečnou metodou je přitom chápána taková, kde je volána metoda od určené třídy (implicitně Message) bez předchozí inicializace zvolenou metodou (implicitně getPayLoad(Message)). Na obrázku je zobrazeno, že třída CallGraph je poděděna z AbstractGraph<CallGraphEdge, CallGraphNode> a pracuje s třídami CallGrapNode (poděděnou z AbstractVertex<CallGraphEdge, CallGraphNode>) a CallGraphEdge (poděděnou z AbstractEdge<CallGraphEdge, CallGraphNode>). Pátou třídou, která se nachází na Ilustraci 18 a je využita při analýze je InheritanceGraph. Tato třída slouží k tvorbě grafu dědičnosti a je poděděna z AbstractGraph<InheritanceEdge, ClassVertex>. Uzly grafu dědičnosti představuje třída ClassVertex poděděná z AbstractVertex<InheritanceEdge, ClassVertex>. Hrany grafu dědičnosti představuje třída InheritanceEdge poděděná z AbstractEdge<InheritanceEdge, ClassVertex>. Graf dědičnosti stejně jako graf volání se rozšiřuje při každém navštívení metody visitClassContext() třídy TestMethodGetPayLoad o analyzovanou třídu popsanou ClassContextem.



Ilustrace 18: Diagram tříd 2. detektoru – TestMethodGetPayLoad.

4.3.2 Implementace

Třetí detektor byl implementován způsobem znázorněným na následující Ilustraci 19.



Ilustrace 19: Diagram aktivit – TestMethodGetPayLoad.

Celá implementace se dá přitom rozdělit do čtyř částí. První částí je práce třídy TestMethodGetPayLoad zobrazená na Ilustraci 19. Druhá část je zobrazena na Ilustraci 19 jako „rozšiř graf volání o vybranou třídu“ funkčně ji má na starosti metoda CallGraph.extend() a podrobněji ji zobrazuje Ilustrace 20. Třetí část je zobrazena na Ilustraci 19 jako „propaguj označení metod skrz graf volání“ na starosti ji má metoda CallGraph.flagPropagate(). Podrobněji ji zobrazuje Ilustrace 16, která byla uvedena u předešlého detektoru a protože tyto detektory využívají stejný princip propagace, tak zde nebude znovu uvedena. Čtvrtá část je na Ilustraci 19 zobrazena jako „*použij graf dědičnosti k vytvoření seznamu potenciálních chyb ze seznamu war_methods_list*“ a podrobněji je uvedena na Ilustraci 21.

Na Ilustraci 19 je vidět, že proces analýzy začíná načtením vstupních parametrů detektoru. Parametry se zadávají jako systémové proměnné. V případě nenastavení vstupních parametrů, pracuje

detektor ve standardním režimu. V opačném případě přizpůsobuje analýzu vstupním parametrům. Parametry, které je možné využít ke konfiguraci detektoru jsou následující:

- *RH_3_DEFINE_USE* – přepíná detektor mezi implicitním (neparametrizovaném)/explicitním (parametrizovaném) použitím. Explicitní použití detektoru je zaručeno nastavením tohoto parametru na hodnotu „1“. Implicitního použití detektoru lze docílit při zadání hodnoty „0“, nedefinováním nebo nekorektním definováním tohoto parametru.
- *RH_3_ANALYSE_CLASSES_WITH_INTERFACES* – určuje množinu rozhraní, ze které musí třída nejméně jedno implementovat, aby bylo v jejich metodách případné porušení neinicializování objektu (standardně pomocí metody *getPayload(Message)*) před jeho použitím (standardně voláním metody třídy *Message*) považováno za potenciální chybu. Při implicitním běhu detektoru bývá v této množině identifikace jednoho rozhraní:
  - *org/jboss/soa/esb/actions/ActionPipelineProcessor*

  Explicitně se interface udává v podobném formátu. Formát byl definován v kapitole 2.1.1. Jména jednotlivých rozhraní jsou přitom oddělována znakem „|“.
- *RH_3_METHOD_WHICH_MAKE_SAFE_METHODS* – představuje inicializační metodu (standardně *getPayload(Message)*), která by se měla vyskytovat před použitím objektu (standardně voláním metody třídy *Message*). Implicitně bývá tento parametr nastaven na:
  - *org/jboss/soa/esb/message/MessagePayloadProxy.getPayload(Lorg/jboss/soa/esb/message/Message;)Ljava/lang/Object;*

  Explicitně se metoda zadává v podobném formátu. Formát byl definován v kapitole 2.1.1.
- *RH_3_ANALYSE_METHOD_WITH_SIGNATURE* – určuje signaturu metod, ve kterých se budou hledat potenciální chyby. Návratový typ ze signatury přitom ještě definuje třídu objektu (*OBJ*) před jehož použitím musí být provedena inicializace prostřednictvím volání *RH_3_METHOD_WHICH_MAKE_SAFE_METHODS*. Při implicitním běhu detektoru bývá signatura definována jako:
  - *(Lorg/jboss/soa/esb/message/Message;)Lorg/jboss/soa/esb/message/Message;*

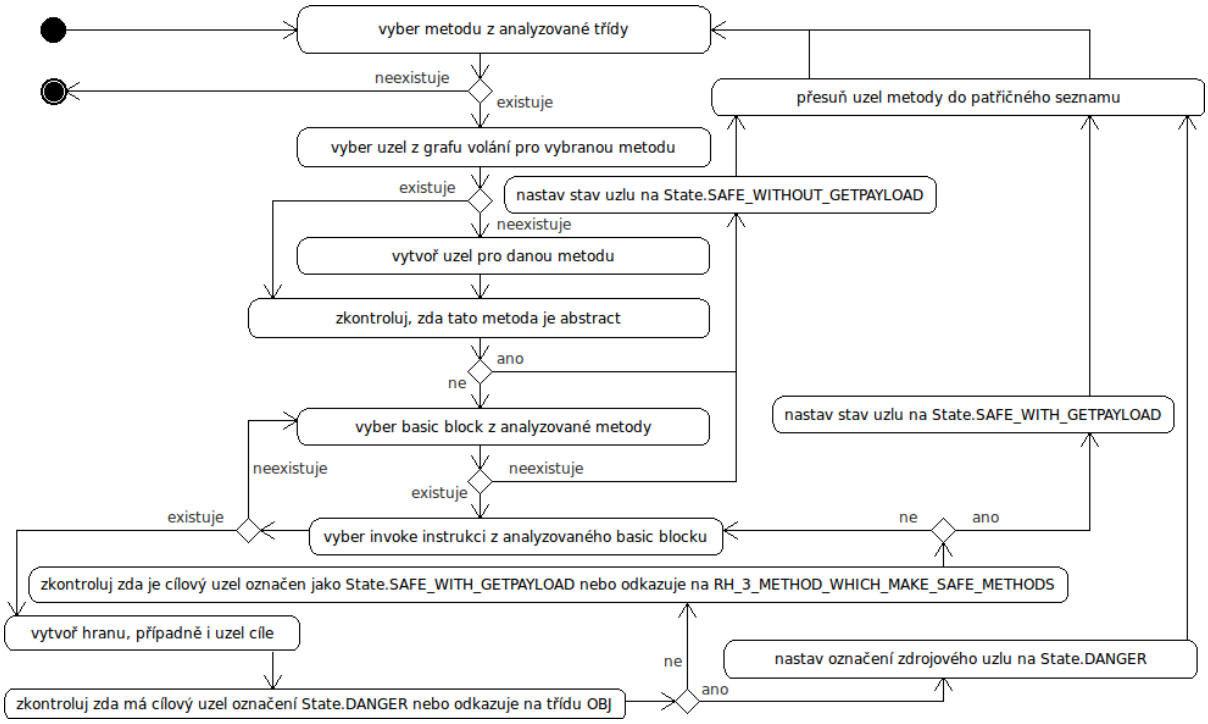
  Explicitně se signatura zadává v podobném formátu. Formát byl definován v kapitole 2.1.1.

Jakmile jsou načteny parametry začíná proces samotné analýzy. Vytvoří se nový (prázdný) graf volání a graf dědičnosti. Následuje rozšíření grafu volání a grafu dědičnosti o všechny třídy předhozené na vstup nástroje FindBugs. Graf dědičnosti se rozšiřuje o hrany spojující současně analyzovanou třídu s třídami rozhraní jež implementuje a třídou rodičovské třídy. Rozšiřování grafu volání v sobě nese prvky analýzy a bude podrobněji popsáno na Ilustraci 20. Po rozšíření obou grafů se spustí propagace označení State.DANGER. Princip propagace je stejný jako u 2. detektoru a jeho popis se dá najít na Ilustraci 16.

Když skončí propagace označení, přechází se do druhé části analýzy. Ve všech třídách se hledají metody se signaturou shodnou s RH_3_ANALYSE_METHOD_WITH_SIGNATURE. Když se nějaká taková metoda nalezne, tak je testováno, zda je před voláním metody z třídy OBJ použito volání RH_3_METHOD_WHICH_MAKE_SAFE_METHODS. K tomu je využito jak samotné prohledávání byte-kódu, tak v této chvíli i vytvořený graf volání. Graf volání se využívá z důvodu, že volání metody třídy OBJ nebo RH_3_METHOD_WHICH_MAKE_SAFE_METHODS se může skrývat pod některou metodou na níž se právě analyzovaná metoda odkazuje přes invoke instrukci. V případě, že se najde nějaká metoda, kde před použitím OBJ není volána metoda ze třídy RH_3_METHOD_WHICH_MAKE_SAFE_METHODS, tak je tato metoda uložena do seznamu war_methods_list (představující seznam metod, kde byl objekt použit před inicializací). Po analýze všech metod se použije graf dědičnosti k vytvoření seznamu potenciálních chyb (SEZ – metody uložené v tomto seznamu se od metod z war_methods liší v tom, že všechny patří do tříd implementujících nejméně jedno z rozhraní *RH_3_ANALYSE_CLASSES_WITH_INTERFACES*) z metod uvedených ve war_methods_list. Princip této činnosti zachycuje Ilustrace 21. Následně jsou všechny záznamy ze seznamu SEZ s využitím třídy BugReporter nahlášeny jako potenciální chyby. Chybové hlášení o potenciální chybě nalezené 3. detektorem se hlásí na standardní výstup ve formátu, který může přiblížit následující příklad chybového hlášení: „H C PV3: Miss method getPayLoad(Method) in testCases.TestRH3.process(Message) At TestRH3.java:[lines 83-84]“, kde „H C PV3“ nám identifikuje hlášení 3. detektoru nasledované popisem a pozicí chyby.

Nyní přichází na řadu popis rozšiřování grafu volání (na Ilustraci 19 se objevující pod „rozšiř graf volání o vybranou třídu“) zobrazený na Ilustraci 20. V detektoru tuto práci provádí metoda CallGraph.extend(). Metoda extend() dostane ClassContext reprezentující třídu, o kterou má být rozšířen graf volání. CallGraph je postupně rozšířen o všechny její metody následujícím způsobem. Zjistí se zda pro metodu třídy existuje uzel. Pokud neexistuje, tak se vytvoří (označení se mu nastaví na State.UNKNOWN) a vloží se do seznamu neznámých metod *(unknown-list*). Dále se zkontroluje, zda se jedná o abstraktní metodu. Pokud ano, tak je tato metoda bezpečná, takže se značení uzlu metody přeznačí na State.SAFE_WITHOUT_GETPAYLOAD a přesune se z unknown-list do seznamu bezpečných metod (*safe-list*). V opačném případě se postupně prohledají všechny instrukce metody. Analýza se přitom zaměřuje na invoke instrukce. Pokud je nějaká invoke instrukce nalezena a ještě pro ni neexistuje hrana tak se vytvoří. V případě, že by zatím neexistoval cílový uzel hrany, je také vytvořen (označen je jako State.UNKNOWN a vložen do seznamu unknown-list). Následuje kontrola, zda označení cílového uzlu má označení State.DANGER nebo se odkazuje na OBJ (výše – ze signatury definovanou třídu). Pokud ano, tak je zdrojový uzel označen jako State.DANGER a přesune se do seznamu nebezpečných metod (danger-list). V opačném případě je testováno, zda cílový uzel má označení State.SAFE_WITH_GETPAYLOAD nebo odkazuje na

RH_3_METHOD_WHICH_MAKE_SAFE_METHODS. Pokud ano, tak je zdrojový uzel označen jako State.SAFE_WITH_GETPAYLOAD a přesunut do safe-list. V opačném případě uzel zůstává State.UNKNOWN. Pokud zdrojový uzel nezískal označení State.SAFE_WITH_GETPAYLOAD ani State.DANGER, tak se pokračuje v hledání dalších invoke instrukcí a označení uzlu se nemění. Když není nalezena žádná invoke instrukce volající uzel s označením State.DANGER, OBJ, nebo State.SAFE_WITH_GETPAYLOAD získá uzel označení State.SAFE_WITHOUT_GETPAYLOAD a může být změněn už jen metodou CallGraph.flagPropagate().

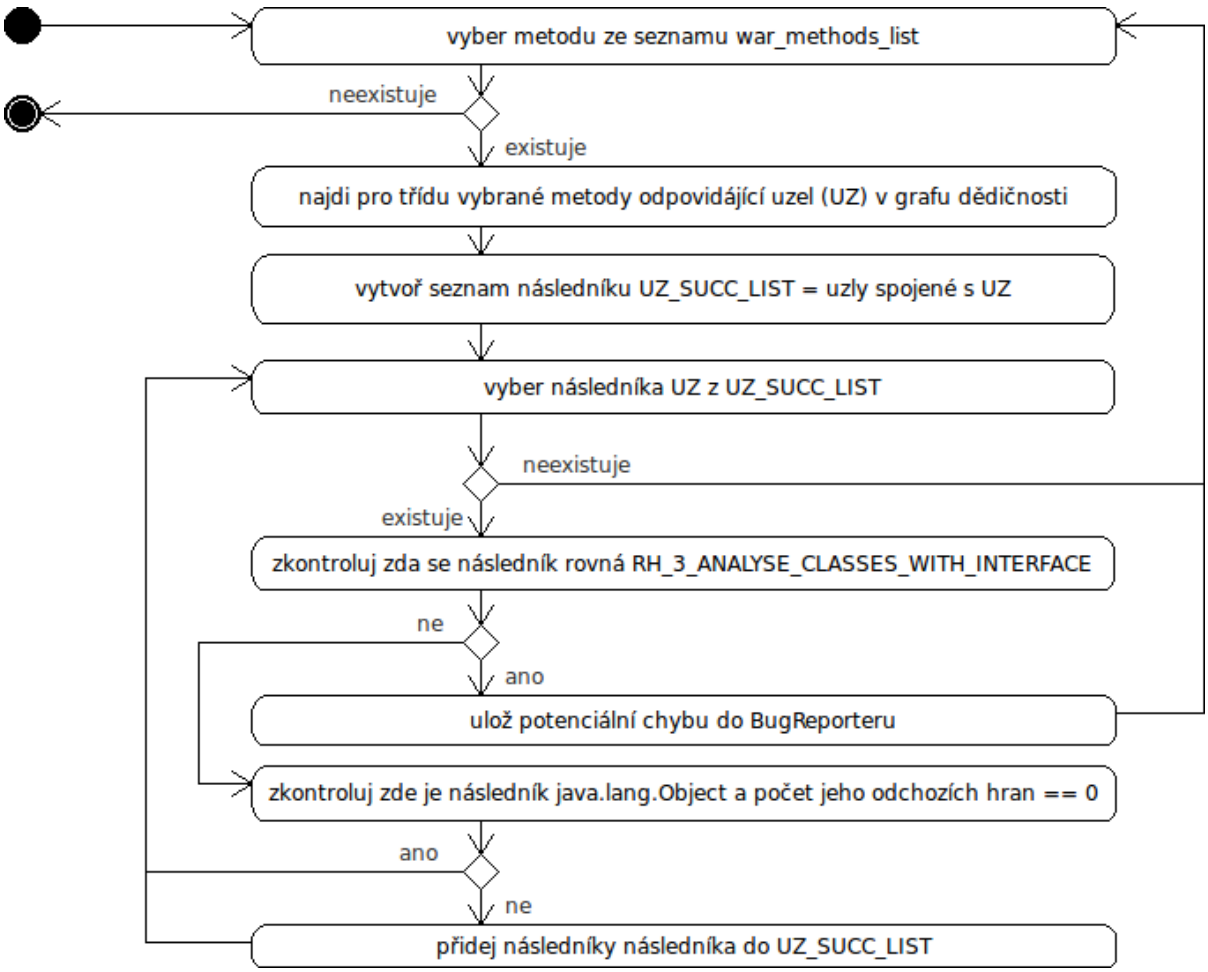


Ilustrace 20: Diagram aktivit – rozšiřování grafu volání – CallGraph.extend().

Následující Ilustrace 21 popisuje využití grafu dědičnosti v analýze. V Ilustraci 19 popisující TestMethodGetPayLoad se využití grafu dědičnosti skrývá pod „použij graf dědičnosti k vytvoření seznamu potenciálních chyb ze seznamu war_methods_list“. Analýza za pomoci grafu dědičnosti zde vystupuje jako filtr. Ze seznamu metod war_methods_list se zjistí, které z těchto metod implementují rozhraní RH_3_ANALYSE_CLASSES_WITH_INTERFACE a jsou tedy považovány za potenciální chyby.

Na Ilustraci 21 je vidět, že se postupně analyzují všechny metody uložené ve war_methods_list. Vybere se jedna metoda. Zjistí se uzel (UZ) z grafu dědičnosti odpovídající třídě do které metoda patří. Vytvoří se seznam následníků UZ_SUCC_LIST představující všechny uzly přímo spojené s UZ. Vybere se jeden z následníku z UZ_SUCC_LIST. Pokud následník neexistuje, tak analýza metody končí. Výsledkem je, že metoda neobsahuje chybu z důvodu, že její třída

neimplementuje rozhraní RH_3_ANALYSE_CLASSES_WITH_INTERFACE. Přejde se tedy k další metodě z war_methods_list. V opačném případě (následník existuje), je provedena kontrola na shodu s rozhraním RH_3_ANALYSE_CLASSES_WITH_INTERFACE. Pokud je nalezena shoda, tak se do BugReporteru uloží záznam o chybě. Jinak je téže následník kontrolován na shodu s java.lang.Object. V případě shody a současně když má následník 0 následníků, přechází se na výběr dalšího následníka z UZ_SUCC_LIST. V opačném případě se uloží všichni následníci následníka do seznamu UZ_SUCC_LIST. Tímto se zajistí, že i rozhraní implementované následníkem budou považovány jako rozhraní metody z war_methods_list, kterou analyzujeme. Takto se bude tvořit tranzitivní uzávěr. Celý proces pokračuje výběrem dalšího následníka z UZ_SUCC_LIST.



Ilustrace 21: Diagram aktivit – využití grafu dědičnosti při analýze.

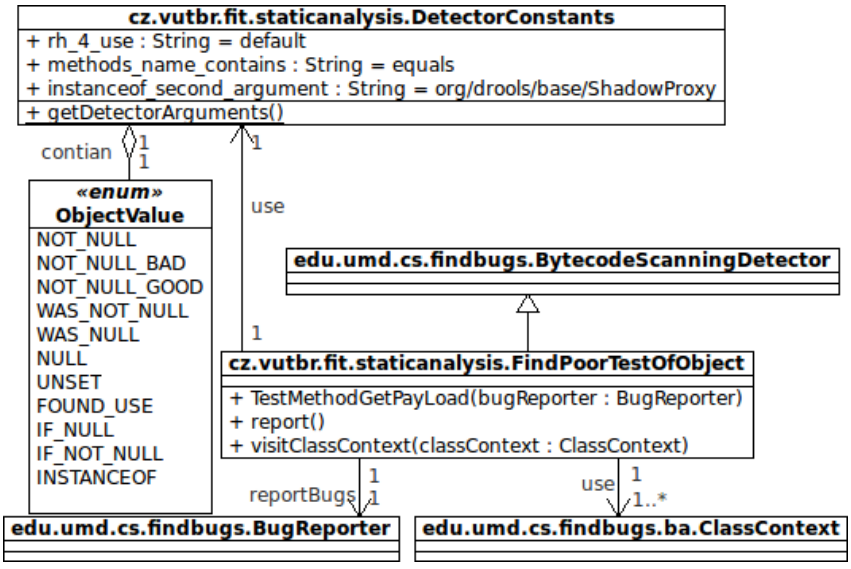
4.4 Detekce nedokonalého testování objektů

V této části bude popsán návrh a implementace 4. detektoru – *FindPoorTestOfObject.* Detektor byl použit pro detekci chyby popsané v kapitole 3.4. Jeho úkolem je hledat nedokonalé testování objektů před použitím metody `equals()` v metodách obsahujících ve svém názvu řetězec „*equals*“. Detektor byl mimo to rozšířen tak, aby si uživatel mohl zvolit název metod, ve kterých se má hledání nedokonalého testování objektů provádět. Před voláním metody `equals()` se 2. argument implicitně testuje na shodu s třídou *ShadowProxy*. Tato třída se dá v explicitním použití detektoru také určit. Popis implicitní a explicitní funkčnosti detektoru bude popsán v části – 4.4.2 zabývající se implementací detektoru. V příloze jsou pod tabulkou 19 zobrazeny testy provedené s detektorem.

Detektor byl vytvořen tak, aby zastával defenzivní programování. Důsledkem toho je, že může občas produkovat false alarmy. Příklad, kdy se false alarm objeví může být např. tento. Detektor má za úkol (v implicitním režimu) otestovat metodu isObjectEquals(Object arg0, Object arg1). Předpokládejme, že před voláním této metody je objekt arg1 vždy otestován na hodnotu null a toto testování se už v metodě isObjectEquals(Object arg0, Object arg1) neobjevuje. V analyzované metodě dochází k situaci, kdy je voláno arg0.equals(arg1) a detektor vyprodukuje false alarm. Je to z důvodu, že detektor neví, že objekt arg1 je vždy před voláním metody isObjectEquals(Objcect arg0, Object arg1) testován na null. Příklad kdy k vyprodukování false alarmu došlo je v kapitole 6.4.

4.4.1 Návrh

Ilustraci 22 zobrazuje diagram tříd 4. detektoru.

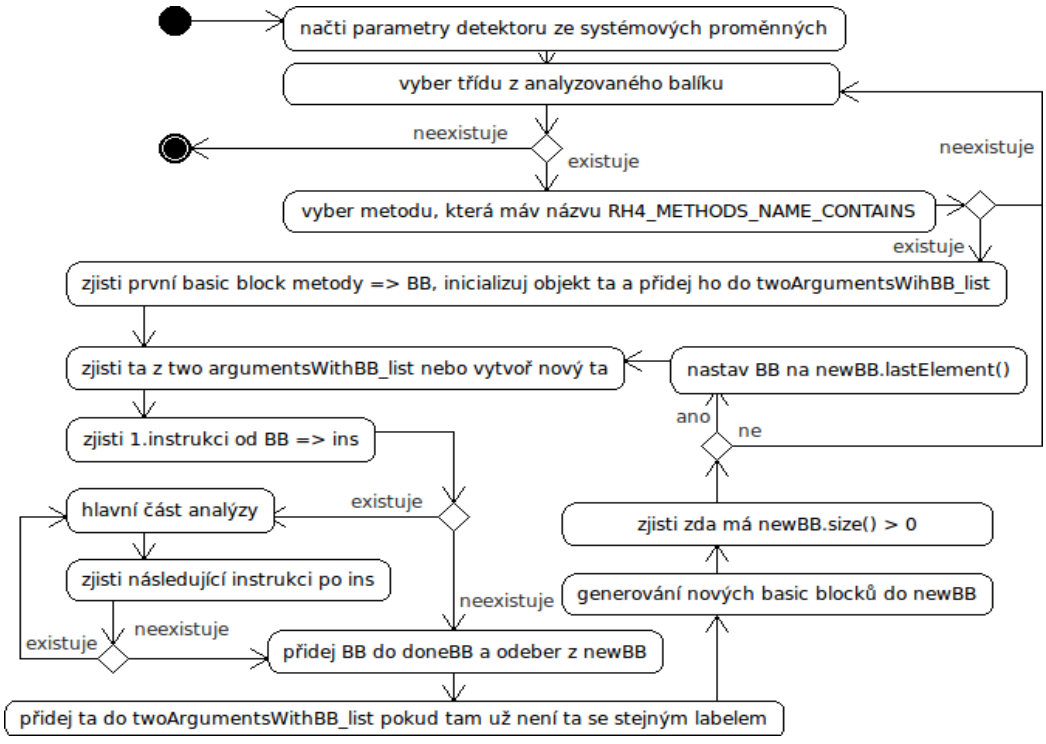


Ilustrace 22: Diagram tříd 4. detektoru – FindPoorTestOfObject.

Detektor schopný hledat nedokonalé testování objektů před použitím metody equals() je dědí z třídy BytecodeScanningDetector. Na je zobrazeno, že celá funkčnost 4. detektoru je zapouzdřena ve třídě FindPoorTestOfObject. Tato třída k práci ještě využívá další tři třídy. Třída ClassContext představuje jednotlivé analyzované třídy, které jsou detektoru předávány jádrem nástroje FindBugs prostřednictvím metody visitClassContext(). Druhou třídou využívanou při analýze je BugReporter sloužící k hlášení nalezených chyb. Třetí třídou je DetectorConstants starající se o načtení a správu argumentů detektoru.

4.4.2 Implementace

Čtvrtý detektor byl implementován způsobem znázorněným na Ilustraci 23. Celá implementace se dá rozdělit do čtyř částí. První částí je práce třídy *FindPoorTestObject* zobrazená na Ilustraci 23. Druhá část je zobrazena na Ilustraci 23 pod označením „hlavní část analýzy“ podrobněji ji zobrazuje Tabulka 4 popisující akce, které způsobí detekce některých pro analýzu důležitých instrukcí. Třetí část je na Tabulce 4 k nalezení pod „*handleIReturnInstruction()*“ podrobněji zobrazená Tabulkou 5 popisující vyhodnocení skutečných a předpokládaných návratových hodnot při detekci instrukce ireturn. Čtvrtá část je zobrazena na Ilustraci 23 jako „generování basic blocků do *newBB*“. Podrobněji ji zobrazuje Tabulka 6 popisující jak se přes CFG přenáší pro analýzu potřebné informace.



Ilustrace 23: Diagram aktivit – FindPoorTestObject.

Parametry využitelné ke konfiguraci detektoru jsou následující:

- *RH_4_DEFINE_USE* – přepíná detektor mezi implicitním (neparametrizovaném)/explicitním (parametrizovaném) použitím. Explicitní použití detektoru je zaručeno nastavením tohoto parametru na hodnotu „1“. Implicitního použití detektoru lze docílit při zadání hodnoty „0“, nedefinováním nebo nekorektním definováním tohoto parametru.
- *RH_4_METHODS_NAME_CONTAINS* – určuje řetězec, který se musí nacházet ve jméně metody, aby byla metoda analyzována. Metody se jménem neobsahujícím tento řetězec jsou při analýze přeskakovány. Implicitně bývá tento parametr nastaven na „*equals*“. V případě, že při explicitním použití není tento parametr nastaven, tak se nastaví na prázdný řetězec a jsou analyzovány všechny metody.
- *RH_4_INSTANCEOF_SECOND_ARGUMENT* – definuje jméno třídy na kterou musí být pomocí instrukce *instanceof* testován druhý objekt před voláním metody *equals()*. Implicitně bývá tento parametr nastaven na „*org/drools/base/ShadowProxy*“. Explicitně se parametr nastavuje s ohledem na pravidla zmíněné v kapitole 2.1.1.

Jakmile jsou načteny parametry začíná proces samotné analýzy. Z balíku tříd, který byl vložen na vstup nástroje FindBugs se postupně analyzují všechny třídy. Analýza pokračuje, dokud neprohledá všechny třídy reprezentované pomocí tříd ClassContext. Před analýzou metod třídy se zkontroluje, zda jméno metody obsahuje RH_4_METHODS_NAME_CONTAINS. V případě, že není shoda nalezena, tak se metoda přeskakuje. V implicitním režimu detektoru, kde se RH_4_METHODS_NAME_CONTAINS=equals se bude pokračovat např. v analýze metody: org.jboss.internal.Test.equals() a naopak metoda: org.jboss.internal.Test.process() bude přeskočena, protože její jméno neobsahuje „equals“. Provede se patřičná inicializace objektů. Při prohledávání se podle Tabulka 4 testují všechny instrukce od basic blocků patřících metodě. Tabulka 4 zobrazuje akce (Výsledek – akce), které se provedou při detekci pro analýzu důležitých instrukcí (Instrukce) a splnění podmínek (Podmínka). Prvky využité v tabulce popisuje následující legenda nad tabulkou. Popis co jednotlivé byte-kódové instrukce dělají je k nalezení v tabulce 2 patřící do kapitole 2.1.

Následující tabulka popisuje část funkcionality bloku, který je na Ilustraci 24 nazýván „hlavní část analýzy“. U instrukcí, kde není specifikováno ELSE není při ELSE provedena žádná akce.

Instrukce	**Podmínka**	**Výsledek – akce**
IF_ACMPNE		ta.ifacmpne = true
INSTANCEOF	**IF** ((ins má argument DC.INSTANCEOF_SECOND_ARGUMENT) && (ta.arg2 != OV.NOT_NULL) && (ta.arg2 != OV.WAS_NOT_NULL) && (ta.arg2 != OV.FOUND_USE))	reportNewBug()
	ELSE	ta.arg2 = OV.INSTANCEOF
INVOKEVIRTUAL	**IF** ((ta.arg1 == OV.*NOT_NULL* || ta.arg1 == OV.*WAS_NOT_NULL*) && (ta.arg2 == OV.*NOT_NULL_GOOD*) && methodName.equals("equals"))	ta.invokevirtual = true
	ELSE IF ((ta.arg1 == OV.*UNSET* || ta. OV == OV.*NULL* || ta.arg1 == OV.*WAS_NULL* || ta.arg1 == OV.*FOUND_USE* || ta.arg2 == OV.*UNSET* || ta.arg2 == OV.*NULL* || ta.arg2 == OV.*WAS_NULL* || OV OV.*FOUND_USE*) && methodName.equals("equals"))	reportNewBug()
IFNULL	**IF** (ta.arg1 == OV.*FOUND_USE* || ta.arg1 == OV.*WAS_NOT_NULL* || ta.arg1 == OV.*WAS_NULL*)	ta.arg1 = OV.*IF_NULL*
	ELSE IF (ta.arg2 == OV.*FOUND_USE* || ta.arg2 == OV.*WAS_NOT_NULL* || ta.arg2 == OV.*WAS_NULL*)	ta.arg2 = OV.*IF_NULL*
IFNONNULL	**IF** (ta.arg1 == OV.*FOUND_USE* || ta.arg1 == OV.*WAS_NOT_NULL* || ta.arg1 == OV.*WAS_NULL*)	ta.arg1 = OV.*IF_NOT_NULL*
	ELSE IF (ta.arg2 == OV.*FOUND_USE* || ta.arg2 == OV.*WAS_NOT_NULL* || ta.arg2 == OV.*WAS_NULL*)	ta.arg2 = OV.*IF_NOT_NULL*
ALOAD_1	**IF** (ta.arg1 == OV.*UNSET)*	ta.arg1 = OV.*FOUND_USE*
	ELSE IF (ta.arg1 == OV.*NOT_NULL*)	ta.arg1 = OV.*WAS_NOT_NULL*
	ELSE IF (ta.arg1 == OV.*NULL*)	ta.arg1 = OV.*WAS_NULL*
ALOAD_2	**IF** (ta.arg2 == OV.*UNSET*)	ta.arg2 = OV.*FOUND_USE*
	ELSE IF (ta.arg2 == OV.*NOT_NULL*)	ta.arg2 = OV.*WAS_NOT_NULL*
	ELSE IF (ta.arg2 == OV.*NULL*)	ta.arg2 = OV.*WAS_NULL*
IRETURN		handleIReturnInstruction()

Tabulka 4: Popis hlavní části analýzy.

Legenda (pro Tabulku 4 a Tabulku 5):

- *ta* – objekt sdružující následující informace:
  - *bbLabel* – určuje označení basic blocku pro který ta platí
  - *arg1* – určuje hodnotu 1. argumentu
  - *arg2* – určuje hodnotu 2. argumentu
  - *invokevirtual* – určuje zda byla na cestě grafu toku dat (*CFG*) nalezena instrukce *invokevirtual* odkazující na metodu *equals()*
  - *ifacmpne* – určuje zda byla na cestě *CFG* nalezena instrukce *ifacmpne*
- *ins* – je právě analyzovaná instrukce
- *DetectorConstants* – třída sdružující konstanty. Dále pod označením *DC*.
- *ObjectValue* – je výčet popisující stav objektů. Patří do třídy *DC* dále označován jako *OV*.
- *methodName* – je jméno právě analyzované metody
- *reportNewBug()* – představuje metodu, která hlásí pomocí *BugReporteru* chybu

handleIReturnInstruction() - představuje metodu zpracovávající nalezení instrukce *IReturn*.

Metoda handleIReturnInstruction() je volána v případě detekce instrukce IReturn značící, že se bude navracet hodnota z ConstantFrame. Pro představu co se nachází pod voláním handleIReturnInstruction() z předchozí Tabulky 4 je zde uvedena Tabulka 5 zachycující funkcionalitu této metody.

Část tabulky vyhodnocující se v případě, že vrchol ConstantFrame *je* konstantní hodnota:			
ta.arg1	**ta.arg2**	**Chyba při detekování:**	**Správný výstup:**
OV.*NULL*||OV.*WAS_NULL*	OV.*NULL*||OV.*WAS_NULL*	*1	true
OV.NOT_NULL||*OV.WAS_NOT_NULL*	*OV.NULL*||*OV.WAS_NULL*	*2	false
OV.*NOT_NULL*||OV.*WAS_NOT_NULL*	OV.*NOT_NULL_GOOD*	*3	*4
OV.*NULL*||OV.*WAS_NULL*	OV.*NOT_NULL_GOOD*	*2	false
OV.*NOT_NULL*||OV.*WAS_NOT_NULL*	OV.*NOT_NULL_BAD*	*5	
OV.*NULL*||OV.*WAS_NULL*	OV.*NOT_NULL_BAD*	*2	false
OV.*NOT_NULL*||OV.*WAS_NOT_NULL*	OV.*NOT_NULL*||OV.*WAS_NOT_NULL*	*6	
Část tabulky vyhodnocující se v případě, že vrchol ConstantFrame *není* konstantní hodnota:			
OV.*NOT_NULL*||OV.*WAS_NOT_NULL*	OV.*NOT_NULL_BAD*	*7	
ta.arg1!=OV.UNSET && ta.arg2!=OV.UNSET && ta.arg1!=OV.FOUND_USE && ta.arg2!=OV.FOUND_USE && ta.arg1!=OV.NULL && ta.arg2!=OV.NULL		*8	
ELSE	ELSE	*9	

Tabulka 5: Popis metody handleIReturnInstruction().

Vysvětlivky (*X z Tabulky 5):

1. *False* na vrcholu *ConstantFrame*.
2. *True* na vrcholu *ConstantFrame*.
3. Nebyla volána *invokevirtual* metoda (*ta.invokevirtual != true*) nebo na vrcholu *ConstantFrame* je konstantní hodnota.
4. Je očekáváno, že byla vykonána *invokevirtual* instrukce. V případě, že je volána *invokevirtual* metoda *(ta.invokevirtual == true*), tak je návratová hodnota instrukce *ireturn* závislá na hodnotě volání metody => vrchol *ConstantFrame* není konstanta.
5. Pro detekci nepodstatné – žádná chyba v tomto stavu není hlášena.
6. Při detekci je vždy hlášena chyba.
7. Pokud platí hodnoty objektů (*ta.arg1* a současně *ta.arg2*) nebo byla na předchozí cestě *CFG* detekována instrukce *ifacmpne* nebude v žádném případě nahlášena chyba.
8. Pokud platí hodnoty objektů *(ta.arg1* a současně *ta.arg2*) a byla na předchozí cestě *CFG* detekována instrukce *invokevirtual* nebude v žádném případě nahlášena chyba.
9. V ostatních případech bude vždy chyba nahlášena.

Obrázek ukazuje kontrolu, kterou provádí metoda handleIReturnInstruction() nad návratovými hodnotami. Jednotlivé řádky ukazují jaké jsou očekávané návratové hodnoty (sloupec Správný výstup), při zadaných hodnotách dvou porovnávaných argumentů (sloupce *ta.arg1*, *ta.arg2*) a při jakých návratových hodnotách (sloupec Chyba při detekování) se hlásí potenciální chyby.

Po popisu zpracování instrukcí (Tabulka 4 a Tabulka 5) se v basic blocku zjistí následující instrukce, pro kterou se opět spustí hlavní část analýzy. Tyto nové instrukce se generují do doby než jsou vyčerpány všechny instrukce z BB. Poté se BB přidá do seznamu *doneBB* (zpracovaných basic blocků) a odebere se ze seznamu *newBB* (nových – nezpracovaných basic blocků). Objekt ta se přidá do seznamu *twoArgumentsWithBB_list*. Tento seznam je zde z důvodu, že při procházení CFG je využíváno procházení do hloubky. Při prohledávání je třeba si pamatovat jaké hodnoty argumentů jsou v jednotlivých basic blocích, aby se tyto hodnoty mohly načíst při zpětném sledování (vynořování). Následuje generování nových basic blocků zobrazené na ilustraci Ilustraci 24 jako „generování nových basic blocků do newBB“. Při generování nových basic blocků se z CFG zjišťují basic blocky následující po BB. Podle toho jaký druh hrany (důležité druhy hran byly popsány v kapitole 2.2.1) spojuje následující basic block (BB_NEW) s již prohledaným BB se vytváří *newTA* (ze starého ta), které se následně vkládají do twoArgumentsWithBB_list. Objekty z twoArgumentsWithBB_list jsou s basic blocky v newBB spárovány pomocí ID basic blocku (newTA.bbLabel). Toho je následně využito při prohledávání basic blocků, protože jsou tímto

způsobem předány známé informace o argumentech mezi dvěma následujícími basic blocky. Vytváření objektu *newTA* založené na objektu *ta* a hraně (typu *EdgeTypes*) spojující *BB* a *BB_NEW* shrnuje Tabulka 6. Podmínka vázající se k ta je zobrazena jako *ta.arg,* což ve skutečnosti znamená, testování *ta.arg1* nebo *ta.arg2*. Vždy se testují a nastavují oba argumenty. Z důvodu, že jsou ale pravidla pro jejich nastavování stejné, jsou zde zmíněny pouze jednou v neutrální podobě. Totéž platí k *newTA.arg* popisující *newTA.arg1* a *newTA.arg2*. Hodnoty *ta.invokevirtual* a *ta.ifacmpne* se přenášejí přes všechny hrany nezměněné.

Hrana:	**Podmínka vázající se k ta:**	**Vytvořené newTA:**
IFCMP_EDGE	ta.arg==OV.IF_NULL|| ta.arg==OV.NULL|| ta.arg==OV.WAS_NULL	newTA.arg=OV.NULL
	ta.arg==OV.IF_NOT_NULL|| ta.arg==OV.NOT_NULL|| ta.arg==OV.WAS_NOT_NULL	newTA.arg=OV.NOT_NULL
	ta.arg==OV.FOUND_USE	newTA.arg=OV.FOUND_USE
	ta.arg==OV.INSTANCEOF|| ta.arg==OV.NOT_NULL_BAD	newTA.arg=OV.NOT_NULL_BAD
	ta.arg == OV.NOT_NULL_GOOD	newTA.arg=OV.NOT_NULL_GOOD
FALL_THROUGH_EDGE	ta.arg==OV.IF_NULL|| ta.arg==OV.NOT_NULL|| ta.arg==OV.WAS_NOT_NULL	newTA.arg=OV.NOT_NULL
	ta.arg==OV.IF_NOT_NULL|| ta.arg==OV.NULL|| ta.arg==OV.WAS_NULL	newTA.arg=OV.NULL
	ta.arg==OV.FOUND_USE	newTA.arg=OV.FOUND_USE
	ta.arg==OV.INSTANCEOF|| ta.arg==OV.NOT_NULL_GOOD	newTA.arg=OV.NOT_NULL_GOOD
	ta.arg==OV.NOT_NULL_BAD	newTA.arg=OV.NOT_NULL_BAD
SWITCH_EDGE,SWITCH_DEFAULT_EDGE, GOTO_EDGE	ta.arg==OV.NOT_NULL|| ta.arg==OV.WAS_NOT_NULL	newTA.arg=OV.NOT_NULL
	ta.arg==OV.NULL|| ta.arg==OV.WAS_NULL	newTA.arg=OV.NULL
	ta.arg==OV.INSTANCEOF|| ta.arg==OV.NOT_NULL_GOOD	newTA.arg=OV.NOT_NULL_GOOD
	ta.arg==OV.NOT_NULL_BAD	newTA.arg=OV.NOT_NULL_BAD
	ta.arg==OV.FOUND_USE	newTA.arg=OV.FOUND_USE

Tabulka 6: Pravidla pro vytváření newTA na základě hrany a starých hodnot ta.

Po vytvoření nových basic blocků a k něm příslušných *newTA* se provede kontrola zda velikost seznamu *newBB* obsahuje nějaké nové basic blocky, které musí být prohledány. V případě, že ano, tak se jeden basic block vybere a celý proces analýzy s ním začne od začátku. Ukládání nových basic blocků do *newBB* a jejich následné vybírání je děláno jako prohledávání do hloubky.

V případě, že už v *newBB* není žádný basic block k prohledání a při analýze nebyla nahlášena chyba je metoda považována za bezpečnou. Analýza pokračuje prohledáváním dalších metod a tříd a končí když už jsou všechny třídy vložené na vstup nástroje FindBugs prohledány.

Chyby hlášené detektorem (zobrazené na Tabulkách 4 a Tabulkách 5) produkují chybové hlášení v následujícím formátu: „H C PV4: Found poor objects test in testCases.TestRH4.equals(Object, Object) At TestRH4..java:[lines 123-157]“, kde „H C PV4“ identifikuje detektor, který chybu našel následováno popisem chyby a místem, kde byla potenciální chyba nalezena.

4.5 Detekce nesynchronizované změny atributů

V této části bude popsán návrh a implementace 5. detektoru – *TestSuddenAttributesChange*. Detektor byl použit pro detekci chyby popsané v kapitole 3.5. Jeho úkolem je hledání nesynchronizované změny atributů tříd implementujících nejméně jedno rozhraní z množiny definovaných rozhraní. Při implicitním běhu je v množině rozhraní pouze ActionPipelineProcessor. Potencionální chyby jsou hlášeny prostřednictvím BugReporteru.

Detektor byl mimo to rozšířen, aby si uživatel mohl zvolit rozhraní tříd, které mají být analyzovány. Nastavení detektoru je proveditelné skrz systémové proměnné. Popis implicitní a explicitní funkčnosti detektoru bude popsán v části – 4.5.2 zabývající se implementací detektoru.

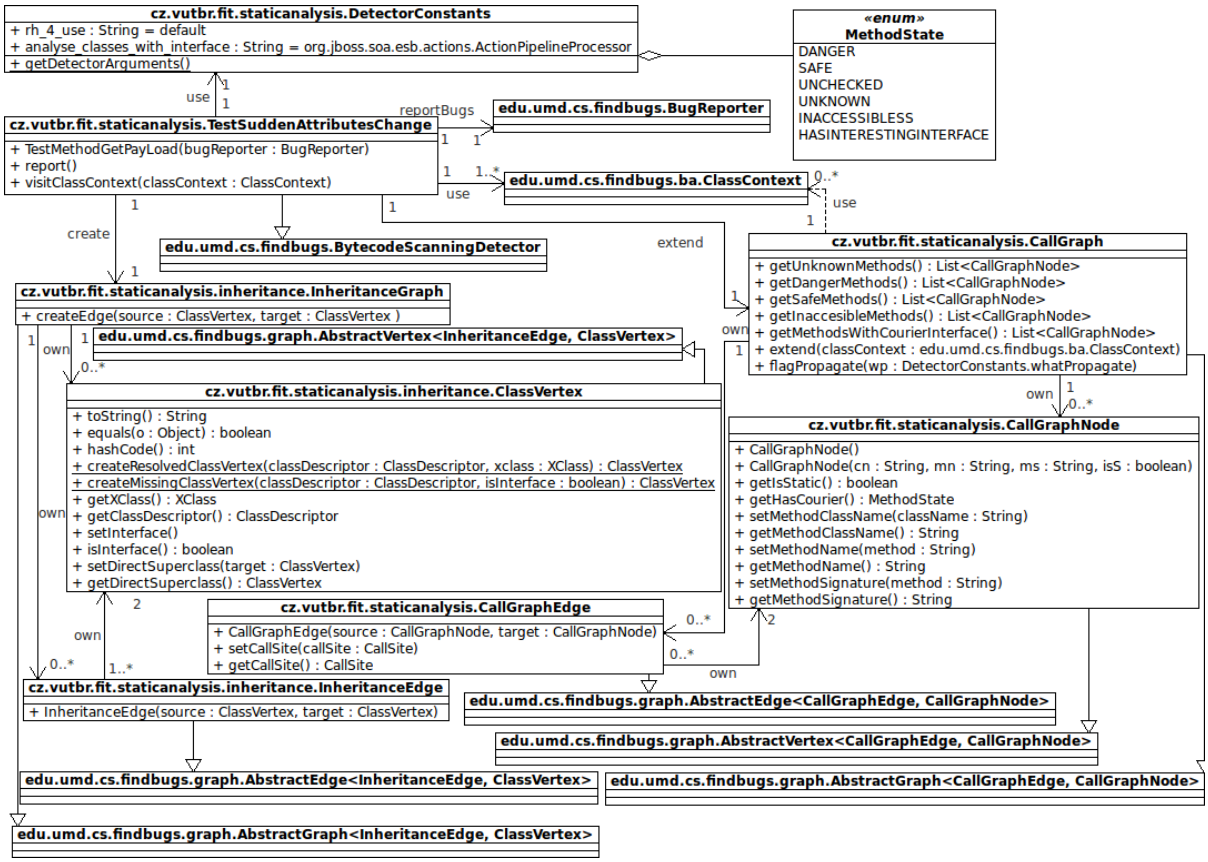
Detektor je schopen najít pouze ty nesynchronizované změny, které nenáleží do žádného synchronizačního bloku. V případě, že je atribut v synchronizovaném bloku, ale je synchronizován špatným objektem, zůstane tato chyba neodhalená. Je to z důvodu, že nástroj FindBugs neumožňuje vytvořit detektor, který by byl schopen tento druh chyb detekovat. V tabulce 20 nacházející se v přílohách jsou k nalezení výsledky testů prováděných s detektorem.

4.5.1 Návrh

Detektor TestSuddenAttributesChange je poděděn z BytecodeScanningDetector. Tuto závislost nám ukazuje diagram tříd zobrazený na Ilustraci 24.

Většina funkčnosti 5. detektoru je zapouzdřena ve třídách TestSuddenAttributesChange a CallGraph. Třída TestSuddenAttributesChange využívá k práci pět tříd. Jádro analýzy se skrývá pod metodou

extend() třídy CallGraph. Třída ClassContext představuje jednotlivé analyzované třídy, které jsou detektoru předávány jádrem nástroje FindBugs prostřednictvím metody visitClassContext(). Druhou třídou využívanou při analýze je BugReporter sloužící k hlášení nalezených chyb. Třetí třídou je DetectorConstants zajišťující načtení parametrů (metodou getDetectorArguments()) a zprostředkování veškerých konstant využívaných při analýze. Čtvrtou třídou je CallGraph. Třída CallGraph vytváří graf volání. Při jeho tvorbě prochází byte-kódem jednotlivých metod a snaží se najít potenciálně nebezpečné metody. Potenciálně nebezpečnou metodou je chápána taková, kde se nesynchronizovaně modifikuje některá z tříd přímo nebo nepřímo implementující alespoň jedno rozhraní z *RH_5_ANALYSE_CLASSES_WITH_INTERFACES* (implicitně ActionPipelineProcessor).



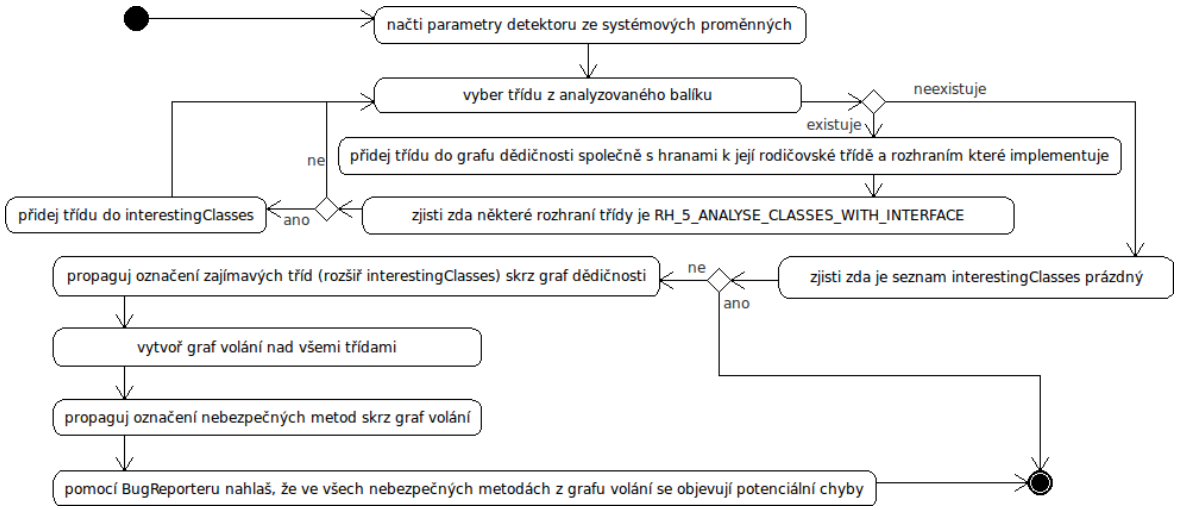
Ilustrace 24: Diagram tříd 5. detektoru – TestSuddenAttributesChange.

Na obrázku je zobrazeno, že třída CallGraph je poděděna z AbstractGraph<CallGraphEdge, CallGraphNode> a pracuje s třídami CallGrapNode (poděděnou z AbstractVertex<CallGraphEdge, CallGraphNode>) a CallGraphEdge (poděděnou z AbstractEdge<CallGraphEdge, CallGraphNode>). Pátou třídou, která se nachází na Ilustraci 24 a je využita při analýze je InherentanceGraph. Tato třída slouží k tvorbě grafu dědičnosti a je poděděna z AbstractGraph<InherentanceEdge, ClassVertex>. Uzly grafu dědičnosti představuje třída ClassVertex poděděná z AbstractVertex<InherentanceEdge,

ClassVertex>. Hrany grafu dědičnosti představuje třída InherentanceEdge poděděná z AbstractEdge<InherentanceEdge, ClassVertex>. Graf dědičnosti se rozšiřuje při každém navštívení metody visitClassContext() třídy TestSuddenAttributesChange o analyzovanou třídu popsanou ClassContextem. Graf volání se rozšiřuje až poté co je vytvořen celý graf dědičnosti a je jim propagováno označení zajímavých tříd (interestingClasses). Je to z důvodu, že při rozšiřování grafu volání se provádí analýza u které je potřeba vědět všechny třídy přímo nebo nepřímo implementující některé rozhraní z RH_5_ANALYSE_CLASSES_WITH_INTERFACES.

4.5.2 Implementace

Třetí detektor byl implementován způsobem znázorněným na následující Ilustraci 25. Nejdřív se načtou parametry detektoru a vytvoří se graf dědičnosti. Při vytváření grafu dědičnosti se zjišťuje, které z analyzovaných tříd implementují některé z rozhraní patřící do RH_5_ANALYSE_CLASSES_WITH_INTERFACES. Když se na nějakou takovou třídu narazí, tak se vloží do seznamu interestingClasses. Poté co je vytvořen graf dědičnosti nad všemi analyzovanými třídami se zjistí, zda je seznam interestingClasses prázdný. Pokud ano, tak se analýza ukončí. V opačném případě se propaguje označení zajímavých tříd (rozšiřuje se interestingClasses) skrz graf dědičnosti. Pak se vytvoří graf volání nad všemi třídami (což zahrnuje vytvoření seznam nebezpečných metod). Označení nebezpečných metod se propaguje skrz graf volání. Nakonec se nahlásí chyby, které obsahují nebezpečné třídy a analýza končí.



Ilustrace 25: Diagram aktivit – TestSuddenAttributesChange.

Celá implementace se dá přitom rozdělit do čtyř částí. První částí je práce třídy TestSuddenAttributesChange zobrazená na Ilustraci 25. Druhá část je na Ilustraci 25 zobrazena jako

„*propaguj označení zajímavých tříd skrz graf dědičnosti*“ a podrobněji je uvedena na Ilustraci 26. Třetí část je zobrazena na Ilustraci 25 jako „*vytvoř graf volání nad všemi třídami*“ funkčně ji má na starosti metoda *CallGraph.extend()* a podrobněji ji zobrazuje Ilustrace 27. Čtvrtá část je zobrazena na Ilustraci 25 jako „propaguj označení nebezpečných metod skrz graf volání“ na starosti ji má metoda *CallGraph.flagPropagate()*. Podrobněji ji zobrazuje Ilustrace 16, která byla uvedena u 2. detektoru a protože tyto detektory využívají stejný princip propagace, tak zde nebude znovu uvedena.

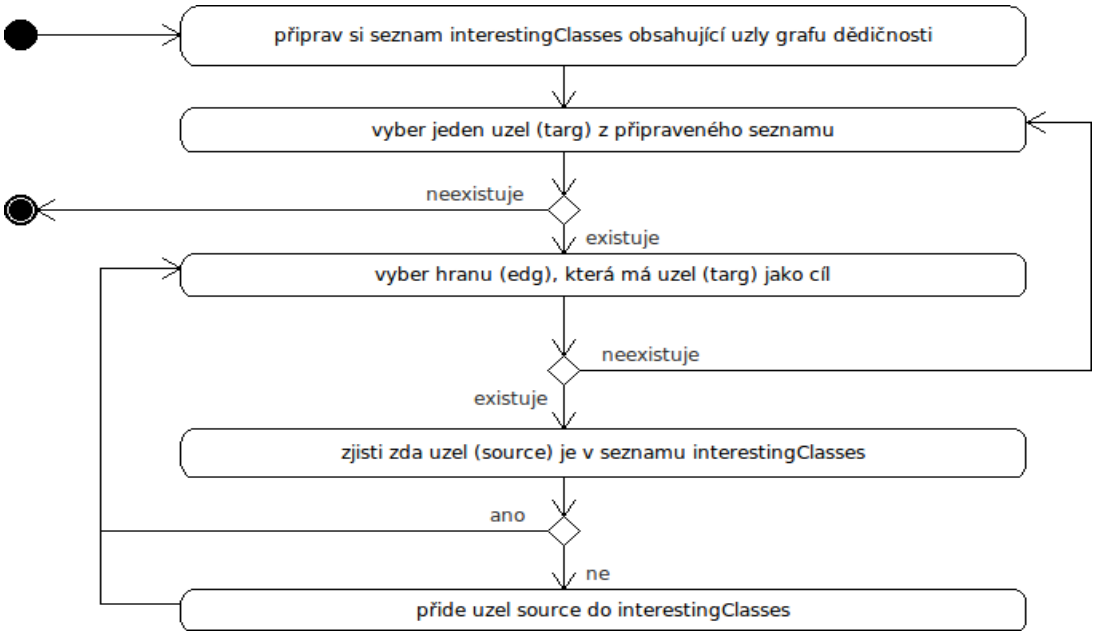
Na Ilustraci 25 je vidět, že proces analýzy začíná načtením vstupních parametrů detektoru. Parametry se zadávají jako systémové proměnné. V případě nenastavení vstupních parametrů, pracuje detektor ve standardním (implicitním) režimu. V opačném případě přizpůsobuje analýzu vstupním parametrům. Parametry, které je možné využít ke konfiguraci detektoru jsou následující:

- *RH_5_DEFINE_USE* – přepíná detektor mezi implicitním (neparametrizovaném)/explicitním (parametrizovaném) použitím. Explicitní použití detektoru je zaručeno nastavením tohoto parametru na hodnotu „1“. Implicitního použití detektoru lze docílit při zadání hodnoty „0“, nedefinováním nebo nekorektním definováním tohoto parametru.
- *RH_5_ANALYSE_CLASSES_WITH_INTERFACES* představuje množinu rozhraní. Pokud třída implementuje minimálně jedno z těchto rozhraní, tak se v této třídě bude hledat nesynchronizovaná změna atributů. Implicitně má množina pouze jeden prvek:
  - *org.jboss.soa.esb.actions.ActionPipelineProcessor*

  Explicitně se rozhraní zadává v podobném formátu nebo je možné využít formát definován v kapitole 2.1.1. Jednotlivé rozhraní se oddělují znakem „|“.

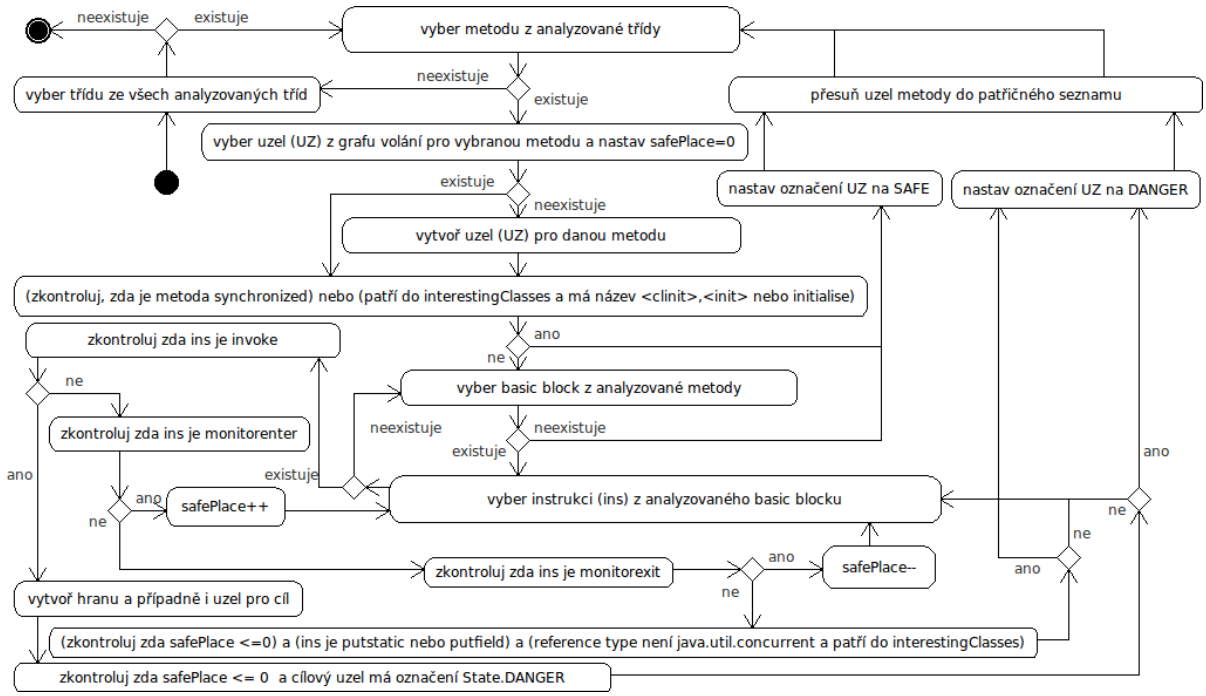
Po načtení parametrů programu se postupně pro všechny třídy vložené na vstup FindBugs vytvoří graf dědičnosti. Při jeho vytváření se kontroluje, zda některá z analyzovaných tříd implementuje rozhraní RH_5_ANALYSE_CLASSES_WITH_INTERFACES. Pokud se taková třída najde, tak je přidána do seznamu interestingClasses. Jakmile je vytváření grafu dědičnosti u konce spustí se propagace zajímavých tříd (rozšiřování interestingClasses) skrz graf dědičnosti. Tato propagace je zobrazená na Ilustraci 26 a funguje následujícím způsobem.

Propagace se spouští postupně pro všechny třídy z interestingClasses. Z grafu dědičnosti se vybere uzel (targ) odpovídající vybrané třídě z interestingClasses. Následně se najdou všechny hrany, které mají targ jako cíl. U všech takovýchto hran se zjišťuje, zda zdrojový uzel (source) je v interestingClasses. Pokud source v interestingClasses chybí, tak se do něj přidá. Celý proces propagace končí v momentě, kdy už není uzel, který by mohl být vložen do interestingClasses.



Ilustrace 26: Diagram aktivit - propaguj označení zajímavých tříd skrz graf dědičnosti.

Po propagaci označení zajímavých tříd skrz graf dědičnosti přichází na řadu vytvoření grafu volání zobrazené na Ilustraci 27.



Ilustrace 27: Diagram aktivit - vytvoř graf volání nad všemi třídami.

Graf volání se postupně vytváří pro všechny analyzované třídy. Vybere se metoda analyzované třídy. Pokud existuje uzel (UZ) pro metodu, tak se vybere. V opačném případě se UZ vytvoří. Proměnná safePlace se inicializuje na nulu. Tato proměnná značí svou kladnou hodnotou, že se při analýze nacházíme v synchronizovaném bloku (mezi instrukcemi monitorenter a monitorexit). Pokud je analyzovaná metoda synchronized, tak se její označení nastaví na MethodState.SAFE a UZ se přesune do seznamu bezpečných metod safeMethods. Nastavení s přesunem UZ se provede i když metoda patří do třídy z interestingClasses a její jméno je <clinit>,<init>, initialise nebo destroy.

Pro metodu nenastavenou na MethodState.SAFE se začne s CFG analýzou. Vybere se vstupní basic block a začnou se testovat jeho instrukce. Pro analýzu je důležitá detekce následujících instrukcí: invoke, monitorenter, monitorexit, putstatic, putfield. Pokud je testovaná instrukce invoke instrukcí vytvoří se hrana a pokud neexistuje, tak i uzel cíle. V případě, že se invoke instrukce nenachází v synchronizovaném bloku (safePlace <= 0) a cílový uzel je označen jako MethodState.DANGER, nastaví se i označení UZ na hodnotu MethodState.DANGER a přesune se do seznamu nebezpečných metod dangerMethods. Pro metody z této analýzy budou na konci analýzy vygenerovány chybové hlášení prostřednictvím BugReporteru. Při nalezení instrukce monitorenter se inkrementuje proměnná safePlace (značící při své kladné hodnotě, že se instrukce nachází v synchronizovaném bloku). Když je testovaná instrukce monitorexit, tak se dekrementuje proměnná safePlace. Pokud se při detekci instrukcí narazí na putstatic nebo putfield a platí, že se instrukce nachází v nesynchronizovaném bloku (safePlace <= 0) nepracuje s objektem z java.util.concurrent a pracuje s objektem třídy patřící do interestingClasses, tak je UZ označeno hodnotou MethodState.DANGER. UZ se přesune do dangerMethods.

Když testy instrukce nezpůsobí označení UZ na MethodState.DANGER, tak se vezme následující instrukce z BB a celý proces se opakuje. Po vyčerpání všech instrukcí z BB se vygenerují basic blocky do kterých je možné se dostat z BB. Do těchto basic blocků se předá hodnota safePlace a provádí se jejich analýza. Vytváření grafu volání skončí tehdy když pokrývá všechny třídy (reprezenované třídou ClassContext), které dostal detektor přes metodu visitClassContext() od jádra nástroje FindBugs.

Po tomto vygenerování grafu volání se spustí propagace označení MethodState.DANGER. Tato propagace skrz graf volání probíhá stejně jako ve 3. detektoru a názorně ji popisuje Ilustrace 16. Propagací se může zvýšit počet metod v dangerMethods. Nakonec se ještě v metodě report() třídy TestSuddenAttributesChange vytvoří chybová hlášení pro všechny metody z dangerMethods a detektor skončí. Detektor produkuje chybové hlášení ve formátu popsaném následujícím příkladem: „H C PV5: Found sudden attributes change in testCases.TestRH5example4.firstMethodDanger() At TestRH5example4.java:[lines 12-13]“ kde „H C PV5“ identifikuje hlášení 5. detektoru nasledované popisem a pozicí chyby.

5 Návod k použití detektorů

Kapitola popisuje požadavky potřebné k bezproblémovému běhu detektorů a postup jak spustit detektory. Pak se zde taky nachází popis chybových hlášení generovaných detektory.

Před spuštěním testů je potřeba mít k dispozici:

- *virtuální stroj* kompatibilní se Sun's JDK 1.4 a vyšší,
- nástroj *FindBugs* 1.3.2 a vyšší,
- detektory (pluginy): *FindStaticMethodClassForName.jar, FindCourierUseOutOfInvoker.jar, TestMethodGetPayLoad.jar ,FindPoorTestOfObject.jar ,TestSuddenAttributesChange.jar* umístěné v adresáři plugin nacházející se v domovské složce FindBugs,
- testovanou aplikaci v byte kódu nejlépe v .jar balíku.

Virtuální stroj kompatibilní s JDK 1.4 je možné stáhnout na stránkách společnosti Sun. Projekt byl vyvíjen a testován pod Java(TM) SE Runtime Environment (build 1.6.0_15-b03). Nástroj FindBugs lze získat z domovských stránek nástroje. Při vývoji detektorů byla využita verze 1.3.2-dev-20080405.

Všechny detektory se dají z adresáře bin nacházejícího se v domovském adresáři FindBugs spustit příkazem:

„./findbugs -textui -visitors FindStaticMethodClassForName,FindCourierUseOutOfInvoker,TestMethodGetPayLoad,FindPoorTestOfObject,TestSuddenAttributesChange TestCases.jar“

Příkaz říká, že se bude spouštět nástroj FindBugs v textovém režimu s prvním až pátým detektorem nad balíkem *TestCases.jar*.

Chybové hlášení produkované detektory jsou shrnuty v Tabulce 7.

Detektor	**Identifikace hlášení**	**Popis chyby**
FindStaticMethodClassForName	H C PV1	Nalezeno zakázané volání statické metody.
FindCourierUseOutOfInvoker	H C PV2	Nalezeno volání metody mimo povolené třídy.
TestMethodGetPayLoad	H C PV3	Chybí volání getPayLoad() před použitím objektu.
FindPoorTestOfObject	H C PV4	Nalezeno špatné testování objektů.
TestSuddenAttributesChange	H C PV5	Nalezena nesynchronizovaná změna atributů.

Tabulka 7: Chybové hlášení.

Detektory je mimo to možné za pomocí využití systémových proměnných spustit v explicitním režimu. Příklad takovéhoto spuštění pro 1. detektor může vypadat např. takto: „

./findbugs -textui -visitors FindStaticMethodClassForName TestCases.jar
-DRH_1_PROHIBITED_METHODS=org/jboss/soa/esb/couriers/CourierFactory.getInstanc
e:\()Lorg/jboss/soa/esb/couriers/CourierFactory\;
-DRH_1_CLASSPATH_PREFIX=testCases -DRH_1_DEFINE_USE=1 “

Takového spuštění bude hledat statické volání *CourierFactory.getInstance()* v třídách jejichž cesta začíná řetězcem *testCases*.

Parametry jednotlivých detektorů byly podrobně představeny v části zabývající se implementací detektorů začínající na straně 34. Souhrnně jsou zobrazeny v Tabulce 21 nacházející se v přílohách.

6 Experimenty

Tato kapitola shrnuje experimenty provedené s vytvořenými detektory na byte-kódu od:

- jbossesb-4.0.GA – rosetta
- jbossesb-4.4.GA – rosetta
- jbossesb-4.7 – rosetta = 701 tříd a 5081 metod
- Drools verze 4.0.4
- Drools verze 4.0.5 = 943 tříd a 8804 metod
- breakingwoods (jbossesb-apachecamel-sample, jboss-esb-ejb-provider_1.0, JBoss ESB Splitter 1.0, jbossesb-apache-dbutilsv1.0, googlespreadsheet_gateway, mail_gateway_v1.0, twitter-adapter-v1.0, esbg-project-generator_v1.27_JBossESB4.7) = 39 tříd a 291 metod

6.1 Testování FindStaticMethodClassForName

Výsledky experimentů 1. detektoru zobrazené v Tabulce 8 a Tabulce 9 ukazují, že při testování jbossesb-4.4.GA – rosetta byly nalezeny 2 staré výskyty chyb (nahlášené pod SOA-795), které v nové verzi komponenty Jbossesb-4.7 – rosetta již nalezeny nebyly. Dále bylo nalezeno 7 starých výskytu chyb v jbossesb-4.0.GA – rosetta, které se v nové verzi již nevyskytují. Detektoru se dokonce podařilo najít jeden nový výskyt chyby v projektu spadající do breakingwoods konkrétně v metodě EJBProvider.doInitialise(). Chybové hlášení vyprodukované detektorem při testování jdou vidět v Tabulce 8 a Tabulce 9 . Zkratky z pravého sloupec tabulky reprezentují: nb – ukazuje počet nových chyb, rb – počet starých chyb, fa – počet vyprodukovaných false alarmů.

Co se testovalo	**Výstup detektoru**	**nb/rb/fa**
jbossesb-4.4.GA - rosetta	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.actions.EJBProcessor.process(Message) At EJBProcessor.java:[lines 96-121]	**0/2/0**
	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.actions.EJBProcessor.invoke(Class, Object, String, Object[]) At EJBProcessor.java:[lines 233-248]	
jbossesb-4.7 - rosetta		0/0/0
breakingwoods	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.providers.EJBProvider.doInitialise() At EJBProvider.java:[lines 69-86]	**1/1/0**

Tabulka 8: Výsledky testování prvního detektoru – FindStaticMethodClassForName.

Co se testovalo	Výstup detektoru	nb/rb/fa
Drools verze 4.0.4		0/0/0
Drools verze 4.0.5		0/0/0
jbossesb-4.0.GA – rosetta	H C PV: Found prohibited method call (Class.forName()) in org.jboss.internal.soa.esb.message.format.MessageFactoryImpl.reset() At MessageFactoryImpl.java:[lines 54-104]	**0/7/0**
	H C PV: Found prohibited method call (Class.forName()) in new org.jboss.soa.esb.helpers.persist.SimpleDataSource(String, String, String, String) At SimpleDataSource.java:[lines 113-124]	
	H C PV: Found prohibited method call (Class.forName()) in new org.jboss.internal.soa.esb.message.format.xml.marshal.MarshalUnmarshalManager() At MarshalUnmarshalManager.java:[lines 52-102]	
	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.notification.NotificationTarget.fromParams(ConfigTree) At NotificationTarget.java:[lines 104-151]	
	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.parameters.ParamRepositoryFactory.synchedGetInstance() At ParamRepositoryFactory.java:[lines 69-105]	
	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.services.persistence.MessageStoreFactory.<static initializer>() At MessageStoreFactory.java:[lines 42-50]	
	H C PV: Found prohibited method call (Class.forName()) in org.jboss.soa.esb.helpers.persist.SqlDbTable.initFields() At SqlDbTable.java:[lines 178-201]	

Tabulka 9: Výsledky testování prvního detektoru – FindStaticMethodClassForName.

Největší úspěch detektor dosáhl při analýze breakingwoods/jboss-esb-ejb-provider, kde se mu podařilo najít jednu novou chybu, která byla potvrzena zaměstnancem Red Hatu.

6.2 Testování FindCourierUseOutOfInvoker

Výsledky experimentů 2. detektoru zobrazené v Tabulce 10 a Tabulce 11 ukazují, že při testování jbossesb-4.4.GA – rosetta bylo nalezeno 9 starých výskytů chyb, které v nové verzi komponenty Jbossesb-4.7 – rosetta již nalezeny nebyly. Ve *jbossesb-4.0.GA – rosetta* byla nalezena 1 stará chyba, která v Jbossesb-4.7 – rosetta již nalezena nebyla. Žádné nové chyby tento detektor nenašel.

Co se testovalo	Výstup detektoru	nb/rb/fa
jbossesb-4.0.GA – rosetta	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.couriers.JmsCourier$1.onException(JMSException) At JmsCourier.java:[lines 495-496]	**0/1/0**
jbossesb-4.4.GA - rosetta	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor.unRegisterService(String, String) At InVMRegistryInterceptor.java:[lines 56-64]	**0/9/0**
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.addressing.eprs.DefaultInVMReplyToEpr.getThreadEPR() At DefaultInVMReplyToEpr.java:[lines 41-57]	
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor._unRegisterEPR(String, String, EPR) At InVMRegistryInterceptor.java:[lines 84-102]	
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.couriers.tx.InVMXAResource.commit(Xid, boolean) At InVMXAResource.java:[lines 73-95]	
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.couriers.tx.InVMXAResource.rollback(Xid) At InVMXAResource.java:[lines 135-170]	
	H C PV2: Found prohibited method call (courier out of invoker) in new org.jboss.internal.soa.esb.addressing.eprs.DefaultInVMReplyToEpr(InVMEpr) At DefaultInVMReplyToEpr.java:[lines 35-36]	
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor.unRegisterInVMService(String, String) At InVMRegistryInterceptor.java:[lines 67-71]	
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.internal.soa.esb.services.registry.InVMRegistryInterceptor.unRegisterEPR(String, String, EPR) At InVMRegistryInterceptor.java:[lines 76-81]	
	H C PV2: Found prohibited method call (courier out of invoker) in org.jboss.soa.esb.addressing.util.DefaultReplyTo.getReplyTo(EPR) At DefaultReplyTo.java:[lines 58-88]	

Tabulka 10: Výsledky testování druhého detektoru – FindCourierUseOutOfInvoker.

Co se testovalo	Výstup detektoru	nb/rb/fa
Jbossesb-4.7 - rosetta		**0/0/0**
breakingwoods		0/0/0
Drools verze 4.0.4		0/0/0
Drools verze 4.0.5		0/0/0

Tabulka 11: Výsledky testování druhého detektoru – FindCourierUseOutOfInvoker.

6.3 Testování TestMethodGetPayLoad.

Výsledky experimentů 3. detektoru zobrazené v Tabulce 12 ukazují, že při testování výše zmíněných balíků detektor nenalezl žádnou chybu.

Co se testovalo	Výstup detektoru	nb/rb/fa
Jbossesb-4.7 - rosetta		0/0/0
jbossesb-4.4.GA – rosetta		0/0/0
jbossesb-4.0.GA – rosetta		0/0/0
Drools verze 4.0.4		0/0/0
Drools verze 4.0.5		0/0/0
breakingwoods		0/0/0

Tabulka 12: Výsledky testování třetího detektoru – TestMethodGetPayLoad.

6.4 Testování FindPoorTestOfObject

Výsledky experimentů 4. detektoru jsou zobrazené v Tabulce 13. Byl proveden test komponenty Drools verze 4.0.4 obsahující chybu JBRULES-1429. Tato chyba byla opravena v Drools verzi 4.0.5, která byla také otestována. To jak testování dopadlo shrnuje Tabulka 13.

Tabulka 13 zobrazuje, že chyba v Drools verzi 4.0.4 byla nalezena a v Drools verzi 4.0.5 se už nevyskytuje. Dále tato tabulka ukazuje, že při analýze komponenty rosetta (Jbossesb-4.7 – rosetta, jbossesb-4.4.GA – rosetta, jbossesb-4.0.GA – rosetta) byl nalezen jeden false alarm, který v této komponentě přervává ve všech otestovaných verzích. V metodě (equalsObject(final Object lhs, final Object rhs)) totiž sice dochází k situaci, kdy před porovnáním argumentů není provedeno testování argumentu rhs na null, metoda je ovšem vždy volána z místa, před kterým se toto testování na null provádí, takže jeho absence v metodě equalsObject nevadí.

Co se testovalo	Výstup detektoru	nb/rb/fa
Jbossesb-4.7 - rosetta	H C PV4: Found poor objects test in org.jboss.soa.esb.addressing.PortReference.equalsObject (Object, Object) At PortReference.java:[lines 426-431]	0/0/1
jbossesb-4.4.GA – rosetta	H C PV4: Found poor objects test in org.jboss.soa.esb.addressing.PortReference.equalsObject (Object, Object) At PortReference.java:[lines 426-431]	0/0/1
jbossesb-4.0.GA – rosetta	H C PV4: Found poor objects test in org.jboss.soa.esb.addressing.PortReference.equalsObject (Object, Object) At PortReference.java:[lines 381-386]	0/0/1
breakingwoods		0/0/0
Drools verze 4.0.5		**0/0/0**
Drools verzi 4.0.4	H C PV4: Found poor objects test in org.drools.base.evaluators.ObjectFactory$ObjectEqualsComparator.equals(Object, Object) At ObjectFactory.java:[lines 547-578]	**0/1/0**

Tabulka 13: Výsledky testování čtvrtého detektoru – FindPoorTestOfObject.

6.5 Testování TestSuddenAttributesChange

Výsledky experimentů 5. detektoru jsou zobrazené v Tabulkách 14 a 15.

Co se testovalo	Výstup detektoru	nb/rb/fa
Jbossesb-4.7 - rosetta	H C PV5: Found sudden attributes change in org.jboss.soa.esb.mock.MockAction.process(Message) At MockAction.java:[lines 41-49] H C PV5: Found sudden attributes change in org.jboss.soa.esb.actions.scripting.GroovyActionProcessor.getScript(Message) At GroovyActionProcessor.java:[lines 168-194] H C PV5: Found sudden attributes change in org.jboss.soa.esb.actions.scripting.GroovyActionProcessor.process(Message) At GroovyActionProcessor.java:[lines 138-163]	**3/0/0**
jbossesb-4.4.GA – rosetta	H C PV5: Found sudden attributes change in org.jboss.soa.esb.mock.MockAction.process(Message) At MockAction.java:[lines 41-49]	**3/0/0**
	H C PV5: Found sudden attributes change in org.jboss.soa.esb.actions.scripting.GroovyActionProcessor.getScript(Message) At GroovyActionProcessor.java:[lines 165-191]	
	H C PV5: Found sudden attributes change in org.jboss.soa.esb.actions.scripting.GroovyActionProcessor.process(Message) At GroovyActionProcessor.java:[lines 136-160]	

Tabulka 14: Výsledky testování pátého detektoru – TestSuddenAttributesChange.

Testování tohoto detektoru se dá považovat za nejúspěšnější protože se pomoci něj podařily najít tři chyby v *Jbossesb-4.7-rosetta*. Ve skutečnosti chyby objevující se v metodě *process()* a getScript() jsou na sobě závislé, takže se jedná o chybu jednu propagovanou přes graf volání.

Co se testovalo	**Výstup detektoru**	**nb/rb/fa**
jbossesb-4.0.GA – rosetta		0/0/0
Drools verze 4.0.4		0/0/0
Drools verze 4.0.5		0/0/0
breakingwoods		0/0/0

Tabulka 15: Výsledky testování pátého detektoru – TestSuddenAttributesChange.

7 Závěr diplomové práce

Hlavním úkolem práce je hledání častých chyb v JBoss aplikačním serveru od firmy Red Hat pomocí zde navržených FindBugs detektorů. Častá chyba je taková, která se v JBoss aplikačním serveru opakuje (nebo se její opakující výskyt v budoucnu předpokládá) a vytvořený detektor slouží k indikaci jejího dalšího možného výskytu. Chyby, pro které se vytvářely detektory, byly čerpány z bug tracking systému JIRA a na základě konzultací se zaměstnancem firmy Red Hat.

První implementovaný detektor je schopný hledat nechtěné volání statické metody `Class.forName()` v projektu JBossESB. Testy vytvořeného detektoru prokazují, že je schopen nalézt dříve nahlášené instance chyby. *Nová (nenahlášená) instance této chyby byla nalezena v projektu breakingwoods.* Chybová hlášení jsou zobrazena v Tabulce 8.

Druhý implementovaný detektor je schopný hledat použití `CourierFactory` a tříd implementujících `DeliverOnlyCouriers` nebo `PickupOnlyCouriers` mimo předem definované třídy zobrazené v kapitole 3.2. Testy vytvořeného detektoru ukazují, že je schopen tento druh chyb hledat. Detektoru se při analýze podařilo nalézt pouze staré (nahlášené) výskyty této chyby.

Třetí implementovaný detektor je schopen hledat volání metod třídy `Message` a to v případě, že se volání nacházejí v metodě se signaturou `(Lorg/jboss/soa/esb/message/Message;)Lorg/jboss/soa/esb/ message/Message;` patřící do třídy implementující `ActionPipelineProcessor` a před tímto voláním chybí volání `getPayload(Message)`. Testy prokázaly, že detektor je schopen hledat tento druh chyb. Nová (nenahlášená) instance chyby ovšem nebyla nalezena.

Čtvrtý implementovaný detektor je schopen hledat nedokonalé testování argumentů na null před jejich porovnáním pomocí metody `equals()`. Testy ukazují, že detektor je tento druh chyb schopen hledat. Detektor při analýze našel ovšem pouze staré, již nahlášené chyby.

Pátý implementovaný detektor je schopen hledat nesynchronizovanou změnu atributů (v případě, že změna atributů není v synchronizovaném bloku) třídy `ActionPipelineProcessor`. Testy ukazují, že detektor umí tento druh chyb hledat. *Detektoru se podařilo najít tři nové nesynchronizované změny atributů (v komponentě Rosetta od jbossesb-4.7),* které byly nahlášeny a potvrzeny jako chyby. Chybová hlášení jsou vidět v Tabulce 14.

Všechny detektory je mimo spuštění v standardním režimu možné spustit v režimu explicitním a přizpůsobit si tak detekci vlastním potřebám. Parametry využitelné při explicitním spouštění jsou k nalezení v Tabulce 21. Podrobnější popis parametrů je možné nalézt v sekci zabývající se implementací u jednotlivých detektorů.

Při analýze v praxi využívaného software (rosetta, breakingwoods) se podařilo najít **4 nové výskyty chyb**. Výsledky práce byly publikovány na konferenci EEICT 2010 [73]. Přínos práce vidím ve vytvoření znovupoužitelných detektorů, které mohou být využity při dalším vývoji JBoss

aplikačního serveru. Detekce nových chyb jistě přispěje ke zkvalitnění produktů firmy Red Hat a lepšímu rozvoji open-source. Dále si velice cením nových zkušeností se statickou analýzou a nástrojem FindBugs získaných díky této práce. V neposlední řadě jsem rád, že jsem si vyzkoušel, jak mohou být ve škole nabyté vědomosti, použity v praxi.

Literatura

[1] Seneca Annaeus Lucius. Slova tesaná do mramoru. Praha. Vyšehrad. 2000. ISBN: 978-80-7021-438-4

[2] P. Vyvial: Statická analýza Java programů, bakalářská práce, Brno, FIT VUT v Brně, 2008.

[3] NASA. National Aeronautics and Space Administration – MARIN1. [Online], [rev. 2009-06-29], [cit. 2009-11-04]. Dostupné na URL: <http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1>

[4] D. E. Hoffman. WashPost – CIA slipped bugs to Soviets. [Online], [rev. 2009-11-04], [cit. 2009-11-04]. Dostupné na URL: <http://www.msnbc.msn.com/id/4394002>

[5] J. L. Lions. Massachusestts Institute of Technology – Ariane 5 Flight 501 Failure – Full Report. [Online], [rev. 1996-07-16], [cit. 2009-11-04]. Dostupné na URL: <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html>

[6] K. Richta, J. Sochor: Softwarové inženýrství I. Vydavatelství ČVUT, Praha 1996 (dotisk 1998). ISBN: 80-01-01428-2

[7] D. Engler and M. Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In Proc. of VMCAI'04, volume 2937 of LNCS. Springer, 2004.

[8] New York Independent System Operator. Interim Report August 14, 2003 Blackout. New York. 2004.

[9] F. Nielson, H.R. Nielson, and C. Hankin. Principles of Program Analysis, Springer-Verlag, 2005.

[10] JBoss community. Webové stránky komunity JBoss aplikačního serveru. [Online], [rev. 2009-11-06], [cit. 2009-11-07]. Dostupné na URL: <http://www.jboss.org/>

[11] Pavel Vyvial. Healing assurance in java programs. In Proceedings of the 14th conference Student EEICT 2008, volume 1, pages 204 – 206. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií a Fakulta informačních technologií, 2008.

[12] Chess, B., West,J.: Secure Programming with Static Analysis. Addison-Wesley Professional, 2007.

[13] Oliver Burn. Checkstyle – checkstyle 4.4. [Online], Verze 5.0 (2009), [rev. 2009-10-20], [cit. 2009-11-14]. Dostupné na URL: <http://checkstyle.sourceforge.net/>

[14] Bill Pugh, David Hovemeyer, and Ben Langmead. FindBugs – find bugs in java programs. [Online], Verze 1.3.9 (2009), [rev. 2009-08-22], [cit. 2009-11-14]. Dostupné na URL: <http://findbugs.sourceforge.net/>

[15] Neal Ford. Intellij idea – the most intelligent java ide. [Online], Verze 8.1.4 (2009), [rev. 2009-11-13], [cit. 2009-11-14]. Dostupné na URL: <http://www.jetbrains.com/idea/>

[16] David Dixon-Peugh, Tom Copeland, and Xavier Le Vourch. Pmd. [Online], Verze 4.2.5 (2009), [rev. 2009-02-08], [cit. 2009-11-14]. Dostupné na URL: <http://pmd.sourceforge.net/>

[17] Oliver Cole, Chris Elford, and Harm Sluiman. Tptp. [Online], Verze 4.6.1 (2009), [rev. 2009-09-25], [cit. 2009-11-14]. Dostupné na URL: <http://www.eclipse.org/tptp/>

[18] Debra J. Richardson. Static Analysis – Analysis of Models, Data flow analyses, Finite-state verification. rev. [2000-05-09], [cit. 2009-11-14].
Dostupné na URL: <http://www.ics.uci.edu/~djr/classes/ics224/lectures/08-StaticAnalysis.pdf>

[19] Adam Kolawa. When, Why and How: Code Analysis. Parasoft Corporation. Monrovia. 2008.

[20] Markus Dahm. Bcel – byte code engineering library. [Online], Verze 5.2 (2006), [rev. 2006-06-03], [cit. 2009-11-15]. Dostupné na URL: <http://jakarta.apache.org/bcel/>

[21] Eric Bruneton, Eugene Kuleshov, and Andrei Loskutov. Asm – java bytecode manipulation and analysis framework. [Online], Verze 3.2 (2009), [rev. 2009-06-11], [cit. 2009-11-15]. Dostupné na URL: <http://asm.ow2.org/>

[22] Maarten Coene. Dom4j – flexible xml framework for java. [Online], Verze 2.0 (2008), [rev. 2009-10-15], [cit. 2009-11-15]. Dostupné na URL: <http://www.dom4j.org/>

[23] David Hovemeyer. The architecture of findbugs. [Online], [rev. 2008-01-09], [cit. 2009-11-16]. Dostupné na URL: <http://www.cs.vassar.edu/hovemeye/architecture.pdf>

[24] JBoss community. JBoss Application Server Installation And Getting Start Guide. [Online], [rev. 2009-11-17],[cit. 2009-11-17]. Dostupné na URL: <https://www.jboss.org/community/docs/DOC-12923>

[25] JBoss community. Webové stránky JBoss aplikačního serveru. [Online], [rev. 2009-11-17], [cit. 2009-11-17]. Dostupné na URL: <http://www.jboss.com/>

[26] Javid Jamae and Peter Johnson.: JBoss in Action. Configurating the JBoss Application Server. 2009. ISBN: 1933988029.

[27] Dimitris Andreadis. JBoss Application Server 5 and Beyond. Slajdy z JAZOON09 – The International Conference on Java technology. Zurich. 2009.

[28] JBoss community. JBoss AOP – User Guide – The Case For Aspects. [Online], [rev. 2009-11-19], [cit. 2009-11-20]. Dostupné na URL: <http://www.jboss.org/file-access/default/members/jbossaop/freezone/docs/2.0.0.GA/docs/aspect-framework/userguide/en/pdf/jbossaop_userguide.pdf >

[29] JBoss community. Webové stránky projektu Hibernate. [Online], [rev. 2009-11-18], [cit. 2009-11-20]. Dostupné na URL: <https://www.hibernate.org/>

[30] Miko Matsumura. JBoss Application Server – JBoss White paper. [Online], [rev. 2009-11-20], [cit. 2009-11-21]. Dostupné na URL: <http://www.jboss.com/pdf/JBossAS-EnterpriseInfrastructure.pdf >

[31] John R. Koza. Genetic Programming: A Paradigm For Genetically Breeding Populations Of Computer Programs To Solve Problems. Stanford University. Stanford. 1990.

[32] Michael Newman. Software Errors Cost U.S. Economy $59.5 Billion Annually. [Online],

[rev. 2002-06-28], [cit. 2009-12-23].
Dostupné na URL: <http://www.nist.gov/public_affairs/releases/n02-10.htm>

[33] Arun Kumar. Software bug and their common types. [Online], [rev. 2009-05-07], [cit. 2009-12-25]. Dostupné na URL: <http://www.articlealley.com/article_881983_11.html>

[34] Law A., Kelton D.: Simulation Modelling and Analysis, McGraw-Hill, 1991

[35] Berard B., Bidoit M., Finkel A. Laroussinie F., Petit A. Petrucci L., Schnoebelen P.: Systems and Software Verification. Springer-Verlag New Yourk Inc. New York. NY. 2001.

[36] Cem Kaner: Exploratory Testing. Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, November 2006

[37] Duffy, David A.: Principles of Automated Theorem Proving. John Wiley & Sons. 1991.

[38] C. Baier, J.-P. Katoen. Principles of Model Checking. MIT Press. 2008.

[39] C. Kaner, J. Falk, and H. Nguyen.: Testing Computer Software. Wiley. 1999.

[40] Cornell, C., Horstmann, C.: Core Java (2nd ed.), SunSoft Press, Prentice Hall, ISBN 0-13-596891-7, 1997.

[41] Grand, M.: Java Language Reference (2nd ed.), ISBN 1-56592-326-X, O'Reilly & Associates, 1997.

[42] Symantec community. Symantec's Just-In-Time Java Compiler To Be Integrated Into Sun JDK 1.1. [Online], [rev. 1997-04-07], [cit. 2009-12-27]. Dostupné na URL: <http://www.symantec.com/about/news/release/article.jsp?prid=19970407_03>

[43] Jones, Richard, and Rafael Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, New York, 1996.

[44] Peter Haggar: Java bytecode – Understanding bytecode makes you a better programmer. [Online], [rev. 2001-07-01], [cit. 2009-12-27]. Dostupné na URL:

<http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/>

[45] Marrs T., Davis S.: JBoss at Work: A Practical Guide, ISBN 0596007345, O'Reilly Media, Inc. 2005.

[46] Bader D. A., Pennington R.: Cluster Computing: Applications. The International Journal of High Performance Computing, 15(2):181-185. 2001.

[47] Horne G. E., Meyer T. E.: Data Farming: Discovering surprise. In Proceedings of the 2004 Winter Simulation Conference, Woodbridge, U.S.A, 2004.

[48] Weiß C., Premraj R., Zimmermann T., Zeller A.: How Long will it Take to Fix This Bug?. In Proceedings of the Fourth International Workshop on Mining Software Repositories. Computer Society. 2007.

[49] Monson-Haefel R., Burke B.: Enterprise JavaBeans 3.0, Fifth Edition. Developing Enterprise Java Components. O'Reilly Media. 2006.

[50] Hirt A.: Pro SQL Server 2008 Failover Clustering (Expert's Voice in SQL Server). Apress. ISBN13: 978-1-4302-1966-8. 2009.

[51] Cote M.: Java Authentication and Authorization Service (JAAS) in Action © 2005. [Online], [rev. 2009-10-27], [cit. 2010-01-02]. Dostupné na URL: <http://www.jaasbook.com/>

[52] Moncillo R., Sun Microsystems, Inc.: Java Community ProcessSM (JCPSM) 2.1. Final Release. Version 1.0. [Online], [rev. 2003-11-24], [cit. 2010-01-02].Dostupné na URL: <http://java.sun.com/j2ee/javaacc/index.html>

[53] The O'Reilly Java Authors.: Java Enterprise Best Practices. O'Reilly Media, Inc.. ISBN-13: 978-0-596-00384-5. 2002.

[54] Bergsten H.: JavaServer Faces. O'Reilly Media. 2004.

[55] Sharma R., Stearns B., Ng T.: J2EE(TM) Connector Architecture and Enterprise Application Integration. Addison-Wesley Professional. ISBN:0201775808. 2001.

[56] Reese G.: Database Programming with JDBC and Java. O'Reilly Media. ISBN-13: 978-1565926165. 2000.

[57] Chappell A D., Monson-Haefel R.: Java Message Service. O'Reilly Media. 2000.

[58] Sing L. et al.: Professional Java Server Programming: with Servlets, JavaServer Pages (JSP), XML, Enterprise JavaBeans (EJB), JNDI, Corba, Jini and Javaspaces. Peer Information Inc. ISBN-13:978-1861002778. 1999.

[59] Little M., Maron J., Pavlik G.: Java Transaction Processing: Design and Implementation. Prentice Hall PTR. ISBN-13: 978-0130352903. 2004.

[60] Kiselev I.: Aspect-Oriented Programming with AspectJ. Sams. ISBN-13:978-0672324109. 2002.

[61] Hansen D. M.: SOA Using Java(TM) Web Services. Prentice Hall PTR. ISBN-13:978-0130449689. 2007.

[62] Grosso W.: Java RMI. O'Reilly Media. 2001.

[63] Monson-Haefel R.: J2EE Web Services: XML, SOAP, WSDL, UDDI, WS-I, JAX-RPC, JAXR, SAAJ, JAXP. Addison-Wesley Professional. ISBN-13: 978-0321146182. 2003.

[64] Nickull D., Hinchcliffe D., Governor J.: Web 2.0 Architectures: What Entrepreneurs and information Architects Nedd to Know. O'Reilly. ISBN-13: 978-0596514433. 2009.

[65] McAffer J., VanderLei P., Archer S.: OSGi and Equinox: Creating Highly Modular Java Systems. Addison-Wesley Professional. ISBN-13: 978-0321585714. 2009.

[66] Harrop R., Machacek J.: Pro Spring. Apress. ISBN-13: 978-1590594612. 2005.

[67] Fleury M., Stark S., Richards N., JBoss, Inc.: JBoss® 4.0 The Official Guide. Sams. ISBN-13: 978-0-672-32648-6. 2005.

[68] Joseph J., Fellenstein C.: Grid Computing. IBM Press. ISBN-13: 978-0-13-145660-0. 2003.

[69] Maheswari U. J. And Varghese K.: A Structured Approach to Form Dependency Structure Matrix for Construction Projects. In Proceedings of the 22nd International Symposium on Automation and Robotics in Construction, Italy, 2005.

[70] Kay M.: Xpath 2.0 Programmer's Reference (Programmer to Progreammer). Wrox. ISBN-13: 978-0764569104. 2004.

[71] GNU: GNU Lesser General Public License. [Online], [rev. 2007-6-29], [cit. 2010-01-08]. Dostupné na URL: <http://www.gnu.org/copyleft/lesser.html>

[72] Ray Gans. Developer resources for java technology. [Online], Verze 6u20 (2010), [rev. 2010-04-30], [cit. 2010-05-07]. Dostupné na URL: <http://java.sun.com/>

[73] Pavel Vyvial. Static detection Of Common Bugs In JBoss Application Server. In Proceedings of the 16th conference Student EEICT 2010, volume 3, pages 133 – 135. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií a Fakulta informačních technologií, 2010.

[74] Bz Research LLC. Third Annual Java Use and Awareness Study. SD Times. 2004.

Seznam příloh

Příloha 1. Zdrojové texty testovacích tříd společně s výsledky testů.

Příloha 2. Seznam použitelných parametrů jednotlivých detektorů.

Příloha 3. CD obsahuje

- zdrojové texty práce – /src
- spustitelné detektory ve formě pluginu – /bin
- zdrojové texty testovacích tříd – /test
- elektronickou podobu technické zprávy – /technickaZprava.pdf
- elektronickou podobu technické zprávy (upravitelná verze) - /doc/technickaZprava.odt
- programovou dokumentaci – /doc

Dodatek A

Zdrojové kódy testovacích tříd

V této příloze jsou k nalezení zdrojové kódy testovacích tříd společně s chybovými hlášeními vyprodukovanými detektory při testování.

Výsledek testu a testovací třída 1. detektoru

Tabulka 16 zobrazuje výsledky testu provedeného s 1. detektorem (FindStaticMethodClassForName) nad testovací třídou (TestRH1example1) zobrazenou na Ilustraci 28.

Co se testovalo	**Výstup detektoru**	**Počet chyb**
TestRH1example1	H C PV: Found prohibited method call (Class.forName()) in org.jboss.internal.soa.esb.couriers.TestRH1example1.firstMethodDANGER() At TestRH1example1.java:[lines 10-14]	1

Tabulka 16: Výsledky testování druhého detektoru – FindStaticMethodClassForName.

```
package org.jboss.internal.soa.esb.couriers;

public class TestRH1example1 {

	public TestRH1example1(){

	}

	public void firstMethodDANGER(){
		try {
			Class.forName("");
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
	}

}
```

Ilustrace 28: Testovací třída – org.jboss.internal.soa.esb.couriers.TestRH1example1.

Výsledek testu a testovací třídy 2. detektoru

Tabulka 17 zobrazuje výsledky testu provedené s 2. detektorem (FindCourierUseOutOfInvoker) nad testovacími třídami (TestRH2example1, TestRH2example2, TestRH2example3) zobrazenými na Ilustracích 29, 30 a 31.

Co se testovalo	**Výstup detektoru**	**nb/rb/fa**
TestRH2example1, TestRH2example2, TestRH2example3	H C PV2: Found prohibited method call (courier out of invoker) in testCases.TestRH2example1.firstMethodDANGER() At TestRH2example1.java:[lines 11-12] H C PV2: Found prohibited method call (courier out of invoker) in testCases.TestRH2example1.fourthMethodDANGER() At TestRH2example1.java:[lines 24-25] H C PV2: Found prohibited method call (courier out of invoker) in testCases.TestRH2example1.secondMethodDANGER() At TestRH2example1.java:[lines 15-16] H C PV2: Found prohibited method call (courier out of invoker) in testCases.TestRH2example1.thirdMethodDANGER() At TestRH2example1.java:[lines 19-21]	0/4/0

Tabulka 17: Výsledky testování druhého detektoru – FindCourierUseOutOfInvoker.

```
package testCases;
import org.jboss.soa.esb.couriers.CourierFactory;

public class TestRH2example1 {

	public TestRH2example1(){
	}

	public void firstMethodDANGER(){
		CourierFactory.getInstance();
	}

	public void secondMethodDANGER(){
		TestRH2example2.firstMethodFromDeliverOnlyCouriers();
	}

	public void thirdMethodDANGER(){
		TestRH2example3 trh2e3 = new TestRH2example3();
		trh2e3.firstMethodFromPickupOnlyCouriers();
	}

	public void fourthMethodDANGER(){
		firstMethodDANGER();
	}
}
```

Ilustrace 29: Testovací třída – testCases.TestRH2example1.

```
package testCases;
import org.jboss.internal.soa.esb.couriers.DeliverOnlyCourier;

public class TestRH2example2 implements DeliverOnlyCourier {

	public TestRH2example2(){
	}

	public static void firstMethodFromDeliverOnlyCouriers(){
	}
}
```

Ilustrace 30: Testovací třída – testCases.TestRH2example2.

```
package testCases;
import org.jboss.internal.soa.esb.couriers.PickupOnlyCourier;

public class TestRH2example3 implements PickupOnlyCourier {

	public TestRH2example3(){
	}

	public void firstMethodFromPickupOnlyCouriers(){
	}
}
```

Ilustrace 31: Testovací třída – testCases.TestRH2example3.

Výsledek testu a testovací třída 3. detektoru

Tabulka 18 zobrazuje výsledky testů provedené s 3. detektorem (FindPoorTestOfObject) nad testovací třídou (TestRH3example1) zobrazenou na Ilustracích 32, 33 a 34.

Co se testovalo	**Výstup detektoru**	**nb/rb/fa**
TestRH3example1	H C PV3: Miss method getPayLoad(Method) in testCases.TestRH3example1.sevenMethodDANGER(Message) At TestRH3example1.java:[lines 57-58] H C PV3: Miss method getPayLoad(Method) in testCases.TestRH3example1.fourthMethodDANGER(Message) At TestRH3example1.java:[lines 37-38] H C PV3: Miss method getPayLoad(Method) in testCases.TestRH3example1.sixMethodDANGER(Message) At TestRH3example1.java:[lines 49-52] H C PV3: Miss method getPayLoad(Method) in testCases.TestRH3example1.thirdMethodDANGER(Message) At TestRH3example1.java:[lines 31-32]	0/4/0

Tabulka 18: Výsledky testování třetího detektoru – TestMethodGetPayLoad.

```
package testCases;
import org.jboss.soa.esb.listeners.message.MessageDeliverException;
import org.jboss.soa.esb.message.Message;
import org.jboss.soa.esb.actions.ActionLifecycleException;
import org.jboss.soa.esb.actions.ActionPipelineProcessor;
import org.jboss.soa.esb.actions.ActionProcessingException;
import org.jboss.soa.esb.message.MessagePayloadProxy;

public class TestRH3example1 implements ActionPipelineProcessor{

	public void testRH3example() {
	}

	public Message firstMethodSAFE(Message mes) throws
	MessageDeliverException{
		MessagePayloadProxy mpp = new MessagePayloadProxy(null);
		mpp.getPayload(mes);
		mes.getProperties();
		return null;
	}

	public Message secondMethodSAFE(Message mes) throws
	MessageDeliverException{
		MessagePayloadProxy mpp = new MessagePayloadProxy(null);
		return null;
	}
```

Ilustrace 32: Testovací třída – testCases.TestRH3example1 – první část.

```
public Message thirdMethodDANGER(Message mes){
      mes.getProperties();
      return null;
}

public Message fourthMethodDANGER(Message arg0) throws
ActionProcessingException {
      fifthMethodSAFE(arg0);
      return null;
}

public void fifthMethodSAFE(Message arg0) throws
ActionProcessingException {
      arg0.getProperties();
      return;
}

public Message sixMethodDANGER(Message arg0) throws
ActionProcessingException, MessageDeliverException {
      fifthMethodSAFE(arg0);
      MessagePayloadProxy mpp = new MessagePayloadProxy(null);
      mpp.getPayload(arg0);
      return null;
}

public Message sevenMethodDANGER(Message arg0) throws
ActionProcessingException, MessageDeliverException {
      eightMethodDANGER(arg0);
      return null;
}

public void eightMethodDANGER(Message arg0) throws
ActionProcessingException, MessageDeliverException {
      fifthMethodSAFE(arg0);
}

public Message nineMethodSAFE(Message arg0) throws
ActionProcessingException, MessageDeliverException {
      tenMethodSAFE(arg0);
      return null;
}

public void tenMethodSAFE(Message arg0) throws
ActionProcessingException, MessageDeliverException {
      MessagePayloadProxy mpp = new MessagePayloadProxy(null);
      mpp.getPayload(arg0);
      fifthMethodSAFE(arg0);
}
```

Ilustrace 33: Testovací třída – testCases.TestRH3example1 – druhá část.

```
	public Message process(Message arg0) throws
	ActionProcessingException {
		return null;
	}

	public void processException(Message arg0, Throwable arg1) {
	}

	public void processSuccess(Message arg0) {
	}

	public void destroy() throws ActionLifecycleException {
	}

	public void initialise() throws ActionLifecycleException {
	}
}
```

Ilustrace 34: Testovací třída – testCases.TestRH3example1 – třetí část.

Výsledek testu a testovací třída 4. detektoru

Tabulka 19 zobrazuje výsledky testu provedené s 4. detektorem (FindPoorTestOfObject) nad testovací třídou (TestRH4example3) zobrazenou na Ilustracích 35, 36, 37 a 38.

Co se testovalo	Výstup detektoru	nb/rb/fa
TestRH4example3	H C PV4: Found poor objects test in testCases.TestRH4example2.equalsDANGER(Object, Object) At TestRH4example2.java:[lines 47-78] H C PV4: Found poor objects test in testCases.TestRH4example3.firstDANGERmethodDANGERequals(Object, Object) At TestRH4example3.java:[lines 108-111] H C PV4: Found poor objects test in testCases.TestRH4example3.secondDANGERmethodDANGERequals(Object, Object) At TestRH4example3.java:[line 115] H C PV4: Found poor objects test in testCases.TestRH4example3.equalsDANGER(Object, Object) At TestRH4example3.java:[lines 127-133]	0/4/0

Tabulka 19: Výsledky testování čtvrtého detektoru – FindPoorTestOfObject.

```
package testCases;
import org.drools.base.ShadowProxy;

public class TestRH4example3 {

	public TestRH4example3(){
	}

	public boolean firstMethodSAFEequals(Object obj0, Object obj1){
		if(obj1 == null){
			return obj0 == null;
		}

		if(obj0 == null){
			return false;
		}

		if(obj1 instanceof ShadowProxy){
			return obj1.equals(obj0);
		}
		return false;
	}
```

Ilustrace 35: Testovací třída – testCases.TestRH4example3 – první část.

```
public boolean secondMethodSAFEequals(Object obj0, Object obj1){
	if(obj1 != null){
		if(obj0 == null){
			return false;
		}
		if(obj0 instanceof ShadowProxy){
			return obj0.equals(obj1);
		}
	}

	if (obj0 == null){
		return true;
	}
	return false;
}

public boolean thirdMethodSAFEequals(Object obj0, Object obj1){
	if(obj1 != null){
		if(obj0 != null){
			if(obj0 instanceof ShadowProxy){
				return obj0.equals(obj1);
			}
		}
		return false;
	}
	if (obj0 == null){
		return true;
	}
	return false;
}
```

Ilustrace 36: Testovací třída – testCases.TestRH4example3 – druhá část.

```
public boolean fourthMethodSAFEequals(Object arg0, Object arg1){
      if ( arg0 == null || arg1 == null ) {
            return arg0 == arg1;
      }
      if( arg1 instanceof ShadowProxy ) {
            return arg1.equals( arg0 );
      }
      return false;
}

public boolean fifthMethodSAFEequals(Object arg0, Object arg1){
      if ( arg0 == null && arg1 != null ) {
            return false;
      }

      if ( arg0 != null && arg1 == null ) {
            return false;
      }

      if (arg0 == null && arg1 == null){
            return true;
      }

      if( arg1 instanceof ShadowProxy ) {
            return arg1.equals( arg0 );
      }
      return false;
}

public boolean sixthMethodSAFEequals(Object arg0, Object arg1){
      if ( (arg0 == null && arg1 != null)
      || (arg0 != null && arg1 == null)) {
            return false;
      }

      if (arg0 == null && arg1 == null){
            return true;
      }

      if( arg1 instanceof ShadowProxy ) {
            return arg1.equals( arg0 );
      }
      return false;
}

public boolean firstDANGERmethodDANGERequals(Object obj1, Object obj2){
      if(obj1 instanceof ShadowProxy){
            return obj1.equals(obj2);
      }
      return false;
}
```

Ilustrace 37: Testovací třída – testCases.TestRH4example3 – třetí část.

```
	public boolean secondDANGERmethodDANGERequals(Object obj1, Object obj2){
		return obj1.equals(obj2);
	}

	public void firstHelpMethod(){
		Object obj1 = new Object();
		Object obj2 = new Object();
		firstMethodSAFEequals(obj1,obj2);
	}

	public boolean equalsDANGER( Object arg0, Object arg1 ) {
		if ( arg0 == null ) {
			return arg1 == null;
		}
		if( arg1 != null && arg1 instanceof ShadowProxy ) {
			return arg1.equals( arg0 );
		}
		return helpMet();
	}

	public boolean equalsSAFE( Object arg0, Object arg1 ) {
		if ( arg0 == null || arg1 == null ) {
			return arg0 == arg1;
		}
		if( arg1 instanceof ShadowProxy ) {
			return arg1.equals( arg0 );
		}
		return helpMet();
	}

	public boolean helpMet(){
		return true;
	}
}
```

Ilustrace 38: Testovací třída – testCases.TestRH4example3 – čtvrtá část.

Výsledek testu a testovací třídy 5. detektoru

Tabulka 20 zobrazuje výsledky testu provedené s 5. detektorem (TestSuddenAttributesChange) nad testovacími třídami (TestRH5example1, TestRH5example2, TestRH5example3, TestRH5example4 a TestRH5example5) zobrazenými na Ilustracích 39, 40, 41, 42 a 43.

Co se testovalo	**Výstup detektoru**	**nb/rb/fa**
TestRH5example1, TestRH5example2, TestRH5example3, TestRH5example4, TestRH5example5	H C PV5: Found sudden attributes change in testCases.TestRH5example5.firstMethodDANGER() At TestRH5example5.java:[lines 12-13] H C PV5: Found sudden attributes change in testCases.TestRH5example3.firstMethodDANGER() At TestRH5example3.java:[lines 12-13] H C PV5: Found sudden attributes change in testCases.TestRH5example3.fifthMethodDANGER() At TestRH5example3.java:[lines 31-32] H C PV5: Found sudden attributes change in testCases.TestRH5example1.metodaDANGER() At TestRH5example1.java:[lines 22-23] H C PV5: Found sudden attributes change in testCases.TestRH5example1.metoda2DANGER() At TestRH5example1.java:[lines 26-27]	0/5/0

Tabulka 20: Výsledky testování pátého detektoru – TestSuddenAttributesChange.

```
package testCases;
public class TestRH5example5 extends TestRH5example1{

	public static String atribut;

	public TestRH5example5(){
		atribut = "ahoj";
	}

	public static void firstMethodDANGER(){
		atribut = "cau";
	}
}
```

Ilustrace 39: Testovací třída – testCases.TestRH5example5.

```
package testCases;
import java.util.concurrent.ArrayBlockingQueue;
import org.jboss.soa.esb.actions.ActionLifecycleException;
import org.jboss.soa.esb.actions.ActionPipelineProcessor;
import org.jboss.soa.esb.actions.ActionProcessingException;
import org.jboss.soa.esb.message.Message;

public class TestRH5example1 implements ActionPipelineProcessor{

	public String firstAttr;
	public String secondAttr;
	public ArrayBlockingQueue<String> abq;

	public TestRH5example1(){
		firstAttr = "jiny retezec";
		abq = new ArrayBlockingQueue<String>(10);
	}

	public void metodaDANGER(){
		secondAttr = "ret";
	}

	public void metoda2DANGER(){
		metodaDANGER();
	}

	public Message process(Message arg0) throws
ActionProcessingException {
		return null;
	}

	public void processException(Message arg0, Throwable arg1) {
	}

	public void processSuccess(Message arg0) {
	}

	public void destroy() throws ActionLifecycleException {
		firstAttr = "";
	}

	public void initialise() throws ActionLifecycleException {
		firstAttr = "retezec";
		metodaDANGER();
	}
}
```

Ilustrace 40: Testovací třída – testCases.TestRH5example1.

```
package testCases;
import java.util.concurrent.ArrayBlockingQueue;
import org.jboss.soa.esb.actions.ActionLifecycleException;
import org.jboss.soa.esb.actions.ActionPipelineProcessor;
import org.jboss.soa.esb.actions.ActionProcessingException;
import org.jboss.soa.esb.message.Message;

public class TestRH5example2 implements ActionPipelineProcessor{

	public static String firstAttr;
	public static String secondAttr;
	public static ArrayBlockingQueue<String> abq;

	public TestRH5example2(){
		firstAttr = "jiny retezec";
		abq = new ArrayBlockingQueue<String>(10);
	}

	public Message process(Message arg0) throws
ActionProcessingException {
		return null;
	}

	public void processException(Message arg0, Throwable arg1) {
	}

	public void processSuccess(Message arg0) {
	}

	public void destroy() throws ActionLifecycleException {
		firstAttr = "";

	}

	public void initialise() throws ActionLifecycleException {
		firstAttr = "retezec";
	}
}
```

Ilustrace 41: Testovací třída – testCases.TestRH5example2.

```
package testCases;

public class TestRH5example3 {
	TestRH5example1 obj;

	public TestRH5example3(){
		obj = new TestRH5example1();
	}

	public void firstMethodDANGER(){
		obj.firstAttr = "retez";
	}

	public void secondMethodSAFE(){
		Object obj2 = new Object();
		synchronized(obj2){
			obj.firstAttr = "retez2";
		}
	}

	public synchronized void thirdMethodSAFE(){
		obj.firstAttr = "retez3";
	}

	public void fourthMethodSAFE(){
		obj.abq.add("retez");
	}

	public void fifthMethodDANGER() {
		TestRH5example2.firstAttr = "retez";
	}

	public void sixthMethodSAFE(){
		Object obj2 = new Object();
		synchronized(obj2){
			TestRH5example2.firstAttr = "retez2";
		}
		TestRH5example2.abq.add("retezec");
	}

	public void sevenMethodDANGER(){
		TestRH5example4.firstMethodDanger();
	}
}
```

Ilustrace 42: Testovací třída – testCases.TestRH5example3.

```
package testCases;
import org.jboss.internal.soa.esb.couriers.PickupOnlyCourier;

public class TestRH5example4 implements PickupOnlyCourier{

	public static String atribut;

	public TestRH5example4(){
		atribut = "ahoj";
	}

	public static void firstMethodDANGER(){
		atribut = "cau";
	}
}
```

Ilustrace 43: Testovací třída – testCases.TestRH5example4.

Dodatek B

Použitelné parametry jednotlivých detektorů

Tabulka 21 zobrazuje souhrn parametrů použitelných při explicitním spouštění detektorů. Popis jednotlivých parametrů se nachází v implementačních částech detektorů v kapitole 4.

Detektor	**Parametry**
FindStaticMethodClassForName	RH_1_DEFINE_USE
	RH_1_CLASSPATH_PREFIX
	RH_1_CLASSPATH_CONTINUE
	RH_1_PROHIBITED_METHODS
FindCourierUseOutOfInvoker	RH_2_DEFINE_USE
	RH_2_SAFE_CLASSES
	RH_2_CATCH_CLASSES
	RH_2_CATCH_CLASSES_WITH_INTERFACES
TestMethodGetPayLoad	RH_3_DEFINE_USE
	RH_3_ANALYSE_CLASSES_WITH_INTERFACES
	RH_3_METHOD_WHICH_MAKE_SAFE_METHODS
	RH_3_ANALYSE_METHOD_WITH_SIGNATURE
FindPoorTestOfObject	RH_4_DEFINE_USE
	RH_4_METHODS_NAME_CONTAINS
	RH_4_INSTANCEOF_SECOND_ARGUMENT
TestSuddenAttributesChange	RH_5_DEFINE_USE
	RH_5_ANALYSE_CLASSES_WITH_INTERFACES

Tabulka 21: Parametry detektorů.