MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Studie nástrojů pro trasování a testování programů v Javě

BAKALÁRSKA PRÁCA

Matej Majdiš

Brno, 2015

Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Vedúci práce: RNDr. Adam Rambousek

Zhrnutie

Cieľom tejto práce je analýza vlastností, spolu s implementáciou súboru príkladov pre nástroje, *Byteman* a *Javassist*. Teoretická časť charakterizuje bajtkód, *Java Virtual Machine (JVM)*, načítavanie tried pomocou zavádzačov a dôležité vlastnosti spomínaných nástrojov. Praktická časť zahŕňa ukážky príkladov, ktoré demonštrujú funkcionalitu nástrojov v rôznych oblastiach vývoja a testovania softvéru.

Kľúčové slová

Byteman, Javassist, bajtkód, JVM, Java Virtual Machine, modifikácia, class-loader, zavádzač tried

Poďakovanie

TODO...

Obsah

Kapitola 1

Úvod

Java je v súčasnosti jedným z najpoužívanejších programovacích jazykov. Od syntakticky veľmi podobných jazykov, ako napríklad C++, alebo C# sa líši prekladom zdrojových tried do medzikódu, často označovaného ako bajtkód (*bytecode, p-code, portable code*).

Preklad a spustenie aplikácie v Jave štandardne prebieha v niekoľkých nasledujúcich fázach:

1. Preklad do medzikódu: Prekladač [1] kompiluje zdrojový kód do bajtkódu. V praxi to znamená, že každej triede, prípadne rozhraniu, je priradený súbor *class*, ktorý obsahuje najmä inštrukcie popisujúce jej vlastnosti a funkcionalitu.

2. Načítanie a Interpretácia: Virtuálny stroj Javy (ďalej len JVM [2]) načíta inštrukcie *class* súboru potrebnej triedy, ktoré ďalej spracúva jedným z nasledujúcich spôsobov:

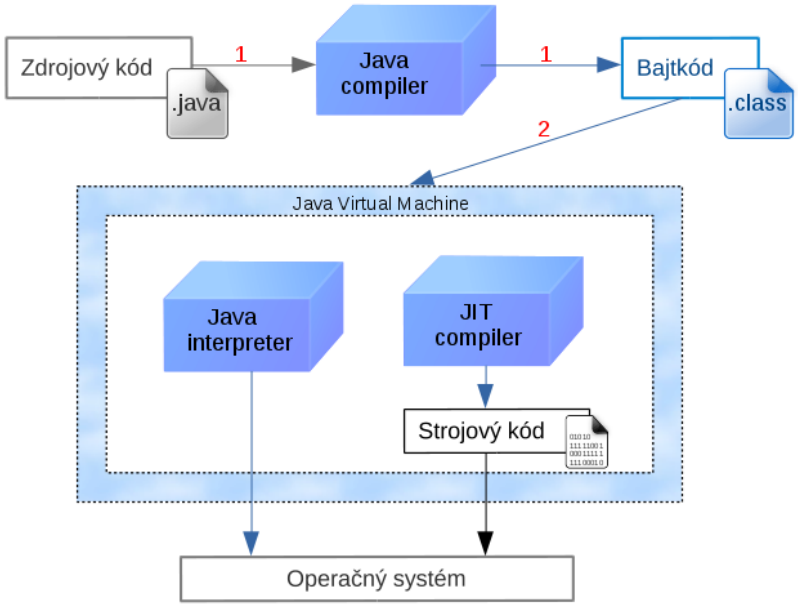
   - JIT prekladač (*Just In Time compiler*): Štandardne je z bajtkódu najskôr vygenerovaný strojový kód (*machine code*) konkrétneho zariadenia (platformy), ktorý je následne interpretovaný, teda priamo vykonávaný procesorom.
   - Java interpret: Ďalším spôsobom spracovania bajtkódu je využitie interpreta jazyka Java, ktorý bajtkód ihneď spracováva a sám interpretuje.

Výhodou prekladu do bajtkódu je prenosnosť výsledných aplikácií. Samotný bajtkód je platformovo nezávislý. Program preto nie je nutné nijako prispôsobovať jednotlivým operačným systémom, ktoré sa líšia výhradne v implementácií JVM.

---

1. Najčastejšie využívaným prekladačom je *javac*, ktorý je súčasťou JDK (Java Development Kit - súbor nástrojov určený na vývoj aplikácií pre platformu Java).
2. Java Virtual Machine, špecifikácia je dostupná na `http://docs.oracle.com/javase/specs/jvms/se7/html`

Class súbory obsahujúce bajtkód je možné modifikovať. Triedy a rozhrania danej aplikácie, uložené v týchto súboroch, podľa potreby načítava JVM. Vkladanie nových metód a tried na úrovni bajtkódu, pred načítaním *class* súboru do JVM, sa nazýva injekcia bajtkódu (*bytecode injection*, ďalej len BI). Pridávanie novej funkcionality pomocou BI, bez nutnosti zastavenia behu programu, je často využívané pri testovaní a trasovaní programov.



Obr. 1.1: Grafické znázornenie prekladu a spustenia programu, zdroj: vlastné spracovanie

Táto práca sa zaoberá popisom vlastností a implementáciou niekoľkých príkladov používajúcich nástroje na modifikáciu *class* súborov. Konkrétne nástroje spomínané a používané v tejto práci sú *Byteman* a *Javassist*.

Dokument sa skladá z dvoch logických celkov. Kapitoly dva až päť tvoria teoretickú časť, ktorá popisuje bajtkód, *Java Virtual Machine*, zavádzače tried, vlastnosti samotných nástrojov *Byteman a Javassist*. Šiesta kapitola spomínané nástroje porovnáva.

Kapitola sedem sa venuje praktickým ukážkam, ktoré boli vytvorené vrámci bakalárskej práce. Podkapitoly sa zaoberajú príkladmi, ktoré pokrývajú oblasti detekcie volania výnimiek, detekcie nesprávne ošetrených výnimiek, zlepšenia produkčného kódu a optimalizácie neefektívnych častí kódu. Každá z ukážok demonštruje funkcionalitu jedného zo spomínaných nástrojov.

Kapitola 2

Bajtkód

Po preklade zdrojových kódov prekladačom *javac* je každej triede, prípadne rozhraniu programu, priradený jeden *class* súbor, popisujúci jej vlastnosti a funkcionalitu.

Pri načítaní *class* súboru prijme JVM takzvaný prúd inštrukcií bajtkódu (*bytecode stream*) pre každú metódu triedy. V prípade volania konkrétnej metódy za behu programu, sú inštrukcie danej metódy vykonávané. Každá z nich je reprezentovaná číselnou hodnotou, nazývanou *opcode*. Zároveň má každá inštrukcia aj textovú podobu (*mnemonic*). V *class* súboroch sú uložené v numerickej podobe.

Táto kapitola popisuje základný formát *class* súboru a následne stručne charakterizuje inštrukčnú sadu bajtkódu. [1]

2.1 Štruktúra *class* súboru

Class súbor vždy pozostáva z jedinej štruktúry *ClassFile*. Táto štruktúra jednoznačne identifikuje konkrétnu triedu, prípadne rozhranie, definuje jej premenné a metódy.

Nasledujúci popis určuje sadu dátových typov použitých v texte. Typy *u1*, *u2*, a *u4* reprezentujú neznamienkové jedno, dvoj, alebo štvor-bajtové číslo. Pre znázornenie *class* súboru je použitá pseudo-štruktúra v notácií jazyka C. Obsah štruktúry je popísaný ako zoznam po sebe nasledujúcich položiek.

Formát *ClassFile* štruktúry

```
ClassFile {
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool[constant_pool_count-1];
```

1. Nasledujúci text vychádza zo 4. až 6. kapitoly špecifikácie JVM [LYBB13].

```
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Konštanta *magic* identifikuje formát súboru *class*, jej hodnota je vždy rovná *0xCAFEBABE*.

Položky *minor_version* a *major_version* určujú verziu *class* súboru. Napríklad *minor_version* s hodnotou *m* a *major_version* s hodnotou *M* indikujú verziu s hodnotou *M.m*.

Hodnota položky *constant_pool_count* je rovná množstvu záznamov v *constant_pool[]*, plus jeden.

Úložisko záznamov *constant_pool[]* (v podobe poľa štruktúr) obsahuje rôzne dôležité konštanty: mená tried, mená rozhraní, názvy premenných a iné. Každý záznam spomínaného poľa pozostáva zo značky (*tag*) a indexu (*name_index*). Značka určuje typ záznamu. Tabuľka všetkých značiek je uvedená v prílohe A.1. Pomocou unikátneho indexu je následne možné odkazovať sa na záznamy v ďalších častiach bajtkódu. Existuje niekoľko typov štruktúr [2] reprezentujúcich rôzne druhy ich záznamov. Napríklad štruktúra *CONSTANT_String_info* reprezentuje objekty typu *String*, zatiaľ čo ďalšie podobné štruktúry ako napríklad *CONSTANT_Methodref_info* a *CONSTANT_InterfaceMethodref_info* reprezentujú metódy triedy, prípadne rozhrania.

Hodnota *access_flags* ukladá oprávnenia prístupu k informáciám a vlastnosti tejto triedy pomocou indikátorov. Napríklad nastavenie indikátora *ACC_INTERFACE* znamená, že *class* súbor popisuje rozhranie. Tabuľka indikátorov je uvedená v prílohe A.2.

Položka *this_class* obsahuje index položky poľa *constant_pool[]* odkazujúci na štruktúru *CONSTANT_Class_info* [3]. Reprezentuje triedu, respektíve rozhranie, definované týmto *class* súborom.

Hodnotou *super_class* je taktiež index poľa *constant_pool[]* odkazujúci na štruktúru typu *CONSTANT_Class_info*. Reprezentuje priamu nadtriedu

2. Všetky štruktúry *constant_pool[]* sú popísane v špecifikácií JVM [LYBB13].
3. *CONSTANT_Class_info* je štruktúra *constant_pool*, ktorá udáva triedu, alebo rozhranie.

triedy definovanej aktuálnym *class* súborom. V prípade, že ide o súbor popisujúci rozhranie, index odkazuje na triedu *Object*. Trieda *Object* má ako jediná hodnotu *super_class* nulovú.

Počet prípadných rozhraní, ktoré daná trieda implementuje, vyjadruje položka *interface_count*. V prípade rozhrania je táto položka rovná počtu jeho priamych nadrozhraní.

Pole *interfaces[]* obsahuje indexy *constant_pool[]*, odkazujúce na štruktúru typu *CONSTANT_Class_info*. Zahŕňa indexy všetkých rozhraní, ktoré sú implementované aktuálnou triedou, prípadne sú priamymi nadrozhraniami *class* súboru.

Položka *fields_count* je rovná počtu premenných triedy a premenných inštancií (*fields*) *class* súboru.

Štruktúry typu *field_info* sú združené v poli *fields[]*. Toto pole zahŕňa každú premennú triedy. Neobsahuje však zdedené atribúty. Podrobne sa štruktúrou *field_info* zaoberá kapitola 2.1.2.

Hodnota položky *methods_count* vyjadruje počet štruktúr *method_info* v poli *methods[]*.

Položka *methods[]* je poľom štruktúr typu *method_info*. Každá štruktúra *method_info* popisuje metódu aktuálnej triedy, respektíve rozhrania. Zahŕňa konštruktory, metódy triedy a metódy inštancií. Neobsahuje žiadne zdedené metódy. Štruktúru *method_info* popisuje podkapitola 2.1.3.

Hodnota *attributes_count* je rovná počtu atribútov poľa *attributes[] class* súboru.

Pole *attributes[]* obsahuje štruktúry typu *attribute_info*. Atribútmi štruktúry *ClassFile* sú napríklad: *SourceFile*, *Deprecated*, *InnerClasses* a iné. Atribút *SourceFile* slúži na reprezentáciu mena *class* súboru. Spomínané pole môže obsahovať maximálne jeden takýto atribút. Atribút *Deprecated* môže byť použitý v prípade, že bola daná trieda nahradená (*deprecated*). Pri jej volaní môže prekladač upozorniť užívateľa, že sa odkazuje na nahradenú triedu [4]. Vo všeobecnosti sa štruktúre *attribute_info* venuje podkapitola 2.1.4.

2.1.1 Reprezentácia dátových typov

Dátové typy sú v *class* súboroch reprezentované vo formáte reťazcov s kódovaním *UTF-8*. Delíme ich na:

- dátové typy premenných,
  - primitívne dátové typy

4. Rovnakým spôsobom je možné atribút *Deprecated* aplikovať aj na premenné a metódy.

  - referenčné dátové typy
  - polia

- dátové typy metód.

Primitívnym dátovým typom (*byte*, *int*, *boolean*, ...) je priradený popis v podobe znaku (*B*, *I*, ...). Napríklad premenná typu *int* je reprezentovaná znakom: *I*.

Referenčné dátové typy reprezentuje popis v tvare: *L<classname>;*, kde *classname* je meno triedy, alebo rozhrania daného referenčného dátového typu. Premenná typu *Object* je interpretovaná ako: *java/lang/Object;*.

Identifikačný reťazec jednorozmerného poľa typu *T* sa značí *[T*, pričom počet znakov *[* je rovný dimenzii poľa. Napríklad trojrozmerné pole typu double : *double d[][][]* generuje reťazec: *[[[D*.

Reťazec dátového typu metódy sa skladá z reťazcov pre dátový typ parametrov, ohraničených v zátvorkách *(P*)* a reťazca pre dátový typ návratovej hodnoty *R*. Univerzálny tvar reťazca je potom: *(P*)R*. V prípade hodnoty *null* je reťazcom návratovej hodnoty znak *V*. Napríklad metódu *boolean long pow (int n, int k)* reprezentuje reťazec: *(II)J*, v prípade metódy *Object method(byte b)* by šlo o reťazec: *(B)Ljava/lang/Object;*. Komplexný prehľad reprezentácie dátových typov je uvedený v prílohe A.3.

2.1.2 Premenné tried a inštancií

Premenné tried inštancií (*fields*) *class* súboru sú v poli *fields[]* reprezentované pomocou štruktúry *field_info*. Formát štruktúry *field_info* je nasledovný:

Štruktúra *field_info*

```
field_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Položka *access_flags* je indikátorom oprávnenia prístupu k danej premennej. Mená všetkých indikátorov spolu s ich interpretáciou a hodnotou sú uvedené v prílohe A.4.

Dvojbajtová hodnota *name_index* je index *constant_pool[]*, reprezentujúci meno premennej.

Podobne ako *name_index*, aj *descriptor_index* je dvojbajtová položka odkazujúca sa na štruktúru v *constant_pool*. Na rozdiel od mena premennej však popisuje dátový typ premennej.

Položka *attributes_count* vyjadruje počet všetkých atribútov uložených v poli *attributes[]*.

Pole *attributes[]* môže obsahovať ľubovoľné množstvo atribútov popisujúcich premennú. Štruktúra reprezentujúca atribút je daná všeobecným predpisom *attribute_info*. Atribúty premenných musia byť reprezentované jednou zo štruktúr *ConstantValue*, *Synthetic*, *Signature*, *Deprecated*, *RuntimeVisibleAnnotations*, prípadne *RuntimeInvisibleAnnotations*. Atribút s menom *ConstantValue* popisuje konštantné statické premenné. *Synthetic* je používaný u položiek, ktoré sa nevyskytujú v zdrojovom kóde. Podrobne sa štruktúrou *attribute_info* zaoberá podkapitola 2.1.4.

2.1.3 Metódy

Každá metóda triedy, prípadne rozhrania, je v poli *methods[]* uložená pomocou štruktúry *method_info*. Táto štruktúra má nasledujúci formát:

Formát *method_info*

```
method_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Indikátor *access_flags* zahŕňa nastavenia prístupových práv a vlastností metódy. Tabuľka indikátorov *access_flags* štruktúry *method_info* sa nachádza v prílohe A.5.

Položky *name_index* a *descriptor_index* sú podobne ako u *field_info* indexmi do *constant_pool*. Tieto indexy v *constant_pool* odkazujú na štruktúry charakterizujúce meno a dátový typ metódy. Reprezentáciu dátových typov popisuje podkapitola 2.1.1.

Hodnota položky *attributes_count* vyjadruje presný počet atribútov v poli *attributes[]*.

Pole *attributes[]* obsahuje dodatočné atribúty danej metódy. Jednotlivé položky poľa sú reprezentované elementárnym predpisom *attributes_info*. Každá z položiek je jednou zo štruktúr: *Code*, *Exceptions*, *Synthetic*,*Signature*, *Deprecated*, *RuntimeVisibleAnnotations*, *RuntimeInvisibleAnnotations*, *RuntimeVisibleParameterAnnotations*, *RuntimeInvisibleParameterAnnotations*, prípadne

AnnotationDefault. Atribút *Code* je jedným z najdôležitejších. Obsahuje inštrukcie bajtkódu popisujúce fungovanie metódy. Okrem abstraktných a natívnych metód, musí každá z nich obsahovať práve jeden atribút typu *Code*. Atribút *Exceptions* zahŕňa indexy výnimiek, ktoré metóda volá. Bližším popisom základného formátu štruktúry *attributes_info* sa zaoberá nasledujúca podkapitola.

2.1.4 Atribúty

Pojem atribút tejto podkapitoly vyjadruje práve atribúty používané v poli *attributes[]* štruktúr *field_info*, *method_info* a *Code_attributes*. Všeobecný predpis všetkých atribútov je vyjadrený štruktúrou *attribute_info*. Existuje niekoľko základných preddefinovaných atribútov: *SourceFile*, *ConstantValue*, *Code*, *Exceptions*, *InnerClasses*, *Synthetic*, *LineNumberTable*, *LocalVariableTable*, *Deprecated* a iné. Líšia sa funkcionalitou a využitím jednotlivými časťami *class* súboru. Všetky atribúty vychádzajú z už spomínaného všeobecného predpisu *attribute_info*, ktorý má nasledujúci formát:

Štruktúra *attribute_info*

```
attribute_info {
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_lenght];
}
```

Položka *attributes_name_index* je indexom do *constant_pool*, odkazujúcim na meno atribútu. Štvorbajtová položka *attribute_length* je rovná hodnote vyjadrujúcej dĺžku následných informácií, ktoré sú uložené v štruktúre *info[attribute_length]*. Informácie sa líšia na základe odlišnej funkcionality a využitia jednotlivých atribútov.

2.2 Inštrukcie JVM

Po načítaní *class* súboru sa JVM najskôr uistí, že je daný súbor v správnom formáte popísanom v kapitole 2.1. Tento proces sa nazýva kontrola formátu (*format checking*). Prvé štyri bajty musia vždy obsahovať magickú konštantu *magic*. Všetky rozoznané atribúty musia mať správnu dĺžku. *Class* súbor nesmie byť skrátený, ani obsahovať nadbytočné bajty. Podobne úložisko *constant_pool* nesmie obsahovať žiadne nerozoznateľné informácie.

Inštrukcie bajtkódu načítanej metódy sú uložené v poli *code[]* atribútu *Code*, štruktúry *method_info* daného *class* súboru. Štruktúra *Code_attribute* re-

prezentujúca atribút *Code* musí spĺňať obmedzenia definované JVM. Tieto obmedzenia rozdeľujeme na dve základné kategórie:

- Statické obmedzenia: Stanovujú rozloženie inštrukcií v poli *code* a priradenie operandov jednotlivým inštrukciám. Niektorými z nich sú napríklad:
  - prvá inštrukcia musí začínať na indexe 0,
  - pole *code* nesmie byť prázdne.
- Štrukturované obmedzenia: Špecifikujú vzťahy medzi inštrukciami JVM. Ide o podmienky, ako napríklad:
  - žiadna lokálna premenná nemôže byť volaná predtým, ako jej bola priradená hodnota,
  - pred volaním (nestatickej) metódy, respektíve premennej, musí byť inicializovaná inštancia triedy, ktorá ju obsahuje.

Prekladače jazyka Java generujú *class* súbory, ktoré spĺňajú požiadavky popísané v predchádzajúcom odseku. JVM však nemá žiadnu záruku, že *class* súbor, ktorý požaduje, bol generovaný prekladačom. Metódou verifikácie [5] *class* súboru môže JVM určiť, či daný súbor pochádza z dôveryhodného zdroja.

2.2.1 Dátové typy

Dátové typy JVM delíme do troch základných kategórií:

- Primitívne dátové typy: byte, short, int, long, boolean, float, double.
- Referenčné dátové typy: pole, inštancia triedy, rozhranie.
- Typ *returnAddress*: používaný výhradne inštrukciami *jsr*, *ret* a *jsr_w*.

Väčšina uvedených typov má veľkosť 32 bitov, typy *long* a *double* sú však 64 bitové, preto zaberajú dva sloty v zásobníku.

---

5. Ďalšie príklady obmedzení a podrobný popis verifikácie *class* súborov je dostupný v špecifikácií JVM [LYBB13]

2.2.2 Architektúra a inštrukčná sada

Architektúra JVM je založená na dátovej štruktúre zásobník [6], ktorej základnými operáciami sú *push* - vloženie prvku do zásobníka a *pop* - výber prvku z vrcholu zásobníka. JVM nemá registre na ukladanie hodnôt, preto musia byť pred použitím všetky uložené na zásobník.

Na nasledujúcom jednoduchom príklade sú popísané základné inštrukcie bajtkódu pre prácu s premennými a konštantami.

Metóda *greaterThen* pred a po kompilácií

```
public int greaterThen(int intOne, int intTwo) {
  if (intOne > intTwo) {
    return 0;
  } else {
    return 1;
  }
}
```

```
0: iload_1
1: iload_2
2: if_icmple  7
5: iconst_0
6: ireturn
7: iconst_1
8: ireturn
```

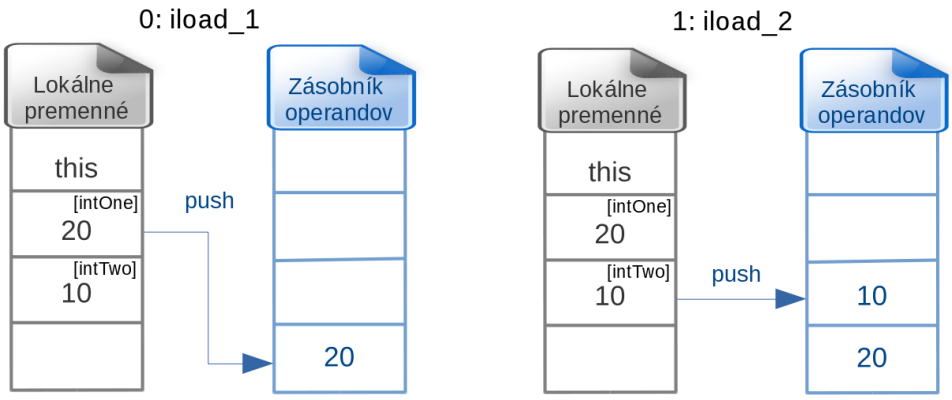
Inštrukcie *iload_1* a *iload_2* pridajú do zásobníka operandov (ďalej len zásobník) hodnoty lokálnych premenných na indexoch 1 a 2. V tomto prípade ide o parametre *intOne* a *intTwo*.

Vo všeobecnosti môžeme túto inštrukciu chápať ako *xload* s predponou *x*, označujúcou primitívny dátový typ (napríklad: *lload* pre long, *fload* pre float). Existujú dva tvary, volania tejto inštrukcie:

- *load_<n>*, kde *n* označuje index (celé číslo) lokálnej premennej, zároveň musí platiť: $0 \leq n \leq 4$,
- *load vindex*, kde pozíciou lokálnej premennej je hodnota *vindex*.

Ďalšou inštrukciou je *if_icmple* s parametrom 7, ktorá porovná dva objekty na vrchole zásobníka a prejde na siedmu inštrukciu v prípade, že je hodnota položky na vrchole zásobníka väčšia ako hodnota druhej položky. V príklade sú na zásobníku len položky, vložené predchádzajúcimi inštrukciami. Podmienka teda platí v prípade, že hodnota parametra *intOne* je

6. Dátová štruktúra zásobník (*stack*) funguje na princípe FIFO (*first in first out*), kde posledný vložený prvok je prvým vybraným.



Obr. 2.1: Znázornenie funkcionality inštrukcií *iload_1* a *iload_2*, zdroj: vlastné spracovanie

menšia než hodnota *intTwo*. Vo všeobecnosti je možné podmienené výrazy vyjadriť pomocou inštrukcií: *if_acmp<cond>*, *if_icmp<cond>*, *if<cond>*, *ifnonnull*, *ifnull*.

Inštrukcie *iconst_0* a *iconst_1* vložia na zásobník hodnotu 0, respektíve 1, v závislosti od vyhodnotenia podmienky *if_icmple*. Táto hodnota je následne vrátená inštrukciou *ireturn*. Inštrukcie *iconst_<n>*, *a ireturn* sú taktiež dostupné vo variantách s predponou ľubovoľného primitívneho dátového typu.

Dôkladný popis všetkých inštrukcií, vrátane ich parametrov, je uvedený v špecifikácií JVM [LYBB13].

Kapitola 3

Zavádzače tried

Zavádzač je objekt zodpovedný za načítavanie tried. V JVM je reprezentovaný abstraktnou triedou *ClassLoader*. Pomocou mena *class* súboru by mal zavádzač nájsť a generovať obsah, definujúci danú triedu. Každá trieda obsahuje referenciu na konkrétny objekt typu *ClassLoader*, ktorý ju definoval. [Ora11]

Zvyčajne je trieda do JVM načítaná len v prípade, že je potrebná. Načítané sú zároveň všetky triedy, na ktoré sa odkazuje. Pomocou zavádzačov je možné za behu programu dynamicky načítať ďalšie triedy, prípadne nové inštancie pôvodných tried.

Pri štandardnom načítaní triedy niektorá z implementácií triedy *ClassLoader* vykoná nasledujúce tri kroky:

1. Skontroluje, či trieda už nebola načítaná.
2. Ak nebola, požiada nadtriedu o načítanie danej triedy.
3. V prípade, že neuspeje, pokúsi sa načítať triedu pomocou vlastného zavádzača.

3.0.3 Dynamické načítavanie tried

K načítaniu novej triedy, napríklad *MyClass*, do spusteného programu je potrebný objekt zavádzača. Získať ho je možné pomocou volania metódy *MyClass.class.getClassLoader();*. Novú triedu reprezentovanú súborom *class* následne vráti metóda *loadClass(class)* zavádzača.

3.0.4 Znovunačítanie triedy

Dynamické znovunačítanie triedy je mierne komplikovanejšie. Vstavané implementácie triedy *ClassLoader* vždy kontrolujú, či trieda už nebola do JVM načítaná. Preto nie je možné žiadnu triedu načítať dvakrát pomocou vstavaných zavádzačov. Je nutné navrhnúť vlastnú implementáciu. Nasledujúci príklad znázorňuje formu implementácie vlastného zavádzača.

Príklad zavádzača tried pre načítanie *class* súborov zo serveru

```
class NetworkClassLoader extends ClassLoader {
    String host;
    int port;

    public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassData(String name) {
        // load the class data from the connection
         ...
    }
}
```

[Ora11]

Ďalšou komplikáciou je metóda *ClassLoader.resolve()*, ktorá zabezpečuje linkovanie. Ide o metódu s atribútom *final*, z čoho vyplýva, že ju nie je možné prepísať. Nepovolí však žiadnemu zavádzaču linkovať dvakrát tú istú triedu. Preto je nutné pri každom ďalšom znovunačítaní triedy vytvoriť jeho novú inštanciu.

Kapitola 4

Byteman

Byteman je nástroj manipulujúci s bajtkódom, určený na modifikáciu Java aplikácií počas načítavania JVM, alebo za behu programu. Používa sa najmä na zjednodušenie trasovania a testovania aplikácií. Umožňuje používateľovi pridávať novú funkcionalitu do ktorejkoľvek časti programu. Funguje bez nutnosti prepisovania a opätovnej kompilácie.

Modifikácia bajtkódu aplikácie je vykonávaná za behu programu. Preto je dovolená zmena Java kódu, popisujúceho časť tried JVM ako napríklad *String*, *Thread* a podobne. Vďaka tejto funkcionalite je taktiež možné napríklad trasovanie správania sa JVM.

Byteman používa jednoduchý jazyk ECA pravidiel [1] založený na Jave. Tieto ECA pravidlá používa na špecifikáciu kde, kedy a ako má byť pôvodný Java kód transformovaný, aby modifikoval danú triedu, prípadne rozhranie [Reda].

4.1 Byteman Agent

Aby mohol *Byteman* manipulovať s programom, musí mať tento program aktivovaný takzvaný *Byteman Agent*, ktorý konfiguruje JVM pre prácu s pravidlami jeho jazyka.

Pri inštalácii agenta s prekladom programu je riešením použitie argumentu nástroja *java -javaagent*, ktorý zadáva cestu k *JAR* súboru, popisujúcemu pravidlá jazyka. Agentovi je možné pomocou argumentov konfigurovať dve možnosti funkcionality:

- Základnou možnosťou je použiť argument *script:[PATH]*, ktorý načíta do programu skript definovaný pravidlami v súbore s cestou *PATH*.
- V prípade potreby načítavania pravidiel do programu aj po spustení, je nutné nastaviť argument *listener* na hodnotu *true*. Do takto spuste-

1. ECA (*event-condition-action*) pravidlá pozostávajú z udalosti, podmienky a akcie. Význam pravidla znamená: Ak nastane udalosť, skontroluj podmienku a v prípade, že platí, vykonaj akciu [Sel95].

ného programu je možné následne pomocou skriptu *bmsubmit.sh* pridávať a odoberať ľubovoľné pravidlá.

Byteman je nastavený, aby neinjektoval kód tried JVM. Pri zmene tried ako *String a Thread* je preto nutné zmeniť túto vlastnosť nastavením prepínača [*system property org.jboss.byteman.transform.all*]. Zároveň je nutné zaistiť, aby bol *Byteman Agent* načítaný (rovnako ako tieto triedy) defaultným (*bootstrap*) zavádzačom.

Agenta je možné inštalovať taktiež do spustených aplikácií [2], bez nutnosti ich opätovného spustenia. Slúži na to skript *bminstall.sh*. *Byteman* je následne možné využiť ako nástroj na podrobnú kontrolu správania sa programu [Reda].

4.2 Štruktúra jazyka pravidiel

Pravidlá jazyka Byteman sú definované v skriptoch s príponou *btm*. Každé pravidlo pozostáva zo sekvencie definícií. Všeobecný predpis takto definovaného pravidla je nasledovný:

Kostra pravidla

```
RULE <rule name>
CLASS <class name>
METHOD <method name>
BIND <bindings>
IF <condition>
DO <actions>
ENDRULE
```

Definície musia byť zadané v správnom poradí, pričom prvou je vždy *RULE* a poslednou *ENDRULE*. Základnými kategóriami pre rozdelenie definícií pravidiel sú: Udalosti, Závislosti, Výrazy, Podmienky, Akcie a iné. Všetky dôležité kategórie a definície sú popísané v nasledujúcich sekciách kapitoly.

4.2.1 Udalosti

Udalosti pravidla (*Rule Events*) identifikujú jeho umiestnenie v metóde, nachádzajúcej sa v cieľovej triede.

Kľúčové slovo *RULE*, nasledované menom pravidla, je ľubovoľným textovým reťazcom, pričom musí obsahovať medzeru. Kvôli rozlišovaniu jednotlivých pravidiel by mali byť tieto mená unikátne.

2. Typicky ide o dlho bežiace aplikácie ako napríklad aplikačný server JBoss

Rovnako definície *CLASS* a *METHOD* sú nasledované menami triedy a metódy, do ktorej bude pravidlo načítané. Meno triedy, prípadne rozhrania je možné špecifikovať aj bez cesty k balíku, v ktorom sa nachádza. V takomto prípade, Byteman spracováva každú triedu s daným menom, ktorá je do JVM načítaná. Definíciu *CLASS* je možné nahradiť kľúčovým slovom *INTERFACE*, ktoré funguje rovnakým spôsobom ako *CLASS*, no popisuje rozhranie. Doplnením znaku ^ na začiatok mena, je možné zabezpečiť dedičnosť, t.j. prenos pravidiel na potomkov. Metódu je možné okrem samotného názvu špecifikovať aj jej návratovým typom, prípadne argumentami. Tieto bližšie špecifikácie nie sú povinné, preto je možné načítať pravidlo do viacerých preťažených metód zároveň.

Po nájdení metódy, respektíve metód, ktoré boli určené definíciami typu *CLASS*, *METHOD*, prípadne *INTERFACE*, je do každej z nich vložený spúšťač *(triggerpoint)*. Tento spúšťač presne identifikuje miesto v metóde, kde bude bajtkód injektovaný. Pomocou špecifikácie je možné zvoliť rôzne pozície jeho umiestnenia v triede. Predvolenou hodnotou je *AT ENTRY*, čo zaistí vloženie spúšťača na začiatok, teda pred prvú inštrukciu [3] danej metódy. Ďalšími možnosťami umiestnenia tohto spúšťača sú napríklad: *AT EXIT*, *AT LINE*, *AT READ*, *AT WRITE*. Tieto definície vkladajú spúšťač na koniec metódy, prípadne pred operáciu čítania, alebo zápisu do premennej. Tabuľka všetkých možností umiestnení sa nachádza v prílohe A.6.

4.2.2 Závislosti

Skript definujúci pravidlo jazyka *Byteman* je možné obohatiť o závislosti pravidiel (*Rule Bindings*). Tieto závislosti počítajú hodnoty niektorých premenných, ktoré môžu byť použité v ďalšom tele pravidla. Sú počítané pri každom spustení pravidla. Závislosti sú definované pomocou klauzule *BIND*, nasledovanej názvom a prípadne typom premennej. Každej premennej je pomocou výrazu nasledujúcim znamienko „=“ priradená špecifická hodnota, napríklad:

Závislosť

```
RULE Bindings example
...
BIND thisClass = $0;
...
ENDRULE
```

3. Výnimkou sú inštrukcie volajúce konštruktor predka, prípadne alternatívny konštruktor.

Vytvára premennú *thisClass*, ktorej bude automaticky odvodený dátový typ a priraďuje jej hodnotu reprezentujúcu metódu tohto pravidla.

4.2.3 Výrazy

Výrazy (*Rule expressions*) sa nachádzajú na pravej strane definície závislosti. Existujú dva základné typy výrazov:

- Jednoduché výrazy ako: referencie na predošlé závislosti, vstavané operátory (*$!, $; $@, ...*), referencie na lokálne premenné v okolí spúšťača a mnohé iné.
- Výrazy zložené z iných výrazov pomocou štandardných operátorov jazyka Java.

Podmienkami pravidla (Rule Conditions) sú výrazy typu *boolean*. Tieto pravidlá nasledované klauzulu *IF* sú overené po inicializácií závislostí.

4.2.4 Akcie

Jednou z najdôležitejších súčastí pravidla sú akcie pravidiel (*Rule Actions*). Sú tvorené výrazmi, návratovými klauzulami, prípadne klauzulou *throw*. Na začiatku definície je klauzula *DO*, ktorá je nasledovaná jednotlivými akciami. Každá akcia je na samostatnom riadku, oddeľuje ich bodkočiarka.

Príklad použitia akcie

```
RULE Actions example
...
DO System.out.println("This method is:" + $0);
   return;
...
ENDRULE
```

Tento príklad znázorňuje zjednodušené pravidlo, ktorého akciou je výpis názvu metódy na štandardný výstup.

4.3 Použitie

Primárne bol *Byteman* určený na podporu testovania multivláknových a multi-JVM aplikácií za použitia techniky nazývanej *Fault Injection* [4]. Zahŕňa

4. V oblasti testovania softvéru tento pojem označuje techniku pre zlepšenie pokrytia testov vyvolaním chyby. Odstraňuje najmä chyby, ktoré by sa inak vyskytovali len zriedka. [Roe11]

preto funkcionalitu, ktorá bola navrhnutá na riešenie problémov súvisiacich s týmto typom testovania. *Byteman* poskytuje podporu automatizácie v štyroch hlavných oblastiach:

- trasovanie špecifických väzieb kódu a zobrazovanie stavu danej aplikácie, prípadne JVM,
- narúšanie normálneho priebehu zmenou stavov, volanie neplánovaných metód, vynucovanie návratových volaní, prípadne vyvolávanie neočakávaných výnimiek,
- organizácia časovania aktivít vykonávaných nezávislými vláknami aplikácie,
- monitorovanie a zhromažďovanie štatistík, pre sumarizáciu aplikácie a operácií JVM.

V súčasnosti je *Byteman* využívaný oveľa širšie pôvodný ako nástroj na testovanie [Reda].

Najjednoduchším použitím *Bytemana* je vkladanie kódu, ktorý trasuje správanie sa aplikácie. Zároveň môže byť využitý na monitorovanie, alebo ladenie, ako aj na úpravu kódu pri testovaní a overenie správneho fungovania aplikácie. Pri vkladaní kódu na veľmi špecifické miesta je možné vyhnúť sa režijným nákladom, ktoré často rastú pri ladení, alebo trasovaní produktu [RH].

Kapitola 5

Javassist

Ďalším nástrojom určeným na manipuláciu s bajtkódom je *Javassist*. Tento nástroj využíva na manipuláciu s bajtkódom odlišný prístup ako *Byteman*. Narozdiel od ECA pravidiel, *Javassist* používa na reprezentáciu *class* súborov triedu *Javassist.CtClass*. Súbory je možné pomocou tejto triedy modifikovať [1] a následne modifikácie zapísať.

Na modifikáciu definície triedy je nutné najskôr získať referenciu na objekt *CtClass* z objektu *ClassPool* pomocou jeho metódy *get()*.

Získanie objektu *CtClass*

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("test.Rectangle");
```

Vo vyššie uvedenom príklade je objekt typu *CtClass*, ktorý reprezentuje triedu *test.Rectangle* vrátený objektom *ClassPool* a uložený do premennej *cc*. Samotný objekt *ClassPool* vrátila metóda *getDefault*, ktorá prehľadáva defaultnú systémovú cestu. Z implementačného pohľadu je *ClassPool* hešovacia tabuľka objektov *CtClass*, ktorá používa mená tried ako kľúče. Metóda *get()* v *ClassPool* prehľadáva túto hešovaciu tabuľku, aby našla objekty typu *CtClass*, príslušné danému kľúču [Shi].

Ďalšou možnosťou ovplyvňovania správania sa aplikácie je pridanie novo definovanej triedy. *Javassist* túto funkcionalitu umožňuje pomocou metódy *makeClass()*, volanej na objekte typu *ClassPool*.

Definícia novej triedy

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.makeClass("Point");
```

V tomto prípade je pomocou získaného kontajnera *pool* definovaná nová trieda *Point*.

---

1. Možnosti modifikácie triedy pomocou *CtClass* popisuje kapitola 5.1.

5.1 Modifikácie

Objekt *CtClass* implementuje množstvo metód určených na modifikáciu triedy, ktorú reprezentuje. Výhodou použitia tohto nástroja spočíva aj v kompatibilite s rozhraním *Java Reflection* [2]. *CtClass* poskytuje metódy *getName()*, *getSuperClass()*, *getMethods()* a mnohé iné. Taktiež obsahuje metódy pre úpravu definície triedy. Povoľuje pridanie nového atribútu, konštruktora a metódy, prípadne modifikáciu tela existujúcej metódy [Shi].

5.1.1 Rozhranie pre prácu s bajtkódom

Okrem štandardného rozhrania popísaného v predchádzajúcich odsekoch poskytuje *Javassist* aj rozhranie pre priamu prácu s bajtkódom požadovanej triedy, respektíve metódy.

Rozhranie pracuje priamo s bajtkódom *class* súboru. Napríklad metóda *getClassFile()* triedy *CtClass* vracia objekt typu *ClassFile*, reprezentujúci daný súbor. Podobne metóda *getMethodInfo()* triedy *CtMethod* vracia objekt typu *MethodInfo*, predstavujúci štruktúru *method_info* daného *class* súboru. Rozhranie používa notáciu JVM popísanú v kapitole 2 a podrobne v špecifikácii JVM [LYBB13].

5.2 Zápis do *class* súboru

Modifikácie vykonané v bajtkóde načítaných, prípadne novo vytvorených objektov, sa prejavia ihneď po zavolaní metódy *writeFile()* triedy *CtClass*. Táto metóda preloží objekt *CtClass* do *class* súboru, ktorý zapíše na disk. *Javassist* taktiež poskytuje metódu *toBytecode()*, ktorá vráti modifikované inštrukcie bajtkódu do poľa typu *byte* [Shi].

2. *Java Reflection* je rozhranie bežne používané programami, ktoré vyžadujú schopnosť modifikácie správania sa spustenej Java aplikácie. [Ora]

Kapitola 6

Porovnanie

Každý zo spomínaných nástrojov pristupuje k manipulácií s bajtkódom odlišným spôsobom. Zatiaľ čo *Byteman* využíva jazyk ECA pravidiel, ktoré načítava pomocou takzvaného agenta, *Javassist* je knižnicou jazyka Java. Táto kapitola oba nástroje stručne hodnotí a porovnáva:

6.1 Vlastnosti nástroja *Byteman*

Výhodami použitia nástroja *Byteman* sú:

- Prehľadnosť jazyka: Jazyk ECA pravidiel použitý nástrojom *Javassist* je veľmi prehľadný a dobre zapamätateľný.
- Možnosť automatického generovania skriptov: Vďaka jednoznačnej štruktúre jazyka a jeho skriptov je možné pravidlá pomerne jednoducho generovať automaticky.

Na druhej strane má tento nástroj aj nevýhody. Najvýraznejšími z nich sú nasledujúce:

- Nutnosť manipulácie s vstavaným programom *java agent* spustenej aplikácie.
- Obmedzená funkcionalita: Najväčšou nevýhodou nástroja *Byteman* je nemožnosť editovania pôvodného kódu. *Byteman* slúži ako nástroj na pridávanie novej funkcionality, nijakým spôsobom však nedokáže editovať, prípadne odstrániť funkcionalitu pôvodnú.

6.2 Vlastnosti knižnice *Javassist*

Použitie knižnice *Javassist* má nasledujúce výhody:

- Jednoduchá manipulácia: Keďže je *Javassist* knižnicou jazyka Java, je manipulácia s ním pomerne nenáročná.

- Dvojité aplikačné rozhranie: *Javassist* poskytuje, na rozdiel od nástroja *Byteman*, okrem základného aj nízkoúrovňové rozhranie pre priamu manipuláciu s bajtkódom.
- Široké uplatnenie: Na rozdiel od nástroja *Byteman*, funkcionalita knižnice *Javassist* poskytuje aj nástroje na úpravu existujúceho kódu.

Takisto tento nástroj má niekoľko slabých stránok, nie sú však výraznými prekážkami v následnej manipulácií s ním:

- Mohutnosť aplikácie: Pri jednoduchých úlohách by mohlo mať použitie knižnice *Javassist* za následok značné časové oneskorenie.
- Nemožnosť automatizácie: Vzhľadom na povahu tejto knižnice nie je možné automatické generovanie zdrojového kódu jazyka Java pre manipuláciu s bajtkódom.

6.3 Zhrnutie

Z vyššie uvedených vlastností nástrojov *Byteman* a *Javassist* vyplývajú možnosti ich využitia. Vo väčšine aplikácií pre manipuláciu s bajtkódom spôsobom, ktorý modifikuje pôvodnú funkcionalitu, je nutné použitie knižnice *Javassist*. Pre úpravy programov, napríklad v prípade trasovania správania sa aplikácií, je vhodnou alternatívou nástroj *Byteman*.

Kapitola 7

Praktické ukážky

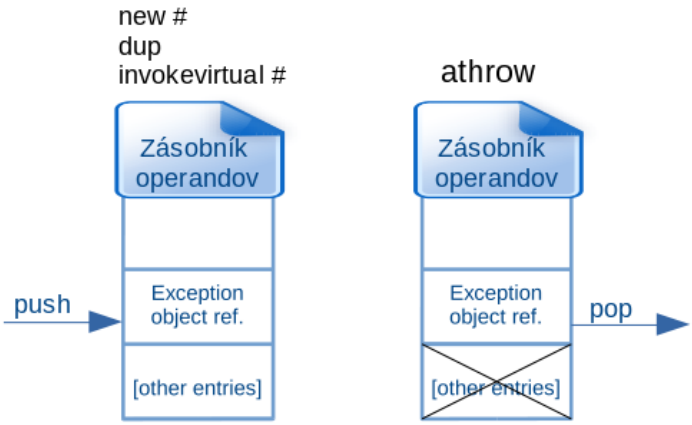
Kľúčovou súčasťou bakalárskej práce sú praktické ukážky. Každý z príkladov je navrhnutý pre jeden z nástrojov *Byteman*, respektíve *Javassist*. Konkrétne demonštrujú základné možnosti jeho využitia pre vývoj a trasovanie programov v Jave. Ukážky pokrývajú štyri oblasti:

- detekcia volania výnimiek,
- detekcia nesprávneho ošetrenia výnimiek,
- zlepšenie a sprehľadnenie produkčného kódu,
- optimalizácia neefektívnych častí kódu.

Príklady obsahujú dve ukážky z oblasti detekcie volania a ošetrovania výnimiek a dve ukážky, zamerané na optimalizáciu a zlepšenie kódu. V praxi sa nástroje ako *Byteman* a *Javassist* využívajú najmä v aplikačných serveroch, prípadné iných projektoch, ktorých opätovná kompilácia by bola príliš časovo a technicky náročná. V ukážkach budem z praktických dôvodov využívať na testovanie ich funkcionality menšie programové celky a demonštračné programy.

7.1 Detekcia volania výnimiek

Prvou z ukážok je detekcia volania výnimiek spusteného programu. Ide o detekciu všetkých výnimiek, ktoré sú v požadovanej triede volané pomocou kľúčového slova *throw*. V bajtkóde je konštrukcia tohto kľúčového slova reprezentovaná inštrukciou *athrow*, ktorá zavolá výnimku predtým pridanú na vrchol zásobníka a zároveň zásobník vyčistí.



Obr. 7.1: Grafické znázornenie pridania objektu výnimky a jej následné vyvolanie pomocou inštrukcie *athrow*, zdroj: vlastné spracovanie

Detekcia volaných výnimiek je vhodným príkladom na ukážku využitia nástroja *Byteman*. Ako bolo uvedené v kapitole 4, *Byteman* využíva na popis modifikácie bajtkódu ECA pravidlá. Zovšeobecnené ECA pravidlá pre tento príklad sú v nasledujúcom tvare.

Všeobecný formát ECA pravidla pre detekciu výnimiek metódy <m> volanej z triedy <C>

```
RULE detect throw, method <m>, class <T>
  CLASS <T>
  METHOD <m>
  AT THROW ALL
  BIND exception:Throwable= $^
  IF true
  DO System.out.println("Detected athrow, exception: " + exception)
ENDRULE
```

Klauzula *RULE* udáva názov pravidla pre konkrétnu metódu a triedu. Nasledujú klauzuly *CLASS* a *METHOD*, ktoré špecifikujú ich názvy. V ďalšej časti pravidla sa nachádza jeho logika, ktorá popisuje zachytávanie výnimiek a reakciu na ich volanie v podobe výpisu na štandardný výstup.

Dôležitou súčasťou ukážkového programu je skript *loadScripts.sh*, ktorý tieto pravidlá generuje a načítava do už spustenej aplikácie [1]. Jediným potrebným argumentom je cesta ku *class* súboru triedy, ktorej pravidlá bude

1. Keďže skript pravidlá ihneď po vygenerovaní načítava, je nutné, aby už pred jeho spustením bežala aplikácia, ktorej výnimky bude sledovať. Táto aplikácia musí byť spustená s prepínačom *agent listener*.

skript generovať. Program následne monitoruje metódy zadanej triedy, vrátane konštruktora. Taktiež je možné generovať a načítavať pravidlá pre viacero tried zároveň. V tomto prípade je nutné zadať skriptu *loadScripts.sh* cesty k ich *class* súborom oddelené medzerou.

Po vygenerovaní a načítaní pravidiel program reaguje na každú volanú výnimku zadanej triedy, respektíve tried a pri detekcii na ňu upozorní. Pre účely tejto práce používame ako demonštračný príklad aplikáciu s názvom *fileChooser* [2]. V reálnom prostredí by bolo možné detekciu volania výnimiek využiť napríklad v aplikačných serveroch, kde nie je volanie niektorých z nich vždy viditeľné.

7.2 Detekcia nesprávneho ošetrenia výnimiek

Nasledujúci príklad sa zaoberá detekciou nesprávne ošetrených výnimiek. V prípade zachytenia výnimky *catch* blokom by mal program na túto situáciu vždy nejakým spôsobom reagovať (napríklad: logovaním udalosti, volaním inej výnimky, riadeným pádom programu, ...). Prázdne *catch* bloky sú preto vo väčšine prípadov nesprávnym ošetrením danej výnimky.

Keďže ide o ukážku nástroja *Javassist* projektom je Java aplikácia vo formáte *Maven*. Program postupne prechádza všetky metódy a konštruktory triedy, ktorú monitoruje. Pri nájdení prázdneho *catch* bloku uloží informácie o jeho polohe do logu a na záver vypíše získané údaje.

Výstup aplikácie po kontrole triedy example.tables.JDBCAdapter

```
- Class JDBCAdapter -
Apr 28, 2015 6:15:18 PM application.CatchBlockTracer trace
INFO: -> Suspicious catch block found on line: 116 in method:
      example.tables.JDBCAdapter.executeQuery(java.lang.String)
Apr 28, 2015 6:15:18 PM application.CatchBlockTracer trace
INFO: -> Suspicious catch block found on line: 266 in method:
      example.tables.JDBCAdapter.setValueAt(java.lang.Object,...)
Apr 28, 2015 6:15:18 PM application.CatchBlockTracer trace
INFO: -> Suspicious catch block found on line: 81 in method:
      example.tables.JDBCAdapter(java.lang.String,...)
Apr 28, 2015 6:15:18 PM application.CatchBlockTracer trace
INFO: -> Suspicious catch block found on line: 79 in method:
      example.tables.JDBCAdapter(java.lang.String,...)
SUMMARY: 4 suspicious catch blocks found in class JDBCAdapter
```

V tomto výstupe vidíme, že boli nájdené 4 nesprávne ošetrené výnimky na riadkoch: 79, 81, 166, 266.

2. Tento demonštračný príklad bol prevzatý z balíka Java Development Kit Demos and Samples.

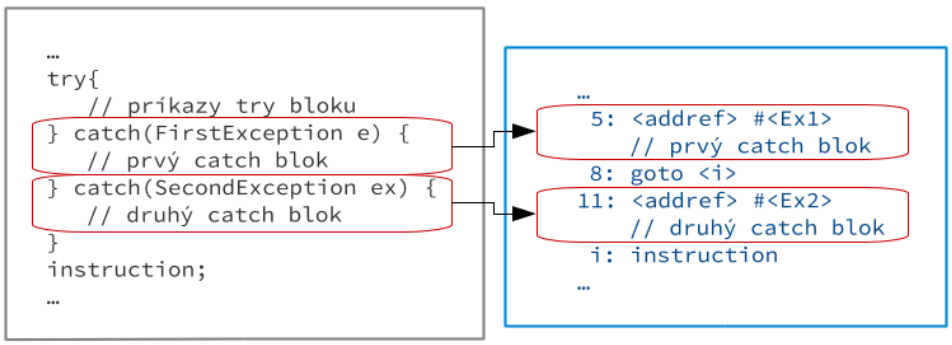
7.2.1 Štruktúra a funkčná logika aplikácie

Aplikačnú logiku som rozdelil medzi nasledujúce štyri triedy balíka *application*: *Tracer*, *CatchBlockTracer*, *CBDetector* a *CBIndicesHandler*.

Trieda *Tracer* slúži na spustenie aplikácie pre ľubovoľnú triedu <C>. Obsahuje metódu *traceCatchBlocks(Class classToTrace)*, ktorej argumentom je objekt reprezentujúci <C>. Táto metóda prevedie triedu na objekt *CtClass* a následne z neho získa všetky metódy a konštruktory v podobe polí tried *CtMethod* a *CtConstructor*. Následne za pomoci triedy *CatchBlockTracer* získa informácie o polohe všetkých prázdnych *catch* blokov týchto metód a konštruktorov.

Ako bolo uvedené v predchádzajúcom odseku, trieda *CatchBlockTracer* je určená na spracovanie objektov *CtMethod* a *CtConstructor*. Hlavnou metódou triedy je *trace(final CtBehavior cm)*. Metóda má jeden argument typu *CtBehavior*, ktorého potomkami sú práve triedy *CtMethod* a *CtConstructor*. Uchováva si informácie o pozícií *catch* blokov v podobe mapy, ktorej kľúčom je index prvej inštrukcie *catch* bloku v bajtkóde a hodnotou je jeho pozícia v zdrojovom súbore. Túto mapu získava za pomoci tried *CBDetector* a *CBIndicesHandler*. Po získaní údajov o polohe všetkých blokov v danej metóde je každý z nich skontrolovaný metódou *isEmpty*. V prípade preukázania nesprávne ošetrenej výnimky sú informácie *catch* bloku uložené a logované.

Mechanizmus kontroly prázdneho bloku prebieha pomocou nízkoúrovňového rozhrania knižnice *Javassist* pre prácu s bajtkódom. V bajtkóde je každý *catch* blok reprezentovaný vo forme znázornenej na obrázku 7.2.



Obr. 7.2: Grafické znázornenie reprezentácie *catch* bloku v bajtkóde, zdroj: vlastné spracovanie

Na začiatku je na zásobník vložená referencia na objekt danej výnimky. Nasleduje telo *catch* bloku. Premenná <*i*> reprezentuje index inštrukcie vykonávanej bezprostredne po poslednom bloku. V prípade, že aktuálny *catch* blok nie je posledným v rade, je za jeho telo vložená inštrukcia *goto*, ktorá odkazuje na index inštrukcie <*i*>. Metóda *isEmpty* preto kontroluje, či telo bloku obsahuje iba inštrukciu skoku na index <*i*>, prípadne či je úplne prázdne.

Poslednými triedami na spodnej časti hierarchie v abstrakcii získavania polohy *catch* blokov sú *CBDetector* a *CBIndicesHandler*. Slúžia na získanie mapy indexov *catch* blokov pre zadanú metódu, prípadne konštruktor. Dôležitým nástrojom samotného vyhľadávania je metóda triedy *CtBehavior instrument(ExprEditor)*. Argumentom tejto metódy je práve objekt triedy *CBIndicesHandler*.

7.2.2 Testovacie príklady

Okrem samotnej aplikácie projekt obsahuje aj dva balíky *example.simple* a *example.tables*, na ktorých je možné funkcionalitu aplikácie testovať. Každý z balíkov obsahuje vlastnú spustiteľnú triedu *Demo*, ktorá volá metódu *application.Tracer.traceCatchBlocks* hlavnej aplikácie na jednotlivých testovacích triedach.

Balík *example.simple* obsahuje dve testovacie triedy, ktoré som vytvoril pre overenie funkčnosti aplikácie balíka *application*. Po spustení aplikácia upozorní na nesprávne ošetrenie *catch* blokov v triede *PersonFactory*.

Balík *example.tables* obsahuje komplexnejšie testovacie triedy, prevzaté z ukážok *Java Development Kit Demos and Samples*. Výstupom spustenia triedy *Demo* je kontrola tried *JDBCAdapter*, *OldJTable* a *TableExample*. Na nesprávne ošetrené výnimky sú tentokrát upozornené metódy a konštruktor triedy *JDBCAdapter*.

Triedy testovacích balíkov je možné ľubovoľne modifikovať pre jednoduché testovanie funkcionality aplikácie. V prípade potreby testovania aplikácie na novej triede je nutné volať metódu hlavného balíka *application Tracer.traceCatchBlocks* s argumentom špecifikujúcim danú triedu.

7.3 Zlepšenie a sprehľadnenie produkčného kódu

Program prispievajúci k zlepšeniu produkčného kódu sa špecializuje na náhradu priamych volaní atribútov generovanými *get* a *set* metódami. Vo všeobecnosti aplikácia manipuluje s bajtkódom dvoch tried. Prvou z tried je <*C1*>. Obsahuje atribúty, ku ktorým však neexistujú prístupové *get* a *set*

metódy, atribúty sú teda používané priamo. Trieda <*C2*> zapisuje a číta obsah týchto atribútov <*C1*>. Popísaná implementácia porušuje základný princíp zapuzdrenia, ktorý by mal byť dodržaný za každých okolností. Projekt pod názvom *code-improvement* preto slúži na generovanie prístupových metód k atribútom triedy <*C1*> a následnú náhradu priamych volaní týmito metódami v triede <*C2*>.

Aplikácia manipuluje s bajtkódom a *class* súbormi požadovaných tried. Zmeny sa preto neprejavia v zdrojovom kóde. Rovnako by nemali nijako ovplyvniť vnútornú logiku modifikovaných tried. Okrem informácií zobrazených na štandardnom výstupe, je zmeny možné pozorovať aj v *class* súboroch tried [3].

7.3.1 Štruktúra a funkčná logika aplikácie

Aplikácia *code-improvement* je projektom typu *Maven*. Demonštruje využitie nástroja *Javassist*. Projekt je tvorený dvoma balíkmi: *application* a *example*. Balík *application* obsahuje samotný program nahradzujúci priame volania atribútov prístupovými metódami, ktorého základná funkcionalita bola načrtnutá v predchádzajúcich odsekoch. Balík *example* je jednoduchým testovacím príkladom pre kontrolu funkčnosti tried balíka *application*.

Hlavným balíkom zabezpečujúcim fungovanie programu je *application*. Funkcionalitu som tentokrát rozložil medzi 3 triedy: *FieldChecker*, *AccGenerator* a *AccCreator*.

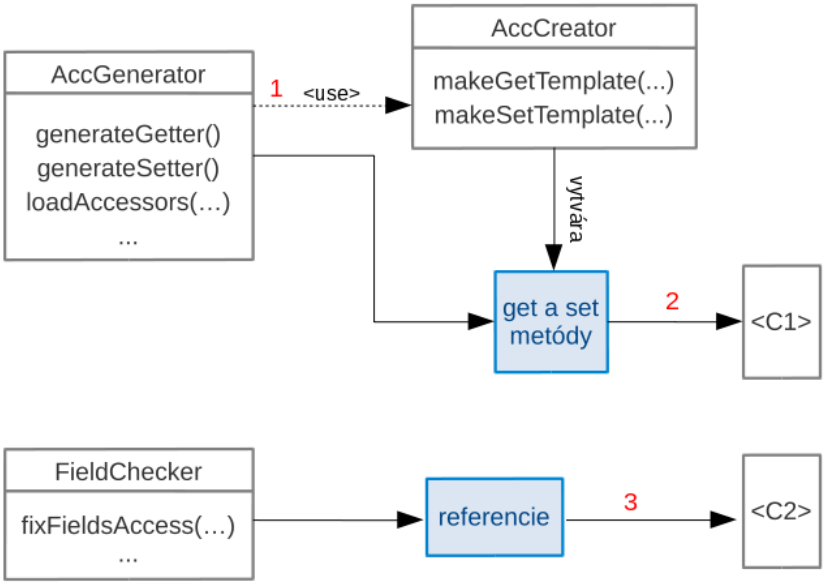
Trieda *FieldChecker* je jedinou triedou, ktorá slúži na prístup k aplikácii. Po vytvorení objektu *FieldChecker* pre triedy <*C1*> a <*C2*> dôjde volaním metódy *FieldChecker.fixFieldsAccess()* k manipulácií s bajtkódom <*C1*> a <*C2*> spôsobom popísaním v prvom odseku tejto kapitoly. Metóda je preto len prístupovým bodom, ktorý využíva funkcionalitu ostatných metód a tried balíka *application*.

Program postupne iteruje cez všetky atribúty triedy <*C1*>. Pre každý z nich vytvorí *AccGenerator* a následne *AccCreator* prístupové *get* a *set* metódy. Akonáhle sú tieto metódy úspešne načítané do bajtkódu <*C1*>, vykoná metóda *replaceFieldAccess()* náhradu priameho volania atribútu prístupovými metódami. Využíva k tomu najmä nástroj *CodeConverter* knižnice *Javassist*.

Za generovanie a načítanie prístupových metód je zodpovedná trieda *AccGenerator* a jej pomocná trieda *AccCreator*. Táto pomocná trieda slúži na generovanie konkrétnych šablón prístupových metód pre zadaný atribút. Šablóny sa následne trieda *AccGenerator* pokúsi načítať do <*C1*>.

---

3. Class súbory je možné zobraziť v čitateľnej podobe napríklad prostredníctvom nástroja *javap* príkazom *[javap -c <path>]*.



Obr. 7.3: Grafické znázornenie funkcionality tried zodpovedných za vytváranie prístupových metód a následnú náhradu volaní atribútov, zdroj: vlastné spracovanie

7.3.2 Testovací príklad

Balík *example* obsahuje jednoduchý príklad na otestovanie vyššie popísanej funkcionality. Spustiteľnou triedou daného príkladu je *example.Demo*. Po jej spustení sa program pokúsi:

1. Vytvoriť a inicializovať triedu *application.FieldChecker*.
2. Vykonať modifikáciu tried *example.Initializer* a *example.Triangle* (nedodržiavajú princíp zapuzdrenia) volaním metódy *fixFieldsAccess()*.
3. Skontrolovať funkčnosť modifikovaných tried ich použitím.

V prípade úspešnej modifikácie požadovaných tried by mal výstup vyzerať približne nasledovne:

Výstup testovacieho príkladu pre aplikáciu *code-improvement*

```
+ Getter for field: [a] in class [example.Triangle] was
  successfully created
+ Setter for field: [a] in class [example.Triangle] was
  successfully created
```

```
-> Read and write operations replaced for field
   [Triangle.a] in class [example.Initializer]

+ Getter for field: [b] in class [example.Triangle] was
  successfully created
+ Setter for field: [b] in class [example.Triangle] was
  successfully created
-> Read and write operations replaced for field
   [Triangle.b] in class [example.Initializer]

+ Getter for field: [c] in class [example.Triangle] was
  successfully created
+ Setter for field: [c] in class [example.Triangle] was
  successfully created
-> Read and write operations replaced for field
   [Triangle.c] in class [example.Initializer]

------------------------------
Trying to use modified classes:

Created: Circle {radius is 5.0Vertex centre: [0.0, 0.0]}
Created: Vertices of Triangle are {Vertex A: [-1.0, -1.0],
         Vertex B: [1.0, -1.0], Vertex C: [1.0, 1.0]}

Get and set methods successfully tested.
```

Ak modifikácia neprebehla, napríklad v prípade opätovného spustenia na rovnakých triedach, vypíše program hlášku, ktorá upozorňuje užívateľa, že triedy, ktoré zadal, už boli v minulosti upravené.

Program je možné aplikovať na ľubovoľné triedy, následne definované v konštruktore *application.FieldChecker* [4].

7.4 Optimalizácia neefektívnych častí kódu

Posledný z príkladov sa zaoberá optimalizáciou neefektívne generovaných častí bajtkódu. Vo všeobecnosti vykonáva na bajtkóde mnoho optimalizácií JVM. Existuje však mnoho druhov ďalších úprav, pomocou ktorých je možné bajtkód výrazne zefektívniť.

Jednými z najčastejšie sa vyskytujúcich inštrukcií sú inštrukcie typu *store* a *load*. Ich úlohou je vkladanie, respektíve výber položiek zo zásobníka. Pri opakovanej modifikácií jednej premennej teda vzniká veľké množstvo nadbytočných operácií zápisu a čítania jej hodnoty. Vhodnou optimalizáciou je preto odstránenie nadbytočných inštrukcií.

---

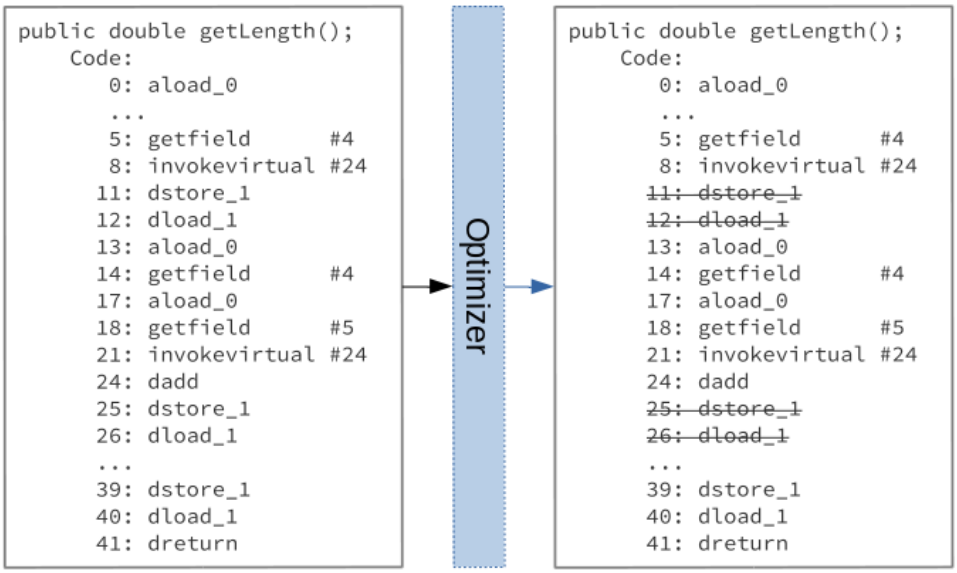
4. V prípade, že ide o triedy mimo projektu *code-improvement* je potrebné zadať do argumentu konštruktora aj cestu k balíku obsahujúcemu *class* súbory triedy <*C2*>.

Cieľom tohto programu je identifikovať a odstrániť nadbytočné inštrukcie čítania a zápisu v prípade aritmetických operácií na premenných typu *double*.

Keďže vyžaduje priamu prácu s bajtkódom a jeho modifikáciu, je táto optimalizácia vhodným príkladom pre demonštráciu funkcionality oboch rozhraní knižnice *Javassist*.

7.4.1 Štruktúra a funkčná logika aplikácie

Triedy popisujúce logiku aplikácie sa opäť nachádzajú v balíku *application*. Program sa skladá z troch tried: *ArithmeticOptimizer*, *MethodModifier* a *InstructionVerifier*. Triedou pre prístup k programu je *ArithmeticOptimizer*. Zvyšné triedy sú prístupné len v rámci balíka *application*, mimo neho by nemali byť nijako používané.



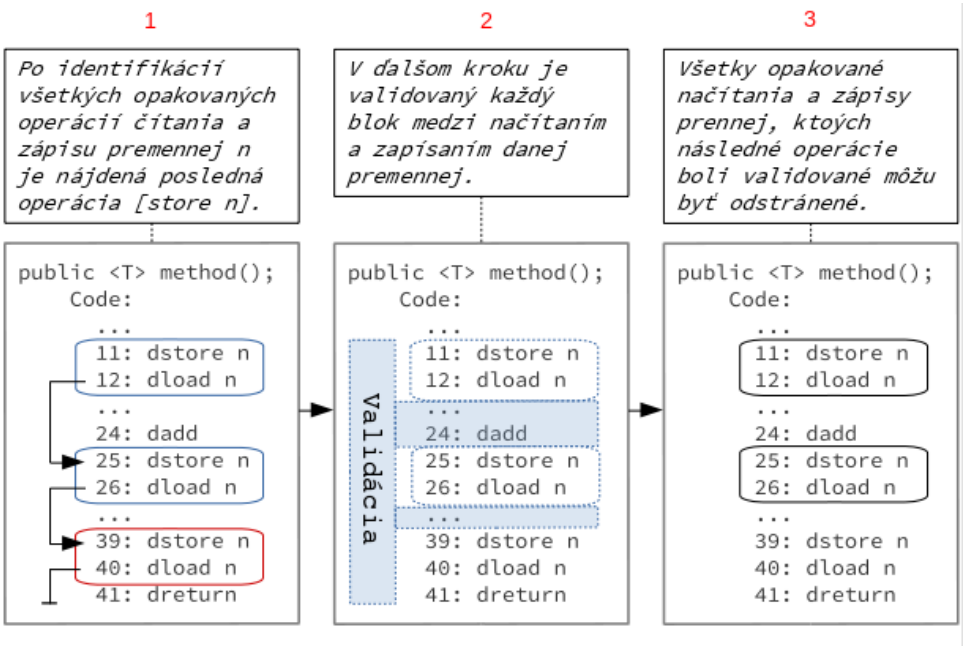
Obr. 7.4: Znázornenie optimalizácie bajtkódu jednej z metód balíka *example.model*, zdroj: vlastné spracovanie

Kľúčovou triedou programu je *ArithmeticOptimizer* s prístupovou metódou *optimizeClass(String classNameToOpt)*. Po jej inicializácií a spustení program iteruje cez všetky metódy triedy definovanej jediným argumentom *classNameToOpt*. Každá metóda je následne optimalizovaná pomocou

triedy *MethodModifier*. Na záver je prepísaný *class* súbor danej triedy a poskytnutá sumarizácie zmien.

Ako bolo uvedené vyššie, trieda *MethodModifier* slúži na priamu optimalizáciu zadanej metódy. Daná metóda je reprezentovaná pomocou triedy *MethodInfo*, ktorá knižnici *Javassist* slúži ako popis rovnomenného atribútu *class* súboru. Najdôležitejšou metódu triedy *MethodModifier* je *optimize()*. Táto metóda prechádza všetky inštrukcie bajtkódu a v prípade nájdeného nadbytočného páru *dstore*, *dload* tieto inštrukcie odstráni.

Program zvyčajne spracováva viac než jednu premennú pre danú metódu. Obrázok 7.5 popisuje kvôli prehľadnosti proces identifikácie nadbytočných inštrukcií čítania a zápisu jednej premennej <*n*>:



Obr. 7.5: Znázornenie identifikácie nadbytočných inštrukcií čítania a zápisu premennej, zdroj: vlastné spracovanie

Ako nástroj na identifikáciu typu aktuálnej inštrukcie slúži programu trieda *InstructionVerifier*. Obsahuje metódy určujúce inštrukcie *dstore* a *dload*. Pomocou týchto metód je zároveň možné získať hodnotu argumentu spomínaných inštrukcií. Metóda *isArithmetic(int op)* rozhoduje, či inštrukcia definovaná argumentom *op* reprezentuje aritmetickú operáciu.

Balík *example* obsahuje dva príklady, na ktorých je možné aplikáciu testovať. Každý z príkladov má spustiteľnú triedu *Demo*, ktorá sa pokúsi optimalizovať vlastnú testovaciu triedu. Balík *example.simple* optimalizuje jednoduchú triedu *ArithmeticExample*, ktorá vykonáva aritmetické operácie. Balík *example.model* je príklad podobný demonštračnému príkladu úlohy z predchádzajúcej kapitoly. Podrobné fungovanie testovacích príkladov je uvedené v *Javadocu* a komentároch [5].

Aplikáciu je takisto možné spustiť na ľubovoľnej triede vhodnej na optimalizáciu aritmetických operácií volaním metódy *application.ArithmeticOptimizer.optimizeClass(String classNameToOpt)*.

5. Podrobnosti o uvedených príkladoch sú uvedené aj v *README* súbore projektu.

Kapitola 8

Záver

Hlavným cieľom práce bola analýza, porovnanie a implementácia sady príkladov nástrojov určených na manipuláciu s bajtkódom jazyka Java. Nástrojmi použitými v tejto práci boli *Byteman* a *Javassist*.

Problematika manipulácie s bajtkódom má veľmi široké uplatnenie v mnohých odvetviach vývoja a testovania softvéru. Demonštračné príklady boli preto orientované na štyri vybrané oblasti:

- detekcia volania výnimiek,
- detekcia nesprávneho ošetrenia výnimiek,
- zlepšenie produkčného kódu,
- optimalizácia neefektívnych častí kódu.

Analýzou vlastností spomínaných nástrojov vyplynuli možnosti ich funkcionality a následné uplatnenie v príkladoch.

Z vlastností nástroja *Byteman* vyplýva, obmedzenie jeho použitia u obmedzenej množiny aplikácií. Tento nástroj nie je možné využiť na editovanie pôvodného bajtkódu. Jeho funkcionalita je týmto obmedzená na pridávanie vlastného kódu. Uplatnenie bolo možné v prípade detekcie volania výnimiek spustenej aplikácie. Pri používaní sa nástroj javí stabilný a veľmi dobre ovládateľný.

Knižnica *Javassist* bola vďaka svojej rozsiahlej funkcionalite použitá takmer vo všetkých príkladoch. Manipulácia s týmto nástrojom bola o niečo náročnejšia. Výsledné príklady sú však badateľne využiteľnejšie. Najmä programy v oblasti zlepšenia a optimalizácie kódu by bolo po rozpracovaní možné použiť v praxi.

Vo väčšine projektov, ktoré manipulujú s bajtkódom je možné použiť knižnicu *Javassist*. Nástroj *Byteman* je však vhodnou alternatívou najmä v oblastiach trasovania a testovania programov.

Literatúra

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013. http://docs.oracle.com/javase/specs/jvms/se7/html/.

[Ora] Oracle. *Java Reflection API*. http://docs.oracle.com/javase/tutorial/reflect/index.html.

[Ora11] Oracle. *Class ClassLoader documentation*, 7th edition, 2011. http://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html.

[Reda] Red Hat and individual contributors. *Byteman Programmer's Guide*. http://downloads.jboss.org/byteman/2.2.1/ProgrammersGuide.pdf.

[Redb] Red Hat and individual contributors. *Javassist Online API manual*. http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/html/index.html.

[RH] Inc. Red Hat. Byteman project description. http://byteman.jboss.org. Accessed: 2015-03-09.

[Roe11] Kevin Roebuck. *Software Testing: High-impact Strategies - What You Need to Know Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Tebbo, 2011.

[Sel95] Timos Sellis. *Rules in Database Systems: Second International Workshop, RIDS '95, Glyfada, Athens, Greece, September 25 - 27, 1995. Proceedings*. Lecture Notes in Artificial Intelligence. Springer, 1995.

[Shi] Shigeru Chiba. *Getting Started with Javassist*. http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/tutorial/tutorial.html.

Dodatok A

Tabuľky

Zdrojom tabuliek 1 až 5 je špecifikácia JVM [LYBB13].

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Tabuľka A.1: Tabuľka značiek určujúcich typ záznamu v *constant_pool*. Stĺpec *Constant Type* označuje názov typu, stĺpec *value* prirad'uje každému typu číselnú hodnotu.

Meno Indikátora	Hodnota	Interpretácia
ACC_PUBLIC	0x0001	Deklarovaná ako verejná; prístupná aj mimo balíka.
ACC_FINAL	0x0010	Deklarovaná ako final; žiadne podtriedy po inicializácií.
ACC_SUPER	0x0020	Volá metódu nadtriedy, hlavne inštrukcia invokespecial.
ACC_INTERFACE	0x0200	Je rozhranie, nie trieda.
ACC_ABSTRACT	0x0400	Deklarovaná ako abstraktná, nemôže byť inštanciovaná.
ACC_SYNTHETIC	0x1000	Deklarovaná ako synthetic, nieje prítomná v zdrojovom kóde.
ACC_ANNOTATION	0x2000	Deklarovaná ako typ annotation.
ACC_ENUM	0x4000	Deklarovaná ako typ enum.

Tabuľka A.2: Tabuľka indikátorov prístupových práv *ClassFile* štruktúry.

Reprezentácia pomocou reťazca	**Typ**	**Interpretácia**
B	byte	znamienkové celé číslo veľkosti jedného bajtu
C	char	Znak s kódovaním UTF-16
D	double	číselná hodnota s dvojitou presnosťou a plávajúcou desatinnou čiarkou
F	float	číselná hodnota s plávajúcou desatinnou čiarkou
I	int	celé číslo
J	long	celé číslo väčšieho rozsahu
L ClassName ;	referencia	inštancia triedy ClassName
S	short	znamienkové celé číslo krátkeho rozsahu
Z	boolean	pravda alebo nepravda
[	reference	jednorozmerné pole

Tabuľka A.3: Tabuľka reprezentácie dátových typov pre premenné.

Meno Indikátora	Hodnota	Interpretácia
ACC_PUBLIC	0x0001	Deklarovaná ako verejná; prístupná aj mimo balíka.
ACC_PRIVATE	0x0002	Deklarovaná ako privátna; použiteľná len vrámci triedy, v ktorej bola definovaná.
ACC_PROTECTED	0x0004	Deklarovaná ako protected; prístupná aj podtriedam.
ACC_STATIC	0x0008	Deklarovaná ako statická.
ACC_FINAL	0x0010	Deklarovaná ako final; žiadne ďalšie priradenia po inicializácií.
ACC_VOLATILE	0x0040	Deklarovaná ako volatile; nemôže byť uložená do medzipamäte.
ACC_TRANSIENT	0x0080	Deklarovaná ako transient; nieje čítaná ani modifikovaná objektovým manažérom.
ACC_SYNTHETIC	0x1000	Deklarovaná ako synthetic, nieje prítomná v zdrojovom kóde.
ACC_ENUM	0x4000	Deklarovaná ako prvok objektu enum

Tabuľka A.4: Tabuľka indikátorov prístupových práv a vlastností štruktúry *field_info*.

Meno Indikátora	Hodnota	Interpretácia
ACC_PUBLIC	0x0001	Deklarovaná ako verejná; prístupná aj mimo balíka.
ACC_PRIVATE	0x0002	Deklarovaná ako privátna; použiteľná len vrámci triedy, v ktorej bola definovaná.
ACC_PROTECTED	0x0004	Deklarovaná ako protected; prístupná aj podtriedam.
ACC_STATIC	0x0008	Deklarovaná ako statická.
ACC_FINAL	0x0010	Deklarovaná ako final; nemôže byť prepísaná.
ACC_SYNCHRONIZED	0x0020	Deklarovaná ako synchronized; pri volaní je zabalená za použitia monitora.
ACC_BRIDGE	0x0040	Bridge metóda; je generovaná prekladačom.
ACC_VARARGS	0x0080	Deklarovaná s dynamickým počtom argumentov.
ACC_NATIVE	0x0100	Deklarovaná ako natívna; implementovaná v inom jazyku ako Java.
ACC_ABSTRACT	0x0400	Deklarovaná ako abstraktná, nieje implementovaná.
ACC_STRICKT	0x0800	Deklarovaná ako stricktfp, výpočty s plávajúcou čiarkou sú FP - strict.
ACC_SYNTHETIC	0x1000	Deklarovaná ako synthetic, nieje prítomná v zdrojovom kóde.

Tabuľka A.5: Tabuľka indikátorov prístupových práv a vlastností štruktúry *method_info*.

Spúšťač	**Argumenty**
AT ENTRY	-
AT EXIT	-
AT LINE	number
AT READ	[type .] field [count | ALL]
AT READ	$var-or-idx [count | ALL]
AFTER READ	[type .] field [count | ALL]
AFTER READ	$var-or-idx [count | ALL]
AT WRITE	[type .] field [count | ALL]
AT WRITE	$var-or-idx [count | ALL]
AFTER WRITE	[type .] field [count | ALL]
AFTER WRITE	$var-or-idx [count | ALL]
AT INVOKE	[type .] method [(argtypes)] [count | ALL]
AFTER INVOKE	[type .] method [(argtypes)][count | ALL]
AT SYNCHRONIZE	[count | ALL]
AFTER SYNCHRONIZE	[count | ALL]
AT THROW	[count | ALL]

Tabuľka A.6: Tabuľka možných umiestnení spúšťačov jazyka ECA pravidiel pre nástroj *Byteman*. [Reda]

Dodatok B

Obsah CD

TODO - popis obsahu CD