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

TODO...

Kľúčové slová

TODO...

Poďakovanie

TODO...

Obsah

1 **Úvod** . . . 3
1.1 *Cieľ práce* . . . 4
1.2 *Členenie práce* . . . 4
2 **Bajtkód** . . . 5
2.1 *Štruktúra class súboru* . . . 5
2.1.1 Reprezentácia dátových typov . . . 7
2.1.2 Premenné tried a inštancií . . . 8
2.1.3 Metódy . . . 9
2.1.4 Atribúty . . . 10
2.2 *Inštrukcie JVM* . . . 10
2.2.1 Dátové typy . . . 11
2.2.2 Architektúra a inštrukčná sada . . . 11
3 **Classloadery** . . . 14
3.0.3 Dynamické načítavanie tried . . . 14
3.0.4 Znovunačítanie triedy . . . 14
4 **Byteman** . . . 16
4.1 *Byteman Agent* . . . 17
4.2 *Štruktúra jazyka pravidel* . . . 18
4.2.1 Udalosti . . . 18
4.2.2 Závislosti . . . 19
4.2.3 Výrazy . . . 19
4.2.4 Akcie . . . 20
4.2.5 Vstavané volania . . . 20
5 **Javasist** . . . 21
5.1 *Frozen classes* . . . 22
5.2 *ClassLoadery v Javassist* . . . 22
5.3 *Modifikácie* . . . 22
5.4 *Bytecode level API in Javassist* . . . 22
6 **Porovnanie** . . . 23
7 **Praktické ukážky** . . . 24
7.1 *Detekcia volania výnimiek* . . . 24
7.2 *Detekcia nesprávneho ošetrenia výnimiek* . . . 26

7.2.1 Štruktúra a funkčná logika aplikácie 27
7.2.2 Testovacie príklady 28
7.3 *Zlepšenie a zprehľadnenie produkčného kódu* 29
7.3.1 Štruktúra a funkčná logika aplikácie 29
7.3.2 Testovací príklad 30
7.4 *Optimalizácia neefektívnych častí kódu* 31
7.4.1 Štruktúra a funkčná logika aplikácie 32
Literatúra 33
A **Tabuľky** 34

Kapitola 1

Úvod

Java je dnes jedným z najpoužívanejších programovacích jazykov. Od syntakticky podobných programovacích 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 programu napísaných v programovacom jazyku Java prebieha v nasledujúcich fázach:

1. Preklad do medzikódu: Java compiler [1] preloží zdrojový kód do bajtkódu. V praxy to znamená, že každej triede, alebo rozhraniu je priradený súbor *class*, ktorý obsahuje inštrukcie popisujúce fungovanie danej triedy.

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

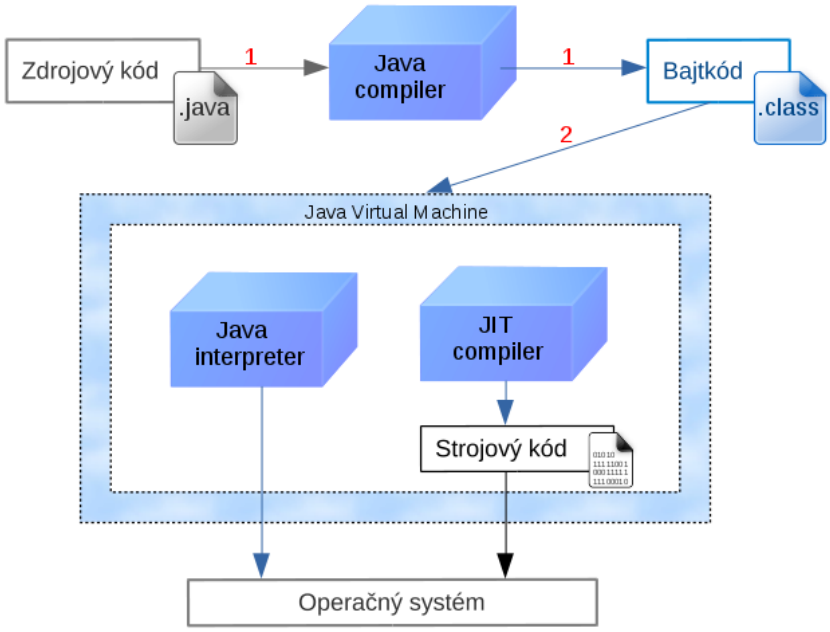
   - JIT prekladač (*Just In Time compiler*): Štandardne je z bajtkódu najskôr vygenerovaný strojový (*machine code*) konkrétneho zariadenia, ktorý je následne interpretovaný priamo vykonávaný procesorom.
   - Java interpreter: Ďalším spôsobom spracovania bajtkódu je využitie Java interpretru, ktorý bajtkódkód spracováva a sám interpretuje.

Výhodou prekladu do bajtkódu je jeho a prenositeľnosť. Samotný bajtkód je platformovo nezávislí. Program teda nieje nutné prispôsobovať jednotlivým operačným systémom, ktoré sa líšia len v implementácií JVM.

---

1. Najčastejšie využívaným Javacompilerom je *javac*, ktorý je súčasťou JDK (Java Development Kit).
2. Java Virtual Machine, špecifikácia je dostupná na `http://docs.oracle.com/javase/specs/jvms/se7/html`

Class súbory obsahujúce bajtkód je možné za behu programu modifikovať. Jednotlivé triedy a rozhrania 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 (*tracing*) programov.



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

1.1 Cieľ práce

TODO...

1.2 Členenie práce

TODO...

Kapitola 2

Bajtkód

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

Pri načítavaní *class* súbru JVM dostane 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 inštrukcí bajtkódu je reprezentovaná číselnou hodnotou nazývanou *opcode*. Zároveň má každá inštrukcia aj textovú podobu (*mnemonic*), ktorá je jej menom. V *class* súboorch sú inštrukcie uložené v numerickej podobe.

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

2.1 Štruktúra *class* súboru

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

Nasledujúci popis definuje sadu datových typov. Typy *u1*, *u2*, a *u4* reprezentujú neznamienkové jedno, dvoj, alebo štvorbajtové číslo. *ClassFile* je zobrazená ako pseudoštruktúra v notácií jazyka C. Obsah štruktúry je popísaný ako po sebe nasledujúce položky.

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 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 počtu záznamov v *constant_pool[]* plus jeden.

Úložisko záznamov *constant_pool[]* (v podobe poľa štruktúr) zahŕňa rôzne konštanty: mená tried a rozhraní, mená premenných a iné. Každý záznam *constant_pool[]* sa skladá zo značky (*tag*) a indexu (*name index*). Značka určuje typ záznamu. Tabuľka značiek je uvedená v prílohe A.1. Pomocou unikátneho indexu, je možné odkazovať sa na záznamy v d'alších častiach bajtkódu. Existuje niekoľko typov štruktúr [2] reprezentujúcich rôzne druhy záznamov. Napríklad štruktúra *CONSTANT_String_info* reprezentuje objekty typu *String* zatiaľ čo štruktúry *CONSTANT_Methodref_info* a *CONSTANT_InterfaceMethodref_info* reprezentujú metódy danej triedy, alebo rozhrania.

Hodnota *access_flags* popisuje oprávnenia prístupu k informáciam a vlastnosti tejto triedy, respektíve rozhrania 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 *constant_pool[]* odkazujúci na štruktúru typu *CONSTANT_Class_info* [3]. Reprezentuje triedu, respektíve rozhranie, definované týmo class súborom.

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

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

definovanej týmto *class* súborom. V prípade, že tento *class* súbor popisuje rozhranie, index odkazuje na triedu *Object*. Trieda *Obejct* má ako jediná hodnotu *super_class* nulovú.

Počet rozhraní, ktoré trieda implementuje vyjadruje položka *interface_count*, v prípade rozhrania je táto položka rovná počtu priamych nadrozhraní.

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

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

Štruktúry typu *field_info* sú združené v poli *fields[]*. Toto pole zahŕňa každú premennú danej triedy, respektíve rozhrania. Nezahŕňa zdedené atribúty. Podrobne sa štruktúrou *field_info* sa zaoberá kapitola 2.1.2.

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

Položka *methods[]* je pole štruktúr typu *method_info*. Každá štruktúra *method_info* popisuje metódu tejto triedy, respektíve rozhrania. Zahŕňa konštruktory, metódy triedy a metód inštancí. Neobsahuje však žiadne zdedené metódy. Štruktúru *method_info* popisuje kapitola 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. Pole *attributes[] class* súboru môže obsahovať maximálne jeden takýto atribút. Atribút *Depricated* môže byť použitý v prípade, že bola daná trieda nahradená (*depricated*). Pri volaní takejto triedy môže prekladač upozorniť užívateľa, že sa odkazuje na nahradenú triedu [4]. Vo všeobecnosti sa štruktúre *attribute_info* sa venuje kapitola 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
  - referenčné dátové typy

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

  - polia

- dátové typy metód

Primitívnym dátovým typom (*byte*, *integer*, ...) 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 premenná typu: *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*. Tvar reťazca dátového typu metódy je potom *(P*)R*. V prípade návratovej 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 datových 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 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 datový typ premennej. Reprezentáciou datových typov sa zaoberá kapitola 2.1.1.

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

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

2.1.3 Metódy

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

Štruktúra *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. Tabuľaka indikátorov *access_flags* štruktúry *method_info* sa nachádza v prílohe A.5.

Položky *name_index* a *descriptor_index* sú podobne ako u štruktúry *field_info* indexmi do *constant_pool*. Tieto indexy v *constant_pool* odkazujú na štruktúry popisujúce meno a datový typ metódy. Reprezentácia dátových typov je popísaná v kapitole 2.1.1.

Hodnotou položky *attributes_count* je počet atirbútov poľa *attributes[*.

Pole *attributes[]* zahŕňa dodatočné atribúty (položky) danej metódy. Každá položka poľa je reprezentovaná všeobecným predpisom *attributes_info*. Počet štruktúr v poli nieje obmedzený, každá položka však musí byť jednou zo štruktúr: *Code*, *Exceptions*, *Synthetic*,*Signature*, *Deprecated*, *RuntimeVisibleAnnotations*, *RuntimeInvisibleAnnotations*, *RuntimeVisibleParameterAnnotations*, *RuntimeInvisibleParameterAnnotations*, alebo *AnnotationDefault*. Atribút *Code* je jedným z najdôležitejších. Obshauje inštrukcie bajtkódu popisujúce fungovanie metódy. Okrem metód deklarovaných ako abstraktná,

alebo natívna musí každá metóda obsahovať práve jeden atribút *Code*. Atribút *Exceptions* zahŕňa indexy výnimiek, ktoré metóda vyhadzuje. Popisom formátu štruktúry *attributes_info* sa zaoberá kapitola 2.1.4.

2.1.4 Atribúty

Pojem atribút v tomto texte vyjadruje 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. Tento proces sa nazýca kontrola formátu (*format checking*). Prvé štyri bajty musia obsahovať tzv. magickú konštantu *magic*. Všetky rozoznané atribúty musia mať správnu dĺžku

2.2 Inštrukcie JVM

Po načítaní *class* súboru JVM sa JVM nasjkôr uistí, že je tento súbor v správnom formáte popísanom v kapitole 2.1. Štvorbajtová položka *attribute_length* je rovná hodnote vyjadrujúcej dĺžku následných informácií uložených v *info[attribute_length]*. Informácie sa líšia na základe odlišnej funkcionality a využitá jednotlivých atribútov. *Class* súbor nesmie byť skrátený ani obsahovať nadbytočné bajty, takisto 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* reprezentujúca atribút *Code* musí spĺňať obmedzenia definované JVM. Tieto obmedzenie 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 požiadavnky 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 ištrukciami *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.

2.2.2 Architektúra a inštrukčná sada

Architektúra JVM je založená na datovej štruktúre zásobník [6], ktorej základnými operáciami sú *push* - vloženie prvku do zásobníka a *pop* - výber

5. Ďalšie príklady obmedzení a podorbný popis verifikácie *class* súborov je dostupný v špecifikácií JVM [LYBB13]
6. Dátová štruktúra zásobník (*stack*) funguje na princípe FIFO (*first in first out*), kde posledný vložený prvok je prvým vybraným.

prvku z vrcholu zásobníka. JVM nemá registre na ukladanie hodnôt, preto musia byť pred použitím všetky uložené na zásobník.

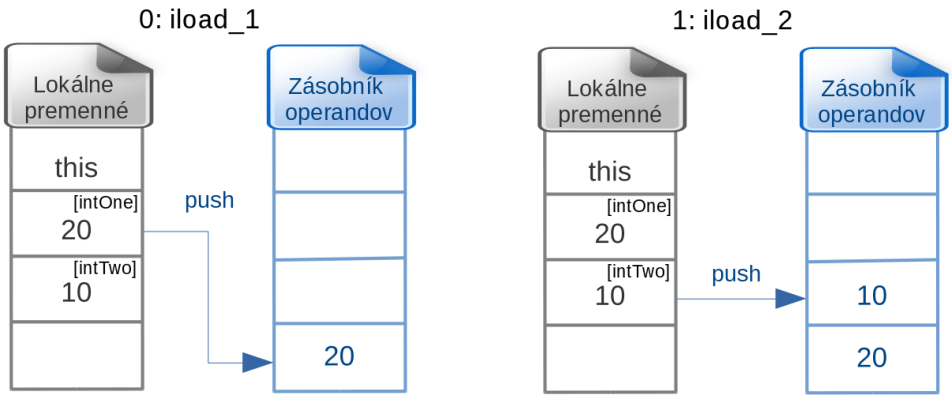
Na nasledujúcom jednoduchom príklade sú popísané základné inštrukcie bajtkódu pre prácu s premennými a konštantami.

Metóda *greatherThen* 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*.



Obr. 2.1: Znázornenie funkcionality inštrukcií *iload_1* a *iload_2*.

Vo všeobecnosti môžeme túto inštrukciu chápať ako *xload* s predponou extitx označujúcou ľubovoľný primitívny datový 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 lokalnej 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 menšia 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 vrátene ich parametrov možno nájsť v špecifikácií JVM [LYBB13].

Kapitola 3

Classloadery

Class loader je objekt zodpovedný za načítavanie tried. Trieda *ClassLoader* je abstraktná. Pomocou mena *class* súboru by mal *class loader* nájsť a generovať obsah definujúci danú triedu. Každá trieda obsahuje referenciu na *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 *class loaderov* je možné za behu programu dynamicky načítať d'alšie triedy, prípadne načítať nové inštancie pôvodných tried.

Pri štandardnom načítaní triedy niektorá z implementácií *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 nuspeje pokúsi sa načítať triedu pomocou vlastného *class loaderu*

3.0.3 Dynamické načítavanie tried

K načítaniu novej triedy za behu programu je potrebný *class loader*. Získať ho je možné pomocou príkazu *MyClass.class.getClassLoader();*. Novú triedu reprezentovanú súborom *class* následne vráti metóda *class loaderu*: *loadClass(class)*.

3.0.4 Znovunačítanie triedy

Dynamické znovunačítanie triedy je komplkovanejšie. Vstavané implementácie triedy *ClassLoader* vždy kontrolujú, či trieda už nebola do JVM načítaná. Preto nieje možné žiadnu triedu načítať dvakrát pomocou vstavaných *class loaderov*. Je nutné navrhnúť vlastnú implementáciu.

-

! PRÍKLAD VLASTNÉHO CLASSLOADERA - TODO ... !

-

Ďalšou komplikáciou je trieda *ClassLoader.resolve()*, ktorá zabezbečuje linkovanie. Táto trieda je *final*, z čoho vyplíva, že ju nieje možné prepísať, nepovolí však žiadnemu *class loaderu* linkovať dva-krát tú istú triedu. Preto je nutné pri kaďom ďalšom znovunačítaní triedy vytvoriť novú inštanciu *class loaderu*.

-

! PRÍKLAD POUŽITIA CLASSLOADERA - TODO ... !

Kapitola 4

Byteman

Byteman je nástroj manipulujúci s bajtkódom určný na zásah do bajtkódu 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 pôvodnej aplikácie.

Byteman modifikuje bajtkód aplikácie za behu programu. Preto môže zmeniť Java kód, popisujúci časť treid JVM ako napríklad *String*, *Thread* a podobne. Vďaka tejto funkcionalite je taktiež možné napríklad trasovanie správania sa JVM.

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

Primárne bol *Byteman* určený na podoru testovania multivláknových a multi-JVM aplikácií za použitia techniky nazývanej *fault injection* [2]. Zahŕňa preto funkcionalitu, ktorá bola navrhnutá na riešenie problémov súvisiacich s týmto typom testovania. *Byteman* poskytuje podporu pre automatizáciu v štyroch hlavných oblastiach:

- trasovanie špecifických väzieb kódu a zobrazovanie stavu aplikácie, prípadne JVM,
- narúšanie normálneho priebehu zmenou stavov, volanie nenaplánovaných metód, vynucovanie návratových volaní, prípadne vyhadzovanie neočakávaných výnimiek,
- organizácia časovania aktivít vykonaných nezávislými vláknami aplikácie,

---

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

- monitorovanie a zhromažďovanie štatistík, sumarizujúcich aplikáciu a operácie JVM.

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

Najjednoduchším použitím *Bytemana* je vkladanie kódu, ktorý trasuje správanie sa aplikácie. Táto metóda môže byť využitá 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 nákaldom, ktoré často rastú pri ladení, alebo trasovaní proguktu [RH].

4.1 Byteman Agent

Aby mohol *Byteman* manipulovať s programom, musí na ňom bežať *Byteman Agent*, ktorý konfiguruje JVM pre prácu s pravidlami jeho jazyka.

Pri inštalácií agenta s prekladom programu je riešením použitie argumentu príkazu *java -javaagent*, ktorý zadáva cestu k *JAR* súboru popisujúcemu pravidlá jazyka. Agentovi je možné pomocou argumentov nakonfigurovať dve základné 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 spusteného programu je možné následne pomocou skriptu *bmsubmit.sh* pridávať a odoberať ľubovoľné pravidlá.

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

Agenta je možné inštalovať taktiež do už bežaicich aplikácií [3] bez nutnosti ich opätovného spustenia. Slúži na to skipt *bminstall.sh*. *Byteman* je následne možné využiť ako nástroj na kontrolu správania sa programu [Reda].

3. typicky ide o dlho bežiace aplikácie ako napríklad aplikačný server JBoss

4.2 Štruktúra jazyka pravidel

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
```

Každá z definícií, ktorú je možné v pravidle použiť patrí do niektorej z kategórií: Udalosti, Závislosti, Výrazy, Podmienky, Akcie, Vstavané volania. Tieto kategórie a jednotlivé definície popisujú nasledujúce podkapitoy. Definície musia byť zároveň zadané v správnom poradí pričom prvou je vždy *RULE* a poslednou *ENDRULE*.

4.2.1 Udalosti

Udalosti pravidiel (*Rule Events*) identifikujú umiestneneie pravidla v cieľovej metóde, ktorá sa nachádza v cieľovej triede.

Za kľučvým slovom *RULE* musí nasledovať meno pravidla, ktoré je ľubovoľným textovým reťazcom, pričom musí obsahovať medzeru. Kvôli rozlišovaniu jednotlivých pravidiel by mali byť tieto mená unikátne.

Rovnako za kľúčovými slovami *CLASS* a *METHOD* sa nachádza meno triedy a metódy do ktorej bude pravidlo načítané. Meno triedy môže byť špecifikované aj bez cesty ku balíku, v ktorom sa nachádza. V takomto prípade, Byteman spracuváva každú triedu s týmto menom, ktorá je do JVM načítaná. Definíciu *CLASS* je možné nahradiť kľúčovým slovom *INTERFACE*, ktoré rovňakým spôsobom ako *CLASS* popisuje rozhranie. Doplnením znaku ňa začiatok mena triedy, respektíve rozrania je možné zabezpečiť dedičnosť - prenos pravidiel na potomkov. Metódu je možné okrem samotného názvu špecifikovať aj jej návratovým typom, prípadne argumentami. Tieto bližšie špecifikácie niesu povinné, preto je možné načítať pravidlo do viacerých preťažených metód zároveň.

Po nájdení metódy, respektíve metód, ktoré vyplívajú s definícií CLASS, METHOD, prípadne INTERFACE je do každej z nich vložený takzvaný spúšťač *trigger point*. Tento spúšťač presne identifikuje miesto v metóde,

kde bude bajtkód injektovaný. Pomocou špecifikácie umiestnenia je možné zvoliť rôzne umiestenie tohto spúšťača. Defaultne je zvolené umiestnenie *AT ENTRY*, čo znamená, že spúšťač bude vložený na začiatok - pred prvú inštrukciu [4] danej metódy. Ďalšímie možnosťami umiestnenia 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 je uvedená v: TODO....

4.2.2 Závislosti

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

Závislosť

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

Vytvára premennú thisClass, korej bude automaticky odvodený typ a priradzjue 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 napríklad: referencie na predošlé závislosti, referencie na lokálne premenné v okolí spúšťača, vstavané operátory ako napríkald: *$!*, *$*, *$@*, ... a mnohé iné.
- Výrazy zložené z iných výrazov pomocou štandardných operátorov jazyka Java. operácie JVM.

Tabuľka zákaldných vsatavaných operátorov je uvedená v TODO...

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

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

4.2.4 Akcie

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

Akcie

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

Tento príklad zobrazuje zjednodušené pravidlo, ktrého akciou je výpsi názvu metódy na štandardný výstup. Narozdiel od ECA pravidiel nástroja *Byteman*, Javassist používa na reprezentáciu *class* súboru triedu *Javassist.CtClass*.

4.2.5 Vstavané volania

Vstavané volania (*Built-In Calls*) TODO... Rule Helpers?

Kapitola 5

Javasist

Ďalším nástrojom určeným na manipuláciu s bajtkódom je *Javassist*. Tento nástroj využíva na manipuláciu s bajtkódom odlišný prístup ako *Byteman*. Narozdiel od ECA pravidie, *Javassist* používa na reprezentáciu *class* súboru treidu *Javassist.CtClass*. *Class* súbor 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 uvedenoum 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* príslušný danému kľúču [Shi].

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

Definícia novej triedy

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

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

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

1. Možnosti modofikácie triedy pomocou *CtClass* popisuje kapitola TODO...

disk. *Javassist* taktiež poskytuje metódu *toBytecide()*, ktorá vráti modifikované inštrukcie bajtkódu do poľa typu *byte* [Shi].

5.1 Frozen classes

TODO...

5.2 ClassLoadery v Javassist

TODO...

5.3 Modifikácie

TODO...

5.4 Bytecode level API in Javassist

TODO...

Kapitola 6

Porovnanie

TODO...

Kapitola 7

Praktické ukážky

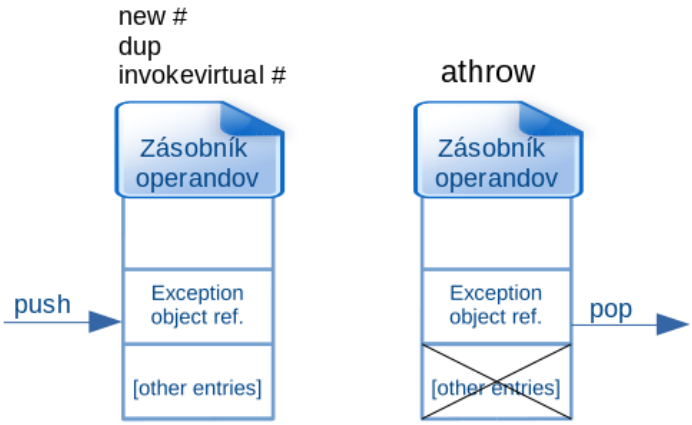
Dôležitou súčasťou bakalárskej práce sú praktické ukážky. Každý z príkladov je navrhnutý pre jeden z nástrojov *Byteman*, respektíve *Javassist*. Konkrétne ukazujú základné možnosti jeho využitia pre vývoj a trasovanie programov Jave. Ukážky pokrývajú 4 oblasti:

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

Príklady obsahujú 2 ukážky z oblasti detekcie volania a ošetrovania výnimiek a 2 ukazky, zamerané na optimalizáciu a zlepšenie kódu. V praxi sa nástroje ako *Byteman* a *Javssist* využívajú najmä v aplikačných serveroch prípadné iných projektoch, ktorých opätovná kompilácia by bola príliš časovo a technicky náročná. V ukážkach budeme z praktických dôvodov využívať na testovanie funkcionality menšie prgramové celky a demonštračné programy.

7.1 Detekcia volania výnimiek

Prvou z ukážok je detekcia volani výnimiek spusteného programu. Ide o detekciu všetkých výniemiek, ktoré sú v požadaovanej 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

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 výniemiek 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 preínačom *agent listener*, ktorý je možné pridať aj za behu do dlhodobo spustenej aplikácie.

skript generovať. Program následne monitoruje každú metódu zadanej triedy, vrátane konštruktora. V prípade potreby je možné generovať a načítavať skritpy pre viacero tried zároveň. V tomto prípade je nutné zadať skriptu *loadScripts* 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 používam ako demonštračný príklad program *fileChooser* [2]. V reálnom prostredí by bolo možné detekciu volania výnimiek využiť napríklad v aplikačných serveroch, kde nieje 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.

Detekcia nesprávneho ošetrenia výnimiek je vhodným demonštaračným príkladom pre ukážku funkcionality knižnice *Javassist*. 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 prázdného catch bloku uloží informácie o jeho polohe do logu a na záver vypíše získané údaje.Výstup môže vyzerať napríklad nasledovne:

Výstup aplikácie po kontorle 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
```

Podorbný popis spustenia aplikácie a načítania skriptov sa nachádza v README súbore programu.

2. Tento demonštračný príklad je súčasťou balíka Java Development Kit Demos and Samples.

```
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.

7.2.1 Štruktúra a funkčná logika aplikácie

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

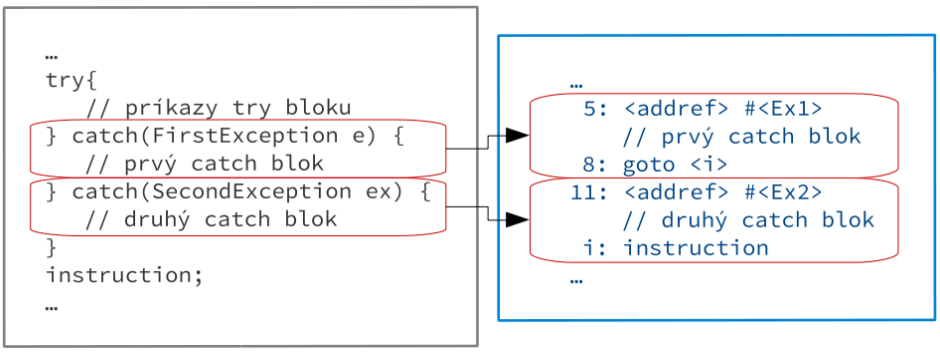
Trieda *Tracer* slúži na spustenie samotnej aplikácie pre ľubovoľnú triedu <C>. Obsahuje metódu *traceCatchBlocks(Class classToTrace)*, ktorej argumentom je objekt reprezentujúcie <C>. Táto metóda prevedie triedu na objekt typu *CtClass* a následne z neho získa všetky metódy a konštruktory v podobe polí tried *CtMethod* a *CtConstrucor*. Následne za pomoci triedy *CatchBlockTracer* získa informácie o všetkých prázdnych *catch* blokoch 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)*. Táto metóda má argument typu CtBehaviour, ktorého potomkami sú práve triedy *CtMethod* a *CtConstructor*. Uchováva si informácie o pozícií cathc blokov v podobe mapy, ktorej kľúčom je index prvej inštrukcie cathc bloku v bajtkóde a hodnotou je pozícia catch bloku v zdrojovm súbore. Túto mapu získava za pomoci tried CBDetector a CBIndicesHandler. Po získaní údajov o polohe všetkých catch blokov v danej metóde je každý z nich skontrolovaný metódou *isEmpty*. V prípade preukázania nesprávne ošetrenej výnimky je sú informácie o catch bloku uložené a logované.

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

Na začiatku je na zásobník vložená referencia na objekt výnimky. Nasleduje telo *catch* bloku. Inštrukcia po poslednom *catch* bloku má index <i>.V prípde, že aktuálny *catch* blok nieje posledným v rade je za jeho telo umiestnená inštrukcia *goto*, ktorá odkazuje na index inštrukcie <i>. V prípade posledného *catch* nasleduje index <i> ihenď po jeho tele. Metóda *isEmpty* preto kontroluje či telo catch bloku obsahuje iba inštrukciu skoku na index <i>, prípadne či je tele catch bloku úple prázdne.

Poslednými triedami na spodnej časti hierarchie v abstarkcii získavania polohy catch blokov sú *CBDetector* a *CBIndicesHandler*. Slúžia na získa-



Obr. 7.2: Grafické znázornenie reprezentácie *catch* bloku v bajtkóde

nie mapy indexov catch blokov pre zadanú metódu, prípadne konštruktor. Dôležitým nástorjom na samotné vyhľadávanie catch blokov je metóda *CtBehavior.instrument(ExprEditor)*. Argumentom tejto metódy je práve objekt triedy *CBIndicesHandler*, keďže *CBIndicesHandler* je potomkom triedy *ExprEditor*.

7.2.2 Testovacie príklady

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

Balík *example.simple* obsahuje dve testovacie triedy, ktoré som vytvoril pre overenie samotnej funkčnosti aplikácie v balíku *application*. Po spustení aplikácia upozorní na nesprávne ošetrene'*catch* bloky v triede *PersonFactory*.

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

Triedy v testovacích balíkoch je možné ľubovoľne modifikovať pre jednoduché toestovanie funkcionality aplikácie. V prípade potreby testovania aplikácie na novej triede je nutné zavolať metódu *application.Tracer.traceCatchBlocks* s argumentom špecifikujúcim danú triedu.

7.3 Zlepšenie a zprehľadnenie 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. Triada <C1> obsahuje atribúty, ku ktorím však neexistujú prístupové *get* a *set* metódy, atribúty sú teda používané priamo. Trieda <C2> zapisuje, respektíve číta obsah týchto atribútov. Takáto implementácia porušuje základný princíp zapúzdrenia, ktorý by mal byť dodržaný za každých okolností. Projetk 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. Tiež by nemali nijako ovplyvniť vnútornú logiku modifikovaných tried. Okrem informácií zobrazených na štandardnom výstupe je zmeny možné pozorovať aj v *class* súboroch daných tried [3].

7.3.1 Štruktúra a funkčná logika aplikácie

Aplikácia sa nazýva *code-improvement* je Java projektom typu *Maven*. Demnonštruje využitie nástroja *Javassist*. Projekt je tvorený dvoma balíkmi: *application* a *example*. Balík *application* obsahuje progaram nahradzujúci priame voalania atribútov prístupovými metódami, ktorého základná funkcionalita bola načrtnutá v predchádzajúci odsekoch. Balík *example* je jednoduchým testovacím príkladom pre kontorlu funkčnosti tried balíka *application*.

TODO DIAGRAM...

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

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

Program postupne iteruje cez všetky atribúty triedy <C1>. Pre každý atribút vytvoria triedy *AccGenerator* a následne *AccCreator* prístupové *get* a

3. Class súbory je možné zobraziť v čitateľnej podoba nepríklad prostrerdníctvom nástroja *javap* príkazom *[javap -c <path>]*.

set metódy. Akonáhle sú tieto metódy úspešne načítané do bajtkódu tiredy <C1>, vykoná metóda *replaceFieldAccess()* náhradu priameho volania atribútu prístupovými metódami. Využíva k tom najmä nástroj *CodeConverter* knižnice *Jvassist*.

TODO OBRAZOK...

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

TODO MECHANIZMUS...

7.3.2 Testovací príklad

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

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

V prípade úspešnej modifikácie by mal výstup vyzerať 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 modifikácie neprebehla, napríklad v prípade opätovného spustenia na rovnakých triedach, vypýše program hlášku, ktorá upozorňuje užívateľa, že triedy, ktoré zadal už boli v minulosti upravené.

Program je možné aplikovať na ľubovoľné triedy definované v konštruktore triedy *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.

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

TODO Obrazok pred a po...

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

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

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.

Kľúčovou triedou je teda *ArithmeticOptimizer* a jej metóda *optimizeClass(String classNameToOpt)*. Po inicializácií a jej volaní program iteruje cez všetky metódy triedy definovanej 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. Táto 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 nájdenéo nadbytočného páru *dstore*, *dload* tieto inštrukcie odstráni.

Identifikácie nadbytočných inštrukcí čítania a zápisuprebieha nasledovne: TODO...

Ako nástroj na identifikáciu typu inštrukcie slúži programu trieda *instructionVerifier*. Obsahuje metódy určujúce inštrukcie *dstore* a *dload*. Pomocou týchto metód je možné takisto získať hodnotu argumentu spomínanýhc 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(classNameToOpt)*.

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

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/.

[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.

[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 nasledujúcich tabuliek 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 datových 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*.