VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZPĚTNÝ PŘEKLADAČ BAJTKÓDU JAZYKA JAVA

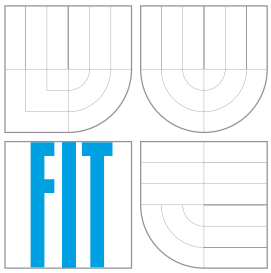
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAROMÍR HŘIBAL

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZPĚTNÝ PŘEKLADAČ BAJTKÓDU JAZYKA JAVA

JAVA BYTECODE DISASSEMBLER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAROMÍR HŘIBAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZBYNĚK KŘIVKA, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

První část této bakalářské práce se zabývá základními principy virtuálního stroje jazyka Java a detailněji se věnuje jeho instrukční sadě a formátu .class souborů, dohromady známých jako bajtkód. Následující část prezentuje rešerši existujících nástrojů pro manipulaci s bajtkódem. Poslední část popisuje vytvoření pluginu do vývojového prostředí Eclipse, který realizuje uživatelsky přívětivé zobrazení zadaného bajtkódu spolu s původním zdrojovým kódem, ze kterého byl bajtkód generován. Nástroj předpokládá dostupnost těchto zdrojových kódů, takže neplní roli dekompilátoru.

Abstract

The first part of this thesis studies the fundamental principles of Java Virtual Machine and presents in depth look at its instruction set and .class file format, both together well known as bytecode. The next part presents an overview of the existing tools for bytecode manipulation. The last part of this work describes the development of a new plugin for Eclipse IDE. This plugin allows the user to get more comfortable view of the given bytecode and to see the original source code from which the bytecode was generated. The plugin requires the source code to be accessible so it is not a typical decompilation tool.

Klíčová slova

Java, Zpětný překlad, Java bajtkód, Java virtuální stroj

Keywords

Java, Disassembler, Java bytecode, Java Virtual Machine

Citace

Jaromír Hřibal: Zpětný překladač bajtkódu jazyka Java, bakalářská práce, Brno, FIT VUT v Brně, 2015

Zpětný překladač bajtkódu jazyka Java

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D.. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jaromír Hřibal
2. srpna 2015

Poděkování

Chtěl bych tímto poděkovat svému vedoucímu panu Ing. Zbyňku Křivkovi, Ph.D. za cenné rady, trpělivost, pozitivní přístup a věnovaný čas.

Obsah

Kapitola 1

Úvod

Jedním z hlavních důvodů rozvoje IT sektoru, ke kterému v posledních dvou dekádách došlo, je vznik mnoha moderních programovacích jazyků, které umožňují snadněji a pohodlněji vytvářet počítačový software.

Mezi tyto jazyky se řadí také jazyk Java. Vývoj tohoto objektově orientovaného, interpretovaného programovacího jazyka, jehož původní název byl Oak, započal v roce 1991 James Gosling. Veřejnosti byl jazyk představen v roce 1996 společností Sun Microsystem, kdy byla uvolněna jeho první verze Java 1.0. Jazyk byl původně určen pro programování vestavěných systémů a důvodem vzniku byla tehdejší nespokojenost s možnostmi jazyka C++ pro jejich programování (přenositelnost, paměťová náročnost, velký počet chyb plynoucích ze složitosti, nutnost vlastní správy paměti).

V současnosti je Java jeden z nejpoužívanějších programovacích jazyků, jehož popularita stále sílí a programátoři, kteří tento jazyk ovládají, patří mezi nejvyhledávanější. Její použití je od stolních počítačů po mobilní zařízení a vestavěné systémy obecně.

Hlavním důvodem úspěchu Javy je fakt, že programy v ní vytvořené jsou přenositelné a tedy mohou běžet na různých platformách, bez nutnosti znovu překládat zdrojový kód (*"Write Once, Run Anywhere"*). Umožňuje tedy binární kompatibilitu, na rozdíl např. od jazyka C, kde lze dosáhnout přenositelnosti pouze na úrovni zdrojových kódů. Mezi další důležité vlastnosti jazyka patří jednoduchost, syntaxe vycházející z dobře známých jazyků jako C a C++ a automatická správa paměti.

Protože programy napsané v jazyce Java jsou určeny pro různé platformy, není mezikód vzniklý překladem zdrojových kódu určen přímo pro procesor konkrétního zařízení, ale pro speciální program, který implementuje specifikaci Java Virtual Machine [15] *(dále jen JVM)* a dokáže tento mezikód vykonávat. Aby na konkrétním zařízení mohl být spuštěn program v jazyce Java, musí pro něho tedy existovat implementace specifikace JVM. První specifikace JVM vycházela z virtuálního stroje navrženého Jamesem Goslingem pro již zmíněný jazyk Oak, ze kterého se Java později vyvinula.

Java Virtual Machine umožňuje interpretovat mezikód, který se nazývá Java bajtkód, dále jen bajtkód (*z angl. bytecode*), a který je výstupem překladu zdrojových kódů v Javě. Výhodou využití virtuálního stroje pro interpretaci bajtkódu, který sám o sobě není pevně svázán s jazykem Java ani s konkrétní platformou, je, že umožňuje vytvořit překladače do bajtkódu i pro jiné jazyky, než je Java. Mezi takové jazyky se řadí např. Scala.

Vzhledem k tomu, kam až se Java od svého vzniku posunula, lze s nadsázkou přemýšlet o tom, že Java změnila svět.

1.1 Motivace

Přestože pro psaní programů bežících na JVM postačí zvládnutí syntaxe a sémantiky nějakého jazyka, pro který existuje překladač do bajtkódu, tak znalost bajtkódu a interního fungování JVM přináší řadu výhod.

Výhody znalosti fungování JVM

- porozumění smyslu jednotlivých instrukcí a jak mohou být překládaný různé jazykové konstrukce
- možnost odhalení chyby v překladači na základě vygenerovaného bajtkódu
- možnost přímé modifikace vygenerovaných `.class` souborů
- možnost tvorby softwarových komponent pro generování `.class` souborů
- možnost portace JVM na další platformy
- možnost vytvoření vlastní platformy s využitím JVM např. pro tvorbu realtime webových aplikací (vytvoření dynamicky typovaného jazyka zaměřeného na konkrétní doménu, zabudování webového serveru do JVM)

Motivací pro praktickou část této práce, tedy tvorbu vlastního pluginu do Eclipse IDE, je poskytnout alternativu k dvěma již existujícím pluginům, které jsou zmíněny v kapitole zabývající se rešerší existujících nástrojů a dále také navržení tohoto pluginu tak, aby poskytoval výstup složený ze všech `.class` souborů, které jsou generovaný z jednoho `.java` souboru, což ostatní pluginy nedovedou. Výhodou je celkový přehled o tom, co je z daného `.java` souboru generováno a zkušený programátor může např. identifikovat nesrovnalosti, které mohou znamenat i použití vadného kompilátoru.

1.2 Náplň práce

Tato práce se zabývá základním popisem JVM vycházejícím ze specifikace, popisem formátu výstupních jednotek překladačů pro JVM a popisem jednotlivých typů instrukcí bajtkódu.

V průběhu následující kapitoly je u některých pasáží uvedena poznámka o tom, jak konkrétní problematiku řeší referenční implementace JVM specifikace — 32 bitová verze HotSpot na platformě x86.

Další kapitola popisuje některé existující nástroje pro manipulaci s bajtkódem.

Součástí práce je také implementace pluginu do vývojového prostředí Eclipse. Tento plugin umožňuje zobrazit bajtkód generovaný z `.java` souborů Java projektu a jeho implementace je detailně popsána v pozdější kapitole.

Kapitola 2

Java Virtual Machine

Java Virtual Machine (*dále jen JVM*) je obecný název pro software, který odpovídá specifikaci Java Virtual Machine. Na JVM lze nahlížet jako na program, který dokáže korektně zpracovávat `.class` soubory, interpretovat instrukční sadu které se říká Java bajtkód (*dále jen bajtkód*) a manipulovat s různými paměťovými oblastmi.

2.1 Paměťové oblasti

JVM používá několik paměťových oblastí, které se popsány dále.

Java halda

Java halda (*z angl. Java heap*) je oblast společná pro všechna vlákna a slouží jako místo, odkud JVM alokuje potřebnou paměť pro vytváření polí a instancí tříd. Tato paměť je v automatické správě JVM a stará se o ni garbage collector.

HotSpot používá generační haldu (*z angl. generation heap*), kdy je halda rozdělena na 2 částí a každá slouží pro ukládání objektů s různou délkou života. Tyto části se nazývají Young Generation a Old Generation [16].

Oblast Young Generation je dále rozdělena na tři části - prostor pro nové objekty (*z angl. Eden*) a dva prostory (*z angl. Survivor Spaces*), kde jeden je vždy prázdný (*TO* prostor) a druhý obsahuje objekty, které přežily alespoň jeden cyklus garbage collectoru (*FROM* prostor).

Při cyklu garbage collectoru se vychází z množiny tzv. kořenových objektů (*z angl. Root Objects*), které pocházejí např. ze statických datových členů tříd a z objektů vyskytujících se na zásobnících vláken. Objekty, na které se lze z těchto kořenových objektů dostat skrze řetězec referencí, jsou z části pro nové objekty a z *FROM* prostoru přesunuty do *TO* prostoru.

Při dalším cyklu se proces opakuje, akorát si mezi sebou roli prohodí prostory *TO* a *FROM*, tedy *FROM* prostor se po skončení cyklu stává *TO* prostorem a opačně.

Při každém přesouvání objektů z *FROM* prostoru do *TO* prostoru mohou být některé objekty, které už přežily dostatečný počet cyklů, přesunuty do Old Generation.

Jednotlivá vlákna mohou mít z oblasti pro nové objekty vyhrazenu část prostoru, která je označována jako TLAB (*z angl. Thread Local Allocation Buffer*) [13] a ze které si dané vlákno alokuje prostor pro nové objekty. Díky tomu není nutné synchronizovat vytváření nových objektů z různých vláken a efektivita se zvýší.

Method Area

Tato oblast slouží pro ukládání různých struktur jako např. Run-Time Constant Pool nebo bajtkód metod.

HotSpot objekty reprezentující metody, Run-Time Constant Pool a další interní struktury vytváří v oblasti nazývané Permanent Generation [16]. Tato oblast slouží k ukládání veškerých objektů, které povětšinou existují po celou dobu běhu programu. Např. Java objekt reprezentující výjimku `java.lang.OutOfMemoryError` je také alokován v této oblasti.

Programový čítač

Každé JVM vlákno vykonávající kód metody má svůj programový čítač (*z angl. program counter)*, který obsahuje adresu právě vykonávané instrukce. Pokud je metoda nativní, obsah programového čítače není definován.

HotSpot implementuje dva typy interpretů a to Cpp interpret (*z angl. Cpp Interpreter*) a Šablonový interpret (*z angl. Template Interpreter*). Cpp interpret si adresu právě vykonávané instrukce udržuje ve členu `BytecodeInterpreter._bcp` . Šablonový interpret si udržuje adresu právě vykonávané instrukce v registru `esi`. Šablonový intepret je výchozím interpretem a může být až 10x rychlejší [14].

Zásobník vlákna

Tento zásobník je vytvořen pro každé JVM vlákno a slouží pro ukládání aktivačních rámců při volání metod.

HotSpot nevytváří žádnou dodatečnou strukturu, ale používá přímo nativní zásobník vlákna.

Run-Time Constant Pool

Tato struktura je uložena v oblasti Method Area a reprezentuje část Class file formátu, kterou je `constant_pool` tabulka. Struktura vzniká ve chvíli, kdy JVM načítá `.class` soubor.

HotSpot reprezentuje Run-Time Constant Pool třídami `constantPoolOopDesc` a `constantPoolCacheOopDesc`.

Třida `constantPoolOopDesc` obsahuje data jako konstanty (run-time reprezentace záznamu `CONSTANT_Integer_info` apod.) a `constantPoolCacheOopDesc` obsahuje data jako reference na instance třídy `methodOopDesc`, které reprezentují metody (záznam `CONSTANT_MethodRef_info` po dynamickém linkování).

Když se např. provádí instrukce `invokevirtual`, tak pokud patřičný záznam v `constantPoolCacheOopDesc` ještě není slinkován, tak se tak učiní a další provádění instrukce, která se bude odkazovat na stejný záznam, už nevyžaduje linkování.

Rámce

Rámce (*z angl. Frames*) hrají důležitou roli při vykonávání bajtkódu. Vždy je v jednom vlákně aktivní pouze jeden rámec a to rámec metody, která se právě vykonává. Hlavní část rámce tvoří zásobník operandů a pole lokálních proměnných. Po skončení metody dochází k odstranění rámce.

HotSpot ukládá aktivační rámce přímo na nativním zásobníku vlákna. Struktura rámce se liší podle toho, jestli je použit Cpp interpret nebo Šablonový intepret. Např. velikost zásobníku operandů u Šablonového interpretu se dynamicky mění, protože jeho vrchol je uložen v registru `esp` a vkládání hodnot se realizuje nativní instrukcí `push`. Naproti tomu u Cpp interpretu je velikost zásobníku fixně dána a manipuluje se s ním pomocí třídy `BytecodeInterpreter`. Oba typy rámců obsahují kromě oblasti pro lokální proměnné a operandy také hlavně oblast pro ukládání záznamů o zamčených objektech. Při volání interpretované metody se část rámce volajícího stane součástí nově vytvořeného rámce volaného. Toto je možné, protože lokální proměnné jsou uloženy na začátku rámce a zásobník operandů na konci. Protože parametry metody jsou očekávány v lokálních proměnných, může se konec zásobníku operandů volajícího stát začátkem lokálních proměnných volaného. Díky tomu není nutné hodnoty kopírovat a vytvoření nového rámce je rychlejší.

Aktivní rámec vlákna je možné identifikovat ze členu `JavaThread._anchor`.

2.2 Překlad zdrojových souborů

Program v jazyce Java se sestává z jednoho či více souborů s příponou `.java`, které obsahují definice tříd. Java nemá globální prostor, jako např. jazyk C a veškerý kód musí být rozdělen do tříd. Před spuštěním programu musí být zdrojové kódy nejprve přeloženy Java kompilátorem. Výstupem kompilátoru pro jeden zdrojový soubor je typicky jeden či více souborů s příponou `.class`, odpovídajících Class file formátu. Každý `.class` soubor odpovídá definici jedné Java třídy. Běhové prostředí Javy už pracuje pouze s `.class` soubory.

Výhody Class file formátu

Překlad zdrojových souborů do jednotek odpovídajících Class file formátu má řadu výhod.

- **rychlejší interpretace** - není nutné zdrojové kódy převádět do mezikódu při každém prvním použití po spuštění programu, jako to standardně dělá např. jazyk PHP se svými skripty
- **platformová nezávislost** - protože Class file formát není nijak svázán s konkrétní platformou, program je plně přenositelný a jedinou nutností je, aby pro danou platformu existovala implementace virtuálního stroje a případně tříd, které mají nativní metody
- **nezávislost na programovacím jazyce** - jakýkoliv programovací jazyk, pro který bude existovat překladač generující Class file formát, může benefitovat z vlastností JVM a používat ji jako své běhové prostředí. JVM pracuje pouze s Class file formátem a nepředpokládá použití konkrétního jazyka.

Ukázková definice třídy v jazyce Java

Následující ukázka 2.1 zobrazuje známý *Hello World* v jazyce Java.

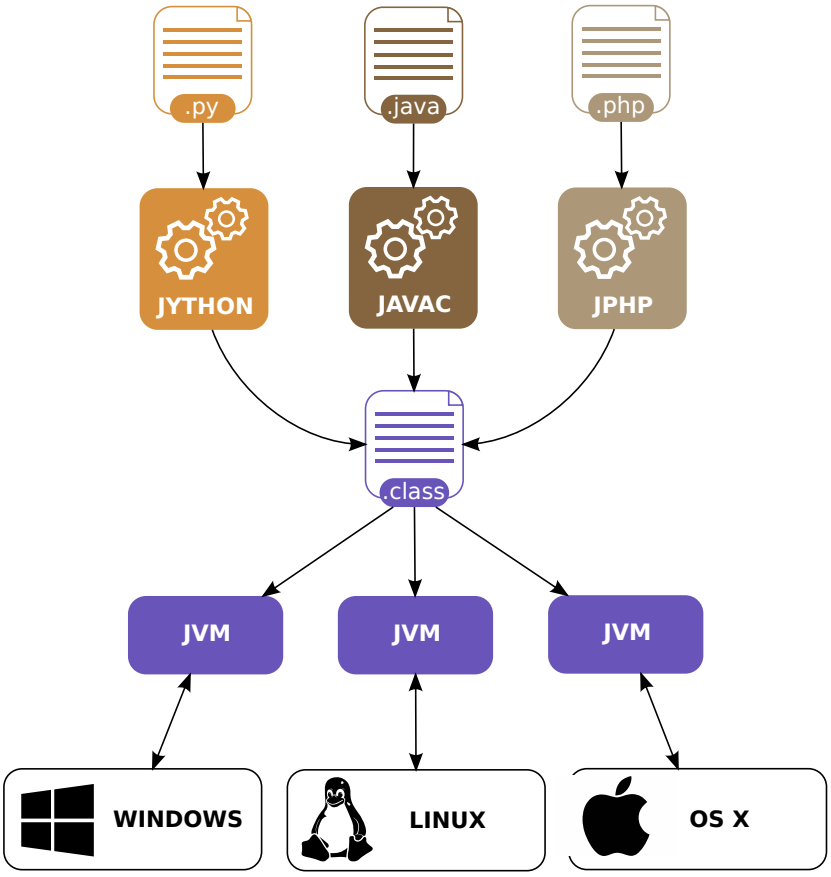
Ukázka 2.1: Definice třídy v jazyce Java

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Transformace zdrojových textů

Následující obrázek ilustruje flexibilitu JVM a Class file formátu 2.1.

Obrázek 2.1: Transformace ze zdrojových textů ke spuštění aplikace



2.3 Datové typy

JVM stejně jako Java rozděluje datové typy na dvě základní kategorie - primitivní typy a referenční typy.

Primitivní datové typy

Mezi primitivní datové typy patří datové typy pro celá čísla, čísla s plovoucí řádovou čárkou, datový typ pro logické hodnoty a datový typ `returnAddress`.

Překladač generuje pro různé primitivní typy jiné instrukce, není tedy nutné proměnné nějakým způsobem označovat.

Přehled primitivních datových typů

V závorce je uvedena velikost daného typu.

- **celočíselné datové typy** - mezi tyto datové typy patří `byte(8)`, `short(16)`, `int(32)`, `long(64)` a `char(16)`. Všechny tyto typy, s výjimkou typu `char`, jsou znaménkové a jejich výchozí hodnota je 0. Typ `char` je neznaménkový, slouží pro ukládání znaků v kódování UTF-16 a jeho výchozí hodnota je `\u0000`.
- **datové typy s plovoucí řádovou čárkou** - mezi tyto typy patří `float(32)`, který odpovídá formátu IEEE 754 s jednoduchou přesnosti, a `double(64)` který odpovídá formátu IEEE 754 s dvojnásobnou přesností.
- **datový typ boolean** - slouží pro ukládání logických hodnot `true` a `false`. Výchozí hodnota je `false`. Pro tento datový typ neexistují žádné zvláštní instrukce a veškeré operace používají instrukce pro datový typ `int` nebo `byte`. Hodnota `false` je reprezentována jako 0 a hodnota `true` jako 1.
- **datový typ returnAddress** - tento typ je používán instrukcemi `jsr`, `ret` a `jsr_w` a reprezentuje ukazatel na instrukci metody. Tento typ nelze přímo použít.

Referenční datové typy

JVM rozeznává tři druhy referenčních datových typů. Výchozí hodnotou těchto typů je speciální hodnota `null`.

- **reference na třídu** - hodnotou je reference na instanci třídy, která neimplementuje žádné rozhraní
- **reference na rozhraní** - hodnotou je reference na instanci třídy, která implementuje alespoň jedno rozhraní nebo reference na pole, které implementuje nějaké rozhraní
- **reference na pole** - hodnotou je reference na pole

2.4 Java bajtkód

Java bajtkód čítá celkem 201 instrukcí. Jednotlivé instrukce jsou relativně jednoduché. Každá instrukce se skládá z operačního kódu a případných operandů. Operační kód instrukce má velikost 1 bajt a jednoznačně identifikuje operaci, která se má provést a také počet operandů a jejich velikost. Výjimku tvoří instrukce s proměnnou délkou, jako např. instrukce `lookupswitch`, kde celkovou velikost nelze určit přímo z operačního kódu.

Pokud má operand větší velikost než 1 bajt, ukládá se jeho hodnota od nejvyššího bajtu po nejnižší (big-endian).

K předchozím 201 instrukcím je zde navíc ještě jedna speciální instrukce s operačním kódem `0xca` a symbolickým názvem `breakpoint`. Tuto instrukci mohou použít debuggery pro implementaci bodu přerušení (*z angl. breakpoint*).

HotSpot má navíc ještě několik interních instrukcí jako např. `_fast_agetfield`, které v průběhu programu nahrazují originální instrukce bajtkódu a jejichž operandy mají takový význam, aby došlo ke zrychlení procesu interpretace.

2.4.1 Typy instrukcí

Většina instrukcí slouží pro přípravu zásobníku operandů pro hlavní typy instrukcí jako jsou instrukce pro volání metod či manipulaci s datovými členy tříd.

Některé instrukce existují ve dvou verzích a to s implicitním a explicitním operandem. Výhoda implicitního operandu je ušetření místa potřebného pro bajtkód a také o něco rychlejší interpretace, protože explicitní operand není potřeba načítat. Příkladem je např. instrukce `dload_1` a `dload`.

Datové typy `byte, boolean, short` a `char` nemají zvláštní instrukce jako typy `int, long, float` a `double` a jsou manipulovány pomocí instrukcí pro typ `int`. Jednu z mála výjimek představují instrukce pro manipulaci s prvky pole, kdy jsou použity zvláštní instrukce pro všechny typy kromě `boolean`, který v tomto případě používá instrukce typu `byte`.

HotSpot Jak už bylo uvedeno v předchozí kapitole, HotSpot ukládá rámce, a tedy i zásobník operandů a pole pro lokální proměnné, na nativním zásobníku vlákna.

Tento fakt určitým způsobem souvisí s tím, proč není nutné mít zvláštní instrukce pro manipulaci s datovými typy jako `short`, jejichž velikost je menší než 4 bajty.

Pokud jsou hodnoty těchto datových typů na zásobníku ukládány jako 4 bajtové, tedy velikost typu `int`, je automaticky vynuceno 4 bajtové zarovnání (předpokládá se, že bázová adresa zásobníku je také zarovnána) a nebude docházet k neefektivnímu načítání hodnot. Tento jev je běžný u řady procesorových architektur, které data efektivně načítají pouze z určitých adres, což je dáno také konstrukcí pamětí.

Naproti tomu v poli by bylo neefektivní ukládat menší datové typy jako `int`. Na rozdíl od zásobníku operandů či pole pro lokální proměnné je datová část objektu reprezentujícího pole homogenní strukturou a slouží tedy pro ukládání hodnot pouze jednoho datového typu. Všechny typy pak mohou být ukládány ve svých velikostech a případné zarovnání postačí vynutit jen jednou před prvním prvkem. Proto pro manipulaci s prvky pole specifikace JVM definuje zvláštní instrukce i pro ostatní typy jako `char`.

Následuje přehled všech typů instrukcí vždy s několika představiteli. Stěžejní typy instrukcí jsou popsány všechny. Velikost operandů jednotlivých instrukcí je vždy 1 bajt, v opačném případě je za dvojtečkou uveden výraz udávající velikost v bajtech (op:4 $\longrightarrow$ operand op má velikost 4 bajty).

Instrukce pro přesun hodnot mezi zásobníkem a lokálními proměnnými

- **(0x21) iload op1** - vloží na zásobník operandů `int` hodnotu, nacházející se v poli lokálních proměnných na indexu `op1`
- **(0x27) dload_1** - vloží na zásobník operandů `double` hodnotu, nacházející se v poli lokálních proměnných na indexu 1 (implicitní operand)
- **(0x38) fstore op1** - vyjme `float` hodnotu z vrcholu zásobníku operandů a uloží ji do pole lokálních proměnných na index `op1`

Instrukce pro uložení konstant na zásobník

- **(0x11) sipush op1 op2** - na zásobník operandů vloží `short` hodnotu získanou z operandů `op1` a `op2`
- **(0x12) ldc op1** - podle typu záznamu na indexu `op1` v Run-Time Constant Pool tabulce vloží na zásobník operandů `int`, `float` nebo referenci na objekt
- **(0x14) ldc2_w op1 op2** - podle typu záznamu v Run-Time Constant Pool tabulce na indexu získaného z operandů `op1` a `op2` vloží na zásobník operandů buď `long` nebo `double`
- **(0x1) aconst_null** - na zásobník operandů vloží hodnotu `null`

Aritmetické instrukce

JVM poskytuje instrukce pro operace sčítání, odčítání, násobení, dělení, zbytek po dělení, negace, bitový posun, bitový součet, bitový součin, bitová nonekvivalence, inkrementace lokální proměnné a porovnání

- **(0x61) ladd** - vyjme dvě `long` hodnoty ze zásobníku operandů, sečte je a součet vloží zpět jako `long` hodnotu
- **(0x84) iinc op1 op2** - k lokální proměnné na indexu `op1` přičte hodnotu danou operandem `op2`
- **(0x96) fcmpg a (0x95) fcmpl** - obě instrukce ze zásobníku operandů vyjmou dvě `float` hodnoty `A` a `B` a porovnají je. Na zásobník operandů poté vloží `int` hodnotu -1, pokud je `A` větší než `B`, nebo hodnotu 1 pokud je `A` menší než `B`, nebo hodnotu 0 pokud jsou si hodnoty `A` a `B` rovny.

  Rozdíl mezi těmito instrukcemi je v hodnotě vložené na zásobník operandů v situaci, kdy je hodnota `A` nebo `B` rovna `NaN`. Instrukce `fcmpg` v tomto případě vkládá hodnotu 1 a `fcmpl` hodnotu -1. Kompilátor vygeneruje instrukci `fcmpg` nebo `fcmpl` na základě použitého operátoru porovnání tak, aby nemohlo dojít k nejednoznačnosti výsledku.

  Následující tabulka 2.1 ukazuje, jaké instrukce jsou generovány pro jednotlivé případy. TOS vyjadřuje stav vrcholu zásobníku (*z angl. Top Of Stack*).

Tabulka 2.1: Generování instrukcí `fcmpg` a `fcmpl`

výraz	**porovnání**	**ok/fail TOS**	**skok**
$A == B$	`fcmpg/fcmpl`	0/1,-1	`ifne fail_offset`
$A > B$	`fcmpg`	-1/0,1	`ifge fail_offset`
$A < B$	`fcmpl`	1/0,-1	`ifle fail_offset`
$A >= B$	`fcmpg`	-1,0/1	`ifgt fail_offset`
$A <= B$	`fcmpl`	1,0/-1	`iflt fail_offset`

- **(0x7e) iand** - vyjme ze zásobníku operandů dvě `int` hodnoty, provede jejich bitový součin a výsledek vloží na zásobník operandů jako `int`

Instrukce pro konverzi mezi číselnými datovými typy

Lze provádět konverzi z menšího datového typu na větší, v tom případě může dojít ke ztrátě informace, nebo z většího datového typu na menší. Ke ztrátě informace může dojít i při převádění mezi datovými typy stejných velikostí, pokud je jeden typ celočíselný a druhý s plovoucí řádovou čárkou. Podporované jsou následující konverze:

- int $\longrightarrow$ long, float, double, byte, short, char
- long $\longrightarrow$ float, double, int
- double $\longrightarrow$ int, long, float
- float $\longrightarrow$ double, int, long
- **(0x8a) l2d** - vyjme ze zásobníku operadnů `long` hodnotu a podle standardu IEEE 754 provede konverzi na `double`. Výsledná hodnota je vložena na zásobník operandů.
- **(0x92) i2c** - vyjme ze zásobníku operandů `int` hodnotu, ořízne ji na velikost typu `char` a provede neznaménkové rozšíření na typ `int`. Výsledná `int` hodnota je vložena na zásobník operandů.

Instrukce pro vytváření instancí tříd

Pouze jedna instrukce `new`.

- **(0xbb) new op1 op2** - vytvoří instanci třídy identifikované ze záznamu typu `CONSTANT_Class_info`. Záznam je nalezen pod indexem daným operandy `op1` a `op2` a nesmí identifikovat rozhraní. Reference na instanci je vložena na zásobník operandů. Datové členy instance jsou inicializovány na své výchozí hodnoty, ale samotná instance ještě není plně inicializována, protože nebyl vyvolán konstruktor.

Instrukce pro manipulaci s datovými členy tříd

Pro přístup ke statickým členům se používají instrukce `getstatic` a `putstatic`. Pro přístup k instančním členům se používá dvojice instrukcí `getfield` a `putfield`.

Při ukládání hodnot do datového členu musí být vkládané hodnoty kompatibilní s deskriptorem tohoto členu.

- **(0xb2) gestatic op1 op2** - vloží na zásobníku operandů hodnotu statického datového členu třídy, která je identifikována skrze záznam `CONSTANT_FieldRef_info`. Tento záznam se nachází se na indexu získaného z operandů `op1` a `op2`
- **(0xb3) putstatic op1 op2** - vloží hodnotu vyjmutou z vrcholu zásobníku operandů do statického datového členu třídy, identifikované skrze záznam `CONSTANT_FieldRef_info`. Tento záznam se nachází na indexu získaného z operandů `op1` a `op2`
- **(0xb4) getfield op1 op2** - vloží hodnotu datového členu objektu, odkazovaného referencí vyjmutou ze zásobníku operandů, na zásobník operandů. Datový člen je identifikována záznamem `CONSTANT_FieldRef_info`, který se nachází na indexu získaného z operandů `op1` a `op2`

- **(0xb5) putfield op1 op2** - z vrcholu zásobníku operandů vyjme hodnotu a referenci na objekt, do jehož datového členu tuto hodnotu uloží. Datový člen je identifikován záznamem `CONSTANT_FieldRef_info`, který se nachází na indexu získaného z operandů `op1` a `op2`.

Instrukce pro vytváření polí

Tři instrukce `newarray, anewarray, multianewarray`.

- **(0xbc) newarray op1** - vytvoří jednodimenzionální pole primitivního typu, který je dán hodnotou operandu `op1`. Počet prvků pole musí být uložen na zásobníku operandů jako `int` hodnota, která je poté vyjmuta a nahrazena referencí na nově vzniklé pole. Všechny prvky jsou inicializovány svými výchozími hodnotami. Možné hodnoty operandu `op1` jsou 4 až 11, udávající primitivní datové typy `boolean, char, float, double, byte, short, int, long` v tomto pořadí. Pro vícedimenzionální pole primitivního typu se používá instrukce `anewarray` nebo `multianewarray`, protože už se jedná o pole referencí.

- **(0xbd) anewarray op1 op2** - vytvoří jednodimenzionální pole objektů, přesněji řečeno referencí na objekty. Postup je stejný jako u instrukce `newarray`, s tím rozdílem, že operandy `op1` a `op2` udávají index záznamu `CONSTANT_Class_info`, který identifikuje třídu, rozhraní, nebo pole.

- **(0xc5) multianewarray op1 op2 op3** - slouží pro vytvoření vícedimenzionálního pole.

  Počet dimenzí je dán operandem `op3` a počty prvků jednotlivých dimenzí musí být uloženy jako `int` hodnoty na zásobníku operandů. Počty prvků jsou vkládány v opačném pořadí, než se vyskytují ve zdrojovém kódu, tedy poslední vložená hodnota určuje počet prvků poslední dimenze (`new int[50][20] -> bipush 50; bipush 20`).

  Operandy `op1` a `op2` udávají index záznamu `CONSTANT_Class_info`, který musí identifikovat třídu pole, jehož dimenze je minimálně stejně velká jako hodnota operandu `op3`.

  Prvky dimenze `X` jsou vždy inicializovány referencemi na pole, jejichž typ je dán typem dimenze `X+1`. Pokud není počet prvků dimenze `X+1` uveden, což se stane ve chvíli, kdy je hodnota operandu `op3` menší než počet dimenzí udávaný deskriptorem třídy pole, jsou prvky dimenze `X` inicializovány hodnotou `null` a proces končí.

  S tímto souvisí, kdy je při deklaraci proměnné typu vícedimenzionální pole použita instrukce `anewarray`, místo předpokládané instrukce `multianewarray`. Pokud není uveden počet prvků pro druhou dimenzi, jsou prvky první dimenze inicializované hodnotou `null` a z tohoto pohledu se jedná o stejnou operaci, kterou provádí instrukce `anewarray`, tedy vytvoření pole objektů. Použití instrukce `anewarray` může být v tomto případě efektivnější.

  Na zásobníku operandů je na závěr vložena reference na nově vzniklé pole. Počty prvků jednotlivých dimenzí jsou vyjmuty.

  Následující tabulka 2.2 ukazuje použité instrukce pro inicializaci polí.

Tabulka 2.2: Inicializace polí a použité instrukce

inicializace	**instrukce**
`int[] a = new int[10];`	newarray
`int[][] a = new int[10][];`	anewarray
`int[][][] a = new int[10][10][];`	multianewarray
`Object[] a = new Object[10];`	anewarray
`Object[][] a = new Object[10][10];`	multianewarray

Instrukce pro manipulaci s prvky pole

Tyto instrukce slouží pro přesouvání hodnot mezi zásobníkem operandů a prvky polí. Polem jsou myšleny objekty reprezentující pole, nikoli pole lokálních proměnných.

- **(0x31) daload** - ze zásobníku operandů vyjme index a referenci na pole a následně do zásobníku operandů vloží hodnotu, nacházející se v odkazovaném poli na daném indexu. Prvky pole musí být typu `double`.
- **(0x54) bastore** - ze zásobníku operandů vyjme referenci na pole, index a hodnotu. Tato hodnota je poté uložena do odkazovaného pole na daný index. Pole musí být typu `boolean` nebo `byte`.

Instrukce pro přímou manipulaci se zásobníkem

- **(0x57) pop** - z vrcholu zásobníku operandů vyjme 4 bajtovou hodnotu
- **(0x59) dup** - zduplikuje 4 bajtovou hodnotu na vrcholu zásobníku operandů
- **(0x5f) swap** - prohodí dvě 4 bajtové hodnoty na vrcholu zásobníku operandů

Instrukce skoku

JVM poskytuje instrukce pro podmíněné i nepodmíněné skoky a dále dvě speciální instrukce `tableswitch` a `lookupswitch`, které slouží pro efektivnější implementaci jazykového konstruktu `switch`.

Offset, udávající relativní adresu cíle skoku, se vždy počítá od adresy operačního kódu instrukce.

- **(0xa7) goto op1 op1** - realizuje nepodmíněný skok na offset daný operandy `op1` a `op2`.
- **(0xa8) jsr op1 op2** - realizuje nepodmíněný skok. Vloží na zásobník operandů hodnotu typu `returnAddress`, obsahující adresu následující instrukce a skočí na offset daný operandy `op1` a `op2`.
- **(0xa9) ret op1** - realizuje nepodmíněný skok. Skočí na adresu nacházející se v poli lokálních proměnných na indexu `op1`. Hodnota `returnAddress`, vložena na zásobník operandů např. instrukcí `jsr` proto musí být ze zásobníku operandů přesunuta do pole lokálních proměnných.
- **(0x99) ifeq op1 op2** - vyjme `int` hodnotu ze zásobníku operandů a pokud je tato hodnota 0, tak realizuje podmíněný skok na offset daný operandy `op1` a `op2`.

- **(0xa1) if_icmplt op1 op2** - ze zásobníku operandů vyjme `int` hodnoty `A` a `B` a pokud je `B` < `A`, provede podmíněný skok na offset daný operandy `op1` a `op2`
- **(0xaa) tableswitch [padding] op1:4 op2:4 op3:4 offsets:(op2-op1+1)*4** - tato instrukce s proměnnou velikostí umožňuje efektivněji implementovat konstrukt `switch`, pokud jednotlivé `case` bloky používají číselné hodnoty, které následují hned za sebou (např. všechny hodnoty z intervalu 0-5).

  Za operačním kódem instrukce je vynuceno zarovnání přidáním 0 až 3 bajtů, aby se operandy vyskytovali na adrese, která je při odečtení adresy první instrukce metody dělitelná hodnotou 4 `((&op1-&method)/4==0.0)`.

  Operand `op1` udává offset `default` bloku.

  Operandy `op2` a `op3` udávají nejmenší a největší hodnotu, které se vyskytují v `case` blocích.

  Hodnoty offsetů, reprezentované operandem `offsets`, vyjadřují relativní skok od adresy operačního kódu instrukce `tableswitch`.

  Při provádění instrukce se nejprve vyjme `int` hodnota ze zásobníku a zkontroluje se, jestli je menší než operand `op2` nebo větší než operand `op3`. V takovém případě se použije offset `default` bloku daný operandem `op1`. V opačném případě se od hodnoty odečte operátor `op2` a výsledek se použije jako index do tabulky `offsets` pro zjištění offsetu bloku, který se má vykonat.

  Nalezený offset se přičte k adrese instrukce `tableswitch` a na tomto místě se pokračuje v interpretaci.

  Podmínkou je, jak bylo na začátku zmíněno, aby hodnoty použité v `case` podmínkách byly všechny z určitého intervalu a ani jedna hodnota nechyběla. V opačném případě je tu instrukce `lookupswitch`.
- **(0xab) lookupswitch [padding] op1:4 op2:4 A-B-pairs:op2*8** - tato instrukce, stejně jako `tableswitch`, slouží pro podporu implementace konstruktu `switch` a má také proměnnou délku.

  Operand `op1` udává offset `default` bloku a musí být zarovnán stejně jako v případě `tableswitch`.

  Nyní se `lookupswitch` začíná lišit. Operand `op2` obsahuje počet dvojic `A:B`, které následují za ním.

  Symbol `A` reprezentuje `int` hodnotu použitou v `case` podmínce a symbol `B` reprezentuje offset `case` bloku. Tyto dvojice jsou seřazeny podle hodnoty symbolu `A` od nejmenší po největší.

  Při provádění instrukce se ze zásobníku vyjme `int` hodnota, která se postupně porovnává s hodnotami `A` a při shodě se jako offset použije odpovídající `B`.

  Nalezený offset se přičte k adrese instrukce `lookupswitch` a na výsledné adrese se pokračuje v interpretaci.

Instrukce pro volání metod

JVM poskytuje několik instrukcí pro volání metod a to `invokevirtual, invokespecial, invokestatic` a `invokedynamic`. Všechny instrukce předpokládají, že jsou parametry metody vloženy na na zásobníku operandů. Při volání jsou ze zásobníku operandů vyjmuty

a vloženy do pole lokálních proměnných volané metody. První parametr vkládaný na zásobník operandů má v poli lokálních proměnných index 0, druhý index 1 atd.. V případě instančních metod je prvním parametre na indexu 0 vždy objekt, na kterém je metoda volána.

- **(0xb6) invokevirtual op1 op2** - tato instrukce je použita pro volání nestatické neprivátní metody, která není konstruktorem a není metodou předka. Metoda je identifikována ze záznamu `CONSTANT_Methodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`.
- **(0xb9) invokeinterface op1 op2 op3 0** - tato instrukce slouží pro volání metod rozhraní (když je datový typ proměnné rozhraní). Metoda je identifikována ze záznamu `CONSTANT_InterfaceMethodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`. Operand `op3` musí mít nenulovou hodnotu.
- **(0xb7) invokespecial op1 op2** - tato instrukce slouží pro volání metod předka (volání metody přes klíčové slovo `super`), privátních metod a konstruktorů. Metoda je identifikována ze záznamu
  `CONSTANT_Methodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`.
- **(0xb8) invokestatic op1 op2** - tato instrukce se používá pro volání statických metod. Metoda je identifikována ze záznamu `CONSTANT_Methodref_info`, ležícího na indexu, který je dán operandy `op1` a `op2`.
- **(0xba) invokedynamic op1 op2 0 0** - tato instrukce slouží pro podporu dynamicky typovaných jazyků a její využití je popsáno v jedné z podkapitol. Java kompilátor ji negeneruje.

Následující tabulka 2.3 předpokládá existencí třídy `AA extends BB implements IAA` a zobrazuje použité instrukce při volání různých metod z těla metody třídy `AA` (pomyslná `this` reference se tedy odkazuje na objekt třídy `AA`). Statické metody jsou vždycky volány pomocí `invokestatic` a konstruktory pomocí `invokespecial`, nejsou proto uvedeny.

Tabulka 2.3: Použité instrukce pro různá volání metod

přístup	**výraz**	**instrukce**
public/protected	`aa_method()`	invokevirtual
private	`aa_method()`	invokespecial
public/protected	`bb_method()`	invokevirtual
public/protected	`super.bb_method()`	invokespecial
public	`iaa_method()`	invokevirtual
public	`((IAA)this).iaa_method()`	invokeinterface

Instrukce pro předávání návratových hodnot

Tyto instrukce slouží pro předávání návratových hodnot metod a jsou rozlišeny podle typu hodnoty, kterou vrací.

Návratová hodnota se musí nacházet na vrcholu zásobníku operandů, odkud je vyjmuta a umístěna do zásobníku operandů volajícího (s výjimkou instrukce `return`).

- **(0xac) ireturn** - slouží pro vrácení hodnot typu `boolean, byte, short, char` nebo `int`.
- **(0xb1) return** - slouží pro vrácení z metody, jejíž návratová hodnota je typu `void` a tedy nevrací žádnou hodnotu

Instrukce pro vyhození výjimky

JVM poskytuje pouze jednou instrukci `athrow`.

- **(0xbf) athrow** - na vrcholu zásobníku musí být reference na instanci třídy `Throwable` nebo nějakého jejího potomka.

  Při provádění této instrukce je tato reference ze zásobníku vyjmuta a začne se hledat blok kódu v aktuální metodě, který by tuto výjimku mohl obsloužit. Pokud je takový blok nalezen, je výjimka vložena zpět na zásobník, který je předtím vymazán a začne se vykonat kód tohoto bloku. Poté může metoda dál pokračovat. V opačném případě je metoda ukončena a pokračuje se prohledáváním metody volajícího.

  Bloky pro obsluhu výjimek jsou popsány v atributu `Code`, který se vyskytuje v atributech metody.

Instrukce pro podporu synchronizace

JVM poskytuje dvojici instrukcí `monitorenter` a `monitorexit`. S každým objektem je asociován monitor, nad kterým tyto instrukce operují.

- **(0xc2) monitorenter** - tato instrukce vyjme ze zásobníku referenci na objekt a pokusí se odkazovaný objekt zamknout
- **(0xc3) monitorexit** - na zásobníku musí být reference na objekt, jehož monitor musí být vlastněn aktuálním vláknem. Při provádění této instrukce je tato reference vyjmuta a počet vstupů do monitoru odkazovaného objektu je snížen o 1. Pokud je nyní hodnota čítače vstupů 0, vlákno již monitor nevlastní a další vlákna se mohou pokusit si ho přivlastnit.

Další instrukce

Zde je popsáno několik dalších zajímavých instrukcí.

- **(0xbe) arraylength** - ze zásobníku operandů vyjme referenci na pole a vloží `int` hodnotu, udávající délku pole
- **(0xc1) instanceof op1 op2** - ze zásobníku operandů vyjme referenci a pokud je reference stejného typu jako typ identifikovaný ze záznamu `CONSTANT_Class_info`, který se nachází na indexu daného operandy `op1` a `op2`, pak na zásobník operandů vloží hodnotu 1. V opačném případě vloží na zásobník operandů hodnotu 0.

2.5 Class file formát

Class file formát popisuje strukturu binární jednotky reprezentující zkompilovanou třídu nebo rozhraní. Tato jednotka je zpravidla výstupem překladače, ale může být také vygenerována např. pomocí knihovny pro manipulaci s bajtkódem. Všechny vícebajtové hodnoty jsou uloženy ve formátu big-endian, kdy jsou jednotlivé bajty ukládány od nejvíce významného po nejméně významný. Hodnoty nejsou nijak zarovnány a jsou umístěny hned za sebou.

2.5.1 Struktura class file formátu

Následující tabulka 2.4 zobrazuje strukturu class file formátu. Datové typy `u1`, `u2` a `u4` mají velikosti 1, 2 a 4 bajty.

Tabulka 2.4: Struktura class file formátu

typ	**název**	**počet**
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

- **magic** - speciální hodnota `0xCAFEBABE`, identifikující class file formát
- **minor_version, major_version** - dohromady `(major_version:minor_version)` tvoří číslo určující verzi class file formátu
- **constant_pool_count** - tato hodnota je rovna počtu záznamů v `constant_pool` tabulce plus 1
- **constant_pool** - tato tabulka obsahuje různé záznamy, které jsou popsány později
- **access_flag** - maska příznaků, určujících přístupová práva a vlastnosti jednotky. Možné hodnoty zobrazuje tabulka 2.5.

Tabulka 2.5: Přehled možných příznaků jednotky

název	**hodnota**	**klíčové slovo**	**popis**
ACC_PUBLIC	0x0001	public	přístup i z vnějšku balíčku
ACC_FINAL	0x0010	final	nelze dědit
ACC_SUPER	0x0020		pouze pro zpětou kompatibilitu
ACC_INTERFACE	0x0200	interface	jedná se o rozhraní
ACC_ABSTRACT	0x0400	abstract	nesmí být instancováno
ACC_SYNTHETIC	0x1000		vygenerováno kompilátorem
ACC_ANNOTATION	0x2000	@interface	jedná se o anotaci
ACC_ENUM	0x4000	enum	jedná se o výčtový typ

- **this_class** - index do `constant_pool` tabulky, kde se nachází záznam typu `CONSTANT_Class_info`, který reprezentuje tuto jednotku
- **super_class** - pokud je tato hodnota 0, musí tato jednotka reprezentovat třídu `Object`, což je jediná třída bez přímého rodiče. V opačném případě je to index do `constant_pool` tabulky, kde se nachází záznam typu `CONSTANT_Class_info`, který reprezentuje přímého předka.

  Pokud tato jednotka reprezentuje rozhraní, `super_class` index musí vést na záznam reprezentující třídu `Object`.
- **interfaces_count** - udává počet přímých rozhraní této jednotky
- **interfaces** - každá hodnota tohoto pole musí být indexem do `constant_pool` tabulky, kde se nachází záznam `CONSTANT_Class_info`, reprezentující přímé rozhraní této jednotky. Pole je seřazené tak, aby reflektovalo pořadí, v jakém byly rozhraní uváděny v deklaraci třídy (zleva doprava)
- **fields_count** - hodnota udává počet záznamů v tabulce `fields`
- **fields** - tato tabulka musí být složena ze záznamů typu `field_info`, které poskytují informace o datových členech této jednotky. Zděděné prvky zahrnuty nejsou.
- **methods_count** - udává počet záznamů v tabulce `methods`
- **methods** - tato tabulka musí být složena ze záznamů typu `method_info`, které poskytují informace o metodách jednotky. Kromě normálních metod jsou zahrnuty i konstruktory, statické metody a statické inicializační metody. Tabulka nezahrnuje zděděné metody.
- **attributes_count** - udává počet záznamů v tabulce `attributes`
- **attributes** - tato tabulka musí být složena ze záznamů typu `attribute_info`

2.5.2 Constant Pool

Constant Pool, reprezentovaný v popisu struktury class file formátu jako tabulka `constant_pool`, je tvořen různými typy záznamů, které obsahují data potřebná pro korektní nahrání jednotky a interpretaci bajtkódu metod.

Záznam je obecně reprezentován datovým typem `cp_info` 2.6 a konkrétní záznamy si pak lze představit jako přetypování hodnoty tohoto obecného typu na typ konkrétní. Constant Pool zabírá největší část jednotky.

Tabulka 2.6: Struktura záznamu `cp_info`

typ	**název[počet]**	**popis**
u1	tag	identifikuje typ záznamu
u1	info[2+]	data specifická pro typ záznamu

2.5.3 Datové členy

Každý datový člen je popsán záznamem typu `field_info` 2.7. Tabulka 2.8 ukazuje možné hodnoty položky `access_flags`.

Tabulka 2.7: Struktura záznamu `field_info`

typ	**název[počet]**	popis
u2	access_flags	přístupová práva a vlastnosti
u2	name_index	`CONSTANT_Utf8_info` reprezentující nekvalifikované jméno
u2	descriptor_index	`CONSTANT_Utf8_info` reprezentující deskriptor
u2	attributes_count	počet atributů
attribute_info	attributes[attributes_count]	atributy datového členu

Tabulka 2.8: Přehled možných příznaků datového členu

název	**hodnota**	**klíčové slovo**	**popis**
ACC_PUBLIC	0x0001	public	přístup i z vnějšku balíčku
ACC_PRIVATE	0x0002	private	přístup pouze uvnitř definující třídy
ACC_PROTECTED	0x0004	protected	přístup i z potomků
ACC_STATIC	0x0008	static	přístup přes třídu
ACC_FINAL	0x0010	final	nelze opakovaně měnit
ACC_VOLATILE	0x0040	volatile	nelze cachovat
ACC_TRANSIENT	0x0080	transient	neserializovatelné
ACC_SYNTHETIC	0x1000		vygenerováno, není ve zdrojovém kódu
ACC_ENUM	0x4000		`enum` prvek

Následuje přehled 2.9 všech konkrétních Constant Pool záznamů.

Tabulka 2.9: Typy Constant Pool záznamů

tag	název	popis
7	CONSTANT_ Class_info	Reprezentuje jednotku. `name_index:CONSTANT_Utf8_info` - plně kvalifikované jméno.
9	CONSTANT_ Fieldref_info	Reprezentuje datový člen. `class_index:CONSTANT_Class_info` - třída členu, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor
10	CONSTANT_ Methodref_info	Reprezentuje metodu. `class_index:CONSTANT_Class_info` - třída metody, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor
8	CONSTANT_ String_info	Reprezentuje String konstantu. `string_index:CONSTANT_Utf8_info` - data
11	CONSTANT_ InterfaceMethodref_info	Reprezentuje metodu rozhraní. `class_index:CONSTANT_Class_info` - třída metody, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor
3	CONSTANT_ Integer_info	Reprezentuje `int` konstantu. `bytes` - data
4	CONSTANT_ Float_info	Reprezentuje `float` konstantu. `bytes` - data v IEEE 754
3	CONSTANT_ Long_info	Reprezentuje `long` konstantu. `high_bytes, low_bytes` - data
6	CONSTANT_ Double_info	Reprezentuje `double` konstantu. `high_bytes, low_bytes` - data v IEEE 754
12	CONSTANT_ NameAndType_info	Reprezentuje název a deskriptor členu nebo metody. `name_index:CONSTANT_Utf8_info` - název, `descriptor_index:CONSTANT_Utf8_info` - deskriptor
1	CONSTANT_ Utf8_info	Reprezentuje řetězec znaků. `length` - délka, `bytes` - data
15	CONSTANT_ MethodHandle_info	Obecná funkcionalita (metoda, manipulace s datovým členem). `reference_kind` - určuje chování (např. `REF_invokeVirtual`), `reference_index:CONSTANT_FieldRef_info` nebo `CONSTANT_MethodRef_info` nebo `CONSTANT_InterfaceMethodref_info`
16	CONSTANT_ MethodType_info	Typ metody. `descriptor_index:CONSTANT_Utf8_info` - deskriptor
18	CONSTANT_ InvokeDynamic_info	Odkazován z `invokedynamic` instrukce. `bootstrap_method_attr_index` - index do `bootstrap_methods` v `BootstrapMethods` atributu jednotky, `name_and_type_index:CONSTANT_NameAndType_info` - název a deskriptor volané metody

2.5.4 Metody

Každá metoda je reprezentována záznamem `method_info` 2.10. Tabulka 2.11 ukazuje možné hodnoty položky `access_flags`.

Tabulka 2.10: Struktura záznamu `method_info`

typ	**název[počet]**	popis
u2	access_flags	přístupová práva a vlastnosti
u2	name_index	`CONSTANT_Utf8_info` reprezentující nekvalifikované jméno
u2	descriptor_index	`CONSTANT_Utf8_info` reprezentující deskriptor
u2	attributes_count	počet atributů
attribute_info	attributes[attributes_count]	atributy metody

Tabulka 2.11: Přehled možných příznaků metody

název	**hodnota**	**klíčové slovo**	**popis**
ACC_PUBLIC	0x0001	public	přístup i z vnějšku balíčku
ACC_PRIVATE	0x0002	private	přístup pouze uvnitř definující třídy
ACC_PROTECTED	0x0004	protected	přístup i z potomků
ACC_STATIC	0x0008	static	přístup přes třídu
ACC_FINAL	0x0010	final	nelze překrýt
ACC_SYNCHRONIZED	0x0020	synchronized	synchronizovaný přístup
ACC_BRIDGE	0x0040		generováno kompilátorem
ACC_VARARGS	0x0080		proměnný počet argumentů
ACC_NATIVE	0x0100	native	neimplementováno v Javě
ACC_ABSTRACT	0x0400	abstract	bez implementace
ACC_STRICT	0x0800	strictfp	FP-strict floating-point mód
ACC_SYNTHETIC	0x1000		vygenerováno, není ve zdrojovém kódu

2.5.5 Atributy

Atribut je reprezentován obecným záznamem `attribute_info` 2.12. Konkrétní atribut si pak lze představit jako přetypování obecného typu `attribute_info` na typ konkrétní (např. `SourceFile`). Kompilátory mohou podle potřeby generovat vlastní atributy. Pokud JVM tyto atributy nerozpozná, musí je ignorovat. Tabulka 2.13 zobrazuje přehled známých atributů.

Tabulka 2.12: Struktura obecného záznamu `attribute_info`

typ	**název[počet]**	popis
u2	attribute_name_index	`CONSTANT_Utf8_info` reprezentující název atributu
u4	attribute_length	velikost položky `info` v bajtech
u1	info[attribute_length]	data atributu

Následuje přehled známých atributů. Sloupec **výskyt** vyjadřuje informaci o tom, kde se daný atribut může vyskytovat. Možné hodnoty jsou **F** - atributy datových členů, **M** -

atributy metod, **C** - atributy jednotky, **B** - atributy `Code` atributu.

Tabulka 2.13: Přehled známých atributů

název	**výskyt**	**popis**
ConstantValue	F	hodnota konstantního datového členu
Code	M	instrukce bajtkódu
StackMapTable	B	použit při verifikaci
Exceptions	M	informace o výjimkách
InnerClasses	C	informace o třídách, které nejsou členy balíčku
EnclosingMethod	C	pouze pro lokální nebo anonymní třídu
Synthetic	CMF	daný prvek je vygenerován
Signature	CMF	signatura
SourceFile	C	název souboru se zdrojovým kódem jednotky
SourceDebugExtension	C	rozšířené debug informace
LineNumberTable	B	mapování bajtkódu na řádky zdrojového kódu
LocalVariableTable	B	informace o lokálních proměnných
LocalVariableTypeTable	B	signatura lokálních proměnných
Deprecated	CMF	prvek už by neměl být používán
RuntimeVisible-Annotations	CMF	runtime přístupné anotace
RuntimeInvisble-Annotations	CMF	runtime nepřístupné anotace
RuntimeVisible-ParameterAnnotations	M	runtime přístupné anotace parametrů metody
RuntimeInvisible-ParameterAnnotations	M	runtime nepřístupné anotace parametrů metody
BootstrapMethods	C	informace o bootstrap metodách používaných při provádění instrukce `invokedynamic`

2.5.6 Deskriptor

Deskriptor je řetězec složen z několika značek(**B**-byte, **C**-char, **D**-double, **F**-float, **I**-int, **J**-long, **S**-short, **Z**-boolean, **V**-void, **L***ClassName*;-reference na instanci třídy *ClassName*, [-dimenze pole), identifikujících typ datového členu, metody nebo lokální proměnné. *ClassName* je plně kvalifikované jméno jednotky, kde tečky jsou nahrazeny lomítkem (`java.lang.Object` ⟶ `java/lang/Object`). Následující tabulka 2.14 popisuje syntaxi deskriptoru pomocí regulárních výrazů.

Tabulka 2.14: Syntaxe deskriptoru popsaná PERL kompatibilními regulárními výrazy

alias	**regulární výraz**
BaseType	[BCDFIJSZ]
ObjectType	L*ClassName*;
ComponentType	*FieldType*
ArrayType	\[*ComponentType*
FieldType	*BaseType*|*ObjectType*|*ArrayType*
VoidType	V
ParameterType	*FieldType*
ReturnType	*FieldType*|*VoidType*
MethodType	\(*ParameterType**\)*ReturnType*

Deskriptor datového členu nebo lokální proměnné odpovídá zástupnému jménu *FieldType* a deskriptor metody zástupnému jménu *MethodType*.

Např. deskriptor datového členu `Object[][][]` je *[[[Ljava/lang/Object;* a deskriptor metody `short (int a, double b)` je *(ID)S*.

2.6 Dynamicky typované jazyky a Java Virtual Machine

Java Virtual Machine (*dále jen JVM*) kromě staticky typovaných jazyků jako Java[1] umožňuje běh i pro jazyky typované dynamicky. U těchto jazyků kontrola typů neprobíhá při překladu, ale v průběhu vykonávání kódu. Podporu dynamicky typovaných jazyků zajišťuje zejména instrukce `invokedynamic`. Příkladem takového jazyka je např. jazyk Golo.

Když kompilátor dynamicky typovaného jazyka kompiluje kód do bajtkódu, tak v místech, kde neví, jaký je skutečný typ proměnné (např. parametr metody), generuje při volání metody na dané proměnné místo běžných instrukcí pro volání metody instrukci `invokedynamic`.

Výskyt této instrukce se nazývá *dynamic call site*. Tato instrukce se neodkazuje na záznam `CONSTANT_Methodref_info` nebo `CONSTANT_InterfaceMethodref_info`, protože třída není známa, ale na záznam `CONSATNT_InvokeDynamic_info`.

Když interpret poprvé interpretuje konkrétní `invokedynamic` instrukci, musí nejprve provést dynamické linkování, tedy zjistit, kterou metodu má vlastně zavolat. Ze záznamu `CONSTANT_InvokeDynamic_info`, odkazovaného instrukcí `invokedynamic`, se odvodí tzv. *call site specifier*, což je záznam v `BootstrapMethods` atributu `ClassFile` záznamu.

Call site specifier obsahuje index do Constant Pool tabulky na záznam `CONSTANT_MethodHandle_info` a případné dodatečné parametry pro tzv. bootstrap metodu.

Bootstrap metoda je uživatelsky definovaná metoda, v případě nějakého kompilátoru dynamického jazyka tedy spíše generována kompilátorem, než explicitně implementována programátorem aplikace, jejímž úkolem je vrátit tzv. *call site object*, což je instance potomka třídy `java.lang.CallSite`.

`CONSTANT_MethodRef_info` záznam, reprezentující bootstrap metodu, je získán z `CONSTANT_MethodHandle_info` záznamu odkazovaného z *call site specifier záznamu*. Položka `reference_kind` u tohoto `CONSTANT_MethodHandle_info` záznamu musí mít hodnotu `REF_invokeStatic` nebo `REF_newInvokeSpecial`. Obrázek 2.2 demonstruje popisovanou posloupnost odkazů vedoucí z `invokedynamic` instrukce až k `CONSTANT_MethodRef_info` záznamu.

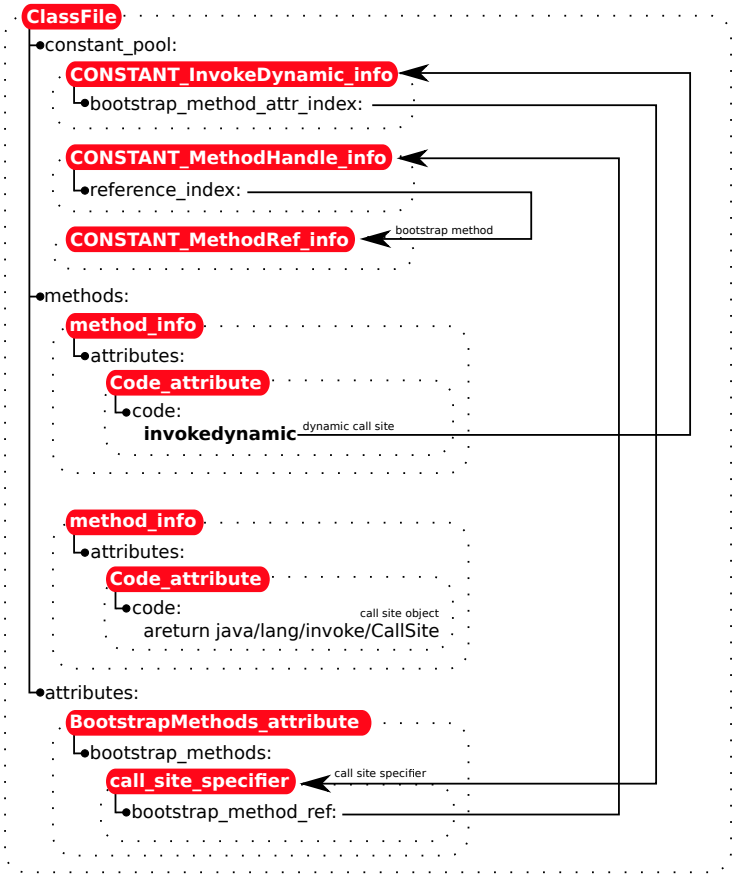
Call site object slouží jako nepřímá reference na tzv. *call site target* , reprezentovaný instancí třídy `java.lang.MethodHandle`.

Výhodou tohoto přístupu, kdy bootstrap metoda nevrací přímo instanci `java.lang.MethodHandle` je v tom, že *call site object* může změnit svůj *call site target*, tedy být znovu slinkován s jinou instancí `java.lang.MethodHandle` a konkrétní výskyt instrukce `invokedynamic` díky tomu může v průběhu života programu spouštět různé metody.

Následující obrázek představuje posloupnost odkazů

[1] Ve skutečnosti Java není úplně striktně staticky typovaný jazyk, protože podporuje runtime přetypování, kdy se kontroluje, jestli je objekt skutečně daného typu. Dalším příkladem dynamické kontroly typů je operátor instanceof.

Obrázek 2.2: Dynamické linkování `invokedynamic` instrukce



Kapitola 3

Existující nástroje pro práci s bajtkódem

Pro manipulaci s `.class` soubory existuje řada nástrojů, které lze rozdělit do 3 skupin. První skupinou jsou dekompilátory, které realizují zpětný překlad `.class` souborů do souborů se zdrojovým kódem v jazyce Java. Druhou skupinou jsou knihovny pro obecnou manipulaci s `.class` soubory. Tyto knihovny umožňuji implementovat ostatní pokročilejší nástroje, jako např. zmíněné dekompilátory. Poslední skupinou a náplni této práce nejbližší jsou nástroje, které poskytují přívětivější pohled na obsah `.class` souboru, zejména na metody a jejich instrukce. Tyto nástroje se nazývají disassemblery.

3.1 Dekompilátory bajtkódu

Smyslem dekompilace je převod zkompilovaných zdrojových textů zpět do jejich podoby před kompilací. V ideálním případě se zdrojový text vzniklý dekompilací nijak neliší od originální verze, která byla kompilována. Pro dekompilaci bajtkódu je populární zejména nástroj Procyon.

Procyon

Nástroj Procyon [7] je vyvíjený Mikem Strobelem a vznikl na základě nespokojenosti [12] s existujícími již zastaralými či dále nevyvíjenými dekompilátory jako JAD nebo JD-GUI. Výhodou Procyonu je jeho schopnost poradit si i s konstrukcemi Javy verze 5 a vyšší, jako např. enum deklarace nebo anotace.

3.2 Knihovny pro manipulaci s bajtkódem

Vzhledem k faktu, že analýza a další druhy manipulace s bajtkódem jsou poměrně časté operace, vzniklo v průběhu let hned několik nástrojů, které umožňují tyto úlohy provést bez nutnosti vymýšlet a implementovat vlastní řešení. Mezi typické operace, které lze s bajtkódem provádět, patří:

- **transformace** - změna jíž existujícího class souboru. Nemusí se jednat pouze o modifikaci instrukcí metod, ale např. o změnu constant poolu.

- **generování** - sestavení výstupu, který odpovídá class formátu, ale nenáleží mu žádné zdrojové kódy
- **analýza** - různé druhy zpracování, které mohou mít výstupy např. v podobě uživatelsky přívětivého pohledu na obsah class souboru.

BCEL

Knihovna BCEL (The Byte Code Engineering Library) [9] je ve správě Apache Foundation a jedná se o jednu z nejstarších knihoven pro manipulaci s bajtkódem. Přestože tuto knihovnu využívá řada nástrojů, není v současnosti již příliš vyvíjena.

Javassist

Javassist (Java Programming Assistant) [11] je knihovna, kterou vyvíjí Shigeru Chiba. Tato knihovna je specifická tím, že umožňuje používat Java příkazy pro generování kódu a tedy není nutná znalost instrukcí JVM.

ASM

ASM knihovna od společnosti Objectweb [8] je v současnosti nejaktivněji vyvíjena. Poskytuje poměrně jednoduché a intuitivní rozhraní, které umožňuje využít dva různé způsoby zpracování class souborů.

První způsob (Core API) je podobný zpracování XML souborů metodou SAX, kde ASM, používající návrhový vzor Visitor, očekává třídy implementující určité rozhraní, jehož jednotlivé metody reprezentují možnou reakci na výskyt nějakého prvku class formátu. Výhodou tohoto přístupu je menší paměťová náročnost, protože není nutné v paměti udržovat celkovou reprezentaci class souboru. Tato metoda ovšem neumožňuje nějakou komplexnější analýzu, kdy je nutné se např. zpětně používat již zpracované prvky. Druhý způsob (Tree API) je podobný zpracování XML souborů metodou DOM, kdy je celý class soubor načten do paměti a výsledná struktura umožňuje přistupovat ke všem dostupným prvkům libovolně. Výhodou je možnost komplexní analýzy, nevýhodou poté větší paměťová náročnost.

3.3 Disassemblery bajtkódu

Disassemblery bajtkódu umožňují uživatelsky přívětivě zobrazit obsah `.class` souborů, zejména `Code` atribut metody, který obsahuje jednotlivé instrukce.

Pro tento uživatelsky přívětivý pohled na bajtkód zavádím termín *uživatelský bajtkód.*

Následující přehled popisuje některé existující disassemblery.

Javap

Tento nástroj je spouštěn z příkazové řádky a standardně se nainstaluje při instalaci Java platformy od společnosti Oracle nebo OpenJDK. Jeho typické použití je spuštění s parametrem `-v` následovaným názvem `.class` souboru , kdy se na výstup vypíší obecné informace o jednotce, obsah Constant Pool tabulky a detailní přehled všech implementovaných metod včetně uživatelského bajtkódu [3].

Příklad spuštění nástroje Javap

```
javap -v Main.class
```

Bytecode Visualizer

Tento nástroj ze skupiny utilit Dr. Garbage tools od vývojářů Sergeje Alekseeva, Petera Palagy a Sebastiana Reschka představuje plugin do Eclipse IDE [2].

Po instalaci asociuje soubory s příponou `.class` se svým editorem. Po otevření `.class` souboru např. skrze průzkumníka projektu (*z angl. Package Explorer*), který zobrazuje i výstupní soubory překladu (*View menu* → *Customize View* → *Java output folder*), se editor otevře.

Standardně se zobrazují deklarace a uživatelský bajtkód. V nastavení pluginu (*Window* → *Preferences* → *Dr. Garbage Bytecode* → *Visualizer* → *General*) je možné nastavit např. zobrazování Constant Pool tabulky nebo tabulky mapování instrukcí na čísla řádků zdrojového kódu (LineNumberTable atribut). Unikátnější vlastností je zobrazování grafu toku vykonávání (*z angl. Control Flow Graph*), ze kterého lze vypozorovat, kde se tok větví např. kvůli instrukcím podmíněných skoků.

Bytecode Outline

Tento disassembler také funguje jako plugin do vývojového prostředí Eclipse, používá knihovnu ASM a vyvíjí ho Andrey Loskutov [1].

Po instalaci je možné zobrazit zobrazit okno (*Window* → *Show View* → *Other* → *Java* → *Bytecode*), které zobrazuje uživatelský bajtkód pro aktivní `.java` soubory. Neumožňuje ale zobrazit obsah Constant Pool tabulky. Unikátnější vlastností je možnost porovnat `.class` soubory.

Bytecode Viewer

Tato samostatná aplikace naprogramovaná v jazyce Java slouží zároveň jako dekompilátor a jejím autorem je Kalen Kinloch [4].

Program v sobě integruje existující nástroje, jako např. Procyon pro dekompilaci a Krakatau pro získání uživatelsky přívětivého bajtkódu. Celkově je aplikace poměrně velká a kromě zmíněného dekompilátoru či disassembleru disponuje i dalšími vlastnostmi, jako např. možnost psaní skriptů v jazyce Groovy pro potřeby analyzování kódu.

Krakatau

Krakatau je sada tří nástrojů - dekompilátor, disassembler a nástroj pro vytváření `.class` souborů. Tyto nástroje jsou realizovány jako skripty v jazyce Python a spouštějí se z příkazové řádky. Nástroj Krakatau je vyvíjen Robertem Groosem [6].

Výstup disassembleru je podobný jako u nástroje Javap, ale Javap mi přijde přehlednější.

Příklad spuštění disassembleru Krakatau

```
python disassemble.py -cpool -out adresar Main.class
```

JBE - Java Bytecode Editor

Tato samostatná aplikace je naprogramována v jazyce Java a jejím autorem je Ando Saabas [5].

Aplikace má jednoduché a přehledné uživatelské prostředí a kromě zobrazení jednotlivých částí `.class` souborů, mezi kterými se dá navigovat pomocí položek v levém sloupci, umožňuje i přímou modifikaci kódu metod zapisováním symbolických názvů a operandů jednotlivých instrukcí.

Ani jeden z výše uvedených disassemblerů nedokáže zobrazit uživatelsky přívětivý bajtkód sestavený z více `.class` souborů, ale vždy se zaměřuje pouze na jeden `.class` soubor, což je hlavní důvod vzniku disassembleru popsaného v další kapitole.

Kapitola 4

Návrh disassembleru

Součástí této práce je kromě teoretické roviny také vytvoření nástroje pro inspekci `.class` souborů, které představují jednotlivé zkompilované Java třídy.

V dalším obsahu nebudou pro větší přehlednost a úsporu místa uváděny plně kvalifikované názvy tříd, tedy včetně balíčku, ale pouze jednoduché názvy. Výjimku z tohoto pravidla potom mohou tvořit názvy tříd, u kterých by mohlo dojít ke kolizi názvu s jinou zmiňovanou třídou.

4.1 Požadavky

Hlavním požadavkem na vznikající nástroj je sestavení uživatelského bajtkódu ze všech `.class` souborů, které vzniknou překladem Java tříd vyskytujících se v jedné kompilační jednotce, která reprezentuje zdrojový kód v Javě verze 7. Termín *kompilační jednotka* v tomto kontextu představuje `.java` soubor.

Dalším významným požadavkem je možnost navigace mezi uživatelsky přívětivým bajtkódem a zdrojovým kódem, který mu odpovídá.

4.2 Použité technologie

Neméně důležitou částí vývoje software je také volba vhodných dostupných frameworků, knihoven a dalších existujících nástrojů, které vývojáři umožní se soustředit především na řešenou problematiku.

Programovací jazyk Java - přestože disassembler není podmínkou vyvíjet v jazyce Java, tak je výhodnější tento jazyk použít, protože poskytuje dostatečný komfort, disponuje vlastnostmi jako automatická správa paměti a je v něm naprogramována řada užitečných knihoven, které je možné při řešení problému použít.

Eclipse IDE - vlastní nástroj bude vyvíjen jako zásuvný modul (plugin) do Eclipse IDE [10], které jako jedno z nejpoužívanějších vývojových prostředí pro programování v jazyce Java navíc poskytuje také robustní infrastrukturu pro tvorbu vlastních rozšíření ve formě pluginů.

Knihovna ASM - tato knihovna, detailněji popsána v předchozí kapitole, je použita pro manipulaci s `.class` soubory a získání potřebných informací o obsažené třídě.

4.3 Úvod do vývojového prostředí Eclipse

Vývojové prostředí Eclipse (dále jen Eclipse IDE) je jedním z projektů komunity Eclipse. Eclipse IDE v sobě umožňuje integrovat rozličné technologie, povětšinou se jedná o další projekty spravované komunitou Eclipse.

Jedním z těchto projektů je také JDT (Java Development Toolkit), což je sada zásuvných modulů, které Eclipse IDE obohacují o možnost použití jako vývojového prostředí jazyka Java. Rozhraní, které poskytuje JDT, je jedním ze základních kamenů při tvorbě rozšíření, přidávajících další funkcionalitu využitelnou Java programátory. Takovým rozšířením bude také nově vznikající nástroj.

Samotné Eclipse IDE je postaveno nad jádrem Equinox, což je framework, implementující specifikaci OSGi, která popisuje dynamický modulární systém pro jazyk Java. OSGi je v současnosti považován za nejvyspělejší modulární systém pro Javu a jeho popularita vzrostla právě na základě Equinox implementace.

4.3.1 Základní model Eclipse IDE

Jakkoliv složité je Eclipse IDE, pro lepší pochopení následujícího textu je vhodné představit základní model - Workspace a Workbench.

Workbench

Na model reprezentující uživatelské prostředí Eclipse IDE lze nahlížet jako na hierarchickou strukturu, kde je kořenem Workbench.

Workbench (`IWorkbench`) je přístupný skrze statickou metodu `getWorkbench` třídy `PlatformUI` a spravuje jednotlivá okna.

Aktivní okno (Workbench Window - `IWorkbenchWindow`) lze získat voláním metody `IWorkbench.getActiveWorkbenchWindow`.

Každé okno má jednu stránku (Page - `IWorkbenchPage`), kterou lze získat pomocí metody `IWorkbenchWindow.getActivePage`.

Jednotlivými prvky na stránce jsou pohledy (View - `IViewPart`) a editory (Editor - `IEditorPart`). Důležitou metodou, kterou je poté potřeba implementovat, je metoda `createPartControl`, jejíž účelem je vytvoření vlastního vizuálního prvku pomocí knihovny SWT pro tvorbu uživatelského rozhraní.

Editor slouží pro manipulaci se zdroji (viz Workspace) typickou posloupností operací otevření-modifikace-uložení-zavření. Eclipse poskytuje zejména implementaci `AbstractDecoratedTextEditor`, která slouží jako základ pro všechny editory zaměřené na editaci zdrojových kódu. O vytvoření patřičného editoru se postará samotná platforma na základě volání `IWorkbenchPage.openEditor`.

Pohledy typicky slouží buď pro podporu aktivního editoru, jako např. Outline View, zobrazující strukturu obsahu aktivního editoru, nebo pro navigaci uvnitř Workspace, jako např. Project Explorer View. Obecně může ale pohled sloužit pro cokoliv.

Rozložení prvků na stránce udává perspektiva (Perspective - `IPerspectiveDescriptor`). Aktivní perspektivu lze získat voláním metody `IWorkbenchPage.getPerspective`.

Workspace

Workspace, přístupny skrze statickou metodu `getWorkspace` třídy `ResourcesPlugin`, reprezentuje umístění, kde jsou uloženy jednotlivé zdroje (`IResource`), typicky v lokálním

souborovém systému. Zdroje mají pro Workspace podobný význam jako soubory a adresáře pro souborový systém. Konkrétně existují 4 typy zdrojů a to soubory (`IFile`), adresáře (`IFolder`), projekty (`IProject`) a tzv. Workspace Root (`IWorkspaceRoot`).

Soubor a adresář mají analogický význam jako v souborovém systému, tedy soubor přímo obsahuje data a adresář slouží jako kontejner na další zdroje, ale data přímo neobsahuje.

Projekty potom sdružují jednotlivé soubory a adresáře.

Workspace Root, který lze získat voláním metody `IWorkspace.getRoot`, je v hierarchii zdrojů nejvýše a obsahuje všechny ostatní zdroje.

Všechny 4 typy zdrojů jsou pak standardně mapovány na adresáře nebo soubory v lokálním souborovém systému. Jedná se tedy o určitou abstrakci, která teoreticky umožňuje mít Workspace například na vzdáleném serveru, pokud bude poskytnuta patřičná implementace jednotlivých tříd. Běžnějším využitím této abstrakce je např. namapování existujícího adresáře v lokálním souborovém systému pod nějaký projekt.

4.3.2 Tvorba zásuvných modulů

Bez možnosti rozšířit Eclipse IDE o další funkcionalitu by se tato aplikace nikdy nestala tak silným hráčem na poli integrovaných vývojových prostředí, jakým v současnosti je. Tvorba vlastních rozšíření je umožněna díky konceptu pluginů. Jednotlivé subsystémy Eclipse IDE jsou sami tvořeny pluginy, které přidávají funkcionalitu do jádra implementováného pomocí již zmiňovaného frameworku Equinox.

Pluginy jsou znovupoužitelné .jar balíčky ve stylu OSGi balíčků (MANIFEST.MF obsahuje validní OSGi hlavičky), které mohou poskytovat funkcionalitu pro jiné pluginy ve formě exportovaných balíčků Java tříd (klasické Java balíčky), navázat na existující body rozšíření (*z angl. Extension-point*) jiných pluginů, případně definovat své vlastní.

Standardní distribuce Eclipse IDE pro Java programátory kromě pluginů realizujících jádro a základní subsystémy obsahuje také zmiňované JDT. Pro tvorbu pluginů je poté nutné ještě doinstalovat modul PDE (Plug-in Development Environment). Oba moduly JDT i PDE jsou pro tvorbu vlastního nástroje potřeba.

Body rozšíření a rozšíření

V předchozí podkapitole padla informace o tom, že pluginy mohou navazovat na existující body rozšíření, případě definovat své vlastní. Tento koncept umožňuje, aby na sobě byly jednotlivé pluginy nezávislé.

Definující plugin ve svém plugin.xml souboru uvede případné body rozšíření pomocí xml značky `extension-point`, jejíž atribut `scheme` se odkazuje na soubor popisující xml schéma, které musí případné rozšíření splňovat. Atribut `id` je potom rozšířením použit jako reference na daný bod rozšíření. Často tento plugin také poskytne definice Java rozhraní, které musí případné rozšíření implementovat a implementaci poté v rozšíření uvést.

Jiný plugin potom může ve svém plugin.xml souboru definovat rozšíření pomocí xml značky `extension`, jejíž atribut `point` musí obsahovat `id` nějakého bodu rozšíření. Tělo značky musí odpovídat schématu použitého bodu rozšíření.

Ve chvíli kdy plugin potřebuje přidaná rozšíření použít, musí nejprve získat registr všech rozšíření (`IExtensionRegistry`) voláním statické metody `RegistryFactory.getRegistry`. Jednotlivá rozšíření jsou reprezentována třídou implementující rozhraní `IConfigurationElement` a z registru rozšíření je lze získat pomocí metody `IExtensionRegistry.getConfigurationElementsFor`.

4.4 Vlastní knihovna a plugin do Eclipse IDE

Správně naprogramovaný software by se měl v co největší míře snažit o separaci grafického uživatelského prostředí (GUI) a vlastní logiky, proto bude tvorba disassembleru rozdělena na dvě části - tvorba knihovny poskytující rozhraní použitelné pro realizaci disassembleru a implementace disassembleru ve formě pluginu do vývojového prostředí Eclipse.

4.4.1 Návrh knihovny

Sestavení uživatelsky přívětivého pohledu na bajtkód, který je generován z překladové jednotky, vyžaduje zpracování všech binárních jednotek, které vzniknou překladem dané překladové jednotky. Místo překladové jednotky bude v dalším textu uvažován .java soubor a místo binární jednotky .class soubor.

Překladač jazyka Java při překladu .java souboru generuje všechny .class soubory do adresáře, který odpovídá Java balíčku uvedenému v .java souboru.

Pro realizaci algoritmu sestavujícího výstup použitelný při implementaci diassembleru je tedy zapotřebí poskytnout knihovně umístění .class souborů a dále identifikovat zdrojový kód, jehož překladem `.class` soubory vznikly. Výhodou dostupnosti zdrojového kódu je možnost provádět různé typy validací (není součástí zadání).

4.4.2 Návrh pluginu

Plugin by měl uživateli umožňovat zobrazit uživatelský bajtkód pro `.java` soubor editovaný v aktivním editoru a dalé také umožnit navigaci mezi tímto bajtkódem a editorem. Uživatelský bajtkód by měl být prezentován v co nejpřehlednější podobě a celá funkcionalita by měla být dostupná v každém Workbench okně a navzájem se neovlivňovat. Vhodné je také použít obarvování textu pro přehlednější prezentaci uživatelskoho bajtkódu.

Kapitola 5

Implementace disassembleru

Tato kapitola popisuje tvorbu knihovny bclib, která může být využita i při tvorbě jiných aplikací a dále pluginu `bcplugin` do vývojového prostředí Eclipse. Důležité prvky jako třídy a jejich metody či datové členy jsou popsány detailněji.

5.1 Knihovna bclib

Knihovna poskytuje zejména rozhraní použitelné klienty ve formě tříd `BytecodeAlgorithm`, `NodeProcessor` a `AbstractNodeVisitor`. Rozhraní `IFile` a `IClassContainer` zajišťují nezávislost knihovny na zdroji dat, může tedy pracovat se soubory v lokálním souborovém systému, nebo např. číst data z FTP serveru. Typické použití této knihovny lze vidět na následující ukázce 5.1.

Ukázka 5.1: Typické použití knihvony bclib

```
IFile javaFile;
IClassContainer classContainer;
// inicializace přechodzích proměnných
Result result = BytecodeAlgorithm.run(
    javaFile, classContainer, null);
ClassSourceResult classResult = result.getClassSourceResult();
Node node = classResult.getResultNode();
AbstractNodeVisitor visitor;
// incializace visitoru
NodeProcessor.process(node, visitor);
```

5.1.1 Třída DefaultASTVisitor

Tato třída rozšiřuje třídu `ASTVisitor` z Eclipse JDT modulu a jejím smyslem je poskytnout větší komfort třídám, které budou z této třídy dědit.

Třída `ASTVisitor` neumožňuje udržovat nějaký stav a pouze postupně volá metody `visit` a `endVisit`, tak jak jsou procházeny jednotlivé instance potomků třídy `ASTNode`.

Pro účely knihovny je potřeba mít informaci o aktuálním kontextu, ve kterém se proces procházení nachází. Např. pokud je volána metoda `visit(MethodDeclaration)`, je potřeba vědět, v jaké třídě se tato metoda nachází.

Třída `DefaultASTVisitor` zajišťuje vytváření kontextů a aktuální kontext lze získat voláním metody `getCurrentContext`. Kontext je instancí potomka třídy `Context` a umožňuje zejména přiřadit k sobě libovolný objekt voláním metody `setUserObject`.

Pro vytvoření kontextu slouží metody ze série `new*Context` a `create*Context`, např. `newFieldContext` a `createFieldContext`. Třída která dědí z `DefaultASTVisitor` může překrýt metody ze série `create*Context` a vracet vlastní implementace pro jednotlivé typy kontextů. Metody ze série `new*Context` jsou volány logikou `DefaultASTVisitoru` a zajišťují náležitou inicializaci nově vzniklých kontextů.

Kromě zmíněných metod ze série `new*Context` a `create*Context` jsou pro případné rozšiřující třídy zajímavé tyto metody.

- `protected void onContextActivated(Context context)` - voláno po aktivaci nového kontextu. Potomek typicky překryje a naváže nějaký vlastní objekt voláním metody `Context.setUserObject`.
- `protected void onContextDeactivated(Context context)` - voláno při deaktivaci kontextu. Potomek může překrýt a např. provést nějaké validace.

5.1.2 Třída Context a její potomci

V aktuální implementaci je vytvářeno několik typů kontextů. Každý kontext je potomkem základní třídy `DefaultASTVisitor.Context` a přidává nějakou funkcionalitu specifickou pro daný kontext. Náležitá instance potomka `ASTNode` je vždy přístupná přes metodu `getASTNode`, protože každý kontext vzniká na základě zpracovávání nějaké instance potomka `ASTNode`. Jednotliví potomci také zajišťují nastavení patřičných informací ohledně místa ve zdrojovém souboru, kde se daný kontext vyskytuje.

- `Context` - Tato abstraktní třída dědí ze třídy `Tree<Context>` a poskytuje obecnou funkcionalitu pro všechny typy kontextů. Každý kontext může manipulovat s uživatelským objektem (`setUserObject, getUserObject`), mít nastaven řádek a pozici ve zdrojovém textu (`setLine, getLine, setOffset, getOffset`) a nastavenou instanci `ASTNode`, ke které se vztahuje (`setAstNode, getAstNode`). Díky předkovi `Tree<Context>` každý kontext disponuje vlastnostmi typickými pro stromové struktury, jako přidávání nových prvku, získání rodiče apod. Metoda `findParentContext` je odpovědná za nalezení nejbližšího rodiče určitého typu, např. nejbližší `ClassContext`.
- `RootContext` - reprezentuje kořenový kontext. V celém stromu kontextů je pouze jeden a nemá žádného rodiče.
- `ClassContext` - reprezentuje třídní kontext, který je vytvářen pro každou třídu (`TypeDeclaration`), anonymní třídu (`AnonymousClassDeclaration`), výčtový typ (`EnumDeclaration`) nebo anotaci (`AnnotationTypeDeclaration`). Typ lze zjistit pomocí metody `getType`. Instance třídy v závorce je přístupná přes metodu `getAstNode`. Tento kontext navíc umožňuje zjistit interní název třídy, ze kterého jde odvodit např. název `.class` souboru, ve kterém by daná třída měla existovat ve své zkompilované podobě. Pro tento a další podobné účely slouží instance třídy `BinaryName`, kterou lze získat voláním metody `getBinaryName`.
- `FieldContext` - tento kontext reprezentuje datový člen třídy a vzniká ve chvíli, kdy visitor zpracovává instanci třídy `VariableDeclarationFragment`, jejímž předchůdcem

je instance `FieldDeclaration`. Mezi důležité vlastnosti patří zjištění jména (`getName`) nebo detekce, jestli se jedná o pole (`isArray`).

- `MethodContext` - tato třída reprezentuje kontext, který vzniká, když visitor zpracovává instanci `MethodDeclaration` a poskytuje metody jako `isConstructor` pro detekci, jestli se jedná o konstruktor, nebo `getName` pro získání jména metody.
- `BlockContext` - tento kontext náleží každému páru otevírací a zavírací složené závorky a reprezentuje složený příkaz.

5.1.3 Třída BytecodeAlgorithm

Tato třída a její statická metoda `run` slouží jako hlavní vstupní bod, poskytující funkcionalitu implementovanou knihovnou a je potomkem třídy `DefaultASTVisitor`.

- `public static Result run(IFile javaFile, IClassContainer classContainer, Map<String, Object> options)` - tato metoda vytvoří novou instanci třídy `BytecodeAlgorithm`, nastaví její datové členy a spustí provádění voláním interní metody `runInternal`. Výsledek algoritmu je vrácen v podobě objektu `Result`.

  Parametr `classContainer` reprezentuje umístění, kde se nacházejí `.class` soubory, vzniklé překladem zdrojového kódu reprezentovaného parametrem `javaFile`.

  Parametr `options` potom umožňuje předat knihovně případné dodatečné parametry, ovlivňující průběh algoritmu. V současnosti knihovna na žádné dodatečné parametry nereaguje, ale protože v budoucnosti tato možnost bude využitelná, je rozhraní navrženo už teď s tímto ohledem.

Hlavní algoritmus

Interní metoda `runInternal`, volaná metodou `run`, vytvoří objekt `ASTParser` pomocí statické metody `ASTParser.newParser`, který poté inicializuje zejména voláním jeho metody `setSource` pro nastavení zdrojového textu. Zdrojový text je získán z `IFile` objektu. Po inicializaci je spuštěno zpracování zdrojového textu pomocí metody `createAST` analyzátoru `ASTParser`, která vrací objekt `CompilationUnit`.

`CompilationUnit` reprezentuje zdrojový kód ve formě hierarchické struktury instancí potomků třídy `ASTNode`, kde kořenem je právě `CompilationUnit`. Tato třída využívá návrhový vzor Visitor a pro zpracování její struktury je možné volát metodu `accept` a předat ji potomka třídy `ASTVisitor`. Tyto třídy pochází z Eclipse modulu JDT.

Získaná instance `CompilationUnit` je knihovnou zpracována ve dvou průchodech.

První průchod

Pro první průchod je použita třída `ClassCollector`, rozšiřující třídu `DefaultASTVisitor`. Smyslem tohoto průchodu je získat informace o všech třídních kontextech (`ClassContext`) a mít k dispozici mapu interních názvů tříd, které jsou unikátní, na instance `ClassContext`.

Toto mapování vzniká překrytím funkce `onContextActivated` a uložením nové `Context` instance do kolekce `HashMap<String, ClassContext>`, pokud se jedná o `ClassContext`. Toto lze ověřit buď použitím operátoru `instanceof`, nebo voláním metody kontextu `isClassContext`. Název třídy lze získat z instance `BinaryName`, kterou disponují všechny `ClassContext` objekty, viz popis kontextů.

Navíc jsou během zpracování visitorem zachyceny všechny instance třídy `ImportDeclaration` a sdružovány ve třídě `Imports`, která je z instance třídy `ClassContext` dostupná přes metodu `getImports`.

Druhý průchod

Pro druhý průchod je použita samotná aktuální instance `BytecodeAlgorithm`. Druhý průchod s využitím `ClassCollector` instance, která je naplněna daty z prvního průchodu, sestavuje výsledek celého algoritmu, který je reprezentován třídou `Result`.

Jádrem druhého průchodu je podobně jako u průchodu prvního překrytí metody `onContextActivated`, která informuje o nově vzniklém kontextu. Na základě typu kontextu je volána některá z metod `setNodeClass`, `setNodeMethod` nebo `setNodeField`, které mají za úkol ke kontextu přiřadit pomocí jeho metody `setUserObject` správně inicializovanou instanci potomka třídy `Node`. V rámci těchto metod bude v budoucnosti také docházet k validacím.

- `private void setNodeClass(ClassContext classContext)` - tato metoda se na základě názvu třídy, získaného z instance `BinaryName` nového třídního kontextu pokusí získat ekvivalentní soubor, reprezentovaný `IFile` instancí a vrácený z `IClassContainer` voláním metody `getClassFile`. Pokud soubor není nalezen, znamená to, že se daná třída v době překladu ve zdrojovém souboru nevyskytovala. V opačném případě je interní metoda `createClassNode` požádána o vytvoření nové instance `ClassNode` , která je použita pro inicializaci třídy `NodeClass`. `NodeClass` instance je poté asociována s novým třídním kontextem pomocí `setUserObject`.

- `private void setNodeField(FieldContext fieldContext)` - tato metoda s novým kontextem datového členu asociuje pomocí metody `setUserObject` instanci třídy `FieldNode`, jejíž datový člen `name` se shoduje s názvem datového členu, který je reprezentován tímto novým kontextem. Instance třídy `FieldNode` je získána z instance třídy `ClassNode`, která je asociována s třídním kontextem, který je vždy rodičem kontextu datového členu.

- `private void setNodeMethod(MethodContext methodContext)` - tato metoda s novým kontextem metody asociuje pomocí metody `setUserObject` instanci třídy `MethodNode`, jejíž datový člen `name` se shoduje s názvem metody, která je reprezentována tímto novým kontextem. Instance třídy `MethodNode` je získána z instance třídy `ClassNode`, která je asociována s třídním kontextem, který je vždy rodičem kontextu metody. Pokud má třída více metod se stejným názvem, není tomuto novému kontextu metody nastaven odpovídající řádek zdrojového kódu, protože bez provádění validací, které jsou pouze rozpracované a v plánu do budoucna, není možné s jistotou určit, že daná instance `MethodNode` má skutečně náležet tomuto novému kontextu metody. Více metod se stejným název umožňuje koncept přetěžování metod, kdy jsou metody rozlišovány také podle počtu a typu parametrů a ne jen podle názvu.

5.1.4 Třída Node a její potomci

Výsledek celého algoritmu je hierarchická struktura typu strom, kde jednotlivé uzly jsou reprezentovány potomky třídy `Node`. Třída `Node` dědí ze třídy `Tree<Node>` a poskytuje tedy metody pro vytváření stromové reprezentace. Výhodou dědění namísto použití datového

členu typu `Tree<Node>` je, že třída `Node` automaticky získává vlastnosti třídy `Tree<Node>` a je tedy možné přímo volat metody jako `Node.add` či `Node.getParent`.

Každý uzel může mít přiřazen řádek a pozici ve zdrojovém souboru (`getSourceLine, setSourceLine, getSourceOffset, setSourceOffset`) a kolekci anotací. Kolekce anotací je reprezentovaná třídou `AnnotationMessages`.

Třída `Node` také implementuje rozhraní `Comparator<Node>` ze standardní Java knihovny a ve vlastní implementaci metody `compare` porovnává uzly podle řádku a pozice. Díky této implementaci je možné každý uzel seřadit pomocí standardní metody `Collections.sort` a výsledkem je seřazení všech přímých potomků daného uzlu podle jejich pozice. Typ uzlu je možné získat voláním metody `getType`.

- `NodeRoot` - třída reprezentující kořenový prvek celé hierarchie
- `NodeClass` - třída reprezentující třídu. Metoda `getAsmClassNode` vrací přiřazený `ClassNode` objekt. Při inicializaci `ClassNode` objektem se všechny jeho metody, datové členy a vlastní vnitřní třídy transformují do odpovídajících potomků třídy `Node` a označí jako nepoužité. Pomocí metody `setAsUsed` lze tyto prvky později označit jako použité. Toto umožňuje kontrolovat, které prvky nebyly nijak využity a tedy se např. nevyskytují ve zdrojovém kódu.
- `NodeField` - třída reprezentuje datový člen. Metoda `getAsmFieldNode` vrací přiřazený `FieldNode` objekt.
- `NodeMethod` - tato třída reprezentuje metodu. Metoda `getAsmMethodNode` vrací přiřazený `MethodNode` objekt. Při inicalizaci `MethodNode` objektem jsou všechny instrukce (`AbstractInsnNode`) transformovány na `NodeInstruction` objekty a přímo vloženy jako potomci tohoto `NodeMethod` objektu.
- `NodeInstruction` - třída reprezentující instrukci. Metoda `getAsmInsnNode` vrací přiřazený `AbstractInsnNode` objekt.

Výsledný strom složený z instancí těchto tříd tedy přímo reflektuje hierarchii analogických prvků ve zdrojovém kódu.

5.1.5 Třída AbstractNodeVisitor

Tato abstraktní třída slouží jako bázová třída pro ostatní třídy, které chtějí zpracovávat výsledný strom složený z potomků třídy `Node` a je využívána v kombinaci se třídou `NodeProcessor`. Potomci mohou např. realizovat výstup ve formě XML a podobně.

Třída se inspiruje návrhovým vzorem Visitor po vzoru JDT modulu nebo ASM knihovny a poskytuje řadu metod, které mohou být překryty a implementovány podle potřeby.

Konkrétní poskytované metody jako např. `visitNodeClass` a `endVisitNodeClass` umožňují zpracovávat třídy, metody, datové členy a instrukce. Pro instrukce jsou navíc volány i specializované metody pro konkrétní typ instrukce, jako např. `visitTableSwitchInsn`.

5.1.6 Třída NodeProcessor

`NodeProcessor` implementuje logiku procházení stromu složeného z potomků třídy `Node` a postupně vyvolává metody na poskytnuté instanci potomka `AbstractNodeVisitor`.

- `public static void process(Node node, AbstractNodeVisitor visitor)` - tato statická metoda inicializuje novou instanci třídy `NodeProcessor` a vyvolá jeho interní metodu `_process`.
- `private void _process()` - tato metoda realizuje vlastní procházení stromu a vyvolávání metod visitoru. Algoritmus využívá zásobník, do kterého je na začátku vložen kořenový uzel (`NodeRoot`). Následující `do-while` blok vždy vybere uzel ze zásobníku a pokud se jedná o `AfterVisitNode` uzel, tak vyvolá patřičnou `after*` metodu visitoru. Pokud se nejedná o `AfterVisitNode` uzel, pak je uzel seřazen voláním jeho metody `sort` a vyvolána patřičná `visit*` metoda visitoru. Poté je do zásobníku vložen nový `AfterVisitNode` uzel a postupně všechny přímé uzly, jejichž rodičem vybraný uzel je. Tyto uzly jsou do zásobníku vkládány v opačném pořadí, neboť ze zásobníku se vybírá v opačném pořadí, než vkládá. `AfterVisitNode` uzel vždy drží takový uzel, pro který je potřeba po zpracování všech jeho přímých potomků vyvolat `after*` metodu visitoru.

5.1.7 Třída Result

Tato třída slouží pouze jako kontejner na třídy `JavaSourceResult` a `ClassSourceResult`. Třídy jsou dostupné přes metody `getJavaSourceResult` respektive `getClassSourceResult`.

Třída `JavaSourceResult` poskytuje metody `getText` a `getFilename`, které vrací zdrojový kód a jméno souboru, které byly algoritmu poskytnuty ve formě instance třídy implementující rozhraní `IFile` při volání `BytecodeAlgorithm.run`, spouštějící logiku knihovny.

Třída `ClassSourceResult` poskytuje metodu `getResultNode`, která vrací výsledný kořenový `NodeRoot` uzel.

5.1.8 Třída BinaryName

Třída `BinaryName` implementuje veškerou logiku pro práci s plně kvalifikovanými názvy tříd a umožňuje snadno generovat další názvy např. pro následující anonymní třídu. Každá třída si tedy udržuje i určitý stav, jako např. následující index pro další anonymní třídu.

Hlavní metody jsou `getNextAnonymName` pro získání jména pro další anonymní třídu, `getNextLocalName` pro jméno další lokální třídy a `getInnerOrStaticNestedName` pro název další vnitřní nebo vnořené metody.

5.1.9 Rozhraní IFile

Toto rozhraní reprezentuje abstrakci souboru. Lze na něj nahlížet jako na zdroj dat. Dostupné implementace jsou `ByteFile` a `StringFile`.

- `InputStream getContent()` - vrací objekt `InputStream` ze standardní knihovny jazyka Java, který reprezentuje posloupnost bajtů dat
- `String getFilename()` - vrací jméno zdroje dat, povětšinou se tedy jedná o jméno nějakého souboru, který je instancí nějaké třídy implementující toto rozhraní reprezentován

5.1.10 Rozhraní IClassContainer

Toto rozhraní reprezentuje abstrakci umístění `.class` souborů.

- `IFile getClassFile(String filename)` - vrací instanci třídy implementující rozhraní `IJavaFile`, která reprezentuje požadovaný `.class` soubor
- `IFile[] getClassFiles()` - vrací všechny `.class` soubory v tomto umístění

5.1.11 Shrnutí

Tato část popisovala stěžejní koncepty implementace knihovny bclib spolu s přehledem hlavních tříd a rozhraní. Hlavním rysem je víceprůchodové kontextově orientované zpracování, které postupně umožňuje přidávat další validace přidáním dalšího průchodu a využitím získaných informací v dalších průchodech.

Validace jsou v aktuální podobě spíše jako plánované rozšíření do budoucna a přestože je jádro knihovny navrženo s tímto ohledem, v odevzdané verzi se validace nijak projevovat nebudou. Smyslem validací je v ideálním případě upozornit na veškeré nesrovnalosti, které je možné detekovat, jako např. neodpovídající deskriptor metody. Toto bude realizovatelné pomocí pokročilejší integrace analyzátoru `ASTParser`.

5.2 Zásuvný modul bcplugin

Samotná knihovna bclib nerealizuje žádný smysluplný výstup, který by se dal přímo prezentovat uživateli, ale je potřeba vytvořit klientskou aplikaci, která tuto knihovnu bude využívat. Pro tento účel vznikl plugin bcplugin do vývojového prostředí Eclipse.

5.2.1 Body rozšíření

Pro navázání pluginu na existující funkcionalitu Eclipse je potřeba využít některých bodů rozšíření, které již poskytují jiné pluginy realizující jádro Eclipse.

- `org.eclipse.ui.startup` - tento bod rozšíření je využíván pro provedení inicializace, poté co je vytvořen Workbench. Dodaná třída musí implementovat rozhraní `org.eclipse.ui.IStartup`. Na instanci této třídy je poté volána metoda `earlyStartup`, ve které mohou být provedeny počáteční úkony. Bcplugin poskytuje třídu `Startup`.
- `org.eclipse.ui.views` - tento bod rozšíření umožňuje definovat další pohledy, které si uživatel může nechat zobrazit na aktivní Workbench stránce. Dodaná třída musí implementovat rozhraní `org.eclipse.ui.IViewPart`. Bcplugin poskytuje třídu `BytecodeView`. Zobrazení pohledu je pak možné navigací *Window* → *Show View* → *Other...* → *BcPlugin* → *Bytecode*.
- `org.eclipse.ui.perspectiveExtensions` - tento bod rozšíření je použit pro modifikaci existujících perspektiv. Bcplugin tento bod rozšíření využívá pro modifikaci perspektivy Java a nastavení úvodní pozice pohledu s uživatelským bajtkódem tak, aby se zobrazoval napravo od hlavního editoru, s tím že hlavní plocha je rozdělena v poměru 1:1 mezi editorem a tímto pohledem.

5.2.2 Třída BcUI

Tato třída implementující rozhraní `IWindowListener` se inspiruje návrhovým vzorem Jedináček (*z angl. Singleton*) a její instance je dostupná voláním statické metody `getInstance`.

Smyslem této třídy je synchronizovat uživatelské prostředí s vnitřními stavy jednotlivých oken. Stav okna je realizován třídou `BcUI.State`.

Při inicializaci v rámci volání metody `earlyStartup` instance `Startup` je pomocí metody `IWorkbench.addWindowListener` zaregistrována instance třídy `BcUI` jako obsluha událostí souvisejících s Workbench okny.

- `public synchronized void windowActivated(IWorkbenchWindow window)` - tato metoda je volána při aktivaci okna, např. pokud uživatel přejde z jednoho okna do jiného. Pokud pro aktivované okno reprezentované parametrem `window` ještě neexistuje instance třídy `State`, což indikuje návratová hodnota `null` metody `getStateForWindow`, tak je nový stav vytvořen. Na instanci třídy `State` je poté zavolána metoda `windowActivated`.

- `public void windowDeactivated(IWorkbenchWindow window)` - tato metoda je volána při deaktivaci okna. Pokud má třída `BcUI` pro dané okno ve správě stav, je na dané instanci `State` volána metoda `windowDeactivated`.

- `public void windowClosed(IWorkbenchWindow window)` - tato metoda je volána při zavření okna. Pokud má třída `BcUI` pro dané okno ve správě stav, je na dané instanci `State` volána metoda `windowClosed` a následně je stav odstraněn.

5.2.3 Třída BcUI.State

Tato třída implementuje rozhraní `IPartListener2` a `IElementChangedListener` a slouží pro udržování stavu okna, konkrétně jeho pohledu s uživatelským bajtkódem.

- `private State(IWorkbenchWindow window)` - v konstruktoru je tato instance pomocí metody `IWorkbenchPage.addPartListener` zaregistrována jako obsluha událostí pohledů a editorů aktivní stránky okna reprezentovaného parametrem `window`. Pomocí statické metody `JavaCore.addElementChangeListener` je také zaregistrována obsluha událostí souvisejících se změnami workspace zdrojů, které jsou ve správě nějakého Java projektu.

- `public void windowActivated()` - když je tato metoda volána poprvé, je zavolána metoda `initBytecodeView`, která se ve spravovaném okně pokusí nalézt pohled s uživatelským bajtkódem, reprezentovaný instancí třídy `BytecodeView`. Dále je zavolána metody `initEditorView`.

- `public void windowClosed()` - v této metodě jsou odstraněny všechny obsluhy událostí, které byly zaregistrovány.

- `private synchronized void initEditorView()` - pokud interní stav indikuje přítomnost pohledu s uživatelským bajtkódem, je aktivní stránka spravovaného okna požádána o vrácení aktivního editoru metodou `IWorkbenchPage.getActiveEditor`. Pokud se jedná o Java editor (`EDITOR_TYPE.JAVA`), což indikuje metoda `getEditorType`, je pomocí metody `BcUtils.getEditorJavaElement` získána instance s rozhraním `IJavaElement`, která reprezentuje element, který je v editoru zobrazen. Pokud interní stav již indikuje nějaký jiný element, je tento element pomocí jeho metody `equals` porovnám s nově získaným elementem a pokud jsou elementy stejné, nic se neděje. Toto řeší situaci, aby se opětovně nespouštěla funkcionalita bclib knihovny, pokud se

element v editoru nezměnil. V opačném případě je volána metoda `handleJavaEditorChange`.

- `private void handleJavaEditorChange(ICompilationUnit compilationUnit, IEditorPart editor)` - tato metoda postupně prochází instance tříd implementujících rozhraní `IClassFromJavaStrategy` a vyvolává jejich metodu `getClassFromJava`. Tato metoda musí vrátit `null` nebo objekt třídy implementující `IClassContainer`. Pokud je takový objekt vrácen, je instancována třída `StringFile`, která je inicializována parametrem `compilationUnit` a následně zavolána metoda `setInput` pohledu uživatelského bajtkódu.

- `public void elementChanged(ElementChangedEvent event)` - tato metoda reaguje na události související se změnami Java modelu a je volána modulem JDT po předchozí registraci pomocí `JavaCore.addElementChangeListener`. Implementace této metody reaguje na událost `POST_CHANGE`, která vznikne např. při uložení souboru. Pokud je detekována změna na elementu reprezentujícím zdrojový kód, který je zobrazován aktivním editorem a pomocí kterého je tvořen aktuální uživatelský bajtkód, je zavolána metoda `startRefresh`, která odstartuje proces vedoucí k znovu vytvoření uživatelského bajtkódu. Metoda `startRefresh` má za účel zejména zajistit, aby byl onen proces spuštěn ve vlákně, které obsluhuje události uživatelského rozhraní. Toto je nutné kvůli korektnímu fungování SWT knihovny pro tvorbu uživatelského rozhraní, nad kterou je Eclipse postaven.

  Změny Java modelu jsou reprezentovány stromovou strukturou tvořenou instancemi tříd implementujících rozhraní `IJavaElementDelta`. Kořenový delta element lze získat pomocí metody `ElementChangedEvent.getDelta` a jeho potomci voláním `IJavaElementDelta.getAffectedChildren`. Samotný změněný Java element vrací metoda `IJavaElementDelta.getElement`. JDT model reprezentuje workspace zdroje a jejich obsah pomocí tříd implementujících rozhraní `IJavaElement`. Jedná se o určitou projekci workspace do „JDT workspace“. Pokud např. ve workspace není žádný Java projekt, pak z hlediska JDT modelu je to jako kdyby byl „JDT workspace“ prázdný.

5.2.4 Třída BytecodeView

`BytecodeView` dědí ze třídy `ViewPart` a realizuje pohled s uživatelským bajtkódem. Každý pohled musí implementovat metodu `createPartControl`, která vytvoří samotné uživatelské rozhraní pohledu pomocí dostupných widgetů knihovny SWT nebo JFace. JFace je určitá nástavba nad běžnými widgety knihovny SWT, která dává možnost použít tyto widgety spolu s modelem, který reprezentuje data, jež mají být prezentována. Celkově je použití JFace komfortnější než manipulace přímo se SWT widgety.

Uživatelské prostředí pohledu je realizováno pomocí třídy `BytecodeViewer`, která rozšiřuje třídu `SourceViewer`. `SourceViewer` je JFace komponenta, v jejímž jádru je SWT widget `StyledText` a jsou díky ni realizovány všechny Eclipse editory na bázi editace či zobrazení zdrojového textu. `SourceViewer` umožňuje mít přiřazen postranní panel, který je reprezentován třídou implementující rozhraní `IVerticalRuler`. Nejběžnější implementací je `CompositeRuler`, který umožňuje zobrazovat více postranních panelů vedle sebe. Tyto přídavné panely musí implementovat rozhraní `IVerticalRulerColumn` a je pomocí nich řešeno např. zobrazování čísel jednotlivých řádků editoru. Pro zobrazení čísel řádků zdrojového kódu, které odpovídají jednotlivým řádkům uživatelského bajtkódu, je použita třída

MappedLineNumberRulerColumn, jejíž kód je totožný s implementací třídy LineNumberRulerColumn z knihovny JFace, navíc s několika změnami pro potřeby uživatelského bajtkódu. Konkrétně není vykreslována posloupnost čísel od 1 do počtu řádků, ale pro každý řádek se číslo, které se má vykreslit, čte z kolekce HashMap<Integer, Integer>, kterou je možné panelu nastavit voláním metody setLineMap. Další modifikací je změna obsluhy kliku myší, kdy je volána metoda BytecodeView.handleLineSelect.

- public void setInput(IFile javaSource, IClassContainer classContainer, IJavaElement javaElement, IEditorPart editor, Map<String,Object> options) - tato metoda je hlavním vstupním bodem pro zobrazení uživatelského bajtkódu. Parametry javaSource, classContainer a options jsou vstupy pro knihovnu bclib (BytecodeAlgorithm.run). Funkce zajišťuje případné stornování právě probíhajícího sestavování uživatelského bajtkódu a start nového. Je dobrým zvykem neblokovat příliš dlouho vlákno zpracovávající události uživatelského rozhraní, proto jsou využity možnosti platformy Eclipse pro provádění souběžných úloh a samotná interakce s bclib knihovnou probíhá až ve třídě WorkJob, která rozšiřuje třídu Job a tedy její instance může být předána Eclipse infrastruktuře pro následné vykonání v jiném vlákně. Předání zajišťuje metoda schedule instance třídy IWorkspaceSiteProgessService, kterou lze z pohledu získat voláním getViewSite().getService(). BytecodeView je o dokončení zpracování informován voláním metody setJobResult, která je volána instancí WorkJobListener, která je zaregistrována na instanci WorkJob pomocí metody addJobChangeListener a tedy obsluhuje události spojené s vykonáváním dané úlohy.

- public void setJobResult(IJobChangeEvent event) - tato metoda kontroluje, jestli dokončená úloha (event.getJob()) skutečně odpovídá poslednímu požadavku a pokud tomu tak je, tak z ní získá voláním metody getUserBytecode instanci třídy UserBytecode. UserBytecode rozšiřuje třídu AbstractNodeVisitor a implementuje logiku vytváření výstupu, který je použitelný pro BytecodeViewer. Algoritmus knihovny bclib probíhal v rámci metody run třídy WorkJob. Implementace této metody se nijak výrazně neliší od dříve uvedeného kódu typického použití knihovny bclib.

5.2.5 Třída UserBytecode a UserBytecodeDocument

Třída UserBytecode rozšiřuje třídu AbstractNodeVisitor a buduje výstup použitelný třídou BytecodeViewer. BytecodeViewer dědí ze třídy SourceViewer a tedy jako vstup očekává instanci třídy implementující rozhraní IDocument.

UserBytecode poskytuje metodu getDocument, která vrací instanci třídy UserBytecodeDocument, která je potomkem třídy Document, což je výchozí implementace rozhraní IDocument. UserBytecodeDocument navíc přidává metody pro zjednodušení budování výsledného textu, který bude skrze BytecodeViewer nakonec zobrazen v SWT komponentě StyledText. Jedná se o metody jako increaseIndent a decreaseIndent, které zvyšují počet počátečních mezer na každém novém řádku, nebo např. startLine, která tyto mezery do budovaného výstupu vkládá. Dalšími důležitými metodami jsou metody addToLineMap, které umožňují vytvářet mapování mezi řádky uživatelského bajtkódu a řádky zdojového kódu. Výslednou mapu lze z UserBytecodeDocument třídy získat voláním getLineMap. Tato mapa je vstupem pro třídu MappedLineNumerRulerColumn, zmiňovanou dříve.

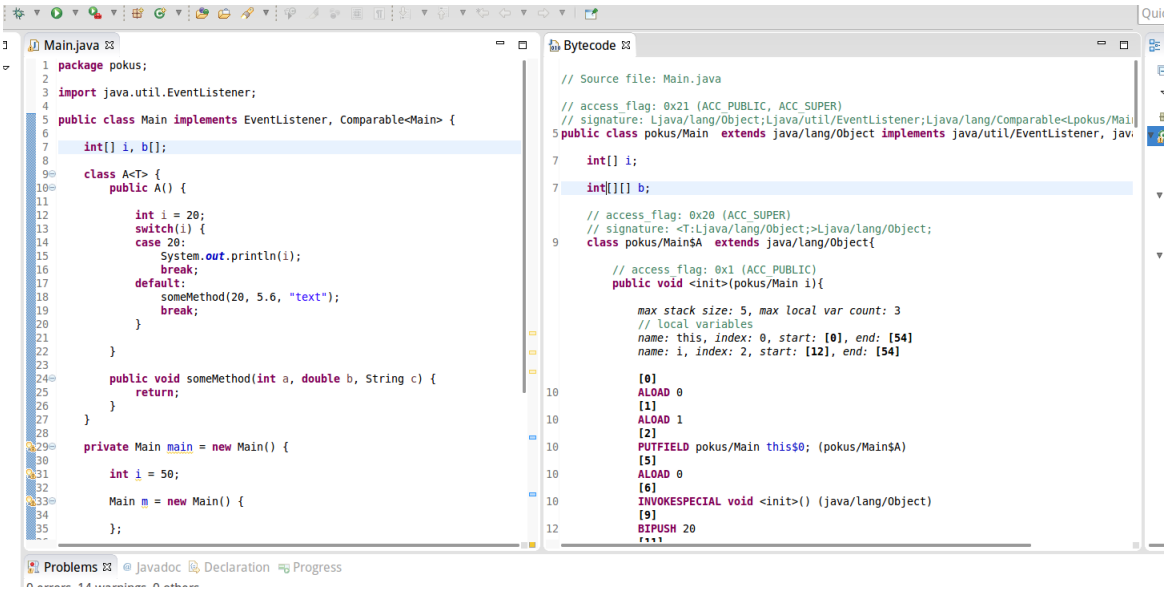
`UserBytecodeDocument` má přímou podporu pro používání stylů a většinou každá metoda, která umožňuje vkládat nějaký text na výstup, přijímá jako parametr instanci třídy `TextStyle` a zajišťuje, aby byl na vkládaný text aplikován požadovaný styl. Držení informace o stylech jednotlivých bloků textu zajišťuje třída `TextPresentation`. Instanci této třídy lze poté použít při volání metody `BytecodeViewer.applyTextPresentation`, což způsobí změnu stylu jednotlivých částí textu vykreslovaných komponentou `StyledText`.

Text na výstupu je vytvářen na základě informací získaných z jednotlivých instancí potomků třídy `Node` jako `NodeClass` či `NodeMethod`. Tyto objekty jsou postupně předávány patřičným metodám, viz `NodeProcessor` a `AbstractNodeVisitor`, který je rodičem třídy `UserBytecode`.

5.2.6 Uživatelské rozhraní

Následující obrázek 5.1 zobrazuje možnou podobu uživatelského rozhraní, tak jak vypadá aktuální verze. Lze vidět, že je zde snaha o konzistentní styl s Java editorem. Pokud uživatel změní styl Java editoru, např. barvu názvu datového členů, změní se adekvátně také pohled s uživatelským bajtkódem.

Obrázek 5.1: Návrh uživatelského rozhraní



O nastavení stylů se stará třída `StyleManager`.

5.2.7 Třída StyleManager a Style

Třída `StyleManager` využívá návrhový vzor Jedináček (*z angl. Singleton*) a její jedinou instanci lze získat voláním statické metody `getDefault`.

Tato třída ve svém konstruktoru zaregistruje obsluhy událostí změn nastavení uživatelského prostředí a při každé změně aktualizuje aktivní styl a informuje všechny obsluhy

události změny stylu. Na obsluhu této události je možné zaregistrovat instanci implementující rozhraní `IStyleListener.` Toto rozhraní má pouze jednu metodu `styleChanged` s jedním parametrem typu `StyleChangeEvent StyleChangedEvent` umožňuje získat nový i starý styl pomocí metod `getNewStyle` a `getOldStyle`. Např. pohled s uživatelským bajtkódem na tyto změny reaguje opětovným nastavením fontu a výchozí barvy textu u instance `BytecodeViewer` či změnou fontu a barvy textu instance `MappedLineNumberRulerColumn`, která realizuje postranní panel s mapováním řádků.

Všechny požadované hodnoty jako barvy apod. jsou dostupné skrze aktivní styl, který je reprezentován instancí třídy `Style` a která poskytuje metody jako `getFont` či `getLineNumberRulerColor`.

5.2.8 Shrnutí

Tento plugin zatím, co se týče celkového provedení, není lepší než ostatní dva existující pluginy Bytecode Visualizer a Bytecode Outline, které jsou zmíněny v rešerši, což by přesahovalo rámec bakalářské práce. Nicméně jako jediný poskytuje uživatelský bajtkód složený ze všech `.class` souborů, které byly generovány ze zdrojového kódu v aktivním editoru. Jako velkou výhodu vidím v tom, že jádro je tvořeno knihovnou bclib, která v sobě integruje použití analyzátoru `ASTParser`, což otevírá možností zejména pro validace a celkově pro vytvoření daleko propracovanějšího nástroje, než jsou ty dva existující a tím i docílit, aby se tento projekt v komunitě Eclipse prosadil. Podrobnější informace o použití a instalaci pluginu jsou dostupné na přiloženém DVD nosiči v souboru `manual.pdf`.

Kapitola 6

Závěr

V této práci byly popsány principy virtuálního stroje jazyka Java, zejména jeho instrukční sada a formát `.class` souborů. Dále byla provedena rešerše některých existujících nástrojů pro manipulaci s bajtkódem.

V rámci této práce vznikla knihovna bclib a velice užitečný plugin do Eclipse IDE, který ji využívá jako základní stavební kámen pro sestrojení uživatelského bajtkódu. Oba produkty jsou volně dostupné z git repozitářů a v plánu je dále je aktivně vyvíjet. Url adresa repozitáře knihovny bclib je https://github.com/hrib/bclib a pluginu bcplugin https://github.com/hrib/bcplugin.

Možných rozšíření do budoucnosti je celá řada. Co se týče knihovny bclib zejména provádění různých validací, které budou schopny odhalit nesrovnalosti mezi zdrojovým kódem a bajtkódem, které by jinak musel odhalit sám uživatel. Plugin může být dále rozšiřován ve směru větší integrace s modulem JDT.

Pro mě je osobním přínosem rozšíření znalostí za hranici běžných vědomostí Java programátora, zejména studium instrukční sady a platformy Eclipse a dále vytvoření projektu, který při dalším vývoji nemusí upadnout v zapomnění a může být Eclipse komunitou přijat a být používán.

Literatura

[1] Bytecode Outline plugin for Eclipse [online]. http://andrei.gmxhome.de/bytecode/index.html, [cit. 2015-05-12].

[2] Bytecode Visualizer [online]. http://www.drgarbage.com/bytecode-visualizer/, [cit. 2015-05-12].

[3] javap - The Java Class File Disassembler [online]. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html, [cit. 2015-05-12].

[4] BYTECODE VIEWER [online]. http://bytecodeviewer.com/, [cit. 2015-05-15].

[5] JBE - Java Bytecode Editor [online]. http://set.ee/jbe/, [cit. 2015-05-15].

[6] Krakatau Bytecode Tools [online]. https://github.com/Storyyeller/Krakatau, [cit. 2015-05-15].

[7] Procyon [online]. https://bitbucket.org/mstrobel/procyon, [cit. 2015-07-29].

[8] ASM [online]. http://asm.ow2.org/, [cit. 2015-07-30].

[9] BCEL [online]. https://commons.apache.org/proper/commons-bcel/, [cit. 2015-07-30].

[10] Eclipse IDE [online]. https://eclipse.org/, [cit. 2015-07-30].

[11] Javassist [online]. http://jboss-javassist.github.io/javassist/, [cit. 2015-07-30].

[12] Procyon / Java Decompiler [online]. https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler/, [cit. 2015-07-30].

[13] Austin, R.: What are Thread Local Allocation Buffers ? [online]. http://robsjava.blogspot.cz/2013/03/what-are-thread-local-allocation-buffers.html, 2013-17-03 [cit. 2015-05-09].

[14] Dinn, A.: OpenJDK Architecture, slide 21 [online]. http://www.dcs.gla.ac.uk/~jsinger/pdfs/sicsa_openjdk/OpenJDKArchitecture.pdf, March 2014 [cit. 2015-05-09].

[15] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: The Java Virtual Machine Specification [online]. https://docs.oracle.com/javase/specs/jvms/se7/html/, 2013-02-28 [cit. 2015-05-09].

[16] Sun Microsystems, Inc.: Memory Management in the Java HotSpot Virtual Machine [online]. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf, 2006 [cit. 2015-05-09].

Příloha A

Obsah CD

- **bp_tex** - zdrojové kódy a obrázky textové části
- **src** - zdrojové kódy knihovny bclib a pluginu bcplugin
- **manual.pdf** - návod na instalaci a použití pluginu
- **plugin** - plugin pro instalaci
- **eclipse** - 64 bitová linuxová verze Eclipse IDE Luna