Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Vizuální interpret bytecode Java

Plzeň 2020 Miroslav Krýsl

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Miroslav KRÝSL**
Osobní číslo:	**A16B0067P**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informatika**
Téma práce:	**Vizuální interpret bytecode Java**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s fungováním interpretu Java bytecode a s instrukcemi Java bytecode.
2. Prostudujte existující emulátory, zejména s ohledem na jejich uživatelskou přívětivost a informace, které poskytují.
3. Navrhněte nástroj pro vizualizaci běhu interpretu Java bytecode, vyberte podmnožinu instrukcí, která bude dostatečná pro demonstraci jeho funkčnosti.
4. Implementujte navržený nástroj pro vybranou funkcionalitu.
5. Otestujte hotovou implementaci především s ohledem na srozumitelnost informací, které poskytuje uživateli.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Richard Lipka, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **7. října 2019**
Termín odevzdání bakalářské práce: **7. května 2020**

L.S.

Doc. Dr. Ing. Vlasta Radová
děkanka

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 15. října 2019

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 21. července 2020

Miroslav Krýsl

Abstract

Visual interpreter of Java bytecode. This bachelor thesis deals with the design and implementation of a visual intrepreter for a Java bytecode subset. The first sections describe the specification of the Java virtual machine – JVM and also existing emulators for Java and other languages. In other parts of the work is the design of a visual interpreter, a brief description of its implementation and user testing results.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací vizuálního intrepreteru pro podmnožinu bajtkódu Javy. V prvních částech je popsána specifikace virtuálního stroje Javy – JVM a také existující emulátory nejen pro Javu. V dalších částech práce je návrh vizuálního interpreteru, stručný popis jeho implementace a výsledky uživatelského testování.

Poděkování

Velice rád bych poděkoval panu Ing. Richardu Lipkovi, Ph.D. za odborné vedení mé bakalářské práce, ochotu, trpělivost a za cenné rady, které mi poskytl při konzultacích. Také bych rád poděkoval i ostatním učitelům Západočeské univerzity za získání odborných znalostí a mé rodině a blízkým za podporu a pomoc během celého studia.

Obsah

1 Úvod

V této bakálářské práci se zabývám návrhem a implementací vizuálního interpreteru podmožiny instrukční sady bajtkódu Javy. Hlavní motivací pro vytvoření tohoto programu je jeho možné budoucí využití při výuce. Studenti budou moci pro tento program vytvořit zdrojový kód, sputit ho a pomocí grafického zobrazení zkoumat jeho chování nebo chování samotných instrukcí.

V první části práce se zabývám chováním virtuálního stroje Javy – JVM dle jeho specifikace. Následuje průzkum existujících emulátorů a debuggerů především s ohledem na informace, které poskytují. V další částech se věnuji návrhu vizuálního iterpreteru, jeho implementaci a následnému otestování na jeho uživatelskou přívětivost.

2 Java Virtual Machine

Java Virtual Machine (JVM) je součástí platformy Java, kterou vyvinula a v roce 1995 představila americká firma Sun Microsystems. Ta byla v roce 2009 prodána firmě Oracle Corporation, jež ve vývoji Javy dále pokračuje.

JVM slouží jako mezivrstva mezi programovacím jazykem Java a operačním systémem nebo přímo hardwarem. Je to abstraktní výpočetní stroj a stejně jako skutečné výpočetní stroje má svou instrukční sadu, tzv. Java bytecode (bajkód), provádí výpočty a manipuluje s různými oblastmi paměti.

JVM je těsně spjaté s jazykem Java, avšak důležité je pro něj pouze formát zdrojových souborů, tzv. Java class file, který obsahuje instrukce bajtkódu, definice symbolů a další přidružené informace.

Díky abstrakci instrukcí bajtkódu nezávislých na konkrétním hardwaru či operačním systému lze bajtkód vytvořit stejný pro jakoukoliv platformu, kterou JVM podporuje. Díky tomu jsou zkompilované zdrojové kódy snadno přenositelné.

JVM jako takový je pouhá specifikace toho, jak se má nevenek virtuální stroj chovat, a kdokoliv podle ní může vytvořit konkrétní implementaci. Hlavní, referenční implementace JVM je open–source virtuální stroj HotSpot, jejž vyvíjí Oracle Corporation. Dále existuje několik dalších implementací, např. GraalVM[1], který je také od Oraclu nebo Zing JVM[2] vyvíjený firmou Azul Systems, Inc.

Přestože se do bajtkódu obvykle překládají programy napsané v jazyce Java, lze do něj překládat i z jiných jazyků. V dnešní době existuje několik jazyků, které jsou přímo určeny pro běh v JVM, např. Scala, Clojure nebo Kotlin, a k některým již existujícím jazykům vznikly implementace využívající běhového prostředí JVM, např. Jython (implementace jazyka Python) nebo JRuby (implementace jazyka Ruby). Během vývoje dokonce do JVM přibyla podpora pro dynamicky typované jazyky.

V této bakalářské práci se zabývám konkrétní verzí JVM – *Java SE 11 Edition* dle jeho oficiální, volně dostupné specifikace [8]. V době tvorby této práce byla tato verze poslední nejnovější s dlouhodobou podporou (LTS – z angl. Long Term Support). V této kapitole je specifikace JVM stručně shrnuta. Všechny zde obsažené informace o JVM pocházejí z oficiální specifikace a z knihy Inside the Java 2 Virtual Machine [12], která sice popisuje JVM

[1] `https://www.graalvm.org/`

[2] `https://www.azul.com/products/zing/`

– *Java SE 2 Edition*, ale specifikace vnitřního fungování se mezi rozdílnými verzemi nijak zásadně nemění.

2.1 Datové typy

JVM operuje se dvěma druhy datových typů — primitivní typy a referenční typy. Hodnoty těchto typů je možné přiřazovat do proměnných, atributů a prvků polí, předávat je jako argumenty metod a vracet jako návratové hodnoty z metod.

Většina instrukcí bajtkódu je určena pro manipulaci s konkrétními datovými typy a s těmito instrukcemi nelze operovat nad typy jinými.

Jednotlivé datové typy jsou níže stručně popsány. Jejich úplná definice je ve specifikaci JVM [8, str. 6–11 a 26–29].

2.1.1 Primitivní datové typy

Primitivní datové typy jsou celočíselné typy, typy s plovoucí řádovou čárkou, pravdivostní typ `boolean` a typ `return_adress`.

Celočíselné typy jsou:

- `byte` — 8–bitové číslo se znaménkem kódované pomocí dvojkového doplňku.

  Rozah hodnot je (-128 – 127).

  Výchozí hodnota je 0.

- `short` — 16–bitové číslo se znaménkem kódované pomocí dvojkového doplňku.

  Rozah hodnot je (-32768 – 32767).

  Výchozí hodnota je 0.

- `int` — 32–bitové číslo se znaménkem kódované pomocí dvojkového doplňku.

  Rozah hodnot je (-2147483648 – 2147483647).

  Výchozí hodnota je 0.

- `long` — 64–bitové číslo se znaménkem kódované pomocí dvojkového doplňku.

  Rozah hodnot je (-9223372036854775808 – 9223372036854775807).

  Výchozí hodnota je 0.

- `char` — 16–bitové číslo bez znaménka představující znak Unicode v základní vícejazyčné rovině, kódovaný pomocí kódování UTF-16.

  Rozah hodnot je (0 – 65535).

  Výchozí hodnota je nulový znak ('\u0000')

Číselné typy s plovoucí řádovou čárkou jsou:

- `float` — 32–bitové číslo s jednoduchou přesností podle standardu IEEE 754 [3]. Rozsah hodnot jsou definován ve standardu.

  Výchozí hodnota je kladná 0.

- `double` — 64–bitové číslo s dvojitou přesností podle standardu IEEE 754 [1]. Rozsah hodnot je definován ve standardu.

  Výchozí hodnota je kladná 0.

Datový typ `return_address`

Datový typ `return_address` je používán instrukcemi bajtkódu *jsr*, *ret* a *jsr_w*. Jeho hodnoty jsou ukazatele na instrukci bajtkódu.

Datový typ `boolean`

Hodnoty datového typu `boolean` představují pravdivostní hodnoty `true` a `false`, výchozí hodnota je `false`.

Přestože JVM definuje tento datový typ, neexistují žádné instrukce bajtkódu určené pro operace přímo nad hodnotami `boolean`. Výrazy jazyka Java operující nad hodnotami boolean jsou kompilovány za použití datového typu `int`.

JVM přímo podporuje vytvoření pole typu `boolean` pomocí instrukce *newarray*, s hodnotami pole se však musí manipulovat za použití instrukcí pro manipulaci s polem typy `byte`. Prvky pole jsou zakódovány pomocí hodnot 1 — `true`, 0 — `false`.

2.1.2 Referenční datové typy

JVM definuje tři druhy referencí (`reference`): `class` (třída), `array` (pole) a `interface` (rozhraní). Jejich hodnoty jsou odkazy na instance tříd, pole, nebo instance tříd implementujících rozhraní.

[3] IEEE Standard for Binary Floating–Point Arithmetic (ANSIIEEE Std. 754–1985, New York)

Typ pole je tvořen typem jeho komponent, je jednorozměrné a jeho délka není typem určena. Vícerozměrná pole jsou implementovaná tak, že typ komponent pole může být také pole. Pokud je tomu tak a typ komponent tohoto pole je také pole atd., musí se na konci tohoto řetězce narazit na typ komponent, který není pole. Tento typ komponent se nazývá typ prvků pole a je to evidentně primitivní typ, nebo reference na třídu, či rozhraní. Každý typ pole je zároveň i implicitně za běhu vytvořená třída, která dědí ze třídy `Object`, tudíž je možné na jakémkoliv poli volat její metody.

Reference mohou nabývat také speciální hodnoty `null`, což znamená, že proměnná neobsahuje žádný odkaz. JVM nijak nespecifikuj přesný způsob reprezentace hodnoty `null`.

Výchozí hodnota všech typů referencí je `null`.

2.1.3 Kategorie datových typů

Vzhledem k tomu, že je JVM původně navrženo pro 32–bitovou architekturu, je na to ve specifikaci instrukcí bajtkódu a také některých oblastí paměti (zásobník operandů a lokální proměnné) brán ohled.

Pro některé datové typy, konkrétně `byte`, `shot`, `char` a `boolean`, neexistují některé odpovídající instrukce pro aritmetické a paměťové operace. Využívá se proto datový typ `int`, jehož rozsah hodnot spolehlivě pojme i hodnoty těchto typů. Na typ `int` jsou tyto datové typy převáďeny pomocí instrukcí bajtkódu tak, aby pro ně byla většina instrukcí provedena korektně i při převedení.

V JVM existují dvě katerorie výpočetních datových typů. Není přesně specifikováno, jak velké mají tyto kategorie být, je pouze určeno, jaké datové typy se do nich musí vejít. Tyto dvě kategorie odpovídají velikostem 32 a 64 bitů. Nemusí tomu ale tak nutně být, protože např. implementace JVM může jako hodnoty referencí (které jsou 2. katergorie) používat hodnoty ukazetelů do paměti, v níž se nachází reprezentace objektů. Velikosti těchto ukazatelů závisí na konkrétní architektuře počítače a může být tedy větší než 32 bitů. Některé instrukce nespecifikují konkrétní datový typ, nad nímž mohou operovat, ale pouze jeho kategorii. Seznam datových typů, jejich skutečných výpočetních typů a jejich kategorií je v tabulce 2.1.

2.2 Package a moduly

Programy určené pro JVM jsou organizovány do skupin, které sa nazývají package (balíčky). Členy package můžou být třídy a rozhraní a je možné u nich určit, zda budou přístupné i mimo tento package. Členy mohou být

Datový typ	Výpočetní typ	Kategorie
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`long`	`long`	2
`double`	`double`	2

Tabulka 2.1: Datové typy, jejich výpočetní datové typy a kategorie

i další jiné package. Jména členu package musí být v rámci package unikátní. Tato pojmenování jsou hierarchická a skládají s z řetězce jmen package a samotného jména třídy nebo rozhraní. Pokud třída nebo rozhraní není členem žádného pojmenovaného package, je členem tzv. nepojmenované package a toto jméno se neuvádí.

Package můžou být dále sdružovány do tzv. modulů, které poskytují ještě vyšší abstrakci členění a řízení přístupu. Moduly definují, jaké package v rámci modulu budou přístupné pro package definované v jiných modulech. Tato funkcionalita je v JVM relativně nová (součástí Javy od verze 9).

Celá definice packagí a modulů je ve specifikaci jazyka Javy [4, 177–203].

2.3 Class file

Zdrojové soubory pro JVM (class file) jsou v binární reprezentaci a jejich struktura a obsah jsou přesně definovány ve specifikaci JVM [8, kap. 4, str. 71–352].

Jeden soubor představuje jednu třídu (class), rozhraní (interface) nebo modul, přidružené pole a metody, jejich kód, tabulku s definicí konstant, tzv. constant pool (2.7.1), jméno rodičovské třídy nebo rozhraní a množinu jmen implementovaných rozhraní. Dále jsou v souborech uloženy doplňující informace, jako je například verze formátu zdrojového souboru, metadata sloužící pro podporu správné interpretace vnořených tříd a pro podporu debuggerů, informace pro správné ošetření výjimek atd.

Definovaná třída nebo rozhraní je vždy pojmenovaná jménem skládajícím se ze jména package, pokud je některé členem a svým vlastním jménem unikátním v rámci package.

2.3.1 Dědičnost a implementace rozhraní

Pokud má třída nebo rozhraní C v class file určenou rodičovskou třídu respektive rozhraní S, přebírá atributy a metody rodiče jako své vlastní, popřípadě je může překrýt svou vlastní definicí (2.9.5). C se potom nazývá přímý potomek S a S přímý rodič (předek) C. S, jeho přímý rodič, přímý rodič jeho přímého rodiče atd. se nazývají rodiče C a pokud má C přímého potomka, jeho přímý potomek také přímého potomka atd., nazívají se tito potomky C.

Pokud má třída nebo rozhraní určená rozhraní, jež implementuje, přebírá atributy a metody těchto rozhraní, jejich rodičů a jejich implementovaných rozhraní jako své vlastní.

Ke kterým atributům a metodám převzatých od rodičů či rozhraní má daná třída nebo rozhraní přístup určují jejich příznaky přístupu (2.4.2, 2.5.2).

Třída může přímo dědit pouze z jedné třídy. Pokud nemá žádného rodiče, dědí implicitně ze třídy `Object` (součástí standartní knihovny Java). Všechna rozhraní musí dědit explicitně pouze ze třídy `Object`. Třída i rozhraní můžou implementovat libovolné množství rozhraní.

V jazyce Java může rozhraní pouze dědit z jiného rozhraní ale nemůže žádné implementovat. Do zdrojového souboru se toto překládá tak, že rodičovské rozhraní je uvedeno jako implementované a jako rodič je uvedena třída `Object`.

Výběr konkrétního pole nebo metody z množiny rodičovských tříd nebo implementovaných rozhraních rozhraní při manipulaci, respektive volání se řídí podle algoritmu rozřešení symbolických odkazů (2.9.3) a také podle konkrétní vykonávané instrukce.

2.3.2 Instance tříd

Ke každé neabstraktní třídě (2.3.3) lze vytvořit libovolný počet tzv. instancí této třídy. Jsou to objekty vyrobené podle vzoru této třídy a mající přístup k jejím atributům a metodám.

Vytvořená instance musí obsahovat všechny nestatické atributy její třídy (i všechny zděděné). To, jakým způsobem se instance reprezentuje v paměti, JVM nijak nespecifikuje.

2.3.3 Příznaky class file

Každý class file může být doplněn o příznaky určující některé jeho vlastnosti. Nejdůležitější příznaky `PUBLIC`, `FINAL`, `INTERFACE` a `ABSTRACT` jsou

níže vysvětleny.

Pokud je nastaven příznak `INTERFACE`, class file představuje rozhraní, jinak představuje třídu. Zároveň musí být nastaven příznak `ABSTRACT`.

Pokud má třída nebo rozhraní nastavený příznak `FINAL`, nelze z této třídy, respektive rozhraní dědit a nazývá se finální.

Pokud se jedná o třídu a je nastaven příznak `ABSTRACT`, třída se nazývá abstraktní a nesmí být instanciována. Zároveň nesmí být nastaven příznak `FINAL`.

Přístupnost třídy nebo rozhraní může upravovat příznak `PUBLIC`. V tomto případě je přístupná pro všechny ostatní třídy a rozhraní a nazývá se veřejná (public). V opačném případě je přístupná pouze pro třídy náležející do stejné package, ve které je třída, respektive rozhraní definované.

2.4 Atributy

Atribut, správně „pole“ (z angl. „field“), se označuje právě jako „atribut“ z důvodu možné záměny s výrazem „pole“ (z angl. „array“), jenž označuje datovou strukturu s konečným počtem prvků stejného typu. Je to pojmenovaná proměnná určitého datového typu, který je určen tzv. deskriptorem atributu, náležející nějaké třídě nebo rozhraní. Každý atribut je v rámci své třídy jednoznačně identifikovatelný podle unikátní kombinace jména a deskriptoru, tzn. že v jedné třídě se může nacházet více stejně pojmenovaných atributů, musí však mít rozdílné deskriptory (např. inicializační metody instancí (2.5.3) jedné třídy mají všechny stejné jméno, ale rozdílné deskriptory).

2.4.1 Deskriptor atributu

Deskriptor atributu představuje datový typ hodnot, kterých může atribut nabývat. Lze ho rekurzivně reprezentovat následovně:

```
DeskriptorAtributu:
DatovýTyp

DatovýTyp:
PrimitivníTyp
Instance
Pole

PrimitivníTyp:
byte
char
double
float
int
long
short
boolean

Instance:
reference JménoTřídy

Pole:
array DatovýTyp
```

2.4.2 Příznaky atributů

Každý atribut může být doplněn o příznaky určující některé jeho vlastnosti. Nejdůležitější příznaky `STATIC`, `FINAL`, `PUBLIC`, `PROTECTED` a `PRIVATE` jsou níže vysvětleny.

Atributy mohou být přidruženy přímo samotné třídě nebo rozhraní, potom se jim říká statické atributy (nebo statické proměnné) a musí být označeny modifikátorem `STATIC`, nebo instancím třídy, ty se nazývají atributy instancí (nebo proměnné instancí). Reprezentace statických atributů je v JVM vytvořena pro příslušnou třídu pouze jednou, atributy instancí jsou vytvořeny pokaždé nové pro každou instanci třídy.

Atributy mohou být finální (konstantní), tzn. že do nich po inicializaci třídy nebo instance nemohou být přiřazeny jiné hodnoty. Tyto atributy musí být označeny příznakem `FINAL`.

Všechny atributy definované ve třídě jsou pro tuto třídu přístupné. Přístupnost atributů pro ostatní třídy je upravena žádným nebo jedním ze tří

příznaků: `PUBLIC` — atribut je přístupný pro všechny ostatní třídy a nazývá se veřejný (public), `PROTECTED` — atribut je přístupný pouze pro potomky třídy, ve které je atribut definovaný, a nazývá se chráněný (protected), `PRIVATE` — atribut není přístupný pro žádné jiné třídy a nazývá se privátní (private). Pokud není určen žádný příznak přístupu, je atribut přístupný pro všechny třídy náležející do stejné package jako třída, ve které je atribut definovaný, a nazývá se package–privátní (package-private).

2.5 Metody

Metody jsou pojmenované spustitelné bloky kódu, které můžou přijímat data jako parametry a vracet návratovou hodnotu. Uspořádaná množina datových typů těchto prametrů spolu s typem návratové hodnoty představuje tzv. deskriptor metody. Každá metoda je v rámci své třídy jednoznačně identifikovatelná unikátní kombinací jména a deskriptoru, tzn. že v jedné třídě se může nacházet více stejně pojmenovaných metod, musí však mít rozdílné deskriptory.

Ke každé metodě musí být přidružen spustitelný kód (implementace metody), kromě abstraktních metod (2.5.2), které nesmí mít žádnou implementaci. Metody rozhraní jsou obvykle abstraktní, ale mohou také poskytovat výchozí implementaci.

2.5.1 Deskriptor metody

Deskriptor metody představuje uspořádané datové typy parametrů metody a také její návratový typ. Lze ho rekurzivně reprezentovat následovně:

```
DeskriptorMetody:
({DatovýTyp}) DeskriptorNávratovéHodnoty

DeskriptorNávratovéHodnoty:
void
DatovýTyp
```

2.5.2 Příznaky metod

Každá metoda může být doplněna o příznaky určující některé vlastnosti metod. Nejdůležitější příznaky `STATIC`, `FINAL`, `ABSTRACT`, `PUBLIC`, `PROTECTED`, `PRIVATE`, `SYNCHRONIZED` a `NATIVE` jsou níže vysvětleny.

Stejně jako atributy mohou být metody přidruženy přímo samotné třídě nebo rozhraní, potom se jim říká statické metody a musí být označeny příznakem `STATIC`, nebo instancím třídy, ty se nazývají metody instancí.

Pokud má metoda nastavený příznak `ABSTRACT`, nazývá se abstraktní a nemá žádnou implementaci.

Metody mohou být finální, tzn. že nemohou být překryté (2.9.5) jinou metodou. Tyto metody jsou označeny příznakem `FINAL`.

Všechny metody definované ve třídě jsou pro tuto třídu přístupné. Přístupnost metod pro ostatní třídy a rozhraní je upravena jedním nebo žádným ze tří příznaků: `PUBLIC` — metoda je přístupná pro všechny ostatní třídy a nazývá se veřejná (public), `PROTECTED` — metoda je přístupná pouze pro potomky třídy, ve které je metoda definovaná, a nazývá se chráněná (protected), `PRIVATE` — metoda není přístupná pro žádné jiné třídy a nazývá se privátní (private). Pokud není určen žádný příznak přístupu, je metoda přístupná pro všechny třídy náležející do stejného package jako třída, ve které je metoda definovaná, a nazývá se package–privátní (package-private).

Metody rozhraní nesmí mít nastaveny příznaky `PROTECTED` a `FINAL`. Dále musí mít nastaven jeden ze dvou příznaků `PUBLIC` nebo `PRIVATE`, pokud se jedná o verzi class file $>= 52.0$. Pokud je verze < 52.0, musí mít metoda nastaveny příznaky `ABSTRACT` a `PUBLIC`.

Pokud je metoda označena příznakem `SYNCHRONIZED`, musí být volání této metody obaleno použitím synchronizace, tzv. monitoru.

Pokud je metoda označena příznakem `NATIVE`, má tato metoda implementaci v nativním kódu a je třeba s ní při volání zacházet jiným způsobem.

2.5.3 Speciální metody

Inicializační metoda instance

Každá třída má jednu nebo více inicializačních metod instancí (v jazyce Java nazývána konstruktor).

Metoda je inicializační metoda instance, pokud splňuje následující:

- Je definována ve třídě, ne v rozhraní.
- Její jméno je `<init>`.
- Její návratový typ je `void`.

Inicializační metoda instance může být zavolána pouze instrukcí *invokespecial* na neinicializované instanci třídy.

Inicializační metoda třídy

Každá třída nebo rozhraní má nejvýše jednu inicializační metodu třídy nebo rozhraní a je voláním této metodoy inicializována samotným JVM.

Metoda je inicializační metoda třídy nebo rozhraní, pokud splňuje následující:

- Její jméno je `<clinit>`.
- Její návratový typ je `void`.
- V class file verze >= 51 je metoda označená `STATIC` a nepřijímá žádné argumenty.

2.6 Symbolické reference

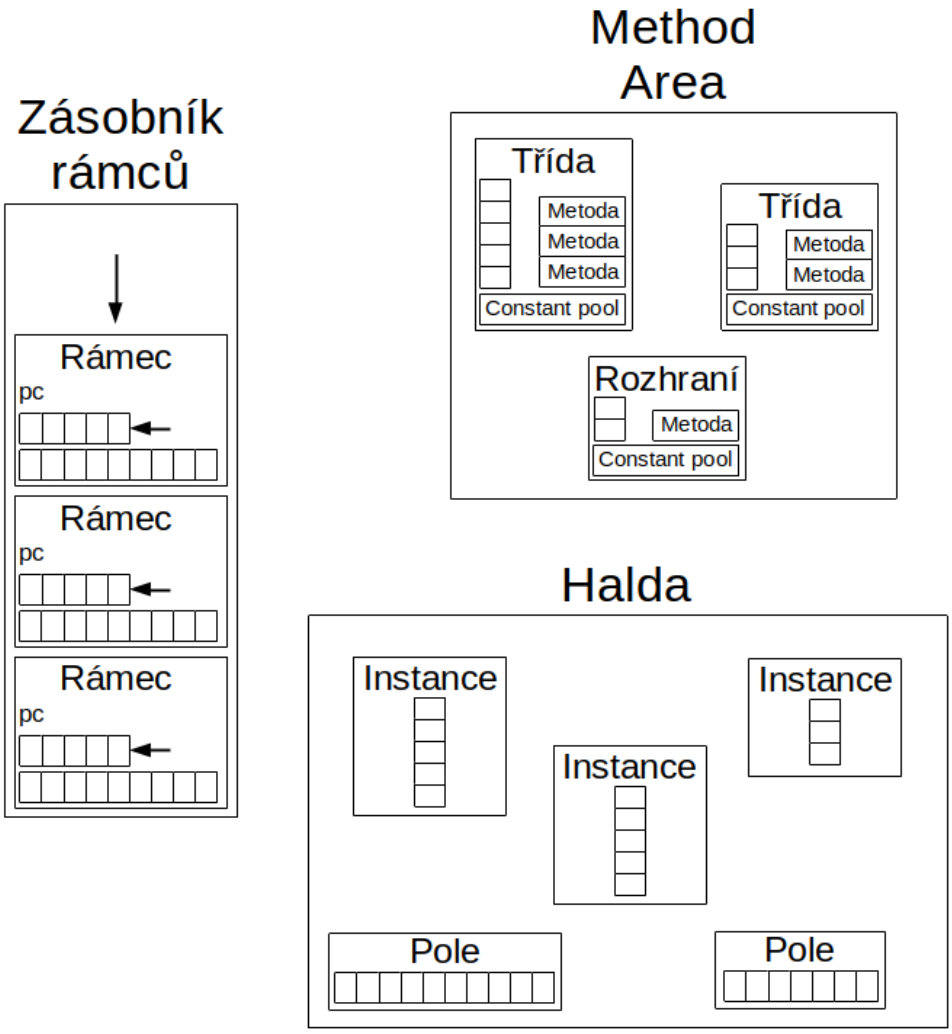
Symbolické reference jsou využívány pro určení třídy nebo rozhraní, metody, atributu nebo typu pole.

Symbolická reference na třídu je pouze jméno třídy, symbolická reference na pole je pouze deskriptor pole, symbolická reference na atribut se skládá ze symbolické reference na třídu, ve které by měl být atribut hledán, jména a deskriptoru atributu a symbolická reference na metodu se skládá ze symbolické reference na třídu, ve které by měla být metoda hledána, jména a deskriptoru metody.

Tyto symbolické reference jsou uloženy v constant poolu (2.7.1), z kterého se při volání instrukcí, jenž je využívají, načtou a podle potřeby rozřeší (2.9.3).

2.7 Oblasti paměti

JVM definuje několik oblastí paměti, které jsou za běhu používané. Jejich struktura definovaná relativně volně [8, str. 23-25], důležité je, aby se JVM chovalo navenek podle specifikace. Všechny oblasti jsou níže posány a na obrázku 2.1 je zobrazen možný příklad implementace rozvržení paměti.



Obrázek 2.1: Příklad možného rozvržení paměti implementace JVM

2.7.1 Constant pool

Constant pool je oblast paměti představující načtený constant pool z class file. Obsahuje tabulku s hodnotami konstant různých typů. K těmto hodnotám se přistupuje za běhu pomocí instrukcí pro načítání konstant.

2.7.2 Method area

Method area je oblast v paměti, ve které se nachází reprezentace všech načtených tříd, definic atributů a metod, stejně jako samotný kód metod. Dále

se zde nachází constant pool pro každou třídu a rozhraní.

2.7.3 Rámce

Pro každou metodu je vytvořen rámec (angl. frame) obahující pole lokálních proměnných (2.7.5) a zásobník operandů (2.7.4). Dále se zde nachází registr `pc` (program counter) obsahují adresu aktuálně vykonávané instrukce v metodě, odkaz na aktuálně vykonávanou metodu a třídu, jíž metoda náleží.

2.7.4 Zásobník operandů

Zásobník operandů je LIFO (last–in–first–out) paměťová struktura sloužící k ukládání operandů výpočetních typů na vrchol. Každá metoda má definovanou maximální velikosti zásobníku, kterou nesmí překročit.

Je rozdělen do buněk velikosti odpovídající 1. kategorii výpočetních typů, čemuž odpovídá i charakter instrukcí určených pro manipulaci se zásobníkem.

Pokud na zásobník uložena hodnota jednoho datového typu, nesmí být tato hodnota vyjmuta jako hodnota typu jiného.

2.7.5 Lokální proměnné

Lokální proměnné je pole konečné velikosti obsahující hodnoty lokálních proměnných aktuálně vykonávané metody. Je rozděleno do buněk velikosti odpovídající 1. kategorii výpočetních typů.

Lze do něj ukládat pouze hodnoty výpočetních typů. Nelze načíst hodnotu z lokální proměnné, pokud do ní už před tím nebyla nějaká hodnota uložena.

Pokud je do buňky lokálních proměnných na indexu i uložena hodnota 2. kategorie výpočetních typů, zabírá tato hodnota i buňku na indexu $i+1$. Tato hodnota lze načíst zpět pouze z indexu i. Pokud dojde k přepsání buňky $i+1$ jinou hodnotou, je hodnota 2. kategorie na indexu i zneplatněna a nelze jí načíst.

Pokud je do lokálních proměnných uložena hodnota jednoho datového typu, nesmí být tato hodnota načtena jako hodnota typu jiného.

2.7.6 Zásobník rámců

Zásobník rámců je LIFO (last–in–first–out) paměťová struktura sloužící k ukládání jednotlivých rámců. Při zavolání metody je pro ní vytvořen nový rámec

a ten je uložen na vrchol zásobníku. Při návratu z metody je tento rámec opět odstraněn. Každé vlákno programu má svůj vlastné zásobník.

2.7.7 Halda

Halda slouží jako oblast paměti sdílená mezi všemi vlákny programu, na které se alokují a ukládají reprezentace vytvořených instancí a polí. Vzhledem k tomu, že vytvořené objekty nemohou být nijak explicitně dealokovány, může být pamět již nepoužívaných objektů dealokována automaticky. JVM nijak nespecifikuje chování automatické správy paměti.

2.8 Instrukce bajtkódu

Instruknčí sada bajtkódu Javy obsahuje intrukce potřebné pro vykonávání programu v JVM. Každá instrukce je jednoznačně identifikovatelná svým kódem (číselná hodnota v rozsahu jednoho bytu) nebo jménem a může obsahovat definované parametry. Vzhledem k tomu, že class soubory jsou v binární reprezentaci, jsou instrukce také v binární reprezentaci. Instrukce se skládá z jednoho bajtu obsahujícího hodnotu kódu instrukce, který může být následován bajty představující hodnoty parametrů.

Jména instrukcí, které jsou určeny pro konkrétní datový typ nebo typy, používají jednoznakový prefix, respektive suffix, podle kterého lze tento datový typ, respektive typy snadno poznat (tabulka 2.2).

b	`byte` nebo `boolean`
s	`short`
c	`char`
i	`int`
l	`long`
f	`float`
d	`double`
a	`reference`

Tabulka 2.2: Prefixy jmen instrukcí značící datový typ

Instrukce lze rozdělit podle jejich určení do několika následujích skupin:

- Instrukce pro ukládání a načítání
- Aritmetické instrukce
- Instrukce převádění mezi datovými typy

- Instrukce pro vytváření a manipulaci s objekty
- Instrukce řídící běh programu
- Instrukce pro volání a návrat z metod
- Instrukce pro vyvolání výjimky
- Instrukce pro synchronizaci

Níže jsou jednotlivé skupiny stručně popsány. Konrétní fungování jednotlivých instrukcí, jejich parametry a stav zásobníku před a po vykonání jsou detailně popsány ve specifikaci JVM [8, p. 393–594].

2.8.1 Instrukce pro ukládání a načítání

Instrukce pro načítání a ukládání slouží k přesunu operandů mezi polem lokálních proměnných, zásobníkem operandů a načítání konstant.

Příslušné instrukce:

- Uložení operandu z vrcholu zásobníku do lokální proměnné:

  `istore`, `lstore`, `fstore`, `dstore`, `istore_<n>`, `lstore_<n>`, `fstore_<n>`, `dstore_<n>`.

- Načtění operandu z lokální proměnné na na vrchol zásobníku:

  `iload`, `lload`, `fload`, `dload`, `iload_<n>`, `lload_<n>`, `fload_<n>`, `dload_<n>`.

- Načtení konstanty na vrchol zásobníku:

  `bipush`, `sipush`, `ldc`, `ldc_w`, `ldc2_w`, `aconst_null`, `iconst_m1`, `iconst_<i>`, `lconst_<l>`, `fconst_<f>`, `dconst_<d>`.

Symboly použité v názvech instrukcí:

- `<n>`: `n` je celé číslo 0–3 a představuje index do pole lokálních proměnných,
- `<i>`: `i` je celé číslo 0–5 a představuje konstantu typu `int`,
- `<l>`: `l` je celé číslo 0–1 a představuje konstantu typu `long`,
- `<f>`: `f` je celé číslo 0–2 a představuje konstantu typu `float`,
- `<d>`: `d` je celé číslo 0–1 a představuje konstantu typu `double`.

2.8.2 Aritmetické instrukce

Aritmetické instrukce vypočítají výsledek aritmetické funkce o jednom nebo dvou parametrech získaných z vrcholu zásobníku a výsledek uloží zpět na zásobníku.

Příslušné instrukce:

- Sčítání:

  `iadd`, `ladd`, `fadd`, `dadd`.

- Odčítání:

  `isub`, `lsub`, `fsub`, `dsub`.

- Násobení:

  `imul`, `lmul`, `fmul`, `dmul`.

- Zbytek po dělení:

  `irem`, `lrem`, `frem`, `drem`.

- Negace:

  `ineg`, `lneg`, `fneg`, `dneg`.

- Aritmetický bitový posuv (se zachováním znaménka):

  `ishl`, `ishr`, , `lshl`, `lshr`.

- Logický bitový posuv (bez zachování znaménka):

  `iushr`, `lushr`.

- Bitový součet (OR):

  `ior`, `lor`.

- Bitový součin (AND):

  `iand`, `land`.

- Bitová nonekvivalence (XOR):

  `ixor`, `lxor`.

- Inkrementace lokální proměnné:

  `iinc`.

- Porovnání:

  `dcmpg`, `dcmpl`, `fcmpg`, `fcmpl`, `lcmp`.

Žádná z instrukcí podle specifikace nijak nedetekuje ani neindikuje přetečení hodnoty, pokud k němu dojde, oprace proběhne v pořádku i tak.

Instrukce dělení a zbytku po dělení u celočíselných typů vyvolá výjimku, pokud je dělitel nulový. U typů s plovoucí řádovou čárkou se dělení nulou nijak neindikuje.

2.8.3 Instrukce převádění mezi datovými typy

Instrukce převádění datových typů poskytují možnost převést mezi sebou primitivní datové typy. Pokud je třeba provést instrukci nepodporující určitý datový typ, je možné použít některou z instrukcí pro převod na jiný typ, který daná instrukce podporuje. Většinou nejsou instrukcemi podporovány menší datové typy (`byte`, `short`, `char`, `boolean`), ty je však možno bezestrátově převést na typ `int`, provést žádanou operaci nebo více operací a poté převést zpět.

Instrukce lze rozdělit do dvou skupin na rozšiřující a zužující.

Rozšiřující instrukce převádí hodnoty datových typů tak, že původní hodnota i se znaménkem zůstane zachována, může však dojít ke ztrátě přesnosti. Jsou to instrukce:

- `i2l`, `i2f`, `i2d`, `l2f`, `l2d` a `f2d`.

Zužující instrukce převádí na takový typ, který nemusí být schopen uchovat některou z možných hodnot zdrojového typu a výsledná hodnota může být diametrálně odlišná od původní. Jsou to instrukce:

- `i2b`, `i2c`, `i2s`, `l2i`, `f2i`, `f2l`, `d2i`, `d2l`, a `d2f`.

2.8.4 Instrukce pro vytváření a manipulaci s objekty

JVM definuje instrukce zvlášť pro objekty typu `array` a zvlášť pro `class`. Jsou to instrukce:

- Vytvoření nové instance třídy: `new`.
- Vytvoření nového pole: `newarray`, `anewarray`, `multianewarray`.
- Přístup k atributům tříd/rozhraní a instancí: `getstatic`, `puststatic`, `getfield`, `putfield`.
- Načtení hodnoty prvku pole: `baload`, `caload`, `saload`, `iaload`, `laload`, `faload`, `daload`, `aaload`.

- Uložení hodnoty jako prvku pole: `bstore`, `cstore`, `sstore`, `istore`, `lstore`, `fstore`, `dstore`, `astore`.
- Získání délky pole: `arraylength`.
- Zjištění vlastností instancí a polí: `instanceof`, `checkcast`.

2.8.5 Instrukce řídící běh programu

Pro podmíněnou (pomocí porovnání hodnot) či nepodmíněnou změnu běhu programu, tedy rozhodnutí, jaká instrukce se vykoná jako další, jsou definovány následující instrukce:

- Podmíněné větvení:

  `ifeq`, `ifne`, `iflt`, `ifle`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmpgtif_icmpge`, `if_acmpeq`, `if_acmpne`,
- Vícenásobné podmíněné větvění:

  `tableswitch`, `lookupswitch`, `multianewarray`.
- Nepodmíněné skoky: `goto`, `goto_w`, `jsr`, `jsr_w`, `ret`.

2.8.6 Instrukce pro volání a návrat z metod

Pro volání metod jsou definovány následující instrukce:

- `invokevirtual` — zavolání virtuální metody (2.9.6) instance s předáním hodnoty reference instance (`this` v jazyce Java) jako prvního (implicitního) parametru a všech dalších uvedených v deskriptoru metody.
- `invokeinterface` — zavolání implementované virtuální metody rozhraní s předáním hodnoty reference instance jako prvního (implicitního) parametru a všech dalších uvedených v deskriptoru metody.
- `invokespecial` — zavolání metody vyžadující speciální zacházení, jedná se buď o inicializační metodu instance nebo metodu třídy nebo jejího rodiče bez ohledu na dynamický výběr metod s předáním parametrů.
- `invokedynamic` — zavolání dynamicky rozřešené metody, která během kompilace ještě nemá známý typ třídy ani deskriptor.

- `invokestatic` — zavolání statické metody s předáním parametrů.

Pro návrat z metod existují instrukce odpovídající návratovým hodnotám, které jsou výpočetního typu nebo `void`:

- `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, `return`

2.8.7 Instrukce pro vyvolání výjimky

Výjimky mohou být explicitně vyvolané instrukcí `athrow`, avšak i jiné instrukce mohou za běhu vyvolat výjimku jakožto důsledek jejich nezdařeného vykonání.

2.8.8 Instrukce pro synchronizaci

JVM podporuje vícevláknovou sinchronizaci za využití synchronizační struktury monitoru jak samotných metod, tak i úseků kódu uvnitř metod.

Synchronizace metod je prováděna implicitně během volání a navrátů z metod, pokud má metoda nastaven modifikátor `SYNCHRONIZED`, synchronizace úseků kódu je prováděna za pomoci instrukcí `monitorenter` pro vstup do monitoru a `monitorexit` pro výstup.

2.9 Běh JVM

Samotný běh JVM je specifikován relativně volně, implementace však musí dodržet určité postupy popsané níže.

Během startu se připraví všechny potřebné struktury nutné pro běh JVM, tedy např. halda, method area, základní class loader, atp. Dále se nalinkuje (2.9.1) iniciální (hlavní) třída, jejíž jméno může být JVM předáno jako parametr, a zavolá se její veřejná statická metoda `public static void main(String[])`, jejíž výkonávání dále řídí další chování programu. Výkon instrukcí této metodu může vést k linkování dalších tříd nebo rozhraní a volání jejich metod.

Výkon programu končí, pokud skončí metoda `main`, nějaké vlákno zavolá metodu `exit` třídy `Runtime` nebo `System`, nebo metodu `halt` třídy *Runtime*.

Detailně běh popisuje kap. 5 ve specifikaci JVM [8, kap.5, str. 353–391].

2.9.1 Načítání, verifikace a inicializace

JVM za běhu dynamicky načítá, verifikuje a inicializuje třídy nebo rozhraní (dále jen entita) ze zdrojových souborů. Proces spočívá v načtení class file

podle jména pomocí tzv. class loaderu, který existuje jako součást JVM, ale může být implementován i uživatelsky. Načtená entita je poté jednoznačně identifikovatelná podle jména a definujícího class loaderu. Při načítání JVM verifikuje správnost binární reprezentace entity, stejně jako korektnost obsaženého bajtkódu, tzn. kontrola přetečení a podtečení zásobníku operandů, přístupy do pole lokálních proměnných na korektních indexech, kontrola shody instrukcí a datových typů hodnot, nad nimiž operují atp. Proces verifikace je detailně popsán ve specifikaci JVM [8, 188–351].

Po načtení entity se vytvoří její běhová reprezentace, načtou se reprezentace všech jejích předků a implementovaných rozhraní a připraví se reprezentace jejích statických polí. Pokud entita definuje metodu inicializace třídy nebo rozhraní, je tado metoda implicitně zavolána.

Celému tomuto procesu se říka linkování třídy nebo rozhraní a je detailně popsán ve specifikaci JVM [8, 357–368 a 385–390].

2.9.2 Řízení přístupu

Řízení přístupu je aplikováno během rozřešení symbolických referencí, aby se zajistilo, že že je odkazovaná entita přístupná z určité třídy.

Třída nebo rozhraní C je přístupná ze třídy D pokud je pravdivé právě jedno z následujících:

- C je veřejná a je členem stejného modulu jako D
- C je veřejná, je členem jiného modulu než D, ale modul C poskytuje package C modulu D.
- C není veřejná, ale je členem stejné package jako D.

Pokud C není přístupná z D, je vyvolána výjimka.

Atribut nebo metoda R deklarovaná ve třídě C je přístupná ze třídy D pokud je pravdivé právě jedno z následujících:

- R je public.
- R je potected a D je buď potomek C nebo $D = C$.

  Pokud R je statické, potom symbolická reference musí obsahovat symbolickou referenci na T a T musí být potomek nebo předek D, nebo $T = D$.
- R buď protected nebo package-private a C je deklarováno ve stejné package jako D

Pokud R není přístupné z D, je vyvolána výjimka.

Detailní popis řízení přístupu se nachází ve specifikaci JVM [8, str. 384–386].

2.9.3 Rozřešení symbolických referencí

Výkonávání některých instrukcí JVM závisí na symbolických referencích předaných jako parametry instrukce. Tyto symbolické reference představují třídu, rozhraní, atribut, nebo metodu. Pro správné vykonání instrukcí je třeba za běhu tyto symbolické instrukce rozřešit a zjistit, na které třídy, rozhraní, atributy, respektive metody odkazují. Toto rozřešení je možné vykonat při každém volání, ale vhledem k tomu, že se konkrétní symbolická reference ve konkrétním constant poolu rozřeší pokaždé stejně, je možné tento výsledek uložit do mezipaměti pro zrychlení příštího výkonu instrukce.

Při rozřešení symbolických referencí nemusí být nalezena žádná odpovídající entita, což znamená, že rozřešení selhalo a příslušný program vyvol8 výjimku.

Rozřešení symbolických referencí na třídy, rozhraní a pole

Pro rozřešení symbolické reference definované v D na třídu nebo rozhraní C se postupuje následovně:

1. Pokud symbolická reference odkazuje na pole s typem prvků `reference`, rozřeší se symbolická reference na typ jeho prvků.
2. C na níž odkazuje symbolická reference se načte s použitím class loaderu definujícího D.
3. Nakonec se aplikuje řízení přístupu (2.9.2).

Detailní popis postupu se nachází ve specifikaci JVM [8, str. 370].

Rozřešení symbolických referencí na atributy

Pro rozřešení symbolické reference definované v D na atribut třídy nebo rozhraní C se musí nejdříve rozřešit symbolická reference na C.

Při rozřešení samotného atributu se nejdříve hledá v C a poté v rodičích C podle následujícího postupu:

1. Pokud C definuje atribut se stejným jménem a deskriptorem jako symbolická reference, je tento atribut výsledkem rozřešení.

2. Jinak je atribut hledáno rekurzivně v rozhraních implementovaných třídou C. Pokud je atribut nalezen, je tento výsledkem rozřešení.

3. Jinak je atribut hledán rekurzivně v rodičovských třídách C.

4. Nakonec se aplikuje řízení přístupu (2.9.2).

Detailní popis postupu se nachází ve specifikaci JVM [8, str. 371].

Rozřešení symbolických referencí na metody

Pro rozřešení symbolické reference definované v D na metodu třídy C se musí nejdříve rozřešit symbolická reference na C.

Rozřešení samotné metody probíhá podle následujícího postupu:

1. Pokud je C rozhraní, rozřešení se nezdařilo.

2. Jinak je metoda hledána v C a v jejích předcích:

   (a) Pokud C deklaruje metodu se stejným jménem a deskriptorem jako symbolická reference, metoda byla nalezena, a tato je výsledkem rozřešení.

   (b) Jinak pokud má C přímého rodiče, pokračuje se rekurzivně s hledáním metody v něm.

3. Jinak je metoda hledána v rozhraních implementovaných třídou C:

   (a) Pokud množina všech *maximálně specifických metod rozhraní* (vysvětleno níže) implementovaných C pro stejné jméno a deskriptor jako symbolická reference obsahuje právě jednu neabstraktní metodu, je tato metoda výsledkem rozřešení.

   (b) Jinak pokud jakékoli rozhraní implementované C deklaruje metodu se stejným jménem a deskriptorem jako symbolická reference a metoda není privátní ani statická, jedna z těchto metod je vybrána neurčeným způsobem a tato je výsledkem rozřešení.

4. Nakonec se aplikuje řízení přístupu (2.9.2).

Maximálně specifická metoda rozhraní implementovaného třídou C o určitém jméně a deskriptoru je metoda, pro kterou platí:

1. je definovaná v rozhraní implementovaných C,

2. je definovaná s daným jménem a deskriptorem,

3. není ani privátní, ani statická,

4. pokud je metoda definovaná v rozhraní I, nexistují žádné další maximálně specifické metody C deklarované v potomcích I.

Detailní popis postupu se nachází ve specifikaci JVM [8, str. 372-374].

2.9.4 Rozřešení symbolických referencí na metody rozhraní

Pro rozřešení symbolické reference definované v D na metodu rozhraní I se musí nejdříve rozřešit symbolická reference na rozhraní I obsažená v této referenci.

Rozřešení samotné metody probíhá podle následujícího postupu:

1. Pokud I není rozhraní, rozřešení se nezdařilo.

2. Jinak, pokud I deklaruje metodu se stejným jménem a deskriptorem jako symbolická reference, metoda byla nalezena, a tato je výsledkem rozřešení.

3. Jinak, pokud množina všech maximálně specifických metod rozhraní (2.9.3) implementovaných I pro stejné jméno a deskriptor jako symbolická reference obsahuje právě jednu neabstraktní metodu, je tato metoda výsledkem rozřešení.

4. Jinak, pokud jakékoli rozhraní implementované I deklaruje metodu se stejným jménem a deskriptorem jako symbolická reference a metoda není privátní ani statická, jedna z těchto metod je vybrána neurčeným způsobem a tato je výsledkem rozřešení.

5. Nakonec se aplikuje řízení přístupu (2.9.2).

Detailní popis postupu se nachází ve specifikaci JVM [8, str. 374-375].

2.9.5 Překrývání metod

Překrývání metod hraje roli při jejich volání, přičemž onkrétní chování závisí na použité instrukci.

Metoda m_C instance může překrýt jinou metodu m_A instance pokud platí všechno následující:

- m_C má stejný deskriptor jako m_A

- m_C není private
- jedno z následujících platí:
  - m_A je public
  - m_A je protected
  - m_A je package-private a platí jedno z následujících:
    1. m_A je deklarována ve stejné package jako m_C
    2. pokud je m_A deklarována ve třídě A a m_C je deklarována ve třídě C, tak existuje metoda m_B deklarována ve třídě B, C je potomek B, B je potomek A, m_C může překrýt m_B a m_B může překrýt m_A

Detailní popis překrývání metod se nachází ve specifikaci JVM [8, str. 386-387].

2.9.6 Výběr virtuální metody pro zavolání

Během výkonu instrukcí *invokeinterface* a *invokevirtual* je pro zavolání vybrána metoda s ohledem na třídu nebo rozhraní C objektu na zásobníku a také na metodu m_R, která byla rozřešena ze symbolické adresy, kterou obsahuje instrukce. Takovému výběru se říká virtuální. Pravidla pro výběr metody jsou následující:

- Pokud je m_R private, poté je m_R vybrána.
- Jinak, pokud C obsahuje deklaraci metody instance m_C která může překrýt m_R, potom m_C je vybrána.
- jinak, pokud má C rodičovskou třídu, hledá se deklarace metody instance která, může překrýt m_R, počínaje přímým rodičem C a pokračuje přímým rodičem tohoto rodiče atd., dokud není metoda nalezena nebo neexistuje žádný další rodič. Pokud je metoda nalezena, je tato vybrána.
- Jinak jsou nalezeny maximálně specifické metody implementovaných rozhraní. Pokud právě jedna nalezená metoda odpovídá jménem i deskriptorem m_R a není abstraktní, pak je vybrána tato metoda.

Detailní popis výběru se nachází ve specifikaci JVM [8, str. 387].

2.10 Výjimky

Výjimky jsou v JVM reprezentovány jako instance třídy `Trowable` a představují vyjímečnou (nebo také chybovou) situaci při výkonu instrukcí. JVM vyvolá výjimku v následujích případech:

- Je vykonána instrukce `athrow`.
- Byl detekován abnormální stav běhu JVM.
- Na instanci třídy vlákna byla zavolána metoda `stop`.
- Při výkonu JVM se vyskytla jiná chyba vnitřního charakteru, např. přetečení zásobníku rámců, nedostatek volné paměti, nezdařené rozřešení symbolických referencí atp.

Každá metoda může mít přidružené žádné nebo více handlerů, které specifikují, jak se má výjimka určitého typu ošetřit, pokud je vyvolána v určitém úseku kódu metody.

Detailní popis chování výjimek je ve specifikaci JVM [8, str. 23-25].

2.11 Knihovny, kompatibilita a přenositelnost

Aby mohl být zdrojový kód přenositelný mezi platformami, musí JVM poskytkout dostatečnou podporu pro implementaci tříd a rozhraní platformy Java. Některé z těchto tříd potřebují pro správné fungování těsnou spolupráci s JVM, jde hlavně o třídy:

- reflexe, například třídy z package `java.lang.reflect` a třída `Class`.
- třídy pro uživatelské načítání tříd a rozhraní, např. `ClassLoader`
- vícevláknový běh programu, např. `Thread`.
- výjimky, např. `Trowable`, `Exception`.

Specifikace JVM umožnujě rozdílné způsoby implementace, je však zaručeno, že zdrojový soubor, jehož binární formát se řídí specifikací (nebo ho lze do něj převést), lze spustit s očekáváným výsledkem na jakékoli implementaci, jenž se také řídí specifikací a implementuje všechny použité třídy (konkrétně jejich API) standartní knihovny platformy Java. Toto platí bezvýhrady v rámci zdrojových souborů určených pro konkrétní verzi Javy. Specifikace nových verzí platformy Java se snaží zachovát zpětnou kompatibilitu,

tzn. že zkompilované zdrojové soubory lze zpouštět i na všech novějších verzích JVM. Binární formát je vždy zpětně kompatibilní, u knihoven Javy ale existují výjimky, ty jsou však marginální a lze je dohledat ve specifikacích.

Existuje i problém tzv. binární kompatibility mezi zdrojovými soubory. Pokud zkompilovaný program využívá rozhraní určité knihovny (také zkompilované), je možné tuto knihovnu upravit, znovu zkompilovat a nahradit starou verzi novou. Aby však nedošlo k porušení kompatibility a zdrojové sobory využívající tuto knihovnu byly i nadále spustitelné, je třeba dodržet přesně definovaná pravidla při úpravě této knihovny. Tato pravidla jsou uvedena ve specifikaci Javy, kde je i detailně popsán problém binární kompatibility [4, str. 399–427].

3 Existující emulátory a debuggery

V této práci jsem prozkoumal některé existující emulátury a debuggery s primárním cílem zjistit, jaké informace poskytují a také jejich uživatelskou přívětivost. Prostudoval jsem konkrétně integrované vývojové prostředí IntelliJ IDEA, které poskytuje debugger programů napsaných v Javě (mimo jiné), dále grafický simulátor procesorové architektury MIPS QtSPIM a také simulátor Ripes pro procesorovou architekturu instrukční sady RISC V.

Existují také dva projekty, které mají podobný účel jako mnou vytvářený vizuální interpreter, tj. zobrazit běh programu v bajtkódu Javy se zobrazením paměťových struktur. Jde o JSwat[1], což je grafický frontend pro JVM debugger, který se ale od roku 2013 nevyvíjí, a plugin Bytecode Visualizer[2] do integrovaného vývojového prostředí Eclipse umožnující debugging samotného bajtkódu uvnitř Eclipse, který již ale také ukončil vývoj a nelze ho nainstalovat na žádné novější verze Eclipse. Z těchto důvodů se o těchto nástrojích pouze zmiňuji a nijak dál se jimi nezabývám.

3.1 IntelliJ IDEA

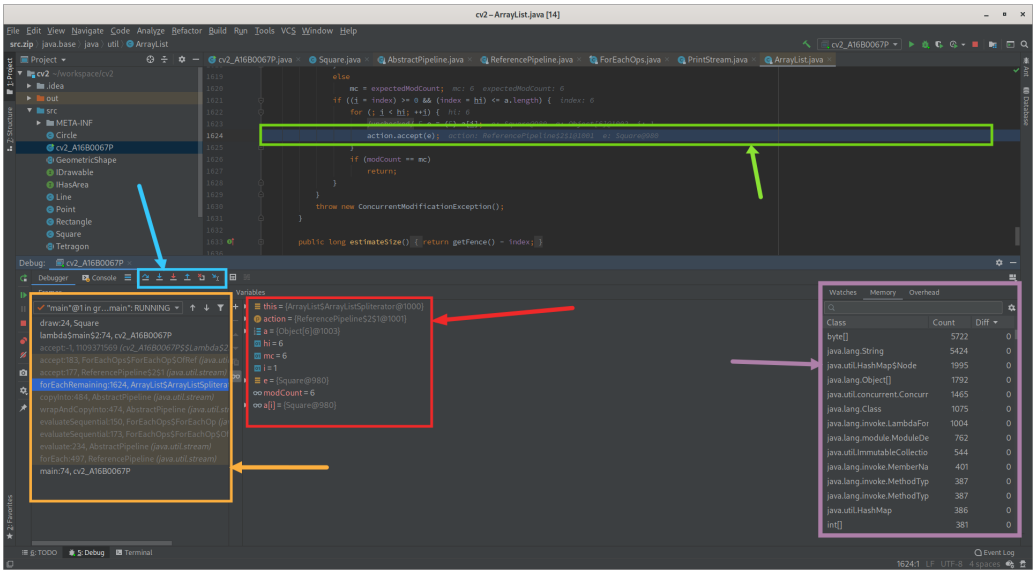
IntelliJ IDEA [6] je integrované vývojové prostředí pro Javu a jiné jazyky běžící na JVM. Poskytuje nástroj pro debugging programů, který je primárně zaměřen na jazyk Java.

Ukázka zobrazení, které poskytuje, je na screenshotu 3.1. V okně je vidět zdrojový text se zvýrazněným aktuálně vykonávaným řádkem (na obr. vyznačeno zeleně), zásobník rámců s rámci označenými jménem třídy a metody společně s hodnotou PC registru (žlutě), obsah lokálních proměnných (červeně), přehled načtěných tříd (fialově) a také panel s tlačítky pomocí kterých se ovládá běh debuggeru (modře). Při běhu programu je vidět, jak se mění hodnoty lokálních proměnných a rámce na zásobníku rámců. Lze přepínat mezi jednotlivými rámci, přičemž se pokaždé zobrazí pole lokálních proměnných pro daný rámec. Po rozkliknutí třídy ze seznamu načtených tříd se zobrazí zobrazí seznam jejích instancí (obr. 3.3). Dále je také možné se

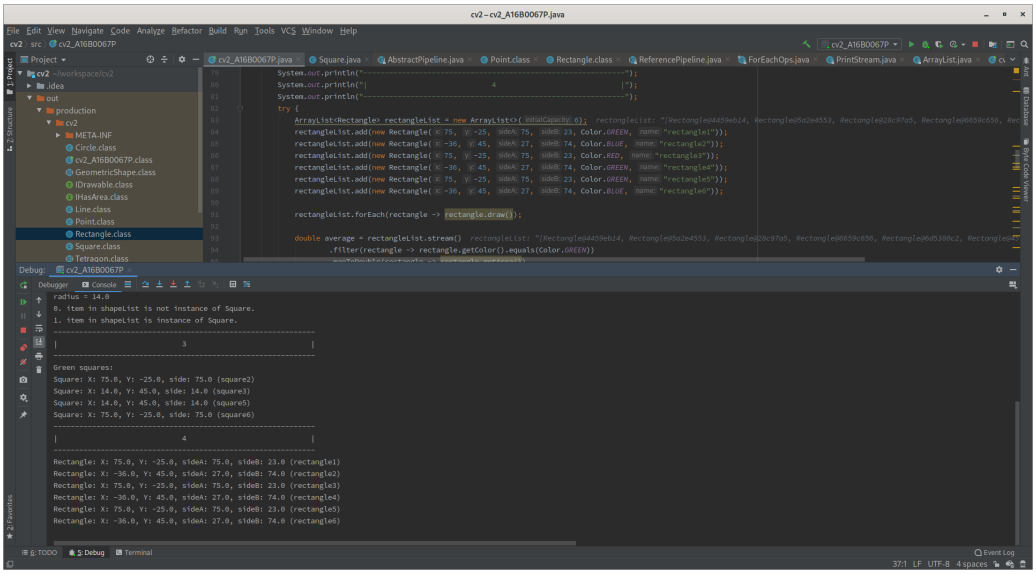
[1] JSwat – `https://github.com/nlfiedler/jswat`

[2] Bytecode Visualizer – `https://marketplace.eclipse.org/content/bytecode\-visualizer`

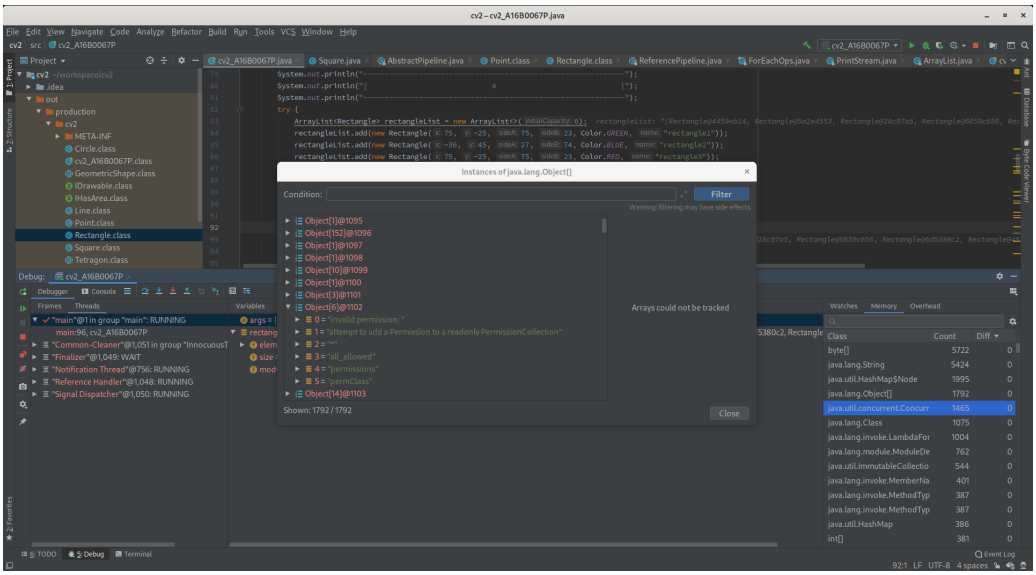
přepnout do panelu konzole, na které se vypisují data standartního výstupu a funguje i jako standartní vstup (obr. 3.2).



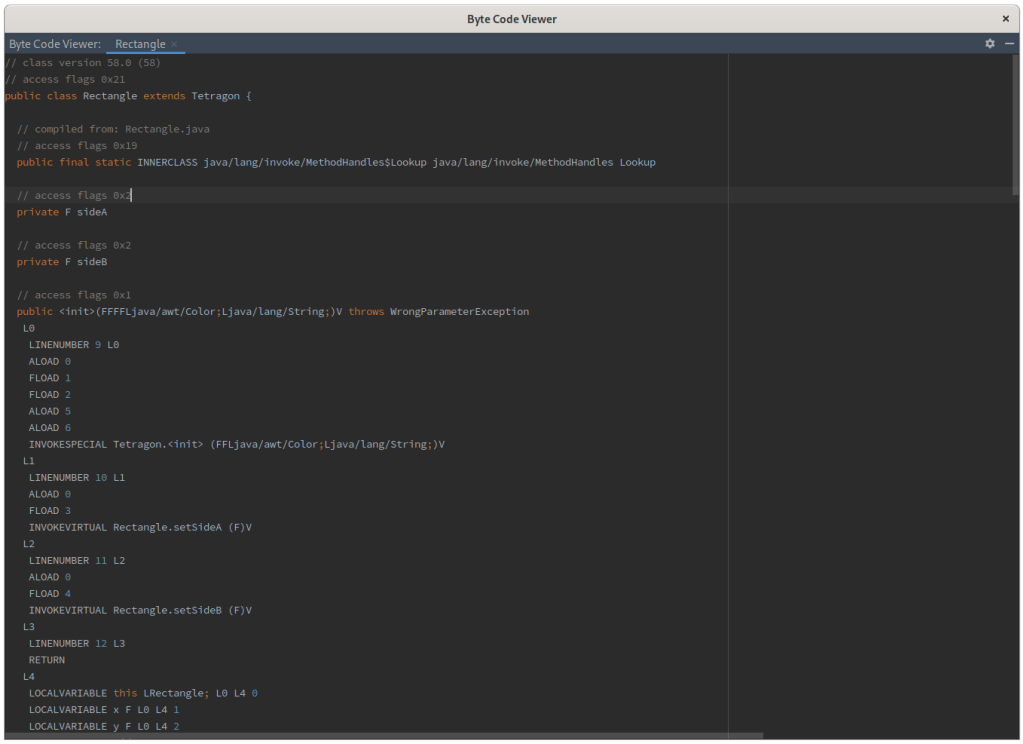
Obrázek 3.1: Ukázka zobrazení debuggeru IntelliJ IDEA



Obrázek 3.2: Ukázka zobrazení debuggeru IntelliJ IDEA – konzole



Obrázek 3.3: Ukázka zobrazení debuggeru IntelliJ IDEA – instance třídy



Obrázek 3.4: Ukázka zobrazení bajtkódu v IntelliJ IDEA

Debugging na úrovni bajtkódu k dispozici není, IntelliJ IDEA ale posky-

tuje nástroj pro zobrazení bajtkódu ze zdrojových souborů (obr. 3.4) a také dekompilaci zpět do zrojového textu Javy.

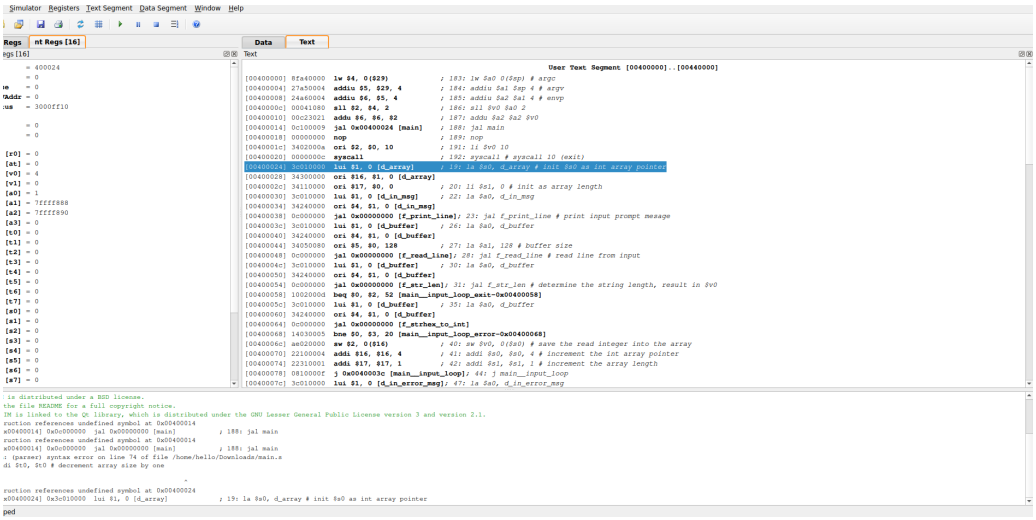
Pro zobrazení běhu programu v Javě se jedná o užitečný nástroj. Zobrazuje všechny potřebné informace, avšak pro detailnější zkoumání běhu na úrovni samotného bajtkódu se nehodí vzhledem k tomu, že zobrazuje průběh programu pouze na úrovni jazyka Javy.

3.2 QtSPIM

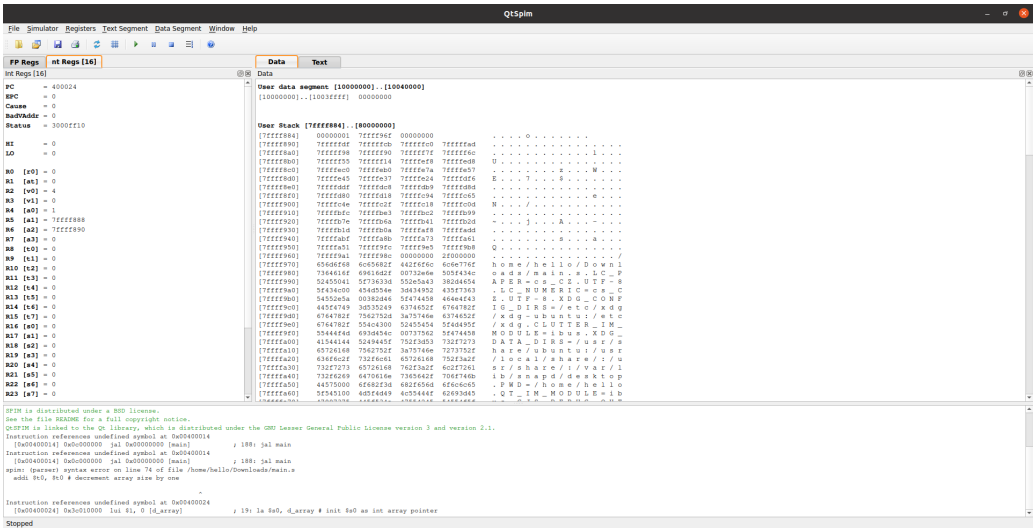
QtSPIM [7] je simulátor pro programy napsané v jazyce symbolických adress pro procesorovou architekturu MIPS32. Poskytuje také jednoduchý debugger a sadu základních systémových volání.

Jeho grafické rozhraní (obr. 3.5) je velice jednoduché. V prostřední části se zobrazuje celý zdrojový text programu nebo lze přepnout na zobrazení paměti (obr. 3.6). V levé části okna je zobrazený seznam registrů s jejich hodnotami a ve spodní části okna je konzole sloužící pro standartní vstup a výstup. V horní části okna je panel sloužící k ovládání programu a samotnému běhu. Při běhu se zvýrazňuje aktuálně vykonávaná instrukce a také je vidět měnící se hodnoty registrů.

Zobrazení paměti je puze tabulka s číselnými honotami na přísušných adresách paměti. Vedle této tabulky je zobrazení číselných dat převedených na znaky v kódování ASCII. Zásobník, s kterým je možné pomocí instrukční sady pracovat, nemá v zobrazení žádné zvláštní postavení, je pouze součástí zobrazené paměti. Takovéto zobrazení paměti je možná příliš jednoduché, ale vzhledem k povaze instrukční sady MIPS je odpovídající a postačující.



Obrázek 3.5: Ukázka zobrazení QtSPIM – program



Obrázek 3.6: Ukázka zobrazení QtSPIM – oblast paměti

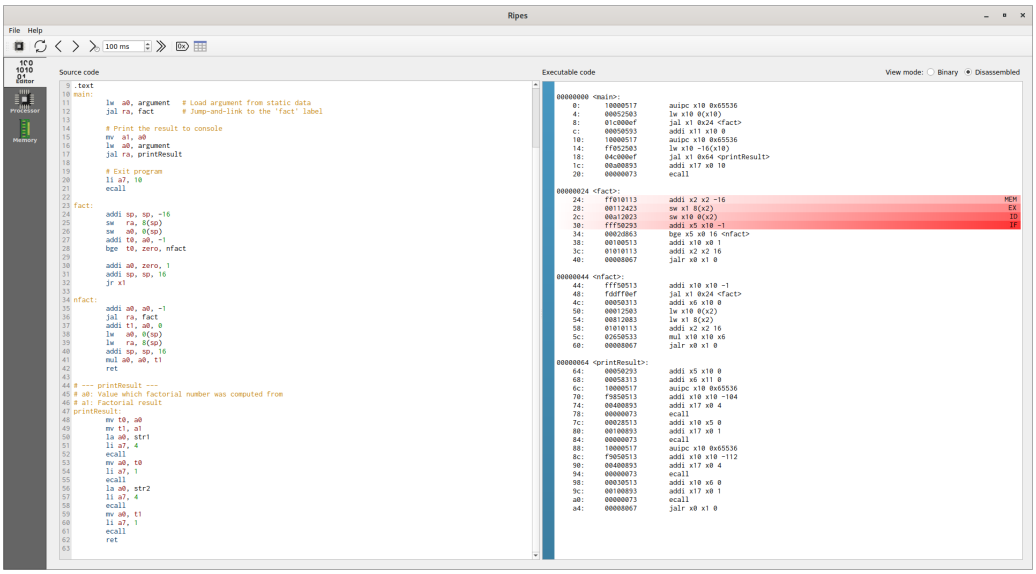
3.3 Ripes

Ripes [9] je simulátor procesorové architektury pro instrukční sadu RISC V. Poskytuje také jednoduchý editor zdrojových textů napsaných v jazyce symbolických adres.

Grafické rozhraní a způsob zobrazení informací je propracovanější než např. u výše popsaného QtSPIMu. V horní části okna se nachází ovládání

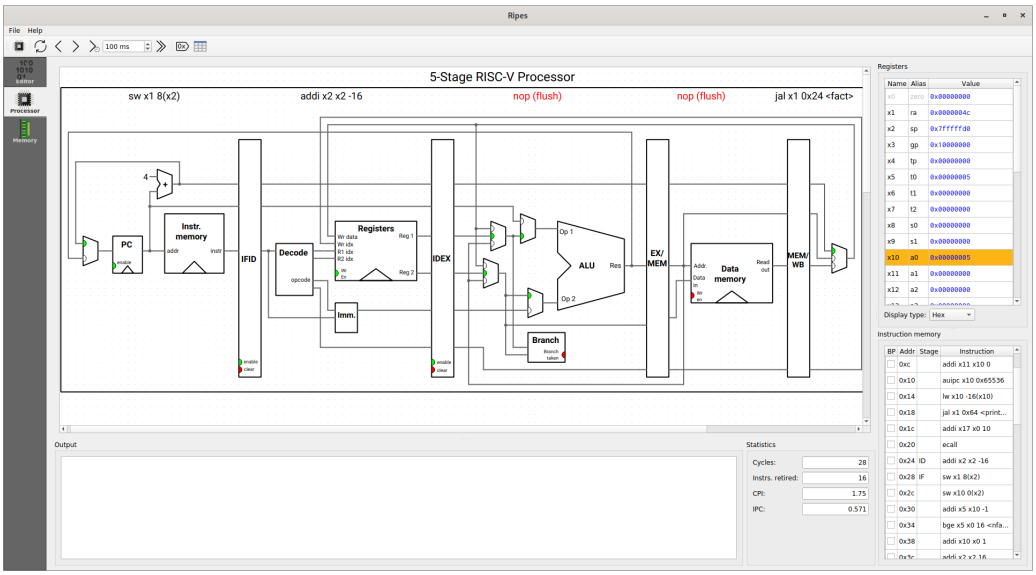
běhu programu, pomocí něhož lze vykonávat další instrukce, nebo spustit automatický běh programu s přednastavenými časovými rozetupy mezi výkonem jednotlivých instrukcí. Okno je rozděleno na tři přepínatelné panely, jejichž zobrazení zabírá celé okno.

První panel s názvem Editor je rozdělen do dvou hlavních částí (obr. 3.7). Nalevo je editor zdrojového textu programu a napravo výsledné přeložené a spustitelné instrukce. Při výkonu instrukcí se zvýrazňuje aktuálně vykonávaá instrukce společně s několika předchozími vykonanými.



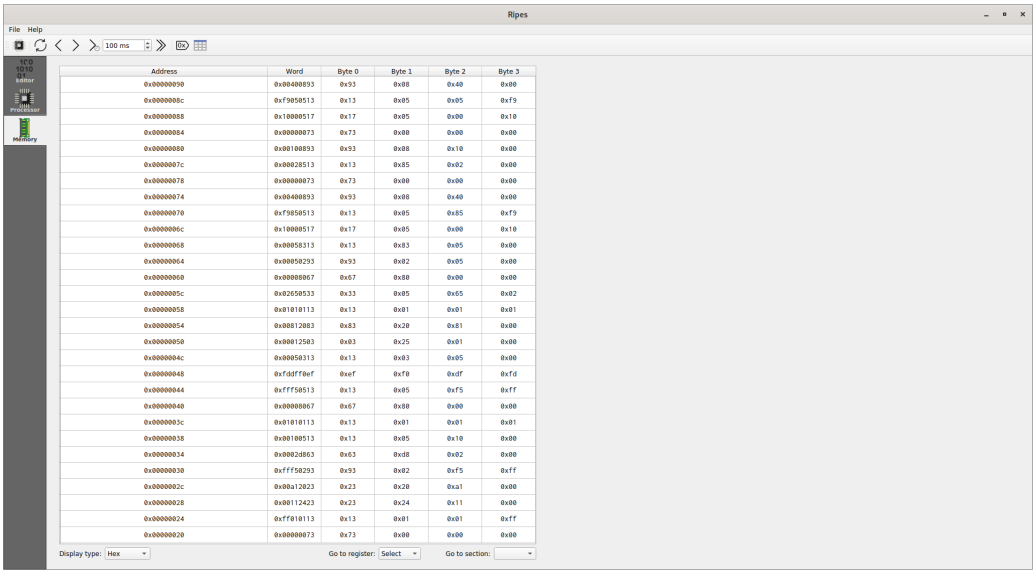
Obrázek 3.7: Ukázka zobrazení Ripes – editor zdrojového textu a spustitelný kód

Při přepnutí do panelu s názvem Processor se zobrazí interaktivní schéma hardwarové struktury procesoru, napravo okénko zobrazující hodnoty registrů a okénko s historií vykonaných instrukcí a ve spodní částí je konzole sloužící pro standartní vstup a výstup (obr. 3.8). Při průběhu programu se zvýrazňují vnitřní pochody uvnitř procesoru a také změny hodnot registrů.



Obrázek 3.8: Ukázka zobrazení Ripes – procesor

Panel s názvem Memory slouží pro zobrazení oblasti pamětí jako celku (obr. 3.9). Pamět je zobrazena jako tabulka hodnot na příslušných adresách. Zobrazení hodnot lze přepínat mezi číselným zobrazením v různých soustavách nebo textovým zobrazením v ASCII kódování.



Obrázek 3.9: Ukázka zobrazení Ripes – oblast paměti

3.4 Shrnutí

Grafické emulátory a debuggery zobrazují dostatek informací o průběhu programu vzhledem k charakteru cílového jazyka. Emulátory/debuggery jazyků s vyšší abstrakcí zobrazují pouze informace relevantní pro danou úroveň abstrakce a nezatěžují uživatele s konkrétními detaily implementace. Například IntelliJ IDEA zobrazuje instance tříd jako celky, kdežto emulátory nízkoúrovňových jazyků, jako je např. QtSPIM pro instrukční sadu MIPS pojem instance a objekt neznají a zobrazují surová data. Simulátor Ripes, který má propracovanější grafické prostředí, zobrazuje i schéma a fungování hardwarových součástí.

Pro návrh vizuálního interpreteru (který následuje v další kapitole) je třeba se držet takového grafického zobrazení, které je dostatečně informativní, ale nezatěžuje uživatele přebytečnými detaily.

4 Návrh interpreteru Java bytecode

V rámci této práce jsem navrhl vizuální interpreter bajtkódu Java. Návrh jsem provedl s ohledem na rozsáhlost celé instrukční sady, komplexity JVM a budoucímu využití této práce. Cílem této práce nebylo vytvořit kompletní implementaci JVM, ale pouze virtuální stroj s jednoduchým grafických zobrazením výkonu programu a určitou funkcionalitou podobnou JVM pro výukové potřeby. Pro jednoduchost použití jsem navrhl i vlastní formát zdrojových souborů.

V následujících částech práce je popsána zvolená funkcionalita, podmnožina instrukcí, návrh struktury virtuálního stroje, grafického rozhraní a formátu zdrojových souborů.

4.1 Funkcionalita

Ve virtuálním stroji musí být možno napsat funkčí jednoduchý program. Stroj musí alespoň základně napodobit fungování JVM. Proto jsem zvolil základní fukcionalitu ve formě aritmetických operací, podmíněných skoků, práci se třídami a jejich instancemi, avšak bez dědičnosti, polymorfismu a řízení přístupu, dále pak volání a návraty z metody a také práci se zásobníke operandů a polem lokálních proměnných. Podporované datové typy jsou stejné jako v JVM, kromě vyřazených typů `char`, `boolean`, `byte`, `short` a `array`. Jde tedy o typy `int`, `long`, `float`, `double` a `reference`.

Interpreter nebude podporovat dědičnost, polymorfismus, abstrakci ani rozhranní.

Instanční metody se speciálním jménem `<init>` a inicializační metody třídy `<clinit>` se volají vždy programem, nikdy virtuálním strojem, instrukcemi - INVOKEVIRTUAL, respektive INVOKESTATIC.

4.2 Podmnožina instrukcí

Pro navrhovaný interpreter jsem zvolil takovou podmnožinu instrukcí, která je dostatečná pro navrženou funkcionalitu. Základní instrukce umožnující samotnou logiku programu jsou aritmetické instrukce a instrukce podmíněných skoků a větvení. V JVM jsou základní stavební jednotky programu

třídy, společně s jejich atributy a metodami. Bylo tedy třeba zvolit i instrukce, které umožňují práci s nimi. Dále jsou zapotření instrukce pro práci se zásobníkem operandů, na nějž se načítají operandy pro aritmetické instrukce a jejich výsledky, a s lokálními proměnnými, do kterých lze ukládat mezivýpočty.

Mnou zvolené instrukce chováním až na výjimky odpovídají instrukcím JVM. V tabulce 7.1 v příloze na straně iii jsou tyto instrukce, jejich parametry a chování rozdělené do skupin podle použití.

Všechny instrukce načítající hodnoty ze zásobníku operandů (dále jen zásobník) tyto hodnoty zároveň i odeberou. Vzhledem k tomu, že jsou položky zásobníku brány jako položky o velikosti výpočetního typu 1, tak i instrukce přímo manipulující se zásobníkem k němu tak přistupují (tzn. např., že instrukce `POP2`, která odstraní dvě položky z vrcholu zásobníku, může odstranit buď dvě hodnoty výpočetního typu 1, nebo jednu hodnotu výpočetního typu 2). Pokud jsou tyto instrukce použity tak, že jejich by efekt ovlivnil pouze část hodnoty výpočetního typu 2, jedná se o chybu. Všechny aritmetické instrukce, pokud není řečeno jinak, berou hodnotu pro výpočet z vrcholu zásobníku a výsledek ukládají zpět na zásobník. Instrukce načítající hodnoty typu `byte` nebo `short` na zásobník nebo do lokálních proměnných tyto hodnoty nejdříve rozšíří na typ `int` se zachováním znaménka.

Některé instrukce mohou přijímat parametry:

- čísla (4.3.10):
  - `u8`,
  - `i8`,
  - `i16`,
  - `i32`,
  - `i64`,
  - `f32`,
  - `f64`,
  - `f64`,
- symbolické reference (4.3.11):
  - `class_ref`,
  - `field_ref`,
  - `method_ref`,

4.3 Formát zdrojových souborů

Pro snadnou editaci zdrojových souborů jsem navrhl textový fomát. Pro kódovaní znaků je použité UTF-8[1].

Soubor je rozdělen na definice složené z jednoho nebo více řádků. Tyto definice mohou být název třídy, definici atributu, nebo definici metody. Každý řádek může mít jednu nebo více položek.

Jako první definice musí být vždy uvedeno celé jméno třídy (4.3.1) a poté následují definice atributů (4.3.2) a metod (4.3.3), jejich pořadí nerozhoduje.

Řádky mohou být ukončeny buď znakem `\n` (U+000A) nebo posloupností znaků `\r` (U+000D) a `\n`. Jednotlivé řádky mohou být od sebe odděleny libovolným počtem prázdných řádků a jednotlivé položky jednoho řádku musí být odděleny alespoň jednou mezerou nebo tabulátorem `\t` (U+0009) a tyto položky v sobě nesmí obsahovat ani jeden z těchto znaků.

Pokud se kdekoliv v souboru objeví posloupnost dvou znaků `\\`, jsou tyto dva znaky a všechno co následuje po nich až do konce řádku ignorovány jakožto komentáře.

Zdrojové soubory musí končit příponou `.mvm`.

Následující text je příklad zdrojového souboru. Jedná se o třídu s vlastním jménem `Circle` uvnitř package `shape`, která je uvnitř package `geometry`. Třída má jeden statický atribut `pi` a jeden atribut instance `radius`. Dále obsahuje tři metody. Metoda `<clinit>` je inicializační metoda třídy, která nastaví počáteční hodnotu `pi` na `3.14159` a potřebuje pole lokálních proměnných o velikosti 0. Metoda `<init>` je inicializační metoda instance přebírající jako argument jednu hodnotu typu `float`, kterou uloží do atributu `radius`, a potřebuje pole lokálních proměnných o velikosti 2 (`this` a první parametr). Metoda computeArea je metoda instance s návratovou hodnotou `float`, která spočte obsahu kruhu reprezentovaného instancí třídy `Circle`.

[1]UTF-8, a transformation format of ISO 10646 – `https://tools.ietf.org/html/rfc3629`

```
geometry.shape.Circle

FIELD
static double pi

FIELD
float radius

METHOD
static void <clinit> () 0
LDC2_W      3.14159
PUTSTATIC   double geometry.shape.Circle pi
END

METHOD
void <init> (float) 2
ALOAD_0
FLOAD_1
PUTFIELD    float geometry.shape.Circle radius
END

METHOD
static float computeArea () 1
GETSTATIC   double geometry.shape.Circle pi
ALOAD_0
GETFIELD    float geometry.shape.Circle radius
F2D
DUP
DMUL
DMUL
D2F
FRETURN
END
```

4.3.1 Definice jména třídy

Řádek s definicí jména třídy obsahuje pouze jednu položku představující celé jméno třídy (4.3.4).

4.3.2 Definice atributu

Definice atributu začíná řádkem obsahujícím klíčové slovo `FIELD`. Následující řádek je definice samotného atributu. Ta se skládá z nepovinného klíčového slova `static`, které značí, že se jedná o statický atribut, jinak se jedná o atribut instance. Dále následuje datový typ atributu (4.3.7) a poté jméno atributu (4.3.5).

4.3.3 Definice metody

Definice metody začíná řádkem obsahujícím klíčové slovo `METHOD`. Následující řádek je definice samotné metody. Ta se skládá z nepovinného klíčového slova `static`, které značí, že se jedná o statickou metodu, jinak se jedná o metodu instance. Dále následuje návratový typ metody (4.3.8), jméno metody (4.3.5), parametry metody (4.3.9) a nakonec číslo typu `u8` značící potřebnou velikost pole lokálních proměnných pro danou metodu.

Následující řádky definují instrukce metody (4.3.11).

Celá definice metody musí být zakončena řádkem obsahujícím klíčové slovo `END`.

4.3.4 Jméno třídy

Plně kvalifikované jméno třídy je složeno ze jména rodičovské package třídy a vlastního jména třídy. Jméno package se skládá rekurzivně ze jméno rodičovské package a samotného jména package. Jednotlivá jména jsou oddělena tečkou (`.`). Tato jména musí obsahovat alespoň jeden znak a nesmí obsahovat znaky `. ; [ / ,`. Dále nesmí být jméno třídy stejné jako název datového typu (4.3.7), pokud ano, nelze tuto třídu v kódu nijak odkazovat. Příklad celého jména třídy:

```
package.subpackage.Trida
```

4.3.5 Jméno atributu

Jméno atributu musí obsahovat alespoň jeden znak a nesmí obsahovat znaky `. ; [ /`.

4.3.6 Jméno metody

Jméno metody musí obsahovat alespoň jeden znak a nesmí obsahovat znaky `. ; [ / < >`. Výjimkou jsou speciální metody se jmény `<init>` a `<clinit>`.

4.3.7 Datový typ

Datový typ může být jeden z následujících řetězců:

- `int`,
- `long`,
- `float`,
- `double`,
- nebo jméno třídy (4.3.4),

4.3.8 Návratový typ

Návratový typ může být jeden z následujících řetězců:

- `void`,
- nebo datový typ (4.3.7),

4.3.9 Parametry metody

Parametry metody musí být řetězec začínající znakem `(` a končící znakem `)` (musí být uzavřeny v závorkách). Mezi závorkami je obsažena posloupnost datových typů (4.3.7), které jsou navzájem odděleny čárkou (`,`). Jako u jiných položek nesmí řetězec nikde obsahovat mezeru nebo tabulátor.

Příklad parametrů metody:
`(int,float,long,ClassName)`

4.3.10 Literály čísel

Ve zdrojových souborech se používají literály čísel, především jako parametry některých instrukcí, ale například také jako číslo značící velikost pole lokálních proměnných pro danou metodu. Všechna čísla musí být v desítkové soustavě a výsledné číslo se musí vejít do cílového typu. Jde o následující typy:

- celočíselné typy – posloupnost číselných znaků (`0` – `9`) s volitelným znaménkem na začátku (`+` nebo `-`, nebo pouze `+` u nezáporných typů).
  - `u8` – nezáporné 8–bitové číslo (0 – 255)

  - `i8` – 8–bitové číslo se znaménkem (-128 – 127), odpovídá datovému typu `byte`
  - `i16` – 16–bitové číslo se znaménkem (-32768 – 32767), odpovídá datovému typu `short`
  - `i32` – 32–bitové číslo se znaménkem (-2147483648 – 2147483647), odpovídá datovému typu `int`
  - `i64` – 64–bitové číslo se znaménkem (-9223372036854775808 – 9223372036854775807), odpovídá datovému typu `long`

- číselné typy s plovoucí řádovou čárkou – posloupnost číselných znaků (`0` – `9`) následovaná tečkou (`.`) s volitelným znaménkem na začátku (`+` nebo `-`). Toto je poté volitelně následované další posloupností číselných znaků a volitelným exponentem. Exponent začíná znakem `e` nebo `E` a následuje posloupnost číselných znaků s volitelným znaménkem na začátku. Pro daný literál se vytvoří nejbližší možná reprezentace v daném datovém typu. Dále jsou akceptované literály `inf` a `-inf` pro nekonečna a `NaN` pro not-a-number.

  - `f32` – odpovídá datovému typu `float`
  - `f64` – datovému typu `double`

  příklady: `3.14  -3.14  2.5E10  5.  inf`

4.3.11 Instrukce

Každá instrukce je definovaná na jednom řádku. Definice se skládá z názvu instrukce velkými písmeny a poté z argumentů instrukce, což mohou být podle konkrétní instrukce číselné literály, symbolické reference, nebo také žádné argumenty. Parametry jsou odděleny mezerami nebo tabulátory.

Symbolické reference jsou na řádku definovány následovně:

- `class_ref`: reference na třídu: jméno třídy (4.3.4),
- `field_ref`: reference na atribut: typ atributu (4.3.7) následovaný jménem třídy a jménem atributu (4.3.5)
- `method_ref`: reference na metodu: návratový typ metody (4.3.8) následovaný jménem třídy, jménem metody (4.3.6) a parametry metody (4.3.6)

Příklady:

```
LDC2_W          3.14159
BIPUSH          10
ALOAD_0
GETFIELD        float geometry.shape.Circle radius
INVOKE_STATIC   float geometry.shape.Circle computeArea ()
INVOKE_VIRTUAL  Rectangle Rectangle scale (double,double)
```

4.3.12 Adresářová struktura

Zdrojové soubory tříd se pro potřebu virtuálního stroje načítají z definovaných kořenových adresářů tak, že jména packagů představují podadresáře a vlastní jména tříd doplněná o příponu `.mvm` jména zdrojových souborů.

4.4 Struktura virtuálního stroje

Virtuální stroj je vnitřně rozdělen do několika vzájemně propojených struktur, které slouží jako oblasti paměti, vykonavatelé logiky a samotné programové struktury. UML diagram návrhu struktur je na obrázku 4.1.

Struktury `Class`, `Field`, `Method` a `Code` jsou reprezentace definic načtených tříd, jejich atributů, metod a také instrukcí. Tyto struktury jsou ukládány ve struktuře `Heap` (halda), stejně tak i instance tříd, které jsou reprezentovány strukturou `Instance`. Každá instance obsahuje pole hodnot atributů a také referenci na svou třídu. O načítání zdrojových souborů do jejich běhových reprezentací se stará `ClassLoader`.

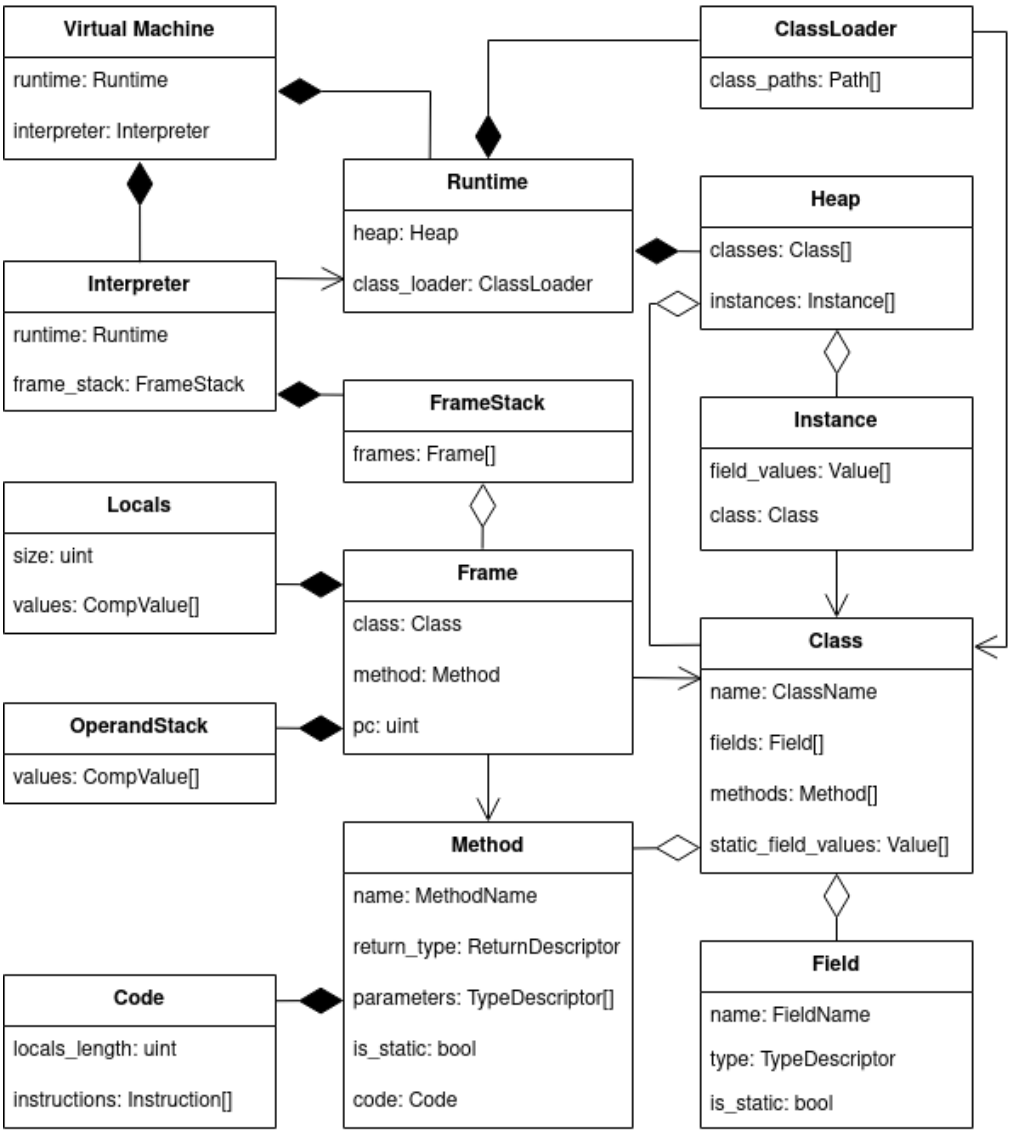
`ClassLoader` i `Heap` jsou zakomponovány ve struktuře `Runtime`, které se stará o propojení logiky class loaderu a haldy – poskytuje rozhraní pro transparentní získání třídy nebo instance.

`Interpreter` provádí samotný výkon instrukcí metod. Obsahuje strukturu `FrameStack` (zásobník rámců), na nějž se ukládají jednotlivé rámce volaných metod. Ty jsou reprezentovány strukturou `Frame`. Každý rámec má svoje vlastní pole lokáních proměnných `Locals` a zásobník operandů `OperandStack`. Dále každý rámec drží informaci o vykonávané metodě, třídě této metody a aktuálně vykonávané instrukci.

Hlavní struktura `VirtualMachine` zastřešuje celý virtuální stroj. Má v sobě zakomponované struktury `Runtime` a `Interpreter` a poskytuje rozhraní pro přístup a ovládání virtuálního stroje.

Paměťovou oblast constant pool jsem v návrhu vůbec nepoužil, constant pool totiž slouží jako úložistě informací pro binární formát souborů, které jsou po načtení uloženy přímo ve strukturách virtuálního stroje, nebo konstant sloužící jako argumenty instrukcí. V mnou navrženém textovém for-

mátu zdrojových souborů nic jako constant pool není třeba a načtené instrukce nesou hodnoty parametrů v sobě.



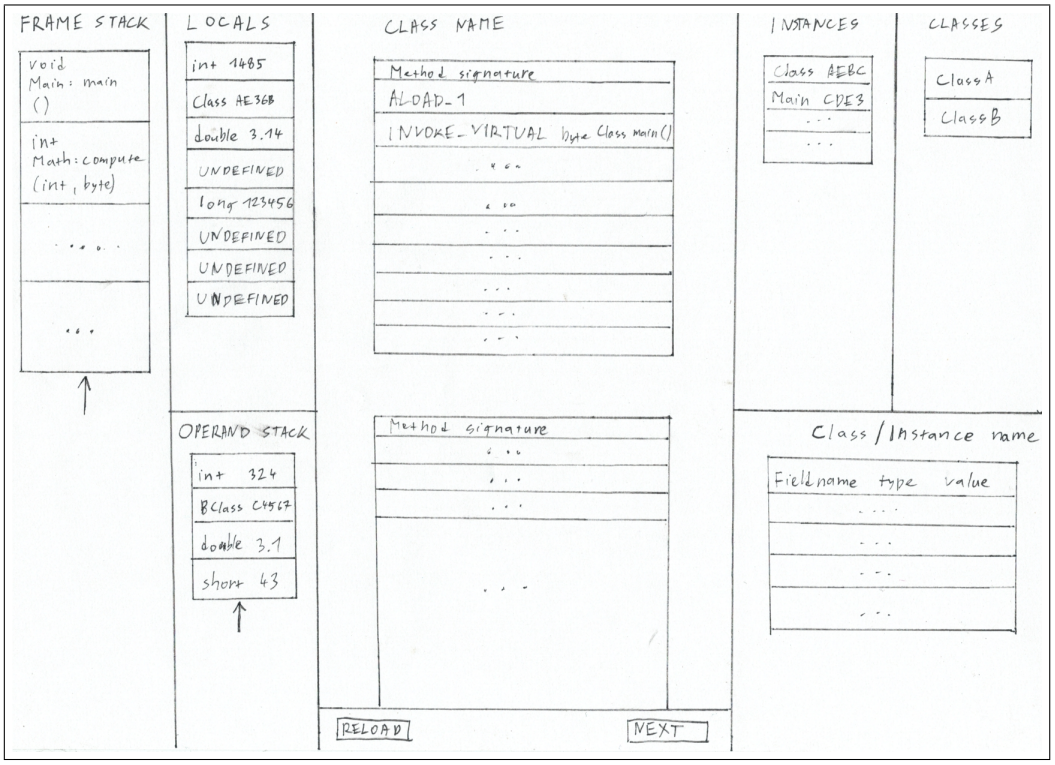
Obrázek 4.1: UML diagram návrhu struktury virtuálního stroje

4.5 Grafické rozhraní

Grafické rozhraní musí obsahovat ovládací prvky virtuálního stroje a dostatečné informace o průběhu progamu, tzn. o aktuálně vykonávaných instrukcích a jejich následcích v kontextu celého stroje. Je třeba zobrazit oblasti paměti, avšak kompaktním a přehledným způsobem.

Návrh grafického rozhraní je na obrázku 4.2. Uprostřed okna programu se nachází zobrazení všech metod jedné třídy a jejich instrukcí. V pravé horní části se nachází zobrazení haldy – instance a třídy ve dvou přehledných seznamech. Ve spodním pravém rohu se zobrazují názvy a hodnoty polí instance nebo statických polí třídy. Na pravé straně okna se nachází zobrazení zásobníku rámců, pole lokálních proměnných a zásobníku operandů.

Na spodní straně okna je panel s ovládacími prvky virtuálního stroje. Důležité je tlačítko po jehož stisknutí se vykoná další instrukce. Po vykonání instrukce se v prostřední části okna zobrazí třída aktuálně vykonávané metody. Příští vykonávaná instrukce je zvýrazněna. Všechny hodnoty nebo struktury ovlivněné předešlou vykonanou instrukcí se také zvýrazní.



Obrázek 4.2: Návrh grafického rozhraní

4.6 Propojení virtuálního stroje a grafického rozhraní

Grafické rozhraní musí reagovat na změny uvnitř virtuálního stroje a zobrazit je. Toto lze provést dvěma způsoby. Buď každá zobrazovaná struktura

bude mít zakomponovaný svůj vlastní mechanismus, pomocí kterého upozorní grafickou komponentu o změně svého stavu nebo bude tento mechanismus zakomponován pouze v interpreteru instrukcí bajtkódu, který při vykonání instrukce má znalost o všech změnách, jelikož je sám provádí. Druhou možnost využívá HotSpot – implementace JVM od Oracle – pro svůj JVM Tool Interface, což je nástroj pro debugging JVM. Tuto možnost jsem zvolil i já z důvodu snažší implementace virtuálního stroje. Virtuální stroj musí ale poskytnout dostatečné rozhraní jak pro své ovládání, tak i pro upozornění na změnu stavů svých vnitřních struktur.

5 Implementace

Implementoval jsem navržený vizuální interpreter podle předchozího návrhu. Podmnožina a chování instrukcí odpovídá návrhu, vnitřní struktura je však mírně odlišná.

Návod pro sestavení a spuštění programu je v příloze na straně i.

5.1 Použitý jazyk, knihovny a návody

Pro implementaci jsem použil programovací jazyk Rust [11] verze 1.45. Tento jazyk není zatím příliš rozšířený, ale zvolil jsem ho kvůli jeho vlastnostem jako jsou např. spolehlisvot a paměťová bezpečnost i ve vícevláknových programech. Dále je dostatečně nízkoúrovňový pro práci s pamětí a datovými typy, což bylo pro implementaci zapotřebí.

Při programování virtuálního stroje jsem se z malé části inspiroval implementací JVM HotSpot od Oracle [2], což mi pomohlo lépe pochopit fungování JVM.

Pro tvorbu GUI jsem zvolil knihovnu GTK [10] verze 3.24, je to momentálně knihovna s nejlepší podporou pro jazyk Rust[1] a zároveň patří mezi nejpoužívanější knihovny pro tvorbu GUI. Pro jazyk Rust poskytuje rozhraní ke GTK knihovna Gtk-rs [5], která slouží jako obalenení volání knihovních funkcí, a také knihovna Relm [1] poskytující abstrakci nad Gtk-rs ve formě MVC modelu (Model–View–Controller). GUI jsem vytvořil s pomocí tutoriálů dostupných u každé knihovny a také podle knihy Rust Programming By Example od autora knihovny Relm [3].

5.2 Struktura programu

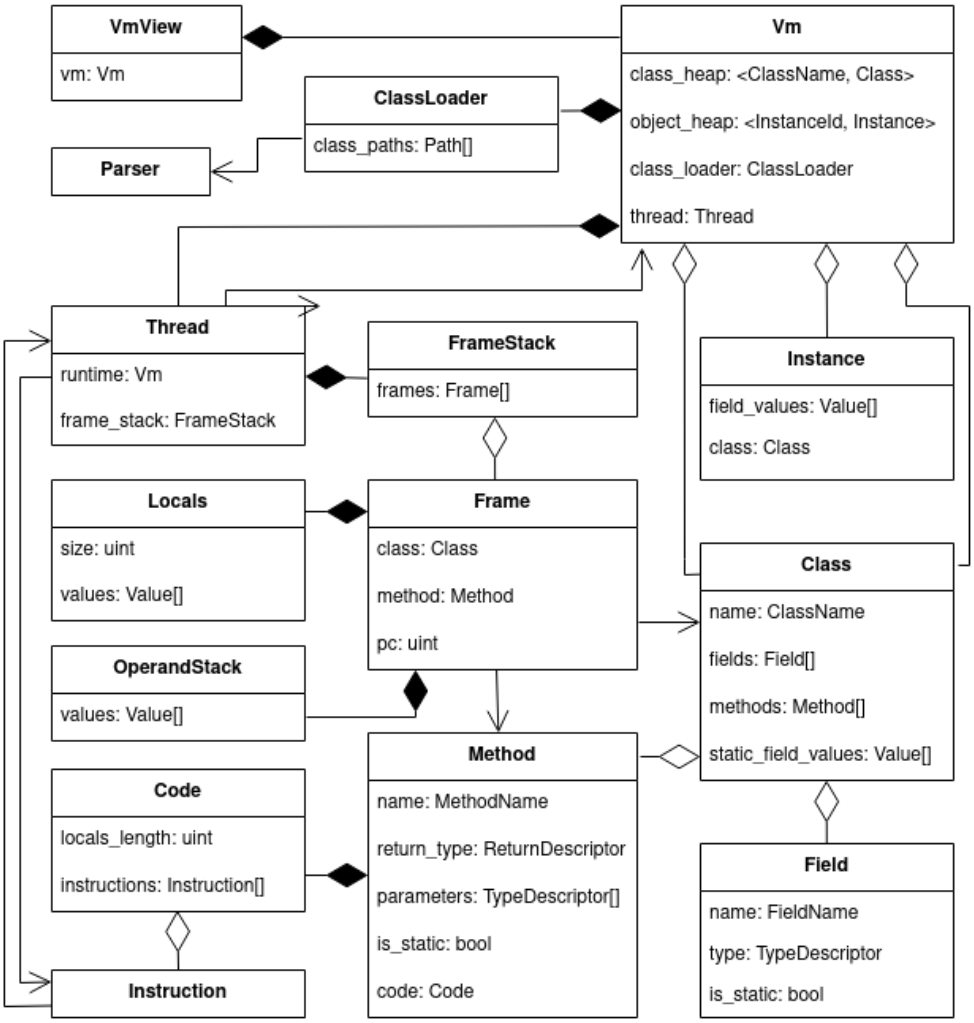
Zdrojové soubory programu jsou rozděleny do modulů následovně:

- `gui` – struktury grafického rozhranní
- `vm` – struktury a logika virtuálního stroje
  - `bytecode` – instrukce bajtkódu a jejich interpretace
  - `class` – reprezentace tříd, metod, atributů a deskriptorů

[1] The state of building user interfaces in Rust – `https://areweguiyet.com/`

– `exec` – hlavní struktury pro běh virtuálního stroje,
– `memory` – paměťové struktury, jako např. zásobník nebo lokání proměnné
– `parse` – parser a reprezentace zdrojových souborů
– `types` – definice datových typů

Na obrázku 5.1 je stručný UML diagram použitých struktur.



Obrázek 5.1: Stručný UML diagram implementace virtuálního stroje

5.2.1 Běh virtuálního stroje a propojení s GUI

Virtuální stroj je reprezentován strukturou `Vm`. Ta obsahuje haldu (5.2.4), class loader (5.2.2) a také strukturu `Thread`, která představuje vlákno pro-

gramu. Toto vlákno je spuštěno skutečně jako samostatné systémové vlákno a je možné mu pomocí definovaných funkcí předávat zprávy – `next` pro vykonání další instrukce a `cancel` pro zastavení běhu. Po vytvoření lze toto vlákno odstartovat pomocí funkce `start` na struktuře `Vm`. Na virtuálním stroji je také možné zavolat funkci `join`, která je blokující, dokud výkon virtuálního stroje neskončí.

Virtuálnímu stroji lze před startem nastavit zpětná volání, která se zavolají v případě že dojde k vnitřní změně, chybě nebo ukončení výkonu. Těchto volání využíva grafické rozhraní pro aktualizaci zobrazovaných informací.

`Vm` dále poskytuje funkce pro získání přístupu k vnitřním strukturám, např. k získání všech instancí, načtených tříd nebo zásobníku rámců, z kterým lze získat informace o stavu virtuálního stroje.

5.2.2 Načítání zdrojových souborů

Načítání zdrojových souborů obstarává struktura `ClassLoader`, která vyhledá, načtě a poté s pomocí struktury `Parser` naparsuje a vytvoří reprezentaci zrojových souborů, které potom převede na reprezentaci tříd, metod a atributů.

V hlavní třídě se vždy hledá metoda `static void main ()`, která je spuštěna jako první.

5.2.3 Datové typy

Datové typy jsou realizovány pomocí struktur `Int`, `Long`, `Float`, `Double` a `Reference`. Číselné datové typy drží přímo číselnou hodnotu, refernce drží ukazatel na instanci, nebo speciální hodnotu null.

Dále existuje obalující typ `Value`, což je `enum` typ s variantami pro každý ze základních datových typů. Ve virtuálním stroji se používá pro generické uchovávání hodnot ze zachování informace o příslušném datovém typu. Je použit například pro ukládání do pole hodnot atributů, do zásobníku operandů, nebo i do lokálních proměnných.

5.2.4 Paměť

Halda je obsažena přímo ve struktuře `Vm` a to ve formě dvou obyčejných hash tabulek, které ukládají třídy a instance pomocí jejich jména, respektive identifikačního čísla.

Zásobník rámců je realizován jako pole s měnitelnou velikostí obsahující ukazatele na vytvořené rámce. Ty lze přidávat na vrchol zásoníku, nebo je odebírat.

Každý rámec `Frame` má vlastní pole lokálních proměnných, zásobník operandů a hodnotu registru PC.

Zásobník operandů `OperandStack` je stejně jako zásobník rámců realizován polem s měnitelnou velikostí. I přestože prvky tohoto pole jsou hodnoty typu `Value`, tak vzhledem k tomu, že některé instrukce operací nad stackem jsou definovány podle typu hodnot, pro které se mohou použít, a jiné zase podle kategorie hodnot (velikost 1 nebo 2), provádí se při provádění těchto operací kontrola podle typu hodnot, respektive podle kategorií. Například operace `DUP_X2` kontroluje, zda je první hodnota na zásobníku kategorie 1 a poté zda je druhá hodnota kategorie 2, nebo se jedná o dvě hodnoty kategorie 1.

Pole lokálních proměnných `Locals` je realizováno jako pole pevné velikosti. Pro uchování informací o datových typech jsou jako prvky tohoto pole použity hodnoty typu `Slot`, které mohou být buď `Undefined`, nebo držet hodnotu typu `Value`. `Undefined` může znamenat dvě věci – že na tento index nebyla uložena zatím žádná hodnota, nebo že tento index je zabraný hodnotou druhé kategorie na předchozím indexu. Pokud se uloží hodnota kategorie 2 na index `i`, automaticky tato hodnota zabírá i index `i + 2`, který se označí jako `Undefined`. Pokud dojde k přepsání této hodnoty na indexu `i + 1`, je předchozí hodnota kategorie 2 na indexu `i` označena jako `Undefined`. Při uložení hodnoty kategorie typu 2 se kontroluje, zda existuje v poli i index `i + 1`, tedy zda je pro tuto hodnotu místo.

5.2.5 Interpretace instrukcí

Instrukce jsou reprezentovány pomocí `enum` typu `Instruction`. Pro každou instrukci exituje vlastní varianta toho `enum`, přičemž každá varianta si s sebou může nést dodatečné informace (parametry).

Interpretace instrukcí je realizována na struktuře `Instruction` pomocí velkého switche (v Rustu „match“) ve funkci `execute`, která přebírá jako argument stukturu `Thread`, nad níž vykonává dané operace. Pro přehlednost je logika pro každou instrukci vyjmuta do vlastní funkce, která se volá uvnitř tohoto switche.

Přidání instrukce znamená přidání varianty `enum`u `Instruction`, implementace výkonu této varianty uvnitř funkce `execute` a také přidání logiky pro parsování instrukce do struktury `Parser`.

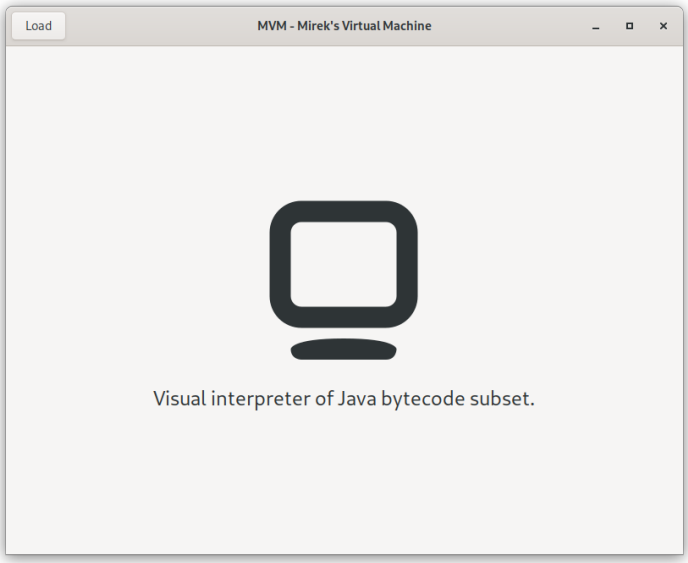
5.2.6 Grafické rozhraní

Grafické rozhraní je implementováno strukturami, které implementují trait (pozn. – podobný význam jako interface) `Widget` z knihovny Relm. Hlavní struktura je `AppWindow` která obsahuje hlavičku okna `AppHeader`. Po načtění zdrojových souborů se vytvoří náhled virtuálního stroje `VmView`, který se stará o komunikaci s virtuálním strojem a zastřešuje a notifikuje všechy ostatní náhledy konkrétních oblastí virtuálního stroje. Tyto náhledy jsou `FrameStackView`, který zobrazuje zásobník rámců, `LocalsView` zobrazující pole lokálních proměnných, `OperandStackView`, který zobrazuje zásobník operandů, `InstructionsView` pro zobrazení instrukcí metody, `ClassesView` a `InstancesView` pro zobrazení seznamů tříd a instancí a `FieldsView` pro zobrazení jejich atributů.

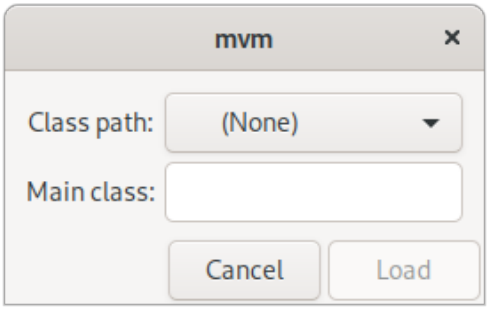
Každý z náhledů má nadefinované zprávy, které je mu možné zaslat nebo které může sám vyslat, a ostaní mu mohou naslouchat. Pomocí těchto zpráv zasílaných pomocí kanálů knihovny Relm lze předávat informace jednotlivým náhledům nebo komunikovat s jinými vlákny bez ztráty responzivity rozhraní („zamrznutí“).

5.3 Uživatelská příručka

Při spuštení programu se zobrazí úvodní obrazovka (obr. 5.2). Po kliknutí na tlačítko `Load` se zobrazí dialog pro načtení programu (5.3). Zde je třeba zvolit kořenový adresář spouštěného programu a také název hlavní třídy, ve které se bude hledat metoda `static void main ()`.



Obrázek 5.2: Úvodní obrazovka vizuálního interpreteru



Obrázek 5.3: Dialog pro načtení porgramu

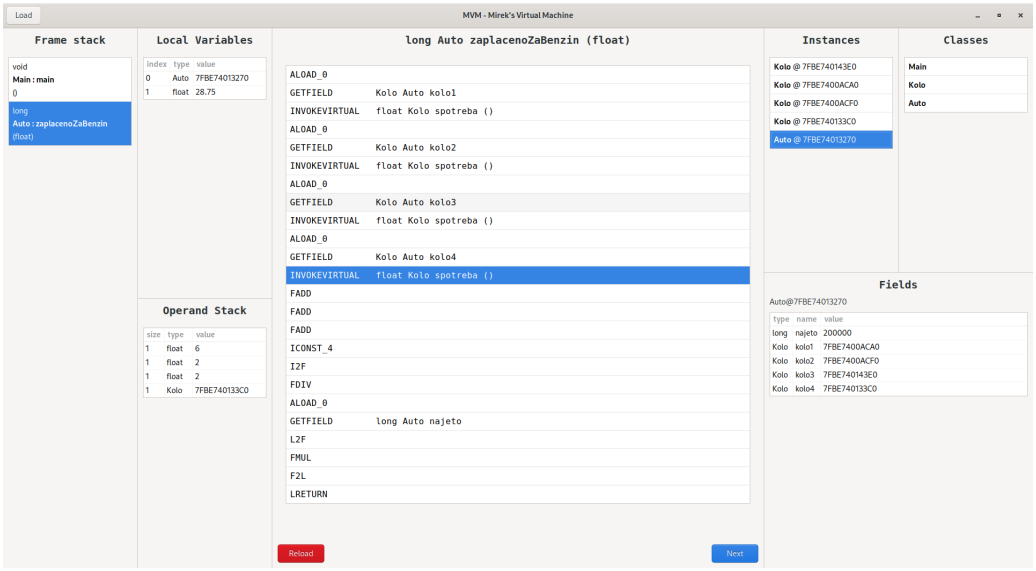
Po načtení programu se zobrazí okno s vizualizací virtuálního stroje. (obr. 5.4). Uprostřed jsou zobrazny instrukce aktuálně vykonávané metody se zvýrazněním příší vykonávané instrukce.

V levé části se nachází zobrazení zásobníku rámcům s popisem, pro kterou třídu a metody byl rámec vytvořen, pole lokáních proměnných s číslovanými pozicemi jedtnotlivých hodnot a také zásobník operandů s informací o velikosti každého operandu. Při kliknutí na rámec se zobrazí aktuální stav lokálních proměnných, zásobníku operandů a průběhu metody pro daný rámec.

V pravé horní čáti se nachází seznam vytvořených instací a seznam načtených tříd. Po kliknutí na třídu nebo instanci se ve spodní pravé části okna se zobrazí hodnoty instačních respektive statických atributů.

Ve spodní části okna je panel se dvěma tlačítky. Tlačítko `Reload` spustí program od začátku a tlačítko `Next` posune běh programu o jednu instrukci dále.

Při výskytu chyby v důsledku špatného formátu zdrojových souborů nebo nesprávném použití instrukce se zobrazí dialog s popisem chyby a výkon programu končí.



Obrázek 5.4: Vizualizace virtuálního stroje a běhu programu

6 Testování

Implementovaný vizuální interpreter jsem poskytl dvoum testerům k otestování. Žádný z nich se nikdy před tím nesetkal se samotným bajtkódem Javy. Poskytl jsem jim úvod do funkcionality programu a vzhledem ke komplexitě instrukční sady bajtkódu jsem pouze základní úvod do instrukční sady.

Připravil jsem pro ně testovací program v bajtkódu, který poté podle návodu spustili, otestovali a zodpověděli několik otázek.

6.1 Návod na testování a otázky

Vyzkoušejte si zapnout program MVM a spustit třídu `Main.mvm` uvnitř adresáře `src`. Proklikejte se až průběhem programu až nakonec a zkuste program spustit odznova. Během toho se seznamte s prostředím a zkuste odhadnout jak fungují některé instrukce.

Zkuste „rozbít“ zdrojový kód a zjistěte, jestli to má vliv na chod programu. Pokud nastane chyba, zjistěte, zda je z jejího popisu jasné, co bylo příčinou této chyby.

6.1.1 Otázky

- Bylo spuštění a ovládání intuitivní?
- Je ze zobrazení jasné co zobrazuje jakou oblast paměti?
- Je z vizualizace jasné, jaký efekt mají instrukce?
- Je něco co se vám líbí na grafickém rozhranní nebo naopak něco, co byste zlepšil?
- Vyskytla se během testování vnitřní chyba programu nesouvisející s intrepretovaným kódem?

6.2 První tester

První tester je student informatiky na vysoké škole. Má zkušenosti z programovacím jazykem Java a základní znalosti některých jazyků symbolických adres.

Bylo spuštění a ovládání intuitivní?

Spuštění a ovládání aplikace bylo jednoduché a srozumitelné. Uživatel si při vstupu vybere třídu, kterou si přeje spustit a aplikace bez dalších překážek třídu načte do grafického prostředí aplikace.

Je ze zobrazení jasné co zobrazuje jakou oblast paměti?

Aplikace nabízí velmi jednoduché grafické rozhraní, které obsahuje nezbytné prvky, které jsou rozumně rozmístěny po obrazovce. Jako další rozšíření aplikace by mohlo následovat grafické rozbrazení změny kroku, tj. uživateli je při každém kroku zvýrazněna změna, která se v daném kroku provedla.

Je z vizualizace jasné, jaký efekt mají instrukce?

Při průchodu algoritmem nahrané třídy lze snadno vidět aktualizované části grafického rozhraní, které specifikují příslušné informace. Ačkoli jde z názvu instrukcí relativně snadno určit jejich význam, tak by mohl přímo v aplikaci existovat jejich seznam z vysvětlením (dokumentace).

Je něco co se vám líbí na grafickém rozhranní nebo naopak něco, co byste zlepšil?

Jedná se o velmi přehledné uživatelské rozhraní. Jednoduchost je hlavním záměrem při tvorbě takovéto aplikace a rozhodně by se od toho nemělo odcházet.

Vyskytla se během testování vnitřní chyba programu nesouvisející s intrepretovaným kódem?

Ve snaze modifikace kódu jsem byl vždy řádně aplikací upozorněn ve chvílích, kdy byla interpretace kódu chybná. Při testování jsem nedošel do situace, kde by aplikace bez důvodu byla ukončena nebo nahlásila interní cybu programu.

6.3 Druhý tester

Druhý tester je zaměstnacem v softwarové firmě, v které se zabývá programováním webových aplikací v Javě.

Bylo spuštění a ovládání intuitivní?

Velmi jednoduché, program je spustitelným souborem.

Je ze zobrazení jasné co zobrazuje jakou oblast paměti?

Každá sekce zobrazení je přehledně popsána tučným nadpisem s textem který jasně popisuje co daná sekce zobrazuje.

Je z vizualizace jasné, jaký efekt mají instrukce?

Většinou ano, text je dostatečně velký aby oko postřehlo, co se změnilo při vykonání instrukce.

Je něco co se vám líbí na grafickém rozhranní nebo naopak něco, co byste zlepšil?

Oceňuji příjemný a jednoduchý design, líbí se mi možnost prokliku na jiný Frame Stack a také možnost prohlédnutí attributů instancí a tříd. Ocenil bych hlášku, která dá uživateli vědět, že program doběhl. Dalším prostorem pro zlepšení by mohlo být přidání dalších možností navigace mezi rámci podobně jako u debuggerů v moderních IDE, tj. přeskočit, vyskočit ven, ... nebo možnost pokročení např. dvouklikem na rámec.

Vyskytla se během testování vnitřní chyba programu nesouvisející s intrepretovaným kódem?

Ne, nevyskytla.

6.4 Shrnutí testů

Uživatelé se v grafickém prostředí orientovali po krátkém úvodu snado a neměli problém spustit a prostudovat chování poskytnutého testovacího programu. Zobrazované informace jsou dostačující pro vizualicaci běhu programu a případné chyby jsou srozumitelně popsány.

Jako rozšíření této práce se může implementovat zvýrazňování změněných částí paměti, pro což by bylo zapotřebí implementovat sofistikovanější mechanismus ve virtuálním stroji pro upozorňování na konkrétní změny, než je stávají implementace, které upozorňuje pouze na změnu obecně.

Další rozšíření programu by mohla být nápověda pro instrukce s vysvětlením jejich funkce, která by se zobrazila např. při klinutní najetí myši na danou instrukci.

6.5 Další nalezené chyby

Během testování se projevilo několik chyb v implementaci chování některých instrukcí, např. při volání instrukcí `INVOKEVIRTUAL` a `INVOKESTATIC` se parametry pro metody získávaly ze zásobníku ve špatném pořadí. Dále se například nekontrolovalo, zda je velikost lokálních proměnných dostatečná pro parametry dané metody, což za běhu programu vedlo k chybě. Všechny tyto nalezené chyby byly opraveny.

7 Závěr

V této práci jsem prostudoval a stručně popsal specifikaci virtuálního stroje platformy Java – JVM, prostudoval existující emulátory a debuggery, navrhl vlastní vizuální interpreter podmnožiny bajtkódu Javy, který jsem poté implementoval a otestoval na jeho schopnost poskytnout dostatek informací o běhu programu.

V kapitole 2 jsem stručně popsal nejdůležitější funkcionalitu JVM podle oficiální specifikace. Popsal jsem datové typy, strukturu, chování a také instrukční sadu JVM.

Kapitola 3 se věnuje existujícím emulátorům a debuggerům z pohledu informativní hodnoty a přívětivosti zovrazovaných informací. Popsal jsem IDE IntelliJ IDEA, simulátor procesorové architektury MIPS QtSPIM a také simulátor procesorové architektury Ripes pro instrukční sadu RISC V.

Návrhu vlastního vizuálního interpreteru podmnožiny bajtkódu Javy, tedy virtuálního stroje s polečně s jeho grafickým rozhraním, jsem se věnoval v kapitole 4, v které je popsán návrh struktury, chování a zvolené podmnožiny instrukcí virtuálního stroje. U návrhu grafického rozhraní jsou popsány zobrazované informační prvky.

V kapitole 5 je popsána samotná implementace vizuálního interpreteru a možnosti budoucího rozšíření funkcionality.

Na závěr jsem v kapitole 6.5 shrnul výsledky testování implementovaného vizuálního interpreteru především z pohledu uživatelů a také popsal úpravy pramenící z těchto výsledků, které by bylo možné provést na vizuálním interpreteru.

Výsledkem této práce je program – vizuální intepreter upravené podmnožiny bajtkódu Javy, kterým má specifikovaný vlastní formát zdrojových souborů. Zdrojové soubory je poté možné spustit a zkoumat jejich chování ve vizuálním interpreteru.

Literatura

[1] Boucher, A. *Relm* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://github.com/antoyo/relm`.

[2] Corporation, O. *OpenJDK* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://github.com/AdoptOpenJDK/openjdk-jdk11`.

[3] Gomez, G. – Boucher, A. *Rust Programming By Example*. Packt Publishing Limited, 2018. ISBN 9781788390637.

[4] Gosling, J. et al. *The Java® Language Specification – Java SE 11 Edition* [online]. Oracle Corporation, 2018. [cit. 2020/07/03]. Dostupné z: `https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf`.

[5] Gtk-rs Developers. *Gtk-rs* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://gtk-rs.org`.

[6] Jet Brains. *IntelliJ IDEA* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://www.jetbrains.com/idea`.

[7] Larus, J. *QtSPIM* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `http://spimsimulator.sourceforge.net`.

[8] Lindholm, T. et al. *The Java® Virtual Machine Specification – Java SE 11 Edition* [online]. Oracle Corporation, 2018. [cit. 2020/07/03]. Dostupné z: `https://docs.oracle.com/javase/specs/jvms/se11/jvms11.pdf`.

[9] Petersen, M. B. *Ripes* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://github.com/mortbopet/Ripes`.

[10] The GTK Team. *GTK* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://www.gtk.org`.

[11] The Rust Team. *Rust Programming Language* [online]. 2020. [cit. 2020/07/03]. Dostupné z: `https://www.rust-lang.org`.

[12] Venners, B. *Inside the Java 2 Virtual Machine*. Computing McGraw-Hill, 2nd edition, 2000. ISBN 978-0071350938.

Přílohy

7.1 Sestavení a spuštění programu vizuálního interpreteru

7.1.1 Návod na sestavení

Sestavení programu vizuálního interpreteru je testováno pouze na Linuxu, konkrétně na distribucích Fedora 32 a Ubuntu 20.04, pro jiné distribuce je ale postup pro sestavení podobný.

Pro sestavení programu na Linuxu je nejdříve třeba naistalovat nástroj pro překlad jazyka Rust (verze 1.45) a také sadu nástrojů pro vývoj GUI rozhraní GTK (verze 3.24) což lze provést následovně:

Rust

Rust je možné naistalovat buď jako balíček z repozitářů podle konkrétní distribuce Linuxu, nebo pomocí nástroje pro správu verzí jazyka Rust `rustup`[1], což se provede následovně:

```
Instalace rustup
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

Instalace nástrojů pro vývoj v Rustu
$ rustup toolchain install 1.45.0

Nastavení použití verze 1.45.0 jako výchozí
$ rustup default 1.45.0
```

GTK

GTK je k dizpozici jako balíčky ve většině distribucí linuxu. Je třeba nainstalovat balíčky pro vývoj:

```
Fedora
$ dnf install gtk3-devel

Ubuntu
```

[1] rustup – `https://rustup.rs/`

```
$ apt install libgtk-3-dev
```

Pro sestavení programů v Rustu se používá nástroj `cargo`, který slouží zároveň i jako správce závislostí a je součástí instalace Rustu. Sestavení se provede následovně:

```
Přepnutí do kořenového adresáře programu
(kde se nachází Cargo.toml)
$ cd cesta/do/adresare

Sestavení
$ cargo build --release
```

Spustitelný soubor `mvm` s programem vznikne v podadresáři `target/release/`.

7.1.2 Spuštění

Program je dodán jako spustitelný soubor přeložený pro Linux, ale lze ho nejříve sestavit a poté přímo spustit, nebo spustit pomocí nástroje `cargo`. Pro běh programu je zapotřebí mít v systému nainstalované knihovny pro běh aplikací využívajících GTK. Ty jsou součástí vývojových balíčků, ale mohou být i samostatně. Na linuxových distribucích Fedora a Ubuntu jsou předinstalované.

```
Sputění pomocí cargo z kořenového adresáře programu
(kde se nachází Cargo.toml)
$ cd cesta/do/adresare
$ cargo run --release

Spuštění programu přímo
$ cd cesta/ke/spsutitelnemu/souboru
$ ./mvm
```

7.2 Tabulka instrukcí

Tabulka 7.1: Popis zvolených instrukcí

Jméno instrukce	**Parametry**	**Popis**
NOP		Žádná operace
	Načítání konstant na zásobník	
ACONST_NULL		`null` typu `reference`
ICONST_M1		-1 typu `int`
ICONST_0		0 typu `int`
ICONST_1		1 typu `int`
ICONST_2		2 typu `int`
ICONST_3		3 typu `int`
ICONST_4		4 typu `int`
ICONST_5		5 typu `int`
LCONST_0		0 typu `long`
LCONST_1		1 typu `long`
FCONST_0		0.0 typu `float`
FCONST_1		1.0 typu `float`
FCONST_2		2.0 typu `float`
DCONST_0		0.0 typu `double`
DCONST_1		1.0 typu `double`
BIPUSH	`i8`	`i8` typu `byte`
SIPUSH	`i16`	`i16` typu `short`
LDC	`i32` nebo `f32`	`i32` typu `int` nebo `f32` typu `float`
LDC_W	`i32` nebo `f32`	`i32` typu `int` nebo `f32` typu `float`
LDC2_W	`i64` nebo `f64`	`i64` typu `int` nebo `f64` typu `float`
	Načítání hodnot z lokálních proměnných na zásobník	
ILOAD	`u8`	`int` z indexu `u8`

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	**Parametry**	**Popis**
LLOAD	u8	`long` z indexu u8
FLOAD	u8	`float` z indexu u8
DLOAD	u8	`double` z indexu u8
ALOAD	u8	`reference` z indexu u8
ILOAD_0		`int` z indexu 0
ILOAD_1		`int` z indexu 1
ILOAD_2		`int` z indexu 2
ILOAD_3		`int` z indexu 3
LLOAD_0		`long` z indexu 0
LLOAD_1		`long` z indexu 1
LLOAD_2		`long` z indexu 2
LLOAD_3		`long` z indexu 3
FLOAD_0		`float` z indexu 0
FLOAD_1		`float` z indexu 1
FLOAD_2		`float` z indexu 2
FLOAD_3		`float` z indexu 3
DLOAD_0		`double` z indexu 0
DLOAD_1		`double` z indexu 1
DLOAD_2		`double` z indexu 2
DLOAD_3		`double` z indexu 3
ALOAD_0		`reference` z indexu 0
ALOAD_1		`reference` z indexu 1
ALOAD_2		`reference` z indexu 2
ALOAD_3		`reference` z indexu 3
Ukládání hodnot ze zásobníku do lokálních proměnných		
ISTORE	u8	`int` na index u8
LSTORE	u8	`long` na index u8
FSTORE	u8	`float` na index u8

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	**Parametry**	**Popis**
DSTORE	u8	`double` na index u8
ASTORE	u8	`reference` na index u8
ISTORE_0		`int` na index 0
ISTORE_1		`int` na index 1
ISTORE_2		`int` na index 2
ISTORE_3		`int` na index 3
LSTORE_0		`long` na index 0
LSTORE_1		`long` na index 1
LSTORE_2		`long` na index 2
LSTORE_3		`long` na index 3
FSTORE_0		`float` na index 0
FSTORE_1		`float` na index 1
FSTORE_2		`float` na index 2
FSTORE_3		`float` na index 3
DSTORE_0		`double` na index 0
DSTORE_1		`double` na index 1
DSTORE_2		`double` na index 2
DSTORE_3		`double` na index 3
ASTORE_0		`reference` na index 0
ASTORE_1		`reference` na index 1
ASTORE_2		`reference` na index 2
ASTORE_3		`reference` na index 3
Manipulace se zásobníkem		
POP		odstraní hodnotu výp. typu 1
POP2		odstraní hodnotu výp. typu 2
DUP		duplikuje hodnotu výp. typu 1 a uloží ji na zásobník
DUP_X1		duplikuje hodnotu výp. typu 1 a uloží ji na zásobník o dvě pozice níže

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	Parametry	Popis
`DUP_X2`		duplikuje hodnotu výp. typu 1 a uloží ji na zásobník o tři pozice níže
`DUP2`		duplikuje dvě hodnoty výp. typu 1, nebo hodnotu typu 2 a uloží je na zásobník
`DUP2_X1`		duplikuje dvě hodnoty výp. typu 1, nebo hodnotu typu 2 a uloží je na zásobník o dvě pozice níže
`DUP2_X2`		duplikuje dvě hodnoty výp. typu 1, nebo hodnotu typu 2 a uloží je na zásobník o tři pozice níže
`SWAP`		prohodí dvě hodnoty výp. typu 1 na vrcholu zásobníku mezi
Aritmetické operace		
`IADD`		sečte dvě hodnoty `int`
`LADD`		sečte dvě hodnoty `long`
`FADD`		sečte dvě hodnoty `float`
`DADD`		sečte dvě hodnoty `double`
`ISUB`		odečte první hodnotu `int` od druhé hodnoty `int`
`LSUB`		odečte první hodnotu `long` od druhé hodnoty `long`
`FSUB`		odečte první hodnotu `float` od druhé hodnoty `float`
`DSUB`		odečte první hodnotu `double` od druhé hodnoty `double`
`IMUL`		vynásobí dvě hodnoty `int`
`LMUL`		vynásobí dvě hodnoty `long`
`FMUL`		vynásobí dvě hodnoty `float`
`DMUL`		vynásobí dvě hodnoty `double`
`IDIV`		vydělí druhou hodnoty `int` první hodnotou `int`

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	Parametry	Popis
`LDIV`		vydělí druhou hodnoty `long` první hodnotou `long`
`FDIV`		vydělí druhou hodnoty `float` první hodnotou `float`
`DDIV`		vydělí druhou hodnoty `double` první hodnotou `double`
`IREM`		zjistí zbytek po dělení druhé hodnoty `int` první hodnotou `int`
`LREM`		zjistí zbytek po dělení druhé hodnoty `long` první hodnotou `long`
`FREM`		zjistí zbytek po dělení druhé hodnoty `float` první hodnotou `float`
`DREM`		zjistí zbytek po dělení druhé hodnoty `double` první hodnotou `double`
`INEG`		zneguje hodnotu `int`
`LNEG`		zneguje hodnotu `long`
`FNEG`		zneguje hodnotu `float`
`DNEG`		zneguje hodnotu `double`
`ISHL`		provede aritmetický bitový posuv doleva druhé hodnoty `int` o první hodnotu `int`
`LSHL`		provede aritmetický bitový posuv doleva druhé hodnoty `long` o první hodnotu `int`
`ISHR`		provede aritmetický bitový posuv doprava druhé hodnoty `int` o první hodnotu `int`
`LSHR`		provede aritmetický bitový posuv doprava druhé hodnoty `long` o první hodnotu `int`

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	**Parametry**	**Popis**
IUSHR		provede logický bitový posuv doprave druhé hodnoty `int` o první hodnotu `int`
LUSHR		provede logický bitový posuv doprave druhé hodnoty `long` o první hodnotu `int`
IAND		provede bitový součet dvou hodnot `int`
LAND		provede bitový součet dvou hodnot `long`
IOR		provede bitový součin dvou hodnot `int`
LOR		provede bitový součin dvou hodnot `long`
IXOR		provede bitovou nonekvivalenci dvou hodnot `int`
LXOR		provede bitovou nonekvivalenci dvou hodnot `long`
IINC	`u8`, `u8`	inkrementuje hodnotu lok. proměnné `int` na indexu o hodnotě prvního `u8` o hodnotu druhého `u8`
	Převádění datových typů	
I2L		`int` na `long`
I2F		`int` na `float`
I2D		`int` na `double`
L2I		`long` na `int`
L2F		`long` na `float`
L2D		`long` na `double`
F2I		`float` na `int`
F2L		`float` na `long`
F2D		`float` na `double`

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	**Parametry**	**Popis**
D2I		`double` na `int`
D2L		`double` na `long`
D2F		`double` na `float`
I2B		`int` na `byte`
I2S		`int` na `short`
	Porovnávání	
LCMP		Pokud jsou si dvě hodnoty `long` rovny, na zásobník se načte hodnota 0 typu `int`, pokud je druhá hodnota větší než první, načte se hodnota 1, a pokud je menší, hodnota -1
FCMPL		Pokud je druhá hodnota `float` větší než první `float`, na zásobník se načte hodnota 1 typu `int`, jinak pokud jsou si rovny, načte se hodnota 0, jinak pokud je menší, načte se hodnota -1, jinak pokud je jedna z hodnot rovna `NaN`, načte se hodnota 1
FCMPG		Jako předchozí, ale pokud je jedna z hodnot rovna `NaN`, načte se hodnota -1
DCMPL		Pokud je druhá hodnota `double` větší než první `double`, na zásobník se načte hodnota 1 typu `int`, jinak pokud jsou si rovny, načte se hodnota 0, jinak pokud je menší, načte se hodnota -1, jinak pokud je jedna z hodnot rovna `NaN`, načte se hodnota 1

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	Parametry	Popis
`DCMPG`		Jako předchozí, ale pokud je jedna z hodnot rovna `NaN`, načte se hodnota -1
`IFEQ`	`i16`	Pokud je hodnota `int` rovna 0, skočí na instrukci `pc + i16`
`IFNE`	`i16`	Pokud hodnota `int` není rovna 0, skočí na instrukci `pc + i16`
`IFLT`	`i16`	Pokud je hodnota `int` menší než 0, skočí na instrukci `pc + i16`
`IFGE`	`i16`	Pokud je hodnota `int` větší nebo rovna 0, skočí na instrukci `pc + i16`
`IFGT`	`i16`	Pokud je hodnota `int` větší než 0, skočí na instrukci `pc + i16`
`IFLE`	`i16`	Pokud je hodnota `int` menší nebo rovna 0, skočí na instrukci `pc + i16`
`IF_ICMPEQ`	`i16`	Pokud se dvě hodnoty `int` rovnají, skočí na instrukci `pc + i16`
`IF_ICMPNE`	`i16`	Pokud se dvě hodnoty `int` nerovnají, skočí na instrukci `pc + i16`
`IF_ICMPLT`	`i16`	Pokud je první hodnota `int` menší než druhá hodnota `int`, skočí na instrukci `pc + i16`
`IF_ICMPGE`	`i16`	Pokud je první hodnota `int` větší nebo rovna druhé hodnotě `int`, skočí na instrukci `pc + i16`
`IF_ICMPGT`	`i16`	Pokud je první hodnota `int` větší než druhá hodnota `int`, skočí na instrukci `pc + i16`
`IF_ICMPLE`	`i16`	Pokud je první hodnota `int` menší nebo rovna druhé hodnotě `int`, skočí na instrukci `pc + i16`

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	**Parametry**	**Popis**
`IF_ACMPEQ`	`i16`	Pokud se dvě hodnoty `reference` rovnají, skočí na instrukci `pc` + `i16`
`IF_ACMPNE`	`i16`	Pokud se dvě hodnoty `reference` nerovnají, skočí na instrukci `pc` + `i16`
`IFNULL`	`i16`	Pokud se hodnota `reference` rovná `null`, skočí na instrukci `pc` + `i16`
`IFNONNULL`	`i16`	Pokud se hodnota `reference` nerovná `null`, skočí na instrukci `pc` + `i16`
	Řízení běhu	
`GOTO`	`i16`	skok na instrukci `pc` + `i16`
`IRETURN`		návrat z metody s vrácením hodnoty `int`
`LRETURN`		návrat z metody s vrácením hodnoty `long`
`FRETURN`		návrat z metody s vrácením hodnoty `float`
`DRETURN`		návrat z metody s vrácením hodnoty `double`
`ARETURN`		návrat z metody s vrácením hodnoty `reference`
`RETURN`		návrat z metody
	Práce s objekty	
`GETSTATIC`	`field_ref`	načtení statického atributu na zásobník
`PUTSTATIC`	`field_ref`	uložení hodnoty do statického atributu
`GETFIELD`	`field_ref`	načtení atributu instance na zásobník
`PUTFIELD`	`field_ref`	uložení hodnoty do atributu instance
`INVOKEVIRTUAL`	`method_ref`	zavolání virtuální metody na instanci, získané ze zásobníku

Pokračování na další straně

Pokračování z předchozí strany

Jméno instrukce	**Parametry**	**Popis**
`INVOKESTATIC`	`method_ref`	zavolání statické metody
`NEW`	`class_ref`	vytvoření nové instance třídy, reference je uložena na záasobník