Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Katedra informatiky a výpočetní techniky

Studijní program Informatika

Bakalářská práce

Monitorování alokací paměti za běhu Java aplikací

Jiří Velek

Vedoucí práce: Ing. Ph.D. Richard Lipka

Katedra informatiky a výpočetní techniky

Fakulta aplikovaných věd Západočeské univerzity v Plzni

Plzeň 2023

Čestné prohlášení

Prohlašuji, že jsem bakalářskou práci na téma

„Monitorování alokací paměti za běhu Java aplikací“

vypracoval samostatně pod odborným dohledem vedoucí bakalářské práce za použití pramenů uvedených v přiložené bibliografii.

Plzeň dne 9. 5. 2023 v. r. Jiří Velek

Poděkování

Obsah

1 Úvod

2 Paměťový model Javy

Paměťový model jazyka Java je klíčovým prvkem pro správu paměti v Java aplikacích a pro jejich optimální výkon. Paměťový model popisuje způsob, jakým Java ukládá a spravuje data v paměti během běhu aplikace.

2.1 Java Virtual Machine

Software napsaný v jazyce Java je spuštěn v prostředí Java Virtual Machine (JVM). JVM má za úkol odstranit závislost napsaných programů na konkrétním operačním systému [1]. JVM je abstraktní počítač, který má stejně jako reálný počítač instrukční sadu a je schopný manipulovat s pamětí za běhu programu.

Při kompilaci zdrojového kódu jazyka Java vzniká byte kód, který je uložen v class souboru. Class soubor obsahuje instrukce pro virtuální stroj Javy, který je interpretuje a spouští. To znamená, že může existovat více programovacích jazyků, v nichž napsané programy se přeloží do množiny class souborů, kterou bude JVM schopný vykonat.

Class soubor je tvořen JVM instrukcemi (byte kódem), tabulkou symbolů a dalšími důležitými informacemi pro běh programu.

2.2 Datové typy

Ke správnému vykonání jednotlivých instrukcí jsou potřeba datové typy. JVM pracuje se dvěma datovými typy: primitivními a referenčními [2]. Tyto typy (resp. jejich hodnoty) mohou být uloženy do proměnných, předány metodě jako argument, využity jako návratová hodnota, nebo s nimi lze provádět výpočty. Velikosti jednotlivých datových typů a jejich výchozí hodnoty jsou v Tabulka *2.1*: Velikosti a výchozí hodnoty primitivních datových typů.

2.2.1 Primitivní datové typy

Primitivní datové typy se dělí na celočíselné a typy s plovoucí desetinnou čárkou, binární typ a speciální typ `returnAddress` [2]. Celočíselné typy se dále dělí na:

- `byte`
- `short`
- `int`
- `long`
- `char`

Typy s plovoucí desetinnou čárkou se dělí na:

- `float`
- `double`

2.2.2 Referenční typy

V JVM existují tři různé referenční typy: `class, array a interface` [2]. Speciální referenční hodnotou je také hodnota `null`, která značí absenci objektu (reference na neexistující objekt).

2.3 Paměťové oblasti

Paměť je v Javě rozdělena do několika oblastí, které jsou přizpůsobeny různým velikostem datových typů. Některé oblasti mohou mít specializovaná využití, některé mohou být víceúčelové.

2.3.1 Zásobník

Každému vláknu je přiřazena soukromá paměť, nazývaná zásobník [2]. Zásobník v Javě je podobný zásobníkům v ostatních programovací jazycích (například C) – ukládají se do něj lokální proměnné, částečné výpočty a hraje velkou roli ve volání metod a návratu z nich.

Přístup k pamětí v zásobníku je realizován v LIFO(Last-In-First-Out) pořadí. Přistupovat k němu lze tedy pouze pomocí `push` a `pop` instrukcí [2]. Paměť zásobníku je limitována na několik MiB a po naplnění této paměti JVM vyhodí výjimku `java.lang.StackOverFlowError`.

2.3.2 Zásobník nativních metod

JVM je také vybaven tzv. zásobníkem nativních metod, také nazývaný „C zásobník“ [2]. Tento typ zásobníku zajišťuje podporu nativních metod, které jsou napsané v jiných programovacích jazycích než Java.

2.3.3 Zásobníkový rámec

Při invokaci metody je vytvořen zásobníkový rámec, který je po dokončení průběhu metody smazán [2]. Rámce jsou alokovány ze zásobníku. Každý rámec obsahuje jedno pole lokálních proměnných, zásobník operandů a referenci na fond konstant.

2.3.4 Halda

JVM má haldu, která obsahuje sdílenou paměť mezi všemi vlákny. Halda je paměťová oblast, která slouží k uložení veškerých instancí tříd a polí [2].
Halda je vytvořena při spuštění JVM a její velikost může, ale nemusí být konstantní. Paměť v haldě je spravována garbage kolektorem. Velikost paměti v haldě je v řádu několika desítek či stovek GiB. Jestliže výpočet potřebuje více paměti, než má systém k dispozici, JVM vyhodí výjimku `java.lang.OutOfMemoryError`.

2.3.5 Oblast metod

Oblast metod je sdílená paměť mezi všemi vlákny, obsahující kód a metadata všech metod a konstruktorů, včetně speciálních metod určených k inicializaci instancí tříd [2].

2.3.6 Fond konstant

Každá třída obsahuje fond konstant. Obsahuje několik typů konstant od numerických literálů známých při kompilaci programu, po reference na metody či atributy, které musí být vypočítány za běhu programu [2].

2.4 Správa paměti

JVM spravuje paměť automatickým systémem správy úložiště (garbage kolektorem). JVM si při spuštění požádá operační systém o paměť a z této oblasti poté alokuje objekty. Programátor tedy nemusí přemýšlet nad alokací a uvolňování paměti objektů, a může se soustředit na psaní výkonného kódu.

2.4.1 Garbage kolektor

Garbage kolekce je proces sledující haldu, který identifikuje objekty, k nimž program již nemá přístup a uvolňuje paměť ve které byl objekt uložen. Kolektor prochází seznam všech objektů a hledá, zda k němu existuje přístup. Tento přístup je ovšem velmi neefektivní a výzkum prokazuje, že většina objektů žijí krátkou dobu. Tomuto výzkumu se říká generační hypotéza a je důvodem vzniku generační kolekce.

2.4.2 Generační kolekce

Halda je rozdělena do několika částí (generací):

Mladá generace

Do mladé generace jsou umístěny všechny nově vytvořené objekty. Při zaplnění této generace se spustí menší garbage kolekce [3]. Pokud objekt tuto kolekci přežije, je přesunut do první oblasti přeživších (S0).

Oblast přeživších

Oblast přeživších je rozdělena na dvě oblasti (S0, S1). Ukládá objekty, které nebyly zasaženy kolekcí v mladé generaci. Pokud je objekt v první z těchto oblastí a nezasáhne ho menší kolekce, je přesunut do druhé. Je-li v druhé z nich a nezasáhne ho menší kolekce, je přesunut do staré generace [3].

Stará generace

Obsahuje objekty přesunuty z oblasti S1. Při zaplnění této části se spustí větší garbage kolekce [3]. Větší garbage kolekce trvá déle než menší. Z tohoto důvodu je objektům zabráněno ihned vstoupit do staré generace.

Permanentní generace

Permanentní generace není ovlivněna garbage kolektorem. Je vytvořena při spuštění programu. Jsou v ní obsaženy metadata tříd, fond konstant, byte kód metod a konstruktorů (oblast metod) [3].

Generační hypotéza nemusí být pro některé aplikace pravdivá [4]. Výkon JVM je proto negativně ovlivněn s aplikacemi, které očekávají „střední“ délku života jejich objektů.

2.4.3 Garbage kolektory v Javě

Java nabízí několik garbage kolektorů, které se dají přepínat při spuštění programu. Všechny tyto kolektory při čištění paměti (kromě `Concurrent Mark & Sweep`) pozastaví všechna běžící vlákna JVM a po dokončení kolekce je zase spustí.

Mark & Sweep

`Mark & Sweep` kolektor pracuje ve staré generaci. Využívá dvě základní operace. První operace (mark) prochází objekty a označuje je jako dosažitelné či nikoliv. Druhá operace (sweep) smaže všechny objekty označené jako nedosažitelné a jejich paměť uvolní.

`Mark & Sweep` má i paralelizovanou verzi nazývanou `Concurrent Mark & Sweep`.

Copy

Kopírovací kolektor pracuje v mladé generaci a oblastmi přeživších [5]. Objekty, které nebyly zasaženy kolekcí kopíruje mezi mladou generací a oblastmi přeživších. Po první kolekci překopíruje nezasažené objekty do S0, po několika kolekcích do S1.

Po uplynutí několika dalších kolekcí jsou objekty z S1 překopírovány do staré generace. Celý tento proces by se dal představit jako „fronta“, kde jednotlivé objekty čekají na přesun do další generace, nebo na odmítnutí, jelikož k objektu není přístup.

Garbage-First (G1)

Garbage-first kolektor je k dispozici od verze Java 7 [6]. Využívá Garbage-First algoritmus. Od Javy verze 9 je Garbage-First kolektor výchozím kolektorem. Garbage-First algoritmus funguje tak, že rozdělí haldu na mnoho menších částí stejné velikosti. Název „Garbage-First“ je zvolen proto, že při kolekci mají prioritu oblasti, které obsahují nejméně dosažitelných objektů. Při použití Garbage-First kolektoru neplatí generační hypotéza, jelikož tento kolektor nerozděluje haldu na generace.

2.5 Alokace paměti

Java má dva způsoby alokace paměti. Alokace paměti na zásobníku, nebo alokace paměti na haldě.

2.5.1 Alokace paměti na zásobníku

Alokace paměti na zásobníku probíhá při kompilaci programu. Je typická pro primitivní a referenční datové typy. Probíhá tak, že se při deklaraci proměnné na zásobníku uvolní paměťová buňka určité velikosti (viz Tabulka *2.1*: Velikosti a výchozí hodnoty primitivních datových typů).

Datový typ	Velikost buňky (v bytech)	Výchozí hodnota
`byte`	1	0
`short`	2	0
`int`	4	0
`long`	8	0L
`float`	4	0.0f
`double`	8	0.0d
`char`	2	‚\u0000‘
`Object`	Není přesně definováno	Null
`boolean`	Není přesně definováno	False

Tabulka 2.1: Velikosti a výchozí hodnoty primitivních datových typů

2.5.2 Alokace paměti na haldě

Alokace paměti na haldě probíhá za běhu programu. V kódu Javy se pozná klíčovým slovem `new`. Nastane ale také při inicializaci programu při statické inicializaci. K alokaci paměti na haldě slouží v JVM byte kódu několik instrukcí. Po vykonání této instrukce je na JVM zásobníku uložena nová reference odkazující na alokovanou paměť.

New

Alokuje paměť pro objekt [7]. Instrukce přebírá dva parametry (2 byty), nazývané `indexbyte1`, `indexbyte2`, kde hodnota výrazu „`(indexbyte1 << 8) | indexbyte2`" je index do fondu konstant. Obsah prvku na daném indexu musí být symbolická reference na třídu, nebo rozhraní. Všechny proměnné dané instance jsou inicializovány na jejich výchozí hodnoty (viz. Tabulka 1).

Newarray

Alokuje paměť pro pole primitivních datových typů [7]. Instrukce přebírá dva parametry (velikost pole a datový typ). Velikost pole je celočíselného datového typu. Datový typ je kód, indikující, který typ pole vytvořit (viz. Tabulka 2.2: Kódové označení primitivních datových typů). Všechny prvky vytvořeného pole jsou inicializovány na jejich výchozí hodnoty (viz. Tabulka *2.1*: Velikosti a výchozí hodnoty primitivních datových typů).

Typ pole	Kód typu
`T_BOOLEAN`	4
`T_CHAR`	5
`T_FLOAT`	6
`T_DOUBLE`	7
`T_BYTE`	8
`T_SHORT`	9
`T_INT`	10
`T_LONG`	11

Tabulka 2.2: Kódové označení primitivních datových typů

Anewarray

Alokuje paměť pro pole referencí [7]. Instrukce přebírá 3 parametry (`indexbyte1`, `indexbyte2`, velikost pole), `indexbyte1` a `indexbyte2` stejně jako u instrukce `new` určují typ reference, velikost pole je celočíselného datového typu. Všechny prvky nově vytvořeného pole jsou inicializovány na `null`, což je výchozí hodnota pro referenční datové typy.

Multianewarray

Alokuje paměť pro vícerozměrné pole [7]. Přebírá 4 a více argumentů (`indexbyte1`, `indexbyte2`, počet dimenzí, počet prvků v jedné dimenzi). Pomocí `indexbyte1` a `indexbyte2` se stejně jako u `new` určí typ reference. Počet dimenzí jsou typu neznaménkový byte a počet prvků v dimenzi je celočíselného datového typu. Prvky v poli první dimenze jsou inicializovány na pole typu druhé dimenze atd. Prvky polí v poslední dimenzi jsou inicializovány na jejich výchozí hodnotu (viz. Tabulka 1).

2.6 Vytváření instancí objektů

Vytváření instancí objektů klíčovým slovem `new` je několika krokový proces. Při vytváření objektu je pro něj potřeba alokovat paměť a inicializovat ho voláním metody `<init>`. Sekvence pro Java kód „`new MyClass(i, j, k);`“ může vypadat například, jako ve Fragment kódu 2.1 [8].

```
new #1           // alokuje paměť pro instanci třídy MyClass
dup              // vytvoří duplikát reference objektu na zásobníku
iload_1          // vloží na zásobník proměnnou i
iload_2          // vloží na zásobník proměnnou j
iload_3          // vloží na zásobník proměnnou k
invokespecial #5 // zavolá metodu objektu <init>, která připraví objekt
                 // pro plnohodnotné využití
```

Fragment kódu 2.1: Byte kód vytváření instancí objektu

3 Důvody neefektivního využití paměti v Javě

I přesto, že Java umožňuje jednoduchou správu paměti pomocí automatické garbage kolekce, mohou nastat situace, kdy je paměť neefektivně využívána. Tyto situace se mohou prokazovat různými způsoby – program může skončit výjimkou, nebo se po nějaké době běhu výrazně zpomalí.

3.1 Statické proměnné

Prvním příkladem, kdy může dojít k úniku paměti v Javě, je velmi časté využívání statických proměnných [9]. Fragment kódu 3.1: Únik paměti v Javě nadměrným využitím statických proměnnýchukazuje, jak by k takovému úniku paměti mohlo dojít.

```
public class StaticTest {
    public static List<Double> list = new ArrayList<>();

    public void populateList() {
        for (int i = 0; i < 10000000; i++) {
            list.add(i);
        }
    }
}
```

Fragment kódu 3.1: Únik paměti v Javě nadměrným využitím statických proměnných

V Javě mají statické proměnné většinou životnost délky běhu celého programu [9], list proto po zavolání metody `populateList` nikdy neuvolní svoji paměť, i když ho žádná z částí programu nepotřebuje. K vyřešení takového problému by přitom stačilo buď odstranit klíčové slovo `static`, nebo po posledním použití listu dovolit garbage kolektoru uvolnit paměť (například nastavením proměnné na hodnotu `null`).

3.2 Neuzavřené zdroje

Při otevírání nových spojení a streamů JVM alokuje paměť [9]. Typickým příkladem může být připojení k databázi, nebo čtení či zápis do souboru. Při neuzavření těchto zdrojů je paměť blokována a garbage kolektor ji nemůže uvolnit. Fragment kódu 3.2:

Únik paměti při neuzavření zdroje ukazuje, jak vzniká únik paměti při neuzavírání zdrojů.

```
public void readFromURL() {
    try {
        URL url = new URL("http://example.com");
        URLConnection urlConnection = url.openConnection();
        InputStream is = urlConnection.getInputStream();
        byte[] bytes = is.readAllBytes();
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

Fragment kódu 3.2: Únik paměti při neuzavření zdroje

Řešením tohoto problému je využití `finally` bloku, ve kterém se bude zdroj uzavírat. Od Java verze 8 lze také využít jazykový konstrukt try-with-resources, který automaticky uzavírá zdroje.

3.3 Řetězce

Standardní knihovna jazyka Java nabízí tři různé třídy pro práci s řetězci [10]. Pro práci s neměnnými řetězci se využívá třída `String`. Pro variabilní řetězce existují dvě třídy `StringBuffer` a `StringBuilder`.

Nejčastěji vytvářenými objekty v Java programech jsou třídy typu `String`. Prvním častým důvodem neefektivního využití řetězců je duplicita jejich hodnot. Druhým důvodem neefektivního využití paměti je alokace objektu reprezentující řetězec, ale aplikace ho nikdy nevyužije.

3.3.1 Fond řetězců

Fond řetězců je místo na haldě, která slouží k ukládání řetězců s unikátní hodnotou. Při vytváření řetězce je zjištěno, zda řetězec ve fondu již existuje. Pokud existuje, JVM může vrátit referenci na řetězec ve fondu. V opačném případě může JVM řetězec do fondu přidat. Pokud chce programátor toto chování vynutit, může využít metodu `String.intern` [11]. Pokud programátor chce toto chování naopak potlačit, lze

využít konstruktor třídy `String` a fond řetězců se vůbec nepoužije. Tímto fondem lze vyřešit problém duplicitních řetězců.

3.4 Duplicitní objekty

Stejně jako duplicitní řetězce mohou v Java aplikacích dělat problémy obecně objekty. Pokud jsou tyto objekty neměnné, pak zbytečně zabírají paměť. Mohlo by se jednat například o objekty s návrhovým vzorem přepravka, či různé formátovače řetězců.
Hodnoty typu String lze internovat a šetřit tak využitou paměť. U ostatních datových typů taková možnost neexistuje. V knihovně Guava proto existuje třída `Interners`, která obsahuje metody pro internování všeobecných objektů [12]. Tyto metody ukládají objekty do datové struktury `ConcurrentHashMap`, která je dostupná ve standardní knihovně.
Při použití internování může nastat situace, kdy program využívá ještě více paměti než předtím [12]. Tato situace nastane, pokud v je v paměti velmi nízký počet duplikátů. Každý internovaný objekt se ukládá jako instance třídy `WeakKeyDummyValueEntry`, která v paměti zabírá 40 bytů. Jestliže programátor internuje například objekty o velikosti 48 bytů, pak jedno internování bude zabírat velikost 88 bytů. Tudíž bude zabírat cca. dvojnásobné množství paměti. Při větším počtu objektů toto může být problém a program může skončit výjimkou `OutOfMemoryError`.

3.5 Řídce zaplněné kolekce

Kolekce jsou nevyhnutelnou součástí Javy. Jejich špatné využití může způsobit nejen pomalý běh programu, ale také neefektivní využití paměti.
Řídce zaplněné kolekce mohou představovat problém, kdy je paměť pro kolekci alokována, ale není využita. Kolekce ArrayList až do verze Javy 8 měla výchozí kapacitu 10 prvků [13]. Od verze Java 8 a výše se pole inicializuje až po přidání prvního prvku [14].

4 Java Runtime Monitoring

K nalezení neefektivního využití paměti je využit proces nazývaný runtime monitoring. K tomuto procesu slouží několik nástrojů, které sledují obsah paměti a poskytují vývojářům detailní informace o obsahu paměti. Tyto nástroje mohou být spouštěny z příkazové řádky, nebo obsahují grafické rozhraní.

4.1 JDK Mission Control

JDK Mission Control (JMC) je množina nástrojů pro pokročilý monitoring, profilování a řešení problémů v Java aplikacích [15]. JMC umožňuje efektivní a detailní analýzu oblastí jako například výkon kódu, paměti a latence, aniž by vytvářel režijní náklady, které jsou většinou s profilovacími a monitorovacími nástroji spojovány.

4.2 Java VisualVM

Java VisualVM je nástroj, který poskytuje grafické rozhraní pro prohlížení detailních informací o běžících Java aplikacích. Tento nástroj využívá ostatních nástrojů jako JConsole a jstat pro získání informací z JVM. Tyto informace poté reorganizuje a prezentuje uživateli graficky. Java VisualVM lze jednoduše rozšířit podobou pluginů, které uživateli mohou poskytovat další užitečné informace o běžících aplikacích.

4.3 VisualGC

VisualGC využívá JVM instrumentace pro grafické znázornění garbage kolekce. Pomocí nástroje lze zjistit obsazení jednotlivých oblastí haldy, nebo počty objektů, které mají určitý věk v mladé generaci [16]. Tyto informace jsou uživateli prezentovány pomocí grafů.

4.4 Memory Analyzer

V roce 2021 vznikl na katedře informatiky a výpočetní techniky nástroj pro sledování duplicit objektů. Nástroj vznikl jako nadstavba předchozího nástroje se stejným názvem kvůli jeho nedostatkům [17]. Například vyhledávání duplicit je v původním nástroji realizováno mělkým porovnáváním, což je v praxi nedostatečné.
Nástroj je schopen přečíst snímek stavu haldy aplikace a z ní detekovat duplikáty vytvořených objektů.

4.4.1 Analyzátor referencí

Nástroj obsahuje tzv. analyzátor referencí, který počítá celkový počet referencí a počet `null` referencí ve stavu haldy aplikace. Tato informace může uživateli říci, zda program neobsahuje příliš mnoho `null` referencí.

4.4.2 Analyzátor duplicitních objektů

Analyzátor pracuje porovnáváním objektů do hloubky. Udržuje si kolekci objektů a porovnává vždy poslední prvek z kolekce se všemi ostatními. Potom všechny nalezené duplikáty v kolekci nastaví na `null`. Samotné porovnávání je realizováno pomocí rekurzivního algoritmu, který pokračuje, dokud neporovnává dva primitivní datové typy.
Algoritmus nejprve seřadí atributy třídy podle toho, zda jsou primitivního datového typu (nebo datového typu `String`), či nikoliv. Poté porovná všechny atributy primitivního datového typu, a až poté porovnává všechny atributy referenčního datového typu. Algoritmus pracuje paralelně a výsledky dřívějších porovnávání ukládá, aby nebylo třeba procházet pro stejný objekt rekurzí vícekrát. Samotný autor uvádí, že ukládání výsledků předchozích porovnání se příliš nevyplácí, kvůli zpomalení aplikace a velikosti datové struktury pro ukládání.

5 Možnosti instrumentace v Javě

Instrumentace v Javě je způsob, kterým můžeme změnit načítání tříd a jejich byte kód vložením vlastního kódu přímo za běhu programu [18]. Dvě metody, které toto umožňují jsou Java Instrumentation API, nebo koncept Aspektově Orientovaného Programování.

5.1 Java Instrumentation API

Java Instrumentation API umožňuje uživateli přístup k třídám načteným JVM a modifikovat jejích byte kód [18]. Výhoda Java Instrumentation API je, že je součástí balíku `java.lang` a není proto potřeba instalovat žádné externí knihovny. Realizace instrumentace pomocí Java Instrumentation API je rozdělena do několika částí:

5.1.1 Třída agenta

Třída agenta obsahuje metodu `premain` [18]. Tato metoda je vykonána před načtením metody `main` instrumentovaného programu a může mít následující hlavičky:

1. `public static void premain(String agentArgs, Instrumentation inst);`
2. `public static void premain(String agentArgs);`

Třída `Instrumentation` obsahuje metody, které nám dovolují přidat transformery byte kódu.

5.1.2 Transformer

Třídy, které transformují byte kód se nazývají transformer. Transformery musí implementovat rozhraní `ClassFileTransformer` [18]. Rozhraní obsahuje metodu `transform`, která obdrží byte kód původní třídy. Lze poté vytvořit kopii tohoto byte kódu, upravit jí a pomocí návratové hodnoty nahradit kód původní třídy.

5.1.3 Agent

Agent je JAR soubor, obsahující transformery a třídu agenta [18]. Pro agenta je definováno několik manifestových atributů. Zde jsou uvedeny zásadní atributy pro tuto práci [19]:

- `Premain-Class` – obsahuje název třídy agenta, ve které se nachází metoda `premain`.
- `Can-Redefine-Classes` – pravdivostní hodnota. Říká, zda agent může redefinovat třídy.
- `Can-Retransform-Classes` – pravdivostní hodnota. Říká, zda agent může retransformovat třídy.

5.1.4 Spouštění programu s agentem

Agent musí obsahovat metodu `premain` a jeho manifestový soubor musí obsahovat atribut `Premain-Class` [19]. Poté lze aplikace spustit z příkazové řádky pomocí následujícího příkazu: `-javaagent:<jarpath>[=<options>]`

5.2 Aspektově Orientované Programování

Aspektově orientované programování (AOP) doplňuje objektově orientované programování (OOP) tím, že nabízí jiný způsob přemýšlení nad strukturou programu [20].

5.2.1 Join point

Join point je bod v toku řízení programu. Říká, kde bude původní program ovlivněn chováním definovaným v aspektu.

5.2.2 Pointcut

Pointcut je množina obsahující konkrétních join pointů. Je definován wildcardy, které mohou být nahrazeny libovolnou posloupností znaků, nebo parametry metod [21].

Některé nástroje umožňují pracovat s pointcuty jako s množinami a zavádí operátory průniku, sjednocení a negace.

5.2.3 Advice

Advice je část kódu proveditelný v místě join pointu, který byl vybrán pointcutem. Zápisem je podobný metodě a lze určit, zda se provede před, po nebo „okolo“ (around) daného pointcutu. Advice „okolo“ dokáže vykonat kód před i po daném join pointu [22].

5.2.4 Aspekt

Aspekt je programová entita podobná třídě. Obsahuje definice pointcutů a k nim přiřazené advice. Může ale také obsahovat libovolné proměnné, k nimž se dá v jednotlivých advice přistupovat [21].

5.2.5 AspectJ

Jedním ze zástupců AOP v Javě je knihovna AspectJ. AspectJ je obecně účelové rozšíření programového jazyka Java.

5.3 ASM

ASM je všestranný framework pro analýzu a manipulaci byte kódu v Javě [23]. Využívá se k úpravě existujících tříd, či vytvoření nových dynamicky. ASM nabízí podobnou funkcionalitu, jako jiné frameworky, ale je primárně zaměřen na výkon. Framework využívá BSD licenci a v době psaní je poslední verze ASM 9.5 vydána 24. března 2023.

5.3.1 Stromové API

Z procházené třídy sestrojí stromovou datovou strukturu [24]. Kořen stromu slouží jako vstupní bod do dané třídy a jako potomky obsahuje atributy, metody, vnitřní třídy

a další informace. Stromové API poskytuje plnou kontrolu procházené třídy a je velmi jednoduché třídu transformovat.

5.3.2 Událostní API

Událostní API je založeno na návrhovém vzoru `Visitor` [25]. Je složeno z následujících komponent:

- `ClassReader` – čte class soubory a zahajuje transformaci třídy
- `ClassVisitor` – poskytuje metody k transformaci třídy
- `ClassWriter` – slouží k výstupu transformované třídy

`ClassVisitor` obsahuje všechny visit metody, které nám dovolují přistupovat k jednotlivým komponentám (atributy, metody atd.) třídy. Prakticky je toto realizováno pomocí podtřídy od `ClassVisitor`, která implementuje veškeré změny v dané třídě. Událostní API je velmi rychlé, ale má nižší kontrolu a vyžaduje vyšší disciplínu programátora, jelikož vyžaduje dodržování pořadí volání metod jednotlivých visitorů [24].

5.4 Javassist

Javassist (Java programming assistant) je knihovna pro úpravu byte kódu [26]. Povoluje Java programům definovat nové třídy za běhu programu, nebo modifikovat class soubor po jeho načtení. Na rozdíl od ostatních podobných knihoven, Javassist poskytuje dvě různá API: API zdrojového kódu a API byte kódu. Pokud programátor využije API zdrojového kódu, může upravit třídu bez jakékoliv znalosti Java byte kódu. Na druhou stranu, byte kódové API dovoluje programátorovi upravovat třídu stejně, jako ostatní knihovny.

5.4.1 Reprezentace třídy

Prvním krokem pro využití knihovny Javassist je vytvoření `CtClass` objektu. Tento objekt reprezentuje byte kód třídy načtené pomocí JVM [27]. Dále je možné tuto třídu

pozorovat či upravovat. Tyto úpravy se projeví na byte kódu vytvořeném tímto objektem. Načtení tohoto byte kódu se dá považovat za reflexi daného `CtClass` objektu. Javassist také dovoluje definovat novou třídu bez čtení jakéhokoliv class souboru. To může být užitečné, pokud program potřebuje dynamicky definovat třídu. Tato nová třída je prázdná a neobsahuje žádné atributy či metody. Ty mohou být dodány později.

5.4.2 Introspekce

Javassist poskytuje několik metod pro introspekci tříd. Tato část Javassist API je kompatibilní s Java reflection API. Výjimkou je, že Javassist neposkytuje možnost vytvoření nové instance, nebo invokace metody. Důvodem je, že tato funkcionalita je při načítání tříd zbytečná [27].

5.4.3 Záměna

Javassist nabízí metody pro záměnu definic tříd [27]. Tyto metody jsou kategorizovány do několika skupin:

- Metody pro změnu modifikátorů třídy
- Metody pro změnu hierarchie tříd
- Metody pro přidávání nových atributů či metod

6 Návrh implementace

V kapitole 4.4 byl popsán nástroj, který je schopný najít duplikáty ze snímku stavu haldy. Tento nástroj byl popsaným nástrojem inspirován a je navržen tak, aby dokázal duplikáty nalézt již za běhu programu.

6.1 Sběr alokovaných referencí za běhu programu

Při alokaci paměti, je objektu vytvořena reference. Tuto referenci je potřeba zachytit ihned po alokaci, aby bylo zřejmé, kde v programu dochází k nejfrekventovanější duplikaci objektů v paměti.

6.1.1 Java Agent

Zachycení reference je realizováno pomocí Java Agenta, který se do procesu připojí ihned při jeho spuštění. Proto je spouštění agenta tak, jak je popsáno v kapitole 5.1.4 vhodné pro tento nástroj. Tento agent je zodpovědný za fungování instrumentace.

6.1.2 Instrumentace

Samotné zachycení reference realizuje instrumentace. Z Fragment kódu *2.1* je vidět, že ke každé instrukci `new` je přiřazena instrukce `invokespecial`, až poté je objekt použitelný. Z tohoto pozorování lze říci, že za danou `invokespecial` instrukci je možné vložit vlastní byte kód, který zavolá příslušnou metodu. Tato metoda uloží referenci do kolekce všech alokovaných objektů.

6.2 Získání pozice alokace objektu v kódu

Pro analýzu je vhodné znát informaci o tom, kde k alokaci v programu dochází. V Javě lze za běhu programu procházet jednotlivé zásobníkové rámce pomocí třídy `StackTraceElement` a z nich získat název třídy, metody a číslo řádku, kde k alokaci došlo [28]. Tyto informace lze společně s alokovaným objektem uložit do společné datové struktury.

6.3 Výpočet velikosti objektů

Je dobré vědět, kolik bytů jednotlivý objekt v paměti zabírá. Tato informace je přístupná pomocí instrumentace metodou `Instrumentation.getObjectSize`. Toto řešení je skoro ideální, až na případ, kdy objektem je vícerozměrné pole. Je tedy nutné skombinovat instrumentaci s reflexí pro získání kompletní velikosti vícerozměrného pole.

6.4 Hledání duplicitních objektů

Kolekce alokovaných objektů může obsahovat duplicity. Tyto duplicity mohou naznačovat neefektivně využitou paměť. Závažnost neefektivnosti záleží na počtu duplicitních objektů. K nalezení všech duplicit je důležité objekty správně porovnávat.

6.5 Porovnávání objektů

K označení objektů jako duplikát je třeba zavést porovnávání objektů. Klasické porovnávání objektů pomocí metody `equals` není dostačující, jelikož její implementace ve třídě není povinná. Zavedeme tak porovnávání objektů dvojího typu. Objekty lze porovnávat buďto mělce, nebo do hloubky.

6.5.1 Mělké porovnávání

Mělké porovnávání spočívá v tom, že se datové typy porovnávají podle hodnoty. Toto porovnávání není příliš efektivní, jelikož nedokáže najít shodu, pokud atributy referenčních datových typů ukazují na různé instance se stejným obsahem. Tento problém řeší porovnávání do hloubky.

6.5.2 Porovnávání do hloubky

Hlavní rozdíl mezi porovnáváním do hloubky a mělkým porovnáváním je v referenčních datových typech. Z porovnávaných objektů se vytvoří datová struktura podobná stromu, která obsahuje referenční datové typy jako uzly a primitivní datové typy, nebo `String` jako listy. Potom stačí porovnat tyto dva stromy. Nejedná se přímo o strom,

jelikož v této datové struktuře můžou vzniknout cykly, které datová struktura stromu nepovoluje.

Porovnávání do hloubky dokáže nalézt všechny duplicity s časovou složitostí $O(V)$, kde V je celkový počet uzlů stromu (počet objektů na haldě). Tento přístup porovnávání vede na rekurzivní algoritmus, který lze předčasně přerušit, jestliže najde dvě odlišné hodnoty.

6.5.3 Detekce cyklů

Referenční datové typy na sebe mohou cyklicky odkazovat. Poté v grafu objektů vznikají cykly a při jejich porovnávání může dojít k zacyklení algoritmu. Je proto důležité využít mechanismus, který cyklus včas odhalí a tím zabrání zacyklení.

Jedná se o klasický grafový problém, lze tedy využít známé algoritmy detekce cyklu. Nejprve se označí veškeré uzly jako nenavštívené, a při procházení se postupně označují za navštívené. Pokud se algoritmus pokusí navštívit uzel, který již navštívený byl, průchod grafem se zastaví a objekty jsou prohlášeny za stejné.

6.5.4 Porovnávání polí

Nejprve se u polí porovná jejich velikost, jestli jsou velikosti stejné, pak se porovnávají hodnoty polí. Dále se pole porovnávají podle datového typu jeho hodnoty.

Pole s prvky primitivního datového typu

Pokud jsou v poli prvky primitivního datového typu, pak lze pole porovnat cyklem, který porovnává prvky na stejných pozicích. Balík `java.util` obsahuje metodu `Arrays.equals`, která porovná dvě pole stejného datového typu. Tato metoda je přetížená pro všechny primitivní datové typy. Není proto potřeba psát vlastní metodu pro porovnávání polí každého primitivního datového typu.

Pole s prvky instancí třídy

Pro prvky s datovým polem instance třídy lze v cyklu porovnat hodnoty na stejných pozicích. Tyto instance objektů jsou opět porovnávány do hloubky.

Vícerozměrná pole

Vícerozměrná pole jsou porovnávána rekurzivně. V cyklu se zjišťuje, zda jsou prvky na stejných pozicích další pole. Pokud nejsou, zavolá se metoda na porovnání hodnot prvků do hloubky.

6.6 Závěrečná sumarizace

Na konci běhu programu dostane uživatel zpětnou vazbu o běhu jeho programu. Tato sumarizace lze realizovat pomocí instrumentace metody `main`. Výpis by měl obsahovat počty duplikátů jednotlivých objektů, kde byly tyto objekty vytvořeny a kolik bytů jednotlivá třída, metoda, či řádka alokovala.

7 Implementace

Při implementaci praktické části této bakalářské práce byl kladen důraz na přesnost a správnost funkcionality. Dále byl kladen důraz na dobrou dokumentaci, jelikož se předpokládá, že kód se bude dále rozšiřovat například v dalších bakalářských či diplomových pracích. Kód by měl být dobře okomentovaný a pochopitelný i ostatními programátory.

7.1 Využité technologie

Jelikož se práce věnuje především jazyce Java, je i samotný nástroj napsán v jazyce Java verze 17 (OpenJDK 17, Hotspot). Z první kapitoly je ale zřejmé, že by nástroj mohl být napsán v jakémkoliv jiném jazyce, přeložitelném do Java byte kódu. K sestavení programu je využit nástroj Apache Maven, a to díky jeho jednoduché konfiguraci, dohledatelnosti a instalaci knihoven.

K úpravě byte kódu jednotlivých metod je využita knihovna Javassist verze 3.29. Hlavním důvodem výběru této knihovny je jednoduchost využití oproti ASM. Javassist je na rozdíl od ASM vysokoúrovňová a méně komplexní knihovna s vysokou úrovní abstrakce. V práci je využito API byte kódu i API zdrojového kódu.

Výpis je realizován pomocí knihovny Log4j 2. Tato knihovna nabízí vývojářům snadné spravování různých úrovní logování a ladění programů [29]. První verze knihovny Log4j byla široce využívána v mnoha aplikacích, v průběhu let se její vývoj zpomalil z důvodu nutnosti kompatibility s velmi starými verzemi Javy. Log4j verze 2 přinesla knihovně mnoho vylepšení, jako například podpora pro asynchronní logování, možnost rozšíření knihovny o libovolné vlastnosti, filtry logovacích zpráv a jejich vylepšená podpora pro formátování a logování. Druhá verze přináší také lepší výkon a škálovatelnost.

7.2 Struktura programu

Program je strukturován do dvou modulů.

agent

Tento modul obsahuje kód agenta, se kterým se poté spouští aplikace. Modul je rozdělený na dva balíky – kód agenta a kód, který je vložen do původní aplikace a je z ní volán.

Kód agenta má na starost veškerou transformaci načítaných tříd. Nejprve přečte soubor s prefixy tříd, které má ignorovat při transformaci a poté provádí transformaci. Agent také řeší výpočet velikosti objektů a vícerozměrných polí při jejich alokaci.

Kód, který se vkládá do aplikace pomocí agenta slouží ke zpracování alokovaných objektů. Je zde kód pro porovnávání instancí objektů, počítání velikostí polí a počítání alokací jednotlivých částí původního kódu.

tests

Modul obsahující jednotlivé testovací aplikace, pomocí kterých je nástroj testován.

7.3 Implementace agenta

Vstupní bod agenta je metoda `premain` ve třídě `MemoryAllocationAgent` Tato metoda uloží instanci instrumentace a přidá jí instanci transformeru. Metoda také inicializuje knihovnu Log4j konfiguračním souborem. Dále třída obsahuje metodu `getObjectSize` pro získání velikosti objektu a metodu `getMultidimensionalArraySize`, která se využívá pro výpočet velikosti vícerozměrných polí.

```
public static void premain(String args, Instrumentation inst) {
  instr = inst;
  Configurator.initialize(null, "log4j2.xml");
  instr.addTransformer(new MemoryAllocationDetectionTransformer());
}
```

Fragment kódu 7.1: Implementace metody `premain`

7.4 Transformace tříd

Transformace tříd probíhá ve třídě `MemoryAllocationDetectionTransformer`. Tato třída implementuje rozhraní `ClassFileTransformer` a tudíž implementuje metodu `transform`, kde dochází k transformaci byte kódu.

7.4.1 Filtrování tříd a balíků

Třída transformera obsahuje jednoduchou konfiguraci pomocí prefixového filtru, který říká, kterým třídám či balíkům chce uživatel editovat byte kód. Tento filtr je naplněn hodnoty z konfiguračního souboru. Každá řádka konfiguračního souboru odpovídá jednomu filtrovanému výrazu. Pokud má název třídy stejný prefix jako některá z hodnot z filtrovaných výrazů, třída se nebude transformovat.

```
private boolean filter(String className) {
  for (String prefix : prefixFilter) {
    if (className.startsWith(prefix)) {
      return false;
    }
  }

  return true;
}
```

Fragment kódu 7.2: Filtr názvů tříd

7.4.2 Transformace vstupní metody aplikace

Nástroj ke svému správnému fungování potřebuje upravit vstupní metodu instrumentované aplikace. Pro získání názvu třídy se vstupní metodou `main` nástroj přečte manifestové atributy v JAR souboru instrumentované aplikace. Poté porovná, zda je tato třída stejná jako právě instrumentovaná třída. Zároveň porovná, zda se instrumentovaná metoda nazývá `main`. Jestliže jsou obě podmínky splněny, metoda je upravena.

```
public static String getMainClass(ProtectionDomain protectionDomain) {
  String jarFile = protectionDomain.getCodeSource().getLocation()
                                                   .toURI().getPath();
  try(JarFile jar = new JarFile(jarFile)) {
    return jar.getManifest().getMainAttributes().getValue("Main-Class");
  }
}
```

Fragment kódu 7.3: Získání třídy obsahující metodu `main`

Po vykonání metody `main` jsou přidána volání dvou metod. První metoda má na starost výpis sumarizace informace o alokacích. Druhá metoda spouští algoritmus vyhledávání duplikátů instancí objektů v paměti. Volání metod je přidáno pomocí knihovny Javassist a jejího API zdrojového kódu.

7.4.3 Vyhledávání alokací a jejich transformace

Jak již bylo zmíněno v kapitole 2.5.2, v JVM existují 4 instrukce pro alokaci paměti. Kvůli různé velikosti instrukcí a jejich jinému chování je potřeba se každé této instrukcí věnovat individuálně. Vyhledávání a transformace je realizováno knihovnou Javassist pomocí API byte kódu.

`newarray`

Bytová instrukce `newarray` zabírá efektivně 2 byty. První byte je samotná instrukce. Druhým bytem je primitivní datový typ zakódovaný pomocí Tabulka *2.2*. Za tyto dva byty je vložen kód, který udělá kopii reference a zavolá metodu, která tuto referenci uloží do kolekce alokovaných objektů.

```
private void opcodeArrayAllocation(CodeIterator iterator, int pos,
        ConstPool constPool, CodeAttribute codeAttribute, int typeIndex) {
  int classInfo = constPool.addClassInfo("inject.AllocationDetector");
  String registerMethodName = arrayRegisterMethods[typeIndex];
  String registerMethodSignature = arrayRegisterMethodSignatures[typeIndex];

  int registerMethodConstPoolIndex = constPool.addMethodrefInfo(classInfo,
                              registerMethodName, registerMethodSignature);
  int effectiveSize = bytecodeEffectiveSize[typeIndex];

  Gap dupOpcodePos = iterator.insertGapAt(pos + effectiveSize, 4, true);
  iterator.writeByte(Opcode.DUP, dupOpcodePos.position);
  iterator.writeByte(Opcode.INVOKESTATIC, dupOpcodePos.position + 1);
  iterator.write16bit(registerMethodConstPoolIndex,
                                     dupOpcodePos.position + 2);
  codeAttribute.setMaxStack(codeAttribute.getMaxStack() + 1);
  codeAttribute.setMaxLocals(codeAttribute.getMaxLocals() + 1);
}
```

Fragment kódu 7.4: Vložení byte kódu k instrukci `newarray`

anewarray

Instrukce `anewarray` je velká 3 byty. První byte je samotná instrukce a následující dva byty společně určují index do fondu konstant. Za tyto 3 byty je vložen stejný kód, jako pro instrukci `newarray`. Z Fragment kódu *7.5*: Upravený byte kód instrukce `anewarray` je vidět, jak se změní byte kód po instrumentaci.

```
iload_5             # pole s pěti prvky
anewarray 00 21     # fond konstant na indexu 21
dup                 # kopie nové reference
invokestatic 00 49 # volání metody, která uloží referenci
```

Fragment kódu 7.5: Upravený byte kód instrukce `anewarray`

multianewarray

Efektivní velikost instrukce `multianewarray` je 4 byty. První byte je opět samotná instrukce. Další dva byty určují index do fondu konstant. Poslední byte udává počet dimenzí vícerozměrného pole. Implementace je stejná jako u instrukcí `anewarray` a `newarray`.

new

Instrukce `new` má jiné chování než instrukce pro alokaci polí. Rozdíl je v tom, že tato instrukce je vždy v kombinaci s instrukcí `invokespecial`. Je proto instrumentace instrukce `new` rozdělena na dvě části. První část instrumentace hledá pozici instrukce `new` a vloží za ní byte kód, který vytvoří kopii nové reference. Druhá část instrumentace vyhledává pro každou instrukci `new` příslušnou instrukci `invokespecial` a po jejím vykonání zavolá metodu, která referenci uloží do kolekce alokovaných objektů. Párování instrukcí `new` s instrukcí `invokespecial` je zajištěno pomocí datové struktury zásobník. Po nalezení instrukce `new` se na vrchol zásobníku přidá název třídy, kterou momentálně instrumentovaná instrukce alokuje. Při nalezení instrukce `invokespecial` se instrumentační metoda podívá na vrchol zásobníku, a pokud je název třídy stejný a obsahuje „`<init>`”, proběhne instrumentace a prvek na vrcholu zásobníku je odstraněn.

```
private void opcodeNew(CodeIterator iterator, int pos, ConstPool constPool,
                       CodeAttribute codeAttribute, Deque<String> stack) {
  String className = Utils.getClassInfo(iterator, pos, constPool);
  stack.push(className);

  Gap dupOpcodePos = iterator.insertGapAt(pos + 3, 1, true);

  iterator.writeByte(Opcode.DUP, dupOpcodePos.position);

  codeAttribute.setMaxStack(codeAttribute.getMaxStack() + 1);
  codeAttribute.setMaxLocals(codeAttribute.getMaxLocals() + 1);
}
```

Fragment kódu 7.6: Vložení instrukce `DUP` u instrukce `new`

```
private void newInsertInvoke(CodeIterator iterator, int pos,
  ConstPool constPool, CodeAttribute codeAttribute, Deque<String> stack) {

  String top = stack.peek();

  String className = Utils.getClassName(iterator, pos, constPool);
  String methodName = Utils.getMethodName(iterator, pos, constPool);

  if(!methodName.contains("<init>")) return;
  if(!className.equals(top)) return;
  stack.pop();

  int classInfo = constPool.addClassInfo("inject.AllocationDetector");
  int registerMethodIndex = constPool.addMethodrefInfo(
         classInfo, "registerObject", "(Ljava/lang/Object;)V"
  );

  Gap dupOpcodePos = iterator.insertGapAt(pos + 3, 3, true);
  iterator.writeByte(Opcode.INVOKESTATIC, dupOpcodePos.position);
  iterator.write16bit(registerMethodIndex, dupOpcodePos.position + 1);
}
```

Fragment kódu 7.7: Vložení instrukce `invokespecial` pomocí zásobníku

Tato metoda má jednu nevýhodu. V současné verzi nedokáže pracovat s dědičností a polymorfismem, jelikož zjišťuje, zda deklarovaný datový typ je stejný, jako typ pro který je volán konstruktor.

7.5 Ukládání alokovaných objektů

Ve třídě `inject.AllocationDetector` se nachází datová struktura pro ukládání alokovaných objektů. Datová struktura je typu: `Map<String, List<ObjectWithTrace>>`. Klíč do této mapy je název třídy. Hodnoty mapy jsou seznamy obsahující instance objektu typu `ObjectWithTrace`. Tento objekt obsahuje dvojici instance alokovaného objektu a `StackTraceElement` z místa, kde byl objekt alokován. Fragment kódu *7.8*: Přidávání objektu do kolekce alokovaných objektů ukazuje, jak jsou objekty přidávány do kolekce alokovaných objektů.

```
private static void addObject(Object obj, StackTraceElement stackTrace) {
  String className = obj.getClass().getName();
  if(!objectMap.containsKey(className)) {
    objectMap.put(className, new ArrayList<>());
  }

  List<ObjectWithTrace> list = objectMap.get(className);
  list.add(new ObjectWithTrace(obj, stackTrace));
}
```

Fragment kódu 7.8: Přidávání objektu do kolekce alokovaných objektů

Jelikož každá instance objektu tímto procesem projde právě jednou, je zaručené, že seznamy obsahují každou instanci objektů právě jednou. Dalo by se tedy říci, že tyto seznamy reprezentují množiny objektů stejného datového typu. K reprezentaci byl ovšem zvolen seznam, neboť reprezentace pomocí množin vyžaduje volbu klíče (`HashSet`), nebo implementaci porovnávání objektů (`TreeSet`).

7.6 Výpočet velikosti objektů

Velikost objektů je vypočítána metodou `getObjectSize`, kterou obsahuje třída `java.lang.Instrumentation`. Výjimka je u výpočtu velikostí vícerozměrných polí. Vícerozměrná pole jsou rekurzivně procházena a velikosti jednotlivých polí jsou akumulovány. Algoritmus prochází jednotlivá pole prvek po prvku, a zjišťuje, zda je prvek na dané pozici pole a neobsahuje hodnotu `null`. Pak rekurzivně volá

sám sebe s prvkem na dané pozici, dokud nenarazí na prvek, který není pole. K výpočtu velikostí menších polí je opět využitá metoda `getObjectSize`.

```
public static long getMultidimensionalArraySize(Object array) {
  long arraySize = getObjectSize(array);
  int length = java.lang.reflect.Array.getLength(array);

  for (int i = 0; i < length; i++) {
    Object element = java.lang.reflect.Array.get(array, i);
    if (element != null && element.getClass().isArray()) {
      arraySize += getMultidimensionalArraySize(element);
    }
  }

  return arraySize;
}
```

Fragment kódu 7.9: Získání velikosti vícerozměrného pole

7.7 Získání pozice alokace objektu v kódu

Pozice alokace lze získat ze zásobníkového rámce, ze kterého je metoda pro přidávání objektu do kolekce volána. K tomuto zásobníkovému rámci se dá dostat pomocí stopy zásobníku současného vlákna. K tomu slouží metoda Thread.currentThread().getStackTrace(). Tato metoda vrací pole prvků StackTraceElement. Některé verze JVM mohou v některých případech vynechat jeden nebo více zásobníkových rámců [30]. Proto je tato funkcionalita omezená verzí JVM, jelikož nelze za běhu programu zjistit, zda byly některé rámce vynechány.

7.8 Porovnávání objektů

Porovnávání objektů je realizováno pomocí rozhraní `IDuplicateEquals`. Toto rozhraní obsahuje metodu `eq`, která rozhoduje, zda jsou dva objekty rovné. Nástroj je vybaven dvěma třídami implementující toto rozhraní.

DuplicateShallowEquals

Třída `DuplicateShallowEquals` obsahuje implementaci porovnávání objektů pomocí mělkého porovnávání popsaného v kapitole 6.5.1. Algoritmus se nejprve pokusí porovnat třídy pomocí metody `equals`. Pokud metoda `equals` vyhodnotí, že jsou objekty rovné, pak algoritmus „věří“ implementaci metody `equals` a prohlásí, objekty za rovné. Jestliže metoda `equals` řekne, že objekty nejsou rovné, nemusí to být nutně pravda, jelikož její implementace není povinná. Dále algoritmus využívá reflexi k procházení jednotlivých atributů objektu a porovnává je pomocí metody `equals`. Jestliže dojde k neshodě v jednom z atributů, algoritmus skončí předčasně a prohlásí, že objekty jsou různé.

DuplicateDeepEquals

Třída `DuplicateDeepEquals` implementuje porovnávání objektů pomocí porovnání do hloubky popsaného v kapitole 6.5.2. Algoritmus nejprve pracuje podobně jako `DuplicateShallowEquals` tím, že se pokusí porovnat objekty pomocí metody `equals`. Pokud metoda `equals` prohlásí, že objekty nejsou rovné, pak začne prohledávání do hloubky. Pomocí reflexe a rekurzivního průchodu porovnává jednotlivé atributy objektu. Jestliže se objekty v jednom atributu neshodují, algoritmus může předčasně skončit a prohlásit objekty jako za různé. Zacyklení se předchází pomocí datové struktury `HashSet`. Tato datová struktura si ukládá unikátní číslo přidělené každému objektu JVM. Toto číslo se dá získat pomocí metody `System.identityHashCode`, která vrací hodnotu metody `hashCode`, jako by nikdy nebyla objektem překryta. Pokud datová struktura již obsahuje oba porovnávané objekty, označí je za stejné a algoritmus pokračuje. V opačném případě oba objekty do datové struktury přidá a dojde k jejich porovnání.

7.9 Testování

8 Výsledky

9 Závěr

Literatura

[1] “Chapter 1. Introduction.” https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-1.html (accessed Mar. 29, 2023).

[2] “Chapter 2. The Structure of the Java Virtual Machine.” https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html (accessed Mar. 29, 2023).

[3] T. A. Gamage, “Understanding Java Memory Model,” *Platform Engineer*, Oct. 03, 2022. https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973 (accessed Mar. 29, 2023).

[4] “Generational Hypothesis,” *Plumbr – User Experience & Application Performance Monitoring*, May 12, 2015. https://plumbr.io/handbook/garbage-collection-in-java/generational-hypothesis (accessed Mar. 30, 2023).

[5] “Oracle JVM Garbage Collectors Available From JDK 1.7.0_04 And After.” http://www.fasterj.com/articles/oraclecollectors1.shtml (accessed Mar. 30, 2023).

[6] J. Gesso, “Java Memory Management for Java Virtual Machine (JVM) | Betsol,” Jun. 02, 2017. https://www.betsol.com/blog/java-memory-management-for-java-virtual-machine-jvm/ (accessed Mar. 30, 2023).

[7] “Chapter 6. The Java Virtual Machine Instruction Set.” https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-6.html (accessed Mar. 30, 2023).

[8] “Chapter 4. The class File Format.” https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-4.html (accessed Mar. 30, 2023).

[9] rollbarnew, “How to Detect Memory Leaks in Java: Causes, Types, & Tools,” *Rollbar*, Apr. 01, 2021. https://rollbar.com/blog/how-to-detect-memory-leaks-in-java-causes-types-tools/ (accessed Apr. 01, 2023).

[10] K. Kawachiya, K. Ogata, and T. Onodera, “Analysis and reduction of memory inefficiencies in Java strings,” in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, in OOPSLA ’08. New York, NY, USA: Association for Computing Machinery, jen 2008, pp. 385–402. doi: 10.1145/1449764.1449795.

[11] “String (Java Platform SE 8).” https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#intern-- (accessed Apr. 01, 2023).

[12] “Duplicate Objects in Java: Not Just Strings - DZone,” *dzone.com*. https://dzone.com/articles/duplicate-objects-in-java-not-just-strings (accessed Apr. 03, 2023).

[13] “openjdk/jdk7u.” OpenJDK, Sep. 14, 2022. Accessed: Apr. 03, 2023. [Online]. Available: https://github.com/openjdk/jdk7u/blob/6f892a5354bd5063418d74e32ded9653d24a768d/jdk/src/share/classes/java/util/ArrayList.java

[14] “openjdk/jdk8u.” OpenJDK, Apr. 01, 2023. Accessed: Apr. 03, 2023. [Online]. Available: https://github.com/openjdk/jdk8u/blob/89aeae16e85ddfbd581cb86d0b0480b1e2d50e99/jdk/src/share/classes/java/util/ArrayList.java

[15] “JDK Mission Control.” https://www.oracle.com/java/technologies/jdk-mission-control.html (accessed Apr. 03, 2023).
[16] “visualgc - Visual Garbage Collection Monitoring Tool.” https://www.oracle.com/java/technologies/visual-garbage-collection-monitoring-tool.html#VisualGC (accessed Apr. 04, 2023).
[17] H. Hoang ngoc, “Detekce problémů se správou paměti v Java aplikacích,” *Memory bloat in Java programs*, 2021, Accessed: Apr. 04, 2023. [Online]. Available: http://dspace5.zcu.cz/handle/11025/44778
[18] “Java Instrumentation,” *Javapapers*. https://javapapers.com/core-java/java-instrumentation/ (accessed Mar. 30, 2023).
[19] “Java Development Kit Version 17 API Specification.” https://docs.oracle.com/en/java/javase/17/docs/api/java.instrument/java/lang/instrument/package-summary.html (accessed Mar. 30, 2023).
[20] “Chapter 6. Aspect Oriented Programming with Spring.” https://docs.spring.io/spring-framework/docs/2.5.5/reference/aop.html (accessed Apr. 05, 2023).
[21] M. Zikmund, “Aspektově orientované programování a jeho podpora,” Masarykova univerzita, Fakulta informatiky, 2010. Accessed: Apr. 05, 2023. [Online]. Available: https://is.muni.cz/th/lwqfj/
[22] baeldung, “Introduction to Advice Types in Spring | Baeldung,” Dec. 17, 2015. https://www.baeldung.com/spring-aop-advice-tutorial (accessed Apr. 05, 2023).
[23] “ASM.” https://asm.ow2.io/index.html (accessed Apr. 05, 2023).
[24] S. Setunga, “Introduction to Java ByteCode Manipulation with ASM,” *Medium*, Jan. 07, 2020. https://supunsetunga.medium.com/introduction-to-java-bytecode-manipulation-with-asm-9ae71049c7e0 (accessed Apr. 05, 2023).
[25] baeldung, “A Guide to Java Bytecode Manipulation with ASM | Baeldung,” Oct. 23, 2017. https://www.baeldung.com/java-asm (accessed Apr. 05, 2023).
[26] “Javassist by jboss-javassist.” https://www.javassist.org/ (accessed Feb. 21, 2023).
[27] S. Chiba, “Load-Time Structural Reflection in Java,” in *ECOOP 2000 — Object-Oriented Programming*, E. Bertino, Ed., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2000, pp. 313–336. doi: 10.1007/3-540-45102-1_16.
[28] “API reference for Java Platform, Standard Edition.” https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/StackTraceElement.html (accessed Apr. 05, 2023).
[29] “Log4j – Overview.” https://logging.apache.org/log4j/2.x/manual/index.html (accessed Mar. 15, 2023).
[30] “Java Development Kit Version 17 API Specification.” https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html#getStackTrace() (accessed Apr. 05, 2023).