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

Diplomová práce

Nástroj pro analýzu Java memory heapu

Plzeň, 2019 Martin Mach

PROHLÁŠENÍ

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

V Plzni dne 13. května 2019 . . . . . .

Martin Mach

ABSTRAKT

Abstrakt CZ

KLÍČOVÁ SLOVA

Klíčová slova CZ

ABSTRACT

Abstract EN

KEYWORDS

Keywords EN

MACH, Martin *Nástroj pro analýzu Java memory heapu*: diplomová práce. Plzeň: Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2019. 67 s. Vedoucí práce byl Ing. Richard Lipka, Ph.D.

OBSAH

1 Úvod **8**

2 Problém správy paměti **10**

2.1 Garbage collector 10

3 Struktura paměti programu **11**

3.1 Struktura JVM stacku 12

3.1.1 Rámce 12

3.2 Program counter registr 13

3.3 Stack nativních metod 13

3.4 Heap 13

3.5 Non-heap oblast 14

3.5.1 Oblast metod 14

3.6 Runtime constant pool 15

3.6.1 Řetězce 15

3.7 Správa a struktura Java heapu 15

3.7.1 Garbage collection 16

3.7.2 Heap Dump 18

3.7.3 Zpracování dumpu 19

3.7.4 Binární formát dumpu heapu 20

4 Práce s pamětí **22**

4.1 Optimalizace užití paměti 22

4.2 Analýza za běhu programu 22

4.3 Analýza memory dumpu 23

5 Java Virtual Machine **24**

5.1 Java Bytecode 24

5.2 Class loader 25

6 Existující nástroje pro analýzu heapu **26**

6.1 Eclipse MAT 26

6.1.1 Základní informace 26

6.2 VisualVM 27

6.2.1 Základní informace 27

6.3 Java Mission Control 28

6.3.1 Základní informace 28

6.4 JProfiler 29

6.4.1 Základní informace . . . 30
6.5 JHAT . . . 30
6.5.1 Základní informace . . . 31
6.6 OQL . . . 31
6.7 Knihovny . . . 32

7 Možnosti analýzy **33**
7.1 Rovnost objektů . . . 33
7.2 Rovnost složitějších objektů . . . 35
7.2.1 Porovnávání referencovaných objektů . . . 35
7.2.2 Porovnávání řetězců . . . 37
7.2.3 Porovnávání polí a kolekcí . . . 37
7.3 Efektivita využití polí a kolekcí . . . 37

8 Návrh implementace **39**
8.1 Volba platformy . . . 39
8.2 Způsob zpracování heapu . . . 40
8.3 Hprof Heap Dump parser . . . 41
8.3.1 Základní informace . . . 41
8.3.2 Základní struktura . . . 41
8.3.3 RecordHandler . . . 42
8.3.4 Definované entity . . . 44
8.3.5 Použití knihovny . . . 45
8.4 Návrh struktury aplikace . . . 47
8.5 Rozhraní pipeline a modulů . . . 48

9 Implementace **49**
9.1 Zpracování dumpu . . . 49
9.2 Analýza plýtvání pamětí . . . 50
9.2.1 Analýza duplicit . . . 51
9.2.2 Analýza kolekcí . . . 51

10 Ověření implementace **53**
10.1 Testovací aplikace . . . 53
10.2 Testování Spring Boot . . . 56
10.3 Intepretace výsledků . . . 57

11 Závěr **59**

Seznam obrázků **60**

Seznam zkratek **62**

Literatura **63**

Seznam příloh **65**

A Přílohy **66**
A.1 Uživatelská dokumentace 66
A.2 Výpis programu 67

1 ÚVOD

Při vývoji programů se běžně můžeme setkat s neuspokojivými parametry jejich provozu – ať už z hlediska času běhu nebo využití systémových zdrojů. Častým řešením je prosté navýšení těchto prostředků, což dává smysl – cena hardware klesá [1]. Investice do jeho nákupu tak může být nižší, než částka vydaná za případnou práci programátorů, jejichž mzdy naopak rostou [2]. To sice v některých případech může pomoci, nicméně problém samotný to neřeší – pouze oddaluje. Pokud chceme podobným problémům v budoucnu předejít a vyhnout se dalším investicím do výkonu, musíme problémový program upravit – optimalizovat.

Optimalizace může probíhat na několika úrovních. První z možností může být volba efektivnějšího algoritmu – můžeme zkontrolovat, zda jsme pro řešení problému zvolili vhodný algoritmus a zda neexistuje způsob, jakým bychom mohli dotyčné části řešit efektivněji. Tím je možné, především u vyššího počtu zpracovávaných prvků, dosáhnout výrazně rychlejšího běhu programu samotného. Dále je možné rychlost běhu ovlivnit zefektivněním (či prostou redukcí) přístupů k zařízením (IO), ať už se jedná o práci s diskem nebo síťovou činnost. Dále můžeme optimalizovat na úrovni využití systémových prostředků, typicky paměti (RAM). Právě optimalizaci paměťových nároků se v této práci budu věnovat.

I na snižování množství programem užité paměti lze pohlížet z několika různých úhlů. Navrhovaná řešení se mohou výrazně lišit dosaženými úsporami zdrojů, náročností či finanční nákladností. Jako první můžeme zvážit používané technologie. Změna programovacího jazyku je často natolik drahá, že kompletní přepsání dotyčného software ve většině případů nedává, i přes dosaženou optimalizaci, ekononomický smysl. V rámci námi používaného jazyku tak můžeme, kromě našeho kódu samotného, analyzovat používané frameworky a knihovny. V praxi se můžeme setkat s tím, že závislost na knihovně je přidána pouze z důvodu využití jedné či několika jejích funkcionalit. To je vhodné během vývoje z důvodu rychlosti; později je možné některé z těchto knihoven a jimi poskytované funkce nahradit implementací vlastní a teoreticky tak ušetřit jednotky, desítky či stovky megabytů paměti. Tím ovšem neoptimalizujeme data programu, ale velikost programu samotného. Ta často bývá řádově nižší, než paměť spotřebovaná datovými strukturami.

Na řadu tak přichází právě programová data – konkrétně datové struktury a typy, které v aplikaci jako její autoři využíváme. A opět je možné k tomuto problému přistoupit z různých úhlů, lišících se pak především hloubkou a důsledností analýzy. Můžeme řešit, zda námi používané typy odpovídají povaze dat – například, zda rozsah celočíselného typu odpovídá maximální hodnotě dané veličiny.

V praxi mohou být následky nedůsledné optimalizace ještě vážnější. Systémy, které mají obsloužit vysoký počet požadavků za sekundu, je často nutné škálovat – vytvářet nové, nezávislé jednotky těcho systémů. Ty se poté mohou střídat o příchozí požadavky.

Každé plýtvání je tedy znásobeno počtem těchto instancí. To může naprosto zbytečně zvyšovat náklady na provoz; a to ať už v případě vlastního serveru či cloudových služeb (tzv. serverless). Zlepšení správy paměti navíc může pozitivně ovlivnit i rychlost běhu programu – ubyde počet spuštění GC (viz dále).

Cílem této práce je zanalyzovat, k jakým nedostatkům dochází (a zda vůbec) z pohledu neefektivního využití paměti při vývoji Java aplikací. Dále pak vytvořit nástroj, který na tyto nedostatky dokáže poukázat a uživateli napovědět, jakým způsobem by mohl použité prostředky svého programu redukovat a jeho běh tedy optimalizovat. Korektní fungování vytvořeného nástroje bude ověřeno na testovací aplikaci z pohledu správnosti a úplnosti výsledků. Následně pak také prověřeno na větších, rozšířených a běžně používaných Java aplikacích, za účelem posouzení efektivity použití nástroje v praxi, případně také v komerční sféře.

2 PROBLÉM SPRÁVY PAMĚTI

Při vytváření programu má jeho autor na výběr ze dvou způsobů správy paměti – spravované automaticky (typicky mechanismem typu Garbage Collector (GC) apod.) či manuálně, případně kombinací těchto přístupů. Ne každý jazyk nabízí oba – typicky je k dispozici pouze jeden z přístupů, často je správa GC vynucena. Vzhledem k tomu, že majorita nejpopulárnějších programovacích jazyků za poslední roky se řadí mezi vysokoúrovňové, na toto vynucení GC narazíme u většiny z nich, včetně Javy [4][5]. Výjimkou jsou populární nízkoúrovňovější jazyky typu C a C++.

2.1 Garbage collector

GC je nástroj, starající se o správu paměti programu – její přidělování, kontrolu a následné uvolnění, ať už pokud je jí málo a je zapotřebí jinde, v pravidelných intervalech nebo při jejím zaplnění. GC je obecný termín, tj. neodkazuje na žádnou konkrétní implementaci a způsob chování. Často je v rámci jednoho jazyka (respektive běhového prostředí) zároveň implementováno hned několik algoritmů GC a dle okolností je vybrán ten nejefektivnější a v danou chvíli nejvhodnější z nich. Některé algoritmy tak mohou běžet velmi rychle bez minimálního zásahu do běhu programu, zatímco jiné vyžadují pro svůj běh o něco delší čas. Často tak je nutné všechen běh kódu pozastavit; v takových případech toto spuštění GC nazýváme *stop-the-world* („zastavení světa“, běhu). Pro program je toto zastavení transparentní.

Obecně GC funguje tak, že si udržuje seznam referencí na jednotlivé objekty, respektive jejich počet [18]. Pokud je některý z objektů dále nereferencovaný, při dalším běhu GC bude jím zabíraná paměť uvolněna. Nereferencovaným objektem rozumíme, že je nedosažitelný – nikdo na něj neukazuje. K takovým případům samozřejmě může docházet i v případě jazyků, které fungují bez GC, např. `C`. Pokud daný jazyk nezná jiný způsob, jak se k dané paměti opětovně dostat, dochází k tzv. *memory leakům*, tedy únikům paměti. V případě například výše zmíněného `C` nelze definitivně rozhodnout o nedostupnosti paměti – díky ukazatelové aritmetice je možné paměť zpětně dopočítat i v případě, že v jednom časovém okamžiku na něj žádný ukazatel v paměti programu neukazuje. Java koncept ukazatelové aritmetiky nezná, po odstranění poslední reference si tedy můžeme být jisti, že už znovu referencovat nikdy nepůjde. V souvislosti s *memory leaky* je nutné zdůraznit, že v tomto případě mám na mysli odstranění poslední reference v rámci paměti uživatelského programu. Java Virtual Machine (JVM), který paměť spravuje, adresu objektu stále zná a je tak schopný jej v rámci běhu GC odstranit.

3 STRUKTURA PAMĚTI PROGRAMU

Abychom mohli hledat v paměti a analyzovat nedostatky v rámci jejího využití, musíme nejprve porozumět její struktuře, různým typům oblastí a objektům v nich uložených. Přestože se mnoho následujících konceptů a pravidel vztahuje na ostatní jazyky (a v některých případech i na paměť spravovanou samotným operačním systémem), budu se primárně zaměřovat Javu. Specifikace JVM neobsahuje žádnou konkrétní podobu paměti či její rozložení, stejně tak jako nespecifikuje žádný konkrétní algoritmus GC a žádné optimalizace, které se mají nad běžícím kódem vykonávat. Tyto implementační detaily jsou ponechány na možnostech a tvůrčích schopnostech vývojářů, kteří standard implementují. Prostým požadavkem tak je korektní načtení dat ze souboru `class` a validní vykonání příslušných instrukcí.

JVM, stejně jako většina ostatních běhových prostředí a jazyků, rozeznává dva druhy datových typů – primitivní a referenční. Proměnné primitivního typu tak obsahují přímo danou hodnotu, zatímco proměnné referenčních typů referencují (tj. ukazují na) jinde umístěnou paměť, v níž se nachází objekt.

Primitivních datových typů je v Javě několik druhů, konkrétně:

- numerické
  - celočíselné
    - `byte`
    - `short`
    - `int`
    - `long`
    - `char`
  - s plovoucí desetinnou čárkou
    - `float`
    - `double`
- `boolean`
- `returnAddress`
  - Speciální datový typ, sloužící jako pointer na `JVM` instrukce, konkrétně `jsr`, `ret` a `jsr_w`. V běžném programu jej nicméně nemůžeme jako vývojáři použít ani modifikovat; `JVM` jej však zná a využívá.

Referenční datové typy jsou potom následující:

- `class`
- `array`
- `interface`

Typ `array`, tedy pole, samozřejmě obsahuje seznam prvků určitého typu – ten nazýváme typem komponenty. Proměnná referenčního typu je ukazatelem do paměti – místa, kde je obsah objektu umístěn. Prázdný ukazatel, který nereferencuje žádné místo v paměti,

nazýváme `null`. Tato hodnota je výchozí pro všechny proměnné referenčního typu, dokud jim není přiřazena hodnota [21].

V násedujících odstavcích popisuji různé části paměti `JVM` – viz obrázek 1.

3.1 Struktura JVM stacku

V případě, že v Javě mluvíme o stacku (zásobníku), typicky máme na mysli JVM stack. Kromě něj totiž ještě existuje nativní stack, který je vytvořený operačním systémem pro potřeby JVM samotného. Kromě toho je využívaný i pro některé nízkoúrovňovější činnosti. JVM stack je spravovaný běhovým prostředím samotným a slouží pro potřeby aplikací, které v rámci prostředí běží.

Pro každé vlákno aplikace je vytvořený stack. Do něj jsou ukládány lokální proměnné, hodnoty parametrů metod a návratové hodnoty. Rovněž se stará o volání metod, respektive o návrat do volací metody po zavolání metody jiné. Stejně jako v jiných jazycích (třeba C) je paměť lokálních proměnných automaticky uvolněna při odebírání hodnot ze stacku. Takto alokovanou paměť tak není nutné spravovat GC (či v případě C ručně uvolňovat).

Právě velikost JVM stacku je často omezující faktor při vývoji, na který v některých případech (např. nesprávně zastavovaná rekurze) narážíme. Jeho velikost můžeme nastavit jako parametr při spuštění programu – konkrétně `Xss` (respektive `-XX:ThreadStackSize`). Jeho výchozí velikost je závislá na velikosti dostupné (virtuální) paměti. V souvislosti s nedostatkem paměti a velikostí stacku se můžeme setkat se dvěma typy výjimek. `StackOverflowError` je vyhozena v případě, že během výpočtu narazíme na horní hranici velikosti stacku. `JVM` se případně může pokusit velikost dynamicky zvýšit. Pokud však během této operace narazí na velikost výše zmíněné paměti (což je už limitace nastavená operačním systémem a případně i hardwarovou konfigurací), je vyhozena výjimka jiná – `OutOfMemoryError`. Tato výjimka je rovněž produkována v případě, že ručně nastavíme velikost stacku takovou, že při jeho vytváření `JVM` narazí na limity paměti rovnou.

3.1.1 Rámce

Rámce (*frame*) přísluší právě jednomu stacku [23]. Data v něm jako takovém jsou totiž neměnná; jsou pouze přidávány/odebírány odkazy na rámce. V každém okamžiku je pro jedno vlákno aktivní právě jeden rámec.

Lokální proměnné

Každý rámec obsahuje pole s lokálními proměnnými. Vzhledem k tomu, že je rámec vytvořen a vložen do JVM stacku při volání metody a znovu odebrán a zničen při návratu

z ní, právě toto pole zajišťuje uvolnění paměti alokovaných proměnných. Kromě proměnných primitivního typu jsou v tomto poli uloženy rovněž reference na objekty v heapu (viz dále), jejich uvolnění sníží počet referencí na daný objekt a v případě dosažení nulové hodnoty je tento objekt připraven na uvolnění pomocí GC.

Stack operandů

Tento stack slouží pro mezivýpočty při provádění operací, převážně matematických.

3.2 Program counter registr

Program counter (PC) je registr s adresou ukazující na operaci, která se má provést. Vzhledem k tomu, že Java umožňuje vícevláknový běh, má samozřejmě každé vlákno svůj PC.

3.3 Stack nativních metod

Pokud má daná implementace JVM podporovat i tzv. `native` metody, měla by obsahovat i stack nativních metod. Zdůrazňuji, že *měla* – specifikace zde benevolentně ponechává rozhodnutí na konkrétním řešení a tento stack nutně nevynucuje. Díky konceptu nativních metod je možné přímo z Javy volat kód napsaný např. v C.

Stejně jako JVM stack, i stack nativních metod v některých případech produkuje výjimky. Toto chování je totožné pro oba stacky; stejně tak typy těchto chybových stavů jsou stejné.

3.4 Heap

Heap, tj. *halda*, je část paměti, která je pro všechna vlákna programu společná a v Javě tomu není jinak. Díky tomu, že jsou všechny objekty alokovány právě v heapu, k nim můžeme přistupovat napříč metodami, objekty i vlákny. Platnost v ní umístěných objektů tak není omezena žádným blokem platnosti (snad jen s výjimkou běhu programu samotného) – nedojde k uvolnění jimi zabírané paměti automaticky při opuštění tohoto bloku tak, jak je tomu v případě stacku [7][8]. Právě kvůli tomu nad heapem operuje GC, který nepotřebné objekty vyhledává a jejich paměť uvolňuje.

Kromě alokovaných objektů jsou v heapu rovněž uchovávány instanční proměnné. Právě z toho důvodu je nutné si dávat pozor na souběh při běhu ve více vláknech – všechna při přístupu k instanční proměnní manipulují se stejným objektem.

I heap má samozřejmě omezenou velikost. Ta se dá nastavit dvěma přepínači:

- `Xms` – Výchozí velikost heap, s kterou Java nastartuje.

- `Xmx` – Maximální velikost.

V případě 32 bitového systému jsou horní hranicí pro maximální velikost heapu 4 Gb, stejně jako v případě velikosti RAM (bez použití různých triků a rozšíření, jako třeba *PAE*, apod.). Jak vyplývá z existence výše zmíněných přepínačů, JVM dokáže s velikostí heapu dynamicky manipulovat. Program, respektive prostředí pro něj, tak spustí s výchozí hodnotou velikosti a v případě potřeby ji rozšiřuje (nebo naopak zmenšuje) až do maxima. Pokud si běh programu žádá více paměti v heapu, než může JVM alokovat (tj. než má od systému k dispozici), vyhod, stejně jako v případě stacku, výjimku `OutOfMemoryError` [9].

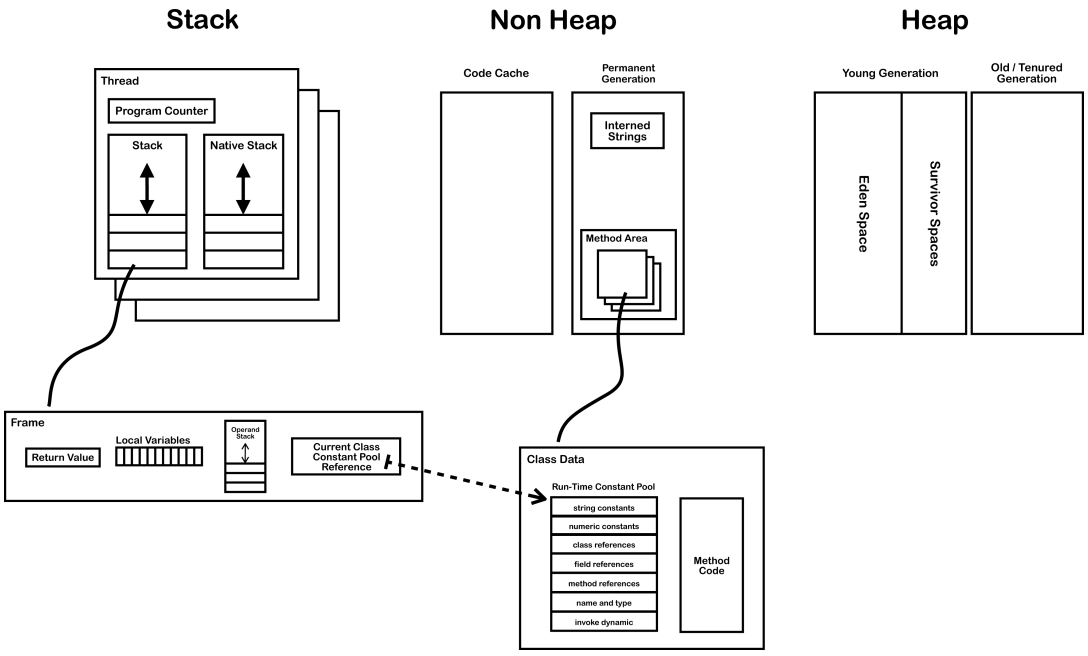
Heap je rozdělen dále do několika prostorů, jak je popsáno dále v kapitole 3.7.

3.5 Non-heap oblast

Kromě oblasti metod bsahuje pomocná data, která jsou potřeba pro interní zpracování a potřeby.

3.5.1 Oblast metod

Oblast, která uchovává data jednotlivých metod. Pro každou načtenou třídu uchovává strukturu se seznamem metod, statickými proměnnými a kódem metod a konstruktorů. Na jednu ze struktur třídy směřuje ukazatel z aktuálně zpracovávaného rámce v stacku, pomocí kterého může aktuálně vykonávaný kód přistupovat ke statickým členům třídy.



Obr. 1: Struktura Java paměti [27].

Ukázka 3.1: Příklad uchovávání řetězců a metody `intern`

```
String s1 = "hello";
String s2 = "hello";
String s3 = new String("hello");

System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
System.out.println(s1 == s3.intern()); // true
```

3.6 Runtime constant pool

Runtime constant pool (volně přeloženo jako *fond konstant pro běh*) je tabulka symbolů, která uchovává různé konstanty a informace nutné pro běh. Obsahuje následující položky.

Informace o třídách Obsahuje některé informace o samotných třídách a rozhraních.

- Pro třídy nebo rozhraní, která nejsou polem, obsahuje přímo jejich název
- Pro pole promitivního typu o `n` dimenzích začíná `n`-krát znakem `[`, pak následuje jméno komponenty pole. Pokud je toto navíc neprimitivního typu, jako prefix je přidán symbol `L` a postfix `;`.

Informace o třídních promenných Obsahuje název a typ.

Informace o metodách Obsahuje název a informace z hlavičky metody.

Informace o rozhraních Obsahuje názvy rozhraní a případně i jejich metody.

Různé typy parametrů, návratové typy atd.

3.6.1 Řetězce

Kromě toho jsou zde uloženy i řetězce (`String`) – to platí však pouze pro tzv. internalizované řetězce. Některé řetězce budou uloženy přímo v heapu. Java při vytváření řetězců může použít existující literál uložený v paměti tak, aby tento řetězec zbytečně nezabíral v paměti místo dvakrát či vícekrát. Toto chování lze případně vynutit voláním metody `intern` nad daným řetězcem – viz ukázka 3.1.

3.7 Správa a struktura Java heapu

Paměťovému modelu v Javě se věnuje *JSR-133*, nicméně přesně nespecifikuje konkrétní rozdělení paměti a způsob jejího přidělování a uvolňování [6]. Následný popis se tedy věnuje implementaci od společnosti Oracle – HotSpot. Zde je paměť rozdělena na 2 logické celky – *young generation space* a *old generation space*. Tato paměť, tedy heap, je rozdělena pouze v rámci JVM a následně je mapována na skutečnou fyzickou paměť.

Young generation space, tedy doslova “prostor mladé generace”, je dále rozdělen na *eden space* a prostory *S0* a *S1*. Eden space slouží k vytváření nových instancí objektů, je zde tedy vyhrazena část paměti nově vytvořenému objektu. Pokud v tomto prostoru není volno, proběhne uvolnění paměti (viz dále) přesunutím některých objektů do S0. Každý takový objekt obsahuje informaci o tom, kolik takových uvolnění daný objekt „přežil“.

Po určitém počtu takových přežití (či jinak také povýšení) je objekt přenesen do objektů staré generace, konkrétně *Tenured space*.

Toto rozdělení objektů do jednotlivých prostorů se jeví jako zbytečná komplikace, má však řadu výhod. První z nich je rychlost – nejvíce operací uvolnění je prováděno právě nad eden spacem, který je z prostorů nejmenší. Dále jsou tak objekty rozdělovány do skupin s podobnou charakteristikou (podobný věk, podobný počet a styl referencí apod.), na kterými je poté možné spustit rozdílné, pro dané skupiny specifické algoritmy pro jejich uvolnění.

3.7.1 Garbage collection

Spuštění GC v Javě nelze vynutit ručně. Systému lze *doporučit* jeho spuštění voláním metody `System.gc()`; JVM se tím ale nemusí řídit a toto volání jednoduše ignorovat [10].

I v Javě můžeme narazit na problém úniků paměti, tzv. *memory leaků*. Typicky se tento problém týká nízkoúrovňových jazyků typu C, nebo takových jazyků, kde je správa paměti v kompletní kompetenci autora programu. Často dojde k „zapomenutí“ některého ukazatele. Jeho smazáním se paměť stává nedostupnou a protože v daném jazyku není GC, bude uvolněna teprve ukončením programu – operačním systémem samotným. Toto chování je nebezpečné, protože pokud program poběží dlouhou dobu a bude alokovat paměť bez jejího následného uvolnění, dříve nebo později narazí na limit kladený ze strany operačního systému. Rovněž může jeho provozování být nepříjemné pro provozovatele programu, protože i když jeho běh operační systém neukončí, program bude zabírat zbytečně velké množství paměti.

Ve spojení s GC by tedy nemělo k únikům paměti typicky dojít. V Javě k nim může dojít především při nedůsledném používání vlastních zavaděčů tříd – *class loaderů* [12]. Za únik paměti můžeme rovněž považovat neuzavřený popisovač otevřeného souboru, databáze či jiného zdroje. Pokud k němu ztratíme přístup, např. po vyhození výjimky bez uzavření tohoto popisovače v bloku `catch` či lépe `finally`, ztrácíme tím, spolu s popisovačem, i menší množství paměti. Při častějším výskytu problému ale v tomto případě pravděpodobně narazíme na horní limit popisovačů Javy či operačního systému – ani tento zdroj není neomezený.

V Javě je několik různých implementací GC. Ty se liší rychlostí a efektivitou běhu a každý se může specializovat na jinou činnost [19]. Z toho některé GC dokáží většinu

práce odvést simultánně s během aplikace (tzv. konkurentní GC); jiné vyžadují dočasné zastavení aplikace.

CMS GC

Algoritmus CMS GC prozkoumává paměťový prostor heapu pomocí běhu ve více vláknech (tak, aby byl pokud možno co nejrychlejší). Skenuje ho tak, aby nalezl nepoužívané objekty. Ty postupně označí pro uvolnění a nakonec nad nimi provede iteraci a všechny uvolní. Takový typ běhu GC je vhodný zejména pro aplikace, které preferují pokud možno co nejkratší dobu jeho běhu. Zároveň se aplikace dělí s GC o výkon procesoru na všech jeho vláknech – CMS je význačný tím, že se co nejvíce práce pokouší vykonat přímo za běhu aplikace. Její běh zastavuje během jednoho cyklu svého běhu dvakrát – poprvé pro označení dosažitelných objektů, tzv. *initial mark pause* fáze. Druhá pauza aplikace proběhne po souběžném běhu zároveň s aplikací a označí zbylé objekty, které nebyly během konkurentního běhu nalezeny. Tato fáze se nazývá *remark pause* [20].

Seriový GC

Typ běhu GC, který zastaví všechna vlákna a provede nad nimi operaci uvolnění. Vzhledem k typu běhu je vhodný pouze pro menší aplikace nebo jednovláknové stroje. Všechny operace uvolnění běží pouze v jednom vlákně – tato vlastnost jej činí zároveň velikce efektivním, protože není zapotřebí žádná synchronizace ani mezivláknová komunikace. Díky tomu, že zmrazí všechna ostatní vlákna, je vysloveně nevhodný pro aplikace, které se na vícevláknový běh spoléhají – typicky například server, který musí poslouchat na portu a obsluhovat příchozí provoz.

Paralelní GC

Je výchozí volbou pro JVM (pokud jej umožňuje hardware), až do Javy 8 (včetně). Jak vyplývá z názvu, jeho běh je rozdělen do více vláken, narozdíl od seriového GC. Je určen pro aplikace velkého rozsahu, které běží ve více vláknech (a na systémech, které tento běh umožňují). Konkrétní počet vláken, ve kterých GC běží, je zvolen automaticky na základě dostupných zdrojů – konkrétně paralelních vláken, které je schopen daný procesor obsluhovat zároveň. Kromě toho je tento počet rovněž možné nastavit ručně (pomocí parametru `-XX:ParallelGCThreads`).

Při běhu paralelního GC má tento typ priority – kritéria, která musí splnit. Jejich priorita je v následujícím pořadí (sestupně):

Maximální čas běhu Nastavuje horní časový limit (v milisekundách), jak dlouho má být trvání jedné pauzy. Výchozí hodnota dobu trvání neomezuje. Pokud ručně nastavíme

kratší čas, než je pro algoritmus optimální, můžeme výrazně snížit propustnost algoritmu.

Propustnost Poměr času stráveného zpracováním garbage kolekce oproti času stráveného mimo ni (tj. času využívaného aplikací). Např. hodnota 9 nastaví tento poměr na 10% ($\frac{1}{1+9}$). Výchozí hodnota je 1%.

Minimální *footprint* (stopa, využití prostředků) Zajišťuje dodržení velikosti heapu.

Při běhu GC je postupováno shora, tj. na propustnost je brán ohled pouze za podmínky, že je splněna maximální doba běhu. Analogicky je splněna podmínka minimálního footprintu.

G1 GC

G1 znamená *Garbage First* a je výchozí volbou pro algoritmus od Javy 9 (včetně). Je určený pro heapy větší velikosti, které pak rozdělí do několika oblastí – regionů. Ty poté paralelně prohledává; prioritu přikládá regionům, které mají nejvyšší poměr zbytečných objektu (což odhaduje heuristicky). Oprito parelelnímu GC nabízí stabilnější běh algoritmu a kratší pauzy.

3.7.2 Heap Dump

Heap dump je textová nebo binární reprezentace paměti, kterou je možné uložit na disk a zachycuje aktuální stav aplikace. Při vytváření je činnost aplikace pozastavena. Dump je možné následně analyzovat a dále zpracovávat, je tak možné prozkoumat vnitřní stav aplikace v určitém bodě a např. řešit příčiny neočekávaného chování.

Vytvoření dumpu

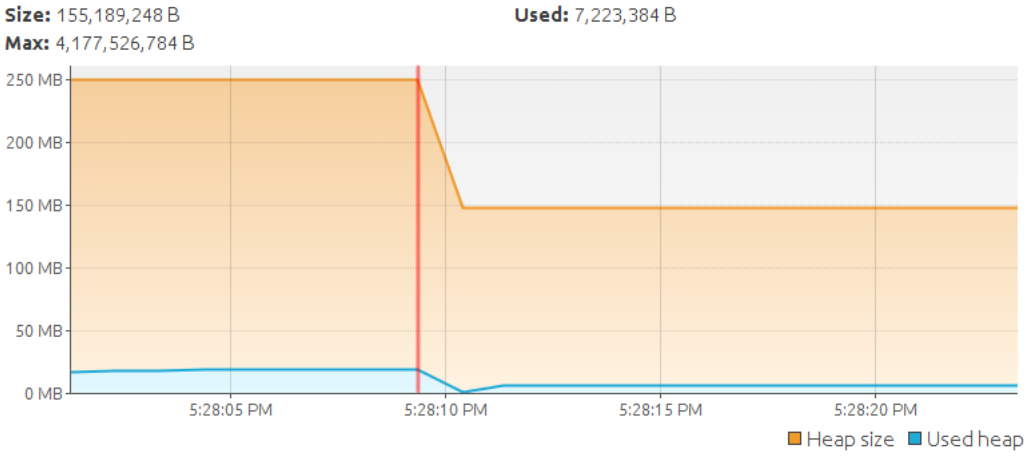
Prostředků k vytvoření dumpu je několik. Při správném nastavení (pomocí parametru `HeapDumpOnOutOfMemoryError`) k němu dojde při nedostatku paměti zcela automaticky. Mezi manuální způsoby vytvoření patří primárně nástroj *JMAP*, který je publikován spolu se standardní distribucí Oracle JVM. Při použití tohoto nástroje je nutné naprosto přesně dodržet číslo verze JMAP a JRE, pod kterým cílová aplikace běží – dumpování rozdílných verzí není podporováno, je nutné dodržet rovnost verzí (major, minor i update). Vzhledem k tomu, že JMAP je součástí JRE, je vhodné použít tuto přibalenou verzi, čímž máme kompatibilitu zajištěnu.

Mezi další způsoby vytvoření dumpu patří různé nástroje, debuggery a profilery typu Eclipse MAT, VisualVM nebo Java Mission Control (viz dále). Výhodou těchto nástrojů je, že dokáží zvládnout vytváření dumpu i napříč verzemi a dokonce implementacemi (z Oracle JDK na OpenJDK apod.)

Další možnost je využít některou z knihovních funkcí a vytvářet tak dump programově. Zde je možné využít např. MBeans. Nízkoúrovňovou možností je poté například použití Unixového nástroje *gcore*, respektive *GDB*, který se postará o vytvoření dumpu paměti procesu (pod daným Process Identifier – Identifikátor procesu (PID)), tzv. *core dump*. Z něj lze memory dump vyextrahovat. Pokročilejší nástroje typu VisualVM umí pracovat i napřímo s core dumpem.

Vztah dumpu vůči paměti procesu

Heap Dump přímo odpovídá části paměťového prostoru procesu, resp. heapu. Je tedy přímým otiskem části fyzické paměti tak, jak je uložena, v určitém časovém okamžiku. Toto je možné si experimentálně ověřit – jak bylo zmíněno výše, z otisku fyzické paměti je možné heap dump získat. Je tedy evidentní, že při jeho vytvoření některým z výše uvedených způsobů nedochází k žádným úpravám a snímek je vytvořen „tak jak je“.



Obr. 2: Využití paměti a provedení garbage kolekce před vytvořením dumpu (červeně).

3.7.3 Zpracování dumpu

Při zpracování dumpu je nutné zohlednit fakt, že je vyexportovaný kompletní paměťový prostor (což výchází z výše uvedeného faktu, že i z *core dumpu* lze memory dump získat) a nachází se zde tedy i data objektů, které nás nutně nemusí zajímat – typicky knihovny nebo objekty Javy. Toto je možné zohlednit a filtrovat na základě jmenného prostoru (namespace), do kterého objekt patří.

Pro práci nad heapem (typicky ve formátu HPROF) je možné využít některou z implementací dotazovacího jazyku OQL (Object Query Language).

3.7.4 Binární formát dumpu heapu

Binární formát dumpu Java heapu (*hprof*) má specifikovaný formát, který je popsán v tabulkách 1 a 2.

Hlavička

Velikost	Popis
$n + 1$ B	Formát a verze dumpu délky n. Velikost není specifikována, pole bytů je ukončeno hodnotou `null`.
4 B	Velikost identifikátorů. Mohou mít stejnou velikost jako ukazatele hostitelského systému, ale není to nutné.
4 B	Část data (high-order word) z Unix timestampu.
4 B	Část data (low-order word) z Unix timestampu.

Tab. 1: Hlavička `hprof` binárního formátu.

Tělo

Tělo formátu dumpu je složeno ze sekvence (či pole) položek následujícího formátu.

Velikost	Popis
1 B	Tag – typ záznamu – viz tabulka 3.
4 B	Čas – počet milisekund, které uplynule od Unix timestampu v hlavičce.
4 B	Délka – počet bytů n, které následují a slouží jako obsah záznamu.
n B	Obsah záznamu.

Tab. 2: Tělo `hprof` binárního formátu.

Typy záznamů

Tabulka s tělem obsahuje pole záznamů, v nichž je typ záznamu – pole *tag*. Celkově je podporováno 13 následujících typů – z toho většina má ještě, jako obsah, další – více či méně komplexní – strukturu.

Typy dumpovaných hodnot

Některé z výše zmíněných typů (které jsem dále nerozepisoval) už obsahují přímo data programu (ať už uživatelská nebo pomocná). Proto jsou definovány typy těchto dat, které

Tag	Popis
0x01	Dump řetězce.
0x02	Načtení třídy.
0x03	Odstranění (ve smyslu opaku načtění) třídy.
0x04	Rámec (položka) stacku.
0x05	Stack trace.
0x06	Alokace.
0x07	Souhrn některých hodnot heapu.
0x0A	Počátek vlákna.
0x0B	Konec vlákna
0x0C/0x1C	Heap dump, respektive heap dump segment.
0x2C	Konec heap dumpu.
0x0D	CPU vzorkování.
0x0E	Hodnoty některých nastavení.

Tab. 3: Typy položek `hprof` binárního formátu.

jsou shrnuty v tabulce 4. Pole jsou definována pomocí řetězcu s jejich názvem tak, jak by se objevily v Javě, tedy např. `int[][]`.

Označení	Datový typ
2	`Object`.
4	`Boolean`.
5	`Char`.
6	`Float`.
7	`Double`.
8	`Byte`.
9	`Short`.
10	`Int`.
11	`Long`.

Tab. 4: Datové typy položek `hprof` binárního formátu.

4 PRÁCE S PAMĚTÍ

4.1 Optimalizace užití paměti

Jak již bylo zmíněno v úvodu, využití paměti lze optimalizovat na několika úrovních. Vzhledem k tématu této práce se při uvažování optimalizací omezíme pouze na běžící Java aplkaci. Eliminaci případných zbytečných knihoven, frameworků a nástrojů se věnovat rovněž nemusíme. Analyzovat procentuální využití nabízených funkcí knihovny by mohlo být zajímavé, nicméně tento typ zbytečného užívání paměti nelze přímo označit za memory waste.

Můžeme tedy omezit samotné využití objektů – zvážit, zda je potřebujeme, případně zda neexistuje vhodnější způsob či struktura pro jejich uchovávání. Jejich alokaci v paměti bychom rovněž meli omezit na nejkratší možnou dobu, po kterou si budeme jisti jejich využitím. Nicméně i zde je vhodné najít vhodný poměr mezi optimalizací a zbytečným úklidem objektu, který za několik okamžiků znovu budeme vytvářet. Je možné využít recyklaci objektů – místo vytvoření několika objektů vytvoříme pouze jeden, který budeme na několika místech využívat. Typickým příkladem je vykreslování grafických objektů. Pokud budou objekty sdílet určité rysy a lišit se např. jen pozicí, velikostí a barvou, můžeme využít jen jeden objekt a za pomoci přebarvení a transformací jej postupně vykreslit na několika místech, místo toho, abychom pro každou pozici vytvářeli nový.

Nejpřímějším způsobem optimalizace užití paměti se může jevit odstranění objektů, které už nejsou zapotřebí a nikdo je tedy nevlastní. Takové objekty nicméně nemusíme v naší optimalizaci uvažovat. O jejich uvolnění z paměti se postará GC. Této vlastnosti jazyku tedy můžeme využít a nepotřebné objekty ručně odstraňovat – nastavit je jako `null`. K jejich uvolnění dojde i při opuštění aktuálního prostoru – oboru platnosti lokální proměnné. K takovému uvolnění nicméně nedochází přímo zapomocí analýzy GC, ale díky uložení lokálních proměnných v stacku, z kterého jsou při opuštění bloku odstraněny.

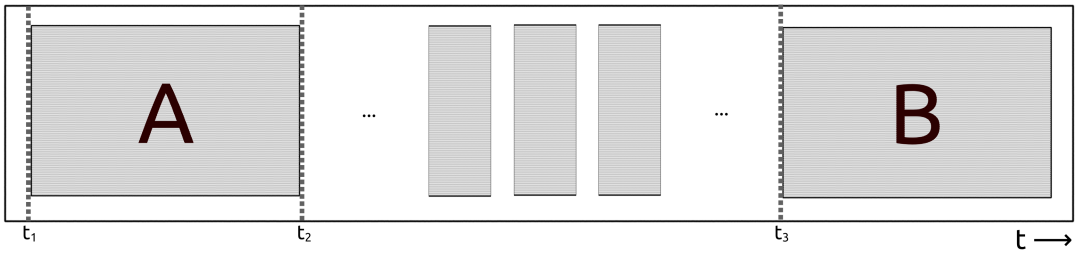
TODO rezepsat dal?

4.2 Analýza za běhu programu

Během běhu programu lze analyzovat aktuální obsah paměti – GC koneckonců nedělá nic jiného. Problémem tohoto přístupu je ovšem dopad na výkon. I samotný GC, ať už umožňující současný běh kódu nebo *stop-the-world*, má výrazný dopad na výkon oproti jazykům či běhovým prostředím bez něj [17]. Navíc, běh GC, stejně tak jako režie běhu samotného prostředí, jsou v drtivé většině případů v maximální možné míře optimalizovány, aby byl dopad na výkon samotného programu co nejmenší a jeho běh co nejrychlejší. Mohou využívat celou řadu nízkoúrovňových optimalizací, kterých – v některých jazycích, včetně

Javy – dosáhnout prakticky nemůžeme. Z toho můžeme usoudit, že dopad analýzi paměti programu za jeho běhu by byl přinejmenším takový, jako dopad běhu GC; pravděpodobně však výrazně vyšší. Stejně tak musíme uvažovat, že běh naší analýzy – ať už pouze za účelem sběru dat pro pozdější zpracování – bude výrazně složitější (či výkonově náročnější), než běh GC.

Dalším důvodem pro vyhnutí se analýze za běhu programu je povaha řešeného problému a tedy implementovaného algoritmu. Protože potřebujeme provádět hloubkovou analýzu (všech či vybraných) objektů programu, jejich neustálá změna by tuto analýzu za běhu programu učinila nemožnou – z tohoto důvodu vyžadují různé implementace GC *stop-the-world*. Respektive bychom se nemohli vyhnout eventuálním falešně negativním či positivním hlášením – to si jednoduše můžeme představit. Jestliže s každým cyklem analýzy zpracujeme jeden objekt a během toho dojde ke změně dat některého z dalších objektů, pak by algoritmus zahlásil při zpracování dalších objektů falečně pozitivní nález. V žádném časovém okamžiku neexistovaly dva objekty se stejnými daty, algoritmus by je tak přesto označil.



Obr. 3: Problém analýzy paměti za běhu programu.

4.3 Analýza memory dumpu

TODO

5 JAVA VIRTUAL MACHINE

Program napsaný v Javě běží typicky v některé z implementací JVM. JVM je program, který slouží jako běhové prostředí pro uživatelský kód – vykonává jeho instrukce a slouží tak jako prostředník mezi ním a operačním systémem (respektive jako interpret jeho kódu, který následně překládá do jiného jazyka, typicky strojového kódu dané architektury či platformy). JVM je možné si představit jako virtuální počítač – má vlastní instrukční sadu a na základě ní provádí operace nad pamětí. Díky tomu je možné jej, v případě potřeby, implementovat i jako CPU – taková hardwarová implementace se nazývá *Java procesor*.

JVM nepracuje s žádným konkrétním vyšším jazykem (zdůrazňuji zde „vyšším“, protože bytecode jazykem je, byť nízkoúrovňovým). Vidí pouze výsledek kompilace do bytecodu (viz dále), což je posloupnost operací z výše zmíněné instrukční sady. Jako analogii je možné zmínit instrukční sadu CPU a strojový kód – na úrovni CPU již také nezjistíme, jaký vyšší programovací jazyk je původcem strojového kódu (tedy za předpokladu podobné implementace kompilátoru porovnávaných jazyků). Stejně tak se chová JVM a bytecode – ostatně existují i další jazyky na platformě JVM, za všechny můžu jmenovat třeba populární Kotlin [22]. Přeložený bytecode těchto jazyků by tedy měl být přinejmenším srovnatelný a neměl by se zásadně lišit.

Některé implementace JVM umožňují přímý překlad do strojového kódu bez potřeby interpretace, např. GraalVM [3]. V takovém případě hovoříme o tzv. Ahead-Of-Time (AOT) přístupu, místo Just-In-Time (JIT) postupu implementovaného v moderních verzích častěji používaných JVM, např. HotSpot od společnosti Oracle – byť je zde silně optimalizovaný a o čistý JIT přístup se nejedná [24].

JVM se stará o načtení kódu ze souboru `.class`, dále o jeho verifikaci, spuštění a zároveň poskytuje tomuto kódu prostředí, v rámci kterého může běžet.

5.1 Java Bytecode

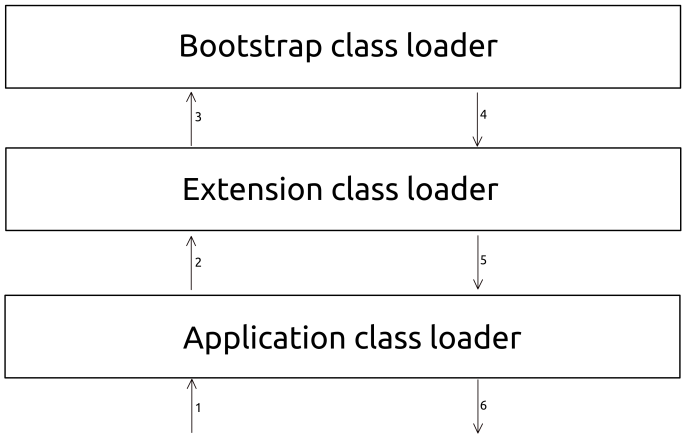
Java využívá dvoufázový překlad, tj. samotný zdrojový kód vytvořený programátorem je nejprve přeložen do *bytecodu* (či česky bytekód). Bytecode je (či by měl být) platformě nezávislý soubor instrukcí, který následně JVM vykoná v prostředí architektury a platformy, na které je spuštěn. To znamená, že zdrojový kód v jazyce Java (či kompatibilních jazycíc využívajících stejné prostředí, např. Kotlin), uložený typicky v souboru `.java`, je přeložen do bytecodu – typicky s typu `.class`. Takové soubory je následně, obecně vzato, možné přenést na jinou platformu či architekturu a jestliže se zde nachází kompatibilní JVM, je možné jej bez úpravy na daném systému vykonat a tedy program spustit. Bytecode se může přenášet formou klasických jednotlivých souborů `.class` či v zabalené formě, tj. archiv typu `.jar` (což je de facto pouze zip archiv s předem danou strukturou a volitelně přidanými informacemi – manifestem, metadaty, podpůrnými soubory (*resources*)).

5.2 Class loader

Jak už jsem zmínil, JVM se stará o načítání dat ze souboru `.class`. Konkrétně se o toto načítání stará objekt nazvaný *class loader*. Ten, typicky, načítá třídy ze souborového systému, konkrétně z cesty (nebo z více cest) definované v proměnné `CLASSPATH`. Nicméně, díky tomu, že koncept class loaderů je poměrně abstraktní, je možné načítat třídy například přes síť, z paměti nebo je dynamicky vytvářet za běhu programu dle potřeby. Každá načtená třída (konkrétně objek typu `Class`) obsahuje referenci na objekt class loaderu, který ji zavedl do programu – konkrétně jej lze obdržet pomocí metody `getClassLoader()` (v případě primitivních typů vrací `null`). `ClassLoader` je definovaný jako abstraktní třída, lze tedy definovat vlastní loadery dle potřeby.

Pokud `JVM` narazí na třídu, podívá se do setu načtených tříd. Jestliže se tam nachází, vezme ji (instanci třídy `Class`) a použije. Pokud ne, požádá o její načtení. Systém class loaderů funguje na principu delegování v následujícím pořadí (typicky, pokud jej neupravíme), jak zobrazuje obrázek 4 [25] [26]. V případě, že třídu žádný z boot loaderů nenalezne, je produkována výjimka `ClassNotFoundException`.

1. Aplikace požádá *Application class loader* o načtení třídy.
2. Ten zavolá *Extension class loader*.
3. *Extension class loader* zavolá *Bootstrap class loader*.
4. *Bootstrap class loader* funguje v rámci výchozích *JDK* a *JRE* knihoven. Pokud třídu nalezne, tak ji vrátí. V opačném případě volá zpět *Extension class loader*.
5. *Extension class loader* se podívá v rámci rozšíření *ext* Javy. V případě neúspechu volá *Application class loader*.
6. *Application class loader* načítá třídy z `CLASSPATH` (a třídy specifikované v Manifestu atd.)



Obr. 4: Postup volání hierarchie class loaderů.

6 EXISTUJÍCÍ NÁSTROJE PRO ANALÝZU HEAPU

Analýza dumpu – ať už manuální či automatická – je využívána v případech, kdy nám debugování na úrovni kódu už nestačí a potřebujeme se podívat, jakým způsobem se program chová na pozadí. Některé typy problémů navíc pomocí debuggeru odhalíme jen těžko (nebo vůbec) a musíme tak zvolit nízkoúrovnější způsob, který dokáže odhalit problém napříč více částmi aplikace. Rovněž je taková analýza vhodná v okamžiku, kdy dojde k pádu (zejména v důsledku nedostatku paměti), kdy chceme zjistit příčinu pádu „post mortem“. Můžeme tak odhalit problematické místo v programu a podrobná analýza nám tak umožní problém opravit.

6.1 Eclipse MAT

IBM Eclipse Memory Analyser Tooling je open source nástroj pro analýzu Java paměti. Po spuštění umožňuje otevřít již vygenerovaný dump Java heapu, umí jej ale i vytvořit. V rámci analýzy nabízí 2 konkrétní volby – analýzu memory leaků a memory bloatu – tj. neefektivního využití paměti a zbytečného plýtvání.

MAT se analýzou memory bloatu přibližuje zamýšlenému výsledku této práce, bohužel ale nenabízí kompletní funkcionalitu v této oblasti. Omezuje se pouze na efektivní práci s řetězci (kterou už obsahuje Java, respektive JVM v základu, viz 3.6.1) a dalšími základnímu typy, např. Map. Cílem práce je ale zpracování všech možných objektů, tato funkcionalita by se tak dala případně rozšířit.

Nástroj je založený na platformě Eclipse RCP (Rich Client Platform), respektive OSGi. Díky tomu je poměrně snadno rozšiřitelný, což je ale vyváženo velkým rozsahem aplikace, který vývoj a rozšíření naopak lehce komplikuje. Buildovacím nástrojem je zde Maven.

6.1.1 Základní informace

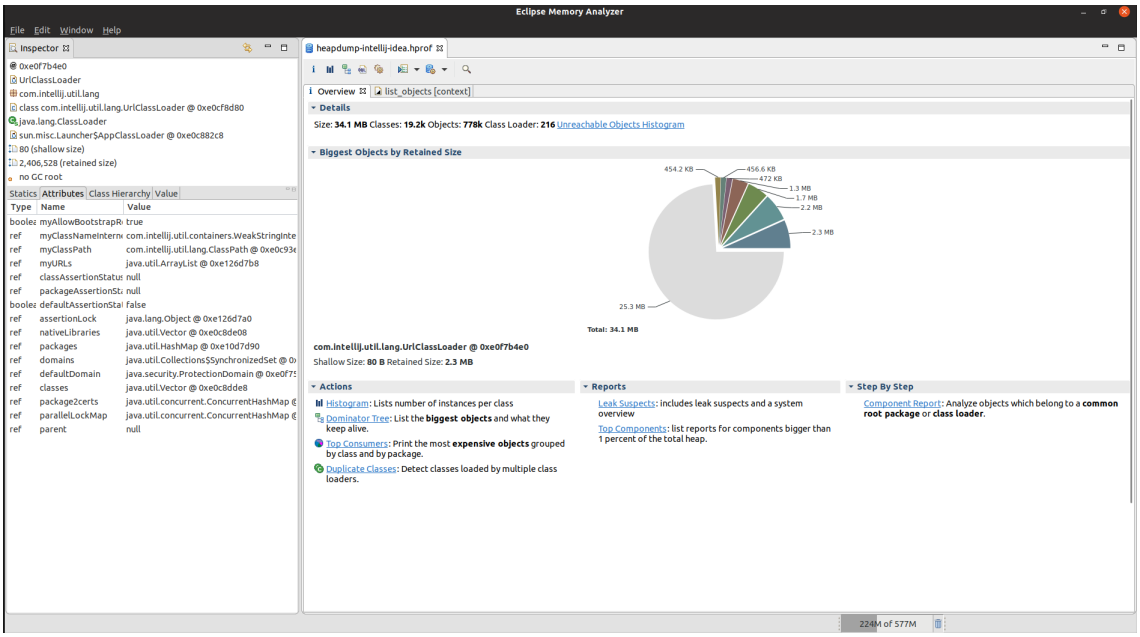
Licence Eclipse Public License 1.0

Jazyk Java

Zdrojový kód `https://git.eclipse.org/c/mat/org.eclipse.mat.git`

Výhody Již obsahuje základní funkcionalitu, která by se dala využít pro potřeby splnění zadání této práce.

Nevýhody Jedná se o velmi obsáhlý projekt, jehož úprava není jednoduchá a pro potřeby této práce je využitelný malý zlomek jeho funkcionality.



Obr. 5: Rozhraní Eclipse MAT

6.2 VisualVM

Open source profiler pro Java platformu. Patří mezi nejpoužívanější profilery, respektive nástroje pro analýzu výkonu v Javě.

Po spuštění nabízí klasické funkce typické pro profilery, tj. využití paměti, zatížení CPU, počet objektů a vláken a podobné statistiky. Kromě toho obsahuje celou řadu dalších funkcí, jako provedení garbage kolekce (její vyžádání, explicitně vyvolání GC není možné) nebo vytvoření heap dumpu.

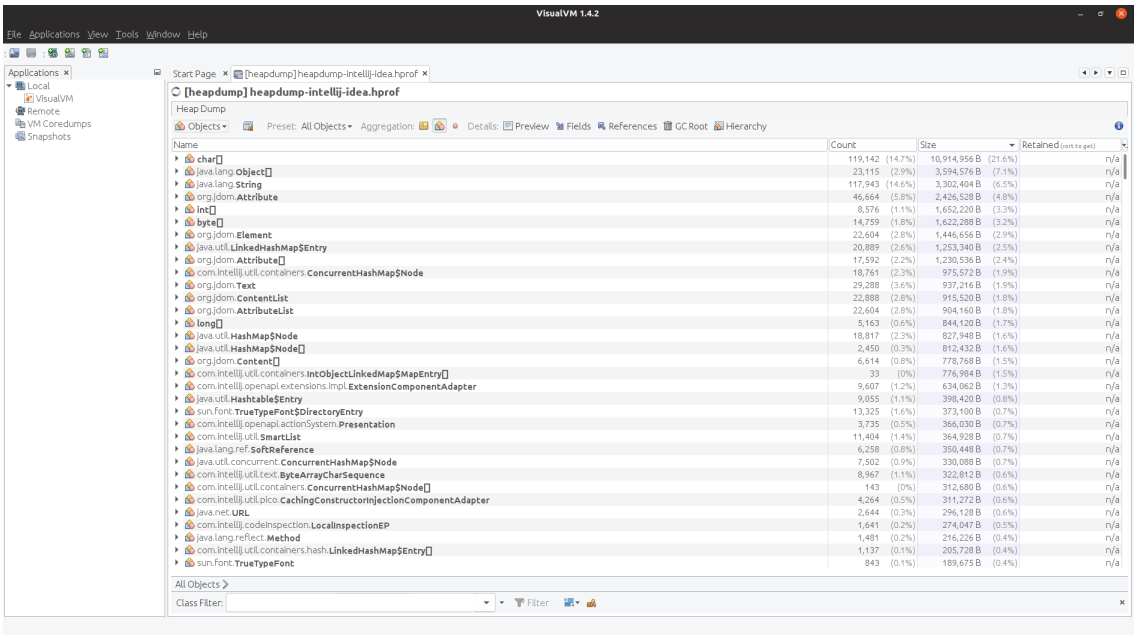
Požadovanou funkcionalitu VisualVM v zásadě neposkytuje, umožňuje pouze k nahlédnutí tabulku s informacemi – kolik bylo vytvořeno instancí jaké třídy, respektive jimi okupovanou paměť. V programu existuje podpora pro OQL syntaxi, což je možné využít, nicméně tento přístup nelze považovat za dostačující.

Pro build je využíván nástroj Ant a v současné době je vyžadována Java verze 7 a vyšší.

6.2.1 Základní informace

Licence GPLv2 + CE
Jazyk Java
Zdrojový kód `https://github.com/oracle/visualvm`
Výhody Populární nástroj, jehož vývoj je (v době psaní této práce) stále velmi aktivní.

Nevýhody Stejně jako v případě Eclipse MAT velmi rozsáhlý a komplikovaný projekt. Kromě předzpracování dumpu neposkytuje žádnou požadovanou funkcionalitu.



Obr. 6: Rozhraní VisualVM

6.3 Java Mission Control

Nástroj poskytovaný přímo spolu s distribucí Oracle JVM od verze 7 (konkrétně 7 Update 40 – 7u40), což je jeho výhodou. Mezi jeho možnosti patří např. využití paměti jednotlivými součástmi Java paměti a také umožňuje zobrazit jednotlivé instance objektů, nicméně neumožňuje jejich další analýzu.

6.3.1 Základní informace

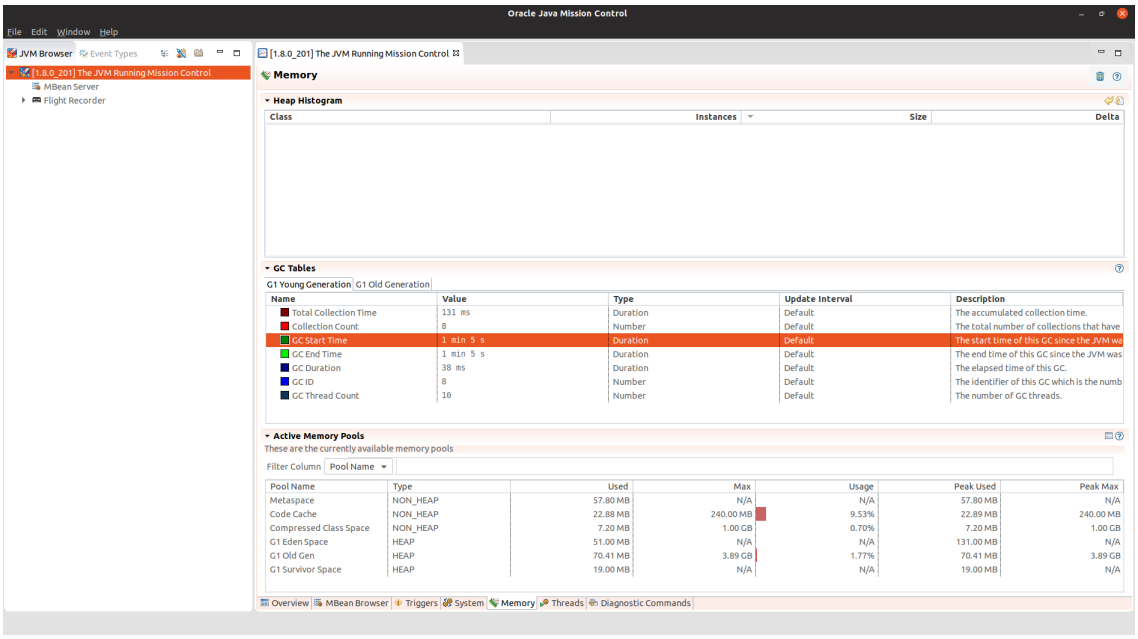
Licence Universal Permissive License (UPL), Version 1.0 nebo BSD

Jazyk Java

Zdrojový kód `https://hg.openjdk.java.net/jmc`

Výhody Nástroj je součástí JDK, jeho rozšíření (formou pluginu) by mohlo výrazně přispět k použitelnosti vytvořené aplikace.

Nevýhody Neumí zpracovávat HPROF soubory.



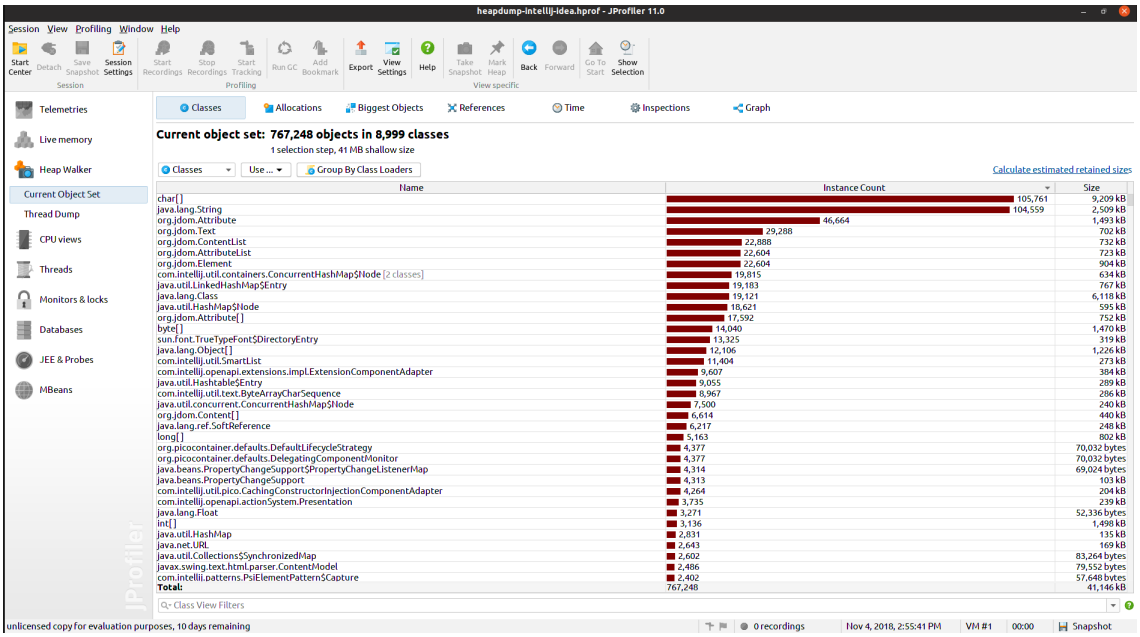
Obr. 7: Rozhraní Java Mission Control

6.4 JProfiler

Komerční profiler, přesto velice používaný. Standardní licence stojí v době psaní 409 euro, akademická potom 179 euro. Je možné zažádat o licenci pro open source produkty, nicméně ta je podmíněna již vydanou verzí a existující webovou stránkou, což činí jakékoliv použití tohoto profileru v rámci práce nepraktickým. Profiler je používán především v komerční sféře, díky svým možnostem a dle výše uvedeného testu i nejvyšší úspěšností v odhalování bugů.

6.4.1 Základní informace

Licence Proprietární – placení software
Jazyk Java
Zdrojový kód Není opensource
Výhody Komeční nástroj, velké možnosti, obsahuje porovnání základních objektů.
Nevýhody Není zdarma.



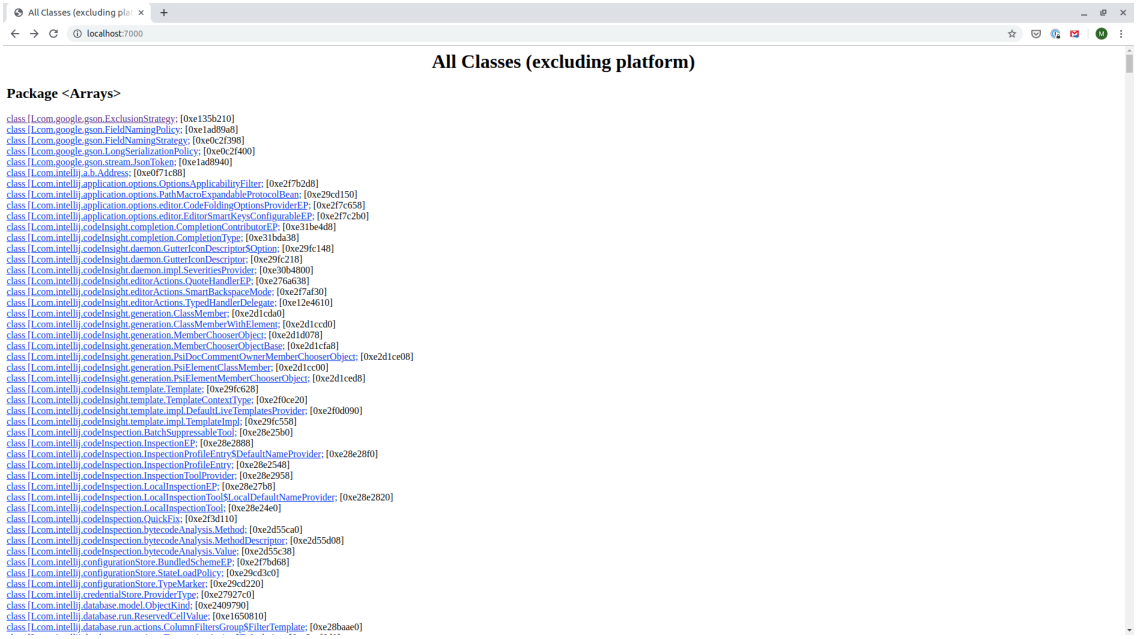
Obr. 8: Rozhraní JProfileru

6.5 JHAT

Nástroj, který je přímou součásti distribuce Oracle JVM od verze 6. Nebyl nikdy oficiálně podporován a od počátku byl označen jako experimentální nástroj, z těchto důvodů byl tedy v Javě 11 naprosto odstraněn [11][13]. V rámci verzí Javy, které jej obsahují, ho lze využít jako Command Line Interface (CLI) aplikaci, která dokáže dump vytvoření pomocí např. *JMAP* otevřít. Následně vytvoří webový server, jehož prostřednictvím poskytuje data získaná ze zpracovaného dumpu. Tato data je možné si poté zobrazit pomocí webového prohlížeče; zajímavým příkladem možného využití je následné rozparsování těchto dat jakožto formátu HTML a jejich další využití. Program je tedy možné využít jako prostředníka pro zpracování [14]. Kromě „prostého“ zobrazení webové stránky umí *JHAT* rovněž poskytovat zpracování pomocí jazyka Object Query Language (OQL).

6.5.1 Základní informace

Licence SUN PUBLIC LICENSE Version 1.0
Jazyk Java
Zdrojový kód `https://download.java.net/openjdk/jdk8/`
Výhody Byl součástí starších verzí JDK, nyní už není.
Nevýhody Již není podporován. Nabízí jen základní funkcionalitu.



Obr. 9: Rozhraní JHAT

6.6 OQL

OQL zmíním i jako samostatný způsob zpracování. Jedná se o jazyk, který slouží pro obecnou manipulaci s objekty, resp. s objektovými dokumenty [15][16]. Na první pohled si nelze nevšimnou jeho podobnosti s dotazovým jazykem SQL. Neomezuje se tedy pouze na zpracování Java heapu (či obecně paměti), ale je standardem, který pro tento účel lze využít. Z toho je možné usoudit, že standard jako takový pro zpracování nestačí – je nutné využít některou z jeho implementací, která takové rozhraní přístupu k Java heapu umožňuje. Jednou z nich je právě výše zmíněný *JHAT* nebo třeba *VisualVM*.

V příkladu 6.1 je možné vidět práci s OQL.

Ukázka 6.1: Příklad OQL

```
select s
from java.lang.String s
where s.value.length >= 100
```

6.7 Knihovny

Knihovny představují způsob přístupu k problému, který by nevyžadoval zásah do již existujícího kódu, ale spíše jeho využití – a to tak, že se starají o jednu specifickou část, kterou tak není nutné implementovat znovu. I výše zmíněné aplikace lze považovat z pohledu API za knihovny, nicméně již vyžadují dodržení jejich struktury a zorientování se v, často velmi rozsáhlém, existujícím kódu.

TODO hprof knihovna

7 MOŽNOSTI ANALÝZY

7.1 Rovnost objektů

Rovnost dvou či více objektů se dá definovat a zjišťovat různými způsoby. Je ale nutné si uvědomit, že v nejhorším případě, tj. pokud chceme najít všechny nadbytečné kopie každého objektu, je složitost této operace $O(2^n)$. Bylo by tedy vhodné se zamyslet, zda neexistuje způsob, jak počet porovnání snížit, případně navrhnout jednoduchou heuristiku, která by dokázala rychle ohodnotit, zda má vůbec smysl pokračovat v podrobnějším porovnání. V následujících případech tedy uvažujme objekty A, B, jejich třídy C_A a C_B a proměnné obou instancí $F_A^0..F_A^n$, respektive $F_B^0..F_B^n$. Hodnotu dané instanční proměnné definujeme jako $V(F)$. Pro potřeby algoritmu předpokládejme, že jsou tyto proměnné v určitém, předem očekávatelném pořadí, např. podle abecedy dle jejich názvu (nebo dle Unicode kódu pro speciální znaky).

První nápovědou samozřejmě může být porovnání tříd obou objektů – C_A a C_B. Pokud platí $C_A = C_B$, zřejmě má smysl se porovnáváním dále zabývat. V případě jejich nerovnosti není ale možné další porovnání zavrhnout, porovnat je nutné (či možné) i jejich rodiče. V případě dědičnosti dvou tříd nelze automaticky vyloučit – pokud potomek k rodičovské třídě nepřidává žádná data, pouze funkcionalitu, pak se z pohledu dat v paměti objekty rovnají, i když jsou sémanticky odlišné.

Definujeme-li tedy funkci pro zjištění přímého rodiče $P(C)$, potom rovnost tříd (z pohledu jejich dat) lze vyjádřit jako

$$E_C(C_A, C_B) \Leftrightarrow C_A = C_B \vee E_C(P(C_A), C_B) \vee E_C(C_A, P(C_B)). \tag{7.1}$$

Samozřejmě je nutné definovat i zastavovací podmínku, v případě Javy by tedy jeden z parametrů nesměl být třída typu *Object*. Formálně je tedy možné tuto rovnost definovat i jako množinu

$$E_C^*(C_A, C_B) = P_c(C_A) \cap P_c(C_B) \notin \emptyset, \tag{7.2}$$

kde $P_c(C)$ je množina třídy samotné a všech jejich rodičů bez "univerzálního předka" všech tříd, v tomto případě *Object*:

$$P_c(C) = \{C, P(C), P(P(C)), \ldots, Object\} \setminus Object. \tag{7.3}$$

Tím máme nastavnu podmínku nutnou – bez rovnosti (tak, jak je výše definována) tříd nemá smysl porovnávat objekty. Nyní se můžeme zaměřit na porovnávání objektů a jejich proměnných samotných. Vzhledem k tomu, že máme zaručenu rovnost tříd, můžeme

očekávat zhruba stejnou strukturu, názvy a typy instančních proměnných. Nicméně nemůžeme očekávat přímou rovnost těchto parametrů, díky tomu, jak jsem výše zadefinoval rovnost proměnných. Proto platí:

$$F_A \subset F_B \vee F_A \subset .F_B \tag{7.4}$$

Naším cílem je ovšem porovnání rovnosti. Musí tedy platit:

$$\|F_A\| = \|F_B\|. \tag{7.5}$$

Tato operace je pouhé porovnání dvou celočíselných hodnot (plus pravděpodobně zjištění velikosti pole), což je zanedbatelná operace, která algoritmus nezpomalí. Tímto je zajištěno, že porovnávané objekty patří pod stejnou třídu či jejího potomka, který nepřidává žádné instanční proměnné, tj. je z pohledu paměti zaměnitelný. To samozřejmě neznamená, že je stejný z pohledu sémantiky či funkcionality; to je však cena za to, že hledáme maximální možnou úsporu. Toto je ovšem funkcionalita, která je jednoduše regulovatelná, tj. v algoritmu je možnost přidat parametr, který bude ovlivňovat způsob porovnávání rovnosti tříd a případně zajistí, že takto budou označeny pouze třídy skutečně stejné, bez ohledu na dědičnost.

Nyní jsou tedy objekty stejných (dle výše definovaných měřítek) tříd a mají stejný počet proměnných. Vzhledem k tomu, že proměnnou nelze při dědičnosti odebrat ani pozměnit, můžeme předpokládat její existenci. Maximálně je možné ji pouze překrýt, případně změnit pomocí reflexe, což jsou okrajové případy, jejich řešení by algoritmus pouze zbytečně zpomalovalo.

Nyní je tedy možné definovat rovnost objektů následovně:

$$E(A,B) \Leftrightarrow E_C(C_A, C_B) \wedge \forall F_A^i, F_B^j \epsilon F_A, F_B : i = j : F_A^i = F_B^j. \tag{7.6}$$

Jinak řečeno – objekty A, B jsou si rovny právě tehdy, když jsou si rovny jejich třídy pomocí výše navrhnuté metody a zároveň platí, že jsou si rovny všechny dvojice proměnných daných objektů.

Teoreticky by samozřejmě bylo možné vyhledávat i napříč různými třídami, pouze na základě rovnosti všech hodnot proměnných. To by ovšem nedávalo velký smysl. Objekty, které algoritmus má vyhledat, jsou kandidáty pro odstranění z programu a nahrazení zbylou, jedinou instancí. To by v případě nerovnosti tříd nebylo možné. Naproti tomu, v případě nerovnosti tříd, kdy je ovšem jedna z nich součástí cesty dědičnosti třídy druhé, to možné je – samozřejmě ale ne vždy. Minimálně to ale dává prostor se zamyslet, zda je skutečně nutné používat dědičnost, když jsou data obou objektů naprosto totožná. Pokud chceme dodržovat sémanticky správný pohled na objekty, nemusíme je kopírovat; stačí uchovávat pouze jedinou instanci a na správných místech použít přetypování (což je de facto polymorfismus).

7.2 Rovnost složitějších objektů

Výše navrhnutý indikátor rovnosti ovšem postačuje pouze pro objekty tříd, jejichž proměnné jsou všechny primitivního typu. To ovšem neplatí v běžném programu netriviálního rozsahu prakticky nikdy. Už například prosté použití řetězce, tedy třídy `String`, tuto podmínku nesplňuje.

Jak bylo popsáno v kapitole 3, referenční – tedy neprimitivní – typy jsou odkazovány pomocí jejich adresy. Výše definovaný algoritsmus by tak porovnával ji – pouze celočíselnou hodnotu. Takový algoritmus ovšem ve skutečnosti není o nic lepší, než prosté porovnání bytů daného objektu přímo v dumpu. Je tedy nutné zvolit vhodnější přístup.

7.2.1 Porovnávání referencovaných objektů

V případě, že je typ jedné z proměnných třídy jinou třídou (a proměnná tedy obsahuje referenci na instanci této třídy nebo `null`), měli bychom porovnat i rovnost tohoto odkazovaného objektu s objektem odkazovaným z druhé instance. Zobecníme tedy výše definovanou rovnost na funkci $E_F(f_1, f_2)$:

$$E(A, B) \Leftrightarrow E_C(C_A, C_B) \wedge \forall F_A^i, F_B^j \epsilon F_A, F_B : i = j : E_F(F_A^i, F_B^j) \tag{7.7}$$

a tuto funkci definujeme jako:

$$E_F(f_1, f_2) = \begin{cases} f_1 = f_2 & \text{pokud je typ proměnné primitivní} \\ E(f_1, f_2) & \text{pokud je typ proměnné referenční} \end{cases}. \tag{7.8}$$

Z těchto vztahů ale vyplývají dva problémy:

- Není zajištěna ochrana proti zacyklení v rámci rekurze.
- Pro rekurzi není nastavena zastavovací podmínka. Nabízí se samozřejmě otázka, zda takový problém v praxi vůbec potřebujeme řešit – složitější struktury se budou v hloubce pravděpodobně často lišit. Častěji se tedy budeme spíše setkávat s ukončením rekurze díky této podmínce, pomocí ochrany proti zacyklení nebo prostým prohledáním celého stromu. Definice zastavovací podmínky je tedy spíše pouze formální.

První problém můžeme vyřešit jednoduše v rámci původní rovnice, její menší úpravou:

$$E(A, B) \Leftrightarrow E_C(C_A, C_B) \wedge \forall F_A^i, F_B^j \epsilon F_A, F_B : i = j \wedge D(A, B, F_A^i, F_B^j) : E_F(F_A^i, F_B^j) \tag{7.9}$$

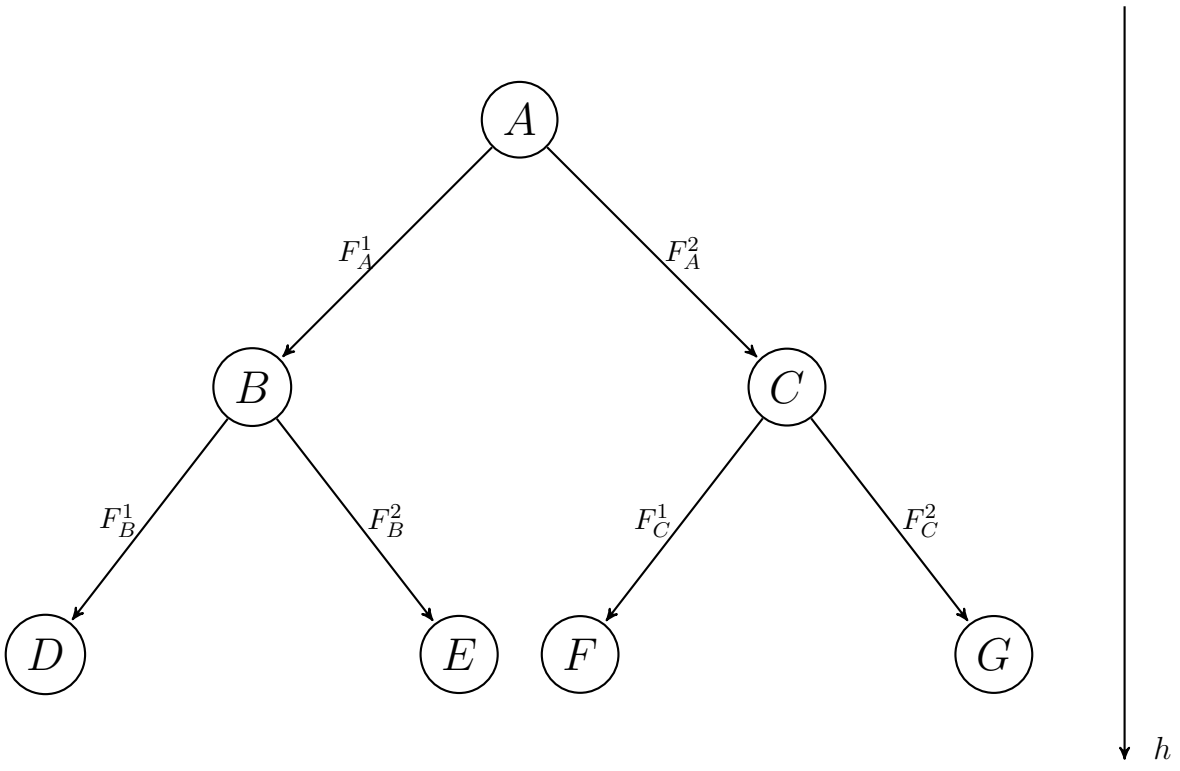
A pomocnou funkci $D(A, B, f_1, f_2)$, která zjišťuje prostou odlišnost objektů (na úrovni reference, adresy) a slouží tedy jako ochrana proti zacyklení:

$$D(A, B, f_1, f_2) = f_1 \neq A \vee f_1 \neq B \vee f_2 \neq A \vee f_2 \neq B \tag{7.10}$$

Problém druhý, tedy zastavovací podmínku, můžeme vyřešit pouhým počítáním hloubky (zanoření) porovnávání a přerušením tohoto zanořování v určité hloubce – ideálně nastavitelné externě tak, abychom mohli volitelně ovlivňovat rychlost, a nevyhnutelně tak i přesnost, algoritmu. Formálně je tento problém tedy možné definovat tak, že porovnávání referenčních typů budeme chápat jako strom, jehož rozvoj ukončíme při dosažení určité hloubky h. Další možností je hloubku zanořování nelimitovat, pouze se omezit na porovnávání těch částí stromu, které nejsou zacyklené a nevedou tedy k opakovanému porovávání. Tento problém musíme řešit tak či tak, proto tento způsob můžeme chápat jako speciální případ prvního postupu s vysokým h, formálně tedy $h = \infty$.

Tento strom je vidět na obrázku 10. Mějme dva objekty, A_1 a A_2. Oba mají po 2 proměnných, $F^1_{A_1}, F^2_{A_1}$, respektive $F^1_{A_2}, F^2_{A_2}$. Označme operaci tohoto porovnání jako A a porovnání dvojic třídních proměnných jako F^1_A, F^2_A. Hloubka zanoření je v tuto chvíli $h = 0$. Rekurzivně takto budeme sestupovat, dále tedy provádíme porovnání B a C obdobně, jako porovnání A, tentokrát s hloubkou zanoření $h = 1$.

Takto bychom sestupovali (dále s operacemi v hloubce $h = 2$), dokud bychom nenarazili na horní limit h_{max}. V ten okamžik se můžeme zachovat ke všem proměnným jako k primitivním typům a dále rekurzivně nesestupovat.



Obr. 10: Strom porovnávání objektů

7.2.2 Porovnávání řetězců

Porovnání řetězce jako takového je poměrně jednoduché, kromě vlastního řešení, které může být více či méně efektivní, můžeme byty převést opět do objektu typu `String` a využít metodu `equals`, kterou Java nabízí.

Nicméně, pro zrychlení porovnávání můžeme využít výše zmíněného poznatku, že se Java v některých případech snaží uchovávat řetězce v tabulce symbolů. V takovém případě by stačilo pouze porovnat adresu, na kterou proměnná typu `String` ukazuje. To je prosté porovnání celočíselných hodnot, tedy triviální operace.

7.2.3 Porovnávání polí a kolekcí

Porovnávání polí je opět poměrně jednoduché a znamená prosté iterování nad všemi položkami pole a jejich porovnání podle výše zmíněných metod.

Musíme se pouze zamyslet, jakým způsobem chceme toto iterování provádět – respektive, zda jsme ochotni vyměnit rychlost za přesnost. Prvky pole mohou být v různém pořadí. V takovém případě se problém redukuje na problém řazení, kterey je v nejlepším případě $O(NlogN)$ (např. pro *Quick sort*). Nabízí se ovšem otázka, zda k takovému kroku chceme přistupovat – implementace programu může být na pořadí prvků závislá a pokud bychom označili objekty s takovými poli jako duplikáty, mohlo by se jednat o falešně positivní případ. Rovnost polí $E_A(f_1, f_2)$, kde f_1, f_2 jsou třídní proměnné typu pole, obsahující prvky $p_1, p_2, ..., p_N$ je tedy možné definovat jako:

$$E_A(f_1, f_2) = \forall f_1^i, f_2^j \epsilon f_1, f_2 : i = j : E_F(f_1^i, f_2^j) \tag{7.11}$$

V případě kolekcí takto jednoduše rozhodnout možné není. Musíme si uvědomit, že přestože některé třídy sdílejí určité společné rozhraní (třeba `Collection`, `Map` apod.), jejich implementace uvnitř záleží na typu struktury, kterou zvolili pro reprezentaci ukládaných dat. V případě třídy `ArrayList` je možné takový problém redukovat na problém porovnávání pole; naopak v případě `LinkedList` je nutné několikanásobné porovnání N objektů. To lze samozřejmě také redukovat na problém porovnání pole a dívat se na propojení listu na – de facto – iterátor. Nicméně už taková operace vyžaduje mezikrok navíc.

7.3 Efektivita využití polí a kolekcí

Efektivitu využití pole či kolekce definuji jako poměr obsazených prvků vůči volným (tedy s hodnotou `null`). Na první pohled má tato metrika smysl zjišťovat pouze u objektů tříd, jejichž implementace pole využívá jako strukturu pro ukládání samotných dat (stejně jako je zmíněno výše), nicméně i implementace využívající spojové listy mohou `null` obsahovat.

Efektivita využití R pole A s A_N nenulovými prvky je definována jednoduše jako

$$R(A) = \frac{A_N}{\|A\|}. \tag{7.12}$$

Pole, jehož využití paměti je neefektivní, bude mít hodnotu této metriky nižší než stanovenou hranici. Tuto hranici ude možné nastavovat pomocí parametru a případně její výchozí hodnotu upravit dle výsledků běhu analýzy reálných programů. Jako uměle stanovenou počáteční hodnotu navrhuji vybrat zaplnění poloviny pole (struktury), tedy

$$R(A) < 0.5. \tag{7.13}$$

8 NÁVRH IMPLEMENTACE

V následujících odstavcích a kapitolách se budu věnovat analýze řešení výše nastíněných problémů. Výsledné řešení musí být využitelné v praxi – mělo by tak být robustní, vytvořené za využití stabilní softwarové platformy a výsledek by měl být pokud možno k dispozici co nejrychleji. Pro potřeby automatizace by mělo být k dispozici CLI rozhraní či API, grafické uživatelské rozhraní je tedy vedlejší a jeho vytvoření není nutné.

8.1 Volba platformy

Vzhledem k tomu, že při řešení problému nejsme závislí na Javě samotné, ani v rámci API, nejsou na volbu platformy kladeny žádné speciální požadavky. Je nutné pouze načíst soubor HPROF a nad ním začít provádět analýzu – jediným požadavkem je tak pouze základní IO.

Volbou jiné platformy než JVM společně s Javou nicméně vytváříme další závislost pro budoucí potenciální uživatele. Vzhledem k tomu, že neexistuje žádný důvod pro volbu jiné platformy (snad kromě případných osobních jazykových preferencí), nejvhodnější volba tak tedy připadá opět na Javu. Dá se navíc předpokládat, že bude poskytovat nástroje pro zpracování memory dumpu pro vlastní platformu. Tento požadavek tak omezuje volbu platformy pouze na JVM.

Teoreticky by bylo možné rovněž využít některý z jazyků využívajících platformu JVM, např. již zmíněný Kotlin. Ten by mohl, za dodržení výše stanovených podmínek, přinést další výhody, třeba lepší podporu (v porovnání s Javou) pro tzv. stream API. To by mohlo znamenat výrazné zefektivnění zápisu některých operací nad listy a mapami, které se u zde řešeného problému dají očekávat ve velkém měřítku. S tím souvisí výborná podpora pro *lambda* zápis. Nelze říci, že by tento „syntaktický cukr“[1] zrychloval běh programu jako takového, ale mohl by výrazně urychlit jeho vývoj. Kromě toho je takový zápis intuitivnější (při alespoň základní zkušenosti s funkcionálním přístupem k programování), díky čemuž je možné očekávat nižší výskyt chyb.

S tím souvisí i další vlastnost Kotlinu – odolnost vůči chybám při přístupu k neinicializovaným objektům, v Javě známým jako `NullPointerException` (`NPE`, podobný koncept zná i velká část ostatních jazyků, včetně nízkoúrovňových, jako třeba *Segmentation fault* v `C`), které vznikají nedostatečným či nedůsledným ošetřením vstupů a návratových hodnot. Kotlin vynucuje tzv. *null safety* pomocí různých operátorů. Rovněž je ošetření volání, která by mohla vyvolat `NPE`, vynuceno na úrovni překladače, tj. na potenciální problém se přijde již při fázi překladu (a ne až za běhu – *runtime* – jako je tomu u Javy), který selže

[1] *Syntactic sugar* – vlastnosti jazyků, které nepřidávají novou funkcionalitu, ale umožňují pohodlnější, rychlejší a úspornější zápis pro programátora.

a donutí nás chybu opravit. `NPE` se tímto přístupem z velké většiny podaří eliminovat – i když se při nedůsledném užívání některých operátorů objevit může.

Přes všechny tyto nesporné výhody Kotlinu (a případných dalších jazyků platformy JVM, jako třeba Closure, Groovy, Scala apod.) je ale nutné se zamyslet, zda cena za ně není příliš vysoká. Především z pohledu udržitelnosti – lze předpokládat, že tento projekt bude publikován jako open source. Proto by bylo vhodné vybrat některý z jazyků s majoritním podílem tak, aby byl případně projekt široce dostupný. Právě dostupnost by se, s využitím některého z méně rozšířených či „exotičtějších“ jazyků, drasticky snižovala.

Z výše uvedených důvodů tedy výchází jako platforma pro vývoj JVM a jazyk Java. Případně je při dalším vývoji možné využít Kotlin jako druhý jazyk, který dokáže s Javou spolupracovat i v rámci jednoho programu/projektu na úrovni API.

8.2 Způsob zpracování heapu

Jak bylo zmíněno v předchozích kapitolách, k přístupu ke zpracování heapu lze přistoupit mnoha různými způsoby. V zásadě je možné je redukovat na dva typy přístupu, a to

- analýza za běhu programu, nebo
- analýza dumpu paměti.

V kapitole 4.2 jsem došel k závěru, že analýza za běhu programu by, stejně jako spuštění GC, vyžadovala *stop-the-world*. Což, společně s faktem, že HPROF formát paměťového dumpu je de facto otisk reálné paměti, redukuje oba problémy na analýzu stavu paměti v určitém časovém okamžiku – jedná se tak vlastně o snímek (*snapshot*) v tomto čase.

Analýza za běhu by nicméně vyžadovala vytvoření modulu, který by se přilinkoval k projektu už během kompilace, tj. vyžadovala by cílenou implementaci autorem projektu. Jako druhou možnost by bylo možné využít některý ze systémů dodávaní modulů do již zkompilovaného projektu, např. frameworky OSGi nebo Spring (pomocí autokonfigurace). Takový přístup ale znovu přidává podmínky, které by autor programu musel splnit, aby mohl analýzu heapu využít – konkrétně závislost na konkrétním frameworku, pro který bych přidal podporu. Ideální by tedy byla možnost analýzy bez konkrétních závislostí na čemkoliv kromě JVM. Kromě skutečně nízkurúrovňových přístupů tak vychází analýza dumpu jako lepší z dvojice výše nabízených možností.

Standardní formát jako HPROF nabízí celou řadu existujících nástrojů a knihoven, které byly popsány v rámci kapitoly 6. Žádný z existujících nástrojů dle mého názoru neposkytuje dostatečnou podporu požadovaných funkcí – přinejmenším do té míry, aby se vyplatilo investovat čas do analýzy jejich architektury a úpravy kódu.

Dle mého názoru je tak nejvhodnější implementovat samostatný, nezávislý nástroj. Nicméně není nutné vyvíjet všechny součásti aplikace znovu. Pokud shrneme zadání a

Ukázka 8.1: Příklad zpracování hlavičky ze souboru HPROF

```
String format = readUntilNull(in);
int idSize = in.readInt();
long startTime = in.readLong();
handler.header(format, idSize, startTime);
```

výše uvedená tvrzení, zjistíme, že jednou z činností, kterou můžeme nahradit, je načtení a zpracování souboru HPROF do „dotazovatelné“ podoby. Na základě této reprezantace dumpu v paměti můžeme dále provádět analýzu, což je činnost, na kterou se tato práce zaměřuje. Právě proto bych rád využil některou z knihoven, které by tuto činnost dokázaly efektivně provést.

Po analýze a prozkoumání existujících projektů vyšel jako nejlepší kandidát projekt *Hprof Heap Dump parser*.

8.3 Hprof Heap Dump parser

Jedná se o poměrně malou, v době psaní této práce stále udržovanou knihovnu, starající se o zpracování souboru HPROF.

8.3.1 Základní informace

Licence Apache License, Version 2.0
Jazyk Java
Zdrojový kód `https://github.com/eaftan/hprof-parser`

8.3.2 Základní struktura

Hlavní třídou, která se stará o zpracování HPROF souboru a která nás tedy bude zajímat, je `HprofParser`.

Jak bylo popsáno v kapitole 3.7.4, HPROF má standardizovaný, binární formát. Knihovna toho využívá a zpracovává ho přesně, jak by se dalo očekávat – dle definovaných bytových délek načítá jednotlivé hodnoty. Protože zná (dle tabulek 1 - 4 ve stejné kapitole) význam těchto binárních hodnot i jejich datové typy, může celý soubor kompletně načíst, zpracovat a vystavět tak v paměti reprezentaci memory dumpu, který obsah souboru HPROF představuje.

Například zpracování hlavičky:

Jak je vidět, načtení přesně odpovídá definici formátu hlavičky v tabulce 1 – položka s formátem souboru nemá stanovenou délku, jedná se o řetězec zakončený hodnotou `null`,

tj. \0 (tzv. *null-terminated* nebo také *C-style* řetězce). Velikost identifikátorů je celočíselná hodnota o velikosti 4 B (tedy *Integer*) a časové razítko jsou dvě 4B položky, což je v knihovně najednou načítáno jako `long`, což je typ, který má v Javě délku 8 B. Následně jsou tyto hodnoty publikovány skrze volání tzv. `RecordHandler` (viz dále).

Po načtení hlavičky je v knihovně iterováno nad následujícími byty, kdy je vždy jako první načten `tag` – typ záznamu. Dle této hodnoty je poté, pomocí přepínače `switch`, prováděna příslušná akce, respektive načtení dalších údajů, které se různí dle hodnoty položky `tag`. Na základě načtených hodnot jsou poté vytvořeny entity, které těmto datům přidávají informace (zejména z pohledu sémantiky dat) a které jsou dále publikovány skrze `RecordHandler`, stejně jako v případě hlavičky.

8.3.3 RecordHandler

Jedná se o rozhraní, které slouží jako posluchač (`listener`) volání třídy `HprofParser`, jejíž funkcionalita byla popsána výše. Má několik metod, nicméně pro potřeby této práce jsou zapotřebí zejména následující (společně s popisem vybraných důležitých parametrů):

- `void header(String format, int idSize, long time)` – Výše popsaná metoda pro zpracování hlavičky.
- `void stringInUTF8(long id, String data)` – Zpracování řetězce.
  - `id` – ID řetězce.
  - `data` – Hodnota řetězce samotného.
- `void loadClass(int classDump, long classObjId, int stackTraceSerialNum, long classNameStringId)` Načtení třídy. Této metody můžeme využít pro registrování všech tříd, které se v právě zpracovávaném dumpu nacházejí. Stejně tak existuje metoda `unloadClass`, kterou zde ale potřebovat nebudeme.
  - `classObjId` – Číselný identifikátor třídy.
  - `classNameStringId` – Číslo řetězce, který uchovává název třídy.
- `void classDump(long classObjId, int stackTraceSerialNum, long superClassObjId, long classLoaderObjId, long signersObjId, long protectionDomainObjId, long reserved1, long reserved2, int instanceSize, Constant[] constants, Static[] statics, InstanceField[] instanceFields)` – Podrobné informace o načítaných třídách, které se dají provázat společně s informacemi načtenými prostřednictvím metody `loadClass` pomocí jednotného `classObjId`.
  - `classObjId` – Číselný identifikátor třídy.
  - `superClassObjId` – Číselný identifikátor třídy, kterou daná třída rozšiřuje. Díky tomuto poli jsme schopni zrekonstruovat kompletní strom dědičnosti všech tříd, až po `Object`.

  - `classLoaderObjId` – Číselný identifikátor `ClassLoaderu`, který se postaral o načtený této konkrétní třídy.
  - `constants` – Pole s informacemi o konstantách, které třída definuje.
  - `statics` – Pole s informacemi o statických položkách.
  - `instanceFields` – Pole s informacemi o instančních proměnných. Pravděpodobně nejdůležitější pole pro potřeby splnění zadání této práce.
- `void instanceDump(long objId, int stackTraceSerialNum, long classObjId, Value<?>[] instanceFieldValues)` – Informace o instancích.
  - `objId` – Číselný identifikátor instance (objektu).
  - `classObjId` – Číselný identifikátor třídy, jejíž instancí je daný objekt. Koresponuje s identifikátory definovanými v rámci metod `classDump` a `loadClass`.
  - `InstanceFieldValues` – Pole s hodnotami třídních proměnných, které byly definovány v metodě `classDump` s podrobnými informacemi o třídě. Jak je možné si všimnout, jedná se o generický typ, jehož hodnota je v době kompilace nejasná (`?`), proto lze očekávat, že v době načtení informací o instanci bude nutné přetypovat tuto hodnotu na základě informací o typu třídní proměnné.
- `void objArrayDump(long objId, int stackTraceSerialNum, long elemClassObjId, long[] elems)` – Informace o poli, jehož prvky jsou referenčního typu. Protože se jedná o reference, hodnoty samotné představují pouze odkazy na objekty – proto jsou typu `long`, což odpovídá hodnotě parametru `objId` definovanému v metodě `instanceDump`.
  - `objId` – Číselný identifikátor objektu, tedy tohoto pole.
  - `elemClassObjId` – Číselný identifikátor třídy, jejíž instance představují prvky tohoto pole. Díky tomuto parametru můžeme pole definovat jako `<Class>[]` (kde `<Class> je název třídy`), namísto generického `Array` či `Object[]`.
  - `elems` – Pole s referencemi na jednotlivé objekty.
- `void primArrayDump(long objId, int stackTraceSerialNum, byte elemType, Value<?>[] elems)` – Informace o poli, jehož prvky jsou primitivního typu. Toto pole obsahuje přímo hodnoty.
  - `objId` – Číselný identifikátor objektu, tedy pole.
  - `elemType` – Identifikátor typu, jehož prvky se v poli nachází. Tento identifikátor odpovídá typům definovaným v tabulce 4.
  - `elems` – Samotné elementy, které se v poli nachází. Opět se jedná o generiku, proto musíme pomocí `elemType` a překladové tabulky zjistit typ a následně hodnotu přetypovat, abychom získali korektní hodnotu.

Kromě těchto metod, které budou potřeba, obsahuje rozhraní ještě velké množství metod, které se starají o zpracování ostatních informací z HPROF dumpu, které jsou v tomto případě zbytečné (informace o vláknech, zásobníku, rámcích, alokacích...). Abychom je nemuseli zbytečně implementovat (či vynechávat), definuje knihovna výchozí implemen-

taci – `NullRecordHandler`, se kterou v dokumentaci doporučuje pracovat v základu. Díky tomu můžeme tuto třídu rozšířit a přetížit pouze metody, které nás skutečně zajímají – tedy ty, které jsou popsány výše. Ostatní tak můžeme ignorovat a zbytečně neznečišťují definici třídy prázdnými metodami.

8.3.4 Definované entity

Knihovna rovněž definuje některé entity (struktury), které umožňují snadnější zpracování dat načtených pomocí výše popsaných metod. Opět jsou některé v tomto případě zbytečné, proto se budu jen (stručně) věnovat těm, které využiji.

ClassInfo

Třída, která uchovává informace o definicích tříd, které se v souboru HPROF nacházejí. Obsahuje některá data z metody `classDump`, konkrétně ta, která jsem popisoval výše.

Instance

Jak vypovídá název této entity (a vlastně i ostatních), objekty této třídy představují instance tříd definovaných prostřednictvím `ClassInfo`.

InstanceField

Představuje instanční proměnnou. Kromě typu (viz dále `Type`) obsahuje i název proměnné (`fieldNameStringId`), byť stále jako referenci typu `long`.

Constant

Třída reprezentující konstanty. Kromě hodnoty, která je uchovávána jako `Value<?>`, je zde `constantPoolIndex`, což představuje index v poli (tedy de facto ukazatel) konstant, o kterém jsem mluvil v kapitole 3.6.

Static

Statická položka třídy. Kromě hodnoty, která je generického typu `Value<?>`, obsahuje i název `staticFieldNameStringId`, jako referenci typu `long`.

Type

Jedná se o výčtový typ `enum`, který definuje typy dle tabulky 4, včetně jejich názvů a délky v bytech.

Value

`Value<T>` je generický typ, který slouží pro uchovávání hodnoty `T`. Kromě této hodnoty navíc uchovává typ `Type`.

Jak si bylo možné všimnou výše, většinou je v knihovně s třídou `Value` pracováno pomocí nespecifikované generiky `?`, což je trochu nešťastné řešení, které programátora využívající tuto knihovnu nutí k přetypování na základě hodnoty proměnné `Type`. To může vyvolat varování typu *Unchecked cast* a jako takové vyžaduje věnování zvýšené pozornosti této části programu, protože může být zdrojem neočekávaných chyb. Takový blok kódu je tedy nutné důsledně ošetřit a ověřit, že přetypování můžeme skutečně bezpečně provést.

8.3.5 Použití knihovny

Protože knihovna přistupuje k načítání dumpu stále poměrně nízkoúrovňově, tedy na úrovni referencí jako celočíselných hodnot `long`, jednotlivých bytů atd., bylo by vhodné si napsat vlastní `RecordHandler`, za dodržení doporučení z dokumentace knihovny, tedy rozšířením třídy `NullRecordHandler`. Jediným účelem této třídy bude zachytit a zaznamenat všechny údaje tak, aby se daly dále pohodlně zpracovat.

Nad touto implementací rozhraní `RecordHandler` navrhuji vytvořit obálku (v terminologii `wrapper`, tj. návrhový vzor `Proxy`), který bude předzpracovaná data využívat (bude tedy `RecordHandler` využívat formou delegování) a již je bude publikovat rovnou do aplikace. Oddělím tak tedy nízkoúrovňovou práci od té vyšší, abstraktnější úrovně, a zároveň tím dojde k oddělení API, čímž bude eventuálně možné nahradit knihovnu pro zpracování HPROF souboru jiným nástrojem či vlastní implementací při zachování API pro vysokoúrovňovou práci. Takové oddělení je ekvivalentem *Repository* a *Service* v případě přístupu k databázi (či jinému úložišti) – při změně databáze, její struktury či přímo typu úložiště pouze změníme *Repository*, jehož funkcionalitu zastřešuje *Service*, který teprve využíváme ve zbytku aplikace – koncentrujeme tak změnu pouze na jedno místo a zbytek aplikace zůstane netknutý. Stejně tak, jako bychom v případě této analogie užívali *Service*, budeme využívat tuto obálku.

Definování závislosti

Problémem, který při použití knihovny nastává, je udržitelnost – respektive způsob jejího získání. V praxi není vhodné získávat závislosti pouhým přímým stažením zdrojových souborů nebo vytvořeného souboru JAR. Ztrácí se tím informace o tom, odkud soubory pochází, není možné je pohodlně a efektivně aktualizovat při budoucích změnách apod.

Proto je nutné (či přinejmenším vhodné) v projektu používat nástroje pro definici a získávání závislostí, jako třeba *Maven* (který budu v projektu využívat), *Ant* nebo *Gradle* (či jejich ekvivalentů v dalších jazycích a platformách). Knihovna *Hprof Heap Dump parser*

Ukázka 8.2: Definování Jitpack repozitáře v Mavenu

```
<repositories>
    <repository>
        <id>jitpack.io</id>
        <url>https://jitpack.io</url>
    </repository>
</repositories>
```

sice definuje Maven modul prostřednictvím souboru *POM*, nicméně není součástí žádného repozitáře – konkrétně toho defaultního, tedy `maven.org`. Existují způsoby, jak proměnit v Maven repozitář přímo GitHub – nicméně repozitář projektu nedefinuje žádnou verzi či stabilní větev, pouze `master`. To je z pohledu vývoje nebezpečné, protože může kód kdykoliv v průběhu vývoje rozbít prostou obnovou závislostí – a může tak námi vyvíjený kód učinit nefuknčním.

Vytvořil jsem proto fork (odnoš, zrcadlo původního repozitáře) ve vlastním profilu[2], ve kterém jsem provedl *Maven release*, čímž jsem vytvořil verzi projektu. Zároveň s vydáním jsem definoval Git tag (což je lidsky čitelný název pro ukazatel na konkrétní verzi kódu v čase) *hprof-parser-1.0*. Tím je zaručeno, že se odkazovaná verze projektu nikdy nezmění a zároveň tím nepřicházím o kontrolu nad případnou aktualizací projektu – tu můžu provést případnou obnovou forku projektu a vydáním další verze. Nicméně je takto zaručeno, že podoba verze *hprof-parser-1.0* bude stále stejná. Tu umožňuje případnou práci v týmu (všichni jeho členové mají dostupné stejné verze závislostí) a také výrazně přispívá ke stabilitě vyvíjeného softwaru.

Pro zpřístupnění repozitáře na GitHubu skrze Maven je možné využít projekt *jitpack.io*. Kromě Mavenu dokáže tento projekt rovněž obsluhovat i Gradle. Pro jeho použití je nutné do projektu přidat Maven repozitář:

Díky tomu máme dostupné všechny veřejné GitHub repozitáře jako Maven závislosti. Poté už můžežme specifikovat závislost samotnou, a to pomocí ID skupiny (`groupId`) ve tvaru `com.github.<username>` (kde `<username>` je uživatelské jméno na GitHubu) a jako ID artefaktu (`artifactId`) uvedeme název repozitáře. Jako verzi můžeme uvést tag (který jsme definovali výše), `master-SNAPSHOT` pro `master` větev (jak vypovídá název, jedná se o `SNAPSHOT` a taková závislost tedy nemá být použita nikde kromě vývoje) nebo ID Git commitu.

Pro knihovnu můžeme tedy přidat závislost:

[2]Fork projektu je dostupný na `https://github.com/mxmxcz/hprof-parser`.

Ukázka 8.3: Definování závislosti v Mavenu

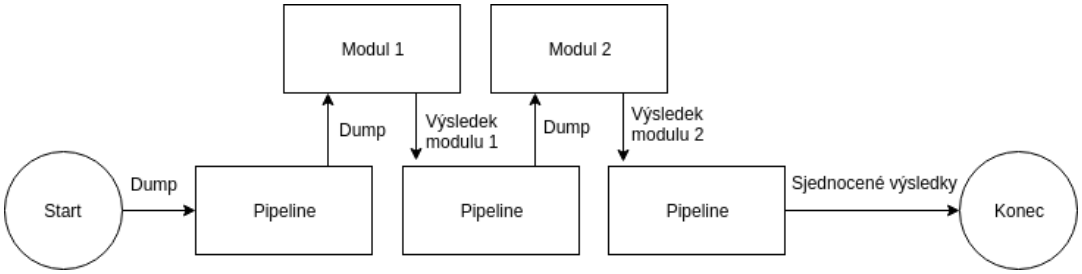
```
<dependencies>
    <dependency>
        <groupId>com.github.mxmxcz</groupId>
        <artifactId>hprof-parser</artifactId>
        <version>hprof-parser-1.0</version>
    </dependency>
</dependencies>
```

8.4 Návrh struktury aplikace

Výše jsem definoval závislost na knihovně, který se bude starat o načtení a předzpracování paměťového dumpu ze souboru HPROF. Tato (stále ještě primitivní, triviální) reprezentace dumpu v paměti bude představovat vstupní data pro algoritmus, který je bude dále transformovat.

Protože se dá očekávat, že mnou implementované analyzátory paměťových nedostatků a neefektivního využití paměti nebudou ani zdaleka kompletní, bylo by vhodné strukturu této části aplikace navrhnout tak, aby byla následná implementace dodatečných typů analýzy co nejjednodušší. Zároveň ale musí mít případný modul k dispozici všechna nutná data z dumpu tak, aby nebyl nástrojem nijak omezen.

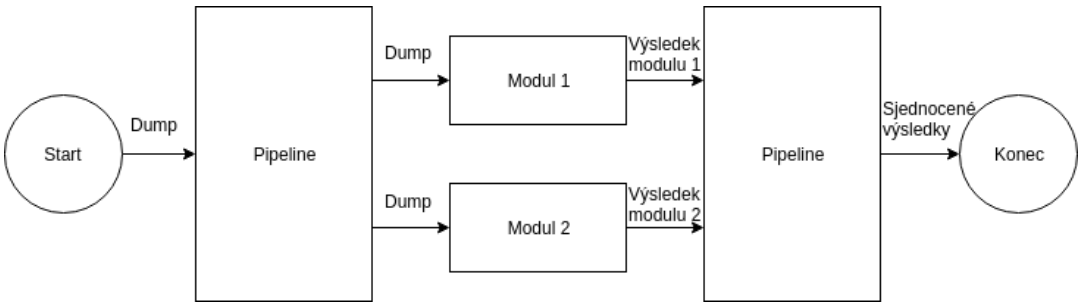
Tento problém jsem se tedy rozhodl řešit formou „výrobní linky“ – často se také používají výrazy *filter* či *pipeline*. Jednotlivé analyzátory problémů (dále také *moduly*) představují kroky na pomyslné výrobní lince, kterou prochází zpracovaný memory dump. Postupně tak na něm analyzátory mohou pracovat a vracet výsledky, které našly – každý z nich se bude specializovat na jeden konkrétní typ. V budoucích verzích by také bylo teoreticky možné definovat závilost modulů s analyzátory na sobě, čímž by mohly využívat výsledky zpracované některým z předchozích modulů – tuto funkcionalitu nicméně v rámci této práce implementovat nehodlám.



Obr. 11: Architektura jednotlivých modulů pro zpracování memory dumpu.

Na obrázku 11 je jednoduchý diagram takové architektury. Jako *pipeline* zde označuji řídící prvek, který má seznam zaregistrovaných modulů, řídí jejich spouštění (včetně třeba pořadí a paralelizace) a sbírá výsledky. Jako vstup přijímá reprezentaci dumpu. Ten předává modulu 1, který nad nám provede svou analýzu a vrátí zpět výsledek. Ten si *pipeline* poznamená a obdobně spustí postupně všechny další moduly. Nakonec výsledky sjednotí a vrátí je jako návratovou hodnotu.

Výhoda této architektury je vysoká míra nezávislosti kódu na sobě – není tak problém přidat nový modul, s pouze minimálním zásahem do existujícího kódu (nebo také teoreticky bez zásahu, za využití např. autokonfigurace ve frameworku Spring). Kromě toho není problém takový kód paralelizovat - moduly jsou na sobě naprosto nezávislé. Architektura takové paralelizace je nastítěna na obrázku 12.



Obr. 12: Paralelní varianta k seriové architektuře z obrázku 11.

8.5 Rozhraní pipeline a modulů

Rozhraní poskytované pro implementace pipeline a modulů musí umožňovat kompletní analýzu zpracovávaného dumpu.

TODO

9 IMPLEMENTACE

Nástroj jsem vytvořil jako konzolovou (CLI) aplikaci, jejíž použití, včetně možných parametrů, je popsáno v uživatelské dokumentaci v příloze A.1. Aplikace slouží pouze jako vstupní bod a obsluhuje knihovnu, která je striktně oddělena – k tomu využívám Maven moduly.

Hlavní rodičovský modul, který zastřešuje všechny ostatní, je `memory-analyzer`. Ten definuje tři submoduly:

analyzer Modul pro analýzu memory dumpu, tj. obsahující hlavní funkcionalitu. Dále se tedy budu věnovat popisu tohoto modulu.

app Poskytuje CLI. Z vytvořeného programu tak dělá spustitelnou aplikaci. Toto oddělení umožňuje jednoduché využití knihovny (tedy modulu `analyzer`) nezávisle v jiné aplikaci.

example-app Slouží pro testovací účely, jako jednoduchý prostředek pro vytváření memory dumpů, na nichž je knihovna testována.

9.1 Zpracování dumpu

Hlavním rozhraním celé knihovny je `MemoryDumpAnalyzer`, respektive jeho výchozí implementace – třída `DefaultMemoryDumpAnalyzer`. Ty obsahují dvě metody, a to pro získání jmenných prostorů (*namespace*) a pro analýzu jako takovou.

`DefaultMemoryDumpAnalyzer` přijímá jako parametr konstruktoru cestu ke zpracovávanému memory dumpu (HPROF). Pro načtení jmenných prostorů potřebuje pouze právě cestu a implementaci abstraktní třídy `RecordHandler`.

Tato třída rozšiřuje třídu `NullRecordHandler`, která byla zmíněna v kapitole 8.3.3. Slouží tedy jako prostředník mezi mnou vytvořeným nástrojem a knihovnou pro zpracování dumpu. Protože `NullRecordHandler` je skutečně nízkoúrovňový a nabízí pouze metody pro přístup k datům memory dumpu, třída `RecordHandler` k ní přidává navíc metodu pro zisk předzpracovaného memory dumpu, třídy `RawMemoryDump`. Prefixem `Raw` označuji v programu třídy, které představují objekty z dumpu, ale nejsou ještě nijak zpracované a např. číselné hodnoty v nich nejsou nijak intepretované (např. reference na objekty jsou stále celočíselné hodnoty typu `long`).

Důvodem pro toto řešení je způsob, jakým knihovna pro zpracování dumpu pracuje. Nenabízí žádný prostředek k získání všech dat, ale s každou načtenou částí dumpu zavolá metodu třídy *NullRecordHandler* dle typu této části – např. pro načtený rámec zásobníku zavolá metodu `stackFrame` s daty tohoto rámce. Obdobně takto volá metody pro ostatní typy.

`RawRecordHandler` je tedy rozšířením `NullRecordHandler`. Při každém takovém volání metody si třída pouze uloží data jako mezivýsledek do jedné z map či listů, které

obsahuje. Díky tomu je po kompletním průchodu knihovnou třída schopna poskytnout výše uvedený `RawMemoryDump`, což je třída agregující tyto výsledky, a to zavoláním metody `getMemoryDump`. Třída rovněž provádí základní převod z číselných identifikátorů typů na textovou reprezentaci (např. z 10 na `int`). Z takto předzpracovaného dumpu je možné rovněž získat jmenné prostory, které obsahuje.

Takto získaný předzpracovaný dump předá `DefaultMemoryDumpAnalyzer` pro další analýzu do třídy `GenericMemoryDumpProcessor`. Tato třída se stará o interpretaci dat a překládá tedy `Raw` objekty do jejich finální podoby. Nahrazuje tedy např. číselné identifikátory řetězců jejich skutečnými hodnotami, místo ID tříd přidává referenci na jejich reprezentaci v podobě třídy `ClassDump` apod.

Jako obal okolo `DefaultMemoryDumpAnalyzer` slouží třída `FilteredMemoryDumpProcessor`. Ta implementuje stejné rozhraní, drtivou většinu funkcionality deleguje na `Default` variantu, ale přidává navíc (jak napovídá název) možnost filtrovat objekty podle specifikovaných jmenných prostorů. Není nutné tak dále pracovat se všemi objekty všech tříd, ale omezit se na ty, které uživatele zajímají.

V tomto okamžiku je tedy k dispozici kompletně zpracovaný memory dump a v paměti se nachází jeho reprezentace pomocí objektů definovaných ve jmenném prostoru `cz.mxmx.memoryanalyzer.model`. Třídy a instance jsou navíc volitelně filtrovány pomocí specifikovaných balíků, do kterých patří (i když jsou samozřejme k dispozici i všechny ostatní objekty z dumpu).

9.2 Analýza plýtvání pamětí

Takto zpracovaný dump, tedy instance třídy `ProcessedMemoryDump` (respektive obecně rozhraní `MemoryDump`), je možné dále zpracovávat. Funkcionalita analýzy plýtvání je oddělena od zpracování dumpu. To tak může uživatel nástroje využít, analýzu plýtvání vynechat a nad zpracovaným dumpem provádět vlastní operace.

Pro analýzu plýtvání je k dispozici rozhraní `WasteAnalyzer`, s pouze jedinou metodou, jak je vidět v ukázce 9.1. Toto rozhraní představuje *pipeline* i *moduly* z analýzy v kapitole 8.4 – sdílejí tedy částečně vnější rozhraní. *Pipeline* se odlišuje až prostřednictvím abstraktní třídy `WasteAnalyzerPipeline`, nicméně nijak zásadně – od dědících tříd je pouze vyžadován seznam dalších implementací rozhraní `WasteAnalyzer`, které má tato *pipeline* obsluhovat. Kromě toho přijímá ještě parametr `multiThreaded`, který přepíná mezi během v jednom či více vláken tak, jak bylo nastíněno na obrázcích 11, respektive 12.

Výchozí implementací je `DefaultWasteAnalyzerPipeline`, kde jsou registrovány jednotlivé moduly a následně jsou spuštěny – standardně pomocí více vláken. Nové moduly lze zaregistrovat jejich přidáním do mapy, která je uložena v konstantě `ANALYZERS`,

Ukázka 9.1: Rozhraní pro analýzu plýtvání paměti ve zpracovaném dumpu

```
public interface WasteAnalyzer {
    List<Waste> findMemoryWaste(MemoryDump memoryDump);
}
```

společně s názvem tohoto modulu. Tím je tento modul automaticky zaveden do běhu programu a spouštěn při analýze, další kroky není třeba provádět.

9.2.1 Analýza duplicit

Tento druh plýtvání hledá třída `DuplicateInstanceWasteAnalyzer`. Omezuje se na jmenné prostory specifikované uživatelem, což je samozřejmě opatření z výkonových důvodů – analýzá má velice neefektivní (exponenciální) složitost, takže i přes toto zúžení analyzované množiny trvá běh tohoto modulu často poměrně dlouho.

Testoval jsem spouštění analýzy paralelně, a to prostřednictvím metody `parallelStream`, nicméně žádná z testovaných metod paralelizace nepřinesla navíc žádné výhody – a to především kvůli potřebě synchronizace výsledků, které je nutné slučovat.

Algoritmus iteruje nad všemi (filtrovanými) instancemi. Zároveň si udržuje seznam instancí již zpracovaných. Ty přeskakuje, stejně jako porovnávání identických objektů (porovnání objektu se sebou samotným). Přeskakuje i objekty bez proměnných – algoritmus tak označoval výčtové typy jako duplicity.

Nejdříve modul ověří rovnost tříd – i s ohledem na dědičnost. Následně i rovnost počtu parametrů, bez níž nemá cenu instance porovnávat. Nakonec provede iteraci nad všemi parametry a zjistí jejich rovnost. V aktuální verzi analýza neřeší reference, tj. chová se k nim jako k číselným hodnotám.

9.2.2 Analýza kolekcí

Analyzování kolekcí se omezuje na objekty tříd rozšiřujících `AbstractList`. Obdobným způsobem by samozřejmě šly řešit i kolekce jiných typů. Detekce tohoto typu plýtvání je řešena ve třídě `ListWasteAnalyzer`. Algoritmus prochází všechny filtrované instance a zjišťuje, zda jejich třída rozšiřuje výše zmíněnou abstraktní třídu. Pokud ano, zjistí, zda počet prázdných položek (tedy obsahujících hodnotu `null`) překračuje určitou mez, nastavitelnou v konstantě `THRESHOLD` (výchozí hodnota je 50 %). Analýza tedy samozřejmě funguje jen na typech, jejichž implementace využívá na pozadí pole. Toto pole prochází a počítá prvky `null`, jejichž počet potom porovná s deklarovanou velikostí pole (prostřednictvím instanční proměnné `size`), za využití výše uvedené konstanty.

Druhým typem nad listy je hledání těch, které obsahují pouze duplicity – všechny jejich prvky mají tedy pouze jednu hodnotu. Tato analýza vynechává listy s hodnotou `null` (ty jsou podchyceny předchozí detekcí) a počet prvků musí být vyšší než 1.

10 OVĚŘENÍ IMPLEMENTACE

Program jsem testoval na několika testovacích souborech, které se lišily velikostí i složitostí – některé z nich byly výsledkem memory dumpu výše zmíněného testovacího modulu (`example-app`), ostatní data potom pocházela z reálných aplikací, spuštěných na mém počátači.

10.1 Testovací aplikace

Testovací modul umí vygenerovat poloprázdné kolekce, kolekce obsahující pouze duplikáty (konkrétně celočíselné) a objekty, které mají stejná data a zároveň vytvoří jednoduchou strukturu rodič-potomek. Jako parametr nástroj přijímá počet instancí, které má od každého typu vytvořit. Jedná se tedy o testování v „laboratorních podmínkách“, na programu, který je specificky navržený pro testování daného problému.

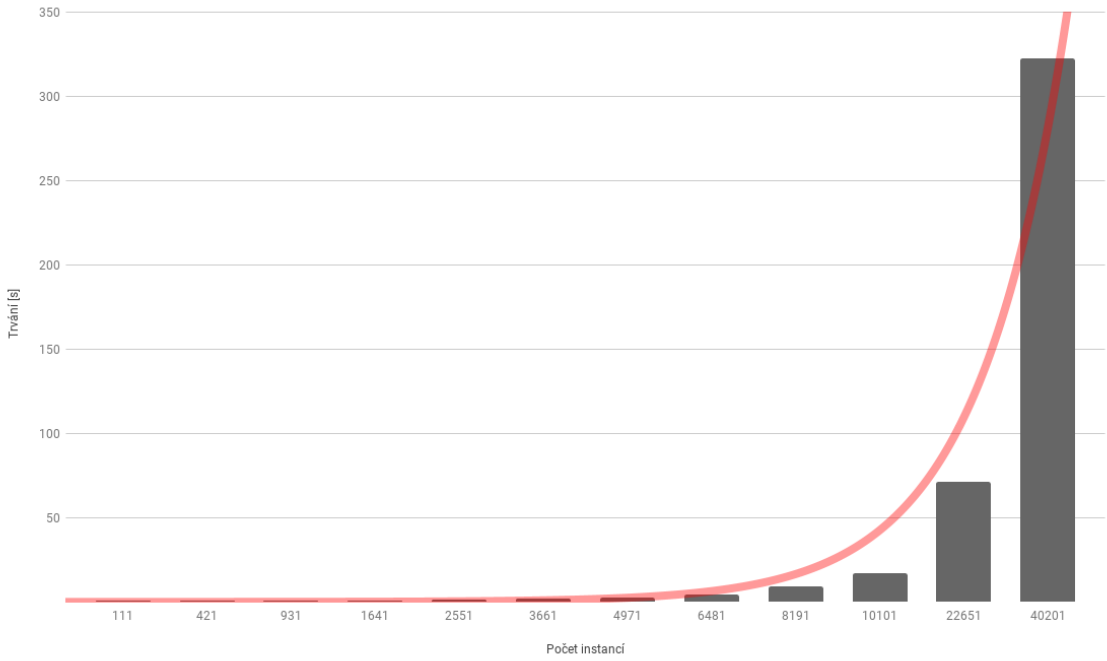
Aplikaci jsem postupně pustil s několika parametry (N), získal její memory dump a nad ním spustil analýzu, omezující se na jmenný prostor `cz.mxmx.memoryanalyzer.example`. Analýzu jsem spustil pro každý soubor 3x a časové trvání jsem zprůměroval. Výsledek ukazuje tabulka 5.

N	Počet instancí	Velikost dumpu [MB]	Nalezené problémy	Trvání [s]
10	111	2,1	15	0,5
20	421	2,6	25	0,5
30	931	3,0	35	0,7
40	1641	3,6	45	0,9
50	2551	4,0	55	1,2
60	3661	4,4	65	1,8
70	4971	5,4	75	2,8
80	6481	5,9	85	4,3
90	8191	6,4	95	9,3
100	10101	6,9	105	17,3
150	22651	11	155	71,4
200	40201	15	205	322,2

Tab. 5: Výsledek měření na testovací umělé aplikaci.

Díky datům v tabulce si lze povšimnou několika věcí. Doba délky běhu analýzy evidentně tolik nezáleží na velikosti dumpu, ale spíše na počtu instanců, které obsahuje. To je způsobeno metodikou testu a vlastnostmi analýzy – díky omezenému počtu tříd, které

testovací aplikace obsahuje. Algoritmus tak musí porovnávat velký počet instancí jako takových – v reálné aplikaci výrazně algoritmus urychluje fakt, že většina instancí se liší třídou, kterou představují. Tuto „identitu“ algoritmus testuje jako první a výrazně urychluje běh. Laboratorní aplikace tedy představuje „nejhorší možnou“ situaci – velké množství instancí pouze dvou tříd. Zhruba v 50 % případů tak musí porovnávat každou dvojici instancí; ve skutečné aplikaci by typicky tento ukazatel byl zlomek tohoto čísla a pohyboval se někde v jednotkách procent (vycházím z předpokladu, že v níže uvedeném testu frameworku *Spring Boot* byl poměr v testovaném jmenném prostoru průměrně zhruba 4 instancí na třídu; zde je to 13400).



Obr. 13: Vztah času běhu analýzy oproti počtu instancí u testovací aplikace (tři třídy). Trend (exponenciální křivka) je vyznačen červeně.

Jak je na grafu 13 vidět, v případě testovaí aplikace se složitost analýzy reálně skutečně blíží exponenciále, tedy hodnotě $O(2^n)$, k níž jsem došel v analýze. Je to opět způsobeno výše uvedeným poměrem instancí a tříd. Nicméně i k tomuto stavu ve výjimečných stavech a v případě specifických aplikací můžeme dospět a je vidět, že v některých případech při běhu aplikace můžeme narážet na toto omezení a výpočet může trvat delší čas. V reálné aplikaci se bude složitost blížit spíše $O(k2^{n_k})$, kde k je počet tříd a n_k je počet instancí dané třídy. Tato složitost je sice stále exponenciální, ale můžeme očekávat, že n_k se bude pohybovat ve výrazně nižších hodnotách, k potom ovlivňuje výběr jmenného prostoru.

Samozřejmě platí, že čím konkrétnější jmenný prostor vybereme a čím méně tříd bude obsahovat, tím rychlejší analýza bude.

Dále lze v tabulce ověřit test úplnosti – jak je vidět, počet nalezených problémů je přímo úměrný N, na jehož základě je vytvořen příslušný počet instancí. V testovací aplikaci je pro každý z N rodičovských prvků vytvořeno N potomků. Všichni tito potomci mají stejná data – vytváří tedy N „agregačních tříd“, v nichž mají všechny objekty stejná data. Každá taková třída je jedním problémem, tedy jedním případem plýtvání paměti. Takových tříd existuje tedy N (pro každého z rodičů). Konstantně je poté vytvořeno 5 listů, na nichž jsou testovány zbylé případy neefektivného využití – plýtvání místem v listech. Předpokládaný počet nalezených problémů je tedy $N + 5$ a jak vidíme v tabulce, tento předpoklad detekce algoritmu splňuje.

Nabízí se samozřejmě otázka, zda je program pro generování testovacích dat samotný správně. Kromě toho, že se jedná o skutečně jednoduchou aplikaci, si lze i zhruba ověřit počet generovaných instancí. Z výše uvedeného postupu pro generaci lze odvodit i předpokládaný počet těchto instancí, a to N^2 pro instance potomků, N pro instance rodičů a 1 instanci pro obslužnou aplikaci samotnou. Výsledný počet instancí je tedy $N^2 + N + 1$, což plně odpovídá reálným hodnotám.

V příloze A.2 je vidět výpis programu (pro $N = 10$). Jak je vidět, z celkového počtu 710 tříd a 7707 instancí program podle jmenného prostoru vyfiltroval 3 třídy a 111 instancí. Nad nimi začal hledat problémy s plýtváním paměti, kterých našel celkem 15 – 10 případů duplikátních instancí, 4 neefektivního využití paměti a 1 případ listu, který má stejné hodnoty. Program běžel 471 ms (doba běhu programu je vypsána ve formátu *ISO-8601*).

Pomocí přepínače obslužné aplikace (který je v tomto případě zapnutý) je možné i vypisovat pole dotčených instancí. Jak je tedy vidět v případě duplikátů, například v prvním případě je hodnota pole `parent` instance třídy `Parent` číslo 29108685520 a pole `name` s hodnotou `Child 1`. Všech 10 instancí třídy má v tomto případě tedy tyto hodnoty, i když se jedná o různé objekty. Analogicky to potom platí pro zbylých 9 případů duplikátů.

Dalším typem plýtvání paměti je v tomto příkladě neefektivní využití paměti. Ve výpisu je vidět, v jaké třídě (včetně čísla instance) se list nachází, v jaké třídní proměnné a kolik místa je vyplýtváno. Rovněž je vidět ID pole s jeho prvky a velikost – ta může být například i 0 (v prvním případě), kdy je pole prázdné, protože má nastavenu výchozí velikost. Java jako výchozí velikost v případě prázdného listu nastavuje 10, což se samozřejmě postupně (skokově) zvyšuje s přibývajícími prvky.

10.2 Testování Spring Boot

Jako reálnou testovací aplikaci, která má otevřený zdrojový kód, jsem si vybral framework *Spring Boot*, v němž jsem vytvořil základní aplikaci *Hello World*. Testoval jsem tedy framework samotný a ne aplikaci v něm vytvořenou. Dump z ní vytvořený má 74 MB. Výsledek testování je vidět v tabulce 6.

Jmenný proystor	**Třídy**	**Instance**	**Nalezené problémy**	**Trvání [s]**
org	2416	9093	347	14,8
org.springframework	1555	6053	329	8,2
org.springframework.boot	380	1506	27	4,2
org.springframework.core	196	1585	5	4,4
org.springframework.web	296	239	37	4,1
org.springframework.boot.web	75	27	1	4,0

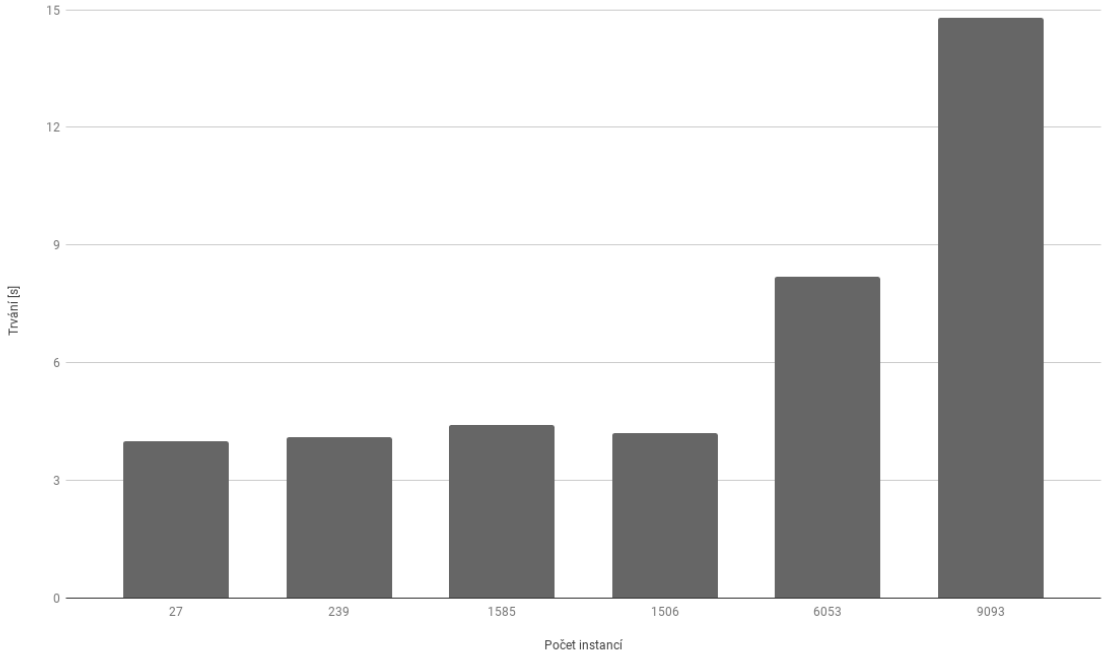
Tab. 6: Výsledek testování reálné aplikace – Spring Boot

Opět jsem každý set spustil třikrát a výsledek trvání zprůměroval. Zvolil jsem v tomto případě odlišnou metodiku – v případě reálné aplikace si samozřejmě nemůžeme zvolit velikost vygenerovaných dat. Proto jsem zvolil různé jmenné prostory s různými parametry, včetně poměrně obecného `org`.

Jak je v tabulce vidět, charakteristiky dumpu se od testovací aplikace zásadně liší. Pro nejobecnější balík, tedy `org`, je poměr instancí na třídu zhruba 4 a všechny hodnoty se drží v nižších jednotkách, v případě posledního rostoru je tříd dokonce třikrát více, než instancí. Jak je vidiět, tato změna výrazně ovlivnila rychlost běhu programu tak, jak jsem uváděl v testu umělé aplikace. Nelze tedy evidentně tvrdit, že je složitost ve všech případech exponenciální, což je pozitivní zjištění. Na grafu 14 je tato závislost doby běhu na počtu instancí vykreslena.

Jak je vidět, minimálně v rozsahu testovacích dat už algoritmus nabízí lepší než exponenciální složitost. Byť bychom mohli z dostupných dat dojít k závěru, že předzpracování dumpu trvá zhruba 4 sekundy a analýza je tak ve 4 případech zanedbatelná, není tomu tak – pro každý jmenný prostor je analýza jiná, tj. je závislá na velikosti tohoto prostoru. Pro potvrzení této hypotézy by toto zpracování muselo být totožné, což neplatí.

Pokud nahlédneme do výsledků, zjistíme, že algoritmus odhalil třeba list `lines` ve třídě `DeferredLog`, která má sice nastavenu velikost 109, nicméně neobsahuje ani jednu hodnotu. Dále například 38 instancí třídy `Signature` obsahuje úplně stejná data, třídy `DefaultFlowMessageFactory'` potom 34 instancí.



Obr. 14: Vztah času běhu analýzy oproti počtu instancí u Hello World Spring Bootu.

Z vypsaných proměnných instancí je vidět, že všechny duplicity jsou objekty poměrně malých tříd. Logicky, u větších tříd je více prostoru pro odlišení – pak stačí, aby pouze jedna např. číselná hodnota měla jinou hodnotu a algoritmus objekt již jako duplicitu nedetekuje.

10.3 Intepretace výsledků

Vytvořený nástroj evidentně dokáže spolehlivě odhalit všechny uměle vytvořené problémy, jak bylo prokázáno na analýze testovací aplikace. Stejně tak dokáže detekovat plýtvání paměti v existujícím, běžně používaném frameworku a to i přes to, že v něm vytvořená aplikace byla svým rozsahem naprosto triviální a vlastně tak framework jen spustila. Lze očekávat, že v programu několikanásobně většího rozsahu bude problém četnější, byť za cenu delší analýzy.

Doba běhu je evidentně závislá nejen na velikosti memory dumpu a počtu tříd a instancí, ale i na dalších charakteristikách, jako poměru instancí vůči třídám a počtu agregačních tříd, které algoritmus vytvoří – tedy kolik různých duplicit se v dumpu nachází. Nicméně délku běhu nelze, kromě minoritních vylepšení, nijak zásadně zkrátit – její limitací je složitost, která je, pro všechny dvojice z množiny objektů, zákonitě exponenciální.

Nicméně si nemyslím, že je doba běhu zásadní překážkou. Analýza podobného typu pravděpodobně nikdy nebude součástí standardního procesu vývoje či testování, ale spíše jednorázovým nástrojem pro řešení problémů. V takovém případě není nutně vyžadována rychlost spuštění, ale spíše korektní a úplná detekce tak, aby bylo možné případný problém efektivně odstranit. V případě, že by to problém byl a analýza duplicit není zapotřebí, je možné (odstraněním jednoho řádku) tento modul vypnout a běh programu tak podstatně zrychlit – tím je ale samozřejmě omezen pouze na neefektivní využití kolekcí, případně na funkcionalitu dalších modulů, které si uživatel může eventuálně doplnit.

11 ZÁVĚR

Jak je vidět, problémy s plýtváním paměti v reálném softwaru existují. Otázkou samozřejmě je, zda jsou jejich dopady natolik závažné, aby způsobovaly potíže, respektive zda jejich odstranění přináší znatelný rozdíl v užití systémových prostředků nebo rychlosti běhu programu.

Navržený nástroj dokáže na tyto problémy poukazovat a ve snímku běžícího programu je detekovat. Zároveň vytváří platformu, na které mohou stavět další nástroje a která je, díky striktnímu oddělení knihovny od aplikace, dostatečně flexibilní pro to, aby byl nástroj využitelný i pro jiné druhy práce s memory dumpem. V programu jsem kladl důraz na to, aby byl program jednoduše rozšiřitelný a aby například přidání nového modulu pro detekci neefektivního využití paměti znamenalo pouze implementaci jednoho rozhraní a přidání nové řádky pro registraci tohoto modulu. V memory dumpu se nepochybně dá nalézt celá řada způsobů, jakými lze v běžícím programu paměť ušetřit a práce zcela jistě všechny nepodchycuje. Kandidátem pro takové vylepšení je zpracování duplicit i s ohledem na reference, což je funkcionalita, která v rámci této práce implementována nebyla.

SEZNAM OBRÁZKŮ

1 Struktura Java paměti [27]. 14
2 Využití paměti a provedení garbage kolekce před vytvořením dumpu (červeně). 19
3 Problém analýzy paměti za běhu programu. 23
4 Postup volání hierarchie class loaderů. 25
5 Rozhraní Eclipse MAT 27
6 Rozhraní VisualVM 28
7 Rozhraní Java Mission Control 29
8 Rozhraní JProfileru 30
9 Rozhraní JHAT 31
10 Strom porovnávání objektů 36
11 Architektura jednotlivých modulů pro zpracování memory dumpu. 47
12 Paralelní varianta k seriové architektuře z obrázku 11. 48
13 Vztah času běhu analýzy oproti počtu instancí u testovací aplikace (tři třídy). Trend (exponenciální křivka) je vyznačen červeně. 54
14 Vztah času běhu analýzy oproti počtu instancí u Hello World Spring Bootu. 57

,

SEZNAM ZKRATEK

JVM Java Virtual Machine

AOT Ahead-Of-Time

JIT Just-In-Time

GC Garbage Collector

PID Process Identifier – Identifikátor procesu

CLI Command Line Interface

OQL Object Query Language

LITERATURA

[1] TODO Citace Moore's law

[2] TODO Citace mzdy programátorů

[3] TODO https://www.graalvm.org/docs/getting-started/#native-images

[4] Developer Survey Results 2018 - Most Popular Technologies. Stack Overflow Insights [online]. 2018 [cit. 2019-03-02]. Dostupné z: https://insights.stackoverflow.com/survey/2018#most-popular-technologies

[5] TIOBE Index for March 2019. TIOBE - The Software Quality Company [online]. 2019 [cit. 2019-03-02]. Dostupné z: https://www.tiobe.com/tiobe-index/

[6] JSR-133. JSR-133: JavaTM Memory Model and Thread Specification: Proposed Final Draft. 2004. Dostupné z: https://download.oracle.com/otndocs/jcp/memory_model-1.0-pfd-spec-oth-JSpec/

[7] Java Threads Scott Oaks, Henry Wong "O'Reilly Media, Inc.", 2004

[8] Java 9 High Performance: Practical techniques and best practices for ... , Mayur Ramgir, Nick Samoylov

[9] https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

[10] TODO https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#gc()

[11] TODO https://openjdk.java.net/jeps/241

[12] https://stackoverflow.com/a/6471947/3819750

[13] TODO https://docs.oracle.com/en/java/javase/11/migrate/migration-guide.pdf

[14] TODO https://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html

[15] TODO http://www.csd.uwo.ca/courses/CS4411b/pdf02manuals/oql.pdf

[16] TODO https://books.google.cz/books?id=bAA9Z5WLdjQC&dq=OQL+specification&hl=cs&source=gbs_navlinks_s

[17] TODO http://www.cs.utexas.edu/users/mckinley/papers/mmtk-sigmetrics-2004.ps

[18] TODO https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

[19] TODO Mastering Java 11: Develop modular and secure Java applications using concurrency and advanced JDK libraries, 2nd Edition

[20] TODO https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

[21] TODO https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html

[22] TODO https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html#jvms-1.2

[23] TODO https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.6

[24] https://www.oracle.com/technetwork/java/whitepaper-135217.html

[25] TODO https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html

[26] TODO Enterprise Java Security: Building Secure J2EE Applications Od autorů: Marco Pistoia, Larry Koved, Nataraj

[27] TODO https://medium.com/@govinda_raj/all-you-need-to-know-jvm-36c0b59ba969

[28] TODO http://hg.openjdk.java.net/jdk6/jdk6/jdk/raw-file/tip/src/share/demo/jvmti/hprof/manual.html

SEZNAM PŘÍLOH

A Přílohy **66**

A.1 Uživatelská dokumentace 66

A.2 Výpis programu 67

A PŘÍLOHY

A.1 Uživatelská dokumentace

A.2 Výpis programu

```
Analyzing classes from namespace cz.mxmx.memoryanalyzer.example in ../sandbox/data/test−heapdump−10.hprof...

Done, found:
    Classes: 710
    Instances: 7707
Namespace cz.mxmx.memoryanalyzer.example
    Classes: 3
    Instances: 111

Analyzing memory waste...
Done, found 15 possible ways to save memory:
    Duplicate instances (10):
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108685520}
            name = Child 1
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108684560}
            name = Child 2
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108686480}
            name = Child 0
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108650888}
            name = Child 3
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108649928}
            name = Child 4
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108647048}
            name = Child 7
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108646088}
            name = Child 8
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108648008}
            name = Child 6
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108645128}
            name = Child 9
        Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
            parent = InstanceDump{classDump=cz.mxmx.memoryanalyzer.example.Parent, instanceId=29108648968}
            name = Child 5
    Ineffective size usage (4):
        List full or mostly consisting of nulls: Values in the java.util.ArrayList in cz.mxmx.memoryanalyzer.example.App(29108546080)\#emptyList have a null value
            ↪ (10000/10000x).
            size = 0
            elementData = 29108787560
        List full or mostly consisting of nulls: Values in the java.util.ArrayList in cz.mxmx.memoryanalyzer.example.App(29108546080)\#listOfDuplicates have a null value
            ↪ (9990/10000x).
            size = 10
            elementData = 29108997736
        List full or mostly consisting of nulls: Values in the java.util.ArrayList in cz.mxmx.memoryanalyzer.example.App(29108546080)\#sameList have a null value
            ↪ (9990/10000x).
            size = 10
            elementData = 29108717000
        List full or mostly consisting of nulls: Values in the java.util.ArrayList in cz.mxmx.memoryanalyzer.example.App(29108546080)\#mostlyEmptyList have a null value
            ↪ (9990/10000x).
            size = 10
            elementData = 29108860312
    List of duplicates (1):
        List full of same values: All values in the list java.util.ArrayList\#listOfDuplicates have the same value (10x).
            size = 10
            elementData = 29108997736
Duration: PT0.471S
```