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

Diplomová práce

Detekce problémů se správou paměti v Java aplikacích

Plzeň 2021 Bc. Hoang Ngoc Hung

ZÁPADOČESKÁ UNIVERZITA V PLZNI

Fakulta aplikovaných věd

Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Hung HOANG NGOC**
Osobní číslo:	**A18N0086P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Detekce problémů se správou paměti v Java aplikacích**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s existujícími nástroji pro analýzu Java memory heapu.
2. Seznamte se problematikou alokace paměti ve virtuálním stroji Javy a se známými typy jejího zbytečného čerpání („memory bloat“, „memoryleak“).
3. Navrhněte možná rozšíření nástroje pro analýzu paměti Javy (DP Nástroj pro analýzu Java memory heap, Martin Mach, obhájeno 2019).
4. U navrženého rozšíření odhadněte potenciál rozšíření odhalovat problémy a reálných aplikacích, implementujte a důkladně otestujte.
5. Ověřte funkčnost vytvořeného rozšíření a rozšířenost detekovaného problému na sadě reálných aplikací.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

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

Datum zadání diplomové práce: **11. září 2020**
Termín odevzdání diplomové práce: **20. května 2021**

L.S.

Doc. Dr. Ing. Vlasta Radová
děkanka

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

V Plzni dne 24. září 2020

Prohlášení

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

V Plzni dne 18. května 2021

Bc. Hoang Ngoc Hung

Poděkování

Rád bych poděkoval Ing. Richardu Lipkovi, Ph.D. za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování této diplomové práce. Velký dík patří také rodině za podporu během studia.

Abstract

Java is an object-oriented programming language that is popular due to its garbage collection and platform independency. Object-oriented programming, as a culture, encourages programmers to only pay attention to model. This trend can lead to memory bloat. Some cases, which can lead to memory bloat, are described. The goal of this thesis is to extend an existing tool, which can analyze memory bloat, to be able to detect deeply duplicated objects. The extended tool analyzed multiple real applications and revealed numerous cases of duplicate objects.

Abstrakt

Java je objektově orientovaný programovací jazyk populární díky automatické správě paměti a platformové nezávislosti. Kultura objektově orientovaného programování vybízí programátory, aby se soustředili především na objektový návrh aplikace, což je jednou z možných příčin neefektivního využití paměti. Práce popisuje některé případy, při kterých může docházet k neefektivnímu využívání paměti. Cílem práce je rozšíření nástroje, který je schopen detekovat případy plýtvání pamětí, o analýzu hluboce duplicitních objektů. Nástroj analyzoval několik reálných aplikací a odhalil velké množství případů duplicitních objektů.

Obsah

1 Úvod

Java je objektově orientovaný programovací jazyk populární díky automatické správě paměti a platformové nezávislosti. Kultura objektově orientovaného programování vybízí programátory, aby se soustředili výhradně na objektový návrh aplikace. Tento trend výrazně ovlivnil návrh a používání Javy, která programátorovi téměř neumožňuje ovlivnit způsob ukládání dat v paměti a nutí programátora vytvářet nové objekty k vyřešení triviálních situací. Programátoři mohou bezstarostně vytvářet krátce či dlouze žijící objekty, aniž by se museli trápit se správou paměti těchto objektů, což může vést k neefektivnímu využití paměti.

V současnosti v Javě vznikají vysoce škálovatelné programy, které jsou příliš abstraktní a trpí zmíněným neefektivním využitím paměti. Tyto programy potřebují poměrně velké množství strojového času a operační paměti k dosažení relativně jednoduchých funkcionalit. Je časté, že instance serveru potřebuje několik gigabajtů operační paměti, aby byla schopna obsloužit několik stovek uživatelů, přestože celá aplikace má být schopna obsloužit uživatelů milióny. Populární aplikační rámce vytváří pro jeden webový požadavek stovky objektů a volají tisíce metod k načtení několika záznamů z databáze. Takovéto aplikace je třeba škálovat horizontálně, to znamená vytvořit více instancí těchto aplikací, aby byly schopny obsloužit co nejvíce uživatelů.

Diplomovou práci lze rozdělit na dvě části, kterými je analýza případů neefektivního využití paměti a rozšíření nástroje, který tyto případy umí detekovat. Nástroj, který se bude rozšiřovat, byl vytvořen Ing. Martinem Machem v rámci jeho diplomové práce. Hlavním rozšířením bude detekce duplicitních instancí, které se bude provádět hlubokým porovnáním. Duplicitní instance zbytečně zabírají paměť a stojí procesorový čas při jejich alokaci a dealokaci. Duplicita je častou příčinou plýtvání paměti. Příkladem je duplicita řetězců, kterou se Java snaží řešit fondem řetězců. Původní verze nástroje trpěla několika nedostatky, kvůli kterým nebyl nástroj použitelný v praxi. Jedním z cílů této práce je opravit tyto nedostatky. Funkčnost nástroje bude ověřena na několika reálných aplikací.

2 Struktura paměti

Manuální správa paměti umožňuje plné využití paměti a výkonu stroje, je ovšem také zdrojem širokého spektra chyb týkajících se bezpečnosti paměti. Programátoři jazyka Java jsou proto od této správy paměti odstíněni, o správu paměti se stará běhové prostředí jazyka.

2.1 Java Virtual Machine

Běhové prostředí jazyka Javy se nazývá Java Virtual Machine (JVM), je jedním ze základních pilířů jazyka [34]. Jedná se o komponentu, která umožňuje jazyku Java být nezávislá na hardwaru a na operačním systému, má relativně malou velikost kompilovaného kódu a chrání uživatele před škodlivými programy [33].

Program JVM je abstraktní výpočetní stroj, jako reálný stroj má instrukční sadu a během běhu je schopen manipulovat s pamětí. Implementace programovacího jazyka virtuálním strojem je běžnou praxí, příkladem může být předchůdce Javy programovací jazyk UCSD Pascal [10].

Virtuální stroj JVM o jazyce Java nic neví, rozumí pouze tzv. `class` souborů, což jsou soubory v konkrétním binárním formátu. V `class` souboru se nachází mezikód, tabulka symbolů a další doplňující informace. Virtuální stroj proto není vázaný na jazyk a jeho schopností může využít jakýkoliv jazyk, který dokáže svůj program v tomto formátu vyjádřit, příkladem jsou např. jazyky Kotlin a Scala.

2.1.1 Specifikace

Specifikace virtuálního stroje JVM má pouze několik požadavků. Jednou z nich je schopnost načítat soubory ve formátu `class` a následně vykonávat instrukce v nich zapsané. Dále definuje několik oblastí pamětí, které každý stroj musí mít. Implementační detaily jako jsou struktura jednotlivých oblastí paměti, její správa a optimalizace jsou však ponechány na vývojářích stroje. Nejznámější a nejpoužívanější implementací je virtuální stroj HotSpot vyvíjený firmou Oracle [30].

Běhové prostředí pracuje stejně jako jazyk Java s dvěma typy, jedná se o typy primitivní a referenční [45]. Pro oba typy existují odpovídající hodnoty, které lze ukládat do proměnných, předávat jako argumenty, vracet z metod nebo s nimi obecně pracovat. Virtuální stroj obsahuje podporu pro práci s objekty, objektem je dynamicky alokovaná instance třídy nebo pole. Reference na objekt má typ `reference`, hodnoty tohoto typu můžeme považovat za ukazatele na objekty. Práce s objekty vždy probíhá skrze hodnoty typu `reference`. Na jeden objekt může odkazovat více než jedna reference.

Primitivní typy

Virtuální stroj řadí mezi typy primitivní typy číselné, logické a speciální typ `returnAddress`. Číselné typy se dále dělí na čísla celá a čísla s plovoucí řádkou. Typy pro celá čísla jsou:

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

Čísla s plouvoucí řádkou lze reprezentovat pomocí dvou typů:

- `float`
- `double`

Všemi číselnými typy lze vyjádřit jak čísla kladná, tak čísla záporná. Neexistuje typ pouze pro kladná čísla jako je tomu např. v jazycích C a C++. Logické hodnoty jsou typu `boolean`.

Referenční typy

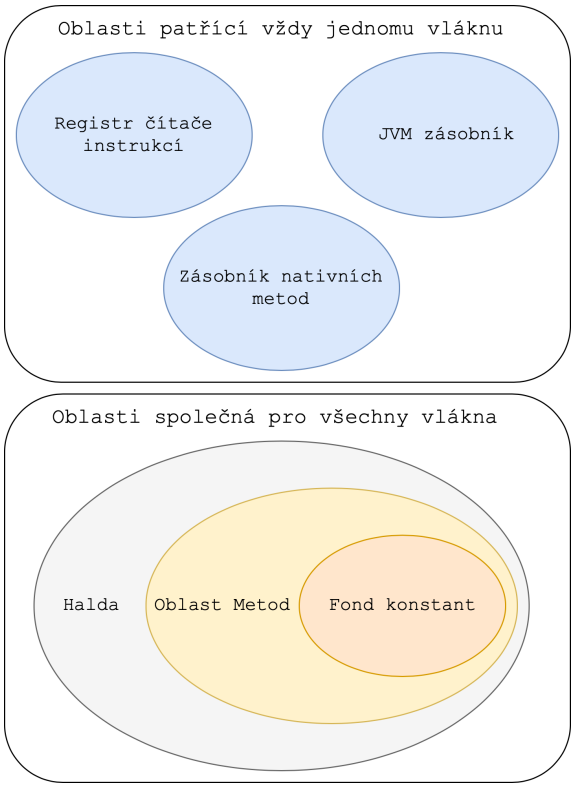
Hodnoty referenčních typů jsou odkazy na instance tříd, polí nebo na instance či pole implementující rozhraní. Existují tři typy referencí a to:

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

Speciální hodnota `null` je také typu `reference`, jedná se o odkaz na neexistující objekt a je výchozí hodnotou tohoto typu.

2.2 Oblasti paměti

Specifikace definuje několik datových oblastí, se kterými virtuální stroj během provádění programu pracuje [33]. Jedná se pouze o logické oblasti, proto oblast nemusí být souvislým blokem v paměti. Některé oblasti jsou vytvořeny při spuštění virtuálního stroje a existují do konce běhu. Ostatní datové oblasti jsou spojeny vždy s jedním vláknem, vytvářejí se při jeho vytvoření a na konci běhu daného vlákna se odstraní. Jednotlivé oblasti a jejich rozdělení jsou na obrázku 2.1.



Obrázek 2.1: Oblasti paměti a jejich rozdělení.

2.2.1 Registr čítače instrukcí

Virtuální stroj JVM má podporu vykonávání více vláken najednou [33]. Každé vlákno má jeden registr čítače instrukcí ve kterém se nachází instrukce, kterou vlákno v danou chvíli provádí. V případě, že vlákno vykonává kód nativní metody, je obsah registru nedefinovaný.

2.2.2 JVM zásobník

Každému vláknu se při vytvoření založí Java Virtual Machine zásobník, který ukládá rámce [33]. Zásobník se používá k uložení lokálních proměnných, částečných výsledků a používá se při volání metod, je tedy analogický zásobníku z jazyka C. Se zásobníkem se pracuje pouze pomocí přidáváním a odebíráním ramců, rámce být alokovány na haldě.

Velikost zásobníku může být pevně daná nebo proměnlivá. Některé implementace dovolují uživateli nastavit počáteční, maximální a minimální velikosti zásobníku. Potřebuje-li výpočet vlákna velikost přesahující maximální velikost zásobníku, virtuální stroj vyhazuje notoricky známou výjimku `StackOverflowError`. V případě, že při vytváření vlákna není dostatek paměti pro vytvoření jeho zásobníku, virtuální stroj vyhazuje výjimku `OutOfMemoryError`.

2.2.3 Zásobník nativních metod

Virtuální stroj může implementovat klasické zásobníky, obecně známé také jako „C“ zásobníky, aby poskytla podporu nativním metodám [33]. Nativní metody jsou napsány v jiném jazyce a jsou zkompilované pro určitý hardware, případně operační systém. Specifikace zásobník nativních metod ale nevyžaduje, proto implementace stroje, které nativní metody neumí načíst a klasický zásobník k běhu nepotřebují, nejsou povinné tento zásobník poskytovat.

2.2.4 Halda

Halda je sdílena všemi vlákny JVM a je vytvořena na začátku běhu virtuálního stroje [33]. Z této oblasti se alokuje paměť pro všechny pole a instance tříd. Jedná se o logickou oblast, proto se nemusí jednat o souvislý blok paměti.

Paměť alokována pro objekty a pole je uvolněna pomocí automatické správy paměti, která je známá pod názvem garbage collection. Specifikace nepředpokládá žádný typ automatické správy, algoritmy uvolnění paměti jsou ponechány na implementaci virtuálního stroje.

2.2.5 Oblast metod

Oblast metod je součástí haldy, proto je také sdílená všemi vlákny JVM [33]. Jedná se o oblast podobné segmentu paměti, kde se uchovává kód programu.

V této oblasti se ukládají struktury pro každou načtenou třídu, jako jsou atributy, data a kód metod a vše týkající se inicializace objektů a tříd.

2.2.6 Fond konstant

Každá načtená třída má vlastní fond konstant, což je obdoba tabulky symbolů [33]. Ve fondu se nachází různé druhy konstant počínaje číselnými literály, odkazy na metody a atributy konče. Jednotlivé fondy konstant načtených tříd se alokují v oblasti metod. Fond konstant obsahuje reference na:

- třídu či rozhraní,
- atribut třídy,
- metodu třídy,
- metodu rozhraní.

Dále fond obsahuje konstanty literálů pro:

- řetězce,
- číselné konstanty.

Všechny reference a konstanty se nacházejí ve strukturách, která mají jména složená z prefixu `CONSTANT`, sufixu `info` a z kořene popisující, co struktura obsahuje. Příkladem je struktura nazvaná `CONSTANT_Integer_info` ukládající konstanty čísel typu `int`.

2.3 Správa paměti

Běhové prostředí spravuje paměť pomocí automatické správy paměti [33]. Virtuální stroj si sám od operačního systému žádá o další paměť, kterou poté sám správně uvolní. Programátoři, kteří vyvíjejí kód cílící JVM, se proto nemusí starat o manuální alokaci a uvolnění paměti.

2.3.1 Kolektory

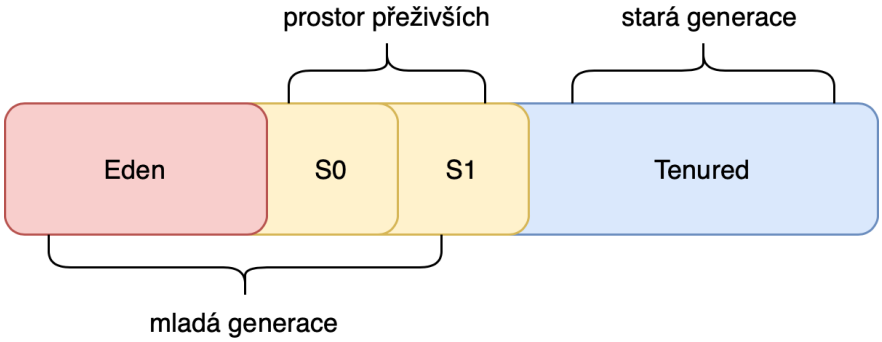
Garbage collection, volně přeloženo sběr odpadu, je proces uvolnění paměti odstraněním objektů, které se již dále nepoužívají a jsou programem nedosažitelné [1]. Proces, který tento sběr provádí, se nazývá kolektor. Název procesu naznačuje, že kolektor aktivně hledá nereferencované objekty pro

sběr, opak je však pravdou, kolektory obvykle hledají stále referencované objekty a zbytek považuje jako nedosažitelný. Jedná se o automatický proces, není třeba značit objekty, které chceme z paměti uvolnit. Procesu, který provádí sběr odpadu, se říká garbage collector a jeho implementace je ponechána na autorech virtuálního stroje. Dále popsané algoritmy a technické detaily se budou týkat implementace HotSpot.

Implementace HotSpot nabízí několik přístupů sběru odpadu, které jsou optimalizované pro různé případy použití. Společnou vlastností obou přístupů je, že se sběr provádí ve dvou krocích. Kolektor v prvním kroku prochází graf dosažitelných objektů a jsou označeny jako živé, nedosažitelné objekty jsou považovány za mrtvé. Druhý krok je tzv. *zametací*, kdy dochází k uvolnění mrtvých objektů [4]. Nepovinným krokem je defragmentace paměti, která usnadní následnou alokaci paměti. Některé kolektory musí před provedením sběru zastavit běžící program, procesu zastavení se říká *stop-the-world*.

2.3.2 Generační princip

Generační hypotéza tvrdí, že velké množství objektů žije velmi krátkou dobu [44]. Všechny kolektory stroje HotSpot tohoto tvrzení využívají pro snížení režie. Paměť je rozdělena na několik částí, minimálně na dvě části, které jsou označovány jako mladá a stará generace [43]. Část paměti vyhrazena pro mladou generaci bývá obvykle menší než část určena pro starou generaci. Rozdělení paměti do generací je zobrazeno na obrázku 2.2.



Obrázek 2.2: Rozdělení objektů do generací.

Využití generační hypotézy spočívá v tom, že se úklid paměti mladé generace provádí častěji. Každý úklid paměti zvedá objektu věk, který je uložený v jeho hlavičce. Pokud objekt přežije a dosáhne určitého věku, je přesunut do staré generace.

Objekty se vytváří v části nazvané *eden*. Při naplnění této paměti se provede úklid paměti a dosažitelné objekty se přesunou do prostoru přeživších. Po několika cyklech běhu kolektoru se přesunou dosažitelné objekty do části *tenured*.

2.3.3 HotSpot kolektory

V této sekci budou stručně představeny některé kolektory, které HotSpot nabízí. Programově nelze donutit kolektor, aby provedl sběr. Kolektor lze požádat, aby úklid paměti provedl co nejdříve zavoláním metody `System.gc()`, to však kolektor může ignorovat a naplánovat si sběr na později.

SerialGC

Jedná se o nejjednodušší a historicky nejstarší implementaci generačního kolektoru. Během sběru jsou zastavena všechna vlákna aplikace, není proto vhodný pro vícevláknové aplikace. Je vhodný pro situace, kdy delší zastavení aplikace při *stop-the-world* nejsou problémem. Použití lze vynutit přepínačem `-XX:+UseSerialGC`.

ParallelGC

Dalším logickým krokem byl přechod k vícevláknovému kolektoru, kterým je `ParallelGC`, v literatuře označován jako kolektor s vysokou propustností. Podobně jako `SerialGC` se jedná o generační kolektor a při běhu zastavuje všechna vlákna. Hlavním rozdílem je tedy vícevláknový běh sběru, proto je vhodný pro aplikace většího rozsahu. Dlouhou dobu byl výchozím kolektorem pro HotSpot a to až do Javy 8, v následujících verzích se dá použít pomocí přepínače `-XX:+UseParallelGC`.

Conurrent Mark And Sweep (CMS)

Kolektor CMS se pokouší zkrátit dobu, kdy je aplikace zastavena, hledáním a označováním živých objektů při běhu aplikace [28]. Je určen pro aplikace, které upřednostňují kratší pauzy a mohou si dovolit sdílet prostředky s kolektorem. Aktivujeme přepínačem `-XX:+UseConcMarkSweepGC`.

Garbage First (G1)

Garbage First kolektor vznikl jako nástupce CMS kolektoru, který na rozdíl od předchozích regionů používá zcela nové rozdělné paměti. Halda je rozdělena do většího množství stejně velkých regionů, díky kterým nemusí kolek-

tor provádět sběr nad celou generací. Každému regionu je přiřazena priorita podle odhadovaného množství odpadu, tento odpad se provádí pomocí heurestiky. Kolektor se zaměřuje nejdříve na regiony s vysokou prioritou, odtud pochází pojmenování Garbage First. Od Javy 9 se stal výchozím kolektorem, přítomen byl ovšem od verze Java 7 a lze ho použít pomocí přepínače `-XX:+UseG1GC`.

3 Neefektivní využití paměti

Jazyk Java odstiňuje programátory od manuální správy paměti, ti si díky tomu nemusí lámat hlavu nad problémy spojené s alokací paměti a s jejím následném uvolnění, přesto pro ně může být obtížné efektivně paměť využívat.

3.1 Řetězce

Řetězce jsou v Javě reprezentovány třemi třídami `String` pro neměnné řetězce, `StringBuffer` a `StringBuilder` pro řetězce, které lze měnit [11]. Nejčastěji vytvářenými objekty během provádění Java programu jsou objekty třídy `String` [18]. Existují dva typy paměťové neefektivity, které se týkají řetězců. Prvním z nich je velké množství instancí třídy `String`, které mají stejnou hodnotu. Tyto hodnoty jsou neměnné, proto se jedná o zbytečnou duplicitu. Druhým je velké množství řetězcových literálů, jejichž hodnotu aplikace ve skutečnosti nepoužije. Příkladem jsou chybové hlášky, které jsou vytvořeny během inicializce třídy obsluhy chyb, velká část z nich však není vůbec během běhu programu použita.

3.1.1 Fond řetězců

Fond řetězců (angl. string pool) je místo na haldě, kam se ukládají tzv. *internované řetězce*. Jedná se o příklad návrhového vzoru muší váha (angl. flyweight pattern). Tento návrhový vzor umožňuje optimalizaci použití paměti sdílením co největšího množství dat mezi objekty [14]. Existuje-li řetězec ve fondu řetězců, běhové prostředí může vrátit referenci na řetězec ve fondu, namísto toho, aby vytvářel řetězec nový. Tímto přístupem lze za cenu výpočetního výkonu navíc snížit počet duplicitních řetězců v paměti. V případě vytvoření řetězce pomocí klíčového slova `new` se vytvoří nový řetězec a fond řetězců se nepoužije.

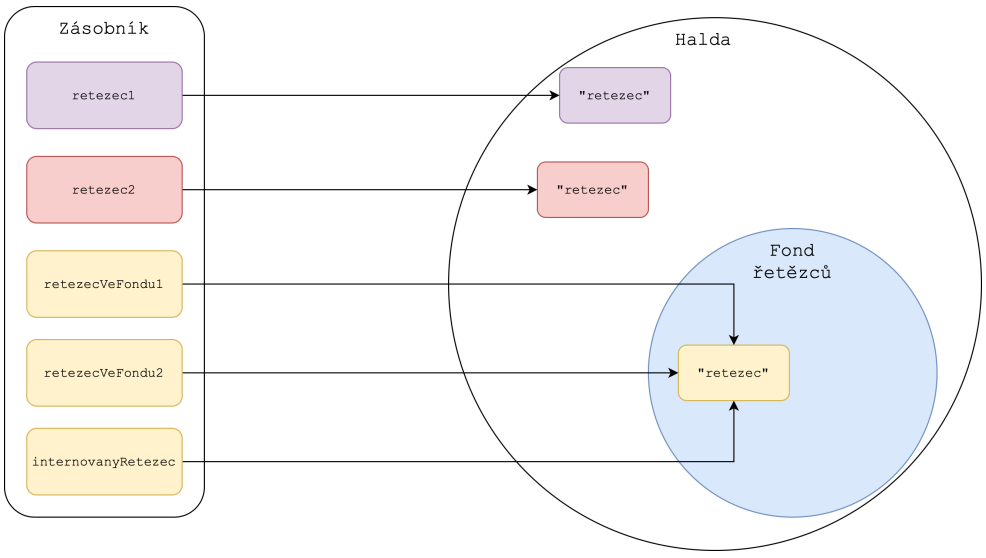
Různé způsoby vytváření řetězců jsou ukázány ve fragmentu kódu 3.1. První dva řetězce jsou vytvořeny pomocí operátoru `new`, ten při každém použití vytvoří novou instanci. Porovnání vytvořených instancí (porovnání č. 1) ukazuje, že jsme vytvořili dvě různé instance s duplicitním obsahem. Vytvoření dalších dvou řetězců je za pomoci řetězcových literálů. Nejsou vytvořeny žádné nové instance, ale pouze se vrátí reference na řetězec z fondu. Druhé

```
String retezec1 = new String("retezec");
String retezec2 = new String("retezec");

assertNotSame(retezec1, retezec2); // (1)

String retezecVeFondu1 = "retezec";
String retezecVeFondu2 = "retezec";

assertSame(retezecVeFondu1, retezecVeFondu2); // (2)

String internovanyRetezec = retezec1.intern();
assertSame(retezecVeFondu2, internovanyRetezec); // (3)
```

Fragment kódu 3.1: Vytváření řetězců.

porovnání demonstruje, že se v obou případech vrátí ta samá reference. Třída `String` nabízí metodu `intern`, pomocí které lze manuálně vyžádat použití řetězce z fondu. Metoda se pokusí najít ekvivalentní řetězec ve fondu, v případě neúspěchu vloží tento řetězec do fondu.

Na obrázku 3.1 je zobrazen stav paměti po provedení fragmentu kódu 3.1. Ná zásobníku se nachází reference a na haldě samotné objekty řetězců. Reference mají stejnou barvu, pokud si jsou ekvivalentní, tj. porovnání pomocí operátoru `==` vrací hodnotu `true`.



Obrázek 3.1: Řetězce v paměti a jejich reference na zásobníku.

3.2 Duplicitní objekty

Problém duplicitních řetězců můžeme zobecnit na duplicitu objektů. Na haldě se může vyskytovat velké množství objektů, které nejsou řetězce a mají duplicitní obsah. Pokud se jedná o neměnné objekty, tak zbytečně zabírají paměť navíc a zatěžují Garbage collection (GC), jelikož by v paměti mohly být pouze jednou.

Příkladem duplicity může být například monitorovací systém, který přijímá periodicky zprávy z monitorovaných entit [8]. Zpráv jsou reprezentovány časovou značkou a hodnotou. V případě, že se ve stejný moment vygeneruje několik zpráv se stejnou hodnotou (např. hodnota může být pouze 0 nebo 1 signalizující stav entity), je vytvořeno několik duplicitních objektů.

Narozdíl od řetězců, kde se používá výše zmíněný fond řetězců, však neexistuje nativní podpora pro deduplikaci obecných objektů, výjimkou jsou některé objekty obalující primitivní datové typy [35]. Populární knihovna `Guava` od společnosti Google nabízí rozhraní `Interner` [13], která poskytuje ekvivalentní chování jako fond řetězců ostatním neměnným typům. Implementace rozhraní interně uchovává odlišné objekty v datové struktuře implementující rozhraní `AbstractMap` ze standardní knihovny [12]. Chceme-li ušetřit místo v paměti, vybíráme objekty z této datové struktury. V případě, že se objekt ve struktuře nenachází, ho tam vloží, jinak vrací referenci na uložený objekt. Tím zajistíme, že se vždy používá jeden a ten samý objekt a v paměti se nenachází zbytečné duplicity.

Postup deduplikace pomocí třídy `Interns` ale není bezchybný. Může se stát, že při nízkém počtu duplicit velikost datové struktury převažuje celkovou velikost duplicit. V tom případě by se místo neušetřilo, ale ztratilo.

3.3 Faktor neefektivnosti

Neefektivnímu využití paměti se říká `memory bloat` [23]. Faktor neefektivnosti definujeme tímto vztahem:

$$faktor = (t - d)/t,$$

kde t je celkový počet bytů všech objektů a d je počet bytů skutečných dat programu. Čitatel je počet bytů použitých na režii jako jsou hlavičky přidané virtuálním strojem (JVM) nebo kolekcemi. Faktor je číslo v intervalu $(0, 1)$, vyšší číslo znamená vyšší nefeektivitu práce s pamětí.

3.3.1 Kategorizace bajtů

Jednotlivé bajty lze rozdělit do čtyř kategorií, které jsou uvedeny v tabulce 3.1, dle jejich účelu [24]. První kategorií jsou primitivní atributy nebo primitivní prvky polí, které v Javě zabírají jeden, dva, čtyři nebo osm bajtů. Druhá kategorie představuje data, které běhové prostředí používá pro správu jednotlivých objektů, říká se jim objektové hlavičky. Běhové prostředí je používá především k usnadnění práce během garbage collection, synchronizace nebo reflexe. Velikost hlavičky obvykle nezávisí na typu instance, většina virtulních strojů vytváří objektové hlavičky o velikosti 12 bajtů, jedná-li se o 32-bitové adresování, nebo 20 bajtů v případě 64-bitového adresování [40]. Hlavička instance objektu se liší hlavičky instance pole, která má navíc 1 - 4 bajty k uložení velikosti pole.

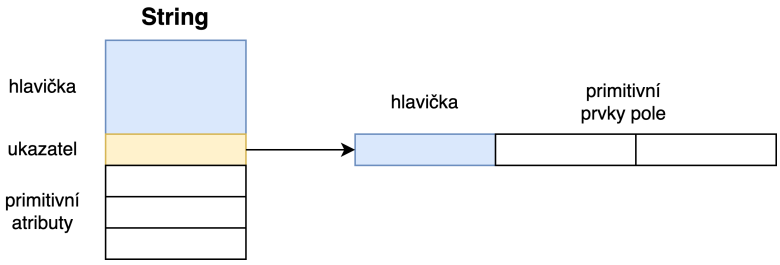
Poslední dvě kategorie popisují bajty vyčleněné pro ukazatele mezi objekty. Každý ukazatel má velikost šířku procesorového slova, typicky 32-bitů nebo 64-bitů. Stejnou velikost zabírá i ukazatel s hodnotou null. Třetí kategorie jsou bajty ukazatele na existující objekt, čtvrtá kategorie je případ null hodnoty.

Kategorie	Popis	Počet bajtů
primitivní	prim. atributy, prim. prvky polí	1 - 8
hlavička	hlavička vytvořená JVM	12 - 20
ukazatel	ukazatele mezi objekty	4
null	ukazatel s hodnotou null	4

Tabulka 3.1: Kategorie typů dat pro 32-bitové virtuální stroje.

3.4 Neefektivita u instancí

Na obrázku 3.2 je ukázána instance dvouznakového řetězce. Na 32-bitové platformě se zarovnáním na 8 bajtů zabírá tento řetězec 32 bajtů. Rozdělení bajtů do kategorií a jejich velikostí ukazuje tabulka 3.2. Řetězec obsahuje dva znaky, každý znak zabírá 2 bajty, pouhé 4 bajty z toho jsou opravdová data. Faktor neefektivnosti této instance je 0.875.



Obrázek 3.2: Řetězec reprezentován instancí třídy `String`.

primitivní	hlavička	ukazatele	null	**celkem**
16	12	4	0	32

Tabulka 3.2: Kategorizace bajtů instance a jejich počet.

3.5 Neefektivita u kolekcí

Standardní kolekce jazyka jsou nedílnou součástí každého Java programu. Vývojářové standardní knihovny vynaložili velké úsilí, aby kolekce byly paměťově efektivní, přesto jsou častým zdrojem plýtváním paměti.

3.5.1 Paměťová režie

Specializované kolekce obvykle potřebují více paměti, aby specializované chování mohli implementovat. Na obrázku 3.3 vidíme, že kolekce `ArrayList` je méně specializovaná než `HashMap`, a proto má menší paměťovou režii.

Obrázek 3.3: Více specializované kolekce potřebují více paměti.

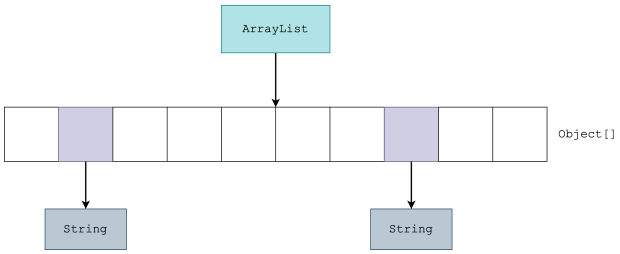
Kolekce `HashMap` umožňuje ukládání dvojic klíč hodnota. Vložení i nalezení těchto hodnot má složitost $O(1)$, tedy konstantní. Toto chování ovšem něco stojí, každý uložený prvek je reprezentován objektem `HashMap$Entry`, který v paměti zabírá 36 bajtů, oproti tomu `ArrayList` potřebuje pouhé 4 bajty k uložení jednoho prvku pomocí prosté reference [3]. V tabulce 3.3 jsou velikosti paměťové režie jednotlivých kolekcí pro jeden a tisíc prvků.

Kolekce	1 prvek	1000 prvků
HashSet	36	36 000
HashMap	36	36 000
LinkedList	24	24 000
ArrayList	4	4 000

Tabulka 3.3: Paměťová režie jednotlivých kolekcí.

3.5.2 Prázdné a řídké kolekce

Běžnou praxí vytváření prázdné kolekce `ArrayList` je pomocí jeho výchozího konstruktoru, tj. konstruktoru bez parametrů. Ten však ve verzích předcházejících verzi Java SE 8 při vytváření objektu inicializoval interní pole o velikosti deseti prvků [25]. Na první pohled se může tato malá neefektivta zdát nepodstatná. Analýza vývojářů knihovny ovšem ukázala, že téměř 85% instancí objektu `ArrayList` je vytvořeno tímto způsobem a plýtvání paměti tímto způsobem je znatelné a je nutné opravit [9]. Řešením byla inicializce pole až při přidání prvního prvku do kolekce [26].



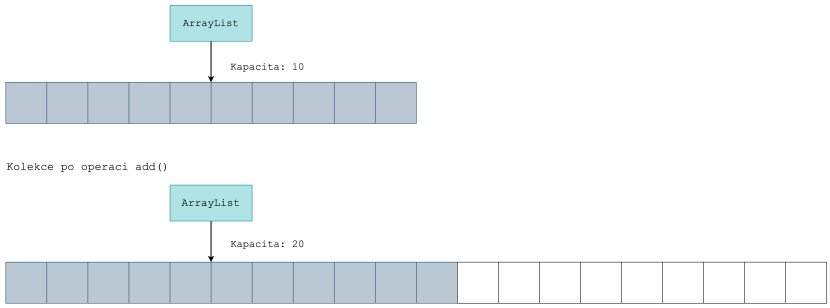
Obrázek 3.4: Řídce obsazené pole s vysokou počáteční kapacitou.

Problematické jsou i kolekce, které jsou řídce obsazeny. Existuje několik příčin, které vedou k tomuto problému. Prvním z nich je příliš vysoká počáteční kapacita kolekce. Příkladem je obrázek 3.4, kde je znázorněna kolekce

`ArrayList` s počáteční kapacitou pro deset prvků, obsahuje však pouze dva objekty třídy `String`, zbylých osm míst jsou `null` reference [7].

Další příčinou je příliš agresivní zvětšování. Kolekce musí v momentě naplnění zvětšit svou kapacitu, to znamená alokovat větší úložiště a přesunout do něho své prvky. Velikost nového úložiště je dána faktorem zvětšení. V případě, že je faktor vysoký, může být alokováno zbytečně mnoho místa, které se nevyužije, v opačném případě může ovšem docházet k opětovnému alokování paměti, jelikož se při zvětšení alokuje příliš málo místa.

Případ zvětšení ilustruje obrázek 3.5[1]. Kolekce měla pro svých deset prvků alokováno pole o velikosti deset, po přidání jednoho prvku se však pole dvakrát zvětšilo. Na první pohled je patrné, že po zvětšení je využití paměti velice neefektivní.



Obrázek 3.5: Zvětšení kolekce `ArrayList` při nedostatku místa.

Poslední příčinou řídkých kolekcí je absence redukce kolekce při odstraňování jeho prvků. Na závěr jsou v tabulce uvedeny výchozí velikosti a faktor zvětšení jednotlivých kolekcí [3].

Kolekce	Výchozí velikost	Faktor zvětšení
HashSet	16	$\times 2$
HashMap	16	$\times 2$
LinkedList	1	$+1$
ArrayList	10	$\times 1,5$

Tabulka 3.4: Výchozí velikosti a faktory zvětšení jednotlivých kolekcí.

[1] Obvykle se kolekce zvětšuje při dosažení určitého poměru naplnění, tzv. *fill ratio*.

3.5.3 Kolekce primitivních typů

Standardní knihovna jazyka Java, narozdíl např. od C++, neobsahuje podporu pro kolekce primitivných typů. Pro uložení primitivních datových typů je třeba obalení do pomocných objektů, které standardní knihovna už nabízí. Ukládání těchto typů do kolekcí je proto paměťově velmi neefektivní.

Neefektivnost může být demonstrována na ukládání primitivního datového typu `int`, který zabírá 4 bajty paměti. Jeho objektový protějšek `Integer` potřebuje čtyřikrát více, tedy 16 bajtů paměti. Použití kolekce pro uložení čísel typu `int` potřebuje tedy čtyřikrát více paměti než kdybychom použili primitivní pole.

3.6 Krátká životnost objektů

Počet krátce žijících objektů rapidně roste a s ním klesá výkonnost [23]. Neustálý vývoj a zlepšování automatické správy paměti může programátory vést k mylné představě, že vytvoření a následné uvolnění paměti po krátce žijících objektech je prakticky zadarmo. Lepší algoritmy GC opravdu výrazně snižují čas alokace a uvolnění paměti, jednoduché dočasné objekty typu `Integer` proto nejsou problematické, je-li počet jejich alokací v rozumné míře. Mnoho webových aplikací však vytvoří několik tisíců krátce žijích objektů pro jedno volání, což může vést k zahlcení L1 a L2 cachí.

Běžnou praxí je inicializce všech atributů objektu bez ohledu na jeho následné použití, např. autoři knihoven nemohou předpovídat, co všechno jejich uživatelé budou potřebovat, je proto pro ně snažší incializovat vše. Tento přístup může mít kaskádový efekt pokud se jednotlivé podstruktury inicializují stejným způsobem.

Dočasné datové struktury mohou zahrnovat mnoho jiných objektů, inicializace takových datový struktur může být komplexní a mít značný dopad na celkový výkon. Fragment kódu 3.2 je příkladem komplexní inicializace, jelikož vytvoření instance `SimpleDateFormat` stojí 123 volání metod a vytvoření 44 nových objektů. Tento objekt byl navrhnut tak, aby měl dlouhou životnost, proto jeho znovupoužití vyrovná cenu drahé inicializace.

```
SimpleDateFormat sdf = new SimpleDateFormat();
```

Fragment kódu 3.2: Vytvoření instance třídy `SimpleDateFormat`.

Vytváření krátce žijících objektů může být uživateli skryto. Pokud bychom chtěli napsat metodu, která určí, jestli se jedná o římské číslo [5], možnou

implementací je využití regulárního výrazu. Implementace je ukázana ve fragmentu kódu 3.3.

```
static boolean isRomanNumeral(String s) {
  return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
    + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

Fragment kódu 3.3: Naivní implementace určení římského čísla.

Problém s touto naivní implementací je metoda `matches` třídy `String`. Při každém volání této metody je voláno drahé vytvoření objektu třídy `Pattern` reprezentující konečný automat, který je však po jediném použití zahozen.

Řešením je vytvořit dlouho žijící objekt konečného automatu pro rozpoznání římského čísla, uložení a opětovného použití při určování římského čísla. Efektivní implementace je uvedena ve fragmentu kódu 3.4. Tento způsob je mnohonásobně rychlejší a navíc neplýtvá čas alokací paměti a jejím následném uvolňování.

```
public class RomanNumerals {
  private static final Pattern ROMAN = Pattern.compile(
  "^(?=.)M*(C[MD]|D?C{0,3})"
  + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
  static boolean isRomanNumeral(String s) {
    return ROMAN.matcher(s).matches();
  }
}
```

Fragment kódu 3.4: Efektivní implementace určení římského čísla.

Standardní knihovna obsahuje objekty obalující primitivní datové typy pro práci s kolekcemi. Konverze mezi obalujícím typem a primitivním je automatická, zabalování a rozbalování se říká autoboxing, respektive unboxing. Tato konverze má programátorovi usnadnit práci při práci s obalujícím typem a primitivem, ale může být skrytým zdrojem velmi neefektivní práce s pamětí.

```
public static void implicitAutoboxing() {
  Long sum = 0L;

  for (long i = 0; i < 1000; i++) {
    sum += i;
  }
}
```

Fragment kódu 3.5: Implicitní autoboxing.

Fragment kódu 3.5 na první pohled vypadá nevinně, pouze v cyklu sečte čísla od 0 do 999. Problém zjistíme v momentě, že analyzujeme, co se děje v

paměti během provádění cyklu. Neustále se vytváří nové objekty typu `Long`, které žijí pouze jeden cyklus a poté se zahodí. Tento fragment kódu je nejen paměťově velmi nefeektivní, ale i pomalý, jelikož zbytečně alokuje a dealokuje paměť na haldě.

Zbytečně krátce žijící objekty jsou dalším příkladem neefektivní práce s pamětí, jelikož je pro ně nutné paměť alokovat a poté je vzápětí uvolnit, výsledkem je zatížení GC a snižení výkonu aplikace.

4 Únik paměti

Nárůst potřebné paměti pro běh programu velmi často souvisí s únikem paměti. Uniklou paměť nelze uvolnit. Nárůst uniklé paměti zvětšuje paměťovou stopu programu, ta má dopad na celkovou výkonnost systému [21].

Únik paměti si většina lidí spojí s jazyky neobsahující automatickou správu paměti, jako jsou jazyky C a C++. Proti paměťovým únikům však nejsou imunní ani jazyky vybavené automatickou správou paměti jako je Java.

4.1 Statické proměnné

```
class MemoryLeak {
    static List<Double> leakedList = new ArrayList<>();
    static void fillLeakedList() {
        for (int i = 0; i < 20_000; i++) {
            leakedList.add((double) i);
        }
    }

    static void doSomethingElse() {
        while (true) {}
    }

    public static void main(String[] args) {
        fillLeakedList();
        doSomethingElse();
    }
}
```

Fragment kódu 4.1: Únik paměti v Javě.

Fragment kódu 4.1 ilustruje příklad programu obsahující únik paměti. Jednoduchá třída `MemoryLeak` obsahuje statický seznam čísel, které se naplní v metodě `fillLeakedList`. Přestože seznam už není dále během života programu využíván, je stále dosažitelný a proto nemůže být uvolněn. Rozsah platnosti proměnných by proto měl být minimální, případně po použití nastavit referenci na hodnotu `null`, tím garbage collectoru dovolíme paměť uvolnit.

4.2 Neuzavřené zdroje

Běhové prostředí musí pro každý otevřený soubor nebo připojení alokovat pamět a tyto zdroje spravovat. Neuzavření zdroje tedy také vede k úniku paměti a je to poměrně běžný prohřešek, kterého se vývojářové dopouštějí. Přístup `try-with-resources` [36] řeší tento problém, nelze ho ovšem vynutit, správné uzavření je tedy stále povinností programátora.

4.3 Implementace `equals` a `hashCode`

Kolekce `HashMap` interně používá metody `equals` a `hashCode` k správě objektů, které uchovává. Neefektivní hashování pomocí `hashCode` může vést k degradaci výkonu [29]. Výsledkem nevhodné implementace `equals` může být pouze uživatelem neočekáváné chování nebo dokonce únik paměti, který je ukázán na fragmentu kódu 4.2.

```
public class Person {
  String name;
  Person(String name) {
    this.name = name;
  }

  @Test
  public void memoryLeakedHashMap() {
        Map<Person, Integer> leakedMap = new HashMap<>();
        final long size = 100;
        final String name = "student";
        for (int i = 0; i < size; i++) {
            leakedMap.put(new Person(name), 0);
        }

        Assert.assertNotEquals(1, leakedMap.size());
        Assert.assertEquals(size, leakedMap.size());
        Assert.assertNull(leakedMap.get(new Person(name)));
  }
}
```

Fragment kódu 4.2: Únik paměti z `HashMap`.

Fragment demonstruje případ nepřekrytí metod `equals` a `hashCode`. Do vytvořené instance `HashMap` se vloží sto záznamů s hodnotou nula a s klíčem instance `Person`. Klíče jsou identické, všechny mají atribut `name` inicializován na hodnotu ‘student‘. Uživatel kolekce by se mohl mylně domnívat, že instance `leakedMap` bude obsahovat pouze jeden prvek, velikost instance je stejná počtu vložených hodnot, resp. klíčů do ní. Na závěr je ukázána ne-

možnost získání vložené hodnoty pomocí klíče. Vložené hodnoty nemohou být uvolněny automatickou správou paměti, ale není možné se k nim dostat pomocí standardní metody `get`[1], je to proto jeden z případů úniku paměti v Javě. Paměť lze uvolnit manuálně zavoláním metody `clear`, která smaže vložené hodnoty a klíče.

Správnou implementací základních metod `equals` a `hashCode` můžeme předejít podobným incidentům.

4.4 Vnořené třídy

Vnořené třídy jsou dalším možným zdrojem úniku paměti. Instance vnořené třídy vždy potřebují k inicializaci instanci vnější třídy, po inicializaci mají implicitní referenci na vnější instanci. Rozsah platnosti instancí vnořených tříd by nikdy neměl být větší než jejich vnějších instancí, porušením tohoto pravidla dochází k zmíněnému úniku paměti. Ukázku úniku paměti pomocí vnořené instance je ve fragmentu kódu 4.3.

```
class LeakFactory {
    Leak createLeak() {
        return new Leak();
    }

    class Leak {}

    public static void main(String[] args) {
        createLeaks();
    }

    public static Leak[] createLeaks() {
        final int size = 100;
        Leak[] leaks = new Leak[size];
        LeakFactory factory = new LeakFactory();

        for (int i = 0; i < size; i++) {
            leaks[i] = factory.createLeak();
        }

        return leaks;
    }
}
```

Fragment kódu 4.3: Únik paměti pomocí vnořené instance.

[1] K hodnotám se dá stále dostat pomocí metody `values` či `entrySet`, kterou jsou však určeny k iteraci nad kolekcí.

Metoda `createLeaks` vytvoří vnější instanci třídy `LeakFactory`, pomocí které se potom vytváří vnořené instance třídy `Leak`. Vnější instance na konci metody zaniká, přesto nemůže být uvolněna z paměti, protože na ni mají implicitní referenci instance vnořené třídy.

Jestliže vnořená třída nepotřebuje přístup do vnější třídy, vnořená třída by měla být statická, tím se vyřeší únik paměti v ukázce.

4.5 Metoda `finalize`

Třídy překrývající metodu `finalize` mohou být dalším zdrojem úniku paměti [2]. Objekty těchto tříd nemohou být okamžitě uvolněny z paměti, ale jsou zařazeny do fronty objektů, které ještě finalizaci neprovedly. Provedení finalizace je naplánovaná a proběhne později.

Problém nastává v momentě, kdy finalizace objektu trvá moc dlouho a GC přidává objekty do fronty rychleji než se fronta vyprazdňuje. Může se potom stát, že aplikaci dojde místo a vyhodí notoricky známou výjimku `OutOfMemoryError`.

4.6 `ThreadLocal`

Většina výše zmíněných příkladů nejsou klasickými úniky paměti ve smyslu, že paměť již nelze v průběhu programu uvolnit a paměť je opravdu ztracena. Tento typ úniku paměti se děje v jazycích jako jsou C nebo C++, ve kterých při alokaci paměti získáme její adresu, tu je třeba si uložit do ukazatele. K úniku dochází v případě, že o adresu alokované paměti v ukazateli přijdeme aniž by byla paměť uvolněna, např. z důvodu konce platnosti ukazatele nebo změnou jeho hodnoty.

Popsaný typ úniku lze vytvořit i v Javě pomocí objektu `ThreadLocal`. Vlákna si mohou pomocí tohoto objektu ukládat data, ke kterým má přístup pouze vlákno, které je uložilo. Každé vlákno má implicitní datovou strukturu, kam si ukládá dvojici tvořenou instancí `ThreadLocal` a datovým objektem. Uložená dvojice žije buď stejně dlouho jako instance `ThreadLocal`, pomocí které se uložila, anebo do konce života vlákna. Manuálně se může dvojice odstranit pomocí metody `remove`.

Problém nastává v momentě, kdy si do zmíněné datové struktury instance ukládá referenci na sebe samotnou. Příklad tohoto problému je ukázán ve

fragmentu 4.4, kde se v cyklu vytváří instance `MemoryLeak`. Každá instance `MemoryLeak` si pomocí atributu `threadLocal` uloží silnou referenci na sebe samotnou. Životnost `threadLocal` je však svázaná s instancí `MemoryLeak`, proto oba objekty budou žít až do konce běhu vlákna.

Výsledek fragmentu 4.4 po nějaké chvíli vyvolá výjimku `OutOfMemory`, jelikož se objekty `MemoryLeak` v paměti hromadí a GC je není schopen uvolnit. V případě, že třída má vlastní zavaděč (tzv. `class loader`), z paměti nelze uvolnit ani tento zavaděč. Vytvořené objekty by se uvolnily až s koncem vlákna.

```
public class TrueMemoryLeak implements Runnable {
    public static void main(String[] args) throws
   InterruptedException {
        Thread thread = new Thread(new TrueMemoryLeak());
        thread.start();
        thread.join();
    }

    @Override
    public void run() {
        while (true) {
            createMemoryLeak();
        }
    }

    // vytvorena instance ma scope az do konce behu vlakna
    // prestoze se nikam neuklada a obvykle by mela byt
   okamzite uvolnena
    public void createMemoryLeak() {
        new MemoryLeak();
    }

    static class MemoryLeak {
        private final ThreadLocal<MemoryLeak> threadLocal =
   new ThreadLocal<>();
        MemoryLeak() {
        // tato radka je pricina memory leaku
        // nelze uvolnit ani class loader teto tridy
            threadLocal.set(this);
        }
    }
}
```

Fragment kódu 4.4: Klasický únik paměti pomocí `ThreadLocal`.

Únik paměti z výše uvedeného příkladu se může zdát málo pravděpodobný, opak je však pravdou. Webové servery využívají fond vláken, aby se vy-

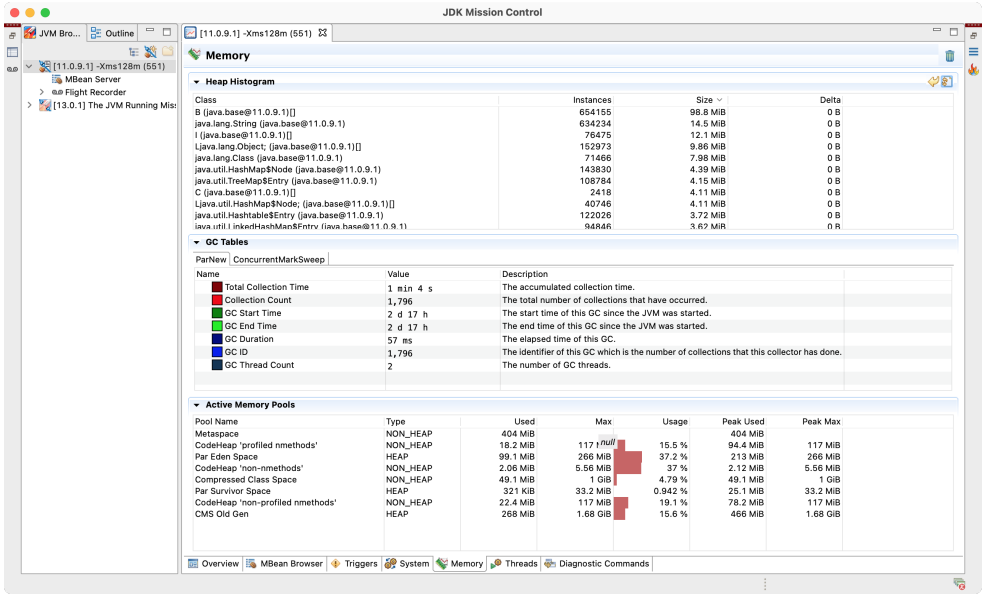
hnuly opakovanému vytváření vláken pro každý příchozí požadavek. Vlákna z fondu žijí po celou dobu běhu serveru. Nesprávným použitím `ThreadLocal` můžeme tedy vytvořit situaci, kdy objekty žijí po celou dobu běhu serveru a nelze je uvolnit. Úniky vedou k vynucenému restartování serveru nebo k ukončení kvůli nedostatku paměti [38]. K opatrnému přístupu při použití třídy `ThreadLocal` radí i její autor Joshua Bloch [6].

5 Existující nástroje pro analýzu paměti

Nástrojů pro analýzu paměti běhového prostředí existuje celá řada. Může se jednat o programy spustitelné z příkazové řádky s textovým výstupem či plnohodnotné desktopové aplikace s komplexním grafickým rozhraním. Některé nástroje provádějí miminální množství analýzy a můžeme je považovat spíše za profilery, sofistikovanější nástroje jsou schopny paměť důkladně analyzovat a odhalit problematické používání paměti.

5.1 JDK Mission Control

JDK Mission Control (JMC) je sada *open source* nástrojů pro monitorování, profilování a případné řešení problémů aplikací běžících na JVM [32]. Od verze Java 7u40 je distribuován společně s vývojovou sadou JDK firmy Oracle. Jednotlivé nástroje lze ovládat pomocí desktopového klienta, který je k vidění na obrázku 5.1.



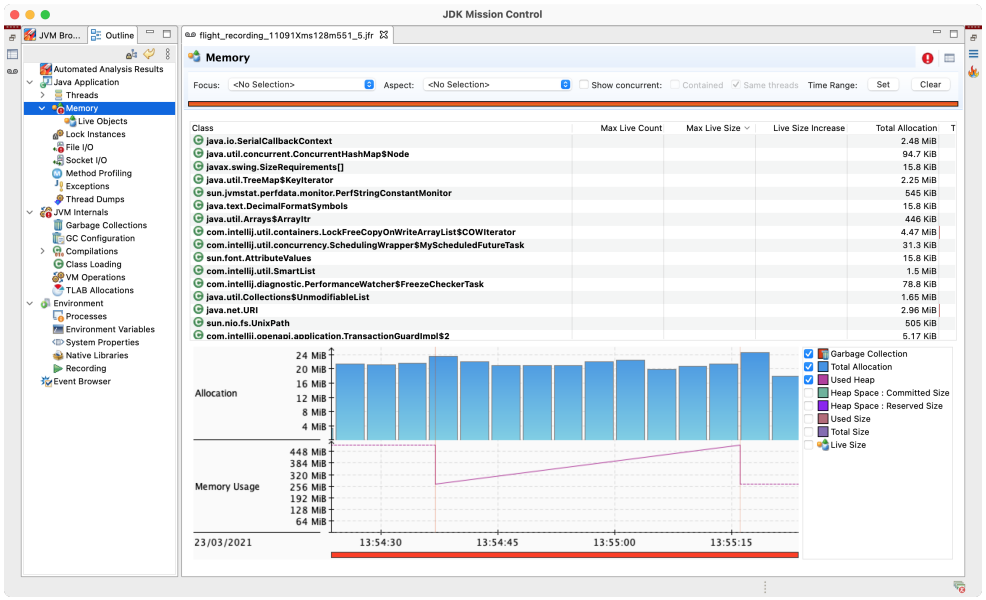
Obrázek 5.1: Grafické rozhraní Java Mission Control.

Nástroje nabízí efektivní a podrobnou analýzu dat jako je výkonnost kódu, paměť a latence. Analýzu lze díky minimální režii provádět nad aplikacemi

v produkčním prostředí. Lze sledovat jednotlivá vlákna či navěsit obsluhu na různé události pomocí tzv. *event triggers*.

5.2 Java Flight Recorder

Java Flight Recorder (JFR) je nástroj pro sběr dat za účelem profilování a diagnostikování problémů u Java aplikací [31]. Je integrován do JVM, proto má velmi nízkou režii a lze ho použít v zatížených produkčních prostředích. Bylo empiricky ukázáno, že při použití výchozích nastavení je dopad na výkon aplikace menší než jedno procento. JFR sbírá jak data o virtuálním stroji, tak o aplikaci v něm běžící. Nasbíraná data lze vizualizovat např. v JMC jako je tomu na obrázku 5.2.



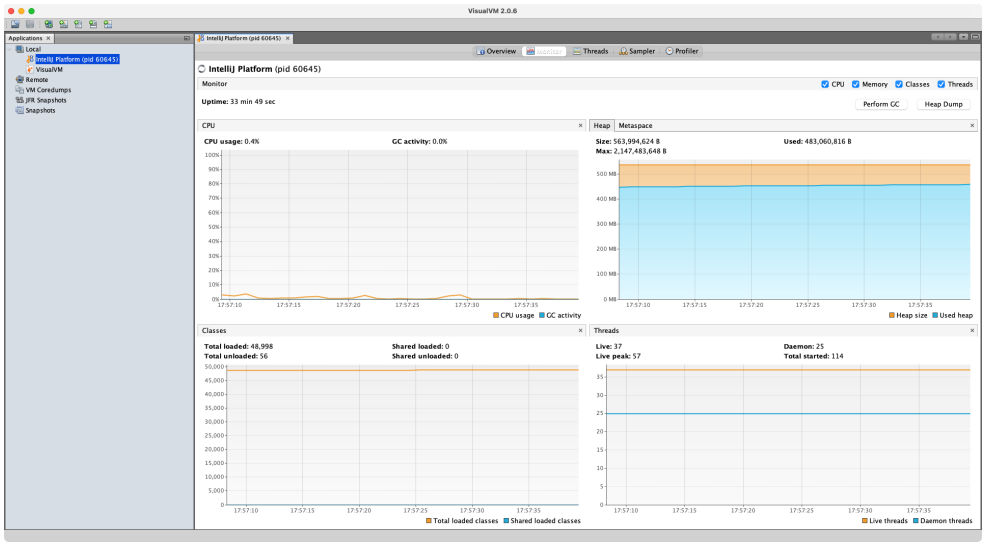
Obrázek 5.2: Vizualizace sesbíraných dat pomocí JFR v JMC.

Java Flight Recorder sbírá data o událostech, angl. *events*, které nastávají ve specifických momentech. Každá událost má název, časovou značku a nepovinný *payload*. Payload obsahuje data týkající se události jako je např. velikost haldy před a po události.

Většina událostí dále obsahuje informace o vláknu, ve kterém událost nastala, délku události a stav zásobníku. Pomocí těchto dodatečných informací lze zrekonstruovat detaily běhu virtuálního stroje či aplikace. JFR produkuje obrovské množství dat, proto je vhodné sbírat pouze pro uživatele relevantní události.

5.3 VisualVM

Jedná se o *open source* nástroj, který poskytuje grafické rozhraní pro prohlížení detailních informací běžících aplikací v JVM [37]. Nástroj se dokáže připojit i ke vzdálenému virtuálnímu stroji. VisualVM nabízí detailní pohled na využití paměti, zatížení procesorů a další užitečné údaje. Další funkcionalitou je vytvoření záznamu stavu haldy, tzv. *heap dumpu*, v daném momentě či vyžádání běhu GC. Data monitorované aplikace jsou zobrazeny v reálném čase, jako je tomu např. na ukázce 5.3.



Obrázek 5.3: Grafické rozhraní nástroje VisualVM.

Používá se především jako profiler pomocí kterého se kontroluje výkonnost aplikace a její práce s pamětí. VisualVM není schopen identifikovat problematické použití paměti, proto je uživatel musí být schopen vyčíst z dostupných dat.

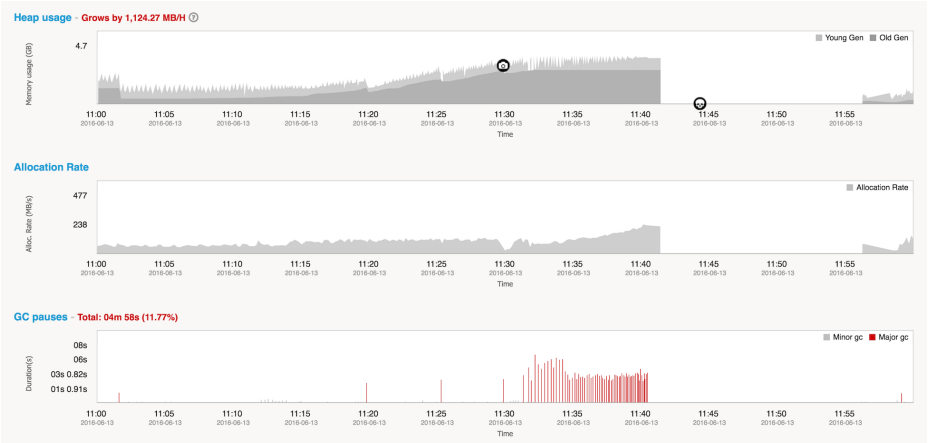
5.4 Plumbr

Plumbr je estonská firma vyvíjející produkty pro monitorování softwaru. Hlavními produkty firmy jsou Real User Monitoring (RUM) pro monitorování výkonu aplikace z pohledu uživatele a Automatic Performance Monitoring (APM) pro detekci příčin problémů s výkonem Java aplikací.

APM je jedním z mála nástrojů, které jsou schopny detekovat úniky paměti. Dále sleduje délky jednotlivých pauz[1] kolektorů, vlákna přistupující

[1] Jedná se o pauzy typu *stop-the-world*.

k zámkům, nadměrný počet volání a další události, které by mohly vést k degradaci výkonu aplikace.



Obrázek 5.4: Vizualizace využití paměti v aplikaci Plumbr[2].

5.5 JXRay

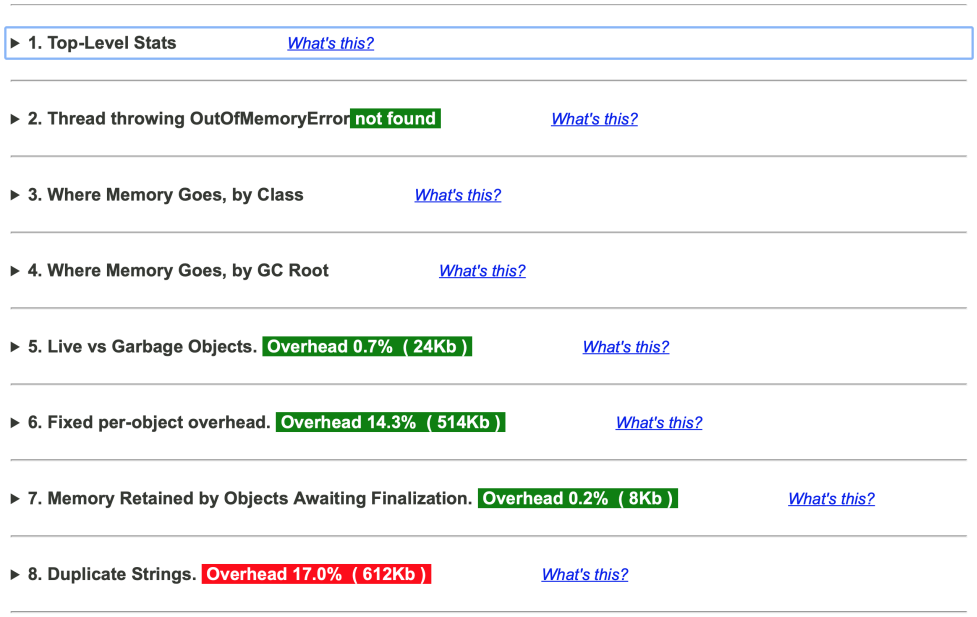
Jedná se o komerční nástroj spustitelný z příkazové řádky pro detekci neefektivního využití paměti v aplikacích [17]. Neprovádí analýzu nad běžícím virtuálním strojem, ale nad soubory ve formátu `hprof`. Jelikož se nejedná o desktopovou aplikaci, výstup analýzy je ve formátu `HTML` a nachází se v něm shrnutí analýzy, nalezené problémy a doporučení jak problémy řešit. Nástroj je možné spustit kdekoliv, včetně cloudových strojů, kde je dostupná pouze příkazová řádka. JXRay lze koupit pro jednu konkrétní osobu, která ho bude moci používat, druhou možností je platba jednoho dolaru za každou úspěšnou analýzu.

JXRay vyhledává většinu známých problémů jako jsou úniky paměti, duplicitní řetězce a ostatní objekty (provádí se mělké porovnání objektů), nevyužité kolekce a pole, nadměrné užití obalujících objektů pro primitivní typy a další. Nástroj je specifický tím, že umí doporučit takové změny, které povedou k vyřešení nalezených problémů.

Na obrázku 5.5 je ukázka výstupu nástroje. Na začátku výstupu se nachází shrnutí nejzávažnějších problémů, dále jsou uvedeny základní údaje jako je počet tříd a instancí. Nakonec jsou uvedeny všechny potenciální problémy,

[2]Obrázek dostupný z: `https://plumbr.io/memory-leak`.

které nástroj zná, a je k nim uvedeno, jestli byly v aplikaci nalezeny a jak je případně řešit.



Obrázek 5.5: Výstup analýzy nástroje JXRay ve formátu `HTML`.

5.6 Memory Analyzer

Nástroj byl vyvinut na Fakultě Aplikovaných Věd v Plzni v rámci diplomové práce zabývající se vytvořením nástroje pro analýzu Java memory heapu [22]. Jedná se o *open source*[3] program spustitelný z příkazové řádky napsaný v jazyce Java, výstup analýzy je v textové podobě.

Memory Analyzer, jako JXRay, neprovádí analýzu nad běžícím virtuálním strojem, ale pouze nad soubory ve formátu `hprof`. Nástroj umí vypsat jmenné prostory, které se v dumpu nachází, analyzovat využití kolekcí typu `List` a hledat duplicitní duplicitní objekty pomocí mělkého porovnání.

Při porovnání nástrojů Memory Analyzer a JXRay by se mohlo na první pohled zdát, že Memory Analyzer nabízí pouze výpis jmenných prostorů a podmnožinu funkčnosti nástroje JXRay. Memory Analyzer navíc umí omezit rozsah analýzy pouze na objekty z jmenného prostoru určeného uživatelem. Jeho největší výhodou je ovšem fakt, že se jedná o program s otevřeným zdrojovým kódem, který je možný rozšířit o další funkcionalitu.

[3]Dostupný na: `https://github.com/mxmxcz/memory-analyzer`.

Na obrázku 5.6 je příklad omezení rozsahu analýzy pouze na jmenný prostor `cz.mxmx`, čímž se výrazně snížil počet analyzovaných objektů. Provedla se analýza duplicitních objektů, neefektivního využití kolekcí typu `List` a kolekcí obsahující duplicity.

```
~/memory-analyzer
Analyzing classes from namespace `cz.mxmx` in `sandbox/data/test-heapdump-10.hprof`...

Done, found:
        Classes: 710
        Instances: 7707
Namespace cz.mxmx
        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.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.
                Duplicates of 'cz.mxmx.memoryanalyzer.example.Child': 10 instances of the 'cz.mxmx.memoryanalyzer.example.Child' class contain exactly the same data.

        Ineffective list 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).
                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).
                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).
                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).

        List of duplicates (1):
                List full of same values: All values in the list java.util.ArrayList#listOfDuplicates have the same value (10x).

Duration: PT0.66486S
```

Obrázek 5.6: Textový výstup nástroje Memory Analyzer.

6 Návrh implementace

6.1 Volba nástroje

V kapitole 5 byly popsány existující nástroje pro detekci problémů se správou paměti v Java aplikacích. Přestože se jedná především o *open source* nástroje, ve většině případů jde o velice komplexní projekty, do kterých přispívají především vývojáři firem Oracle a Red Hat. Každé rozšíření, tzv. *pull request*, musí být schváleno architekty daného nástroje, z tohoto důvodu je komplikované vytvářet rozšíření pro tyto nástroje.

Memory Analyzer byl vybrán jako nástroj, který se bude dále rozšiřovat. Hlavním důvodem pro výběr tohoto nástroje je fakt, že byl vyvinut pod vedením vedoucího této diplomové práce. Nástroj patří mezi středně velké projekty velikostí zdrojového kódu. Jazyk Java, ve kterém je nástroj napsaný, je jedním z preferovaných programovacích jazyků autora.

6.2 Nedostatky zvoleného nástroje

Memory Analyzer má několik nedostatků, které výrazně omezují jeho použití v praxi. Některé nedostatky se týkají nedostatečné funkcionality nástroje, jiné vznikly nesprávnou implementací algoritmů, jedná se o tzv. *bugy*.

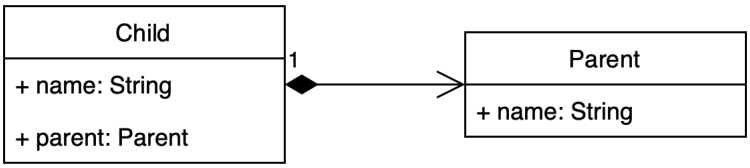
6.2.1 Duplicita objektů

Největším nedostatkem nástroje je analýza duplicitních objektů. Objekty mohou být duplicitní v případě, že pochází ze stejného stromu dědičnosti, tj. mají alespoň jednoho společného předka, kterým není `Object`, a mají stejný počet atributů. Programátoři však mohou pomocí dědičnosti vyjádřit rozdíl mezi objekty pouhým překrytím metody[1]. Objekty v jednom stromu dědičnosti mohou tedy mít stejný počet atributů, ale mohou se výrazně lišit svojí funkcionalitou, chováním a účelem.

Podmínka o stejném stromu dědičnosti není vhodná, jelikož pro programátora se nejedná o objekty stejného typu, a proto nemohou být duplicitní. Vhodnější podmínkou duplicitních objektů je rovnost jejich typů.

[1] Nahrazení funkce, kterou objekt zdědil.

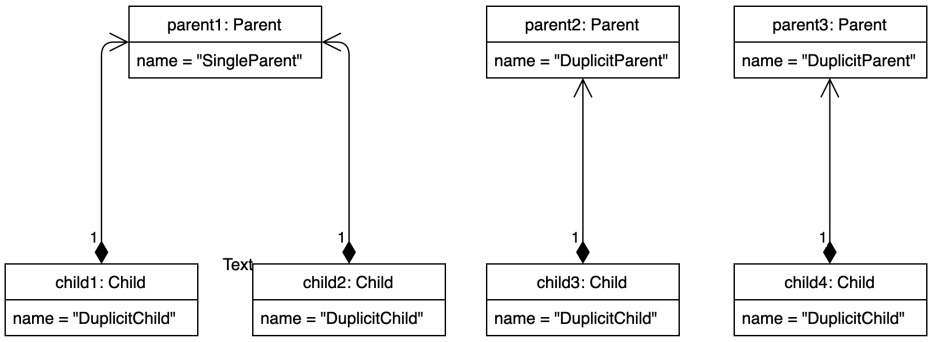
Memory Analyzer, stejně jako JXRay, provádí analýzu duplicitních objektů na základě mělkého porovnání. Referenční typy se porovnávají jako primitivní typy. Reference se tedy porovnávají pomocí jejich hodnot, což jsou adresy objektů na haldě. Na obrázku 6.1 je UML diagram tříd `Child` a `Person`. Na těchto dvou třídách bude ukázáno, kdy mělké porovnání není schopné odhalit duplicitní objekty.



Obrázek 6.1: Diagram tříd `Child` a `Person`.

Na obrázku 6.2 jsou znázorněny dvě situace, v obou situacích jsou objekty třídy `Child` duplicitní. V prvním případě obsahují obě instance třídy `Child` totožný řetězec v atributu `name` a jejich reference `parent` obsahují adresu na jeden a ten samý objekt `parent1`. Tento případ duplicity nástroj Memory Analyzer dokáže rozpoznat, jelikož, jak již bylo řečeno výše, referenční typy jsou porovnávány pomocí jejich hodnot, tj. adres objektů.

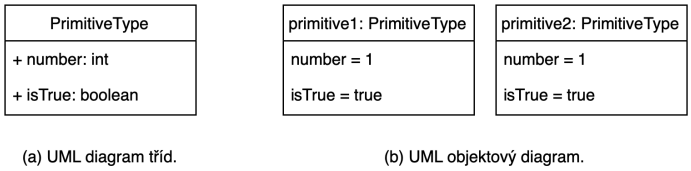
Druhý případ se liší od předchozího tím, že oba objekty třídy `Child` ukazují na dva rozdílné objekty třídy `Parent`, které jsou však duplicitní. V tomto případě není Memory Analyzer schopen rozpoznat, že jsou objekty `child3` a `child4` duplicitní.



Obrázek 6.2: Objektový diagram tříd `Child` a `Parent`.

6.2.2 Porovnání primitivních typů

Memory Analyzer je schopen detekovat duplicitu pouze v případě, že objekty neobsahují atributy primitivního typu. Na obrázku 6.3 jsou dva triviální objekty `primitive1` a `primitive2`, které obsahují jedno číslo typu `int` a logickou hodnotu typu `boolean`. Přestože je na první pohled zřejmé, že se jedná o duplicitní objekty, nástroj není schopen tuto duplicitu detekovat. Memory Analyzer je prakticky nepoužitelný, jelikož každý program obsahuje nemalé množství objektů s atributy tohoto typu.



(a) UML diagram tříd. (b) UML objektový diagram.

Obrázek 6.3: Duplicitní objekty s primitivní datovými typy.

Nástroj používá knihovnu `hprof-parser`[2] pro zpracování *heap dumpu*. Tato knihovna zajišťuje parsování jednotlivých údajů ve snímku, které nástroj dále používá pro analýzu efektivnosti správy paměti. Memory Analyzer nesprávně používá rozhraní knihovny, které vede k chybné analýze objektů s primitivními datovými typy.

6.2.3 Omezení rozsahu analýzy

Programátoři strukturují své programy do balíků, tzv. *packages*, které představují jmenné prostory. Memory Analyzer vždy omezuje rozsah analýzy pouze na jeden jmenný prostor (balík), díky čemuž je analýza podstatně rychlejší. Analýza nad jedním jmenným prostorem je však pro uživatele omezující. Uživatele často zajímají všechny jmenné prostory či nějaký jejich výčet, pouštět nástroj pro jednotlivé jmenné prostory je nepraktické.

Vhodným rozšířením nástroje by bylo přidání možnosti vynechání jmenného prostoru či výčtu jmenných prostorů z analýzy. Základní balíky jazyka jako jsou `java.lang` či `java.utils` nemá pro běžného uživatele smysl analyzovat, jelikož by nebyl schopen případně nalezené problemy řešit.

Nástroj by bylo vhodné rozšířit o více možností při výběru rozsahu, jako je možnost analyzovat celou aplikaci, výčet jmenných prostorů nebo vynechání

[2]Zdrojový kód dostupný na: `https://github.com/eaftan/hprof-parser`.

některých balíků. Větší rozsah analýzy bude logicky vést k delšímu trvání analýzy.

6.2.4 Dědičnost

Dědičnost je jedním ze základních konceptů objektově orientovaného programování. V třídové dědičnosti mohou třídy zdědit atribut a chování od existujících tříd, kterým se následně nazývají rodičovské třídy. Nástroj Memory Analyzer však neumí pracovat s dědičností, respektive s atributy rodičovských tříd. Analýza duplicit porovnává pouze atributy třídy, jejíž konstruktor byl použit pro vytvoření porovnávané instance. Z tohoto důvodu dochází k případům falešně pozitivních duplicit, jelikož instance mají stejné hodnoty v atributech odvozených tříd, ale liší se hodnotami v atributech rodičovských tříd.

6.3 Hluboké porovnání objektů

Hluboké porovnání objektů je technika pro porovnání dvou objektů, které řeší problémy popsané v sekci 6.2.1. Zásadním rozdílem mezi hlubokým porovnáním a mělkým porovnání je způsob jakým porovnávají referenční typy. Pro připomenutí, mělké porovnání objektů porovnává referenční typy pomocí jejich hodnot, což jsou adresy objektů. Dva referenční typy jsou stejné pouze v případě, že obě jejich ohodnoty jsou adresy na jeden a ten samý objekt.

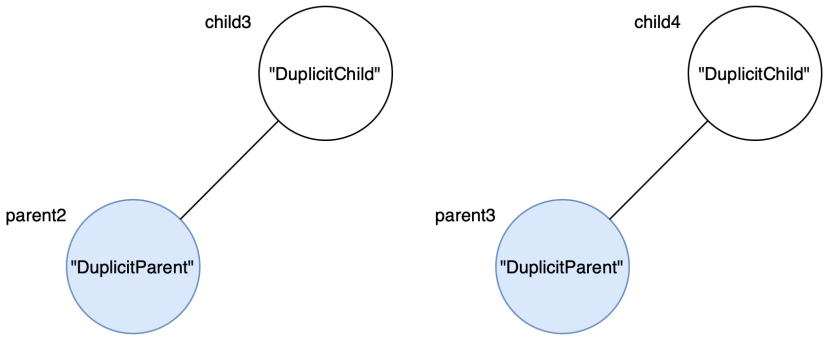
Hluboké porovnání používá odlišný přístup porovnání referenčních typů. Reference jsou porovnány na základě objektů na které ukazují, proto je tento přístup schopen detekovat duplicity v druhém případě na obrázku 6.2. Tento přístup porovnání vede na porovnání dvou stromům podobných struktur, kde kořeny jsou porovnávané objekty a uzly jsou objekty, které jsou přímo či nepřímo kořeny referencovány. Nejedná se přímo o stromové struktury kvůli možnému výskytu cyklů, které stromová struktura nepřipouští, dále budou proto označovány jako *strom objektů*.

Při porovnání dvou objektů se postupně porovnají všechny dvojice odpovídajících atributů dle těchto pravidel:

- primitivní typy se porovnají dle hodnoty,
- řetězce se porovnají dle hodnoty,
- refenční typy se porovnají dle referencovaných objektů.

Hluboké porovnání objektů vede přirozeně na rekurzivní algoritmus. Algoritmus může předčasně skončit v případě, že si některé hodnoty nejsou rovny, porovnávané objekty v tomto případě nejsou duplicitní. Objekty jsou duplicitní v případě, že proběhnou všechna porovnání a všechny porovnávané hodnoty si jsou ekvivalentní.

Obrázek 6.4 ilustruje porovnání objektů `child3` a `child4` z příkladu na obrázku 6.2. Nejdříve se porovnají řetězce v atributu `name`, které jsou stejné. Následně se porovnají objekty `parent2` a `parent3` typu `Parent`, které jsou na obrázku reprezentovány modrými uzly. Tyto objekty obsahují stejný řetězec, proto jsou stejné a duplicitní. Všechny porovnávané dvojice si jsou ekvivalentní, objekty `child3` a `child4` jsou proto duplicitní.



Obrázek 6.4: Hluboké porovnání objektů.

Tento přístup je schopný odhalit všechny duplicity, které se v aplikaci nachází. Jeho složitost je $O(2V)$, kde V je počet uzlů, respektive počet objektů na které se lze přímo či nepřímo dostat z porovnávaného objektu. Celková velikost paměti, která byla alokována pro tyto objekty, se v anglické literatuře označuje jako *retained size*. Při velké provázanosti objektů je potřeba porovnat velké množství uzlů, které výrazně zpomaluje analýzu duplicit. Z tohoto důvodu se vývojáři nástroje JXRay rozhodli použít přístup mělkého porovnání, přestože nejsou schopni detekovat všechny duplicity.

Objekty mohou být duplicitní pouze v případě, že jsou instancemi stejné třídy. Jedná se o změnu od původního návrhu, kdy objekty dvou různých tříd mohly být duplicitní pokud pocházely ze stejného stromu dědičnosti.

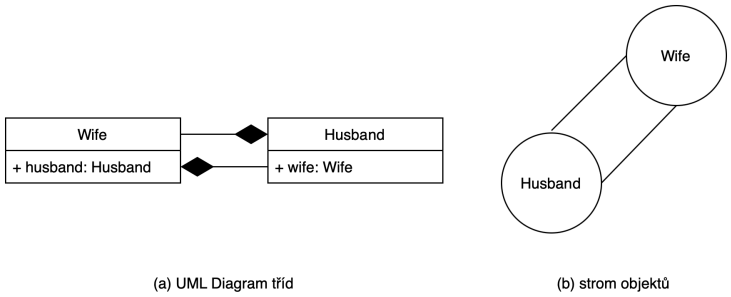
6.3.1 Zacyklení

V případě, že se objekty cyklicky referencují, vznikají v jejich stromech objektů cykly, které mohou při porovnání objektů vést k zacyklení. Příkladem

cyklických referencí jsou reference tříd `Husband` a `Wife` z obrázku 6.5. Při použití výše zmíněného algoritmu by došlo k zacyklení, je proto třeba mít mechanismus, který dokáže cyklus detekovat.

Lze využít faktu, že se jedná o obecnou grafovou strukturu a použít známé algoritmy detekce cyklů v grafech. Na začátku porovnání kořenových objektů jsou všechny uzly označeny jako nenaštívené, na začátku porovnání objektů jsou tyto objekty označeny jako naštívené. Tímto přístupem lze eliminovat zacyklení, je ovšem potřeba přidat mechanismus označování objektů.

Další možný způsob detekce zacyklení je použití unikátních identifikátorů objektů, ke kterým má Memory Analyzer přístup. Během prvního porovnání dvou objektů se nejdříve uloží jejich identifikátory jako dvojice čísel, při opětovném porovnání těchto objektů by byl cyklus detekován pomocí uložené dvojice.



Obrázek 6.5: Cyklické reference objektů.

6.3.2 Porovnání polí a kolekcí

Pole jsou kontajnery pro uložení více hodnot [27], kterými mohou být primitivní či referenční typy. Každé pole má pevně danou velikost, která je stanovena při jeho vytvoření, a nelze ji během života pole změnit. Jedná se o objekty, je vždy alokován na haldě, nelze ho alokovat na zásobníku jako je tomu například v jazycích C a C++.

Pole lze porovnat několika způsoby. První způsob porovnává objekty na stejných pozicích. Dvě pole si jsou rovny pouze v případě, že jsou stejně dlouhá a objekty na stejných pozicích si jsou rovny dle výše popsaného algoritmu.

Druhou možností je pole seřadit a poté porovnat objekty na stejných pozicích. Různé permutace polí by tedy byly považovány za duplicitní. Tento

přístup má řadu nevýhod, největší z nich je čas strávený řazením, které má obecně složitost $O(n \log n)$. Řazení pole je přímočaré v případě primitivních typů, řazení referenčních typů je značně složitější záležitost, jelikož je třeba definovat způsob jakým se budou objekty v poli řadit.

Jakoukoliv skupinu jednotlivých objektů, které jsou reprezentovány jako jedna jednotka, lze označit jako kolekci. Java Collections Framework je sada tříd a rozhraní pro práci s kolekcemi v jazyce Java. Některé kolekce interně uchovávají objekty v poli, příkladem je kolekce `ArrayList`. Porovnání těchto kolekcí je prakticky porovnání dvou polí, které již bylo popsáno výše. Kolekce, které implementují například spojový list, mohou však používat jiný způsob ukládání objektů než je ukládání pomocí pole, a jejich porovnání je třeba provést jiným způsobem.

Analýza duplicitních objektů pomocí hlubokého porovnání je sama o sobě výpočetně náročná operace, proto se kolekce a pole budou porovnávat pouze pomocí základního algoritmu pro porování dvou objektů a speciální algoritmus pro ně vyvinut nebude.

6.4 Počítání referencí

Programy běžící na virtuálním stroji JVM mohou ukládat na zásobník pouze primitivní datové typy a reference na objekty, které lze však alokovat pouze na haldě. Reference jsou proto nedílnou součástí každého programu a jsou také zdroji neefektivního využití paměti, které byly popsány v kapitole 3. Asociace a kompozice objektů lze vyjádřit pouze pomocí referencí, objekty mají proto velmi často vysoký faktor neefektivnosti.

Zajímavým údajem pro uživatele proto může být celkový počet referencí, které se v objektu nachází. Programátor může díky vysokému podílu paměti využité referencemi zjistit, že je program přílíš abstratní a mnoho paměti je využito k vyjádření vztahů mezi objekty.

Jedním z míst, kdy je paměť nevyužita, jsou `null` reference. Jedná se o reference, které neobsahují adresu na žádný validní objekt na haldě, ale speciální hodnotu `null`. Jak bylo výše zmíněno, výchozí hodnotou pro všechny referenční typy je právě hodnota `null`. Tato reference je ve většině případů k ničemu, přesto však zabírá stejně jako klasická reference. Uživateli informace o počtu `null` referencí může říct, jestli nejsou některé reference zbytečné a nebylo by je lepší z programu odstranit.

6.5 Interaktivní dotazování na objekty

Memory Analyzer je neinteraktivní aplikace, která se dá pustit z příkazové řádky. Výstup analýzy problémů se správou paměti je pouze v textovém formátu, který neobsahuje pro uživatele vždy relevantní informace. Příkladem je výstup analýzy duplicitních objektů, kde je uvedeno jméno třídy a počet jejích instancí, které byly analyzovány jako duplicitní. Pro uživatele by mohlo být užitečné mít k dispozici identifikátory duplicitních instancí, aby tyto instance mohl sám podrobněji analyzovat.

Zobrazení informací instance s daným identifikátorem je dalším možným rozšířením nástroje Memory Analyzer. Uživateli se tím výrazně zjednoduší analýza problematických instancí. Nástroj by se rozšířil o interaktivní režim, tzv. *shell*, kterého by se uživatel pomocí identifikátorů dotazoval na objekty, které ho zajímají.

7 Implementace

Během tvorby praktické části diplomové práce byl kladen důraz především na snadnou rozšiřitelnost kódu, jelikož je velice pravděpodobné, že ho další student bude dále rozšiřovat. Kód je lehce rozšiřitelný pokud je řádně otestovaný a dodržuje základní programovací principy jako jsou principy SOLID a DRY. Cílem práce je opravit nedostatky Memory Analyzeru popsané v sekci 6.2 a rozšířit ho o další funkcionalitu.

7.1 Zvolené technologie

Nástroj Memory Analyzer je napsaný v jazyce Java verzi 1.8 a k sestavení programu se používá nástroj Apache Maven [39]. Jelikož nástroj běží na virtuálním stroji JVM, lze ho rozšířit v původním jazyce Java nebo přidat další funkcionalitu v jazyce, který je také přeložitelný pro tento virtuální stroj. Takovým jazykem je například dříve zmíněný jazyk Kotlin [19].

Kotlin je multiplatformní, staticky typovaný programovací jazyk, který lze zkompilovat pro běh v JVM či do jazyka JavaScript [20]. Jazyk byl navržen tak, aby byl interoperatibilní s Javou, verze standardní knihovny Kotlinu pro JVM je dokonce závislá na standardní knihovně Javy. Kotlin vynucuje bezpečnou práci s null referencemi, program nikdy proto nemůže havarovat při dereferencování tohoto typu referencí. Programy napsané v Kotlinu lze sestavit pomocí výše zmíněného nástroje Apache Maven.

Vedoucí vývoje jayzka uvedl, že je Kotlin navržen jako průmyslově spolehlivý objektově orientovaný jazyk, který je lepší než Java. Interoperatibilita s Javou umožňuje společnostem postupný přechod z Javy na Kotlin. Kotlin má podporu od významných Java frameworků jakými jsou například Hibernate či Spring Boot [41].

Kotlin je autorem preferovaný jazyk, nabízí se proto, aby se rozšíření nástroje napsalo v tomto jazyce. Java je však oproti Kotlinu velice populární jazyk při výuce programování. Nástroj Memory Analyzer bude nejspíše dále rozšiřován dalšími studenty, u kterých je pravděpodobnější znalost Javy než Kotlinu. Přes všechny výše uvedené výhody Kotlinu, budou z tohoto důvodu rozšíření implementovány v jazyce Java.

7.2 Struktura programu

Původní autor nástroje rozdělil hlavní modul `memory-analyzer` do tří následujících `maven` submodulů, které se budou během implementace dále rozšiřovat.

`analyzer`

Tento modul se stará o zpracování *heap dump* souboru. Samotné parsování souboru provádí knihovna `hprof-parser`. Knihovna během parsování postupně předává informace ze kterých se vytvoří objekt reprezentující heap dump.

Dále se v něm provádí veškerá analýza problematické práce s pamětí. Jsou tu definované jednotlivé analyzátory, které se sdružují do tzv. *pipeline*. Pipeline je princip výrobní linky, kde analyzátory jsou jednotlivé kroky na této lince. Vstupem každého analyzátoru je objekt reprezentující heap dump a výstup si pipeline ukládá do pole výsledků.

Tímto oddělením lze používat modul `analyzer` jako knihovnu a vytvořit pro ni rozhraní grafické nebo v případě nástroje Memory Analyzer rozhraní pomocí příkazové řádky.

`app`

Poskytuje uživateli rozhraní pomocí příkazové řádky. Jedná se o vstupní bod nástroje, kde se zpracují požadavky od uživatele a následně se požadavek provede. Uživatel si pomocí tohoto modulu může například zvolit, jestli se bude provádět analýza paměti nebo pouze výpis jmenných prostorů.

`example-app`

Jedná se o modul s testovacími aplikacemi, pomocí kterých se vytváří testovací heap dumpy. Aplikace vždy vygenerují uživatelem požadovaný počet objektů.

7.3 Implementace analyzátorů

Jak bylo výše řečeno, vstupem každého analyzátoru je objekt reprezentující heap dump načtený ze souboru. Tento objekt vždy implementuje rozhraní `MemoryDump`, které je ukázáno ve fragmentu kódu 7.1 společně s jeho nejdůležitějšími metodami. Instance tohoto rozhraní musí být schopna poskytnout

analyzátorům relevantní informace nalezené v heap dumpu, což jsou především informace o jmenných prostorech, třídách a jejích instancích. Nejdůležitější je metoda `getUserInstances`, která vrací pouze instance z jmenného prostoru určeného uživatelem.

```
public interface MemoryDump {
    Collection<String> getUserNamespaces();
    Map<Long, InstanceDump> getInstances();
    Map<Long, ClassDump> getClasses();
    Map<Long, InstanceDump> getUserInstances();
    /* jsou uvedeny pouze nejdulezitejsi metody */
}
```

Fragment kódu 7.1: Nejdůležitější metody rozhraní `MemoryDump`.

Vytvoření nového analyzátoru znamená pouze vytvoření třídy, která bude implementovat rozhraní `MemoryAnalyzer`, která vyžaduje pouze jednu metodu `findMemoryWaste` jak je ukázáno ve fragmentu kódu 7.2. Všechny analyzátory se nachází v submodulu `analyzer`.

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

Fragment kódu 7.2: Rozhraní `WasteAnalyzer`.

Výstupem každé analýzy je seznam nalezených problémů, které jsou reprezentovány instancemi rozhraní `Waste`. Každá instance problému obsahuje jeho popis, seznam instancí, které tento problém způsobily, a analyzátor, který daný problém nalezl. Fragment kódu 7.3 ukazuje nejdůležitější metody tohoto rozhraní.

```
public interface Waste extends Comparable<Waste> {
    String getTitle();
    String getDescription();
    List<InstanceDump> getAffectedInstances();
    void addAffectedInstance(InstanceDump instanceDump);
    WasteAnalyzer getSourceWasteAnalyzer();
}
```

Fragment kódu 7.3: Rozhraní `Waste`.

Všechny výše popsané třídy byly vytvořeny původním autorem nástroje a během práce byly pouze mírně upraveny.

7.4 Analyzátor referencí

Jedním z rozšíření nástroje, které bylo navrženo a popsáno v sekci 6.4, je analýza referencí. Byl vytvořen analyzátor `ReferenceWasteAnalyzer`, který

tuto analýzu provádí. Analýza probíhá nad instancemi, které lze získat z heap dumpu pomocí metody `getUserInstance`.

Jednotlivé instance obsahují atributy, u kterých je třeba rozhodnout, jestli se jedná o referenční či primitivní typ. Toto rozhodnutí však není vždy triviální, jelikož instance reprezentující atributy nejsou homogenní. V případě, že se jedná o instanci třídy `String` či `InstanceDump`, se vždy jedná o referenční typ. Atributy však mohou být reprezentovány třídou `Value`, což je třída externí knihovny `hprof-parser`, která se v nástroji používá pro parsování souboru s heap dumpem. Instance této třídy se používají pro atributy primitivního typu, null reference či reference na primitivní pole.

Výstupem analyzátoru je počet všech nalezených referencí a počet null referencí. Tyto dva údaje mohou uživateli říci, jestli program trpí příliš vysokým počtem null referencí s ohledem na celkový počet referencí.

7.5 Analyzátor duplicitních instancí

Hlavním cílem této práce je nová analýza duplicitních objektů pomocí hlubokého porovnání objektů. Původní analýza se prováděla pomocí mělkého porovnání a obsahovala chyby, které neumožňovaly porovnání primitivních typů.

Analýzu duplicitních instancí provádí třída `DuplicateInstanceAnalyzer`, která byla celá změněna a stejný zůstal pouze název třídy. Původní algoritmus porovnával každý objekt s každým. Tento přístup je při použití hlubokého porovnání zbytečně pomalý. Nově se vždy veme prvek z konce kolekce, který se porovná s ostatními objekty. Všechny nalezené duplicity se z kolekce vyřadí pouhým nastavením reference na null. Díky tomu nedochází k žádným posunům v kolekci jako při běžném odstranění a nedochází k dalšímu zbytečnému porovnání.

Porovnání dvou objektů se provádí metodou, která se rekurzivně volá v případě, že se porovnávají dva referenční typy. Jak bylo výše zmíněno, v některých případech může dojít k zacyklení. Tomu se lze vyhnout pomocí ukládání identifikátorů porovnávaných instancí do datové struktury `HashSet`. Před každým porovnáním je tato datová struktura zkontrolována, jestli se daná dvojice již neporovnává v jiném volání metody. Porovnání dvou objektů není omezené hloubkou rekurze a je odolné proti cyklům. Fragment kódu 7.4 ukazuje, jakým způsobem se detekuje cyklus.

```
if (currentlyComparing.contains(ids)) {
    return true;
} else {
    currentlyComparing.add(ids);
}
```

Fragment kódu 7.4: Detekce cyklu během hlubokého porovnání.

7.6 Zrychlení porovnání

Hluboké porovnání je pomalé, pokud jsou objekty v sobě hluboce zanořeny. Během implementace bylo třeba vyřešit, jak algoritmus zrychlit, aby byl použitelný v praxi.

7.6.1 Paralelizace

Jedním z prvních nápadů bylo porovnávat několik dvojic najednou, tj. paralelizovat algoritmus. Díky verzi Java 1.8 není třeba složitého manuálního paralelizování a je možné použít tzv. paralelizované streamy. Paralelizace však nepřinesla tížený efekt zrychlení. Při bližší analýze bylo zjištěno, že důvodem byla potřeba synchronizovat ukládání duplicitních hodnot.

7.6.2 Ukládání výsledků porovnání

Objekty mohou být referencovány více objekty najednou. Může se proto stát, že se během analýzy některé dvojice porovnávají vícekrát. Urychlení by mohlo přinést ukládání výsledků porovnání do datové struktury, ze které lze vybírat prvky s konstantní složitostí. Takové datové struktuře se říká *cache*. Tímto by se předešlo opakovenému porovnání již porovnaných dvojic.

Pro ukládání výsledků jednotlivých porovnání byla použita datová struktura `HashSet`. Při uložení je třeba vytvořit pro danou dvojici unikátní klíč, nejlépe z jejich identifikátorů, jelikož jsou také unikátní. Unikátní klíč lze například vytvořit spojením identifikátorů do řetězce a oddělit je specifickým znakem. Vytváření řetězců je však pomalá operace, proto tento způsob není vhodný. Elegantním řešením je Szudzikova párovací funkce, která pro dvě pozitivní čísla vytvoří unikátní pozitivní hodnotu spojenou právě s touto dvojicí [42]. Funkce je definována jako

$$f(x,y) = \begin{cases} x * x + x + y : x \geq y \\ x + y * y : x < y. \end{cases} \tag{7.1}$$

Unikátní klíče se tímto způsobem vytváří velice rychle, přesto testování neukázalo žádné zrychlení, algoritmus naopak výrazně zpomalil. Důvodem byl velice nízký *cache hit rate*, který se pohyboval kolem 30%. Často docházelo k zbytečnému vytváření klíče a volání metody na výběr prvku. Dalším problémem byla velikost cache, která mnohdy zapříčinila výjimku `OutOfMemoryError`.

7.6.3 Seřazení atributů

Algoritmus je výpočetně náročný především kvůli rekurzivnímu procházení referenčních typů, výjimkou jsou pouze řetězce třídy `String`. V nejhorším případě se musí porovnat všechny objekty, které jsou přímo či nepřímo dostupné z porovnávaných objektů. Porovnání se předčasně ukončí při nálezu nerovnosti, v tomto případě nejsou porovnávané objekty duplicitní.

Při libovolném pořadí, ve kterém se porovnávají atributy objektů, může dojít k zbytečnému rekurzivnímu porovnání objektů. Příkladem může být situace, kdy mají objekty rozdílné hodnoty v atributu primitivního typu, ale nejdříve se porovnají atributy referenčího typu. Tímto může dojít k mnohým zbytečným porovnáním, kterým by se dalo předejít tím, že se nejdříve porovnají atributy primitivního typu. Během profilování algoritmu vyšlo najevo, že se velké množství výpočetního času stráví právě zbytečným porovnáním objektů, které není třeba z výše popsaných důvodů.

Atributy byly původně porovnávany v pořadí, ve kterém je nástroj přijal od knihovny na parsování heap dumpu. Pro seřazení atributů bylo třeba upravit velkou část nástroje, která komunikuje s výše zmíněnou knihovnou. Tato změna výrazně urychlila proces porovnání dvou objektů.

7.7 Interaktivní shell

Jedním z požadavků vedoucího byla implementace interaktivního shellu, který lze použít pro snadnou kontrolu jednotlivých instancí pomocí jejich číselného identifkátoru. Interaktivní shell byl implementován v modulu `app`. Lze zadat jeden či více identifikátorů instancí, které uživatele zajímají.

Příkazy jsou implementovány návrhovým vzorem náštěvník (angl. visitor), který umožňuje snadné přidání dalších příkazů [15]. Návrhový vzor definuje hierarchii tříd se společným předkem. Tento předek deklaruje metodu, která je v každé podtřídě implementována jinak, těmito podtřídami jsou jednotlivé příkazy. Přidání dalšího příkazu se skládá z vytvoření nové

třídy, která bude potomkem třídy `GenericCommand`, a doplnění rozhraní `CommandVisitor` o metodu, která příkaz vykoná. Nakonec je třeba doplnit metodu `parseCommand` o příslušný kód, který dokáže nový příkaz zpracovat. Ve fragmentu kódu 7.5 jsou ukázany relevantní třídy a metody pro přidání nového příkazu.

```
// soubor CommandsVisitor.java
public interface CommandsVisitor {
    void visitIdsCommand(IdsCommand idsCommand);
}

// soubor GenericCommand.java
public abstract class GenericCommand {
    public abstract void accept(CommandsVisitor visitor);
}

// soubor Shell.java
private GenericCommand parseCommand(String command) {
    return new InvalidCommand();
}
```

Fragment kódu 7.5: Relevantní třídy a metody pro přidání nového příkazu.

7.8 Testy

Nástroj byl původně pokryt minimálním počtem testů. Testování softwaru je klíčová součást vývoje, během práce proto byla vytvořena sada jednotkových, integračních a regresních testů. Testy se dle zvyklostí Maven projektů nachází v adresáří `src/test`.

7.8.1 Jednotkové testy

Pro jednotkové testy byl použit testovací rámec JUnit 4 [16]. Testy ověřují především funkcionalitu analyzátorů a jejich veřejených metod. Testy například testují správné chování při hlubokém porovnání objektů s cyklickými referenci.

7.8.2 Integrační testy

Integrační testy testují komunikaci mezi moduly `app` a `analyzer`. Cílem testů je ověřit, že uživatel může nástroj ovládat pomocí rozhraní příkazové řádky. Tyto testy se provádí pomocí jednotkových testů a nachází se v modulu `app`.

7.8.3 Regresní testy

Regresní testy mají za úkol ověřit, že zásahy do aplikace nezpůsobily novou chybu v již funkčních částech aplikace. Testy se pouští nad heap dumpy, pro které byla provedena manuální analýza. Ke správnému fungování testů je třeba adresář, který obsahuje soubory s těmito heap dumpy. Regresní testy byly napsány pouze pro modul `analyzer`.

8 Výsledky

Nástroj rozšířený o novou funkcionalitu byl otestován a porovnán s původním nástrojem na sadě heap dumpů. Jednalo se o heap dumpy jak z reálných aplikací, tak z aplikací, které byly vytvořeny za účelem testování nástroje.

8.1 Ověření na testovacích aplikacích

Ověření implementovaného rozšíření bylo provedeno na heap dumpech, které byly vytvořeny z aplikací nacházejících se v modulu `example-app`. Výhodou tohoto přístupu je především předem známý počet duplicitních instancí, případně jiných problémů týkajících se práce s pamětí.

První dva experimenty jsou založeny na hledání duplicitních instancí dvou tříd, které jsou k vidění ve fragmentu kódu 8.1. Experimenty se provedly s novou i s původní verzí nástroje, třídy proto neobsahují primitivní atributy, jelikož s nimi původní nástroj neumí pracovat.

```
class Child {
    String childName;
    Parent parent;

    public Child(String childName, Parent parent) {
        this.childName = childName;
        this.parent = parent;
    }
}

class Parent {
    String parentName;

    public Parent(String parentName) {
        this.parentName = parentName;
    }
}
```

Fragment kódu 8.1: Třídy `Child` a `Parent`.

První experiment obsahuje duplicity, které je možné nalézt i při použití mělkého porovnání objektů. Způsob, jakým byly vytvořeny duplicitní instance, je ukázán ve fragmentu kódu 8.2. Vnitřní cyklus vždy vytvoří sadu deseti duplicitních instancí třídy `Child`, které obsahují stejný řetězec a referenci ukazující na jednu a tu samou instanci třídy `Parent`. Tento proces

se opakuje desetkrát, celkový počet duplicitních instancí třídy `Child` je sto. Účelem tohoto experimentu je demonstrace schopnosti rozšířeného nástroje nálézt všechny duplicity, které dokáže nalézt nástroj původní.

```
public static void shallowComparison() {
    int instanceDuplicateCount = 10;
    int totalCount = 10;
    List<Child> children = new ArrayList<>();
    for (int i = 0; i < totalCount; i++) {
        Parent parent = new Parent("Parent " + i);
        for (int j = 0; j < instanceDuplicateCount; j++) {
            children.add(new Child("Child " + i, parent));
        }
    }
}
```

Fragment kódu 8.2: Duplicitní objekty detekovatelné mělkým porovnáním.

Výsledky experimentu se nachází v tabulce 8.1. Oba typy detekce byly schopny nalézt všechny duplicitní instance. Rozšířený nástroj je tedy schopný detekovat minimálně stejnou množinu duplicit jako původní nástroj.

Verze nástroje	Počet nalezených duplicit
původní	100
rozšířená	100

Tabulka 8.1: Počet nalezených duplicit při experimentu s mělkou duplicitou.

Druhý experiment má za úkol ověřit správnou implementaci hlubokého porovnání objektů. Na fragmentu kódu 8.3 je k vidění kód, který duplicitní objekty vygeneroval. Kód je velmi podobný kódu z fragmentu kódu 8.2. Zásadní rozdíl je však v tom, že žádné dvě instance třídy `Child` nemají referenci na stejnou instanci třídy `Parent`. Při hledání duplicit je třeba v tomto případě použít hluboké porovnání objektů. Při pokusu bylo vygenerováno sto duplicitních instancí třídy `Child` a sto duplicitních instancí třídy `Parent`.

Výsledky tohoto experimentu se nacházejí v tabulce 8.2. Rozšířený nástroj opět nalezl všechny duplicitní objekty. Původní nástroj byl však schopen detekovat pouze duplicity u instancí třídy `Parent`. Instance třídy `Child` původní nástroj považuje za duplicitní pouze v případě, že mají duplicitní řetězec v atributu `name` a reference `parent` ukazuje na jednu a tu samou instanci.

```
public static void deepComparison() {
    int instanceDuplicateCount = 10;
    int totalCount = 10;
    List<Child> children = new ArrayList<>();
    for (int i = 0; i < totalCount; i++) {
        for (int j = 0; j < instanceDuplicateCount; j++) {
            Parent parent = new Parent("Parent " + i);
            children.add(new Child("Child " + i, parent));
        }
    }
}
```

Fragment kódu 8.3: Duplicitní objekty detekovatelné hlubokým porovnáním.

Verze nástroje	instancí `Child`	instancí `Parent`
původní	0	100
rozšířená	100	100

Tabulka 8.2: Počet nalezených duplicit při experimentu hluboké duplicity.

Poslední experiment testuje schopnost nástroje detekovat duplicity objektů v případě, že se skládají z primitivních typů. Pokus se provedl na velmi primitivní třídě, která obsahuje pouze jeden atribut typu `long`. Tato třída je k vidění na fragmentu kódu 8.4. Během experimentu bylo vygenerováno sto duplicitních instancí této třídy. V tabulce 8.3 se nacházejí výsledky tohoto experimentu. Rozšířený nástroj nalezl všechny duplicitní instance, zato původní nástroj nebyl schopen odhalit žádný případ duplicity, přestože se jedná o triválního příklad duplicitních objektů.

```
class PrimitiveField {
    long number;
    public PrimitiveField(long number) {
        this.number = number;
    }
}
```

Fragment kódu 8.4: Třída s atributem primitivního typu.

Verze nástroje	Počet nalezených duplicit
původní	0
rozšířená	100

Tabulka 8.3: Počet nalezených duplicit při experimentu primitivních typů.

Nově rozšířený a opravený nástroj spolehlivě detekoval všechny případy duplicitních instancí. Z výsledků je patrné, že nová verze nástroje je schopna detekovat výrazně více případů duplict než verze předchozí. Ani jedna z verzí nástroje neumí detekovat duplicity kolekcí a polí.

8.2 Ověření na reálných aplikacích

Nástroj byl otestován na sadě reálných aplikací. Měření bylo prováděno na stroji MacBook Pro 2015 s následující konfigurací:

- Operační systém: MacOS 11.2.1,
- Procesor: 2,7 GHz Dual-Core Intel Core i5,
- Paměť: 8 GB 1867 MHz DDR3,
- GPU: Intel Iris Graphics 6100 1536 MB.

Některé experimenty byly provedeny s novou i původní verzí nástroje, není-li verze nástroje explicitně zmíněna, jedná se o verzi novou. Experimenty měřící čas trvání běhu byly pětkrát opakovány pro každou verzi nástroje. Každé opakování proběhlo s novou instancí virtuálního stroje JVM. Výsledky experimentů jsou uloženy v adresáři `Vysledky`.

8.2.1 Spring Boot

Spring Boot[1] je aplikační rámec cílící platformu JVM, který se používá k vývoji aplikací. V aplikačním rámci byla napsána jednoduchá aplikace typu *Hello World*, ze které byl vytvořen heap dump o velikosti 26 MB. Na tomto heap dumpu byly puštěny obě verze nástroje a byla provedena analýza duplicitních instancí. Během prvního experimentu byl analyzován pouze jmenný prostor `org.springframework`, ve kterém se nacházelo 11597 instancí. Podobné testování proběhlo v [22]. Výsledky experimentu se nachází v tabulce 8.4. Rozšířená verze odhalila výrazně více duplicit než nástroj původní a v tomto případě byla i značně rychlejší.

Dále byly změřeny rychlosti obou verzí analýzy duplicit na několika různých jmenných prostorech. Výsledky měření se nachází v tabulce 8.5. Z výsledků je na první pohled patrné, že je nová verze analýzy v těchto případech rychlejší, přestože provádí podstatně komplexnější analýzu. Nová verze analýzy porovnává pouze instance stejné třídy. Důležitým faktorem rychlosti této

[1]Dokumentace dostupná na: `https://spring.io/projects/spring-boot`.

Verze nástroje	Nalezených duplicit	Doba trvání [s]
původní	767	21.2
rozšířená	2492	0.76

Tabulka 8.4: Analýza jmenného prostoru `org.springframework`.

analýzy je proto průměrný počet instancí na třídu, který je uveden v sloupci T/I ve výše zmíněné tabulce. Tímto údajem se však nelze vždy řídit, jelikož mohou nastat případy, kdy s relativně nízkým průměrným počtem instancí na třídu bude většina instancí náležet několika málo třídám. V těchto případech bude analýza pomalejší než v případě, kdy by každá třída měla stejný počet instancí. Rychlost analýzy dále závisí na provázanosti instancí a především na míře jejich podobnosti, jelikož algoritmus porovnání je předčasně ukončen při prvním výskytu nerovnosti atributů.

Jmenný prostor	Tříd	Instancí	I/T	Pův. [s]	Roz. [s]
java.util.concurrent	113	17748	157.1	52.67	31.93
org	3361	15235	4.5	40.56	0.97
org.springframework	2342	11597	4.9	21.2	0.76
org.apache	570	3167	5.6	1.08	0.29
sun	621	8309	13.4	9.9	1.34

Tabulka 8.5: Porovnání rychlosti analýzy duplicit původní a nové verze.

Duplicitní instance se při analýze sdružují do tzv. *clusterů*. Cluster je skupina instancí, kde všechny instance udržují stejná data, tedy jsou duplicitní. Každá třída může mít těchto clusterů několik. Cluster s vysokým počtem instancí může být známkou neefektivní práce s pamětí a zbytečného vytváření instancí.

V tabulce 8.6 jsou uvedeny názvy tříd s jejich počtem clusterů. Jedná se o výsledek analýzy duplicit za využití hlubokého porovnání jmenného prostoru `org.springframework` ve výše zmíněném heap dumpu. Uvedeno je pouze prvních 10 tříd s nejvyšším počtem duplicit. Z většiny se jedná o datové třídy[2]. V tabulce 8.7 se nachází informace o jednotlivých clusterech. Uvedeno je opět pouze prvních deset clusterů s nejvyšším počtem duplicit. Jednotlivé clustery byly analýzovány za použití implementovaného interaktivního shellu. Na fragmentu kódu 8.5 jsou k vidění instance clusterů třídy

[2]Datová třída je taková třída, která obsahuje pouze data a metody k nim přistupující.

`DefaultValueHolder`. Tato třída má pouze jeden atribut, ve kterém si každá anotace ukládá svoji výchozí hodnotu. Cluster o velikosti 122 duplicitních instancí ukládá pouze prázdný řetězec. Výchozí hodnota anotací se v průběhu programu nejspíše nemění. Nabízí se proto otázka, proč anotace, které mají stejnou výchozí hodnotou, nesdílí jednu a tu samou instanci namísto toho, aby měly každou svou vlastní.

Název třídy	Clusterů	Duplicit
AnnotationAttributes	22	442
DefaultValueHolder	4	349
BeanMetadataAttribute	5	286
ConvertiblePair	116	249
SoftEntryReference	101	211
LinkedMultiValueMap	10	130
MutablePropertyValues	1	119
Signature	14	95
ResolvableType	23	94
Type	27	70

Tabulka 8.6: Analýza Spring Boot: Třídy a jejich clustery.

Cluster	Duplicit
BeanMetadataAttribute	128
DefaultValueHolder	122
MutablePropertyValues	119
AnnotationAttributes	94
BeanMetadataAttribute	78
DefaultValueHolder	77
BeanMetadataAttribute	76
DefaultValueHolder	75
DefaultValueHolder	75
AnnotationAttributes	75

Tabulka 8.7: Analýza Spring Boot: Jednotlivé clustery.

```
id: 29110028248
class: org.springframework.core.annotation.
   AnnotationUtils$DefaultValueHolder
        defaultValue : String = ‘‘
id: 29115614976
class: org.springframework.core.annotation.
   AnnotationUtils$DefaultValueHolder
        defaultValue : String = ‘(inferred)‘
```

Fragment kódu 8.5: Instance clusterů třídy `DefaultValueHolder`.

8.2.2 Automatic Preventive Maintenance

Automatic Preventive Maintenance je software vyvíjený plzeňskou firmou Aimtec[3]. Jedná se o webovou aplikaci, která je napsána v aplikačním rámci Spring Boot. Jejím hlavním účelem je automatizace preventivní údržby hlavního produktu firmy, kterým je DCIx[4]. Údržba zahrnuje například analýzu SQL dotazů či vytíženosti serveru, na kterém produkt běží. Všechny informace, které jsou třeba k analýze, jsou uloženy v databázi.

Během provádění preventivní údržby u zákazníka byl z této aplikace vytvořen heap dump o velikosti 74 MB. Nad tímto heap dumpem byla provedena analýza jmenného prostoru `com.aimtecglobal`, ve kterém se nachází všechny třídy vytvořené vývojáři firmy Aimtec. Výsledky analýzy duplicit se nachází v tabulce 8.8 a výsledky analýzy referencí jsou k vidění v tabulce 8.9. Heap dump obsahuje citlivá data zákazníka, proto není v adresáři `Vysledky` dostupný podrobný výstup těchto analýz.

Tříd	Instancí	I/T	Duplicit	Clusterů	Doba trvání [s]
250	15392	61.6	0	0	12.2

Tabulka 8.8: Analýza duplicit jmenného prostoru `org.aimtecglobal`.

Referencí	Null referencí	Poměr null referencí
63291	11375	0.18

Tabulka 8.9: Analýza referencí jmenného prostoru `org.aimtecglobal`.

[3] `https://www.aimtecglobal.com/en/`
[4] `https://www.aimtecglobal.com/en/dcix/`

Analýza duplicit nenašla žádný případ duplicity, což by mohlo nasvědčovat o chybném fungování analýzy. Nástroj pracuje pouze s daty z databáze ve formě instancí databázových entit. Každá tabulka v databázi používá databázové indexy, a proto má každá instance unikátní klíč (sloupec ID). Analýza duplicit funguje správně, ale není vhodná pro tento typ aplikací. Analýza referencí neodhalila žádnou neefektivitu, null reference tvoří 18% všech referencí.

8.2.3 IntelliJ IDEA

IntelliJ IDEA[5] je komerční vývojové prostředí pro programovaní v jazycích Java, Kotlin, Groovy a dalších. Nástroj je vyvíjený firmou JetBrains, jejíž sídlo se nachází v Praze. IntelliJ IDEA je dostupná v komunitní open source edici a proprietární komerční edici, která je zdarma dostupná studentům pro studijní účely.

Z vývojového prostředí (komerční edice), ve kterém byl otevřený projekt Memory Analyzer, byl vytvořen heap dump o velikost 294 MB. Nad tímto heap dumpem byla provedena analýza jmenného prostoru `com.intellij`. Výsledky analýzy duplicit a referencí se nachází v tabulce 8.10, resp. v tabulce 8.11. Odstraněním duplicitních instancí by se odstranilo 76732 instancí, což je téměř 16% celkového počtu instancí. Null reference tvoří 29% všech referencí.

Tříd	Instancí	I/T	Duplicit	Clusterů	Doba trvání
22831	492248	21.56	79855	3123	14 min 54 s

Tabulka 8.10: Analýza duplicit jmenného prostoru `com.intellij`.

Referencí	Null referencí	Poměr null referencí
802084	231494	0.29

Tabulka 8.11: Analýza referencí jmenného prostoru `com.intellij`.

V tabulce 8.12 se nacházejí informace o třídách s nejvyšším počtem duplicitních instancí. Uvedeno je pouze prvních deset tříd, které lze opět klasifikovat jako třídy datové. V tabulce 8.13 je prvních deset největších clusterů. Na fragmentu 8.6 jsou k vidění některé instance těchto clusterů. Atribut

[5] `https://www.jetbrains.com/idea/`

typu `Object` s hodnotou 0 představuje null referenci. Jedná se o případ, kdy všechny atributy instance mají výchozí hodnotu, což je jedna z častých příčin duplicitních instancí.

Název třídy	Clusterů	Duplicit
XmlTextImpl	156	15493
SmartList	402	7847
XmlTagImpl	3	7230
XmlAttributeImpl	2	3631
FixedHashMap	1	2716
ScaledIconCache	1	2534
XmlTagImplDelegate	2	1630
LockFreeCopyOnWriteArrayList	1	1627
XmlExtensionAdapter	23	1614
LocalInspectionEP	15	1519

Tabulka 8.12: Analýza IntelliJ IDEA: Třídy a jejich clustery.

Cluster	Duplicit
XmlTextImpl	14700
XmlTagImpl	5600
XmlAttributeImpl	3626
SmartList	2982
FixedHashMap	2716
ScaledIconCache	2534
LockFreeCopyOnWriteArrayList	1627
XmlTagImpl	1617
XmlTagImplDelegate	1617
XmlExtensionAdapter	1469

Tabulka 8.13: Analýza IntelliJ IDEA: Jednotlivé clustery.

```
id: 32730291760
class: com.intellij.psi.impl.source.xml.XmlTextImpl
  myGapPhysicalStarts : Object = ‘0‘
  myDisplayText : Object = ‘0‘
  myGapDisplayStarts : Object = ‘0‘

id: 32730292120
class: com.intellij.psi.impl.source.xml.XmlTagImpl
  myHC : int = ‘0‘
  myImpl : Object = ‘0‘
  myAttributes : Object = ‘0‘
  myValue : Object = ‘0‘

id: 32728161952
class: com.intellij.psi.impl.source.xml.XmlAttributeImpl
  myHC : int = ‘0‘
  myImpl : Object = ‘0‘
```

Fragment kódu 8.6: Příklady instancí s duplicitním obsahem.

8.3 Zhodnocení výsledků

Původní a nová verze byly nejdříve otestovány na testovacích aplikacích s uměle vytvořenými duplicitami. Jednalo se o případy mělké duplicity, hluboké duplicity a duplicity primitivních typů. Nová verze nástroje odhalila všechny případy duplicit, zatímco původní verze byla schopna odhalit pouze případ mělké duplicity.

Obě verze byly dále puštěny na heap dumpech jednoduché aplikace vytvořené v aplikačním rámci Spring Boot. Nová verze našla téměř třikrát více duplicit a byla přitom výrazně rychlejší. Otestování na této reálné aplikaci potvrzuje fakt, že nová verze nástroje umí odhalit výrazně více duplicit. Jednalo se především o instance datových tříd.

Nová verze nástroje analyzovala heap dumpy aplikací Automatic Preventive Maintenance a IntelliJ IDEA. Výsledky analýzy první zmíněné aplikace neodhalily žádný případ duplicity. Tyto výsledky poukázaly na nevhodnost použití analýzy duplicit na aplikace, které pracují především s daty pocházejími z databáze. Analýza duplicit heap dumpu vytvořeného z vývojového prostředí IntelliJ IDEA trvala téměř 15 minut. Dlouhá doba analýzy vychází z principu hlubokého porovnání a je třeba s ní počítat. Analýza odhalila, že duplicity tvoří až 16% všech instancí. Duplicitní instance jsou reálným problémem, jehož vyřešením lze ušetřit nemalé množství paměti.

9 Závěr

První kapitola této diplomové práce stručně popisuje strukturu paměti virtuálního stroje JVM a její správu. Nejdříve je čtenář seznámen s virtuálním strojem JVM a jeho specifikací. Dále jsou shrnuty jednotlivé oblasti paměti, které se během běhu programu používají, a s automatickou správou paměti ve formě kolektorů.

Další dvě kapitoly se zabývají plýtváním paměti a jejími úniky. Jsou zde popsány jednotlivé příklady doprovozené fragmenty kódu, které ukazují příčiny těchto neefektivit. Část kapitoly se věnuje neefektivnímu využití paměti kolekcí a instancí, které vznikají příliš vysokou abstrakcí objektového modelu. Jedním z popsaných problémů plýtvání paměti je duplicita objektů, jejíž detekce byla hlavním cílem této diplomové práce.

Dále jsou popsány některé známé nástroje, které se používají k analýze problémů v Java aplikacích. Nástroje jsou stručně představeny s jejich uživatelskými rozhraními. Je zde popsán i nástroj Memory Analyzer, který se během této práce dále rozšiřoval. Nedostatky tohoto nástroje, mezi které patří například neschopnost porovnávat primitivní datové typy a rodičovské atributy, jsou popsány v následující kapitole. V této kapitole je dále navržen způsob hledání duplicit pomocí hlubokého porovnání instancí a analýza referencí.

Všechny nedostatky nástroje, které byly v této práci zmíněny, byly opraveny. Nástroj byl dále rozšířen o analýzy hluboce duplicitních instancí a referencí. Analýza duplicitních instancí byla optimalizována tak, aby byla použitelná na reálných aplikacích, které mají řádově statisíce instancí. Byl implementován interaktivní shell, ve kterém si může uživatel prohlížet stav jednotlivých instancí.

Nástroj byl otestován na testovacích aplikacích s uměle vytvořenými problémy. Rozšířený nástroj všechny problémy úspěšně odhalil. Dále byl otestován na reálných aplikacích, ve kterých odhalil velké množství duplicitních instancí. V případě analýzy vývojového prostředí IntelliJ IDEA tvořily duplicity téměř 16% všech instancí.

Literatura

[1] Altvater, A. *What is Java Garbage Collection?* [online]. 11.5.2017. [cit. 16.3.2021]. Dostupné z: https://stackify.com/what-is-java-garbage-collection/.

[2] baeldung. *Understanding Memory Leaks in Java* [online]. [cit. 20.2.2021]. Dostupné z: https://www.baeldung.com/java-memory-leaks.

[3] Bailey, C. *JavaOne 2013: Memory Efficient Java* [online]. [cit. 6.3.2021]. Dostupné z: https://www.slideshare.net/cnbailey/memory-efficient-java.

[4] Bližňák, J. Heuristika pro výchozí nastavení správce paměti Shenandoah [online]. Master's thesis, Masarykova unverzita, Fakulta informatiky, Brno, 2017. Dostupné z: https://is.muni.cz/th/bgcpq/.

[5] Bloch, J. *Effective Java.* Addison-Wesley, 3 edition, 2018. ISBN 978-0-13-468599-1.

[6] Bloch, J. *Komentář k `ThreadLocal` od Joshua Blocha* [online]. [cit. 8.3.2021]. Dostupné z: http://jsr166-concurrency.10961.n7.nabble.com/Threadlocals-and-memory-leaks-in-J2EE-td3960.html#a3984.

[7] Chis, A. E. et al. Patterns of Memory Inefficiency. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, s. 383–407, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642226540.

[8] Dmitriev, M. *Duplicate Objects in Java: Not Just String* [online]. [cit. 25.2.2021]. Dostupné z: https://dzone.com/articles/duplicate-objects-in-java-not-just-strings.

[9] Duigou, M. *RFR JDK-7143928 : (coll) Optimize for Empty ArrayList and HashMap* [online]. [cit. 6.3.2021]. Dostupné z: http://mail.openjdk.java.net/pipermail/core-libs-dev/2013-April/015585.html.

[10] FreePascal. *UCSD Pascal* [online]. [cit. 9.3.2021]. Dostupné z: https://wiki.freepascal.org/UCSD_Pascal.

[11] Gosling, J. – Joy, B. – Steele, G. *The Java Language Specification.* Addison Wesley, 09 2018.

[12] Guava. *Zdrojový kód* `Interners` [online]. [cit. 5.3.2021]. Dostupné z: https://github.com/google/guava/blob/master/guava/src/com/google/common/collect/Interners.java.

[13] Guava. *Dokumentace* `Interner` [online]. [cit. 5.3.2021]. Dostupné z: https://guava.dev/releases/22.0/api/docs/com/google/common/collect/Interner.html.

[14] Hordejčuk, V. *GoF: Flyweight (muší váha)* [online]. [cit. 21.2.2021]. Dostupné z: https://voho.eu/wiki/flyweight/.

[15] Hordejčuk, V. *Visitor (náštěvník)* [online]. [cit. 21.4.2021]. Dostupné z: http://voho.eu/wiki/visitor/.

[16] Junit. *JUnit 4 About* [online]. [cit. 24.4.2021]. Dostupné z: https://junit.org/junit4/index.html.

[17] JXRay. *JXRay Overview* [online]. [cit. 28.3.2021]. Dostupné z: https://jxray.com.

[18] Kawachiya, K. – Ogata, K. – Onodera, T. Analysis and Reduction of Memory Inefficiencies in Java Strings. *SIGPLAN Not.* October 2008, 43, 10, s. 385–402. ISSN 0362-1340. doi: 10.1145/1449955.1449795. Dostupné z: https://doi.org/10.1145/1449955.1449795.

[19] Kotlin. *Kotlin docs* [online]. [cit. 17.4.2021]. Dostupné z: https://kotlinlang.org/docs/home.html.

[20] Kotlin. *Multiplatform programming* [online]. [cit. 17.4.2021]. Dostupné z: https://kotlinlang.org/docs/multiplatform.html.

[21] Lee, S. *Mitigating the performance impact of memory bloat.* PhD thesis, Georgia Institute of Technology, 2015.

[22] Mach, M. Nástroj pro analýzu Java memory heap. Master's thesis, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Plzeň, 2019. Dostupné z: https://dspace5.zcu.cz/handle/11025/37421.

[23] Mitchell, N. – Schonberg, E. – Sevitsky, G. Four Trends Leading to Java Runtime Bloat. *IEEE Software.* 2010, 27, 1, s. 56–63. doi: 10.1109/MS.2010.7.

[24] Mitchell, N. – Sevitsky, G. The Causes of Bloat, the Limits of Health. *SIGPLAN Not.* October 2007, 42, 10, s. 245–260. ISSN 0362-1340. doi: 10.1145/1297105.1297046. Dostupné z: https://doi.org/10.1145/1297105.1297046.

[25] OpenJDK. *GitHub, Implementace `ArrayList` Java 7* [online]. [cit. 6.3.2021]. Dostupné z: https://github.com/openjdk/jdk7/blob/master/jdk/src/share/classes/java/util/ArrayList.java#L139.

[26] OpenJDK. *GitHub, Implementace `ArrayList` Java 8* [online]. [cit. 6.3.2021]. Dostupné z: https://github.com/openjdk/jdk8u/blob/master/jdk/src/share/classes/java/util/ArrayList.java#L165.

[27] Oracle. *Arrays* [online]. [cit. 7.4.2021]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html.

[28] Oracle. *Concurrent Mark Sweep (CMS) Collector* [online]. [cit. 17.3.2021]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html.

[29] Oracle. *HashMap* [online]. [cit. 25.2.2021]. Dostupné z: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

[30] Oracle. *HotSpot JVM* [online]. [cit. 9.3.2021]. Dostupné z: https://www.oracle.com/java/technologies/whitepaper.html.

[31] Oracle. *About Java Flight Recorder* [online]. [cit. 23.3.2021]. Dostupné z: https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170.

[32] Oracle. *JDK Mission Control* [online]. [cit. 22.3.2021]. Dostupné z: https://www.oracle.com/java/technologies/jdk-mission-control.html.

[33] Oracle. *The Java Virtual Machine specification* [online]. [cit. 22.3.2021]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html.

[34] Oracle. *The Java Virtual Machine* [online]. [cit. 9.3.2021]. Dostupné z: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html.

[35] Oracle. *Chapter 5. Conversions and Promotions* [online]. [cit. 5.3.2021]. Dostupné z: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.1.7.

[36] Oracle. *The try-with-resources Statement* [online]. [cit. 1.3.2021]. Dostupné z: https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

[37] Oracle. *Java VisualVM* [online]. [cit. 23.3.2021]. Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/.

[38] Paul, J. *Tomcat - Memory leak* [online]. [cit. 8.3.2021]. Dostupné z: https://javarevisited.blogspot.com/2012/01/tomcat-javalangoutofmemoryerror-permgen.html.

[39] Project, A. M. *Welcome to Apache Maven* [online]. [cit. 17.4.2021]. Dostupné z: https://maven.apache.org.

[40] Shipilev, A. *Java Objects Inside Out* [online]. [cit. 5.3.2021]. Dostupné z: https://shipilev.net/jvm/objects-inside-out/#_observation_compressed_references_affect_object_header_footprint.

[41] Spring. *Kotlin support* [online]. [cit. 17.4.2021]. Dostupné z: https://docs.spring.io/spring-boot/docs/2.0.x/reference/html/boot-features-kotlin.html.

[42] Szudzik, M. *An Elegant Pairing Function* [online]. [cit. 19.4.2021]. Dostupné z: http://szudzik.com/ElegantPairing.pdf.

[43] Thon, L. *Správa paměti na platformě Java* [online]. [cit. 16.3.2021]. Dostupné z: https://courseware.zcu.cz/CoursewarePortlets2/DownloadDokumentu?id=3692.

[44] Ungar, D. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Not.* April 1984, 19, 5, s. 157–167. ISSN 0362-1340. doi: 10.1145/390011.808261. Dostupné z: https://doi.org/10.1145/390011.808261.

[45] Venners, B. *Inside of the Java Virtual Machine* [online]. [cit. 10.3.2021]. Dostupné z: https://www.artima.com/insidejvm/ed2/jvm3.html.

Seznam obrázků

Seznam tabulek

Seznam fragmentů kódů

A Seznam zkratek

JVM Java Virtual Machine

GC Garbage collection

JMC JDK Mission Control

JFR Java Flight Recorder

RUM Real User Monitoring

APM Automatic Performance Monitoring

B Obsah přílohy

Soubor s přílohami ve formátu ZIP je rozdělen do následujících poadresářů:

`Text_prace`

Adresář obsahuje zdrojový text této diplomové práce v jazyce TEX. Dále se zde nachází obrázky ve formátu `png` a soubor s referencemi ve formátu `bib`.

`Poster`

Adresář obsahuje poster diplomové práce včetně zdrojového souboru ve formátu `pub`.

`Aplikace_a_knihovny`

Adresář obsahuje projekt nástroje Memory Analyzer.

`Vstupni_data`

Adresář se soubory ve formátu `hprof`, které lze použít k vyzkoušení funkcionalit nástroje Memory Analyzer.

`Vysledky`

Adresář s výstupy nástroje Memory Analyzer, které jsou podrobně popsány v kapitole Výsledky.

`Readme.txt`

Soubor s detailním popisem aktuální adresářové struktury.

C Uživatelská příručka

Nástroj Memory Analyzer je napsán v jayzce Java. Ke spuštění a přeložení aplikace je třeba verze jazyka 8 nebo vyšší.

Struktura projektu

Projekt je složen z následujících adresářů a souborů:

`analyzer`

Adresář s knihovnou `analyzer` sloužící k analýze neefektivního využití paměti.

`app`

Adresář s konzolovou aplikací `app` pro provádění analýzy neefektivního využití paměti.

`sandbox`

Adresář obsahuje především testovací soubory ve formátu `hprof`, které jsou třeba při regresních testech. Dále se zde nachází modul `example-app`. Jedná se o konzolovou aplikaci, která slouží ke generování příkladů neefektivního využití paměti.

`experiments`

Adresář s výstupy nástroje při jednotlivých experimentech.

`pom.xml`

V tomto souboru jsou uvedeny výše uvedené submoduly `analyzer`, `app` a `example-app`. Nástroj Maven potřebuje tento soubor k sestavení projektu.

`README.md`

Soubor s uživatelskou příručkou v angličtině.

Kompilace projektu

K sestavení projektu je třeba nástroj Maven verze 3.6 nebo vyšší. Projekt se zkompiluje spuštěním následujícího příkazu v kořenovém adresáři projektu:

```
mvn clean package
```

Zkompilováním se vytvoří odpovídající soubory ve formátu `jar`, které se nachází v adresáři `target` jednotlivých submodulů. Při kompilaci se automaticky spouští vytvořené jednotkové, integrační a regresní testy.

Spuštění testů

Vytvořené testy lze samostatně spustit pomocí následujícího příkazu v kořenovém adresáři projektu

```
mvn clean test
```

Spuštění nástroje

Jedná se o konzolovou aplikaci, kterou lze konfigurovat následujícími parametry:

- `-p/--path` — Parametr očekává cestu k souboru ve formátu `hprof`.
- `-l/--list` — Nástroj vypíše všechny jmenné prostory nacházející se ve specifikovaném heap dumpu.
- `-n/--namespace` — Parametr očekává název jmenného prostoru, který se následně bude analyzovat. Parametr spouští analýzu.
- `-e/--exclude` — Parametr očekává název jmenného prostoru, který se nebude analyzovat. Parametr spouští analýzu.
- `-f/--fields` — Nástroj při výpisu uvede i jednotlivé atributy instancí.
- `-v/--verbose` — Nástroj při výpisu uvede identifikátory jednotlivých instancí clusteru, max. 10.
- `-c/--csv` — Výpis se uloží do souboru `results.csv` ve formátu `csv`.
- `-h/--help` — Výpíše nápovědu.

- `-i/--interactive` — Nástroj bude fungovat v interaktivním režimu.

Následující příkaz ukazuje, jak provést analýzu jmenného prostoru `cz.hoang` nad souborem `heap_dump.hprof`:

```
java -jar app.jar -p heap_dump.hprof -n cz.hoang
```

Další příklady, jak lze spouštět nástroj, se nachází ve výše zmíněném souboru `README.md`.