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

Diplomová práce

Nástroj pro analýzu Java memory heapu

Plzeň 2019 Martin Mach

PROHLÁŠENÍ

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

V Plzni dne 24. března 2019 . . . . . .

Martin Mach

ABSTRAKT

Abstrakt CZ

KLÍČOVÁ SLOVA

Klíčová slova CZ

ABSTRACT

Abstract EN

KEYWORDS

Keywords EN

MACH, Martin *Nástroj pro analýzu Java memory heapu*: diplomová práce. Plzeň: Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2019. 22 s. Vedoucí práce byl Ing. Richard Lipka, Ph.D.

OBSAH

1 ÚVOD

Přes to, že cena hardware klesá a mzdy programátorů rostou, je často nutné přistoupit k optimalizaci existujícího programu, namísto prostého přidání systémových prostředků [1][2]. Optimalizace může probíhat na několika úrovních. První z možností může být volba efektivnějšího algoritmu – můžeme zkontrolovat, zda Big O[1] složitost kritických částí našeho programu dosahuje maximálně polynomu přijatelného stupně, namísto např. exponenciály či faktoriálu. Tím je možné, především u vyššího počtu zpracovávaných prvků, dosáhnout výrazně rychlejšího běhu programu samotného.

Dále můžeme optimalizovat na úrovni využití systémových prostředků, typicky paměti (RAM). Jako první můžeme zvážit používané technologie. Změna programovacího jazyku je často natolik drahá, že kompletní přepsání dotyčného software by nemuselo dávat, i přes dosaženou optimalizaci, ekononomický smysl. V rámci námi používaného jazyku tak můžeme, kromě našeho kódu samotného, analyzovat používané frameworky a knihovny. V praxi se můžeme setkat s tím, že závislost na knihovně je přidána pouze z důvodu využití jedné či několika jejích funkcionalit. To je vhodné během vývoje z důvodu rychlosti; později je možné některé z těchto knihoven a jimi poskytované funkce nahradit implementací vlastní a teoreticky tak ušetřit jednotky, desítky či stovky megabytů paměti.

V praxi mohou být následky nedůsledné optimalizace ještě vážnější. Systémy, které mají obsloužit vysoký počet požadavků za sekundu, je často nutné škálovat – vytvářet nové, nezávislé jednotky těcho systémů. Ty se poté mohou střídat o příchozí požadavky. Každá neoptimalizace je tedy znásobena počtem těchto instancí. To může naprosto zbytečně zvyšovat náklady na provoz; a to ať už v případě vlastního serveru či cloudových služeb (tzv. serverless).

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

[1] Big O složitost je...

2 PROBLÉM SPRÁVY PAMĚTI

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

2.1 Garbage collector

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

Obecně GC funguje tak, že si udržuje seznam referencí na jednotlivé objekty, respektive jejich počet. Pokud je některý z objektů dále nereferencovaný (a tedy nedosažitelný – nikdo na něj neukazuje; Java navíc nezná koncept ukazatelové aritmetiky, po odstranění poslední reference si tedy můžeme být jisti, že už znovu referencovat nikdy nepůjde), při další kolekci bude jím zabíraná paměť uvolněna.

3 ANALÝZA PAMĚTI

4 OPTIMALIZACE UŽITÍ PAMĚTI

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

Můžeme tedy omezit samotné využití objektů – zvážit, zda je potřebujeme, případně zda neexistuje vhodnější způsob či struktura pro jejich uchovávání. Jejich alokaci v paměti bychom rovněž meli omezit na nejkratší možnou dobu, po kterou si budeme jisti jejich využitím. Nicméně i zde je vhodné najít vhodný poměr mezi optimalizací a zbytečným úklidem objektu, který za několik okamžiků znovu budeme vytvářet.

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

5 JAVA VIRTUAL MACHINE

Program napsaný v Javě běží typicky v některé z implementací Java Virtual Machine (JVM). JVM je program, který slouží jako běhové prostředí pro uživatelský kód – vykonává jeho instrukce a slouží tak jako prostředník mezi ním a operačním systémem (respektive jako interpret jeho kódu, který následně překládá do jiného jazyka, typicky strojového kódu dané architektury či platformy).

Některé implementace JVM umožňují přímý překlad do strojového kódu bez potřeby interpretace, např. GraalVM [3]. V takovém případě hovoříme o tzv. Ahead-Of-Time (AOT) přístupu, místo Just-In-Time (JIT) postupu implementovaného v moderních verzích častěji používaných JVM, např. HotSpot od společnosti Oracle [TODO zdroj, že HS používá JIT].

5.1 Java Bytecode

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

6 SPRÁVA A STRUKTURA JAVA PAMĚTI

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

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

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

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

6.1 Garbage collection

Spuštění GC v Javě nelze vynutit ručně. Systému lze *doporučit* jeho exekuci voláním metody `System.gc()`; JVM se tím ale nemusí řídit a toto volání jednoduše ignorovat [7].

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

Ve spojení s GC by tedy nemělo k únikům paměti typicky dojít. V Javě k nim může dojít především při nedůsledném používání vlastních zavaděčů tříd – *class loaderů*. Za únik paměti můžeme rovněž považovat neuzavřený popisovač otevřeného souboru, databáze či

jiného zdroje. Pokud k němu ztratíme přístup, např. po vyhození výjimky bez uzavření tohoto popisovače v bloku `catch` či lépe `finally`, ztrácíme tím, spolu s popisovačem, i menší množství paměti. Při častějším výskytu problému ale v tomto případě pravděpodobně narazíme na horní limit popisovačů Javy či operačního systému – ani tento zdroj není neomezený.

TODO popis různých implementací GC v Javě?

6.2 Heap Dump

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

6.2.1 Vytvoření dumpu

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

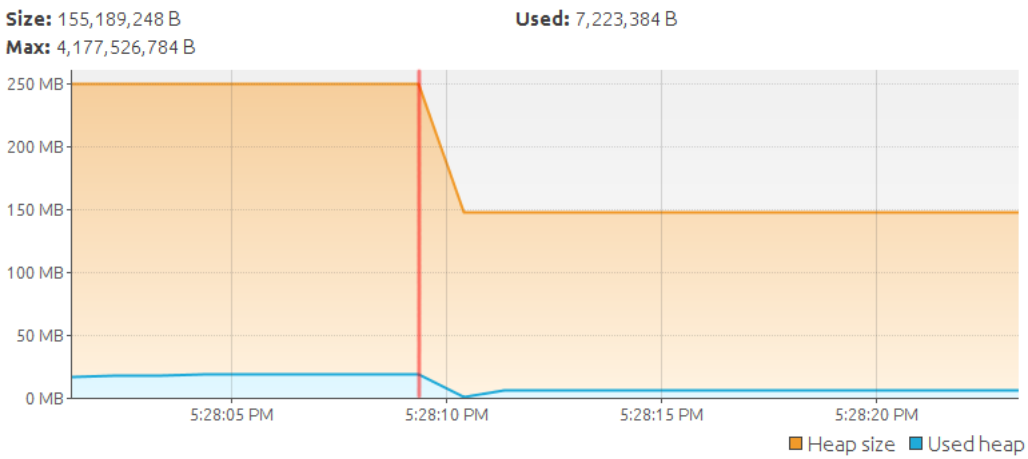
Mezi další způsoby vytvoření dumpu patří různé nástroje, debuggery a profilery typu Eclipse MAT, VisualVM nebo Java Mission Control (viz dále). Výhodou těchto nástrojů je, že dokáží zvládnout vytváření dumpu i napříč verzemi a dokonce implementacemi (z Oracle JDK na OpenJDK apod.)

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

6.2.2 Vztah dumpu vůči paměti procesu

Heap Dump přímo odpovídá části paměťového prostoru procesu, resp. heapu. Je tedy přímým otiskem části fyzické paměti tak, jak je uložena, v určitém časovém okamžiku. Toto je možné si experimentálně ověřit – jak bylo zmíněno výše, z otisku fyzické paměti

je možné heap dump získat. Je tedy evidentní, že při jeho vytvoření některým z výše uvedených způsobů nedochází k žádným úpravám a snímek je vytvořen „tak jak je“.



Obr. 1: Využití paměti a provedení garbage kolekce před vytvořením dumpu (červeně).

6.3 Zpracování dumpu

Při zpracování dumpu je nutné zohlednit fakt, že je vyexportovaný kompletní paměťový prostor [TODO zdroj] a nachází se zde tedy i data objektů, které nás nutně nemusí zajímat – typicky knihovny nebo objekty Javy. Toto je možné zohlednit a filtrovat na základě jmenného prostoru (namespace), do kterého objekt patří.

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

7 EXISTUJÍCÍ NÁSTROJE PRO ANALÝZU HEAPU

7.1 Eclipse MAT

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

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

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

7.2 VisualVM

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

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

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

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

7.3 Java Mission Control

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

7.4 JProfiler

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

7.5 JHAT

Nástroj, který je přímou součásti distribuce Oracle JVM od verze 6. Nebyl nikdy oficiálně podporován a od počátku byl označen jako experimentální nástroj, z těchto důvodů byl tedy v Javě 11 naprosto odstraněn [8][9]. V rámci verzí Javy, které jej obsahují, ho lze využít jako Command Line Interface (CLI) aplikaci, která dokáže dump vytvoření pomocí např. *JMAP* otevřít. Následně vytvoří webový server, jehož prostřednictvím poskytuje data získaná ze zpracovaného dumpu. Tato data je možné si poté zobrazit pomocí webového prohlížeče; zajímavým příkladem možného využití je následné rozparsování těchto dat jakožto formátu HTML a jejich další využití. Program je tedy možné využít jako prostředníka pro zpracování [10]. Kromě „prostého“ zobrazení webové stránky umí *JHAT* rovněž poskytovat zpracování pomocí jazyka OQL.

7.6 OQL

OQL zmíním i jako samostatný způsob zpracování. Jedná se o jazyk, který slouží pro obecnou manipulaci s objekty, resp. s objektovými dokumenty [11][12]. Na první pohled si nelze nevšimnou jeho podobnosti s dotazovým jazykem SQL. Neomezuje se tedy pouze na zpracování Java heapu (či obecně paměti), ale je standardem, který pro tento účel lze využít. Z toho je možné usoudit, že standard jako takový pro zpracování nestačí – je nutné využít některou z jeho implementací, která takové rozhraní přístupu k Java heapu umožňuje. Jednou z nich je právě výše zmíněný *JHAT*.

V příkladu

Ukázka 7.1: Příklad OQL

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

8 MOŽNOSTI ANALÝZY

Rovnost dvou či více objektů se dá definovat a zjišťovat různými způsoby. Je ale nutné si uvědomit, že v nejhorším případě, tj. pokud chceme najít všechny nadbytečné kopie každého objektu, je složitost této operace při nejmenším $O(n^n)$. Bylo by tedy vhodné se zamyslet, zda neexistuje způsob, jak počet porovnání snížit, případně navrhnout jednoduchou heuristiku, která by dokázala rychle ohodnotit, zda má vůbec smysl pokračovat v podrobnějším porovnání. V následujících případech tedy uvažujme objekty A, B, jejich třídy C_A a C_B a proměnné obou instancí $F_A^0..F_A^n$, respektive $F_B^0..F_B^n$.

První nápovědou samozřejmě může být porovnání tříd obou objektů – C_A a C_B. Pokud platí $C_A = C_B$, zřejmě má smysl se porovnáváním dále zabývat. V případě jejich nerovnosti není ale možné další porovnání zavrhnout, porovnat je nutné (či možné) i jejich rodiče. Definujeme-li tedy funkci pro zjištění přímého rodiče $P(C)$, potom rovnost objektů lze vyjádřit jako

$$E(C_A, C_B) \Leftrightarrow C_A = C_B \vee E(P(C_A), C_B) \vee E(C_A, P(C_B)).$$

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

$$E(C_A, C_B) = P_c(C_A) \cap P_c(C_B) \notin \emptyset,$$

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

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

9 ANALÝZA A IMPLEMENTACE

SEZNAM OBRÁZKŮ

SEZNAM ZKRATEK

JVM Java Virtual Machine

AOT Ahead-Of-Time

JIT Just-In-Time

GC Garbage Collector

PID Process Identifier – Identifikátor procesu

CLI Command Line Interface

OQL Object Query Language

LITERATURA

[1] TODO Citace Moore's law

[2] TODO Citace mzdy programátorů

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

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

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

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

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

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

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

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

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

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

SEZNAM PŘÍLOH

A PŘÍLOHY