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

Diplomová práce

Výkonnostní a paměťová optimalizace nástroje JaCC

Plzeň 2016 Jan Ambrož

Prohlášení

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

V Plzni dne 12. května 2016

Jan Ambrož

Poděkování

Rád bych touto cestou poděkoval svému vedoucímu diplomové práce, panu Ing. Kamilu Ježkovi, Ph.D., za odborné vedení, podnětné rady a vstřícnost při zpracování této práce.

Abstract

The subject of this thesis is the static code analysis tool Java Class Comparator, JaCC in short. This tool can verify binary and source compatibility of component applications implemented in Java programming language. The imperfection is its high memory requirements. The main purpose of this thesis is to optimize both memory requirements and performance of this tool.

The JaCC tool progressively creates and keeps big amount of data structures during processing which leads to big memory consumption. In order to optimize the JaCC tool integration of Ehcache libraray was performed. This library provides resources to enhance storage space with another storage level - hard drive memory. With these resources physical memory can act as a cache and can store only limited data structure work set.

The experiments show that the optimization has helped to reduce memory consumption by 25% - 45%. These experiments also confirm the theory that the memory saving rate directly depends on the particular use case of the JaCC tool. The value of this rate is inversely related to the share of data structures which were not the subject of this optimization. The side effect of this optimization is the lenghtening of the average run time up to three times. The main reason behind this is the usage of the hard drive as another storage level with higher access time. Since the optimized solution is fully configurable we can seek for the best compromise between the rate of saved memory and the solution's run time.

Abstrakt

Předmětem této práce je nástroj pro statickou analýzu kódu Java Class Comparator, zkráceně JaCC. Tento nástroj umožňuje ověřovat zdrojovou i binární kompatibilitu komponentových aplikací implementovaných v programovacím jazyce Java. Nedokonalostí tohoto nástroje jsou ovšem jeho vysoké paměťové požadavky. Hlavním cílem této práce je proto optimalizace tohoto nástroje, a to jak z hlediska jeho výkonnosti, tak především z hlediska jeho paměťové náročnosti.

Nástroj JaCC při svém zpracování postupně vytváří a uchovává velké množství datových struktur, které jsou příčinou jeho vysokých paměťových požadavků. V rámci optimalizace nástroje JaCC byla provedena integrace knihovny Ehcache, která nabízí prostředky pro rozšíření úložného prostoru o paměť pevného disku. S využitím těchto prostředků potom může fyzická paměť fungovat jako mezipaměť a může uchovávat jen omezenou pracovní množinu datových struktur.

Experimenty ukazují, že optimalizace nástroje přinesla úsporu paměti v rozmezí 25% - 45%. Tyto experimenty také potvrzují hypotézu, že je míra úspory paměti závislá na konkrétním případu užití nástroje a její hodnota je nepřímo úměrná na podílu datových struktur, které nebyly předmětem optimalizace. Vedlejším efektem této paměťové optimalizace je na druhé straně prodloužení průměrné délky doby běhu na 2 až 3 násobek původní délky doby běhu. Příčinou je použití pevného disku jako dodatečné úložní vrstvy, ovšem s vyšší přístupovou dobou. Jelikož je optimalizované řešení plně přizpůsobitelné, je možné hledat kompromis mezi mírou ušetřené paměti a délkou doby běhu.

Obsah

1 Úvod

V oblasti softwarového inženýrství sehrává dnes komponentově orientované programování svoji významnou roli. Rozdělení aplikace na jednotlivé funkční bloky se velice osvědčilo a víceméně se stalo standardizovanou metodikou programování. Jednotlivé komponenty se lépe spravují, udržují a díky jasně definovanému rozhraní garantují svoji funkčnost. Tento přístup s sebou ovšem přináší úskalí v podobě potenciálních závislostí mezi komponentami. Komponenty se mohou navzájem využívat nebo mezi sebou mohou komunikovat. Aktualizace některé z komponent pak může snadno ovlivnit funkcionalitu závisejících komponent a v konečném důsledku i funkcionalitu celé aplikace.

Nalezení a identifikace původce problému může být někdy složitý úkol. Proto se začaly objevovat nástroje, které těmto problémům měly pomáhat předcházet, nebo je alespoň dokázaly identifikovat v případě, že už nastaly. Jedním z takových nástrojů je i nástroj **JaCC** (**Ja**va **C**lass **C**omparator).

Tento nástroj je vyvíjen na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni a umožňuje ověřovat binární a zdrojovou kompatibilitu komponentových aplikací. Zjednodušeně nástroj funguje tak, že načte jednotlivé komponenty a vytvoří jejich virtuální reprezentaci v paměti, se kterou dále pracuje. Načtené aplikace ovšem mohou být velmi objemné a jejich velikost se pak odráží i na množství využité paměti. V případě aplikací o velikosti v řádu stovek MB již může být průměrná uživatelská velikost paměti 8GB nedostačující.

Cílem této práce je seznámit se s nástrojem JaCC, podrobněji analyzovat jeho implementaci, nalézt prostor pro zlepšení a nakonec tento nástroj optimalizovat. Předmětem optimalizace by měly být především jeho paměťové požadavky, které jsou pro jeho reálné použití u větších aplikací značně limitující.

První kapitola seznamuje čtenáře s technologiemi, jejichž základní znalost je nutnou podmínkou pro pochopení problematiky této diplomové práce. Druhá kapitola pojednává o nástroji JaCC a podrobněji popisuje princip jeho funkčnosti. Třetí kapitola se věnuje vlastnímu procesu optimalizace nástroje JaCC. V závěru je zhodnocen výsledek této práce a je uvedeno možné rozšíření nástroje.

2 Technologie

2.1 Java a správa paměti

Programovací jazyk Java patří mezi tzv. vyšší programovací jazyky s **automatickou správou paměti**. Programátor je tedy zodpovědný pouze za to, kdy bude objekt vytvořen. O vše ostatní, co souvisí s vytvořením a životním cyklem objektu v programovacím jazyce Java, se stará sám programovací jazyk (konkrétně **JVM**). Automatická správa paměti sama zajistí alokaci potřebného množství paměti pro nově vytvořený objekt a naopak sama rozhodne o tom, kdy je daný objekt již nepotřebný a je možné uvolnit stejné množství prostředků. Výhody a nevýhody tohoto přístupu jsou následující:

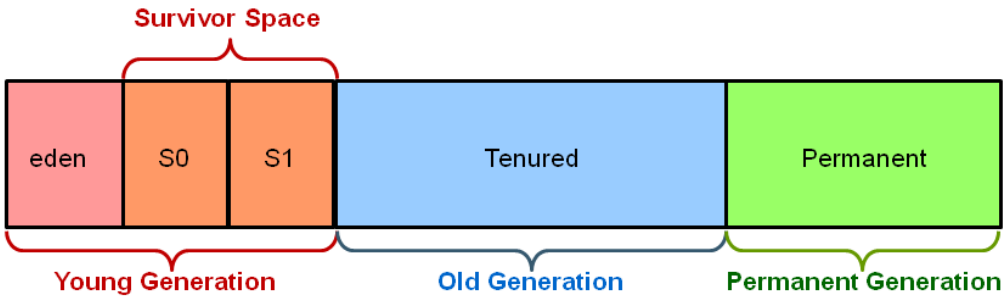
- **výhody**
  - \+ Programátor se může více soustředit na řešení skutečného problému.
  - \+ Nastává menší množství chyb spojených s přístupem do paměti.
  - \+ Zdrojový kód je lépe čitelný.
- **nevýhody**
  - – Programátor nemá plně pod kontrolou využití paměti.
  - – Automatická správa paměti bývá spojena s určitou režií.
  - – Dostupné objekty mohou být ve skutečnosti již nepotřebné.

2.1.1 Alokace paměti

V programovacím jazyce Java se alokace paměti pro nově vytvářený objekt děje současně s jeho inicializací. Nově vytvářené objekty se umisťují na heap do oblasti paměti zvané **Young Generation**. Tato oblast paměti slouží pro uchování objektů s krátkou životností. Pokud se tato oblast paměti naplní, dojde k tzv. *minor garbage kolekci* (viz dále) a nepotřebné objekty jsou z této oblasti paměti odstraněny. Protože se v Young Generation nachází převážné objekty s krátkou životností, dojde často k odstranění většiny z nich. Pokud ovšem doba života některého z objektů překročí určitý práh, je takový objekt přesunut do oblasti paměti Old Generation.

Old Generation slouží k uchování objektů s delší životností. Pokud dojde k naplnění i této oblasti paměti, spouští se tzv. *major garbage kolekce*. Protože je obyčejně tato oblast paměti větší a obsahuje více živých objektů, je doba běhu major garbage kolekce obvykle výrazně delší než u minor garbage kolekce.

Třetím typem oblasti paměti, kterou používá programovací jazyk Java, je **Permanent Generation**. V této oblasti se nachází metadata, které JVM potřebuje pro popis tříd a metod využívaných aktuálně spuštěnou aplikací. Tato oblast paměti se plní za běhu aplikace na základě použitých tříd v aplikaci. V této části paměti se mohou také nacházet třídy a metody Java SE [3].



Obrázek 2.1: Stuktura heap v Javě

Velikost paměti heap či velikost jednotlivých jejích podoblastí lze specifikovat před startem aplikace pomocí parametrů JVM [1]:

- **-Xms, -Xmx**

  Slouží pro nastavení počáteční (`-Xms`) a maximální (`-Xmx`) velikosti heap. Např: `-Xms512m -Xmx4g` vytvoří při spuštění heap o počáteční velikosti 512 MB a dovolí zvýšení jeho velikosti až na 4 GB.

- **-XX:NewSize, -XX:MaxNewSize**

  Slouží pro specifikaci počáteční velikosti (`-XX:NewSize`) a maximální velikosti (`-XX:MaxNewSize`) oblasti paměti Young Generation.

- **-XX:NewRatio**

  Slouží pro určení poměru mezi Young Generation a Old Generation. Pokud je tedy například `-XX:NewRation=3`, je poměr mezi Young Generation a Old Generation 1 ku 3.

2.1.2 Dealokace paměti

Při naplnění Young Generation se provádí **minor gargage kolekce**, při naplnění Old Generation se provádí **major garbage kolekce** a pokud dojde nakonec k naplnění i celé paměti heap, provádí se **full garbage kolekce**. Všechna vlákna, kromě toho, které provádí garbage kolekci, se pozastaví - tento okamžik bývá označován jako **stop-the-world**[3]. Vlákno provádějící garbage kolekci uvolní paměť alokovanou dále nepotřebnými objekty a uvolní se tak místo v paměti pro nové objekty. Pozastavená vlákna poté pokračují dále v běhu od místa, na kterém se pozastavily.

Poznámka: Garbage kolekci lze dokonce programově vyvolat pomocí `System.gc()`. Nicméně toto volání slouží pouze jako nápověda pro garbage kolektor a není garantováno, že se garbage kolekce skutečně ihned spustí [2].

Mark and sweep algoritmus

Vlákno, které provádí garbage kolekci, používá algoritmus **Mark and sweep** pro nalezení a odstranění dále nepotřebných objektů z paměti [3].

Garbage kolekce probíhá ve dvou fázích:

1. **Mark fáze**

   Nejprve se určí tzv. *kořeny* garbage kolekce (*GC roots*). Těmito kořeny mohou být například lokální proměnné, aktivní vlákna či statické proměnné. V dalším kroku se z těchto kořenů spustí průchod grafem přes reference jednotlivých objektů a hledají se všechny dostupné objekty. Všechny nalezené objekty se označí jako *živé* (viz obrázek 2.2).

2. **Sweep fáze**

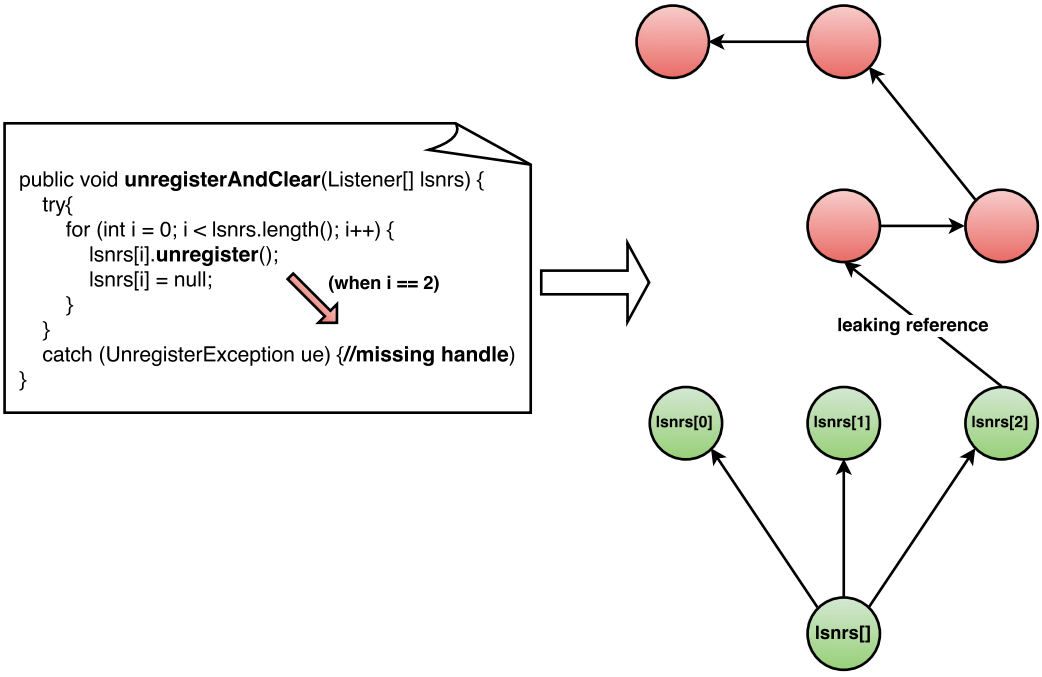
   Údaje o volných blocích paměti, které vznikly mezi objekty označenými jako živé, se uloží do tzv. *free listu*. Tento seznam se později použije při opětovné alokaci volné paměti. Fyzicky tedy nedochází k odstranění obsahu paměti.

Obrázek 2.2: Mark fáze mark-and-sweep algoritmu

Na základě znalosti algoritmu mark-and-sweep nyní víme, na co je třeba si dát ve zdrojovém kódu pozor. Jedná se o reference na dále nepotřebné objekty. Pokud totiž zapomeneme odstranit referenci na objekt, který již nebudeme potřebovat, garbage kolektor nedokáže žádným způsobem takový objekt odhalit. Výsledkem bude nadále zbytečně alokovaná paměť pro nepotřebný objekt. Pokud navíc takový objekt odkazuje na další objekty, které nejsou z žádného jiného živého objektu referencovány, může problém přerůst v daleko závažnější.

Následující obrázek 2.3 znázorňuje jeden z možných problémových scénářů zapomenuté reference. Jednotlivé kruhy znázorňují objekty držené v paměti a čáry mezi objekty představují referenci z jednoho objektu na druhý (ve směru šipky). Červeně označené kruhy reprezentují zbytečně držené objekty v paměti.

Ke vzniku zapomenuté reference může například dojít, pokud provádíme iteraci polem s prvky a nad každým prvkem před jeho odstraněním zavoláme metodu, která může skončit výjimkou. Pokud výjimka nastane a není ošetřena, všechny další prvky - včetně toho, který způsobil výjimku - v poli zůstanou.



Obrázek 2.3: Problém zapomenuté reference

2.1.3 Velikost objektu

Programovací jazyk Java nenabízí žádnou standardní konstrukci, pomocí které by bylo možné zjistit velikost předaného datového typu tak, jak je tomu například u programovacího jazyka C - konstrukce `sizeof`. Jednotlivé datové typy v Javě navíc ani nemusí mít pevnou velikost. Například datový typ `int` má sice 32 bitů, ale nikde už není podle specifikace řečeno, na kolik se má tento datový typ zarovnat ve fyzické paměti. Na 64 bitovém stroji může mít tedy datový typ `int` po zarovnání do fyzické paměti klidně i 64 bitů [4].

Přestože je určení velikosti objektu v Javě poměrně složitý úkol, existuje řada možných přístupů, jak velikost objektu zjistit, nebo alespoň přibližně odhadnout:

- **Serializace**

  Jedním z možných přístupů měření velikosti objektu je serializace objektu do datového proudu. Velikost objektu pak odpovídá délce datového proudu. Během serializace ovšem dochází k různým konverzím a datový proud tak přímo nemusí reflektovat skutečnou podobu objektu ve fyzické paměti. Například datový typ `char` má v Javě 2B. ASCII znaky se ovšem během serializace zakódují pomocí UTF-8 kódování a v datovém proudu bude jejich velikost jen 1B.

- **Násobná replikace**

  Dalším z možných přístupů je násobná replikace stejného objektu a současné sledování navýšení velikosti heapu v JVM. Tento přístup ovšem vyžaduje vytvoření velkého množství objektů a předpokládá navýšení heapu pouze jako důsledek vytvoření totožných objektů. Přístup nelze použít pro měření velkých objektů.

- **Reflexe**

  Reflexí se zde rozumí schopnost programovacího jazyka za běhu zjišťovat informace o určitém programovém objektu. Můžeme se tak dotázat na všechny nestatické atributy konkrétní třídy a stejnou akci provést rekurzivně i nad jejími jednotlivými atributy neprimitivních datových typů. Sečtením velikostí všech nalezených atributů primitivních datových typů nakonec dostaneme přibližný odhad velikosti zkoumaného objektu.

2.2 Sledování využití paměti

Předmětem této diplomové práce je optimalizace nástroje JaCC především z hlediska paměťové složitosti. Obecně samotné optimalizaci libovolné aplikace musí vždy předcházet analýza, která poukáže na její slabá místa. V první části této podkapitoly je proto čtenář seznámen s nástroji, které umožňují sledovat využití paměti běžících Java aplikací. Ve druhé části je pak popsán jiný alternativní způsob analýzy paměti.

2.2.1 Profilery

K dispozici je celá řada nástrojů, které umožňují monitorovat běžící Java aplikace. Tyto nástroje se označují jako tzv. **profilery** a mezi jejich hlavní funkce obvykle patří sledování množství využité paměti, zatížení procesoru a činnosti garbage kolektoru.

Tyto nástroje používají pro měření dva principiálně odlišné způsoby:

1. **Statické profilování**

   V pravidelných intervalech se zjišťuje, jaká část kódu se právě provádí. Na základě naměřených údajů se pak může vytvořit odhad toho, kolik času aplikace strávila vykonáváním jednotlivých bloků kódu.

   + Nedochází ke zpomalení běhu profilovaného programu.
   – Výsledky jsou nepřesné. Vykonávání krátkých bloků kódu nemusí být vůbec zachyceno.

2. **Instrumentace**

   Do kódu se přimíchají pomocné instrukce, které umožňují sledovat aktuálně zpracovávaný kód.

   + Výsledky jsou přesné. Vykonávání krátkých bloků kódu je zachyceno.
   – Je významně ovlivněn běh profilovaného programu - dojde ke značnému zpomalení jeho běhu.

Mezi nejpopulárnější a nepoužívanější Java profilery patří následující nástroje [5]

- **YourKit Java Profiler**

  Komerčně licencovaný nástroj, který umožňuje profilovat Java SE a Jave EE aplikace jak z hlediska využití paměti, tak z hlediska zatížení CPU.

  Oproti ostatním nástrojům disponuje nižší režií běhu a poskytuje vyšší výkon při tvorbě tzv. snapshotů - obraz analyzovaných dat aktuálně zpracovávané aplikace.

- **JProfiler**

  Opět komerčně licencovaný velmi kvalitní Java profiler, který je vyvíjen společností *ej-technologies GmbH* a je primární určen pro použití s Java EE a Java SE aplikacemi.

  Analyzuje aplikace z hlediska výkonnosti, dokáže nalézt úzká hrdla a memory leaky, umožňuje sledovat zatížení CPU a řešit problémy vícevláknových aplikací.

  Aplikace podporuje nejen lokální profilaci, ale i vzdálenou.

- **VisualVM**

  Tento nástroj je distribuován volně společně s *Java JDK*[1] balíkem.

  Umožňuje sbírat detailní informace o aplikacích běžících na JVM. Tyto aplikace mohou běžet jak lokálně, tak vzdáleně. Analyzovaná data je možné ukládat ve formě snapshotů a věnovat se jejich rozboru později nebo je například sdílet s ostatními.

  VisualVM umožňuje sledovat zatížení CPU, využití paměti, dokáže spouštět explicitně garbage kolekci a pořizovat již zmíněné snapshoty.

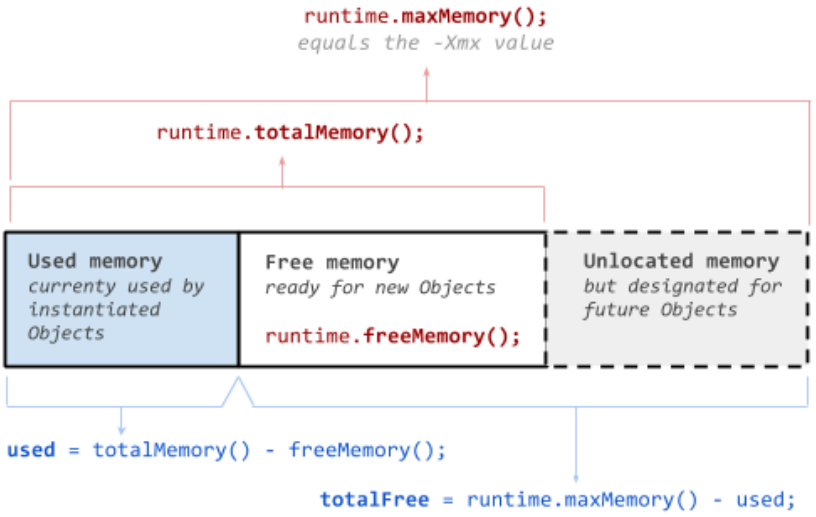
[1] **Java JDK** je balík knihoven a nástrojů, které vytváří prostředí pro běh Java aplikací a poskytují vývojáři prostředky pro implementaci, kompilaci a ladění Java programů.

2.2.2 Alternativní způsob měření

Použití profilovacích nástrojů není jediným možným způsobem jak sledovat množství využité paměti. Alternativní způsob měření nabízí třída `Runtime`, která je součástí *Java API* a jejíž instanci má každá běžící Java aplikace.

Prostřednictvím instance této třídy lze přistupovat k parametrům prostředí, ve kterém je aplikace spuštěna. Instanci lze získat voláním statické metody `getRuntime()` ze třídy `Runtime` a nabízí následující metody:

- **totalMemory()** Vrátí množství aktuálně alokované pamětí JVM v B;
- **freeMemory()** Vrátí množství aktuálně dostupné pamětí v JVM v B;
- **maxMemory()** Vrátí maximální množství paměti, které má JVM povoleno alokovat.



Obrázek 2.4: Paměť v Javě z hlediska dostupnosti

Použití metod třídy `Runtime` z Java API lze kombinovat s ladícím režimem. Aplikaci lze na libovolném místě zdrojového kódu pozastavit a dotázat se na přesný stav paměti. Protože je navíc možné přímo z debug módu vyvolat garbage kolekci, nabízí tento způsob získání reálného přehledu o tom, o kolik narostlo/snížilo se využití paměti po vykonání určitého bloku kódu.

2.3 Ehcache

Ehcache je ryze Java řešení mezipaměti. Toto řešení je rychlé, jednoduché, použitelné s minimální režií a vynucuje minimální množství závislostí. Toto řešení je navíc dobře dokumentované a produkčně otestované [6]. Dostupný je ve dvou variantách:

1. **Základní verze**

   Je distribuována jako open-source projekt a poskytuje základní funkcionalitu pro práci s mezipamětí.

2. **Komerční produkt**

   Komerční produkt nabízí rozšířenou funkcionalitu a jeho licence je placená. Mezi dodatečnou funkcionalitu patří například ukládání dat do tzv. *off-heap* vrstvy (více viz dále).

2.3.1 Klíčové prvky

Klíčovými prvky pro práci s knihovnou Ehcache jsou:

- **správce cache**

  Knihovna Ehcache umožňuje vytvářet libovolné množství mezipamětí (dále jako cache). Pro jejich správu slouží správce cache, který je reprezentovaný třídou `CacheManager`. Instance této třídy umožňuje vytvářet jednotlivé cache a spravovat jejich životní cyklus.

- **cache**

  Jedná se o ústřední prvek nástroje Ehcache. Cache je reprezentovaná stenojmennou třídou `Cache` a její instance umožňuje spravovat a uchovávat jednotlivé položky. Cache může být nakonfigurována a může používat jako úložnou vrstvu i pevný disk (více dále).

- **položka cache**

  Položka je nositelem informace a sestává se z klíče a z hodnoty. Klíč je použit jako identifikátor při správě položky v cache a hodnota umožňuje uchovávat libovolná data. Pokud je zamýšleno položku ukládat na disk, musí být hodnota položky serializovatelná. Položka je reprezentována třídou `Element`.

Vrstvy mezipaměti

Mezipaměť umožňuje konfiguraci následujících úložných vrstev:

- **MemoryStore**
  - Položky v této vrstvě jsou ukládány standardně na **heap**.
  - Nejnižší přístupová doba.
  - Podléhá garbage kolekci.
  - Umožňuje ukládat **libovolné** objekty.
  - Vrstva je **povinná**.
- **OffHeapStore**
  - Slouží jako rozšiřující úložiště pro položky, které se už nevejdou do *MemoryStore*.
  - Nepodléhá garbage kolekci.
  - Její velikost je omezena pouze velikostí RAM.
  - Dostupná pouze v **placené verzi**.
  - Umožňuje ukládat pouze **serializovatelné** objekty.
  - Vrstva je **nepovinná**.
- **DiskStore**
  - Slouží jako rozšiřující úložiště pro položky, které se už nevejdou do paměti (*MemoryStore, případně OffHeapStore*).
  - Její velikost je omezena kapacitou HDD.
  - Nejvyšší přístupová doba.
  - Umožňuje ukládat pouze **serializovatelné** objekty.
  - Vrstva je **nepovinná**.

Jednotlivé možnosti konfigurace jsou popsány v následující podkapitole.

2.3.2 Konfigurace

Jednotlivé instance cache je možné konfigurovat dvěma možnými způsoby:

1. **Staticky**

   Jednotlivé cache lze pojmenovat a nadefinovat jejich konfiguraci *staticky* prostřednictvím XML souboru. Uvnitř stejného souboru je také možné nadefinovat konfiguraci pro výchozí cache.

2. **Dynamicky**

   Jednotlivé parametry cache se nastavují přímo ve zdrojovém kódu. Tato konfigurace tedy probíhá za běhu programu, tedy *dynamicky*.

Významné parametry cache

Mezi nejdůležitější parametry, které je možné cache nastavit, patří:

- **name** Jedná se o jedinečný identifikátor cache.
- **maxEntriesLocalHeap** Udává maximální počet položek v *MemoryStore*.
- **maxEntriesLocalDisk** Udává maximální počet položek v *DiskStore*.
- **eternal** Definuje životnost položky v cache. Pokud je nastaveno na `true`, parametry *timeToIdleSeconds* a *timeToLiveSeconds* se ignorují a položka nikdy nebude z cache odstraněna.
- **diskSpoolBufferSizeMB** Velikost fronty v MB, do které se přidávají položky určené k serializaci na disk - vykonává se asynchronně v samostatném vláknu.
- **timeToIdleSeconds** Maximální doba mezi přístupy k položce v sekundách. Je-li tato doba překročena, položka je zneplatněna.
- **timeToLiveSeconds** Délka života položky v cache v sekundách.
- **persistence** Umožňuje nastavit způsob perzistence položek v cache. Perzistence může být: *žádná*, *dočasná* , *trvalá* (placená verze) a *distribuovaná* (placená verze). *Trvalá* se od *dočasné* odlišuje v tom, že jsou položky dostupné i mezi jednotlivými instancemi běhu programu. Distribuovaná se používá na distribuovaných systémech.

XML konfigurace

Kořenovým elementem konfiguračního XML souboru je element **ehcache**. Mezi hlavní elementy, které se mohou uvnitř tohoho elementu nacházet, patří:

- **diskStore**

  Jedná se o element s jediným atributem *path*, který specifikuje cestu k adresáři, do kterého *DiskStore* ukládá svoje položky.

- **cache**

  Tento element umožňuje definovat pojmenovanou cache a nastavit hodnoty atributů popsaných na předešlé straně.

- **defaultCache**

  Tento element umožňuje konfigurovat parametry pro výchozí cache.

Programová konfigurace

V této části se nachází ukázka možné dynamické konfigurace cache přímo ze zdrojového kódu.

```
1  Cache cache = new Cache(
2    new CacheConfiguration()
3      .name("sampleCache1")
4      .maxEntriesLocalHeap(10000)
5      .maxEntriesLocalDisk(1000)
6      .eternal(false)
7      .diskSpoolBufferSizeMB(20)
8      .timeToIdleSeconds(30)
9      .timeToLiveSeconds(60)
10     .memoryStoreEvictionPolicy(MemoryStoreEvictionPolicy.LFU)
11     .persistence(new PersistenceConfiguration()
12             .strategy(Strategy.LOCALTEMPSWAP)));
```

Obrázek 2.5: Konfigurace cache ve zdrojovém kódu

Nová instance cache je vytvořena pomocí konstruktoru třídy `Cache`. Tento konstruktor na vstupu očekává instanci třídy `CacheConfiguration`, která slouží pro nastavení jednotlivých parametrů cache. Metody pro nastavení hodnot parametrů odpovídají svým jménem názvům atributů.

3 JaCC

JaCC je nástroj pro statickou analýzu kódu, který je vyvíjen na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Tento nástroj je distribuován jako Java knihovna a umožňuje ověřovat vzájemnou kompatibilitu jiných Java knihoven [7].

Nástroj JaCC poskytuje tři základní prostředky pro ověření kompatibility:

1. **Black list/White list**

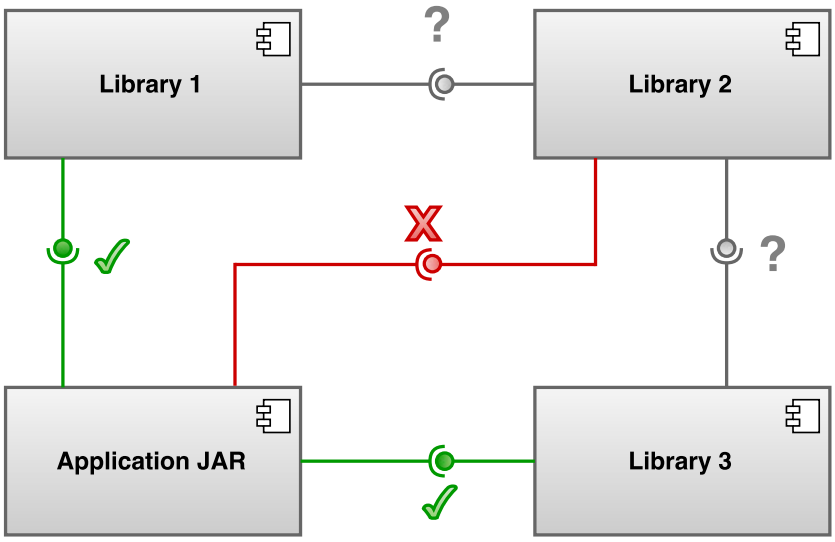
   Tento prostředek umožňuje definovat kontrakt, ve kterém se popíše klientské API, které je zakázáno/dovoleno používat. Výstupem vyhodnocení tohoto prostředku je pak například informace o tom, že je v projektu použito volání nějaké metody, které je ovšem podle kontraktu zakázáno.

2. **1:N komparátor**

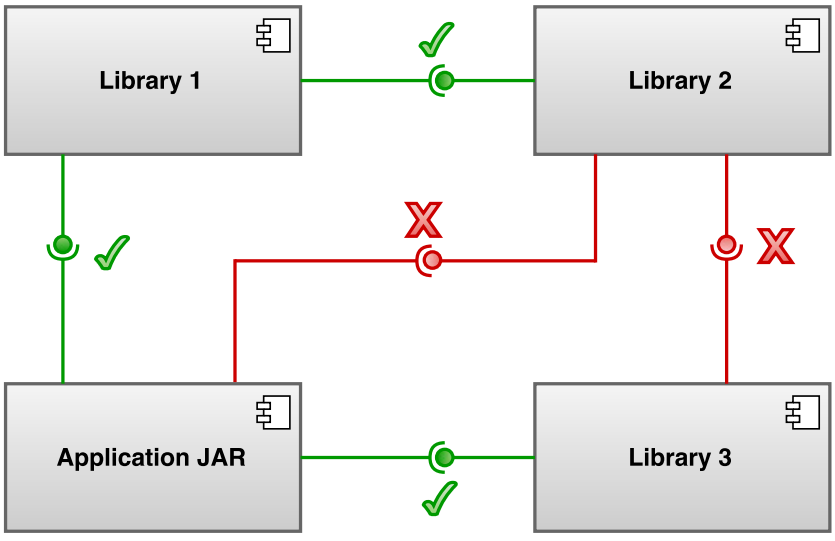
   Umožňuje vyhodnocovat kompatibilitu mezi implementovanou aplikací (1) a použitými cizími knihovnami (N). Toto ověření je ovšem pouze jednoúrovňové a nezachycuje vazby mezi samotnými knihovnami (viz obrázek 3.1).

3. **M:N komparátor**

   Umožňuje vyhodnocovat vzájemnou kompatibilitu všech komponent nehledě na to, zda se jedná o implementovanou aplikaci či cizí knihovnu (viz obrázek 3.2).



Obrázek 3.1: Ověření kompatibility pomocí 1:N komparátoru



Obrázek 3.2: Ověření kompatibility pomocí M:N komparátoru

3.1 Princip funkčnosti

Nástroj JaCC vychází ze dvou základních myšlenek:

1. Bytecode vzniká při kompilaci zdrojového kódu v programovacím jazyce Java a obsahuje informaci o stavbě tříd coby základních stavebních prvcích při vývoji v tomto jazyce. Tuto informaci o stavbě tříd je možné s použitím reverzního inženýrství [1] zpětně extrahovat z bytecode a uložit do vhodných datových struktur, se kterými je možné dále pracovat. Obsahem těchto datových struktur pak mohou být například informace o atributech, konstruktorech či metodách pro danou třídu.

2. Tato myšlenka navazuje na myšlenku z prvního bodu a dává do souvislosti kompatibilitu s datovými strukturami získanými prostřednictvím reverzního inženýrství. Máme-li tak například k dispozici datovou reprezentaci téže třídy z různých knihoven, můžeme vzájemným porovnáním struktur těchto tříd rozhodnout o jejich vzájemné kompatibilitě. Podrobnější informace o této problematice se nacházejí v jedné z následujících pokapitol 3.1.2.

Zpracování nástrojem JaCC probíhá ve 3 hlavních fázích:

1. **Načtení**

   Provede se načtení binárních souborů, jejichž kompatibilita má být ověřena, a vytvoří se datová reprezentace použitých tříd uvnitř bytecode. Výsledkem jsou instance tříd z balíčku `javatypes`.

2. **Komparace**

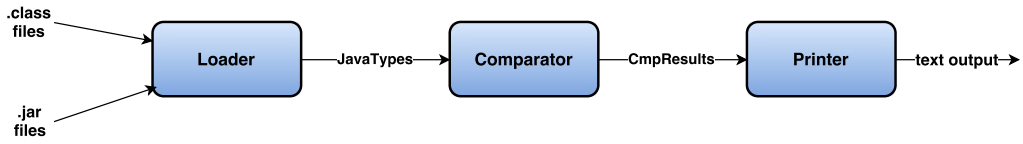
   Provede se porovnání datových struktur pro stejnojmenné třídy a výsledky porovnání se uloží do instancí třídy `CmpResult`.

3. **Výpis**

   Provede se výpis uložených výsledků porovnání.

[1] **Reverzní inženýrství** je v informatice definováno jako proces analýzy předmětného systému s cílem identifikovat komponenty systému a jejich vzájemné vazby a/nebo vytvořit reprezentaci systému v jiné formě nebo na vyšší úrovni abstrakce [8].

Na následujícím obrázku jsou jednotlivé fáze schématicky znázorněny.



Obrázek 3.3: JaCC v kostce

JaCC moduly

JaCC je implementovaný jako *Maven*[12] projekt a je logicky členěn podle funkčnosti do následujících modulů:

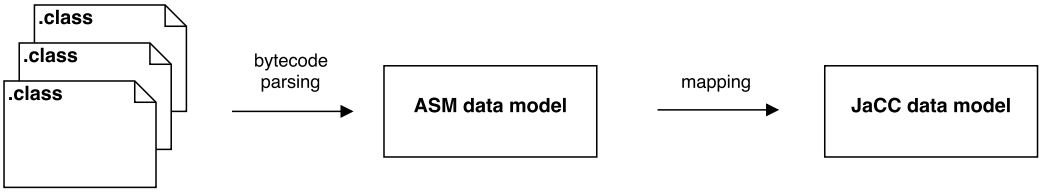
- **javatypes** V rámci tohoto modulu je definovaný datový model, který je použit pro reprezentaci tříd získaných pomocí reverzního inženýrství bytecode. Nachází se zde tedy třídy a rozhraní, které umožňují uchovávat informace o zpracovaných třídách a jejich atributech, metodách, použitých parametrech metod atd.
- **javatypes-cmp** V rámci tohoto modulu je implementována funkcionalita, která se týká komparace datových struktur implementovaných v modulu *javatypes*. Mimo jiné se zde i nachází implementace datové struktury, která umožňuje uchovávat výsledek komparace.
- **javatypes-loader** V rámci tohoto modulu je implementována funkcionalita, která zajišťuje čtení bytecode a převod do interních datových struktur z modulu *javatypes*.
- **types-cmp** Uvnitř tohoto modulu se nachází definice struktur výsledků porovnání v podobě rozhraní. Dále se zde nacházejí výčtové typy použité pro identifikaci konkrétní odlišnosti výsledku porovnání a způsobu, kterým je možné případnou odlišnost odstranit.
- **compatibility-checker-utils** V rámci tohoto modulu je implementována hlavní logika nástroje JaCC. Tento modul využívá všech ostatních modulů a poskytuje různé nástroje pro ověření kompatibility.

3.1.1 Načtení

Klíčovým prvkem při načítání bytecode je open-source *framework*[2] **ASM**, který je určen pro práci s bytecode. Tento framework je zaměřen primárně na jednoduchost použití a na výkonnost. Vedle základní funkcionality, jako je analýza existujících tříd bytecode (uchovávány v `.class` souborech), nabízí i pokročilejší funkce jako je úprava stávajících tříd či dokonce generování tříd nových [13].

Framework ASM je významným způsobem použit ve třídě `AsmDataParser`. Jedná se o jednu z nejvýznamnějších tříd nástroje JaCC, která zajišťuje v následujícím pořadí načtení a analýzu bytecodu, převod výsledků analýzy do datového modelu ASM a nakonec převod z datového modelu ASM do datového modelu JaCC.

Velmi zjednodušeně by se dal popsaný postup zachytit následujícím obrázkem 3.4.



Obrázek 3.4: Detail fáze načítání

[2]**Framework** je softwarová struktura, která slouží jako podpora při programování, vývoji a organizaci jiných softwarových projektů. Může obsahovat knihovny, podpůrné programy, podporu pro návrhové vzory nebo doporučené postupy při vývoji.

Z .class na ASM

Nejprve je třeba načíst bytecode, provést jeho analýzu a nakonec převést výsledky analýzy do interního datového modelu ASM. Pro implementaci této funkcionality byly použity následující třídy ASM, které požadavky na logiku v této části víceméně pokrývají:

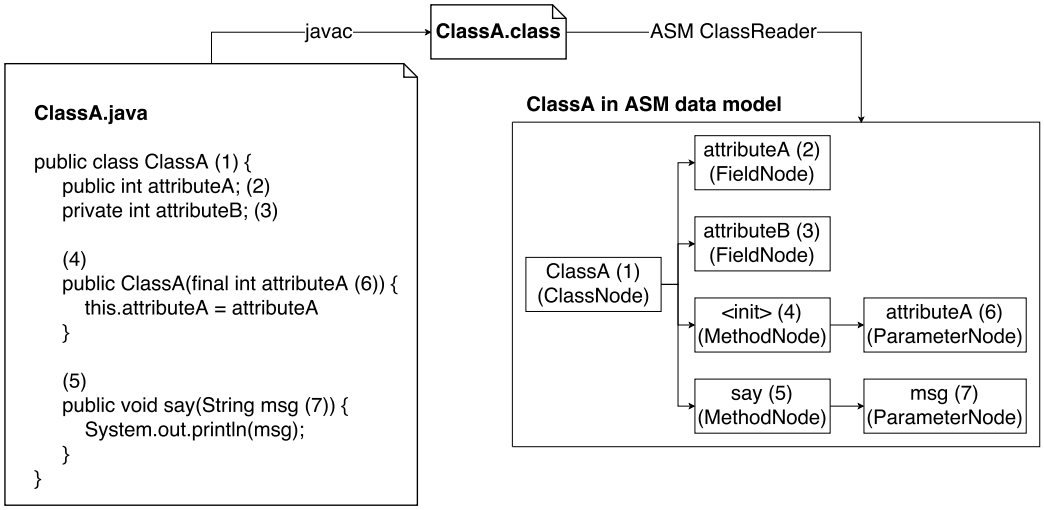
1. **org.objectweb.asm.ClassReader**

   Tato třída umožňuje načítat datový proud obsahující `.class` soubor a analyzovat jeho vnitřní stavbu.

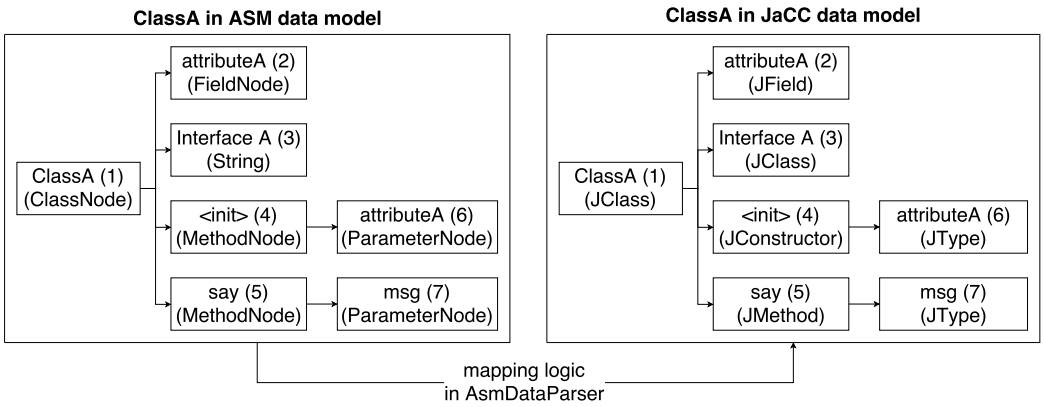
2. **org.objectweb.asm.tree.ClassNode, MethoNode, FieldNode ..**

   Jedná se o třídy datového modelu, které framework ASM používá pro uložení stavby analyzovaných tříd. Z názvů tříd je možné odvodit jejich význam.

Na následujícím obrázku 3.5 je zachycen způsob, jakým ASM ClassReader provádí mapování kompilovaných `.class` souborů do svého datového modelu. Na levé straně se nachází zdrojový kód smyšlené třídy `ClassA` před kompilací. Na pravé straně se potom nachází reprezentace téže třídy v datovém modelu ASM. Pro lepší představu jsou odpovídající si elementy označeny v závorce stejným identifikátorem. V horní části obrázku se nachází popis přechodu mezi levou a pravou stranou obrázku.



Obrázek 3.5: Převod .class souboru do datového modelu ASM



Obrázek 3.6: Mapování datového modelu ASM na datový model JaCC

Z ASM na javatypes

Datový model nástroje JaCC je až na drobné niance velice podobný datovému modelu frameworku ASM. Mapování datového modelu ASM na datový model nástroje JaCC je tedy poměrně přímočaré. Jednotlivé třídy a rozhraní datového modelu JaCC jsou implementovány uvnitř modulu `javatypes`.

Na obrázku 3.6 výše je zachyceno mapování pro již jednou použitou třídu `ClassA`. Nyní má ovšem třída `ClassA` o jeden atribut méně (chybí atribut `attributeB`) a implementuje rozhraní `InterfaceA`. Jak je vidět, v datovém modelu ASM je informace o implementovaném rozhraní zachycena jednoduše pomocí datového typu `String` - držena je pouze informace o názvu implementovaného rozhraní. V datovém modelu nástroje JaCC je ovšem informace o implementovaném rozhraní zachycena pomocí zpětné vazby na datový typ `JClass`, který je již použit pro reprezentaci samotné třídy `ClassA`. Tímto způsobem tedy vzniká cyklická vazba mezi datovými strukturami modelu nástroje JaCC.

Vyplývá z toho tedy následující důsledek. Jednotlivá zpracování kompilovaných tříd se postupně zanořují do doby, než dojde k objevení nového rozhraní či rodičovské třídy, která je již zpracována nebo se právě zpracovává. Třída `AsmDataParser` proto definuje vnitřní třídu `AsmParserState`, která slouží k uchování stavu zpracování a k využití již jednou zpracovaných tříd.

Importované a exportované JClass

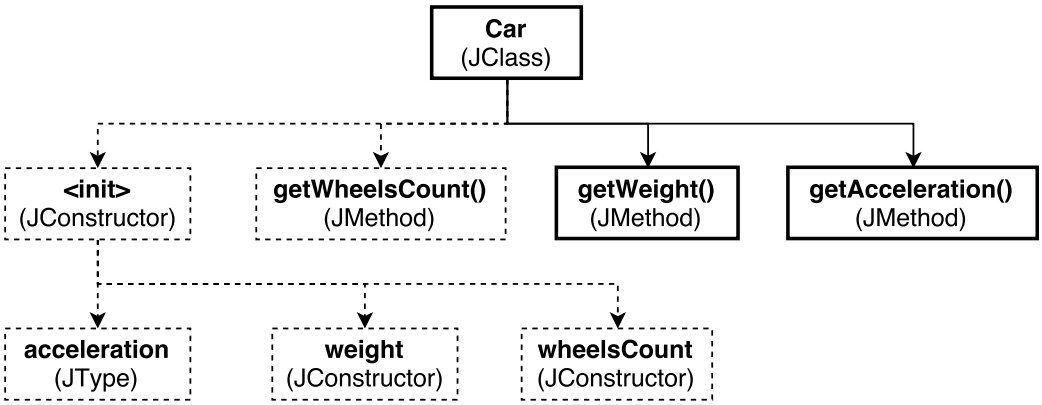
V průběhu načítání bytecode vznikají v datovém modelu nástroje JaCC dva druhy `JClass` instancí, resp. dva druhy tříd v terminologii datového modelu JaCC. Tyto dva druhy tříd se od sebe odlišují hodnotou logického atributu `fullyRepresented`:

1. **fullyRepresented = true** Třída s příznakem `fullyRepresented` nastaveným na `true` deklaruje, že plně reprezentuje třídu původně kompilovanou do odpovídajícího `.class` souboru. Taková třída nese označení **exportovaná**.
2. **fullyRepresented = false** Třída s příznakem `fullyRepresented` nastaveným na `false` vzniká průběžně v době načítání `.class` souborů a slouží k reprezentaci třídy, na níž bylo nějakým způsobem (proměnná, volání statické metody atd.) odkazováno, ale pro niž neexistuje odpovídající `.class` soubor. Takové třídě se říká **importovaná**.

Pro lepší představu toho, jak se od sebe odlišuje *importovaná* a *exportovaná* třída, je opět přiložena praktická ukázka. Mějme následující zdrojový kód:

```
1  import cars.Car;
2  public class ParkingLot {
3
4      private String name;
5      private int capacity;
6      private Car[] cars;
7
8      public ParkingLot(String name, int capacity) {
9          this.name = name;
10         this.capacity = capacity;
11         this.cars = new Car[capacity];
12     }
13
14     public int getAvgCarsWeight() {
15         int sum = 0;
16         for (Car c : cars) { sum += c.getWeight(); }
17         return sum / cars.size();
18     }
19
20     public double getAvgAcceleration() {
21         double sum = 0;
22         for (Car c : cars) { sum += c.getAcceleration(); }
23         return sum / cars.size();
24     }
25 }
```

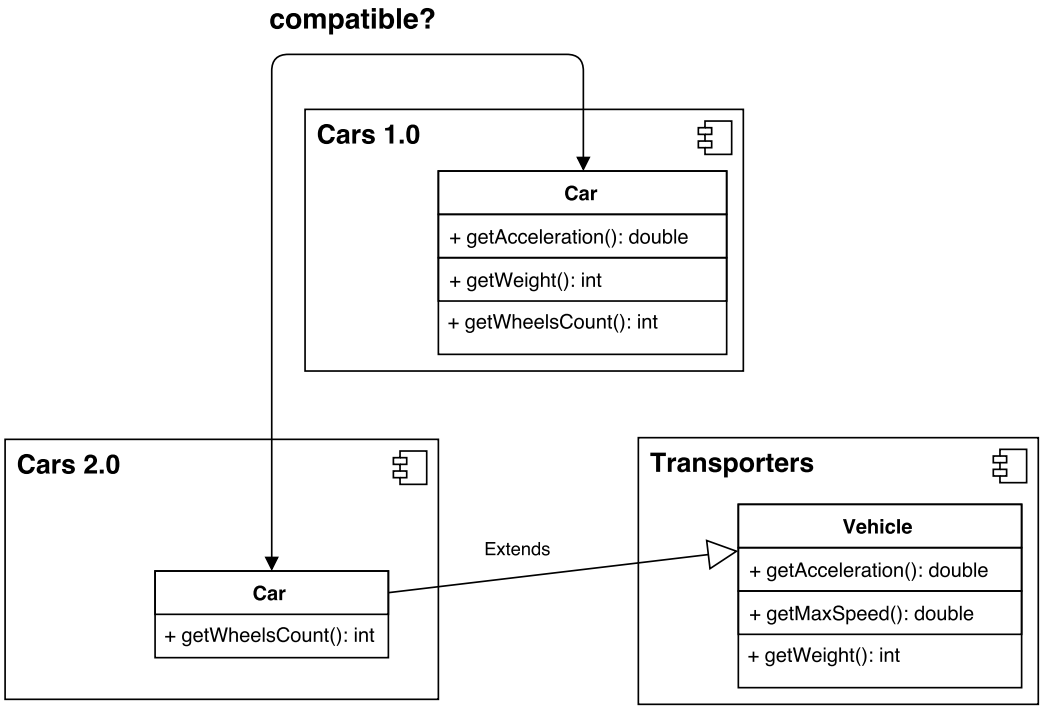
Obrázek 3.7: Ukázková třída `Person`



Obrázek 3.8: Třída `Car`

Řekněme, že se třída `Car` nachází v cizí knihovně, jejíž balíček `cars` v našem projektu importujeme. Nyní třídu `ParkingLot` z ukázky zkompilujeme a necháme zpracovat nástrojem JaCC. Výsledkem zpracování bude jedna exportovaná třída `ParkingLot` a jedna importovaná třída `Car`. To proto, že nástroj JaCC bude mít v době zpracování k dispozici úplnou reprezentaci třídy `ParkingLot` (`ParkingLot.class`), ale ne už úplnou reprezentaci třídy `Car`.

Importovaná třída `Car` bude mít na konci zpracování pouze 2 metody: `int getWeight()` a `double getAcceleration()`. Tato neúplná reprezentace třídy je znázorněna pomocí souvislých čar v obrázku 3.8 výše. V obrázku je znázorněna současně i její potenciálně úplná reprezentace - exportovaná verze - pomocí přerušovaných a souvislých čar. V úplné reprezentaci třída obsahuje ještě navíc konstruktor a metodu `int getWheelsCount()`.



Obrázek 3.9: Interní a externí třídy

Interní a externí JClass

Dalším možným způsobem, jak lze na třídy nahlížet, je z pohledu jejich zdrojové komponenty.

Pokud třída pochází z komponenty, která je předmětem analýzy, jedná se o tzv. **interní** třídu, respektive `JClass`, a hodnota jejího atributu `fromOutside` je `false`. Pokud třída ovšem nepochází ze zdrojové komponenty, jejíž kompatibilitu ověřujeme, ale existuje v jejím kontextu a slouží pouze k rekonstrukci ostatních interních tříd, které na ní závisejí, jedná se o tzv. **externí** třídu a hodnota jejího atributu `fromOutside` je `true`.

Řekněme, že zkoumáme vzájemnou kompatibilitu komponent *Cars 1.0* a *Cars 2.0* (viz obrázek 3.9 výše). Externí třídou bude třída `Vehicle` z komponenty `Transporters` a použije se při rekonstrukci třídy `Car` z komponenty `Cars 2.0`. Následně dojde ke komparaci tříd `Car` z komponent `Cars 1.0` a `Cars 2.0`. Kompatibilita třídy `Vehicle` ovšem už vyhodnocena nebude. Jedná se totiž o externí třídu a ty nejsou předmětem analýzy kompatibility.

3.1.2 Komparace

Po fázi načtení, kdy je k dispozici inicializovaný datový model nástroje JaCC, následuje fáze porovnávání - tedy komparace. Komparace probíhá vždy mezi dvěma stejnými datovými typy datového modelu a provádí ji tzv. *komparátory* z modulu `javatypes-cmp`. Příkladem konkrétní implementace komparátoru může být `JClassComparator` či `JFieldCollectionComparator`.

Podrobný popis algoritmu porovnání odpovídajících si elementů datového modelu nástroje JaCC by byl nad rámec této diplomové práce. Nicméně principiálně a trochu zjednodušeně porovnávání probíhá způsobem popsaným dále v této kapitole.

Pro popis odlišnosti, kterou příslušný komparátor odhalí během komparace, je použit výčtový datový typ `Difference`, který se nachází uvnitř modulu `types-cmp`. Hodnota tohoto výčtového datového typu se vždy vztahuje k výsledku porovnání dvojice elementů téhož datového typu. Tento výčtový typ definuje následující hodnoty:

1. **NON (none)** Žádná odlišnost mezi prvním a druhým elementem.
2. **INS (insertion)** Druhý element je nadbytečný, či nabízí rozšířenou funkcionalitu oproti prvnímu elementu.
3. **DEL (deletion)** Druhý element chybí nebo nabízí omezenou funkcionalitu oproti prvnímu elementu.
4. **GEN (generalization)** Druhý element je zobecněním prvního elementu.
5. **SPE (specialization)** Druhý element je specializací prvního elementu.
6. **MUT (mutation)** Druhý element je naprosto odlišný.
7. **BOX (boxing)** Druhý element vznikl během *un/boxing*[3] prvního elementu.
8. **UNK (unknown)** Neznámá odlišnost mezi prvním a druhým elementem.

[3] **Boxing** označuje automatickou konverzi, kterou provádí Java kompilátor mezi primitivními datovými typy a jejich odpovídajícími obalovými třídami. Například může jít o konverzi z `int` na `Integer` či z `double` na `Double`. **Unboxing** označuje stejnou konverzi, ale v opačném směru.

Vlastní proces porovnávání probíhá opět zjednodušeně následujícím způsobem:

- Porovnání začíná na úrovni tříd, tedy datového typu `JClass`.
- V případě, že je prvním i druhým elementem stejná instance třídy `JClass`, či jsou obě instance prázdné (= `null`), je vrácen výsledek porovnání s odlišností `NON`.
- V případě, že je první element prázdný, je vrácen výsledek porovnání s odlišností `INS`. Pokud je prázdný druhý element, je vrácen výsledek porovnání s odlišností `DEL`.
- Porovnání tříd, kde alespoň jedna ze tříd je externí, ústí v navrácení výsledku porovnání s odlišností `NON`. Externí třídy nejsou určeny pro porovnávání.
- Porovnávají se třídy a zjišťuje se, zda nejsou obě typu pole. Pokud jsou, ale liší se v dimenzi (velikost pole), je vrácen výsledek s odlišností `MUT`. Pokud je jedna třída polem, ale druhá není, je vrácen výsledek s odlišností `MUT`. Ve speciálním případě, kdy je první třída polem a druhá třída typu `Object`, je navrácen výsledek s odlišností `GEN`.
- Pokud porovnávané třídy nemají stejná jména, ale jedna ze tříd je typu `Object`, je vrácen výsledek porovnání s odlišností `SPE` (pokud je typu `Object` první třída) či `GEN` (pokud je typu `Object` druhá třída).
- Porovnání dvou importovaných tříd se považuje za nesmyslné a je navrácen výsledek porovnání s odlišností `NON`. Nemá totiž smysl porovnávat dvě neúplné reprezentace téže třídy.
- Porovnání dvou exportovaných tříd ústí v porovnání jejich modifikátorů. Exportované třídy se poté dále kontrolují, zda jsou, či nejsou obě současně rozhraním. Pokud jedna třída je, ale druhá není, vzniká výsledek porovnání `MUT`.
- Nakonec porovnání postupuje dále k atributům, konstruktorům a metodám porovnávaných elementů typu `JClass`.

Odlišnost `BOX` je použita jen u komparátorů primitivních datových typů.

Výsledky porovnání

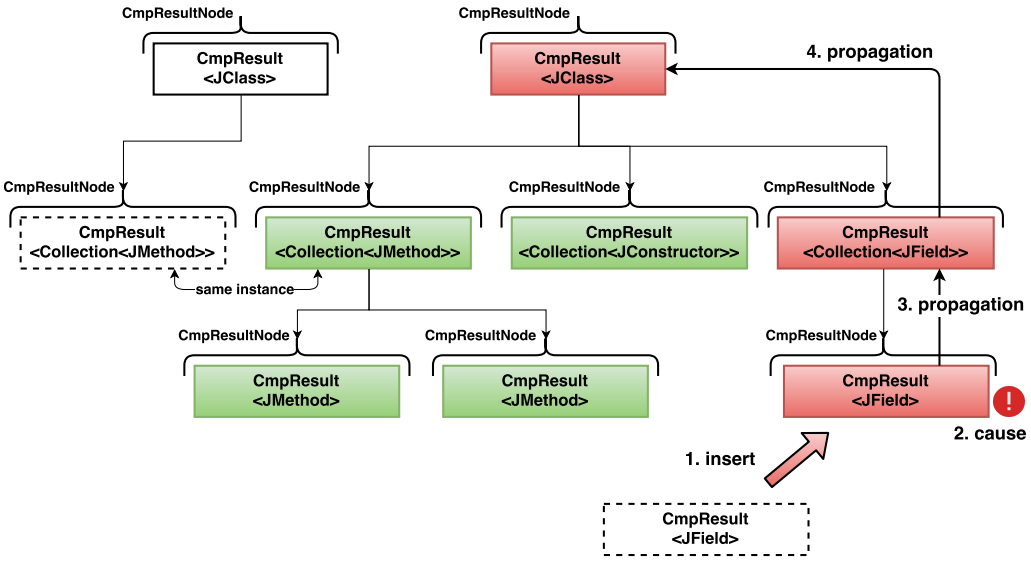
Pro uchování výsledků porovnání je použit datový typ **CmpResult**. Hlavními údaji, který tento datový typ udržuje, jsou:

- **firstObject** První element porovnání.
- **secondObject** Druhý element porovnání.
- **diff** Odlišnost popsaná výčtovým datovým typem `Difference`.
- **code** Textový řetězec sloužící k podrobnějšímu popisu výsledku porovnání.
- **childrenCompatible** Logická hodnota, která říká, zda je celý podstrom výsledků porovnání kompatibilní.

Pro uchování hierarchie výsledků porovnání slouží datový typ **CmpResultNode** (viz obrázek 3.10). Tento datový typ umožňuje vzájemně propojit dvojici výsledků porovnání. Jeho hlavními atributy jsou:

- **result** Držený výsledek porovnání.
- **parent** Rodičovský výsledek porovnání.
- **contentCode** Slouží pro popis kontextu, ve kterém je uložen výsledek porovnání. Například je zde uložen symbolický řetězec pro kontext metod, konstruktorů, parametrů atd.
- **hierarchyCompatible** Logická hodnota, která říká, zda je kompatibilní celý podstrom výsledků porovnání.
- **incompatibilityCause** Logická hodnota, která říká, zda je držený výsledek porovnání příčinou nekompatibility.

Ke vzniku stromové hierarchie výsledků porovnání dochází v průběhu postupného porovnávání jednotlivých tříd, jejich atributů, metod, parametrů těchto metod atd. Tato stromová hierarchie výsledků potom umožňuje vzájemně vyhodnocovat kompatibilitu tříd.



Obrázek 3.10: Propagace výsledku porovnání

Z hlediska kompatibility výsledků porovnání platí jednoduché pravidlo: pokud má být výsledek porovnání kompatibilní, pak musí být všechny výsledky porovnání v jeho podstromu také kompatibilní.

Pokud se tedy například při ověření kompatibility tříd ukáže, že si třídy neodpovídají třeba jen typem jediného parametru některé z metod, jsou tyto parametry považovány za nekompatibilní a při cestě vzhůru stromem výsledků porovnání jsou za nekompatibilní označeny i metody, kde se parametry objevily, a nakonec samotné třídy. Tento přepočet kompatibility směrem vzhůru od příčiny až ke kořeni stromu s výsledky se spouští vždy, když je do stromu výsledků porovnání přidáván nový výsledek porovnání (viz obrázek 3.10 výše).

Dále je nutné říci, že některé výsledky porovnání se mohou objevit na více místech v celé hierarchii výsledků porovnání. Důsledkem potom je, že existuje více instancí `CmpResultNode`, které ukazují na stejný `CmpResult` - zachyceno v levé části obrázku.

3.1.3 Výstup

Výstup nástroje JaCC má textovou podobu a popisuje nalezené nekompatibility. Na následujícím obrázku 3.11 je zachycen fragment jednoho z možných výstupů nástroje JaCC. Pod obrázkem se nachází popis jednotlivých částí fragmentu. Názvy tříd jsou pro přehlednost uvedeny v popisu ve zkrácené podobě bez názvu zdrojového balíčku.

```
1  --< /lib/jbossws-spi-4.2.2.jar
2     org.jboss.wsf.spi.tools.ant.WSConsumeTask
3     org.jboss.wsf.spi.tools.ant.WSProvideTask
4                     |
5     Not found       #1463  <>-- org.apache.tools.ant.types.Reference
6  ------------------------------------------------------------------------
7  --< /lib/aheritrix-1.14.1.jar
8     org.archive.crawler.util.MemFPMergeUriUniqFilter
9     Duplicated        |   <>-- it.unim.dsi.fastutil.longs.LongArrayList
10                    #1528        /lib/dsi.unim.it-1.2.0.jar
11                      |        -> #1323
12                    #1529        /lib/fastutil5-5.0.9.jar
13 ------------------------------------------------------------------------
14    Must remove       |    /lib/axis-schema.jar
15 ------------------------------------------------------------------------
16    Redundant         |    /lib/activation.jar
17                      |    /etc/postgresql-8.1-404.jdbc3.jar
18                      |    /lib/axis-schema.jar
19                      |    /lib/httpunit.jar
```

Obrázek 3.11: Textový výstup JaCC

Na řádcích **1-5** je údaj o importované třídě `Reference`, která nebyla během zpracování nalezena (*Not found* na řádku 5). Třída byla použita uvnitř tříd `WSConsumeTask` a `WSProvideTask` pocházejících z knihovny `jbossws-spi-4.2.2.jar` (řádek 1).

Na řádcích **7-12** je informace o importované třídě `LongArrayList`, která je duplicitně poskytována knihovnou `dsi.unim.it-1.2.0.jar` a `fastutil5-5.0.9.jar`. Knihovna `dsi.unim.it-1.2.0.jar` tuto třídu poskytuje v nekompatibilní podobě a popis nekompatibility má číslo `1323` (řádek 11).

Na řádku **14** je tzv. *redundantně-nekompatibilní* knihovna. Jedná se o knihovnu, jejíž funkcionalita je duplikována a sama knihovna je nekompatibilní. Uživateli je proto doporučeno tuto knihovnu nepoužívat.

Na řádcích **16-19** jsou tzv. *redundantní* knihovny, které jsou pro chod zkoumané aplikace nepotřebné.

4 Optimalizace JaCC

Nástroj z hlediska výkonnosti dosahuje poměrně slušných výsledků a jeho doba běhu se zpravidla pohybuje v jednotkách minut. Co se ovšem týká jeho paměťové náročnosti, tak zde je situace celkem nepříznivá.

Z dostupných prostředků nástroje JaCC má nejvyšší paměťové nároky M:N komparátor. Velikost potřebné paměti pro tento prostředek během analýzy komponentových aplikací, jejichž velikost se pohybuje ve stovkách MB, často přesahuje i hranici 10 GB. Z tohoto důvodu je předmětem této práce především paměťová optimalizace nástroje JaCC.

4.1 Problém

Nejprve se bylo nutné podrobněji seznámit s nástrojem JaCC a porozumět jeho vnitřní implementaci. Předmětem zájmu byl především M:N komparátor.

Tento prostředek očekává na vstupu dva typy `.jar` souborů, resp. komponent:

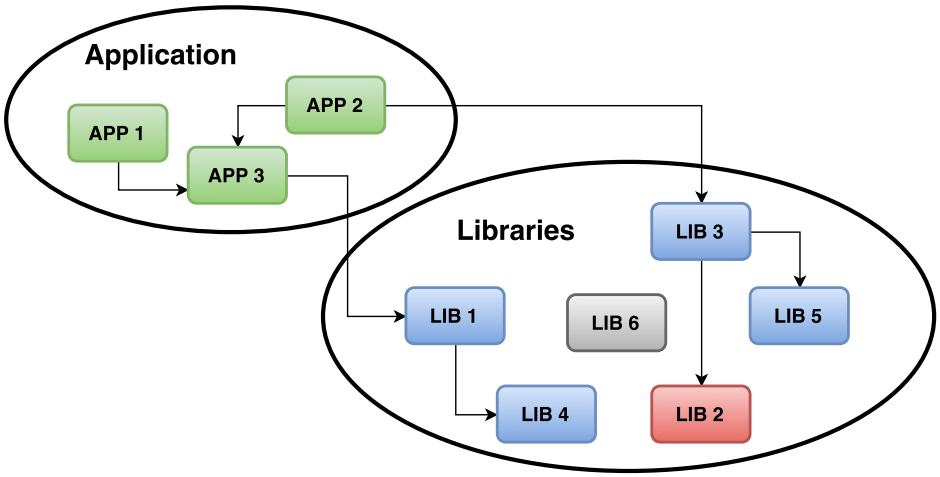
1. **Aplikační JAR**

   Aplikačními JAR se zde rozumí jednotlivé nově implementované JAR v rámci vývoje komponentové aplikace.

2. **Knihovní JAR**

   Knihovními JAR se zde rozumí již existující komponenty třetích stran, které jsou použity pro vývoj komponentové aplikace.

Na následující stránce se nachází schématický obrázek, který zachycuje komponentovou aplikaci a rozděluje ji na aplikační a knihovní komponenty. V rámci komponent knihovních jsou dále barevně odlišeny knihovny redundantní (šedivá barva) a knihovny redundantně-nekompatibilní (červená barva) - více o významu těchto pojmů v kapitole 3.1.3).



Obrázek 4.1: Ukázková komponentová aplikace

M:N komparátor pracuje v následujících krocích:

1. Provede se načtení všech aplikačních JAR a vytvoří se reprezentace všech obsažených tříd - vzniknou aplikační exportované třídy.
2. Načtené aplikační třídy se uloží do zásobníku a označí jako zpracované. Zásobník je následně použit pro inicializaci dalšího zpracování.
3. Třídy se postupně odebírají ze zásobníku a pro každou takto odebranou třídu se provádí načítání tříd, které importují.
4. Pro každý import proběhne dohledání příslušného exportu (stejnojmenné třídy), provede se komparace importované a exportované třídy (exportovaných tříd může být vícero, komparace proběhne několikrát) a dojde k uložení výsledků porovnání. Na pozadí současně vzniká strom závislostí mezi komponentami - nově přidaná vazba je vždy mezi zdrojovou komponentou třídy, která importuje, a zdrojovou komponentou exportované třídy pro odpovídající import.
5. Dosud nezpracované exportované třídy z předchozího kroku se opět ukládají do zásobníku.
6. Návrat k bodu 3., pokud zásobník s třídami není prázdný.
7. Zásobník s třídami je prázdný a zpracování končí. Nedosažitelné komponenty se označí za redundantní a komponenty, které jsou nekompatibilní a duplikovány, se označí za redundantně-nekompatibilní.

Soustava loaderů

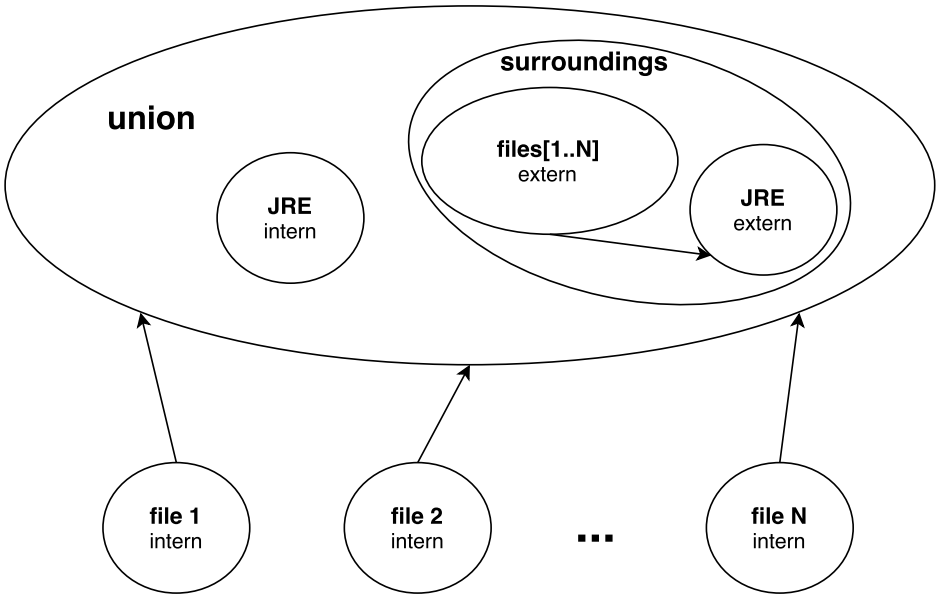
M:N komparátor používá poměrně sofistikovanou soustavu loaderů pro načítání bytecode. Pro lepší představu je přiložen obrázek (viz orázek 4.2), který graficky znázorňuje výsledek inicializace soustavy loaderů M:N komparátoru. Šipka na obrázku značí vazbu na rodičovský loader [1]. Popisky *intern* a *extern* slouží pro odlišení interních a externích tříd (více viz kapitola 3.1.1).

Aby bylo možné rekonstruovat libovolnou třídu analyzované komponentové aplikace, je nejprve nutné vytvořit loader, který bude vytvářet externí třídy všech komponent a Java JRE - minimálně je každá třída potomkem třídy `Object`. Vytvoří se tedy loader JRE a poté loader všech tříd, který použije loader JRE jako rodiče (na obrázku jako **surroundings**).

Loader *surroundings* je nyní třeba sjednotit s loaderem JRE načítajícím nyní ovšem třídy v interní podobě (na obrázku jako **JRE, intern**). Důvod je takový, že nechceme, aby třídy Java JRE byly předmětem analýzy kompatibility. Pokud se někde při zpracování třídy objeví reference na cizí třídu, která je ovšem dostupná jako interní, neoznačí se za import a nestane se tak předmětem vyhodnocení kompatibility. Výsledkem sloučení loaderů je loader, který umožňuje rekonstruovat libovolnou třídu analyzované komponentové aplikace a znemožňuje ověření kompatibility s Java JRE (na obrázku jako **union**).

Poslední fází je vytvoření loaderů interních tříd pro jednotlivé komponenty (na obrázku jako **file1, file2, ... fileN**). Každému takovému loaderu je nastaven loader *union* jako rodičovský. To zaručí, že budou moci být všechny načtené třídy těmito loadery rekonstruovány a třídy Java JRE se neobjeví mezi importy. Pokud se ovšem při zpracování třídy loaderem *fileX* narazí na použití třídy z jiného loaderu *fileY*, stane se tato třída importem a posléze i součástí ověření kompatibility. Vzájemná uzavřenost těchto loaderů z hlediska dostupnosti interních tříd umožňuje vzájemné ověření kompatibility všech komponent.

[1] Pokud loader provádí zpracování třídy a narazí na použití jiné třídy, kterou sám nemůže zpracovat, deleguje požadavek na tuto použitou třídu na **rodičovský loader**. Loader lze navíc neomezeně zřetězovat.



Obrázek 4.2: Soustava loaderů M:N komparátoru

Podstata problému

Na základě analýzy algoritmu použitého uvnitř M:N komparátoru a měření paměti pomocí nástroje JVisualVM a pomocí funkcí Java API (viz kapitola 2.2.2) bylo dospěno k následujícímu závěru.

Hlavní příčinou vysokých paměťových nároků je stavovost algoritmu, která spočívá v postupné kolekci načtených tříd a výsledků porovnání. Postupně dochází k vytváření a hromadění velkého množství datových struktur, které zůstávají v paměti do doby, než M:N komparátor doběhne. Načtené třídy se sbírají z důvodu zvýšení efektivity - bez jejich sběru by se každá třída musela načítat tolikrát, kolikrát by byla použita. A výsledky porovnání se sbírají proto, aby mohly být na konci zpracování zobrazeny uživateli v ucelené formě.

Prvotní snahou bylo upravit algoritmus tak, aby pracoval iterativně a řešil vždy pouze vzájemnou kompatibilitu dvou komponent. Takové řešení by bylo dobře škálovatelné a dalo by se paralelizovat. Tento princip izolovaného zpracování dvojic komponent ovšem není možné realizovat, protože je narušen potenciální potřebou rekonstrukce některé ze tříd určených ke komparaci. Tato rekonstrukce může vyžadovat načtení dalších komponent a tudíž porušení principu izolovaného zpracování.

Dalším možným přístupem je aplikace návrhového vzoru *lazy loading*, který by byl v tomto případě realizován pomocí *proxy* třídy [2] implementující rozhraní `JClass` a uchovávající pouze odkaz na soubor s odpovídajícím bytecode [14]. Vždy, když by se nad instancí této třídy zavolala některá z metod rozhraní `JClass`, provedla by se analýza bytecode, vytvořila se plnohodnotná implementace `JClass` a na tuto instanci by se delegoval původní požadavek. Nevýhodou tohoto přístupu je, že každý dotaz nad třídou vyvolá její opakované načtení z bytecode. Proto je tato myšlenka v následující kapitole dále rozvinuta.

Navrhované řešení

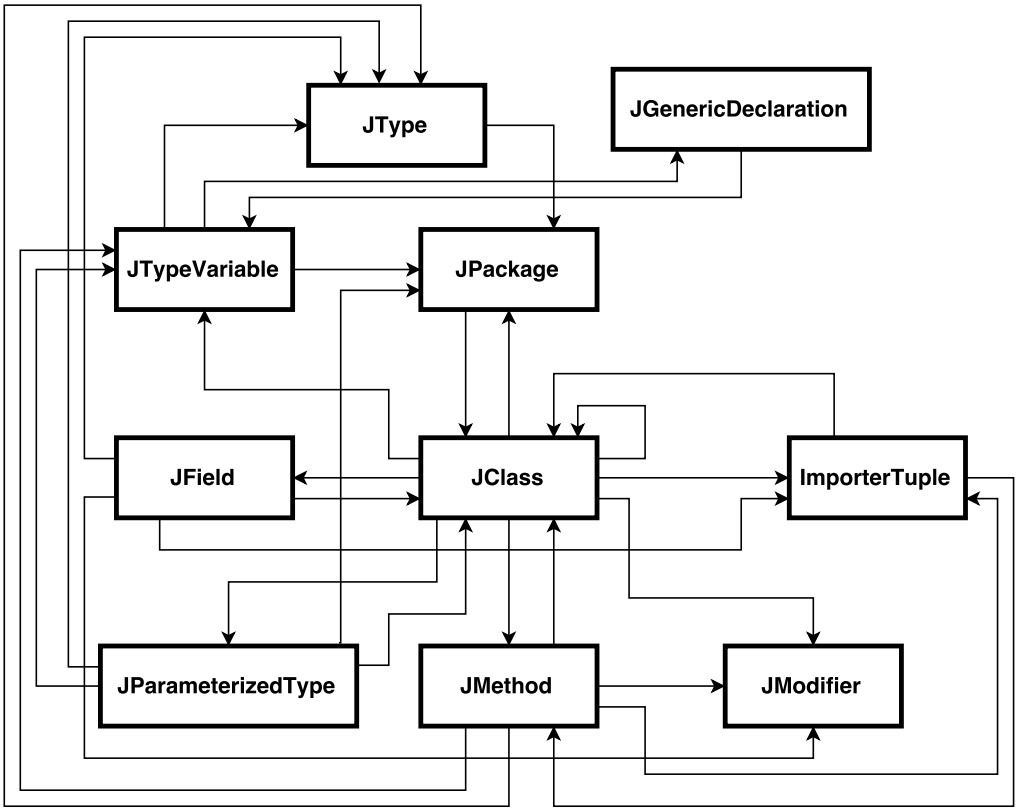
Původní myšlenka aplikace návrhového vzoru *lazy loading* z předchozí kapitoly spočívá v opakovaném načítání třídy při každém dotazu. Provádět opakované načítání bytecode, jeho analýzu a vytvářet odpovídající reprezentaci třídy v paměti znovu a znovu je ovšem zbytečně složité. Lepší postup se zakládá na uložení jednou vyreprezentované třídy na disk pomocí serializace. Při dalším dotazu na třídu pak není třeba znovu načítat bytecode a provádět jeho analýzu, ale třída je přímo deserializována z disku do paměti a připravena k použití.

Protože jsou datové struktury datového modelu nástroje JaCC mezi sebou velmi hustě provázány (viz obrázek 4.3), je třeba za účelem přidání podpory pro serializaci tříd upravit téměř celý datový model.

Dále je třeba nějakým způsobem spravovat načtené třídy a udržovat v paměti alespoň minimální pracovní množinu tříd za účelem zvýšení výkonnosti programu. Spravovat načtené třídy manuálně by bylo nešikovné. Lepší je přenést zodpovědnost za tuto funkcionalitu na některé z dostupných řešení mezipaměti, které ji již implementuje.

K dispozici je celá řada Java knihoven, které toto nabízí. Tyto knihovny poskytují automatickou správu paměti a implementují nejrůznější algoritmy pro rozhodování, jaké objekty udržovat nadále v paměti a jaké odklízet na disk. Pro tyto potřeby byla vzhledem k open-souce licenci, kvalitní dokumentaci a ověřenému řešení zvolena knihovna Ehcache [6].

[2] Návrhový vzor **proxy** se zakládá na použití zástupného objektu, který navenek nabízí stejné rozhraní jako původní objekt, ale jeho chování je možné přizpůsobit dle potřeb bez nutnosti změny okolí. Jednotlivá volání nad tímto objektem lze potom předávat dále, ignorovat je či s nimi jiným libovolným způsobem nakládat.



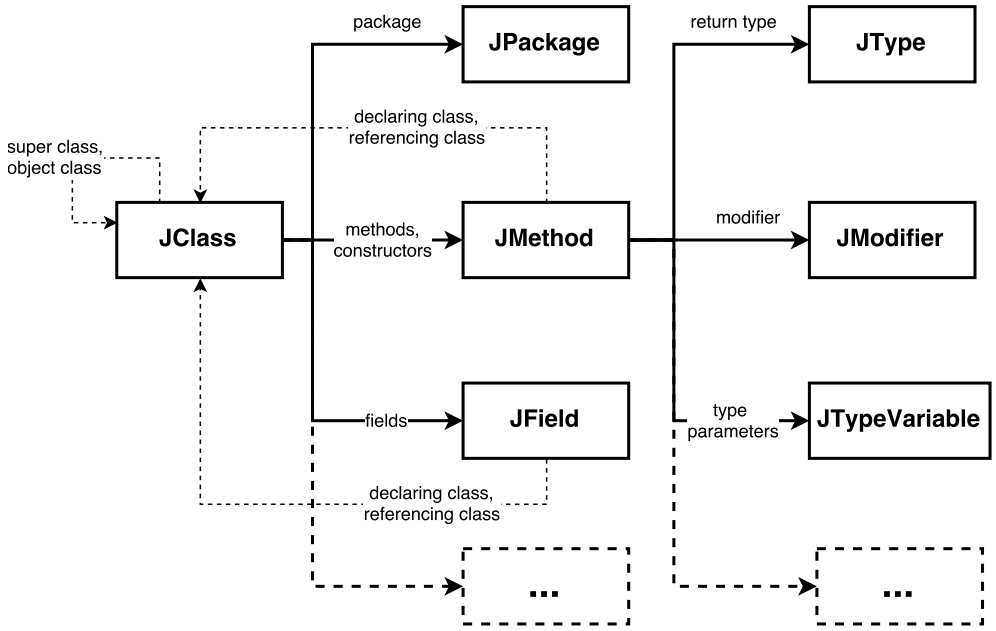
Obrázek 4.3: Závislost datových typů datového modelu JaCC

4.2 Analýza řešení

Serializovatelnost javatypes

Knihovna `Ehcache` umožňuje používat pevný disk jako rozšiřující úložnou vrstvu operační paměti. K ukládání a čtení z disku používá serializaci a deserializaci. Všechny datové typy, které jsou určeny pro de/serializaci, musí podle konvence programovacího jazyka Java implementovat značkovací rozhraní `Serializable`.

Kořenovým elementem datového modelu JaCC, který bude serializován a deserializován na disk, je implementace rozhraní `JClass`. Tato implementace tedy bude muset být označena značkovacím rozhraním `Serializable`. K tomuto datovému typu se ovšem vážou prostřednictvím atributů další datové typy a na ně se opět vážou prostřednictvím jejich atributů další datové typy atd.(viz obrázek 4.3). Pokud bychom tedy vytvořili pomyslný strom závislostí datových typů s kořenem `JClass`, pak všechny datové typy, které se v tomto stromu objeví, musí být serializovatelné, aby mohlo dojít k serializaci celého stromu datových typů (viz obrázek 4.4). O problematice serializace `javatypes` se zmiňuje již J.Rinkes ve své diplomové práci z roku 2015[9]. Při implementaci lze tedy čerpat i z tohoto materiálu.



Obrázek 4.4: Strom závislostí `javatypes`

Lazy loading

Základním elementem, se kterým nástroj JaCC pracuje, je jeho vnitřní reprezentace načtené třídy pomocí datového typu `JClass`. Všechny ostatní datové typy datového modelu JaCC s tímto elementárním datovým typem nějakým způsobem souvisí.

Jak se podle analýzy nástroje ukázalo, problém s pamětí vzniká především z toho důvodu, že v průběhu výpočtu vzniká velké množství instancí tohoto datového typu a tudíž i dalších instancí jiných datových typů, které se na něj pojí.

Řešením tohoto problému je aplikace návrhového vzoru lazy loading s využitím proxy objektů (viz 4.1), který spočívá v uvolnění zdroje až ve chvíli, kdy je potřeba. Zdrojem je v tomto případě datový typ `JClass` a potřeba ho je vždy, když se k němu přistupuje prostřednictvím jeho metod.

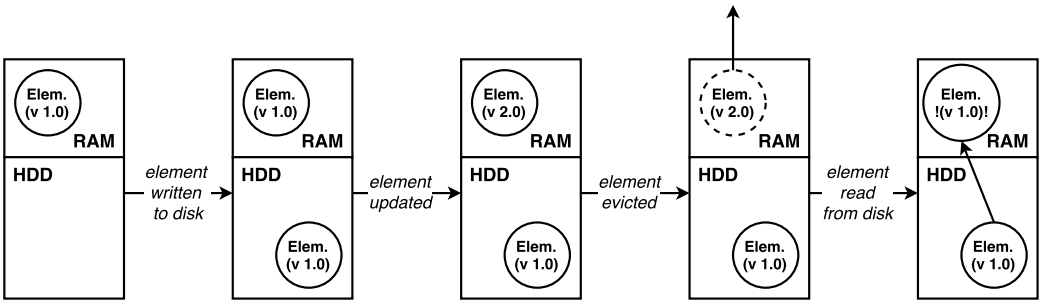
Bude tedy třeba připravit třídu, která bude obalovat cachovanou instanci `JClass` a zprostředkovávat přístup k jejím metodám pomocí návrhového vzoru lazy loading.

Konkurenční přístup

Každá cache, která má nakonfigurovaný *DiskStore* (viz kapitola 2.3.1), provádí zápis nově přidaných elementů na disk asynchronně. Nově vkládané elementy do cache pomocí operace `put()` se umist'ují do zásobníku příslušné cache. Každé cache odpovídá jedno samostatné vlákno, které běží paralelně s hlavním vláknem aplikace a které postupně odebírá jednotlivé elementy ze zásobníku a provádí jejich zápis, respektive serializaci, na disk.

Knihovna Ehcache ovšem už nehlídá, zda se s elementem v době zápisu na disk nějakým způsobem nemanipuluje. Je tedy na vývojáři, aby zajistil, že objekt nebude v době serializace modifikován.

Elementem, který je primárně určen pro zápis na disk, je datový typ `JClass`. Původní implementace tohoto datového typu v podobě `JClassImpl` ovšem neobsahuje žádnou ochranu proti konkurenčnímu přístupu. Bude tedy třeba třídu upravit, či vytvořit novou implementaci rozhraní `JClass` tak, aby byl zajištěn výlučný přístup.



Obrázek 4.5: Aktualizace elementu v cache

Aktualizace elementů

Knihovna Ehcache provádí ukládání elementů po operaci `put` na disk v podobě, která odpovídá poslednímu známému stavu tohoto elementu předtím, než došlo k jeho asynchronnímu zápisu na disk. Od chvíle, kdy vznikne kopie elementu na disku, se všechny další změny (volání setterů, přímá úprava atributů atd.) týkají pouze původní instance uložené ve fyzické paměti. Ehcache se pak už žádným způsobem nestará o propagaci těchto změn zpět na disk, kde se nachází kopie pozměněného elementu ovšem v jeho původní verzi. Pokud tedy dojde k vyklizení elementu z fyzické paměti (počet elementů uchovávaných v paměti může být omezen) a následně k opětovnému načtení téhož elementu z disku, dostaneme element v nižší verzi, než bychom mohli očekávat (viz obrázek 4.5 výše).

Možným způsobem řešení je volání operace `put` pokaždé, když se nějakým způsobem změní stav elementu. Zápis na disk, který s touto operací vždy souvisí, je ovšem drahá operace, a proto se nejedná o optimální přístup.

Lepším způsobem řešení je uložení informace o změně stavu elementu a odchycení události vyklizení elementu z paměti. Pokud element bude mít při vyklizení nastaven příznak změny, dojde nad ním explicitně k zavolání operace `put`. Tento přístup zajistí práci vždy s poslední dostupnou verzí elementu.

Zavedení Ehcache

Aby bylo vůbec možné provádět serializaci a deserializaci `javatypes`, je třeba upravit implementaci nástroje JaCC a přidat podporu pro správu cache z knihovny Ehcache.

Přidání této podpory spočívá ve vytvoření prostředku, který bude uchovávat instance jednotlivých cache a umožní k těmto instancím přistupovat. V ideálním případě by mělo být možné přístup k instancím cache zprostředkovat odkudkoliv bez nutnosti vytváření vazeb na tyto cache.

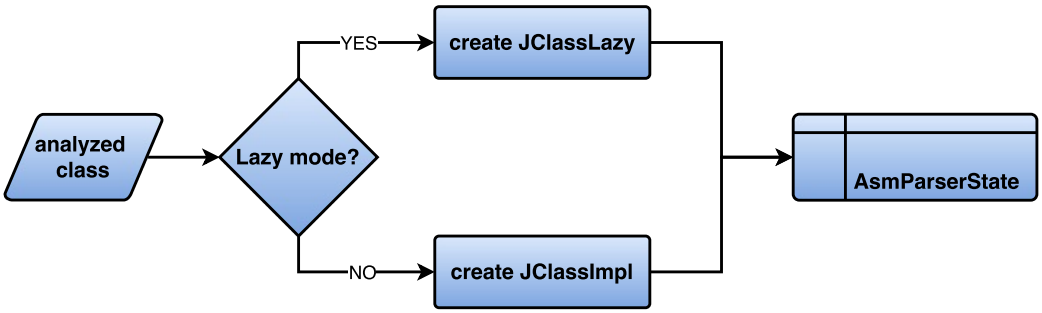
Konfigurace cache by měla být řešena pomocí XML souboru. V rámci úpravy konfigurace cache potom lze tento konfigurační soubor jednoduše editovat bez nutnosti opakovaného sestavení - případ, kdy je konfigurace zanesena přímo do zdrojového kódu.

Parametrizace loaderu

Hlavní třídou, která se stará o parsování bytecode a vytváření datového modelu JaCC, je třída `AsmDataParser`. Tato třída vytváří nové instance rozhraní `JClass` v podobě instancí třídy `JClassImpl` a tyto nově vytvořené instance ukládá do svého úložiště reprezentovaného instancí interní třídy `AsmParserState`.

V rámci rozšíření funkcionality nástroje JaCC o možnost ukládání cachovaných instancí rozhraní `JClass` v podobě `JClassLazy` instancí bude třeba upravit implementaci této části nástroje JaCC tak, aby bylo možné při vytváření nové instance `AsmDataParser` specifikovat typ nově vytvářených instancí rozhraní `JClass`. Na diagramu aktivit 4.6 je zobrazen nový způsob vytváření instancí datového typu `JClass`.

Dále bude nutné upravit implementaci třídy `AsmParserState` tak, aby se maximalizovalo množství informace, která bude součástí cachování, a minimalizovalo množství informace, která bude zůstávat ve fyzické paměti.

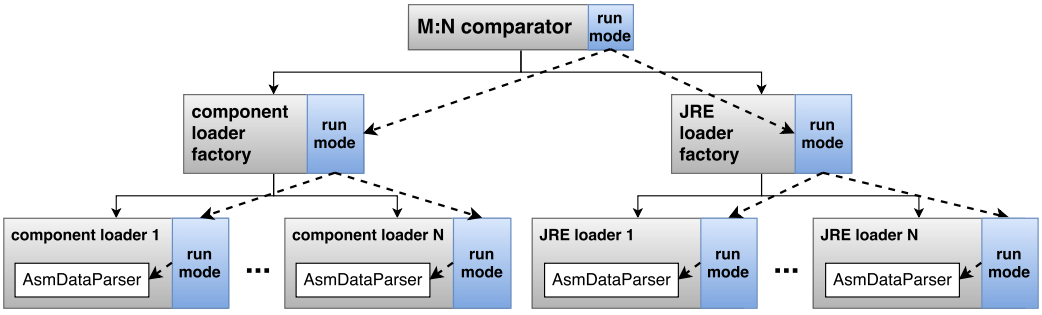


Obrázek 4.6: Parametrizované vytváření `JClass`

Finalizace

V rámci finalizace optimalizace nástroje JaCC bude nutné upravit stávající implementaci tohoto nástroje a integrovat nový způsob vytváření a správy cachovaných instancí `JClass` rozhraní do M:N komparátoru. To vše při zachování původní funkcionality.

Tato integrace se bude zakládat převážně na parametrizaci jednotlivých vrstev nástroje JaCC tak, aby bylo možné specifikovat režim běhu M:N komparátoru (původní/lazy) a informace o režimu běhu se dostala přes továrny na loadery použité při inicializaci M:N komparátoru až k jednotlivým loaderům, které interně používají instance `AsmDataParser` pro zpracování bytecode. Jednotlivé vrstvy, kterých se tato změna dotkne, jsou schématicky znázorněny na následujícím obrázku 4.7. Spojení klíčových slov *run mode* zastupuje režim běhu, pro nějž je nutné zajistit podporu na všech vrstvách.



Obrázek 4.7: Propagace režimu běhu v M:N komparátoru

4.3 Implementace řešení

Serializovatelnost javatypes

V rámci zajištění serializovatelnosti datového typu `JClass` byly označeny značkovacím rozhraním `Serializable` všechny následující třídy a rozhraní modulu `javatypes`.

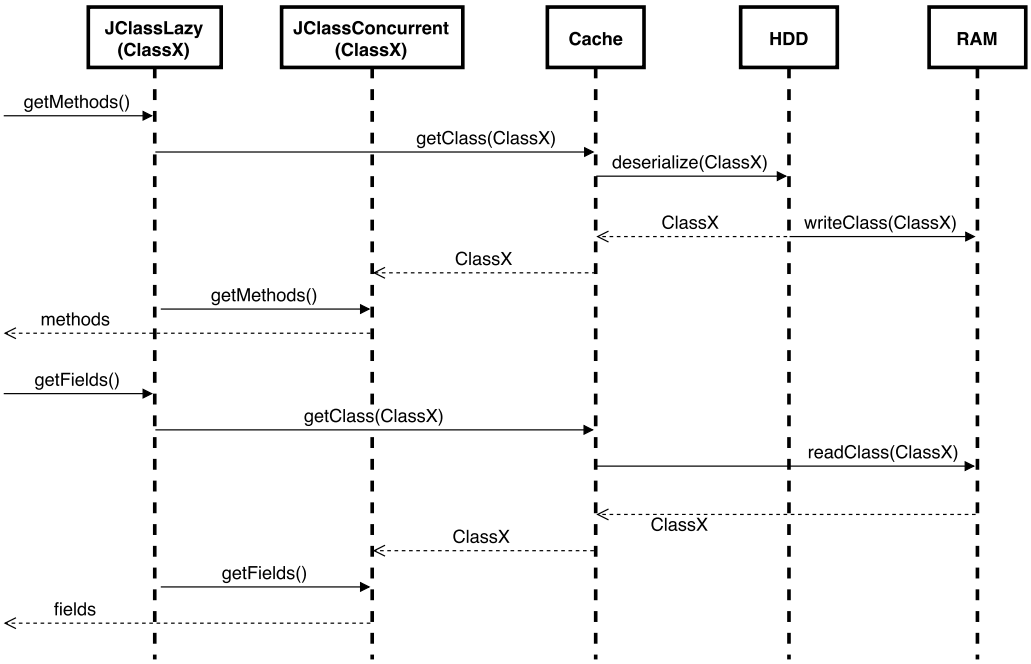
- **JType, JPackage, JModifier**
- **JGenericDeclaration, JBlackBoxComponent**
- **JAnnotation, JAnnotatedElement**
- **CanBeImported, ImporterTuple**

Všechny ostatní třídy a rozhraní z modulu `javatypes` již implementovaly toto značkovací rozhraní, nebo dědily od rozhraní či tříd, které již byly serializovatelné.

Dále byla za účelem vyšší výkonnosti serializace a deserializace implementována obalovací třída `KryoSerializationWrapper`, která umožňuje uchovávat libovolný objekt a tento objekt serializovat a deserializovat s využitím knihovny **Kryo** [10]. Knihovna Kryo dokáže provádět serializaci a deserializaci výrazně rychleji než standardní Java API [11]. Toto tvrzení bylo ověřeno i měřením, které ukázalo zkrácení doby běhu přibližně na polovinu.

Lazy loading

Byla implementována proxy třída `JClassLazy`, která obsahuje pouze název třídy, kterou obaluje a identifikátor cache, která obalovanou třídu spravuje. Třída `JClassLazy` implementuje všechny metody rozhraní `JClass` a obsluhu těchto metod deleguje dále na obalovanou třídu.



Obrázek 4.8: Lazy loading

Na obrázku 4.8 výše se nachází sekvenční diagram, který zachycuje aplikaci návrhového vzoru lazy loading na ukázkovém příkladu v nástroji JaCC. Nejprve přijde požadavek `getMethods()` na získání metod třídy *ClassX*, která je reprezentována instancí proxy třídy `JClassLazy`. Tato proxy třída se nejprve dotáže odpovídající cache na tuto třídu. Cache zjistí, že se třída nachází na pevném disku (HDD), provede deserializaci této třídy a vrátí její instanci. Cache si současně ponechá deserializovanou třídu (instance třídy `JClassConcurrent`) ve fyzické paměti (RAM). Následně se nad deserializovanou třídou pokračuje v obsluze původního požadavku `getMethods()` tím způsobem, že se nad ní zavolá také tato metoda. Získaný výsledek z deserializované instance třídy `JClassConcurrent` se potom vrátí jako odpověď na původní požadavek.

V druhé vertikální polovině diagramu se nachází velice podobný postup obsluhy požadavku `getFields()`. Tento postup se odlišuje pouze v tom, že se již skutečná instance třídy `JClassConcurrent` nachází v paměti a může se k ní přistupovat přímo.

Konkurenční přístup

S ohledem na zachování výkonnosti původního řešení byla vytvořena nová implementace rozhraní `JClass` v podobě třídy `JClassConcurrent`. Ochrana před konkurenčním přístupem se totiž téměř vždy pojí s určitou režií a rozšíření implementace `JClassImpl` o tuto ochranu by tak měla dopad na výkon původního řešení nástroje.

Pro zajištění výlučného přístupu k instanci třídy `JClassConcurrent` v době zápisu a čtení z disku a také v době modifikace této instance bylo použito standardní vysokoúrovňové synchronizační primitivum *monitor*. Použití tohoto synchronizačního prvku je v Javě realizováno pomocí klíčového slova `synchronized`.

Jako `synchronized` byly proto označeny všechny metody třídy `JClassConcurrent`, které nějakým způsobem pozměňují stav objektu. Dále bylo klíčové slovo `synchronized` zaneseno do třídy `KryoSerializationWrapper`, a to konkrétně do metody `writeObject()`, kde dochází k serializaci instance obalované třídy - `JClassConcurrent` (viz fragment kódu 4.9).

Protože se uvnitř stejného monitoru nachází po provedených úpravách metody pro změnu stavu objektu i jeho serializaci, je zajištěno vzájemné vyloučení těchto aktivit.

```
1 private synchronized void writeObject(java.io.ObjectOutputStream out)
2         throws IOException {
3
4     synchronized (object) {
5         Output output = new Output(out);
6         kryos.get().writeClassAndObject(output, object);
7         output.close();
8     }
9 }
```

Obrázek 4.9: Metoda `writeObject()` třídy `KryoSerializationWrapper`

Aktualizace elementů

Nejprve bylo připraveno jednoduché rozhraní `CanBeCached` se 2 metodami:

1. **void setUpdated(boolean updated)**
   Nastavení příznaku o změně stavu objektu.
2. **boolean isUpdated()**
   Získání hodnoty příznaku změny objektu.

Toto rozhraní bylo implementováno ve třídě `JClassConcurrent` a do všech metod, které pozměňují stav objektu (settery a metody, které pozměňují atributy třídy `JClassConcurrent`), byla vložena řádka kódu, která zajišťuje nastavení příznaku změny na `true`.

Poté se bylo třeba napojit na událost vyklizení elementu z fyzické paměti. Ehcache bohužel nepodporuje použití posluchače, který by tuto událost odchytával. Nabízí ovšem možnost dynamické konfigurace cache a nastavení vlastní instance třídy, která bude zodpovědná za výběr elementu k vyklizení z paměti, a tudíž bude mít i přístup k vyklízenému elementu.

Ehcache nabízí třídy, které implementují tuto rozhodovací logiku s využitím cachovacích algoritmů *FIFO* (**F**irst **I**n, **F**irst **O**ut), *LRU* (**L**east **R**ecently **U**sed) a **LFU** (**L**east **F**requently **U**sed). Algoritmus FIFO představuje standardní frontu a vyřazuje položky ve stejném pořadí, ve kterém byly do cache přidány. Algoritmus LRU se dívá na položky z pohledu doby, po kterou k nim nebylo přistupováno, a vyřazuje nejstarší položky. Algoritmus LFU nahlíží na položky z pohledu frekvence použití a vyřazuje ty nejméně používané. Protože je třeba v paměti udržovat především ty třídy, se kterými se často pracuje, a minimalizovat tak počet pomalých diskových operací, byl zvolen cachovací algoritmus LFU [16].

Cachovací algoritmus LFU je implementován v rámci třídy `LfuPolicy` poskytované knihovnou Ehcache. Bylo tedy třeba implementovat vlastní třídu `PutOnEvictedLfuPolicy` a oddědit ji od třídy `LfuPolicy`. Za účelem odchycení události vyklizení elementu z paměti byla přetížena hlavní metoda `selectedBasedOnPolicy()`, která vrací položku určenou k vyklizení z paměti. Její tělo bylo implementováno způsobem popsaným na následující stránce. Protože postup není triviální, je přiložen i diagram aktivit 4.10.

1. Zavoláním téže metody z rodičovské třídy `LfuPolicy` je získán element *elementToEvict* vybraný původně k vyklizení pomocí algoritmu LFU.

2. Pokud element *elementToEvict* neobsahuje instanci `CanBeCached` nebo ji obsahuje, ale její příznak změny má hodnotu `false`, je element *elementToEvict* vyklizen z paměti.

3. Pokud element *elementToEvict* obsahuje instanci `CanBeCached` a tato instance má příznak změny s hodnotou `true`, provede se změna hodnoty tohoto příznaku na `false` a element původně určený k vyklizení se znovu vloží do cache za účelem aktualizace elementu na disku. Následně se rekurzivně volá metoda `selectedBasedOnPolicy` a hledá se nový kandidát na vyklizení z paměti. Množina kandidátů na vyklizení, která je vstupem této metody, ale už neobsahuje element *elementToEvict*. Ten je v paměti držen z důvodu aktualizace jeho obrazu na disku.

Zavedení Ehcache

Za účelem zavedení podpory Ehcache do nástroje JaCC byla implementována třída `JClassCacheManager`. Tato třída poskytuje následující 4 statické metody:

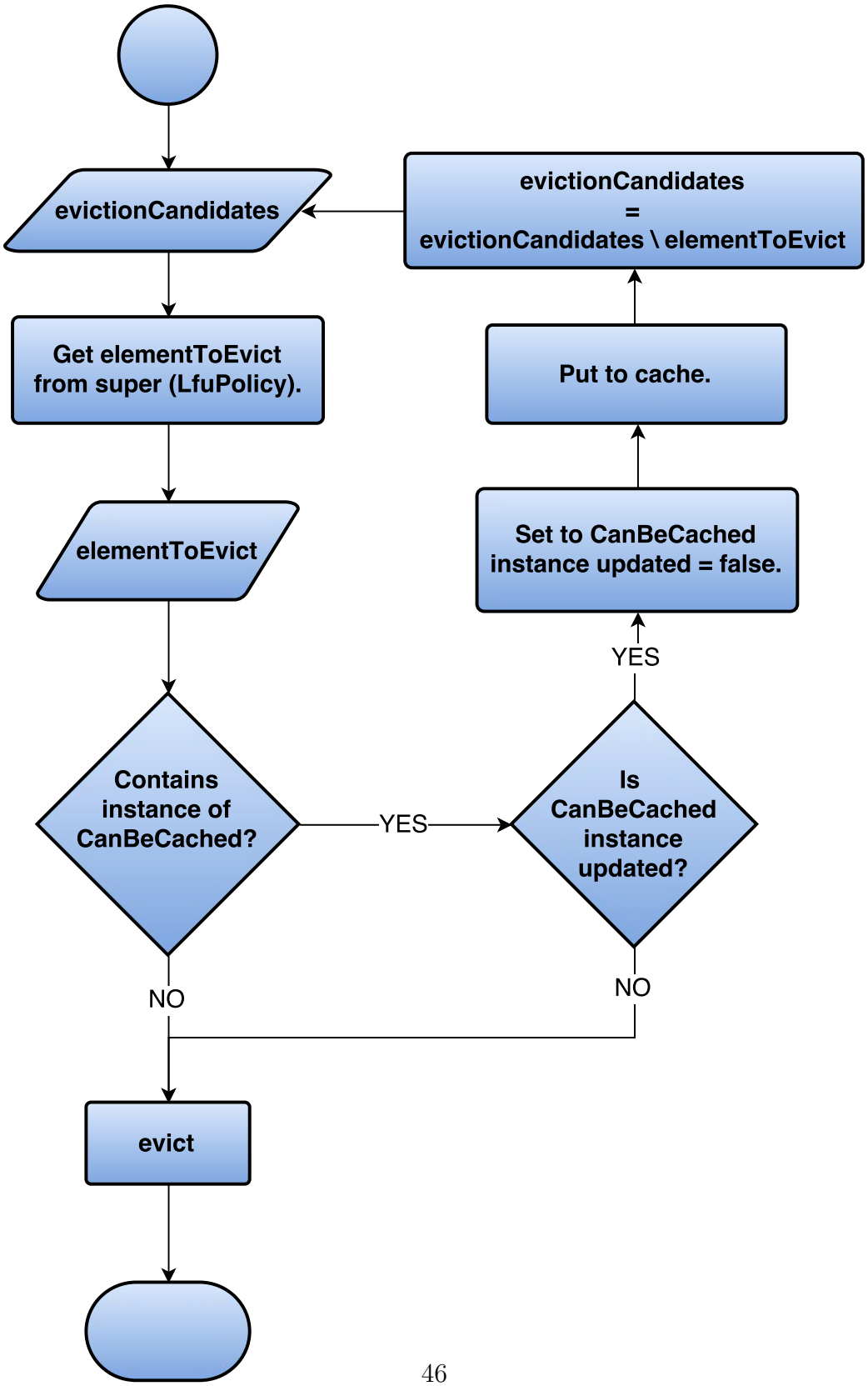
1. `CacheManager getInstance()` Tato metoda je privátní a používají ji ostatní metody. Metoda vrací s využitím návrhového vzoru *jedináček* jedinou instanci `CacheManager`. Její instance je inicializovaná s konfigurací uvedenou v konfiguračním souboru `ehcache.xml` nacházejícím se v adresáři se zdroji pro odpovídající modul.

2. `Cache createCache(String cacheId)` Vytvoří novou instanci `Cache` s ID předaným v parametru.

3. `Cache getCache(String cacheId)` Vrátí instanci `Cache` s ID uvedeným v parametru.

4. `void shutdown()` Ukončí činnost instance `CacheManager`.

Třída `JClassCacheManager` poskytuje díky svým statickým metodám centralizovaný způsob správy cache bez nutnosti vytváření nadbytečných závilostí v programu.



Obrázek 4.10: Vyklizení elementu z paměti

Parametrizace loaderu

JClassFactory Nejprve bylo nutné analyzovat třídu `AsmDataParser` a identifikovat všechny možné způsoby, pomocí kterých docházelo uvnitř této třídy k vytváření nových instancí tříd implementujících rozhraní `JClass`. Na základě výsledků této analýzy bylo implementováno rozhraní `JClassFactory`, které všechny objevené způsoby vytváření `JClass` instancí pokrývá a poskytuje následující tovární metody:

- `JClass createNullRepsentation(String name)`

  Vytvoří novou instanci třídy, která drží pouze informaci o svém jménu. Editace instance této třídy není možná.

- `JClassMutable createSimpleClassRepresentation(String name)`

  Vytvoří novou instanci třídy a inicializuje její jméno.

- `JClassMutable createNonFullyRepresented(String name)`

  Vytvoří novou instanci importované třídy a inicializuje její jméno.

- `JClassMutable createClassRepresentation(JModifier modifiers, String name, JPackage pkg, JClass superclass, String origin, boolean fromOutside`

  Vytvoří novou instanci třídy a inicializuje její modifikátory, název, balíček původu, rodičovskou třídu, název zdrojové komponenty a příznak pro externí třídu.

- `JClassMutable createArrayRepresentation(JModifier modifiers, String name, JClass componentType, int dimension)`

  Vytvoří novou instanci reprezentující pole předaných tříd *componentType* o velikosti *dimension* přístupné pomocí modifikátoru *modifiers* a pojmenované *name*.

Poznámka: V metodách výše se objevuje nový datový typ `JClassMutable`, který dědí od rozhraní `JClass` a slouží pro reprezentaci pozměnitelné třídy. S tímto datovým typem se pracuje uvnitř třídy `AsmDataParser` během rekonstrukce načtené třídy.

Původní konstrukce pro přímé vytváření jednotlivých instancí datového typu `JClass` uvnitř třídy `AsmDataParser` poté mohli být nahrazeny zprostředkovaným vytvářením tříd přes instanci `JClassFactory`.

Následují fragment kódu tento rozdíl ve vytváření tříd v původním a novém řešení zachycuje.

```
1  //1] export class creation - old way
2  JClassImpl importingClass = JClassImpl.createClassRepresentation(
3          modifier, fullClassName,new JPackageImpl(packageName),
4          null, sourcePath, state.getLoader().isInheritanceLoader());
5  //export class creation- new way
6  JClassMutable importingClass = jClassFactory.createClassRepresentation(
7          modifier, fullClassName, new JPackageImpl(packageName),
8          null, sourcePath, state.getLoader().isInheritanceLoader());
9
10 //2] simple class creation- old way
11 JClass result
12         = new JClassImpl(new JModifierImpl(Modifier.PUBLIC),className);
13 //simple class creation- new way
14 JClass result
15         = jClassFactory.createSimpleClassRepresentation(className);
16
17 //3] null class creation- old way
18 JClass r = new JNullClass(modifName);
19 //null class creation- new way
20 JClass r = jClassFactory.createNullRepresentation(modifName);
```

Obrázek 4.11: Vytváření tříd přes `JClassFactory`

Existence rozhraní `JClassFactory` umožnila jednoduše parametrizovat vytváření tříd uvnitř třídy `AsmDataParser`. Použití tohoto rozhraní bylo také rozšířeno do třídy `JClassInheritanceLoader`, jejíž instance slouží k vytváření externích tříd.

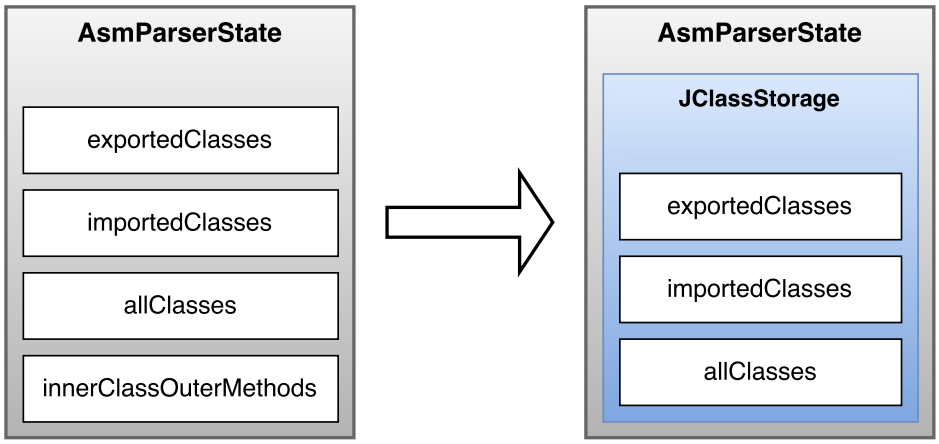
V rámci implementace rozhraní `JClassFactory` byly připraveny následující 2 třídy:

1. **Anonymní třída**

   Jedná se o anonymní třídu `DEFAULT_CLASS_FACTORY` nacházející se přímo uvnitř rozhraní `JClassFactory`. Tato implementace umožňuje vytvářet třídy stejným způsobem jako původní řešení nástroje JaCC.

2. **JClassLazyFactory**

   Každá instance této třídy vytváří při své inicializaci vlastní instanci cache. Tuto cache potom používá instance `JClassLazyFactory` pro ukládání jednotlivých tříd, které vytvoří. Metody pro vytvoření tříd pomocí `JClassLazyFactory` vracejí instance proxy tříd `JClassLazy`, které nesou údaj o umístění tříd v cache.



Obrázek 4.12: Upravená struktura třídy `AsmParserState`

JClassStorage Následovala analýza třídy `AsmParserState`, kterou používá instance třídy `AsmDataParser` jako úložiště pro načtené třídy. Předmětem analýzy byly její atributy, které sloužily pro sběr informace při načítání tříd:

1. `SortedMap<String,JClass>exportedClasses`

   Uvnitř této mapy jsou uloženy všechny načtené exportované třídy seřazené podle jména.

2. `Map<String,Set<JClass>>importedClasses`

   Uvnitř této mapy jsou uloženy všechny načtené importované třídy. Klíčem v mapě je název třídy, která importuje, a hodnotou je množina příslušných importovaných tříd.

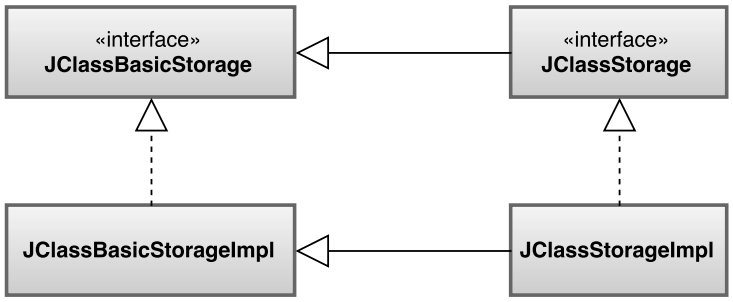
3. `Map<String,JClass>allClasses`

   Tato mapa udržuje všechny načtené třídy.

4. `Map<String,JMethod>innerClassOuterMethods`

   Jedná se o pomocnou mapu, kterou instance třídy `AsmDataParser` používá pro ukládání vnějších metod vnitřních tříd.

První tři atributy třídy `AsmParserState` byly vyčleněny a nahrazeny rozhraním `JClassStorage`, které tyto atributy zapouzdřuje a poskytuje k nim metody pro přístup a modifikaci. Čtvrtý atribut byl úplně odstraněn a informace o vnějších metodách vnitřních tříd byla přesunuta na úroveň datového



Obrázek 4.14: Rozhraní a implementace úložiště JClass

typu `JClass`. Díky tomu se tato informace stala serializovatelnou součástí datového typu `JClass` a přestala pro aplikaci představovat riziko v podobě nadbytečné datové struktury držené potenciálně ve fyzické paměti. Výsledná struktura třídy `AsmParserState` je znázorněna na obrázku 4.12 na předchozí straně. Ukázka nového použití úložiště tříd se nachází níže.

```
1  //1] add any class before
2  state.getAllClasses().put(classToAdd.getName(), classToAdd);
3  //   add any class now
4  state.classStorage.addAnyClass(classToAdd.getName(), classToAdd);
5
6  //2] add import class before
7  state.getImportedClasses().put(importingClass.getName(), imports);
8  //   add import class now
9  state.classStorage.addImportedClass(importingClass, classToAdd);
10
11 //3] get any class before
12 JClass jclass = state.getAllClasses().get(name);
13 //   get any class now
14 JClass jclass = state.classStorage.getAnyClass(name);
```

Obrázek 4.13: Správa tříd uvnitř `AsmDataParser`

Protože dochází k ukládání zpracovaných tříd i ve třídě `JClassInheritanceLoader`, bylo stejně jako v případě `JClassFactory` rozšířeno použití `JClassStorage` i do této třídy. Třída `JClassInheritanceLoader` ale nerozlišuje mezi importovanými a exportovanými třídami a ukládá třídy do jediné společné mapy. Proto bylo vytvořeno zjednodušené rozhraní nerozlišující mezi typy načítaných tříd `JClassStorageBasic`, od kterého bylo rozhraní `JClassStorage` odděděno. Pro obě tato rozhraní byla současně připravena odpovídající implementace (viz obrázek 4.14).

Finalizace

Úprava JAsmClassLoader Rozšíření `JAsmClassLoader` o podporu vytváření cachovaných tříd spočívalo v přidání konstruktoru, který na vstupu očekává instanci rozhraní `JClassFactoryCreator`.

```
1     public JAsmClassLoader(
2             final JClassBasicLoader parentLoader,
3             final DataSourceReader dataLoader,
4             final boolean inheritanceLoader,
5             final boolean exportsOnly,
6             final JClassFactoryCreator jClassFactoryCreator) { ... }
```

Toto rozhraní slouží k vytváření instancí `JClassFactory` pomocí tovární metody `JClassFactory createClassFactory()` a obsahuje dvě anonymní implementace tohoto rozhraní `DEFAULT` a `LAZY`.

```
1     JClassFactoryCreator DEFAULT = new JClassFactoryCreator() {
2         @Override
3         public JClassFactory createClassFactory() {
4             return JClassFactory.DEFAULT_CLASS_FACTORY;
5         }
6     };
7
8     JClassFactoryCreator LAZY = new JClassFactoryCreator() {
9         @Override
10        public JClassFactory createClassFactory() {
11            return new JClassLazyFactory();
12        }
13    };
```

První implementace `DEFAULT` vytváří standardní továrnu na necachované třídy a druhá implementace `LAZY` vytváří továrnu na třídy cachované. Uvnitř konstruktoru `JAsmClassLoader` pak může být vytvořen odpovídající objekt `AsmDataParser` s využitím konkrétní implementace `JClassFactoryCreator`.

```
1     this.parser = new AsmDataParser(
2                       new AsmDataParser.AsmParserState(this, exportsOnly),
3                       jClassFactoryCreator.createClassFactory()
4                   );
```

Důvodem, proč se používají továrny na instance `JClassFactory` namísto samotných instancí `JClassFactory`, je, že se každá továrna na cachované třídy pojí se svojí vlastní cache, do které nově vytvářené třídy ukládá. Pokud by se tedy vytvořila jediná továrna na cachované třídy a nechala se sdílet mezi loadery, velice rychle by začalo docházet ke konfliktům při umisťování nově vytvořených tříd do společné cache.

Továrna na cachované loadery Druhou vrstvou, do které bylo třeba přidat podporu pro cachované vytváření tříd, byly jednotlivé továrny na loadery.

Tvorbu JRE loaderů zprostředkovávají třídy `JClassExportsLoaderFactory` a `JClassLoaderFactory`, které poskytují statické tovární metody pro jejich vytvoření. V rámci rozšíření funkcionality těchto továren o možnost vytvoření cachovaných loaderů JRE byla upravena implementace těchto tříd a do každé z nich byla přidána statická tovární metoda `createCachedJreLoader()`:

```
1     public static JClassLoader createCachedJreLoader(
2             final JClassBasicLoader parentLoader,
3             final String... pcgFilter)
4         File[] files
5             = OracleJreJars.getJreJars(OracleJreJars.getJavaHome());
6
7         return new JAsmClassLoader(
8                 parentLoader,
9                 new JarMemoryCachedReader(
10                        Arrays.asList(pcgFilter),files),
11                false,
12                false,
13                JClassFactoryCreator.LAZY);
14    }
```

K tvorbě loaderů pro jednotlivé komponenty je v původním řešení použita anonymní implementace rozhraní `JClassLoaderCreator` `VARIABLE_DS_LOADER`. Za účelem rozšíření funkcionality o podporu cachovaných tříd byla opět implementována nová anonymní třída `CACHED_VARIABLE_DS_LOADER`, která přetěžovala metody `create()` a `createExports()`:

```
1  @Override
2  public JClassExportsLoader createExports(
3      final JClassBasicLoader parentLoader, final File... data) {
4      try {
5          DataSourceReader reader = new VariableDsReader(data);
6          return new JAsmClassLoader(
7                      parentLoader, reader, false, true,
8                      JClassFactoryCreator.LAZY);
9          } catch (IOException e) {
10          throw new JaccRuntimeException("Problem to open file.", e);
11     }
12  }
13
14 @Override
15 public JClassLoader create(
16     final JClassBasicLoader parentLoader, final File... data) {
17     try {
18         DataSourceReader reader = new VariableDsReader(data);
19         return new JAsmClassLoader(
20                     parentLoader, reader, false, false,
21                     JClassFactoryCreator.LAZY);
22         } catch (IOException e) {
23          throw new JaccRuntimeException("Problem to open file.", e);
24     }
25 }
```

Konfigurační modul M:N komparátoru

K vytvoření instance původního M:N komparátoru je použita statická tovární metoda `getApiInterCompatibilityChecker()` ze třídy `ApiCheckersFactory`. Stejným způsobem proto byla do této třídy přidána statická tovární metoda `getCachedApiInterCompatibilityChecker()`, která slouží pro vytvoření cachované instance M:N komparátoru.

```
1 public static ApiInterCompatibilityChecker<File>
2     getCachedApiInterCompatibilityChecker(
3     final ApiCheckersSetting setting) {
4
5     return Guice.createInjector(new ApiInterCmpModule(setting, true))
6             .getInstance(ApiInterCompatibilityChecker.class);
7 }
```

Statická tovární metoda výše používá framework **Guice** [20] pro konfiguraci nové instance M:N komparátoru prostřednictvím návrhového vzoru dependency injection. Tuto konfiguraci zprostředkovává konfigurační modul reprezentovaný třídou `ApiInterCmpModule`. Implementace tohoto modulu byla upravena a jeho konstruktor byl rozšířen o druhý parametr, který umožňuje specifikovat režim běhu (původní/lazy). Třída `ApiInterCmpModule` si informaci o režimu běhu ponechává nově v logickém atributu `lazyMode` (viz výše).

Třída `ApiInterCmpModule` dědí od abstraktní třídy `AbstractModule` a implementuje jedinou abstraktní metodu `void configure()`. Tato metoda se implicitně volá během realizace dependency injection a slouží pro namapování implementací na příslušné rozhraní či konfiguraci pojmenovaných parametrů. První řádek metody `configure()` byl upraven na:

```
1 install(lazyMode ? new ApiCachedLoaderModule():new ApiLoaderModule());
```

Na základě hodnoty atributu `lazyMode` je vybrán další modul zprostředkovávající rozšířenou konfiguraci. Modul `ApiLoaderModule` byl ponechán a byl použit pro konfiguraci původních továren na loadery pro M:N komparátor.

V rámci zachování původní funkcionality byl proto implementován nový modul `ApiCachedLoaderModule`, který se implementačně odlišuje od původního modulu `ApiLoaderModule` jen v následujících dvou metodách:

```
@Provides
@Named("appLoaderCreator")
JClassLoaderCreator<File> getInheritanceLoaderCreator() {
    return JClassLoaderCreator.CACHED_VARIABLE_DS_LOADER;
}

@Provides
@Named("jreLoaderCreator")
JClassLoaderCreator<String> getJreLoaderCreator() {
    return new JClassLoaderCreator<String>() {

        @Override
        public JClassExportsLoader createExports(
            final JClassBasicLoader parentLoader,
            final String... data) {

            return JClassExportsLoaderFactory
                    .createCachedJreLoader(parentLoader, data);
        }

        @Override
        public JClassLoader create(
            final JClassBasicLoader parentLoader,
            final String... data) {

            return JClassLoaderFactory
                    .createCachedJreLoader(parentLoader, data);
        }
    };
}
```

Tyto metody zajišťují konfiguraci pojmenovaných továren na loadery *appLoaderCreator* a *jreLoaderCreator*. Stejnojmenné továrny na loadery jsou použity i uvnitř M:N komparátoru při inicializaci jeho soustavy loaderů. O toto dodatečné nastavení závislostí pomocí dependency injection se stará knihovna Guice. S použitím cachovaných továren na loadery uvnitř těchto implementovaných metod je tak dokončen proces integrace cachovaného řešení do M:N komparátoru.

4.4 Testování

V rámci otestování nové funkčnosti související s integrací Ehcache, tedy vytvářením a správou cachovaných tříd, bylo nutné upravit implementaci nástroje JaCC.

Cílem bylo upravit konfiguraci nástroje JaCC tak, aby nástroj JaCC pracoval ve výchozím stavu v lazy režimu. Protože tento režim běhu nikterak neovlivňuje funkčnost nástroje, je možné použít pro otestování jeho správné funkčnosti původní sadu testů. Za účelem změny této výchozí konfigurace nástroje JaCC byla dočasně upravena třída `JAsmClassLoader`, která je těžištěm celého nástroje. Jako výchozí režim běhu tohoto loaderu byl nastaven režim lazy. K explicitnímu nastavení režimu běhu třídy `JAsmClassLoader` docházelo pouze uvnitř M:N komparátoru, kde byl ovšem taktéž použit režim lazy. Celý nástroj tedy od tohoto okamžiku běžel v režimu lazy a bylo možné otestovat jeho funkčnost.

Před vlastním spuštěním testů bylo třeba upravit jejich implementaci. Několik testů vytvářelo konkrétní instance rozhraní `JClass` v podobě instancí třídy `JClassImpl`. Uvnitř těla testu se potom takové instanci nastavily hodnoty některých jejích atributů a takto uměle vytvořená rekonstrukce očekávané třídy se přímo porovnávala se třídou vrácenou z loaderu. Protože ovšem třída navrácená z loaderu byla instancí třídy `JClassLazy`, test selhával z důvodu nerovnosti datových typů. Testy proto byly upraveny tak, aby byly nezávislé na režimu běhu nástroje JaCC. Na místech, kde původně docházelo k přímému vytváření instancí třídy `JClassImpl`, nově docházelo k vytváření tříd zprostředkovaně přes instanci třídy `JClassFactory`. Tato instance se získávala z použitého loaderu uvnitř testu. Tímto způsobem se zaručilo vytváření odpovídajících typů tříd uvnitř testů.

V testech se dále také objevovala místa, kde docházelo k volání metod poskytovaných pouze třídou `JClassImpl`. Aby bylo možné volat stejné metody i nad instancí cachované třídy `JClassLazy`, byly tyto metody přesunuty do některého ze společných rozhraní tříd `JClassImpl` a `JClassLazy`. V testu se potom použilo typování na toto společné rozhraní namísto původní konkrétní implementace.

Po těchto úpravách mohlo dojít ke spuštění testů, které ověřily správnou funkčnost nového řešení nástroje JaCC.

4.5 Experiment

Za účelem zhodnocení výsledků, které optimalizace nástroje JaCC přinesla, byla provedena některá měření. Tato měření sledovala využití paměti a výkonnost původní necachované verze (dále jako *původní*) nástroje JaCC a nové cachované verze (dále jako *lazy*) nástroje JaCC. Na základě těchto měření potom mohly být obě verze porovnány.

Měření byla prováděna na počítači s následujícími parametry:

- **Operační systém** OS X El Capitan verze 10.11.4,
- **Java JDK** Oracle Java SE 1.8.0_60,
- **Procesor** Intel Core i5 2,7 GHz,
- **Paměť** 8GB 1867 MHz DDR3,
- **Úložiště** SSD 128 GB.

V rámci měření byla připravena jednoduchá testovací konzolová aplikace, jejíž instance se spouštěly s parametrem JVM `-Xmx6g` a dalšími 3 parametry:

1. **soubor s aplikačními JAR**

   Textový soubor, který na jednotlivých řádkách obsahuje cesty k aplikačním JAR.

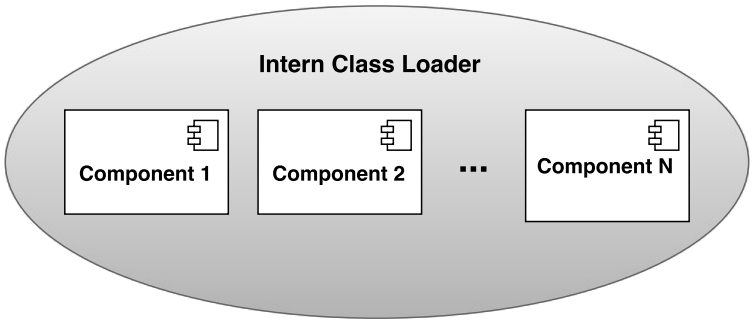
2. **soubor s knihovními JAR**

   Textový soubor, který na jednotlivých řádkách obsahuje cesty ke knihovním JAR.

3. **režim běhu**

   Kombinace znaku 'D', či 'L' a čísla 1-3. Znak 'D'(default) se používá pro spuštění testovací instance původního řešení a znak 'L' (lazy) pro spuštění testovací instance optimalizovaného řešení. Jednotlivá čísla zastupují úroveň testované funkčnosti M:N komparátoru (více dále).

Tato aplikace měřila dobu běhu a aktuální využití paměti před a po explicitně volané garbage kolekci. Měření množství využité paměti probíhalo na místě s nejvyššími očekávanými paměťovými požadavky (ověřeno měřením).



Obrázek 4.15: Loader interních tříd

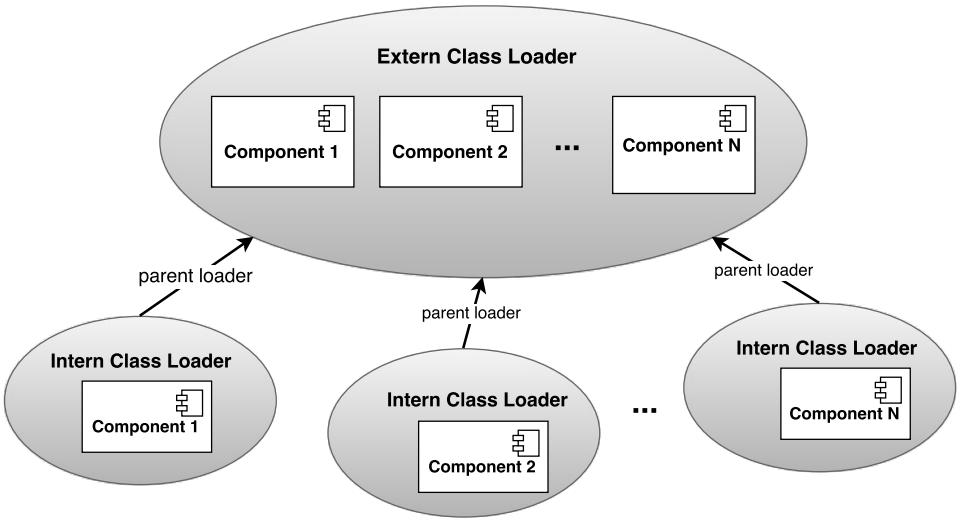
M:N komparátor vytváří vedle standardních interních tříd také třídy externí. V původním řešení nástroje JaCC se stejnojmenná interní a externí třída odlišuje pouze v jediném logickém atributu a zbytek třídy je společný (viz 3.1.1). V optimalizované verzi je ovšem každá třída samostatně serializována, a proto má i každá třída svoje vlastní tělo, které nesdílí s žádnou jinou třídou. Pokud se tedy v paměti objevuje interní třída společně se svým externím protějškem, původní řešení je z hlediska množství využité paměti efektivnější. Existence externích tříd tedy negativně ovlivňuje vliv optimalizace na M:N komparátor.

Druhým faktorem, který negativně ovlivňuje míru zlepšení paměťových požadavků optimalizovaného řešení, je počet udržovaných výsledků komparace. Protože tyto výsledky nejsou ukládány na disk, představují opět potenciál pro snížení efektu optimalizace.

Za účelem ověření těchto hypotéz a posouzení vlivu optimalizace na další typické případy užití nástroje JaCC (vedle M:N komparátoru - **scénář (c)**) byly připraveny další 2 testovací scénáře:

(a) loader interních tříd Jedná se o jednoduchý loader, který slouží pouze pro načítání interních tříd. Protože jsou všechny tyto třídy ukládány na disk a na disku nevzniká žádná duplicita, efekt optimalizace by zde měl být nejvyšší. Loader je znázorněn na obrázku 4.15.

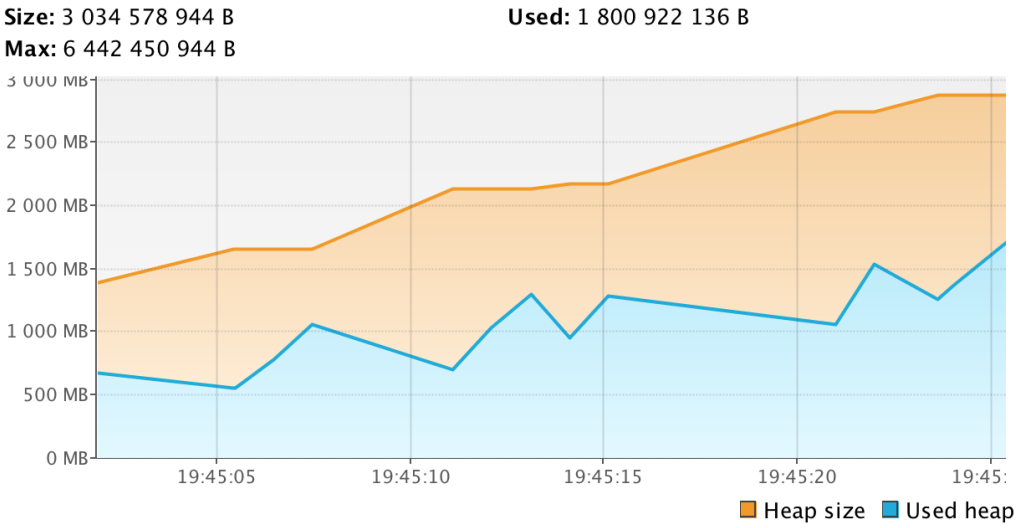
(b) soustava loaderů Dochází zde k vytváření externích tříd, jejichž existence může mít negativní dopad na míru optimalizace. Efekt optimalizace by zde měl být proto nižší, než v předchozím případě. Loader je znázorněn na obrázku 4.16.



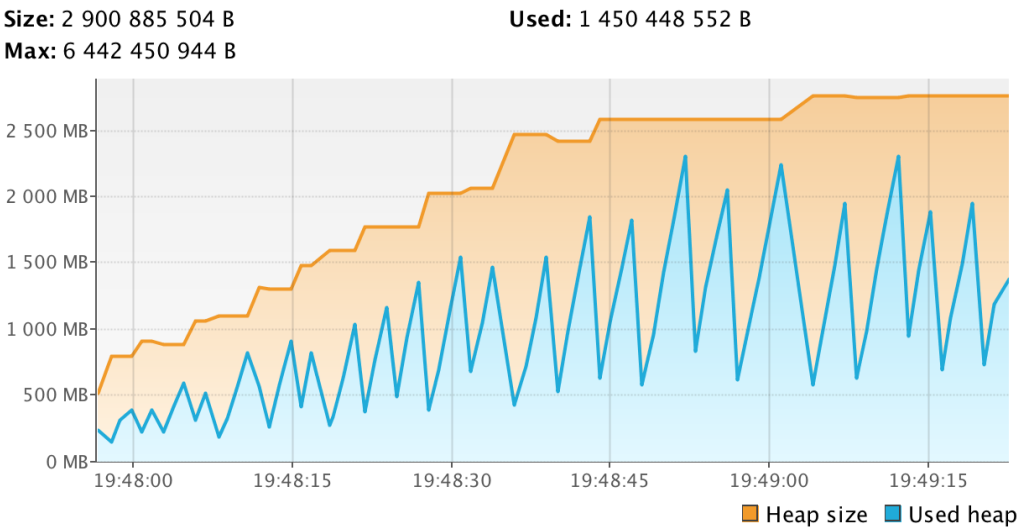
Obrázek 4.16: Soustava loaderů s interními a externími třídami

Předmětem experimentů byly open-source Java aplikace, které jsou k dispozici na stránkách `qualitascorpus.com` [17]. Těmito aplikacemi byly **WCT** (**W**eb **C**omponent **T**ester) ve verzi **1.5.2** [18] a **JasperReports** ve verzi **3.7.3** [19]

Analýza WCT



Obrázek 4.17: Využití paměti - původní JaCC, scénář (a)

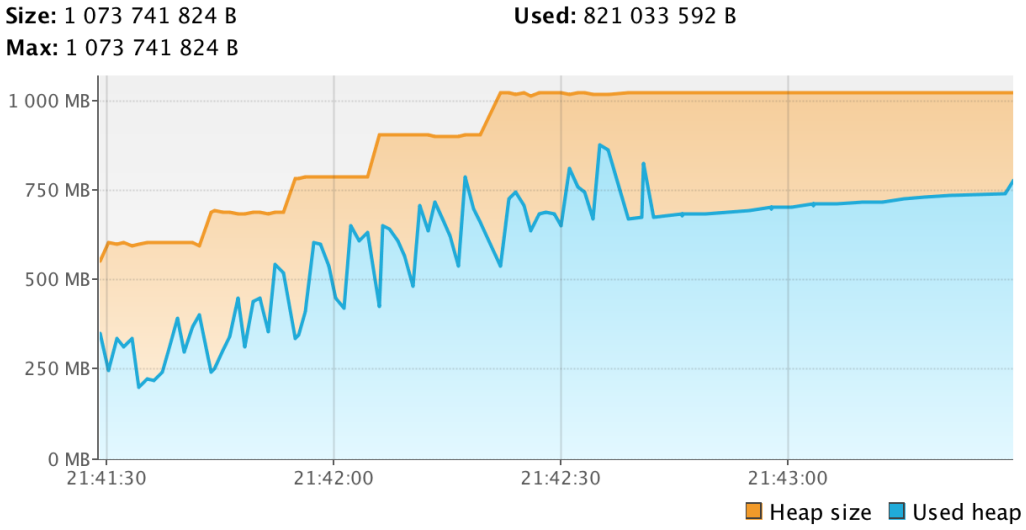


Obrázek 4.18: Využití paměti - optimalizovaný JaCC, scénář (a)

Jak je patrné z obrázků 4.17 a 4.18, charakter využití paměti původního a optimalizovaného řešení nástroje JaCC se značně odlišuje. Na prvním obrázku se nachází graf původního řešení. Graf má lineárně rostoucí tendenci s občasnými mírnými poklesy. Tato tendence je způsobena tím, že se objekty hromadí v paměti a většina z nich tam zůstává do doby, než aplikace doběhne. Ke garbage kolekci dochází pouze okrajově.

Graf na druhém obrázku má z hlediska stability naprosto opačný charakter - výrazně osciluje. Tato oscilace je způsobena aktivní garbage kolekcí, která souvisí s použitím cache, a bylo ji možné také sledovat pomocí profilačního nástroje JVisualVM. Používané objekty v cache se neustále obměňují a vyřazené objekty z cache se musí neustále vyklízet z paměti. Protože program data zpracovává rychleji, než je stíhá z cache vyklízet, je tento charakter oscilace velice výrazný. Za účelem dokázání této hypotézy bylo provedeno dodatečné měření s nastavenou maximální velikostí heap 1 GB (`-Xmx1g`) - přibližně k této spodní hodnotě graf 4.18 osciluje.

Dodatečné měření hypotézu potvrdilo. Aplikace i přes omezenou maximální velikost heap doběhla. Protože ovšem docházelo k velice k časté garbage kolekci, která běh aplikace zpomaluje, běžela aplikace dvakrát déle než dříve. Pokud by aplikace paměti potřebovala více a příčinou oscilace byla skutečná aktuální potřeba paměti, aplikace by selhala z důvodu nedostatku paměti. Jak zachycuje obrázek 4.19, oscilace už v tomto případě byla minimální.

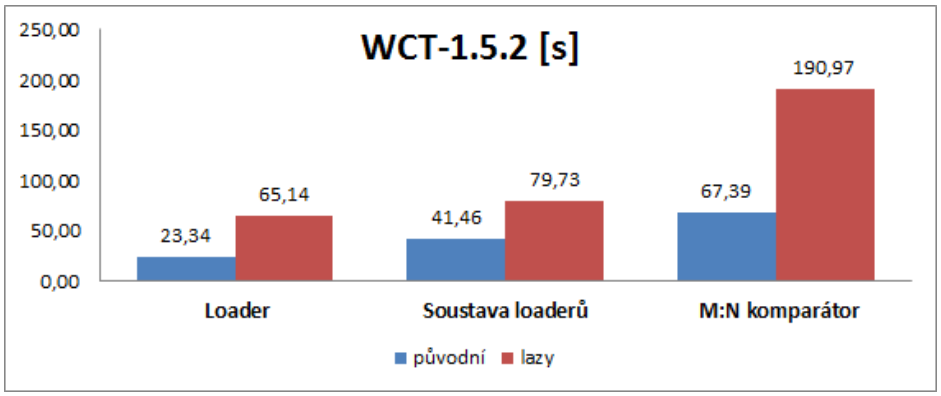


Obrázek 4.19: Využití paměti - původní JaCC, scénář (a), -Xmx1g

V následující části se nacházejí souhrnné výsledky experimentů v podobě tabulek s naměřenými hodnotami a sloupcových grafů srovnávajících původní a optimalizované řešení. V tabulkách se objevuje symbolické značení. Symboly **(a)**, **(b)** a **(c)** zastupují jednotlivé testované scénáře. Písmena **P** a **N** slouží pro rozlišení původní a nové verze JaCC.

	(a)		(b)		(c)	
Měření	**P**	**N**	**P**	**N**	**P**	**N**
1	27,54	64,77	41,47	87,80	64,77	185,37
2	26,73	65,79	42,84	78,82	64,81	186,51
3	22,58	66,44	43,83	78,75	64,89	193,16
4	21,88	62,41	38,88	76,77	69,78	181,59
5	22,22	62,64	46,66	77,08	63,84	183,96
6	22,03	64,43	35,68	83,27	69,92	182,52
7	25,01	65,09	43,23	75,49	68,18	226,34
8	21,33	62,41	43,87	76,64	73,99	185,58
9	22,41	72,85	42,63	77,56	63,75	184,51
10	21,71	64,60	35,46	85,09	70,00	200,18
průměr	**23,34**	**65,14**	**41,46**	**79,73**	**67,39**	**190,97**
odchylka	**2,13**	**2,89**	**3,48**	**3,95**	**3,30**	**12,93**

Obrázek 4.20: Doby zpracování aplikace WCT



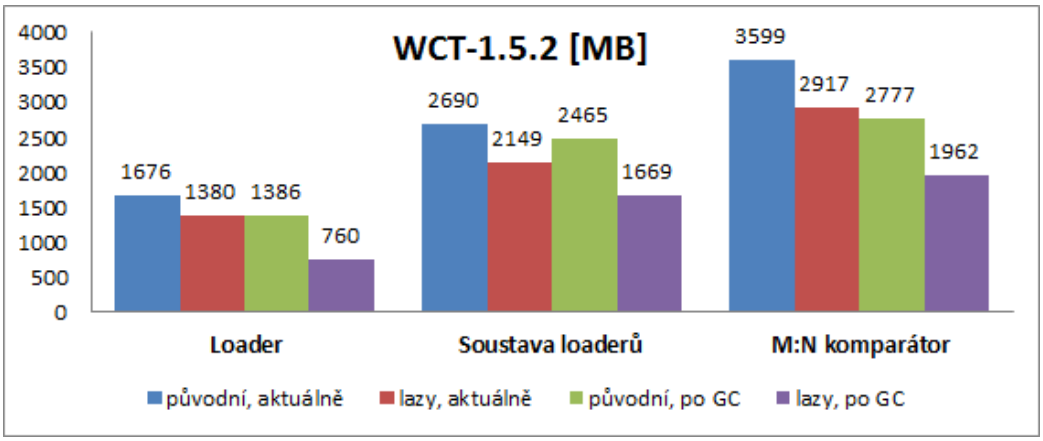
Obrázek 4.21: Doby zpracování aplikace WCT

Jak je vidět na obrázku 4.21 výše, nové řešení JaCC představuje přibližně 2 až 3 násobný nárůst průměrné doby zpracování v případě analýzy aplikace WCT. Toto zpomalení souvisí s použitím pevného disku co by dodatečného úložiště pro velké množství instancí datového typu `JClass`.

Níže se nachází tabulka 4.22, která uchovává jednotlivé naměřené hodnoty využití paměti. Protože byly naměřené hodnoty po garbage kolekci téměř konstantní, je tento údaj zanesen v tabulce na samostatné řádce.

	(a)		(b)		(c)	
Měření	**P**	**N**	**P**	**N**	**P**	**N**
1	1771	2013	2815	2355	3584	2788
2	1751	1439	2515	1808	3652	2711
3	1717	819	2481	2151	3340	3163
4	1585	1466	2481	1766	3730	3091
5	1691	1804	2682	2069	3623	2818
6	1590	1262	2984	1848	3773	3119
7	1759	1985	2665	2154	3423	2855
8	1684	884	2495	2372	3611	3042
9	1556	1273	2780	2335	3803	2585
10	1660	856	3005	2629	3451	2999
průměr	**1676**	**1380**	**2690**	**2149**	**3599**	**2917**
po GC	**1386**	**760**	**2465**	**1669**	**2777**	**1962**
odchylka	**73**	**426**	**191**	**268**	**146**	**184**

Obrázek 4.22: Využití paměti při zpracování aplikace WCT



Obrázek 4.23: Využití paměti při zpracování aplikace WCT

Na základě vypočtených odchylek měření z tabulky 4.22 na předchozí stránce lze usoudit, že k nejvýraznější oscilaci dochází při analýze WCT v prvním scénáři **(a)**. Jak ukazuje graf 4.23 výše, důvodem této výrazné oscilace je s nejvyšší pravděpodobností vysoký poměr mezi dále nepotřebnými a potřebnými objekty v paměti. Úspora paměti před a po garbage kolekci činí na referenčním místě v kódu programu **45%** (1386:760). Úspora paměti před a po garbage kolekci ve scénáři (b) činí **32%** (2465:1669) a ve scénáři (c) **29%** (2777:1962). Jak již ukázalo jedno z předešlých měření, minimalizace využité paměti na tyto naměřené hodnoty pomocí parametru `-Xmx` je realizovatelná, ale má za následek prodloužení doby běhu z důvodu časté činnosti garbage kolektoru.

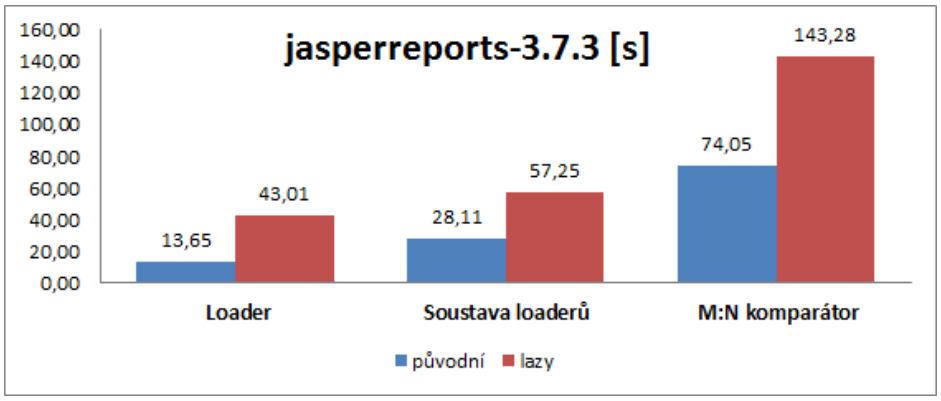
Pokud bychom se zaměřili na úsporu paměti v případě aktuálně naměřených hodnot využití paměti, má stále optimalizované řešení nižší paměťové požadavky, ale už ne v takové míře. V scénáři (a) došlo k **17**% (1676:1380) úspoře paměti, ve scénáři (b) k **20**% (2690:2149) a ve scénáři (c) k **19**% (3599:2917). Posuzovat paměťové požadavky nástroje na základně těchto hodnot ovšem výrazně znevýhodňuje optimalizované řešení, které se pojí s velkým množstvím nepotřebných dat v paměti.

Analýza JasperReports

	(a)		(b)		(c)	
Měření	**P**	**N**	**P**	**N**	**P**	**N**
1	15,00	47,10	28,19	59,36	77,26	144,18
2	14,68	44,82	25,80	57,98	72,11	140,42
3	12,73	37,89	29,14	59,76	76,59	138,89
4	12,25	42,97	26,02	56,13	77,83	148,70
5	12,97	42,88	29,05	52,48	72,68	159,14
6	13,43	43,96	26,54	58,39	76,94	139,75
7	11,29	40,25	25,87	57,34	73,82	141,05
8	16,32	43,57	29,45	54,89	71,15	135,64
9	15,27	44,02	30,98	60,12	69,85	144,58
10	12,57	42,63	30,10	56,09	72,26	140,47
průměr	**13,65**	**43,01**	**28,11**	**57,25**	**74,05**	**143,28**
odchylka	**1,51**	**2,37**	**1,82**	**2,27**	**2,73**	**6,28**

Obrázek 4.24: Doby zpracování aplikace JasperReports

Jak lze vyčíst z hodnot tabulky 4.24 výše, i v případě analýzy aplikace JasperReports nástrojem JaCC došlo v optimalizovaném řešení k nárůstu průměrné délky doby běhu přibližně na 2 až 3 násobek. Na obrázku 4.25 jsou opět původní a nové délky běhu graficky porovnány.



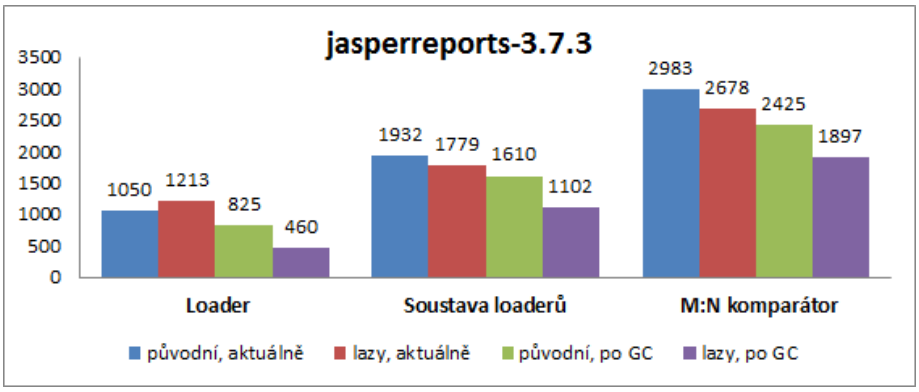
Obrázek 4.25: Doby zpracování aplikace JasperReports

	(a)		(b)		(c)	
Měření	**P**	**N**	**P**	**N**	**P**	**N**
1	1088	1928	1885	1369	2785	2622
2	1006	1113	1992	2103	3126	2803
3	1036	641	2039	2001	3052	2693
4	1007	1002	2001	2179	3017	2536
5	1144	1907	1878	1231	3097	2759
6	1086	1730	1976	1747	3258	2801
7	1056	1190	2062	1389	2790	2348
8	1051	1040	1806	2135	2989	3083
9	1034	609	1717	1620	2890	2183
10	996	969	1959	2012	2830	2950
průměr	**1050**	**1213**	**1932**	**1779**	**2983**	**2678**
po GC	**825**	**460**	**2465**	**1102**	**1610**	**1897**
odchylka	**43**	**458**	**103**	**338**	**150**	**255**

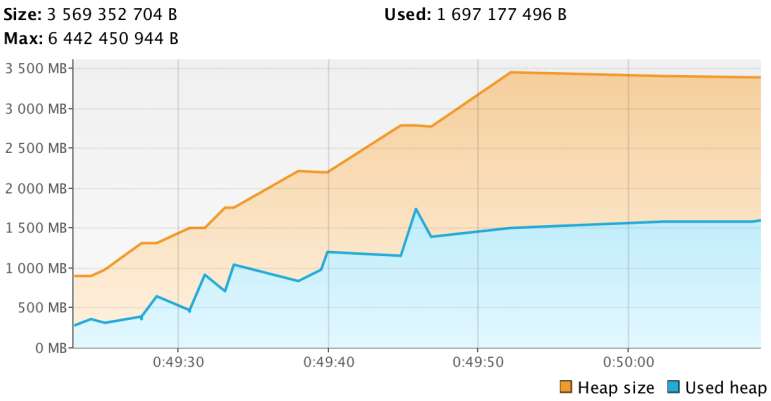
Obrázek 4.26: Využití paměti při zpracování aplikace JasperReports

Naměřené hodnoty využití paměti se svým charakterem opět velice podobaly hodnotám naměřeným při analýze aplikace WCT. V případě hodnot využití paměti naměřených po garbage kolekci byla úspora paměti v scénáři (a) **44%** (825:460), ve scénáři (b) **32%** (1610:1102) a ve scénáři (c) **22%** (2425:1897).

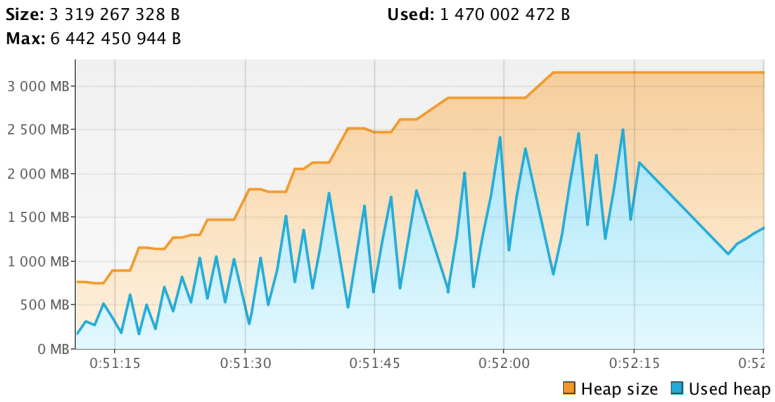
V případě měření aktuálně využité paměti došlo ve scénáři (a) dokonce k mírnému **15%** (1050:1213) nárůstu využití paměti v případě optimalizovaného řešení. Ovšem s ohledem na velikost odchylky nelze tuto hodnotu pokládat za příliš směrodatnou. Úspora aktuálně využité paměti ve scénáři (b) činila **8%** (1932:1779) a ve scénáři (c) **10%** (2983:2678). Na obrázku 4.27 na další straně je opět grafické srovnání původního a optimalizovaného řešení z hlediska využití paměti při analýze aplikace JasperReports. Dále jsou přiloženy také grafy využití paměti pro scénář (b) původního (obrázek 4.28) a optimalizovaného (obrázek 4.29) řešení.



Obrázek 4.27: Využití paměti při zpracování aplikace JasperReports

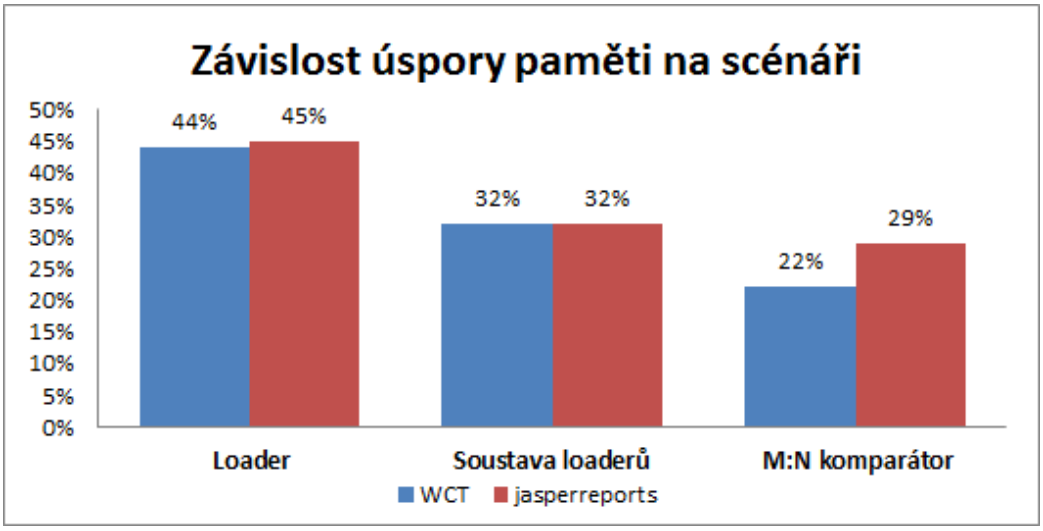


Obrázek 4.28: Využití paměti - původní JaCC, scénář (b)



Obrázek 4.29: Využití paměti - lazy JaCC, scénář (b)

Shrnutí výsledků



Obrázek 4.30: Závislost míry úspory paměti po GC na scénáři

Odchylky naměřených hodnot aktuálního využití paměti byly velmi vysoké. Zároveň skutečné využití paměti představuje hlavně celková velikost potřebných objektů, které přežijí garbage kolekci. Shrnuté výsledky jsou proto založeny na naměřených hodnotách využití paměti po provedené garbage kolekci.

Experimenty potvrdily původní domněnku a ukázaly, že podíl datových struktur, které nejsou součástí cachování, má skutečně dopad na míru optimalizace nástroje JaCC. Pokud se v paměti vytváří pouze struktury interních tříd, je **úspora paměti** skoro **poloviční** (44% WCT, 45% JasperReports). Pokud se v paměti vytváří mimo interní třídy také třídy externí, **úspora paměti** je přibližně **třetinová** (32% WCT i JasperReports) a pokud jsou vedle tříd vytvářeny a uchovávány ještě výsledky porovnání (M:N komparátor), je **úspora paměti** přibližně **čtvrtinová** (22% WCT, 29% JasperReports). Na obrázku 4.30 výše je graficky znázorněna popsaná závislost mezi mírou úspory paměti a testovanou úrovní funkčnosti M:N komparátoru. Dále bylo připraveno speciální měření, které potvrdilo, že k oscilaci využití paměti optimalizovaného řešení nedochází vlivem skutečně potřebné paměti, ale především z důvodu opožděného vyklízení nepotřebných objektů z paměti. Omezení délky doby běhu na naměřené hodnoty využití paměti po garbage kolekci je tedy teoreticky možné.

5 Závěr

Primárním cílem práce bylo optimalizovat nástroj JaCC z hlediska jeho paměťové náročnosti. Dosažení tohoto cíle se sestávalo z několika dílčích úkolů, přičemž tím hlavním byla identifikace zdroje vysokých paměťových požadavků.

Nástroj při svém zpracování vytvářel velké množství datových struktur, které po celou dobu svého běhu uchovával v jediné úložné vrstvě, operační paměti. Tyto datové struktury tvořily převážně reprezentace analyzovaných tříd a výsledky jejich porovnání. Iterativní přístup zpracování, který by z nástroje odstraňoval jeho stavovost, se ukázal jako nereálný. Proto byl vytvořen návrh způsobu optimalizace, který se zakládal na integraci knihovny Ehcache a na zavedení dodatečné úložné vrstvy v podobě paměti pevného disku. V rámci této optimalizace byla provedena úprava datového modelu a částečná restrukturalizace klíčové výpočetní části nástroje. Následovala fáze testování, která ověřila správnou funkčnost upravené verze nástroje.

V závěrečné části práce byly prováděny experimenty, jejichž cílem bylo porovnat původní a upravené řešení z hlediska paměťových požadavků a výkonnosti. V rámci těchto experimentů byla připravena sada 3 typických scénářů případů užití. Vstupem experimentů byly aplikace *WCT* a *JasperReports*(*JR*). Výstupem experimentů byly naměřené doby běhu a množství využité paměti.

Naměřená hodnota uspořené paměti v prvním scénáři činila 44% (WCT) a 45% (JR), ve druhém scénáři u obou aplikací 32% a ve třetím scénáři 22% (WCT) a 29% (JR). Naměřené výsledky potvrdily domněnku, že vyšší podíl datových struktur, které nebyly předmětem optimalizace, má za následek nižší míru úspory paměti. Přesto se povedlo uspořit přibližně čtvrtinu paměti i ve třetím scénáři, kde je tento podíl nejvyšší. V rámci experimentů byla také měřena průměrná doba běhu. Jak se ukázalo, optimalizace paměťových požadavků měla za následek prodloužení průměrné doby běhu na 2 až 3 násobek. Příčinou je použití pevného disku co by dodatečné úložné vrstvy s vyšší přístupovou dobou. Tato vyšší přístupová doba se v konečném důsledku projevuje i na delší době běhu programu.

Výhodou optimalizované verze nástroje JaCC je jeho přizpůsobitelnost. Protože lze libovolně konfigurovat využití jednotlivých úložných vrstev, je možné hledat kompromis mezi mírou ušetřené paměti a délkou doby běhu.

Možným rozšířením nástroje JaCC je dále zavedení podpory pro trvalou perzistenci analyzovaných dat. Při opakovaném načítání stejných komponent by tak docházelo k výrazné úspoře času, protože by nástroj JaCC komponentu rozpoznal a provedl načtení odpovídajícího již inicializovaného datového modelu z disku. Knihovna Ehcache podporuje tuto funkcionalitu pouze v placené licenci.

Literatura

[1] Sun Java System Application Server Enterprise Edition 8.2 Performance Tuning Guide. *Oracle*. [online]. @2010 [cit. 2016-05-09]. Dostupné z: `https://docs.oracle.com/cd/E19900-01/819-4742/abeik/index.html`

[2] System (Java Platform SE 7). *Oracle*. [online]. @2016 [cit. 2016-05-09]. Dostupné z: `https://docs.oracle.com/javase/7/docs/api/java/lang/System.html`

[3] Java Garbage Collection. *Oracle*. [online]. [cit. 2016-05-09]. Dostupné z: `http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html`

[4] Vladimir Roubtsov. Sizeof for Java. *JavaWorld*. [online]. 26.12.2003 [cit. 2016-05-11]. Dostupné z: `http://www.javaworld.com/article/2077408/core-java/sizeof-for-java.html`

[5] 9 tools to help you with Java Performance Tuning. *IDRSolutions*. [online]. 12.6.2014 [cit. 2016-05-09]. Dostupné z: `https://blog.idrsolutions.com/2014/06/java-performance-tuning-tools/`

[6] Open Source Cache Solutions in Java. *Java-Source.net*. [online]. [cit. 2016-05-09]. Dostupné z: `http://java-source.net/open-source/cache-solutions`

[7] JaCC – Java Class Comparator. *Assembla*. [online]. 6.6.2014 [cit. 2016-05-09]. Dostupné z: `https://www.assembla.com/spaces/jacc/wiki`

[8] Jiří Sochor. Údržba softwaru. *ÚVTMU zpravodaj*. [online]. 14.11.2011 [cit. 2016-05-09]. Dostupné z: `http://webserver.ics.muni.cz/zpravodaj/articles/61.html`

[9] Jakub Rinkes. *Migrace nástroje pro statickou analýzu Java byte-code do cloudu*. Plzeň, 2015. Diplomová práce. Západočeská univerzita v Plzni. Fakulta aplikovaných věd. Vedoucí práce Ing. Kamil Ježek, Ph.D.

[10] EsotericSoftware/Kryo: Java serialization and cloning: fast, efficient, automatic. *GitHub*. [online]. [2016] [cit. 2016-05-09]. Dostupné z: `https://github.com/EsotericSoftware/kryo`

[11] Ruediger Moeller. Benchmark – fast serilization. *GitHub*. [online]. 16.3.2014 [cit. 2016-05-09]. Dostupné z: `https://github.com/RuedigerMoeller/fast-serialization/wiki/Benchmark`

[12] Welcome to Apache Maven. *Maven*. [online]. 7.5.2016 [cit. 2016-05-09]. Dostupné z: `https://maven.apache.org/`

[13] Home Page. *ASM*. [online]. 5.3.2016 [cit. 2016-05-09]. Dostupné z: `http://asm.ow2.org/`

[14] GAMMA, Erich. *Design patterns elements of reusable object-oriented software*. Reading: Addison-Wesley, c1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.

[15] GOETZ, Brian a Tim PEIERLS. *Java concurrency in practice*. Upper Saddle River, NJ: Addison-Wesley, c2006. ISBN 0-321-34960-1.

[16] HANDY, Jim. *The cache memory book*. 2nd ed. San Diego: Academic Press, c1998. ISBN 0123229804.

[17] Ewan Tempero. Qualitas Corpus. *Qualitas Corpus*. [online]. 11.9.2013 [cit. 2016-05-09]. Dostupné z: `http://qualitascorpus.com/`

[18] web-component-tester. *GitHub*. [online]. 26.2.2016 [cit. 2016-05-09]. Dostupné z: `https://github.com/Polymer/web-component-tester/blob/master/README.md`

[19] JasperReports Library. *Jaspersoft Community*. [online]. [2016] [cit. 2016-05-10]. Dostupné z: `http://community.jaspersoft.com/project/jasperreports-library`

[20] google/guice:Guice. *GitHub*. [online]. @2015 [cit. 2016-05-09]. Dostupné z: `https://github.com/google/guice`

A Seznam zkratek

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
FIFO	First In, First Out
JaCC	Java Class Comparator
JVM	Java Virtual Machine
LFU	Least Frequently Used
LRU	Least Recently Used
UTF-8	UCS Transformation Format
WCT	Web Component Tester