Key Word In Context srovnání architektonických stylů

Antonín Daněk
Fakulta elektrotechnická
České vysoké učení technické
Praha, Česká Republika
danekant@fel.cvut.cz

Jakub Stejskal
Fakulta elektrotechnická
České vysoké učení technické
Praha, Česká Republika
stejsj11@fel.cvut.cz

***Abstract*—V této studii porovnáváme čtyři oblíbené architektonické styly (hlavní program a podprogramy, Abstraktní datové typy, Události a Roury & Filtry) na příkladu jejich použití pro implementaci Key Word in Context. Zjišťujeme, v čem mají jednotlivé přístupy k návrhu software silné a v čem naopak slabé stránky.**

***Keywords*—Klíčová slova: KWIC, Sdílená paměť, Abstraktní datové typy, Události, Roury & Filtry**

I. Úvod

V dobách před rozšířením fulltextového vyhledávání se KWIC indexy používaly pro snazší orientaci a vyhledávání v textu. Typickým případem využití jsou indexy dříve připojované k Unixových MAN stránkám. KWIC index vzniká vytvořením cyklických permutací jednotlivých úseků (zpravidla řádků) textu a jejich následné seřazení dle abecedního pořádku. Přesto, že v praxi již není využíván, koncept Klíčových slov v kontextu se stal klasickým příkladem pro srovnávání softwarových architektonických vzorů.

A. Moduly KWIC

Aplikaci realizující KWIC je obecně možno rozlišit do pěti funkčních modulů:

Input - vstupní modul načítá ze zdroje (např. textového souboru) vstupní data, což je uspořádaná množina řádků, převádí je do formy potřebné pro zpracování dalšími moduly a dalšímu modulu tato data také předává či je případně ukládá do sdíleného úložiště.

Circular shifter - tento modul vytváří všechny možné cyklické permutace řádků získaných ze vstupního modulu. To může v závislosti na implementaci vyžadovat fyzické vytvoření jednotlivých permutovaných řádků, nebo také pouhé uložení indexů slov na řádcích.

Alphabetizer - získané permutace jsou pomocí tohoto modulu seřazeny dle abecedního pořádku. Seřazenost umožňuje rychejší vyhledávání nad těmito daty v následujícím modulu.

Search - vyhledávací modul nalezne v seřazeném seznamu permutací řádky začínající na zadané slovo a tyto jednotlivé výskyty předává spolu s okolními slovy a dalšími informacemi, jako je např. číslo řádky, na kterém se slovo nacházelo.

Output - tento modul vypisuje výsledná data na výstup - ať už do konzole či souboru.

Jednotlivé implementace zavádějí i některé další moduly, které slouží k řízení aplikace, sdílení dat a podobně.

Různé architektury a jejich implementace se liší v tom, jak jsou moduly reprezentovány (např. metodami či objekty), jakým způsobem spolu moduly komunikují či jak si předávají a sdílejí data. Každý z těchto rysů s sebou přináší možné výhody či nevýhody. Právě popisem a porovnáním těchto charakteristik se zabýváme v následujících kapitolách. Implementace porovnáváme z pěti různých hledisek: výpočetní výkon, modifikovatelnost algoritmů, modifikovatelnost reprezentace dat, rozšiřitelnost a znovupoužitelnost kódu.

B. Společný kód

Protože implementace vstupních a výstupních metod se napříč architekturami prakticky neliší, rozhodli jsme se zamezit duplikaci kódu vytvořením společného vstupně-výstupního rozhraní `IKwicIO`. Moduly Input a Output jednotlivých implementací jsou pak pouze obalovými třídami (resp. metodami) využívajícími tohoto rozhraní. Toto rozhraní má pro snadnější testování implementací dvě různé implementace: souborové `FileIO` a konzolové `ConsoleIO`.

Další sdílenou třídou je knihovna StringOperations, která poskytuje často používané operace s daty, jako je převod pole slov do textového řetězce a zpět.

II. Hlavní program a podprogramy se sdílenou pamětí

A. Stručný popis běhu programu

Tato architektura je z uvedených čtyř zřejmě nejjednodušší. Každý modul je realizován jednou metodou společné třídy. Datové úložiště je reprezentováno třídním atributem, konkrétně typu `ArrayList<String[]>`. Moduly mají do tohoto úložiště přímý přístup a sdílí tak veškeré informace o datech. Díky tomu není nutné vytvářet redundantní informace ani s uloženými daty jakkoli manipulovat - stačí je pouze doplňovat o informace nově získané.

Metody jsou postupně volány hlavním programem, práce každého modulu je proto započata až ve chvíli, kdy předchozí modul zpracoval všechna data.

Program je vykonán následovně:

- Input modul načítá data a ukládá je po řádcích do datového úložiště jako pole slov typu `String`.

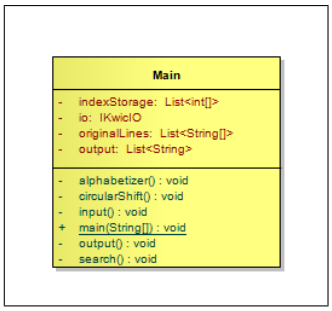

Figure 1. UML reprezentace objektů KWIC implementovaného pomocí Sdílené paměti

- CircularShifter vytváří všechny možné cyklické permutace reprezentované číslem řádku a pořadím prvního slova permutace. Tyto indexy jsou uloženy v seznamu, který je dalším třídním atributem.
- Alphabetizer převádí permutace do textových řetězců a řadí jejich indexy dle abecedy.
- Search provede vyhledání slova nad permutacemi za pomoci seřazených indexů metodou binárního půlení. Nalezené řádky, odešle do modulu Output.
- Modul Output vypisuje získaný výsledek.

B. Zhodnocení implementace pomocí sdílené paměti

Výkon, a to jak časový, tak i paměťový, je silnou stránkou této architektury. Díky sdílené paměti je zde minimální režie a nedochází k redundantnímu výpočtu a ukládání dat. Protože se řazení, na rozdíl od jiných implementací, vykonává na všech datech najednou, lze s výhodou využít efektivnějších řadících algoritmů.

Možnost **změn v algoritmu** je značně omezena požadavkem architektury na sekvenční zpracování a nelimitovaném přístupu k datům ze všech modulů. Například paralelní či postupné vyhodnocování zde téměř nepřichází v úvahu.

Také **změny v datové části** jsou při použití sdílené paměti přirozeně velice náročné, protože jakákoli modifikace se pravděpodobně promítne do všech modulů.

Rozšiřování funkcionality v rámci modulu je možné provádět poměrně jednoduše, protože jednotlivé moduly se prakticky neovlivňují a nekomunikují navzájem. Pouze postupně pracují nad stejnými daty. Tato výhoda ovšem mizí v okamžiku, kdy nemáme možnost modifikovat zdrojový kód programu. Moduly jsou nedělitelně spjaty s programem a pokud nelze z nějakého důvodu program dědit a modul překrýt, je aplikace prakticky nemodifikovatelná.

Znovupoužitelnost komponent této architektury je mizivá, neboť moduly jsou přímo vázány na datové úložiště a hlavní program.

III. Abstraktní datové typy

A. Stručný popis běhu programu

Architektura ADT se liší od předchozí tím, že datové úložiště již není přímo sdíleno mezi moduly. Místo toho každý modul poskytuje ostatním rozhraní pro přístup ke svým

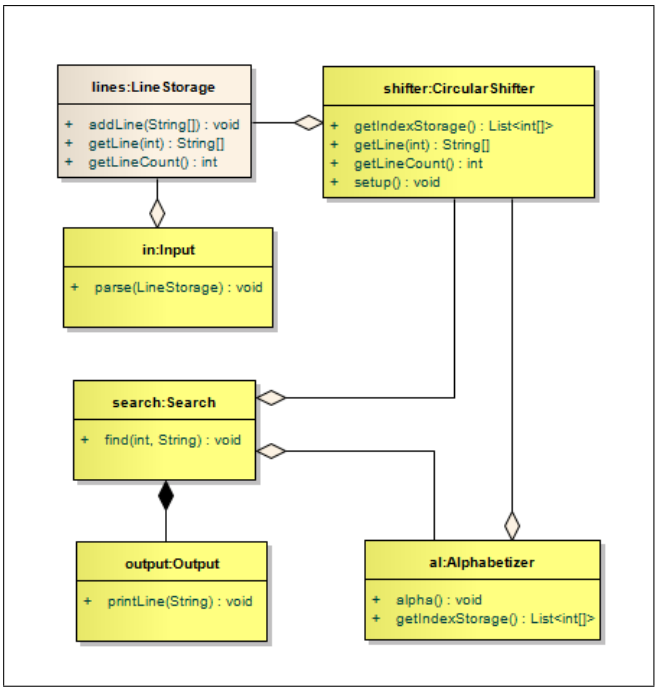

Figure 2. UML reprezentace objektů KWIC implementovaného pomocí ADT

datům. Toto opatření zásadním způsobem ovlivňuje vazby mezi komponentami.

Moduly jsou stejně jako u předchozí implementace volány sekvenčně po zpracování dat předchozími moduly.

Vykonání programu probíhá takto:

- Input modul naplní úložiště dat.
- CircularShifter vytváří indexy permutací podobně jako je tomu u sdílené paměti s tím rozdílem, že k datům je přistupováno přes rozhraní datového úložiště
- Alphabetizer přistupuje k řádkům i jejich indexům přes rozhraní CircularShifteru a abecedně je řadí.
- Search provede vyhledání slova nad permutacemi za pomoci seřazených indexů metodou binárního půlení. Nalezené řádky, odešle do modulu Output.
- Output vypíše výsledky na výstup.

B. Zhodnocení implementace pomocí ADT

Výkon architektury ADT se v zásadě příliš neliší od Sdílené paměti, ačkoliv poněkud přibývá režie potřebné k získání dat.

Změny v algoritmu jednotlivých modulů jsou snadno realizovatelné díky jejich zapouzdření.

Zapouzdření datové reprezentace značně ulehčuje i **změny v datové části**. Dokud je zachováno rozhraní daného modulu, jeho datovou strukturu lze měnit bez vlivu na ostatní komponenty.

Tato architektura naopak není příliš vhodná k **rozšiřování**. Přidání nových komponent by vedlo k nižšímu výkonu a rozšiřování těch stávajících může ohrozit jejich jednoduchost, což je zpravidla nežádoucí.

Vysoká **znovupoužitelnost** komponent je dána tím, že při jejich znovunasazení musíme respektovat pouze jejich rozhraní a rozhraní komponent, které daný modul využívá.

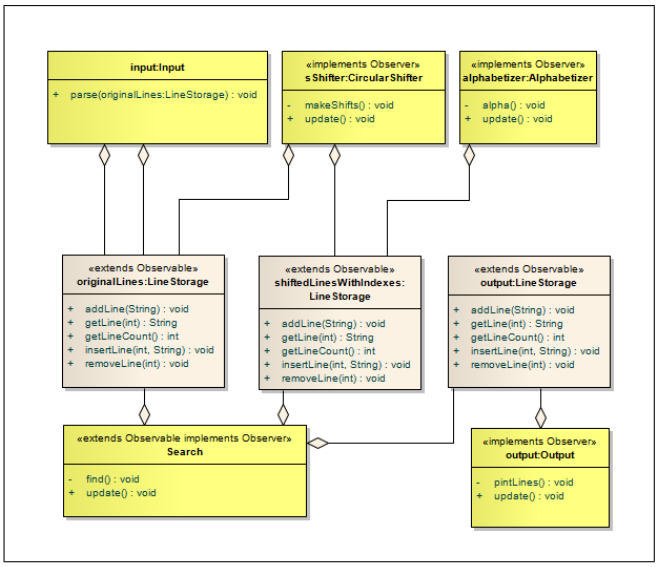

Figure 3. UML reprezentace objektů KWIC implementovaného pomocí událostí

IV. Události

Implementaci KWIC s použitím událostí jsme realizovali návrhovým vzorem Observer.

Jak je vidět na diagramu 3, všechny moduly kromě modulu Input implementují rozhraní Observer, které vyžaduje implementaci metody `update()`. Skrze tuto metodu mohou být moduly informovány o změnách na objektech, které pozorují.

Pozorovanými objekty jsou především instance třídy LineStorage, které slouží k ukládání dat a program obsahuje celkem tři. Jedna instance pro původní data, druhá pro permutace těchto dat s indexy a třetí pro výstupní nalezené řádky.

A. Stručný popis běhu programu

Práce programu začíná v modulu Input, kterému je předána instance LineStorage a je zde plněna řádky ze vstupu. LineStorage je navržen tak, že po přijetí každé řádky informuje své observery o změně. Toto oznámění vyvolá následující sérii událostí:

- Instance `originalLines` informuje CircularShifter o přijetí nového řádku.
- CircularShifter přečte poslední řádku z originalLines a vytvoří ze slov tohoto řádku všechny možné permutace řádků. Tyto permutace přidává do další instance LineStorage - `shiftedLinesWithIndexes`. Aby bylo možné z permutací rekonstruovat originální text, přidává na konec index původního řádku a pořadí slova.
- Instance `shiftedLinesWithIndexes` informuje Alphabetizer o přijetí nového řádku.
- Alphabetizer řádek ze seznamu nejprve odstraní a následně jej vloží na správné místo, aby byla zachována seřazenost všech řádek dle abecedy (jelikož běh programu začíná s prázdným seznamem, budou řádky seřazené po jediném průběhu insertion sortu).

Po přečtení všech řádek ze vstupu a zpracování událostí vyvolaných tímto čtením máme v instanci `shiftedLinesWithIndexes` seřazené všechny permutace řádků ze vstupu, včetně jejich indexů.

Následně je spuštěn běh modulu Search, který se pokusí metodou binárního půlení nalézt hledané slovo a následně dohledat zadaný počet slov, které hledané slovo předcházejí a následují.

Po dokončení práce Search modul informuje (opět pomocí návrhového vzoru Observer) poslední modul - Output, který všechny nalezené položky vypíše.

B. Zhodnocení implementace pomocí událostí

Výkon tohoto řešení je poměrně nízký (viz. tabulka II). Rychlost snižuje režie použitého návrhového vzoru, která vzniká při informování observerů, ale ani použitý řadící algoritmus není ideální. Vhledem ke zpracování po řádcích je řazení postupným vkládáním optimální, nejedná se však o optimální řešení obecně. Pokud bychom nejdříve uložili všechny permutace vstupu a až na závěr je pouze jednou seřadili např. quick sortem, dosáhli bychom výrazně lepších výsledků. Pak bychom ale nedodržovali daný architektonický styl.

Změny v algoritmu lze provádět u tohoto stylu díky rozdělení do jednotlivých tříd poměrně snadno. Hovoříme však o lokálních změnách jednotlivých modulů, pokud bychom chtěli provést nějakou zásadnější úpravu, jako např. již zmíněnou změnu způsobu řazení, museli bychom poměrně radikálně zasáhnout do celého systém, protože jsme omezeni zpracováním po řádcích.

Změny v datové části nelze provést bez zásahu do všech modulů. Při pohledu na diagram 3 je snadno vidět, že jsou mezi jednotlivými moduly datové části sdíleny, proto bychom museli na jejich změnu reagovat všude, kde jsou používány.

Rozšiřitelnost je u událostmi řízené aplikace díky takřka nulovému provázání mezi jednotlivými komponentami velmi jednoduchá. Pokud bychom chtěli do aplikace přidat nový modul - např. logger - stačilo by nám přidat současným Observable objektům nového posluchače a současné moduly by zůstaly zcela bez zásahu.

Tento architektonický styl u svých komponent nabízí i určitou **znovupoužitelnost**, moduly jsou však svázány s rozhraním Observer a pokud jsou navrženy správně dle stejnojmenného návrhového vzoru, stanou se velmi těžko použitelnými jiným způsobem, než prostřednictvím notifikací od sledovaných objektů. Pokud bychom tak chtěli např. použít Alphabetizer v jiné části systému k seřazení řetězců, museli bychom nejdříve vytvořit instanci LineStorage, této instanci přidat mezi observery instanci Alphabetizeru, vytvořit další instanci LineStorage pro výstup a až tehdy načíst vstupní data.

V. Roury & Filtry

Jak je vidět na obrázku 4, všechny moduly tohoto řešení dědí od abstraktního předka Filter, který definuje konstruktory pro připojení potřebných rour. Roury používají knihovní `PipedReader` a `PipedWriter`.

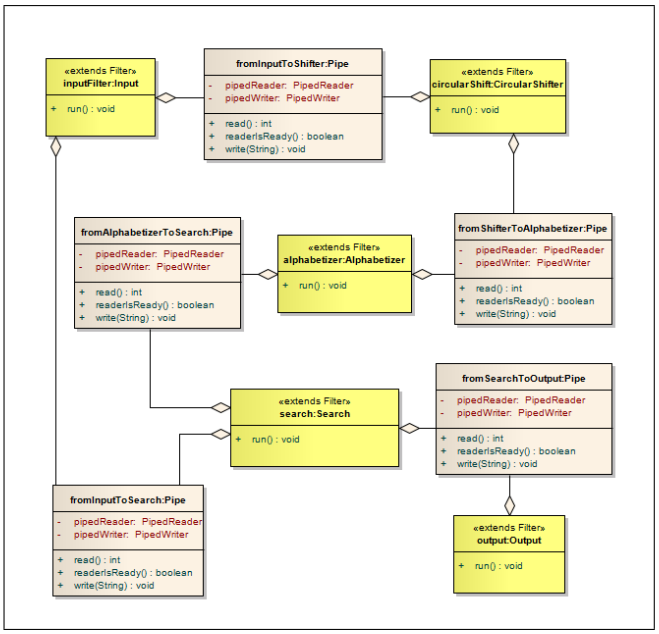

Figure 4. UML reprezentace objektů KWIC implementovaného pomocí rour & filtrů

A. Stručný popis běhu programu

Tato verze programu je realizována tak, že jednotlivé filtry mohou díky implementaci rozhraní `Runnable` běžet v samostatných vláknech. Nejdříve jsou vytvořeny všechny filtry i roury a následně spuštěny vlákna, ve kterých filtry běží.

Každý filtr čeká, dokud nedostane na vstupu nový řádek a v pravidelných intervalech předává řízení ostatním filtrům, aby nedošlo k zaseknutí u jednoho filtru či zahlcení příslušné roury (v době kdy už data na vstupu jsou).

Po spuštění všech filtrů probíhá následující postup práce:

- Input filtr čte po řádcích data ze vstupu a předává je skrze dvě roury jak modulu CircularShifter, tak modulu Search. Po určité době se vzdá řízení (platí pro všechny filtry a nebude znovu zmiňováno).
- CircularShifter parsuje data ze vstupní roury. Po obdržení celé řádky vytvoří všechny možné permutace tohoto řádku a zapíše je do výstupní roury. Aby bylo možné z permutací rekonstruovat originální text, přidává na konec index původního řádku a pořadí slova.
- Alphabetizer parsuje data ze vstupní roury a po obdržení celé řádky pomocí insertion sortu zařadí řádek do lokální cache tak, aby byly řádky seřazené podle abecedy.
- Po přečtení všech řádek ze vstupu a seřazení jejich permutací dojde k odeslání těchto řádek modulu Search (originální řádky má v této době již načtené). Search provede vyhledání slova nad seřazenými permutacemi a pomocí indexů na konci řádek zrekonstruuje z originálních řádek nalezené řádky, které odešle do modulu Output.
- Filtr Output vypíše výsledky na výstup.

Kritérium / Architektura	Sdílená paměť	ADT	Události	Roury& Filtry
Výkon	9	8	3	2
Změny v algoritmu	2	5	8	9
Změny v datové části	1	7	4	3
Rozšiřitelnost	5	3	9	8
Znovupoužitelnost	1	5	6	8

Table I
KWIC - HODNOCENÍ DLE KRITÉRIÍ A ARCHITEKTUR

B. Zhodnocení implementace pomocí rour & filtrů

Výkon tohoto řešení, je o něco horší, než výkon verze s událostmi. Kromě stejného problému s pomalým řadícím algoritmem je zde velká režie při předávání dat mezi jednotlivými filtry, kdy dochází k opětovnému kódování a zpětnému parsování řádků.

Změny v algoritmu lze provádět na úrovni jednotlivých filtrů bez jakéhokoliv ovlivnění filtrů okolních. Kromě toho (na rozdíl od událostmi řízené aplikace) je zde mnohem přístupnější možnost zrušit řádkové zpracování a např. řadit řádky rychlejším algoritmem až po přijetí všech permutací vstupu. Toho by v případě tohoto architektonického stylu bylo možné docílit bez zásahu do jiných filtrů (ačkoliv bychom tímto zásahem přišli o paralelní zpracování).

Vzhledem k tomu, že definovaný formát dat, jaké filtry přijímají a odesílají je jediné rozhraní, které u tohoto stylu musíme vyžadovat, **změny v datové části** nelze provést bez zásahu do příslušných filtrů. Pokud bychom se např. rozhodli oddělovat řádky středníky místo znaky nového řádku, museli bychom toto změnit u všech filtrů.

Díky tomu, že je systém dekomponován mezi jednotlivé filtry, které definují pouze formát dat, jaký z nich vystupuje a jaký požadují, je možné software používající tento architektonický styl snadno **rozšiřovat**. Takové rozšíření pak spočívá pouze ve vytvoření nového filtru a jeho připojení na příslušné roury. Zbytek systému zůstane neovlivněn.

Znovupoužitelnost je u rour & filtrů velmi dobrá. Díky minimálním požadavkům na rozhraní je možné filtr použít kdekoliv jinde, pouze je třeba mu zasílat data ve správném formátu.

VI. ZÁVĚR

O žádném z použitých stylů nemůžeme prohlásit, že by byl pro KWIC či jiný projekt ideální, dokud neznáme kompletní zadání a i tehdy budeme pravděpodobně muset přistoupit na kompromis mezi některými požadovanými vlastnostmi.

A. Výkon a paměťová náročnost

Jak je vidět v tabulce II a grafu 5, nejvýkonějším řešením je styl **Sdílená paměť**. Na druhém místě je Abstraktní datový typ, který je oproti prvnímu místu asi 3* pomalejší, ale v obou případech dochází se zvětšováním vstupních dat pouze k lineárnímu růstu času (na větším vzorku dat bychom

architektonický styl / čas	1000 řádek	3000 řádek	9000 řádek
Sdílená paměť	361ms	1.306ms	3.324ms
Abstraktní datové typy	1.143ms	3.395ms	10.800ms
Události	1.574ms	14.450ms	157.358ms
Roury & Filtry	6.463ms	20.666ms	192.731ms

Table II
KWIC - ČASOVÁ NÁROČNOST JEDNOTLIVÝCH ŘEŠENÍ

architektonický styl / datový objem	1000 řádek	3000 řádek	9000 řádek
Sdílená paměť	2.963kB	2.493kB	5.704kB
Abstraktní datové typy	4.288kB	5.924kB	9.548kB
Události	1.903kB	5.492kB	19.543kB
Roury & Filtry	3.404kB	10.305kB	31.575kB

Table III
KWIC - PAMĚŤOVÁ NÁROČNOST JEDNOTLIVÝCH ŘEŠENÍ

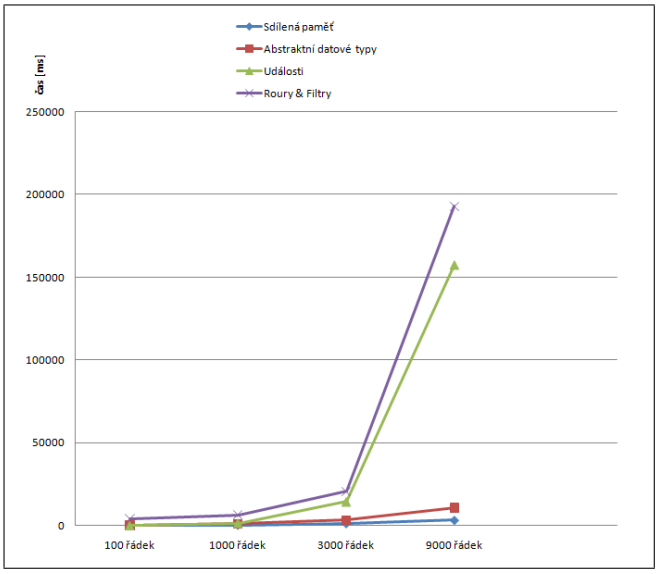

Figure 5. KWIC - časová náročnost jednotlivých řešení - graf

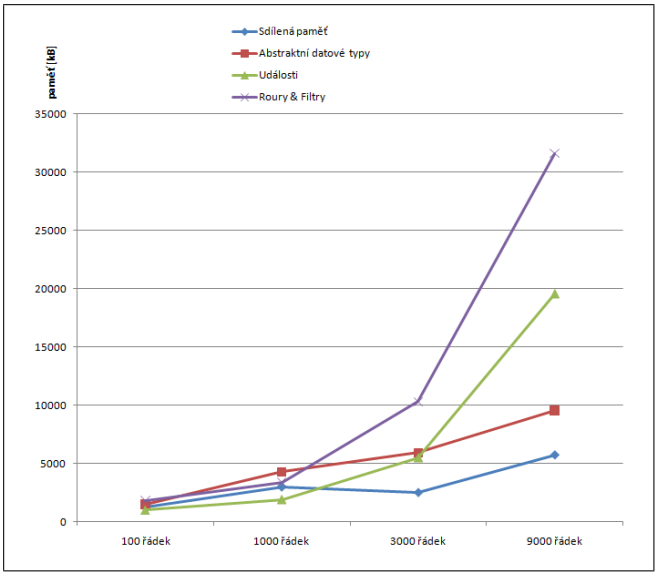

Figure 6. KWIC - paměťová náročnost jednotlivých řešení - graf

pravděpodobně zjistili, že jde o růst n*long(n)). U událostí a filtrů časové nároky rostou kvadratickou rychlostí, což je způsobené použitým typem řadícího algoritmu.

To samé platí i pro paměťové nároky (viz. tabulka III a graf 6), ačkoliv u menších vstupů dochází k určitých výkyvům, které jsou pravděpodobně způsobeny optimalizací alokace paměti v JVM[1].

B. Změnitelnost algoritmu

Pokud bychom navrhovali software, u kterého se dá předem očekávat, že bude potřeba měnit způsob jakým pracuje, nejlepším řešením bude styl **Roury & Filtry** a srovnatelně dobré řešení budou také Události. Vyloženě nevhodným stylem je z důvodu minimálního zapouzdření a nutností zasahovat při změnách do celého programu Sdílená paměť.

C. Změnitelnost datové části

Ke změnám formátu dat s jakým aplikace pracuje je benevolentní pouze styl **Abstraktní datové typy** a to díky jejich dobrému zapouzdření. Velmi nevhodným je pak styl Roury & Filtry, kde definice formátu dat současně definuje i rozhraní filtrů a jeho změna by způsobila nutnost provést změny po celé aplikaci.

[1]Java Virtual Machine

D. Rozšiřovatelnost

Pokud bude náš software v budoucnu hojně rozšiřován, měli bychom zvolit **Události** nebo Roury & Filtry. Oba tyto styly umožňují přidání nového modulu bez nutnosti zásahu do ostatních, původních modulů. Naopak velmi nevhodným stylem v tomto ohledu je Abstraktní datový typ, kde jsou moduly silně provázané.

E. Znovupoužitelnost

Nejsnadněji znovupoužitelné jsou komponenty stylu **Roury & Filtry**, kde je pouze nutné dodržovat formát dat, jaké filtry požadují, žádné další omezení zde nejsou. Naopak nejhůře znovupoužitelné jsou ze zřejmých důvodů moduly stylu Sdílená paměť.

REFERENCES

[1] Architektura softwarových systémů, Ing. Tomáš Černý, 08.05.2011, http://icpc.felk.cvut.cz/web/slides/y36ass/Arch/1_styly.pdf
[2] An Introduction to Software Architecture, David Garlan, Mary Shaw, 08.05.2011, http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf