Vysoká Škola Ekonomická v Praze
Fakulta informatiky a statistiky
Vyšší odborná škola informačních služeb v Praze

Michal Vašák

Návrh a implementace logovací komponenty newsroomového systému Octopus

Bakalářská práce

2008

(zadavaci list)

Prohlašuji, že jsem diplomovou (absolventskou, bakalářskou) práci na téma Návrh a implementace logovací komponenty newsroomového systému Octopus zpracoval samostatně a použil pouze zdrojů, které cituji a uvádím v seznamu použité literatury.

V Praze dne ... *Podpis*

OBSAH

Abstrakt

Tato práce popisuje návrh a vytvoření logovací komponenty pro newsroomový systém Octopus 6. Struktura práce vychází z omezené verze metodiky Agile Unified Process. V první fázi jsou stanoveny požadavky na komponentu a prozkoumána již existující a dostupná řešení logování v jazyce Java. Na konci této části jsou zvoleny nástroje pro fáze další. Ve druhé etapě je poté vytvořen detailní návrh její struktury a formátu výsledných logů. Ve třetí fázi je komponenta naprogramována v jazyce a v této práci představena prostřednictvím zdrojových kódů a několika kapitol blíže popisujících základní části hotového kódu.

Abstract (EN)

This paper describes the concept and implementation of a logging package for newsroom systém Octopus 6. The paper is stuctured along the lines of a reduced version of the Agile Unified Process methodology. In the first stage, requirements for the logging package are set and available solutions for logging in Java-based applications are described. Tools for the following stages are selected. In the second stage the new logging packages architecture is described in detail, together with the structure of the logs outputed by the package. In the third stage the completed package is presented by the means of source code and chapters describing the inner workings of the more important classes in greater detail.

Úvod

Tato práce vznikla, kromě jiného, z potřeby společnosti Octopus Newsroom s.r.o. vytvořit pro jejich právě vyvíjenou aplikace Octopus 6 – Borneo efektivní systém logování chyb a událostí uvnitř této aplikace.

Octopus 6 neboli Borneo je nejnovější verzí zejména ve střední Evropě velmi populárního newsroomového systému Octopus. Nejprve tedy něco málo o tom, co to Octopus vlastně je. Octopus je takzvaný newsroomový systém, to znamená, že jde o komplexní informační systém užívaný v newsroomech, tedy v přípravě a vysílání zpravodajských relací v rádiových nebo televizních stanicích. Systém se skládá z jednoho nebo více aplikačních serveru, jednoho či více databázových serverů a množství klientů – pracovních stanic. Přes tyto klienty se poté do systému mohou přihlašovat uživatelé, ať už přímo v prostorách newsroomu nebo přes internet třeba z druhé strany světa. Systém je dále připojen přes protokoly jako MOS (Media Objects Server Communication Protocol) na další zařízení v newsroomu, jako jsou např. teleprompteryčí zařízení umožňující vkládat do obrazu ze studia titulky nebo videosekvence.

Hlavní funkcí systému je tvorba tzv. bodových scénářů, které se podobají scénáři pro zpravodajskou relaci tvořenemu jednotlivými vstupy. Příkladem informací obsažených v bodovém scénáři může být načasovaní zpráv čtených moderátory, předtočené videosekvence, znělky nebo časy vyhrazené pro živé vstupy. Systém poté během vysílání umožňuje z jednoho místa hlídat plnění naplánované délky jednotlivých částí zpravodajství a dle potřeby některou zprávu z plánu odstranit. Kromě toho se stará o to aby se tyto změny okamžitě promítly například do textů, které vidí moderátoři na teleprompteru.

Další významnou funkcí, která pomáhá při tvorbě rundownů, je schopnost systému shromažďovat a třídit zprávy zasílané televizní stanici od zpravodajských agentur formou RSS kanálů, emailů nebo zpráv vkládaných do systému reportéry dané stanice přímo z terénu. Tyto informace pak slouží jako základ pro jednotlivé story tvořící rundown zpravodajské relace. Přijaté zprávy, systémem nazávané wires, mohou obsahovat i obrazový materiál použitelný. Systém umí třídit došlé wiry podle uživateli zadaných pravidel podobně jako například mailoví klienti (např. Outlook, Thunderbird) . Kromě toho může být uživatel upozorněn na skutečnost že dorazila zpráva která by ho dle jeho nastavení mohla zvlášť zajímat.

Třetí významnou funkcí je správa médií, tedy předtočených video sekvencí. Octopus umožňuje prohlížet jednotlivá média, zařazovat je do vysílání nebo měnit díky propojení se softwarem pro editaci videa jako Apple Final Cut.

(http://www.octopus-news.com/, vlastní zkušenosti z vývoje Octopusu 6.)

Octopus má ještě mnoho dalších funkcí, výše uvedené je jen velmi hrubý nástin.

Octopus 6 je nejnovější verzí výše popsaného newsroomového systému, která je na rozdíl od svých předchůdců programována v jazyce Java. To především umožňuje, aby nový Octopus fungoval na všech operačních systémech schopných spustit interpreta Javy, JDK 1.6, což zahrnuje všechny běžné operační systémy současnosti. Java je objektový programovací jazyk a tak i Octopus 6 je založen na objektovém přístupu. Objekty poté vytvářejí vyšší celky spojené společným účelem, které v této práci budou nazývány komponentami. A právě vytvoření jedné takové komponenty si tato práce klade za cíl.

Za pomoci tzv. Agile Unified Procesu, metodiky vývoje softwarových aplikací kombinující zásady agilního programování s enterprise metodikou Rational Unified Process (obojí je blíže popsáno v podkapitole „Vysvětlení důležitých pojmů“), bude v této práci navrhnuta a vytvořena komponenta pro Octopus 6 zodpovědná za tvorbu logů, tedy souborů obsahujících informace o chybách a důležitých událostech během běhu aplikace.

Cíl

Cílem práce je vytvořit v jazyce Java logovací komponentu (balíček tříd), která by byla co nejvhodnější pro použití v aplikaci Octopus 6 společnosti Octopus Newsroom, s.r.o.

Vysvětlení důležitých pojmů

Tato sekce obsahuje stručná vysvětlení některých pojmů, jejichž vysvětlování v uvnitř textu by narušovalo jeho plynulost. Odkazy z textu do této sekce budou vždy ve formátu

(viz heslo z této sekce)

AUP – Agile Unified Process

Přístup k vývoji softwaru, který vznikl zjednodušením tzv RUP (Rational Unified Process) společnosti IBM. Přístup dělí vývoj softwarové aplikace na 4 fáze – inception (početí),

collaboration (rozpracování), construction (stavba) a transition (přechod) a zároveň napříč fázemi na 7 disciplín (např modelování, testování, implementace, projektový management,-atd.) Nejde o pevně daný scénář vývoje aplikace, ale spíše o hrubou kostru a sadu doporučení napsanou natolik obecně aby se daly snadno přizpůsobit konkrétnímu projektu. Více se lze o AUP dozvědět na webových stránkách společnosti Ambysof, které jsou dostupné na http://www.ambysoft.com/unifiedprocess/agileUP.html
(Amber 2005, AUP)

Java

Java je interpretovaný objektově orientovaný programovací jazyk. Mezi jeho typické vlastnosti patří statické typování (proměnná musí být vytvořena s definovaným datovým typem a ten se nemůže později změnit), vestavěná podpora pro primitivní typy (integer, long, double, char, atd.) a stringy. Kompilátor Javy překládá program do tzv. bytecodu, chod programu pak zajišťuje tzv. Java Virtual Machine – program který sadu instrukcí bytecodu provádí (tedy překládá do strojového kódu). I když výsledný strojový kód se liší podle operačního systému na kterém JVM běží, z pohledu Javy instrukce v bytecodu dělá na všech operačních systémech totéž. JVM tedy izoluje aplikaci od prostředí ve kterém běží a umožňuje tak kódu v Javě nezávislost na operačním systému. JVM také zajišťuje automatické uvolňování nepoužívaných bloků paměti (viz garbage collection)
(Keogh 2005 kap 1, Kiszka 2003 str 52)

Garbage collection

Vlastnost interpreta jazyka Java, tzv. Java Virtual Machine. (viz Java) JVM v pravidelných intervalech nebo v případě, že dochází volná paměť, prochází všechny právě existující objekty v běžícím programu. Pokud nalezne takové, na které už neexistuje reference z jiných objektů v programu JVM je zruší a jimi zabíranou paměť uvolní pro použití jinými objekty. Tento proces si umí poradit i s cyklickými referencemi (více objektů se odkazuje jen mezi sebou navzájem), v určitých specifických případech (handly na soubory či jiné zdroje operačního systému) ale Java stále ještě vyžaduje, aby běžící program nepoužívané objekty likvidoval sám.
(Keogh 2003 - str 91, Java API - třída System)

Verbosita logu

Určuje, jak podrobně log popisuje události, které v systému nastaly. Například při přesunu zprávy mezi 2 složkami zpráv log s nízkou verbositou nezaloguje nic, zatím co log s vyšší verbositou zaloguje jméno přesouvané zprávy a složky mezi kterými je přesouvána a log s nejvyšší verbositou zaloguje i obsah a všechny atributy samotné zprávy.

Vlákno (Thread)

Nazývá jednu posloupnost instrukcí programu.(libovolně rozvětvenou). Aplikaci nemusí tvořit jediné vlákno, většina aplikací dnes funguje na principu více vláken. Jedno vlákno může například zajišťovat, že aplikace reaguje na pohyby myši a klikání, zatímco jiné vlákno v pozadí se zabývá třeba manipulacemi se soubory. V jednovláknové aplikaci by uložení souboru způsobilo, že by aplikace nereagovala, dokud není ukládání hotové, protože instrukce by musely být vykonávány za sebou. Ve vícevláknové aplikaci je ale možné aby se reakce na pohyb myši odehrávala současně s ukládáním souboru. Fyzicky se oba procesy nedějí současně, ale procesor mezi ně dělí svůj čas. O to se ovšem stará operační systém a z pohledu uživatele ani z pohledu tvůrce aplikace to není důležité (víceprocesorové počítače umožňují současně běžet tolika procesům kolik procesorů mají). Používání více vláken je tedy často velmi výhodné, ale může způsobovat problémy tam, kde více threadů používá společný zdroj (blok paměti, soubor, atd) – viz Thread-safety
(Eckel, 2000 – kap 3, Lea, 2000)

Thread-safety

Tento pojem obecně znamená, že vícevláknová aplikace je napsána tak, aby souběžný běh jejích vláken nemohl způsobit žádný problém. Jde hlavně o to, aby bylo zajištěno, že pokud jedno vlákno pracuje s určitými daty, jiné vlákno je nemůže neočekávaně změnit. To obvykle znamená, že k datům používaným (a měněným) více vlákny smí přistupovat v jeden okamžik vždy jen jedno vlákno a že ostatní vlákna nepočítají s tím, že se data nezmění. Další riziko, kterého je třeba se vhodným návrhem aplikace vyvarovat je tzv. deadlock – tj. situace, kdy na sebe dvě a více vláken navzájem z jakéhokoliv důvodu čekají a ani jedno z nich tak nemůže pokračovat ve své posloupnosti instrukcí.
(Shirazi, 2000, Lea, 2002)

StackTrace

V jazyce Java jde o výpis momentálního stavu JVM (Java Virtual Machine). StackTrace tvoří výpisy zásobníků volání pro jednotlivá vlákna a výpis tzv. monitorů. Monitorům v této práci nebude věnována pozornost. StackTrace je zde využíván pouze ve své omezené podobě kterou vytváží JVM v rámci jediného vlákna pokud nastane vyjímka nebo o to program požádá – jde tedy o StackTrace pro jedno jediné vlákno, takze o pouhý výpis zásobníku volání. Ten je tvořen posloupností volání metod, která vedla k určitému bodu (například ke vzniku vyjímky). Takový to omezený stacktrace je seznam, na kterém každý řádek tvoří:

- název třídy,
- název metody této třídy, která byla zavolána,
- číslo řádku, na kterém došlo k zavolání další metody v posloupnosti.

Metody jsou ve výpisu seřazeny od nejvnitřnějších po vnější (tedy pokud A zavola B a ta zavola C, vyjímka vyhozená v C bude mít stacktrace v pořadí C, B, A) (Austin 1998) - citace

Time Stamp

Time Stamp je obecně řetězec, označující určitý moment v čase. V Javě se pro tyto účely používá číslo označující počet milisekund od začátku tzv. Unix Epoch, tj. od půlnoci (00:00:000) 1. 1.1970. Výsledné číslo je pro člověka prakticky nečitelné, ale z hlediska počtu zabraných bitů paměti je úspornější než kdyby si měl systém pamatovat jednotlivé části data (rok, měsíc, den, atd.).

1. Inception

1.1 Požadavky na logovací komponentu newsroomového systému Octopus

Zkušenosti, které společnost Octopus Newsroom získala poskytováním technické podpory pro dřívější verze řady Octopus, vedly ke stanovení následujících, značně podrobných, požadavků na budoucí logovací komponentu.

Logování s různými stupni verbosity

Při volání logovací komponenty a předávání zprávy, kterou by komponenta měla zalogovat, musí být možné určit stupeň důležitosti zprávy. Logovací systém pak musí mít v sobě možnost nastavení logování zpráv jen od určitého stupně důležitosti. To umožní omezit záznamy v logu během běžného chodu aplikace na ty které jsou důležité (omezit verbositu logu) a v případě potíží naopak získat logy co nejpodrobnější.

Tématicky rozdělené logy

Při prohledávání logů by mělo být možné snadno a efektivně odlišit zprávy týkající se rozdílných komponent a procesů probíhajících v rámci aplikace. Základní dělení by mělo být už na úrovni souborů, neměl by vznikat jeden velký log obsahující všechno co se v aplikaci děje.

Co nejkratší blokování logující aplikace

Vlákno které zavolá logovací komponentu nesmí čekat déle než je nezbytně nutné, aby se kvůli logování nezpomaloval běh samotné aplikace. Co největší část zápisu do logů by tedy měla probíhat v pozadí v samostatném procesu.

Možnost logování parametrů a objektů

Logovací komponenta by měla být schopná ve svém volání obdržet jeden nebo více dvojic parametrů (libovolných objektů) a jejich názvů. Tyto parametry by měla zalogovat tak, aby se zajistilo, že zalogovaný obsah bude co nejužitečněji popisovat předaný objekt, umožní vyhledat záznamy s parametry daného jména a nebo hodnoty a že obsah referencí na objekty předané logovací komponentě se nemohl změnit mezi předáním logovací komponentě a samotným zápisem do logů.

Možnost logování vyjímek

Logovací komponenta by měla snadno a jednoduše umožňovat zalogování objektu typu Exception (vyjímka), který v Javě vzniká pokud došlo v nějakém místě aplikace k vyjímečné situaci, jako jsou např. běhové chyby. Dále by také měla ve svém interfacu umožnit předání vyjímky a zapsat ji v co nejvhodnějším tvaru (tj. zapsat stacktrace a případné zřetězené vyjímky).

Rotování logů

Komponenta vytvoří vždy po určité době, obvykle jednou za den, nebo po určitém množství zalogovaných dat nový soubor, ve kterém pokračuje v logování. Starý soubor musí být rozpoznatelný jako log z určitého časového úseku. Po konfigurací stanovené době má být tento soubor zkomprimován, aby se šetřilo místo na disku Po další, opět konfigurací stanovené době má být soubor trvale odstraněn.

Parametrické názvy souborů

Komponenta musí umožňovat v určitých případech rozdělit některý z log souborů na několik tak, aby každý výsledný soubor měl v názvu hodnotu určeného parametru a mohl obsahovat jen záznamy se shodnou hodnotou tohoto parametru.

Příklad:

Systém by mohl mít několik logových souboru s názvem slozka-XXX, kde XXX bude název složky a do každého z nich logovat pouze informace o tom, co se dělo s danou složkou nebo zprávami v ní. Na rozdíl od předem určených tématicky rozdělených souborů, tento systém je dynamický, pokud v systému přibyde složka sám vytvoří automaticky odpovídající log.

Více možných úložišť

Architektura logovací komponenty musí být taková, aby se dala snadno rozšiřovat o další způsoby zápisu logů kromě tvorby log souborů. (například zápis do databáze, výpis přímo na obrazovku).

Thread-safety

Komponenta bude volána z více vláken a proto musí být zajištěno že mezi vlákny nemůže dojít k žádnému druhu konfliktu. (viz. kapitola Vysvětlení důležitých pojmů: Thread a Thread-safety)

Logování vláken

Z každého záznamu v logu musí být patrné, které vlákno vyvolalo zalogování dané zprávy.

Zachování posloupnosti událostí z více vláken

Ačkoliv logovací komponenta může být zavolána několika souběžně běžícími vlákny, která logují různě dlouhé záznamy, musí se pak tyto záznamy objevit v logu v takovém pořadí, v jakém byly komponentě předány, a ne v takovém v jakém je stihla zpracovat.

Zachování posloupnosti událostí ve více souborech

I když jsou záznamy rozděleny do více souborů podle tématu, kterého se týkají, musí existovat jednoduchý a jednoznačný způsob, jak zjistit, v jakém pořadí se záznamy z různých souborů po sobě udály.

Nezávislost na časovém pásmu

Ze záznamu v logu musí být jasné, kdy byl záznam zapsán, a to i bez zjišťování, v jakém časovém pásmu byla v danou dobu aplikace spuštěna.

Jednoduchý internace

Volání logovací komponenty by mělo mít co nejjednodušší syntaxi, při splnění všech výše uvedených požadavků.

Robustnost logovací komponenta

Logovací komponenta by měla být naprogramována tak, aby si dokázala poradit s vyjímečnými stavy, jako je např. nedostatek paměti, problémy se zápisem na disk, smazání logu uživatelem, atd., aniž by způsobila pád samotné aplikace.V případě chyby je důležité, aby bylo zalogováno co nejvíce informací a pokud to je možné aby z logu šlo vyčíst že v něm na daném místě chybí záznamy kvůli problému v logovací komponentě. Logovací komponenta také musí zajistit že se za žádných okolností nedostane do nekonečného cyklu.

1.2 Předchozí řešení logování v Javě

1.2.1 Log4J

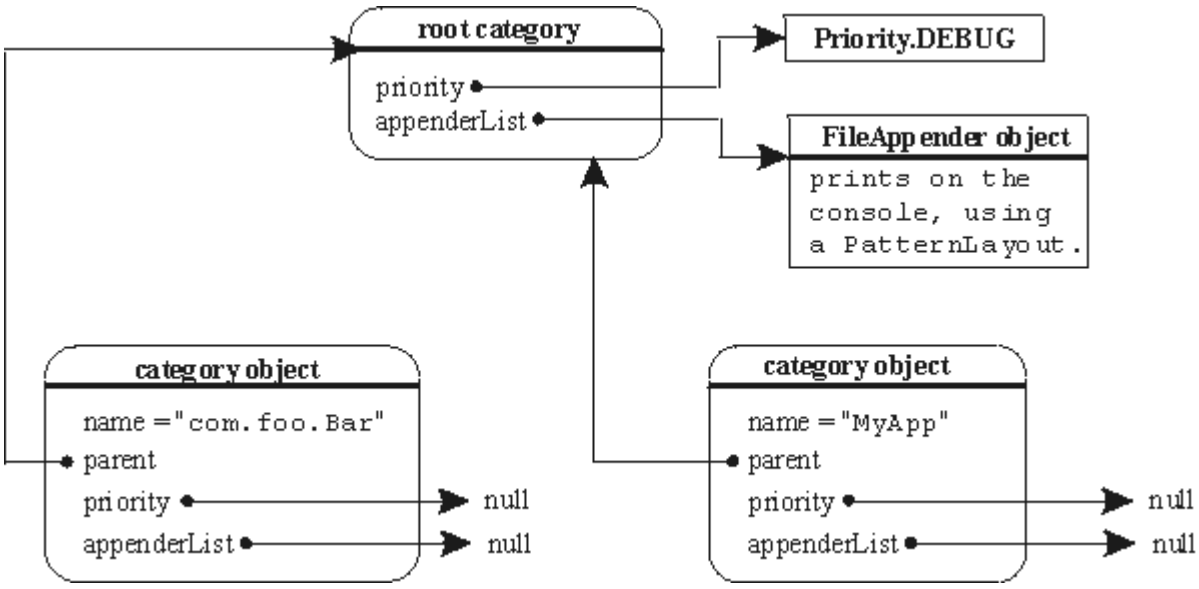
Log4J je balíček pro Javu vyvinutý organizací Apache, obsahující komplexní systém tříd pro tvorbu logů javovskou aplikací. Základem balíčku Log4J je třída Logger, přes kterou probíhá veškerá interakce vnější aplikace s Log4J, s výjimkou konfigurace. Každá instance třídy Logger má svoje jméno, které se potom objevuje ve všech záznamech vytvořených přes tuto instanci. Pokud chce aplikace používající Log4J něco zalogovat, zavolá na Loggeru jednu z metod fatal, error, warn, info, debug nebo trace. Názvy těchto metod odpovídají takzvaným LogLevelům, tedy stupňům důležitosti zprávy. Zavolání metody warn tedy zaloguje předanou zprávu s úrovní důležitosti warn. Zápis v logu vytvořeném Log4J pak vypadá následovně: Nejprve datum a čas, pak název Loggeru, pak úroveň důležitosti a nakonec samotná zpráva.

Příklad:

```
2000-09-07 14:07:41,508 [main] INFO  MyApp - Entering application.
```

Zápis logů probíhá pomocí takzvaných appenderů. Pro každý Logger je konfigurací určený seznam appenderů, které se starají o samotný zápis zalogované zprávy. Appender obecně je třída, která zapisuje zprávy vytvořené Loggerem do nějakého druhu úložiště. Typicky je to soubor (FileAppender), ale může jít i o výpis do konzole, do databáze či zasílání zpráv na adresu v síti. Log4J poskytuje velmi bohatou nabídku různých způsobů jak logy ukládat.

Konfigurace Log4J je poměrně komplikovaná, ale nabízí mnoho možností. Základem konfigurace je seznam používaných Loggerů, jejich úrovní logování (priority) a jejich appenderů. Názvy Loggerů jsou hierarchické a dělené tečkou, tedy Logger *systém.output* je podřízený Loggeru *system* a ten je nadále podřízený Loggeru *root* (kořen hierarchie, vytvořený automaticky). Každý Logger pak dědí vlastnosti svých předků. To znamená, že se seznam appenderů pro každý Logger skládá z appenderů daného Loggeru a ze seznamů všech Loggerů v hierarchii nad ním, a priorita Loggeru, pokud není nastavená, je schodná s prioritou jeho rodiče. Příklad hierarchie logů v konfiguraci:



(obr. 1 - konfigurace Log4J zdroj: Log4J API)

Log4J hodnocení

Log4J je pravděpodobně v současnosti nejkomplexnější logovací systém pro Javu, krerý splňuje většinu z požadavků na logovací komponentu pro systém Octopus.

Log4J umožňuje: - dělení logů dle určitých témat,

- má stupně důležitosti logovaných zpráv,
- umí používat více různých úložišť,
- logovat vyjímky,
- ukládat logy do určitým způsobem rotovaných souborů,
- není problém volat ho z různých vláken.

Mnoho z řešení použitých v tomto balíčku je vhodné pro použití v systému Octopus. Proto se nabízí možnost buď přímo použít Log4J nebo novou logovací komponentu vytvořit jeho rozšířením. Pro přímé použití by bylo třeba, aby Log4J splňoval všechny požadavky na logovací komponentu systému Octopus. Log4J ale přecijen selhává v několika málo ohledech:

- nemá systém umožňující přehledné zalogování parametrů v podobě objektu,

- rozdělení logů do souborů podle tématu je vhodnou konfigurací možné, ale u rychle za sebou jdoucích záznamů se již z více souborů nedá zpětně určit pořadí
- dodatečné dělení souborů s logy podle hodnoty parametru (viz „Požadavky na logovací komponentu systému Octopus“) je nemožné bez přidání vlastního appenderu.

Tím se dostáváme k druhé možnosti, a sice rozšíření log4J. Hlavní překážkou tomu je licence Apache Software Licence, pod kterou je Log4J distribuovan. Ta se hlásí k zásadám tzv. Open-Source hnutí (viz.“Vysvětlení důležitých pojmů“) a jakákoliv aplikace rozšiřující Log4J by musela být distribuována také jako open-source, to znamená zdarma a včetně zdrojového kódu. To je pro použití v komerční aplikaci jako Octopus nevhodné. Logovací komponenta by musela být napsána tak, aby nebyla součástí aplikace Octopus a neodkazovala do jiného kódu v Octopusu, jinak by musela být zveřejněna a volně distribuována společností Octopus Newsroom s.r.o. Licence Log4J ovšem nezakazuje (a ani zakazovat nemůže) využít ve vlastnoručně napsaném logovacím systému přístupy a postupy převzaté z Log4J, což se také v této práci mnohokrát stane.

1.2.2 Java.util.logging

Java.util.logging je balíček vytvořený přímo tvůrci Javy, společností Sun Microsystems, a je součástí základní distribuce jazyka Java. Tento balíček vznikl o něco později než Log4J a v mnoha bodech je mu podobný. Základní třídou je zde opět Logger a shoda s Log4J nekončí jménem. I tento Logger má hierarchické jméno a stejně jako v případě Log4J dědí vlastnosti Loggerů jejichž jméno je v hierarchii nad ním. I v java.util.logging existují stupně důležitosti, v případě tohoto balíčku se jedná o SEVERE, WARNING, INFO, CONFIG, FINE, FINER a FINEST. Každý Logger má opět nastavení, jak vysokou důležitost zpráv ještě bude logovat, tedy odmítne požadavek o zalogování zprávy s nižší důležitostí. Samotný zápis zpráv do souborů či jiných úložišť opět řeší speciální druh tříd, v java.util.logging nazývaných Handlery. Např. pro prostý zápis do souborů existuje třída FileHandler. Určitý rozdíl je v tom, že Handlery na rozdíl od Appenderů v Log4J mají vlastní nastavení stupně důležitosti a je tak možné, aby určitou zprávu zalogovala jen část Handlerů přiřazených k danému Loggeru.

Typickou vlastností java.util.logging je, že zalogované zprávy v sobě obsahují informaci o tom jaká třída a jaká metoda v ní je původcem zprávy. Při volání Loggeru existují kromě sady metod, pojmenovaných podle stupňů důležitosti (severe, warning, etc..), také obecnější metody

log, logp a logrp, které umožňují zalogovat zprávu s předaným jménem třídy a metody, které je odliškné od místa, odkud jsou volány a sadu jednoho nebo více parametrů (nepojmenovaných). Logger má také sadu metod určených speciálně pro zalogování faktu že program vstupuje nebo vystupuje z metody. Všechy tyto metody slouží pouze jako interface pro privátní metodu log (LogReport report), kde LogReport je speciální objekt popisující zprávu k zalogování, vytvořený na základě kombinace informací předaných při volání Loggeru a defaultních nastavení (pro ty informace které Loggeru předány nebyly; pro jméno třídy a metody to znamená že Logger se je pokusí získat ze stacktracu – viz „Důležité pojmy“).

Konfigurace komponenty java.util.logging je typicky (jako u Log4J) čtěna ze souboru třídou LogManager, může být ale také vytvořena speciální třída, která se stará o přečtení konfigurace (odkud to pak záleží čistě na implementaci uvnitř této třídy). Tato konfigurace je také založena na hierarchičnosti Loggerů a dědičnosti v rámci této hierarchie, obsahuje ale také sadu dvojic jméno-hodnota, které slouží pro obecná nastavení například pro Handlery.

java.utils.logging hodnocení

I tento balíček splňuje většinu požadavků na logovací komponentu systému Octopus. Logování názvů tříd a metod odkud zpráva pochází je sice nad rámec požadavků ale může být určitě výhodou. Balíček rozhodně splňuje podmínky dělení na stupně důležitosti, dělení podle témat, různé způsoby zápisu logovaných zpráv, thread-safety, možnost logování parametrů a vyjímek, a další.

Jako u Log4J, i zde není logování vláken odkud zpráva pochází a není zajišťěn jednoznačný způsob jak identifikovat pořadí, v němž se zalogované zprávy udály (je možné, až to z dokumentace neplyne, že je zajištěno pořadí při logování do jednoho souboru, není ale mechanismus jak seřadit záznamy uložené na více místech). Balíček také sám o sobě neumožňuje dávat parametům zalogovaným se zprávou jména a neumí objekty zalogovat specifickým způsobem, tedy například aby objekt používaný Octopusem k přenosu určité zprávy zalogoval tak, aby byla tato zpráva snadno čitelná, a podobně.

To vše by se dalo relativně snadno vyřešit rozšířením balíčku o několik vlastních tříd, kterému v případě java.utils.logging nikterak nebrání licence. Jde o součást Javy SE (Standard Edition) a tedy neexistují podmínky omezující rozšiřování balíčku. Navíc je balíček postaven tak, že s rozšiřováním počítá a je tudíž značně usnadněno vhodnou architekturou balíčku.

Z důvodů které nemohou být v této práci uveřejněny (jde o interní informace společnosti Octopus Newsroom s.r.o.) ale tento přístup nebude použit. Obzvláště společné body architektur java.utils.logging a Log4J, ale budou sloužit jako základ návrhu logovací komponenty navržené na následujících stránkách.

1.3 Volba nástrojů pro návrh a vytvoření logovací komponenty

Jelikož je nezbytně nutné, aby komponenta byla napsána ve stejném jazyce jako zbytek Octopusu 6, tedy v Javě, bude tomu tak. Pro vývoj aplikace v jazyce Java je velice důežitá především sada nástrojů souhrnně nazývaná JDK (Java Development Kit), která je zdarma distibuována společností Sun Microsystems. Z důvodu kompatibility se zbytkem aplikace Octopus 6 bude při implementaci logovací komponenty použita JDK verze 1.6. Druhým v praxi nezbytným nástrojem je IDE (Integrated Development Environment), což je jakási pokročilejší verze klasického tetového editoru, která provádí průběžnou kontrolu syntaxe, umožňuje:

- automatické doplňování tříd a metod, a to i po napsání alespoň 1 písmena a stisku příslušné zkratky nabídne možnosti dle kontextu,
- doplňování importů z definovaných knihoven,
- obsahuje debugger a profiler, nástroje, které velmi usnadňují hledání chyb (první) nebo neefektivního kódu (druhé). IDE toho dnes zvádne ještě mnohem více a předchozí popis je jen velmi rámcový. Konkrétní použité IDE bude, opět z důvodů kompatibility s Octopusem 6, IDE Netbeans 6.0. Ačkoliv není problém kombinovat kód napsaný pomocí různých IDE, byla by to v tomto případě zbytečná komplikace. IDE Netbeans 6.0 je open-spource (viz open source) a tedy zcela zdarma k používání, a to i v případě, že je použito k vývoji komerčních aplikací.

2. Elaboration

2.1 Formát vytvářených logů

Než bude možné vytvořit návrh budoucí logovací komponenty, je třeba navrhnout, jak má vypadat výstup z této kmponenty. Jak má vypadat vstup, je už dostatečně určeno v kapitole zabývající se požadavky na komponentu, co ale není zatím určeno, je jak by měl konkrétně vypadat log vytvořený logovací komponentou. Za vzor bude použita běžná skladba logů z jiných systémů, textový soubor, ve kterém každý záznam představuje jeden řádek. Řádek je pak rozdělen na několik polí, a to předem daným oddělovačem (například tab, pipe, carka, strednik). První pole tvoří obvykle datum a/nebo čas, kdy záznam vznikl a na konci jsou pak obvykle volitelné položky, které nejsou součástí každého záznamu (jako např. sada volitelných parametrů). Vzorem může být tento už jednou představený řádek z logu vytvořeného Log4J:

```
2000-09-07 14:07:41,508 [main] INFO  MyApp - Entering application.
```

V této podobě by ale log nestačil. Jak už bylo řečeno v hodnocení Log4J, nesplňuje takový log dvě kritéria, stanovená pro logovací komponentu systému Octopus – neumožňuje identifikaci vlákna, ve kterém zpráva vznikla a nelze z něj určit pořadí události vůči jiné události z jiného souboru.

První problém má triviální řešení v podobě přidání pole s identifikátorem vlákna. Vhodné se jeví threadId, unikátní identifikátor který má přiděleno každé vlákno v aplikaci Java.

Druhý problém je o trochu složitější. Při použití času včetně milisekund, jak je tomu ve výše uvedeném příkladě z Log4J, by se zdálo, že pro určení pořadí stačí pouhý čas. Jedna milisekunda není ani zdaleka dostatečně malým časovým úsekem, aby během ní nemohlo vzniknout více zpráv. Java sice obsahuje metodu (System.nanoTime), která umožňuje měřit čas v nanosekundách, ale jednak umí pouze zjistit čas relativně k jinému zavolání této funkce (neukazuje současný čas, ale jen číslo které od spuštění programu roste po nanosekundách) a zadruhé přesnost tohoto údaje není nijak garantována (SUN, Java API, třída System)

Jelikož kvůli zajištění jiného z požadavků, tedy aby byla zachována sekvenčnost vzniku zpráv při volání logovací komponenty z různých vláken je nezbytné aby někde v logovací komponentě existovala takzvaně synchronizovaná metoda, tj. metoda, do které mohou vlákna přistupovat jen po jednom a která tedy nedovolí jednomu vláknu předběhnout jiné, které si o zalogování řeklo dříve. Nejsnazší řešení výše nastíněného problému, je tedy v této metodě přidělit každému

záznamu nějaký druh sériového čísla, které bude v rámci běhu logující aplikace unikátní a bude se vzestupně zvětšovat.

Po přidání sekce s parametry na konec záznamu tedy upravená podoba formátu logu vypadá takto:

```
2000-09-07 14:07:41,508 5000342  17
(Datum)       (čas)          (sekv. čís.)(vlákno)
 [main] INFO  entering MyApp. ???
 (Logger)(stupeň)(text zprávy)       (param.)
```

Ani tato podoba ale není konečná. Z příkladu je patrné, že záznam v logu je zbytečně dlouhý na to, že obsahuje jen velmi stručnou informaci o vstupu do aplikace MyApp. Druhým nedostatkem je, že jsme zatím neurčili podobu zápisu parametrů, proto jsou v příkladě o pár řádků výše místo nich použity otazníky. Navíc zjištění pořadí záznamů z více souborů je teď možné, ale stále ještě není pro člověka pročítajícího logy příliš snadné ani rychlé, bude tento návrh počítat s tím, že pro čtení logů bude vytvořena jednoduchá aplikace. Ta bude schopná spojit obsah zadaných souborů do jedné posloupnosti událostí. Kromě filtrování, vyhledávání a dalších běžných funkcí aplikace pro čtení logů. Pokud pro čtení logu bude použita aplikace, je možné jejich formu výrazně zjednodušit. Formát už nemusí být tak snadno čitelný, i když kvůli nečekané potřebě bude zachována dostatečná čitelnost.

V první řadě tabulátory, které uspořádají text do přehledných sloupců a zaplňují řádky nevyužitým místem, lze nyní nahradit jiným oddělovačem (například pipa). Zadruhé již není třeba psát datum a čas ve snadno čitelném formátu. Postačil by tzv. timestamp (viz time stamp), ten je ale pro člověka zcela nečitelný. Proto dostane přednost jednoduchý zápis pomocí čísel s vypuštěním oddělovačů. Např. **2000-09-07 14:07:41,508** bude tedy zapsáno jako **20000907140741508.**

Za tímto číslem by ale mělo následovat ještě seriové číslo záznamu. To je zbytečné. Pro jednoznačné odlišení záznamu postačí (a dokonce se více hodí) použít jen krátké přádavné seriové číslo rozlišující záznamy v průběhu dané milisekundy. Toto číslo pak připojíme k číslu označujícímu datum a čas (jelikož počet záznamů těžko přesáhne pár desítek, měly by stačit tři číslice, tedy omezení na 1000 záznamů za milisekundu). Zbývá tedy vyřešit, jak vhodně zapsat parametry.

2.1.1 Formát parametrů

Bohužel univerzální způsob, jak zapsat libovolný objekt v Javě tak, aby nám jeho zápis dával co největší smysl, neexistuje. Proto do návrhu logovací komponenty bude zahrnut interface Loggable.

Každý objekt, který bude tento interface implementovat, bude mít jednu metodu, která vrátí text co nejlépe popisující daný objekt pro účely zalogování. To samozřejmně nepůjde použít pro instance tříd, které nejsou součástí aplikace Octopus a tudíž nemohou interface implementovat, aniž by od nich byly vytvořeny rozšiřující třídy. To ale nevadí, protože požadavek na specifické logování obsahu objektů je na logovací komponentu kladen kvůli několika třídám používaným v aplikaci Octopus pro transfer dat. Tyto třídy budou implementovat zmíněný interface a samy se starat o to, aby byly zalogovány v co nejprospěšnější podobě, zatímco ostatní objekty budou logovány pomocí obecné metody toString, kterou implementuje povinně každý objekt v jazyce Java. Ačkoliv obecná implementace u tříd které ji nepřepisují svou vlastní vypíše jen název třídy a hashcode instance.

Zbývá ještě přiřadit k takto získaným zápisům předaných objektů jejich jména (požadavky říkají že parametry mají být pojmenované). tečka Toho by šlo docílit lehce pomocí konvence jméno, rovnítko, hodnota. Vzhledem k tomu, že hodnotou v tomto případě často bude výpis obsahu objektů, které samy obsahují více položek s názvem a hodnotou, bude pro tyto účely vhodnější zvolit formát XML, který takovéto vkládání parametrů do parametrů velmi snadno umožňuje. Názvem XML tagu bude název parametru, jeho obsahem pak obsah objektu. Aby se zajistila XML struktura takto zapsaných parametrů, bude interface Loggable ve své jediné metodě vracet obsah objektu ne jako string, ale jako XML node. Příklad parametrů:

<data><d1>145</d1><d2>356</d2></data><text>bla bla bla</text>

2.2 Návrh architektury logovací komponenty

Architektura budoucí logovací komponenty bude vycházet z architektur 2 jiných výše popsaných řešení podobného problému, tedy Log4J a java.utils.logging. To znamená, že interakce s logovací komponentou bude probíhat přes 1 třídu, Logger. Jednotlivé instance budou vytvářeny a spravovány jedinou instancí jiné třídy, v tomto návrh nazvanou třeba LogEngine. Zápis do souborů nebo jiných úložišť pak budou mít na starosti třídy implementující interface Appender, které obdrží řádek připravený k vložení do logu a uloží ho podle svého určení. Tedy FileAppender do souboru, ConsoleAppender vypsáním do konzole, atd. V tomto návrhu bude vytvořen pouze jeden Appender, a to FileAppender. Požadavky na logovací komponentu ale stanoví, že architektura systému musí počítat s možností, že Appenderů bude v budoucnu více.
Vzhledem k tomu, že logovací komponenta je určena jedné konkrétní aplikaci, nemusí být její konfigurace příliš všestranná. Vytváření a načítání konfiguračních souborů by bylo zbytečnou komplikací, protože komponenta může pro svoje nastavení využít konfiguračních proměnných aplikace Octopus. Navíc není třeba mít možnost nastavovat strukturu Loggerů. Vzhledem k tomu, že zdrojový kód Octpusu musí počítat s určitým uspořádáním Loggerů vytvářením této struktury na základě jakékoliv vnější konfigurace dává jen zbytečný prostor pro vznik chyb. Bude tedy výhodnější zakomponovat strukturu Loggerů přímo do zdrojového kódu tak, aby se jakýkoliv rozpor mezi strukturou logů očekávanou Octopusem a tou kterou po své inicializaci vytvoří LogEngine projevil už při kompilaci. Proto v tomto návrhu instance třídy Logger nebudou rozlišovány názvem, jak je tomu u dříve popsaných logovacích balíčků, ale bude zavedena třída LogTopic. Stejně tak hierarchická struktura LogTopiců bude pro zajištění správnosti převedena na hierarchii objektovou. K tomu bude využita speciální třída enum, která umožňuje v Javě vytvářet výčty možností, a ty pak mezi sebou porovnávat. Jelikož enum může obsahovat další, vložené enumy, pro vytvoření hierarchie se hodí. Jelikož ale Java neumožňuje vícenásobnou dědičnost, a každá položka enumu se považuje za odvozenou od třídy Enum, bude LogTopic pouze interface, který budou všechny položky hierarchie implementovat.
Kromě struktury Loggerů, nebo přesněji LogTopiců které Loggery budou používat, je třeba také stanovit nastavení stupňů důležitosti. Každý požadavek na zalogování zprávy by měl mít v sobě uveden stupeň důležitosti, který se porovná se stupněm důležitosti přiřazeným danému LogTopicu, a pouze pokud bude stejný nebo vyšší bude požadavek skutečně zalogován. Nejprve je třeba stanovit samotné stupně. Pro tento účel bude v návrhu sloužit třída LogLevel,

která bude (tentokrát nehierarchický) enum. Hodnoty enumu budou FATAL, ERROR, WARN, INFO, TRACE a DEBUG, v tomto pořadí, které bude sloužit jako pořadí důležitosti (enum umožňuje porovnávat 2 svoje hodnoty podle pořadí), kde FATAL je nejdůležitější a DEBUG nejméně důležitý LogLevel. Zbývá ještě přiřadit jednotlivým LogTopicům jejich LogLevely. Bylo by možné vytvořit například HashMapu a do ní vložit dvojice LogTopic LogLevel. tečka V tomto návrhu ale bude využita už existující struktura LogTopiců. V interfacu LogTopic bude jedna jediná metoda, nazvaná getParams(). Na její zavolání vrátí každý LogTopic v hierarchii svoje parametry. To znamená svůj počáteční LogLevel (změnitelný za chodu) a jméno souboru, do kterého se má ten který LogTopic ukládat. Jméno je jen relativní, bez cesty k souboru nebo přípony, a jiné appendery ho mohou používat způsobem který dává smysl v jejich kontxtu – například pro uložení do databázové tabulky toho jména. Více LogTopiců může být ukládáno do stejného souboru, topic je obsažený v logovaných řádcích a tak není třeba ho odvozovat z názvu souboru. Naopak v mnoha situacích by rozdělení tématicky souvisejících topiců bylo zbytečnou komplikací. Aačkoliv pro čtení logů se v tomto návrhu počítá s GUI aplikací schopnou rozdělení záznamů mezi různé soubory zakrýt a vedlo ke zbytečně velkému počtu log souborů.

Jak bylo už řečeno, pro interakci s logovací komponentou bude sloužit třída Logger. Ta bude mít podobně jako u jiných logovacích balíčků sadu metod nazvaných podle stupně důležitosti (fatal, error, warn, info, …). Tyto metody budou přetížené, tzn. budou existovat v několika verzích lišících se vstupními parametry. Každá instance Loggeru bude mít v sobě defaultní LogTopic, a pokud bude zavolána logovací metoda, která mezi parametry nemá topic, bude použit defaultní topic Loggeru. Každý Logger, ale bude mít i sadu metod umožňující zalogování pod jiným LogTopicem, aby nebylo nutné vytvářet novou instanci Loggeru kvůli zalogování malého počtu zpráv pod jiným topicem. Každá logovací metoda bude mít také na konci nula až n dobrovolných dvojic jméno-parametr. Jejich prostřednictvím budou logu předávány parametry (viz „Formát logů“). V případech, kdy je ve větším množství volání loggeru potřeba aby se stále opakoval stejný parametr, bude možné nastavit defaultní sadu parametrů na instanci Loggeru. Ty poté budou přidávány k parametrům předaným při volání logovací funkce a to tak, že obě skupiny parametrů se sečtou a v případě schody jmen mezi parametrem z defaultu v Loggeru a parametrem předaným voláním metody má přednost parametr předaný (override defaultního nastavení). Protože by bylo zbytečné, aby existovalo více instancí Loggeru se shodnými defaultními parametry, nebudou instance třídy Logger vytvářeny konstruktorem, ale tzv. factory

(továrními) statickými metodami třídy LogEngine. LogEngine si pak bude udržovat cache vytvořených Loggerů a při více požadavcích loggeru se stejným LogTopicem a defaultními parametry bude vracet stále stejný Logger. Kromě toho bude existovat metoda vracející defaultní Logger, tedy jakousi základní instanci Loggeru s topicem nastaveným na LogTopic jménem NO_TOPIC a prázdnou sadnou defaultních parametrů. Protože továrními metodami vytvořené Loggery mohou být sdíleny více třídami i více vlákny, nebude na nich dovoleno po vytvoření měnit LogTopic ani parametry (budou tzv immutable). Bude ale možné vytvořit jejich kopii s nastavitelným topicem i parametry pro případy, kdy by toto omezení znamenalo problém. Každá logovací metoda po tom co podle potřeby doplní jí předané parametry defaulty svého Loggeru zavolá jednu společnou, obecnou metodu jménem log. Ta nejprve posoudí, jestli předaný LogLevel je dostatečně vysoký, aby byla zpráva skutečně zalogována, to znamená jestli je stejný nebo vyšší než momentální LogLevel předaného LogTopicu. Pokud ano vytvoří z předaných informací instanci třídy LogRequest, která vyjadřuje požadavek na zalogování. Při tvorbě LogRequestu dojde k převedení všech referencí na živé objekty na je popisující stringy, aby se zamezilo tomu, že se obsah referencí mezi tímto bodem a samotným zápisem do logu změní. Také se zkontroluje, jestli předané parametry jsou skutečně dvojice název-hodnota a hodnoty jsou převedeny podle jednoduchého schématu. Pokud implementují interface Loggable (viz Formát logu) je dotazována jejich metoda toLogXML, jinak je použita obecná metoda Objektu, metoda toString, a poté obalena XML tagem s názvem parametru.

Aby bylo možné přiřazovat jednotlivým logovaným zprávám sériová čísla (viz kapitola Formát logů) je nutné, aby všechny zprávy prošly jedním “hrdlem“. Vhodným místem pro něj je tedy single instance LogEngine. Instance třídy Logger pro něj řádek výše popsaným způsobem připraví, a synchronizovaná metoda mu pak přidělí seriové číslo vytvořené z data a času v UTC. (Coordinated Universal Time) a dodatkových čísel rozlišujících požadavky, které projdou touto metodou v průběhu každé milisekundy. Pokud má požadavek stejnou milisekundu jako předchozí, jeho dodatkové číslo je o jednu větší než u předchozího, pokud nemá, tvoří ho samé nuly.

V tomto bodě je tedy už požadavek na zalogování (LogRequest) plně vytvořen a mohl by být předán appenderům náležícím k jeho LogTopicům. To by ale znamenalo, že vlákno, které zavolalo logovací metodu vedoucí k vytvoření tohoto LogRequestu by muselo čekat, až bude všemi appendery zapsán do jejich úložišť. To je zbytečné. Tak jako na jedné straně hrdla byly instance třídy Logger, vytvářející LogRequesty, budou na druhé straně čekat třídy Log. Pro

každou sadu LogTopiců se stejným jménem úložiště (např souboru) existuje právě jedna třída Log, která se stará o zápis těchto požadavků. Ta v sobě bude obsahovat frontu, do které LogEngine předá hotový LogTopic, a tím je z jeho pohledu logování dokončeno a může vrátit kontrolu metodě, která o zalogování požádala, aniž by ji nutil čekat, až proběhne samotný zápis, který trvá obvykle řádově déle než všechny kroky před ním. Každý Logger pak má svoje vlastní vlákno které z druhého konce fronty průběžně odebírá LogRequesty a prostřednictvím Appenderů je zapisuje do souborů. To umožňuje rozložit zápis požadavků do souborů do většího časového úseku, aniž by byla brzděna samotná aplikace.

Pokud požadavky do fronty neustále přibývají rychleji něž je Log stíhá zpracovávat (může být i proto, že například zápis do souboru je dočasně nemožný), fronty se prodlužují a hrozí, že zaberou příliš mnoho paměti a celá aplikace selže díky nedostatku operační paměti. Na obranu před touto situací je třeba zavést ochranné mechanismy. Aby to bylo možné, je třeba mít k dispozici alespoň přibližný údaj o délce front. Toho lze docílit tak, že se při každém vložení LogRequestu do fronty současně zvýší počítadlo délky, a naopak při vyjmutí z fronty se počítadlo sníží. LogRequesty ale mohou být velmi nestejné velikosti a tak by tento způsob nebyl příliš efektivní. Za odhad velikosti nejlépe poslouží délka stringu hotového řádku. Při přidání do fronty na některém z Logů se tedy toto číslo přičte k počítadlu a naopak. Pokud by se měl do některé fronty vložit požadavek, přestože společná délka front přesahuje stanovenou hranici, nabízí se dvě možnosti. Buď nechat logující vlákno čekat dokud se ve frontě neuvolní místo, nebo požadavek zcela zamítnout a do fronty nevložit. Ani jedno není ideální. Pokud LogEngine požadavek odmítne, budou v logu chybet záznamy. Log, na který se nelze spolehnout, že obsahuje vše, co má není příliš užitečný. Pokud bude blokovat logující vlákno, bude logovací komponenta aplikace zpomalovat a je tu i riziko, že aplikace přestane odpovídat. Pokud by se logování delší dobu nedařilo, například kvůli problémům při zápisu do souboru nebo databáze. Proto návrh počítá s oběma mechanismy, kdy při určité velikosti logů začíná blokování, které ale má jen omezenou dobu trvání. Pokud i při takovémto zpomalení logujícího procesu délka fronty neklesá, ale naopak překročí další hranici, začne logovací komponenta zahazovat zprávy. Zahazování je vícestupňové, podle velikosti fronty se zahazují buď jen zprávy stupně DEBUG, nebo všechny nechybové zprávy (tedy DEBUG, TRACE a INFO) nebo všechny zprávy až na fatální chyby (FATAL), nebo konečně v nejextrémnějším případě všechny. O každé změně režimu zahazování se logovací komponenta pokusí vytvořit zápis v logu s názvem logging, aby

bylo možné alespoň zjistit fakt, že v době od času a do času b nebyly zprávy se stupněm X a menším logovány.

3. Construction

Hlavním výsledkem poslední fáze, fáze konstrukce, je zdrojový kód v příloze této bakalářské práce. Následující kapitoly slouží jako rozšířené komentáře k základním třídám v tomto zdrojovém kódu. Tyto komentáře se nesnaží nahradit zdrojové kódy tím, že by plně popisovaly všechny algoritmy a struktury v nich, ale mají sloužit pro usnadnění orientace ve zdrojových kódech a přispět k jejich pochopení.

3.1 Logger

Třída Logger bude z hlediska zdrojového kódu tvořena několika částmi. Nejprve v ní bude obsažena sada hierarchických statických enumů implementujících LogTopic, která bude tvořit strukturu LogTopiců. Ta by mohla být stejně dobře obsažena v kterékoliv jiné třídě logovací komponenty, ale umístění v Loggeru je poměrně logické, protože LogTopic je z hlediska volání logovací komponenty parametrem Loggeru. Dále následují statické metody, které pouze volají odpovídající metody na LogEngine – aby z hlediska vnější aplikace tvořila interface k logovací komponentě jen třída Logger. Jsou to metody na přidání odebrání appenderů a na nastavení LogLevelů pro LogTopicy a tovární metody vyrábějící instance Loggeru.
Následuje nejdůležitější sada metod, metody logovací. Těch bude 6x4 veřejných a jedna privátní. Veřejné metody jsou vždy tři pro každý stupeň důležitosti a liší se parametry (tedy tři metody fatal, tři metody error, atd). Každá z těchto metod má za účel jen a jen zavolání jediné privátní metody log(). K tomu účelu nejprve doplní jí přidané informace dle potřeby defaultními nastaveními a pak zavolá log().
Metoda log přijímá LogLevel - získáný z toho, které metoda byla zavolána, LogTopic - pokud nebyl metodě předán použije se defaultní topic Loggeru, Throwable - tedy například vyjímka, nemusí být, může být předána prázdná reference - null, samotnou zprávu (textový řetězec) a pole parametrů (lichá místa v poli názvy, sudá hodnoty).
Metoda log() zavolá stejně pojmenovanou metodu LogEnginu. Ta zkontroluje zda LogLevel je dostatečně vysoký, aby pro daný LogTopic směla být zpráva zalogována. Pokud ano, vytvoří LogRequest, objekt popisující požadavek na zalogování způsobem, který už neobsahuje reference na objekty předané metodě log() (více v kapitole „Třída LogEngine“).

3.2 LogEngine

Třída LogEngine má dvě základní funkce. Zaprvé obsahuje metody pro inicializaci logovací komponenty, především metodu createLogs která projde strukturu LogTopiců a pro všechny unikátní názvy úložišť vytvoří instance třídy Log. Poté si vytvoří interní mapu LogTopiců na tyto Logy, která bude od té doby sloužit k tomu aby LogEngine věděl kterému Logu má předat který požadavek na zalogování. Po této inicializací se hlavní úlohou LogEnginu stává přidělování sériových čísel LogRequestům a jejich předávání správné instanci Logu. Poslední, a z hlediska zdrojového kódu, největší funkcí LogEnginu je pak dozor nad délkou front požadavků na zalogování. LogEngine měří délku těchto fron tak, že pro každý LogRequest který jim projde, zvyšuje počítadlo celkové délky front, a které zavoláním jedné jeho metody mezitím snižují Logy kdykoliv úspěšně vybaví některý požadavek ze své fronty. LogEngine si během inicializace podle celkového množství paměti dostupného JVM (viz „Důležité pojmy – Java“) stanoví 3 hranice maximální délky front. Pokud je překročena první, začíná LogEngine krátkodobě blokovat logování zpráv stupně DEBUG maximálně o desetinu vteřiny. Pokud ani to nepomohlo, přidá do fronty LogRequest s informací o tom, že logovací komponenta pro nedostatek paměti přestala logovat zprávy stupně DEBUG a začne jakékoliv příští zprávy tohoto stupně zahazovat. Pokud se fronty zmenší, vytvoří LogRequest oznamující, že logovací komponenta už opět loguje všechny druhy zpráv. Jestliže se fronty naopak dále rozrůstají, vyřadí LogEngine obdobným způsobem logování stupňů TRACE a INFO, a pokud ani to nepomůže tak i WARN a ERROR (omezí se tedy jen na FATAL) V případě, že i teď fronty dál rostou, což prakticky s jistotou znamená že aplikace selhává na více místech – odtud fatální chyby – a zároveň, že Logy nejsou schopny své fronty vůbec zpracovávat – například proto, že potřebují zapsat na disk, který je ale momentálně plný) pokusí se o tom logovací komponenta zapsat zprávu a logování přeruší úplně.

LogEngine má také funkce pro přidávání a odebírání appenderů Logům, protože samotné instance Log jsou z vnějšku logovací komponenty nedostupné. Tyto funkce pro správu appenderů jsou volány z vnějšku přes proxy metody na třídě Logger.

LogEngine má také vlastní vlákno, které v pradilených intervalech projde existující log soubory (pokud je na konfigurací zadané cestě najde) a jestliže jsou starší než hranice určená konfigurací, zabalí tyto soubory do formátu zip. Stejně tak pokud jsou starší než jiný hranice, pak tyto soubory zcela smaže. Při určování stáří souborů se LogEngine neřídí informacemi filesystému o

stáří souboru, ale názvem adresáře – logovací komponenta vytváří jeden adresář pro každý den, a z jeho názvu je možné zjistit datum, kdy byly logy v něm obsažené vytvořeny. Při smazání těchto logů je samozřejmně smazán i adresář.
Poslední funkcí LogEnginu je pak funkce shutdown(), která se zavolá při ukončování aplikace. Ta má za úkol ukončit činnost všech logovacích vláken, ale nedovolit zavření aplikace dokud tato vlákna nedostanou čas na zalogování všech požadavků ve svých frontách. Jak vlákna postupně dokončují vyprazdňování své fronty, oznamují LogEnginu konec činnosti, a teprve když ho oznámí všechna vlákna nebo po určitém maximálním čase dovolí LogEngine aplikaci aby skončila. Funkci shutdown() volá JVM při ukončení aplikace jelikož je během inicializace logovací komponenty tato metoda nastavena JMV jako tzv. shutdown hook – metoda bez jejíhož zavolání aplikace nesmí ukončit běh (pokud nekončí běhovou chybou).

3.3 Log

Třída Log je tvořená jednak frontou LogRequestů, a jednak vláknem které z této fronty odebírá požadavky a pomocí tomuto Logu přidělených appenderů je zapisuje. Toto vlákno tvoří nekonečný cyklus, který vždy zjistí jaká položka ve frontě je na řadě. Předá ji všem Appenderům, a pokud zprávu zalogují všechny tzv. primární appendery. Například FileAppender je primární appender, zatímco ConsoleAppender, vypisující řádky přímo do konzole, je sekundární appender. Odebere požadavek z fronty a oznámí LogEnginu, že o délku požadavku může snížit svoje počítadlo délky front. Pak se cyklus opakuje. Pokud se stane, že cyklus zjistí že je fronta prázdná, přeje do klidového režimu – sdělí Appenderům že přechází do klidového režimu (což např FileAppender využije k tomu aby zavřel svůj soubor a dal šanci jiným procesům aby s ním v tu dobu manipulovaly) a začne zkoušet jestli je fronta stále prázdná s relativně dlouhým intervalem dvou desetin vteřiny. Jakmile ve frontě nějaký záznam objeví, pokračuje zase způsobem popsaným výše.
Třída Log má kromě metody na přidání LogRequestu do fronty ještě důležitou metodu shutdown(). Tu volá LogEngine na všech instancích Logu během ukončování aplikace. Jestliže byla zavolána metoda shutdown, vlákno vyprazdňující frontu pokračuje v činnosti jed do chvíle kdy je jeho fronta prázdná, jakmile k tomu dojde, ukončí činnost a oznámí LogEnginu, že úspěšně skončilo.

3.4 FileAppender

Tato třída je základní implementací interfacu Appender. Základní funkcí třídy je metoda append, která vezme LogRequest a zapíše ho do svého log souboru. Druhým parametrem je hodnota parametru používaného jako přípona souboru. Pokud takový parametr neexistuje, je hodnota rovna prázdné referenci. File appender umí pracovat se skupinou souborů, které mají názvy odvozené od hlavního souboru a doplněné hodnotou tohoto parametru. File appender pak podle této hodnoty requesty rozděluje mezi tyto soubory a podle potřeby dovytváří soubory pro hodnoty parametru, které zatím neznal. Pro hodnotu parametru rovnou null jsou LogRequesty zapisovány do původního souboru bez přípony.

Metoda shutdown je pokračováním řetězce metod začínajícího metodou shutdown v LogEnginu. Shutdown FileAppenderu znamená, že appender dokončí zápis LogRequestu, který právě zpracovává, vyprázdní všechny buffery přes něž zápis kvůli urychlení prováděl a předá kontrolu opět metodě, která shutdown zavolala.

Metoda rest je volána vláknem patřícím k některé instanci třídy Log (viz komentář ke třídě) Tato metoda vyprázdní a zavře stream připojený k logovému souboru. Smyslem je aby jiná aplikace mohla v době, kdy logovací komponenta do souboru přímo nezapisuje volně s tímto souborem manipulovat, včetně smazání či přesunu (FileAppender na to zareaguje obnovením souboru na tom místě, kde ho očekával).

Metoda rotate je volána pokud se změní systémové datum (tedy o půlnoci každého dne) a způsobí to, že v adresáři s novým dnešním datem je vyroben nový logový soubor stejného jména jako včerejší, a další logování probíhá do něj.

Závěr

Vytvořená logovací komponenta byla odzkoušena v rámci testování aplikace Octopus 6 a po několika úpravách v kódu splnila všechna na začátku této práce stanovená kritéria. Architektura komponenty se tedy ukázala být funkční, a i její implementace, ač nefungovala hned napoprvé bezchybně, nakonec očekávání splnila. Samozřejmně požadavek na co nejkratší blokování aplikace (tedy na co největší časovou efektivitu) je tzv. optimalizačním kritériem jeho splnění tak nelze posoudit na základě jednoduchého testu a je možné že v logovací komponentě ještě dojde k optimalizacím s cílem zrychlit provádění logovacích požadavků. Dříve než bude moci být logovací komponenta používána v praxi je nutné vyvořit k ní ještě program určený ke čtení vytvořených logů. Ten ale již nespadá do rámce této práce, a proto tato práce zmiňuje jen fakt, že se s takovýmto programem při praktickém využití logovací komponenty počítá. V příloze je možné nalézt kompletní zdrojový kód modulu, s ohledem na přání firmy octopus newsroom s.r.o. byl tento kód anonymizován (tedy odkazy na třídy a balíčky aplikace octopus byly změněny na fiktivní balíčky a třídy fiktivní aplikace MyApp)

Seznam nalezených materiálů:

1 Projektování software:

ARLOW, J. NEUSTADT, I. *UML a Unifikovaný proces vývoje aplikací*
Brno : Computer Press Brno, 2008 ISBN: 80-7226-947-X

AMBLER, S.W. *Agile Unified Process v 1.1)*[online]
Ambysoft, 2005
Lze stáhnout na:

AMBLER, S.W. *A Manager's Introduction to The Rational Unified Process (RUP)*[online]
Ambysoft, 2005
http://www.ambysoft.com/unifiedprocess/rupIntroduction.html
(březen 2008)

SUBBU, N. S. *Taking SOA from Paper to Production* in:
System Integration 2006 14th international conference in Prague, Czech Republic, June 11-13, 2006 Praha, VŠE 2005 ISBN: 80-245-1050-2

BALDUINO, R. *Basic Unified Process: A Process for Small and Agile Projects* [online]
http://www.eclipse.org/proposals/beacon/Basic%20Unified%20Process.pdf
(březen 2008)

OBJECT MANAGEMENT GROUP *Introduction to OMG's Unified Modeling Language™* [online] (Updated July 2005 to reflect formal adoption of UML 2.0 Superstructure.)
http://www.omg.org/gettingstarted/what_is_uml.htm
(březen 2008)

KUČEROVÁ, Helena. *Projektování informačních systémů. Sylaby ke kurzu*. Praha : Vyšší odborná škola informačních služeb, 2007. 116 s.

ADAMEC, Stanislav, HORNÝ, Stanislav a ROSICKÝ, Antonín. *Projektování informačních systémů.* Praha : Vysoká škola ekonomická, 1997. 89 s. ISBN 80-7079-848-3

2 Java:

ECKEL, B. *Myslíme v jazyce Java*
Praha: Grada, 2000 ISBN: 80-247-0027-1

SHIRAZI, J. *Java vylaďování výkonu*
Praha: Grada, 2003 ISBN: 80-247-0752-7

PECINOVSKÝ, R. Myslíme objektově v jazyku Java 5.0

Praha: Grada, 2004 ISBN: 80-247-0941-4

LEA, Doug. *Concurrent Programming in Java*
State University of New York at Oswego, jaro 2002

KEOGH, James. *JAVA bez předchozách znalostí*
Brno: CP Books, a.s. ISBN: 2005 80-215-0839-2

KISZKA, Bogdan. *Programování v jazyce Java*
Brno: Computer press Brno, 2003 ISBN 80-7226-989-5

SUN MICROSYSTEMS *Java API* [JavaDoc]
Aplicatioon Programming Interface jazyka Java od tvůrce jazyka, společnosti Sun Microsystems
http://java.sun.com/reference/api/index.html
(březen 2008)

AUSTIN, Calvin. *An Introduction to Java Stack Traces*
on *SunDeveloperNetwork*, 1998 [online]
http://java.sun.com/developer/technicalArticles/Programming/Stacktrace/

3 Předchozí řešení logování v Javě:

Gülcü, C. *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging*
QOS.ch 2003 ISBN: 978-2970036906

Chauhan, M. Logging *with log4j—An Efficient Way to Log Java Applications* on *Developer.com* [online]
http://www.developer.com/open/article.php/3097221
(březen 2008)

APACHE *Log4j 1.2.15 API* [JavaDoc]
Aplicatioon Programming Interface od tvůrce balíčku org.apache.log4J, neziskové organizace Apache Software Foundation
http://logging.apache.org/log4j/1.2/apidocs/index.html
(březen 2008)

SUN MICROSYSTEMS *Java.utils.logging package API* [JavaDoc]
Aplicatioon Programming Interface od tvůrce balíčku java.utils.logging, společnosti Sun Microsystems
http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/package-summary.html
(březen 2008)

jLo documentation [webová stránka]

Online dokumentace k projektu jLo, logovacího frameworku nezávislého na Log4J i java.utils.logging
http://jlo.jzonic.org/TheToc.html
(březen 2008)

JTraceDump popis, dokumentace a zdrojový kód [webová stránka]
http://jtracedump.sourceforge.net/
(březen 2008)

Appendix A: Zdrojové kódy

Interfacy

I) Appender
II) LogTopic
III) Loggable

Třídy

IV) Logger
V) LogEngine (s vnitřní třídou LogRequest)
VI) Log
VII) FileAppender
VIII) ObjectToXMLConverter

I Appender

```
package MyApp.log;

/**
 * Interface describing a log appender;
 * Developers wishing to write their own Octopus Log Appender need to implement this interface
 *
 */
public interface Appender {  //must have a no-argument constructor
    /**
     * if its relevant to the given appenders data storage, set the storage name to the given string
     * @param filename
     */
    public void setFilename(String filename);

    /**
     * Process a LogRequest and append the result to this appenders storage (ex. file, database table,
output stream)
     * @param request - the request to be processed
     * @param attribute - a suffix to be added to the storage name; appenders can disregad it if its not
applicable to their storage type
     * @return true if the request was sucessfully processed
     */
    public boolean append(LogEngine.LogRequest request, String attribute); // append a pre-prepared
string representing a single log entry into a file containing the attribute as part of its filename; return
true if sucessfull, false otherwise;

    /**
     * Called by logger thread whenever it empties its request queue and gets a chance to rest;
     * the appender may use this call to temporarily release resources to avoid blocking them while
waiting for next append call;
     */
    public void rest();


    /**
     * Tells the logger whether this appender is a primary one. A primary appender is one that actually
creates logs that can be later read, for example in files or database,
     * while secondary appender is for example one writing logger output into console. Log request is
considered processed if
     * at least one primary appender managed to process it.
     *
     * @return
     */
    public boolean isPrimary();


    public void shutdown();

        }
```

II LogTopic

```
package MyApp.log;

public interface LogTopic {

    public Logger.Params getParams();
}
```

III Loggable

```
package octopus.utils.log;

import java.util.List;
import org.dom4j.Element;

/**
 * Interface defining its members as capable of returning their representation in XML format for
purposes of logging
 *
 */
public interface Loggable {

    /**
     *returns a XML representation of the object, with its root element being the given paramName
     *the "stack" parameter serves to avoid loops - if you need to call another Loggable.toLogXML
     *or ObjectToXMLConvertor.convert within the code pass it the stack list with your current object
added to it
     **/
    public Element toLogXML(String paramName, List<Object> stack);

}
```

IV Logger

```
package MyApp.log;

import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Vector;
import MyApp.Properties;

/**
 * The class serving as an interface to the rest of the log package; Other packages should call methods
of this class only
 *
 *
 */
public class Logger {


   public enum LogLevel {
      FATAL, ERROR, WARN, INFO, TRACE, DEBUG;
   }


 //---------------------------------TOPICS DECLARATION---------------------------------------------

   public static enum Logging implements LogTopic {
      LOGGING;
      public Params getParams() {return new Params("logging");}
   }
   public static enum General implements LogTopic {
      NO_TOPIC {public Params getParams() {return new Params("noTopic");}},
      public Params getParams() {return new Params("generalLog");}
   }
   public static enum Example1Gui implements LogTopic {
      SOME_GUI_STUFF {public Params getParams() {return new Params("noTopic");}},
      OTHER_GUI_STUFF {public Params getParams() {return new Params("desktop");}};
      public Params getParams() {return new Params("generalLog");}
   }

   public static enum Example2 LogTopic {
      SYSTEM_RULES,
      NOTIFICATIONS,
      WIDGETS,
      EDITING,
      PREVIEW;
      public Params getParams() {return new Params("wires");}
      public static enum Containers implements LogTopic {
         SECTION_GUI,
         CONTAINER_EDITING;
         public Params getParams() {return new Params("wires");}
      }
   }
```

```
//--------------------------------END OF TOPICS DECLARATION--------------------------------



        public static class Params {
        String filename = null;
        Logger.LogLevel defLvl = Logger.LogLevel.DEBUG;
        Class appender = FileAppender.class;
        String attribute = null;

        public Params() {

        }
        public Params(String filename) {
            this.filename = filename;
        }
        public Params(String filename, String attribute) {
            this.filename = filename;
            this.attribute = attribute;
        }
    }
    //...etc

    //-----------------------------------------------------------------------------------------------
    protected static Logger defaultLogger;
    protected LogTopic defTopic = General.NO_TOPIC;
    protected boolean isImmutable = false;
    protected Vector<Object> defParams = new Vector<Object>();
    protected static Map<LogTopic, Map<String, Logger>> loggerCache = new HashMap();



    //-----------------------------constructors/factory methods---------------------------------------

    /**
     * Creates a new Logging
     */
    public Logger() {

    }

    /**
     * Creates a new Logging with given default LogTopic
     */
    public Logger(LogTopic defTopic) {
        this.defTopic = defTopic;
    }


    /**
     * Returns a shared immutable instance of Logger; note that attempting to set default values for
Logging instance
     * obtained via getDefaultLogger results in an exception
     */
    public static synchronized Logger getDefaultLogger() {
        if (defaultLogger == null) defaultLogger = new Logger();
        defaultLogger.isImmutable = true;
```

```
        return defaultLogger;
    }

    /**
     * Returns a shared immutable instance of Logger with a given default topic and parameters; note
that attempting to set default values for Logging instance
     * obtained via getLoggerForTopic results in an exception
     */
    public static Logger getLoggerForTopic(LogTopic topic, Object... params) {
        Logger result = null;
        if (loggerCache.containsKey(topic)) {
            Map<String, Logger> subCache = loggerCache.get(topic);
            if (subCache.containsKey(paramsToString(params))) {
                result = subCache.get(paramsToString(params));
            } else {
                result = new Logger(topic);
                result.setDefaultParameters(params);
                subCache.put(paramsToString(params), result);
            }
        } else {
            result = new Logger(topic);
            result.setDefaultParameters(params);
            Map<String, Logger> map = new HashMap();
            map.put(paramsToString(params), result);
            loggerCache.put(topic, map);
        }
        result.isImmutable = true;
        return result;
    }

    //-----------------------------other static methods---------------------------------------

    public static void setLogDirectory(String directory) {
        LogEngine.setLogDirectory(directory);
    }

    public static void setLogLevel(LogLevel level) {
        LogEngine.setLogLevel(level, null);
    }

    public static void setLogLevel(LogLevel level, LogTopic topic) {
        LogEngine.setLogLevel(level, topic);
    }

    public static void addAppender(Appender appender) {
            LogEngine.addAppender(appender, null);
    }

    public static void addAppender(Appender appender, LogTopic forTopic) {
        List<LogTopic> list = new ArrayList<LogTopic>();
        list.add(forTopic);
        addAppender(appender, list);
    }

    public static void redirectSystemOut() {
        LogEngine.redirectSystemOut();
    }

    public static void redirectSystemErr() {
        LogEngine.redirectSystemErr();
```

```
    }


   /**
     * Add a new appender
     * @param appender - the appender to add
     * @param forTopics
     * topics whose log requests this appender should receive
     * null means add this appender as a 'generic appender' -> one that will get ALL the log requests
      */
    public static void addAppender(Appender appender, List<LogTopic> forTopics) {
        LogEngine.addAppender(appender, forTopics);
    }


    public static void removeAppender(Appender appender, LogTopic forTopic) {
        List<LogTopic> list = new ArrayList<LogTopic>();
        list.add(forTopic);
        removeAppender(appender, list);
    }

    public static void removeAppender(Appender appender, List<LogTopic> forTopics) {
        LogEngine.removeAppender(appender, forTopics);
    }


    //-------------------------------------public methods-------------------------------------------------

    public boolean willLog(LogLevel level, LogTopic topic) {
        return LogEngine.willLog(level, topic);
    }

    public void setDefParams(Object... params) throws LoggerException {
        setDefaultParameters(params);
    }

    public void setDefaultTopic(LogTopic topic) throws LoggerException  {
       if (isImmutable) {
           error(Logging.LOGGING, new LoggerException("default Logger instance is immutable"), "Trying
to set default topic on the shared default instance of Logger", "topic", topic);
           return;
       }
       this.defTopic = topic;
    }

    public Logger copy() {
        Logger result = new Logger();
        result.defParams = this.defParams;
        result.defTopic = this.defTopic;
        result.isImmutable = false;
        return result;
    }

    public void setImmutable() {
        isImmutable = true;
    }

    public boolean isImmutable() {
        return isImmutable;
```

```
    }

    //-------------------------------------public proxy methods for different LogLevels----------

    public void fatal(String message, Object... params) {log(LogLevel.FATAL, message, params);}
    public void fatal(Throwable th, String message, Object... params) {log(LogLevel.FATAL, th,
message, params);}
    public void fatal(LogTopic topic, String message, Object... params) {log(LogLevel.FATAL, topic,
message, params);}
    public void fatal(LogTopic topic, Throwable th, String message, Object... params)
{log(LogLevel.FATAL, topic, th, message, params);}

    public void error(String message, Object... params) {log(LogLevel.ERROR, message, params);}
    public void error(Throwable th, String message, Object... params) {log(LogLevel.ERROR, th,
message, params);}
    public void error(LogTopic topic, String message, Object... params) {log(LogLevel.ERROR, topic,
message, params);}
    public void error(LogTopic topic, Throwable th, String message, Object... params)
{log(LogLevel.ERROR, topic, th, message, params);}

    public void warn(String message, Object... params) {log(LogLevel.WARN, message, params);}
    public void warn(Throwable th, String message, Object... params) {log(LogLevel.WARN, th,
message, params);}
    public void warn(LogTopic topic, String message, Object... params) {log(LogLevel.WARN, topic,
message, params);}
    public void warn(LogTopic topic, Throwable th, String message, Object... params)
{log(LogLevel.WARN, topic, th, message, params);}

    public void info(String message, Object... params) {log(LogLevel.INFO, message, params);}
    public void info(Throwable th, String message, Object... params) {log(LogLevel.INFO, th,  message,
params);}
    public void info(LogTopic topic, String message, Object... params) {log(LogLevel.INFO, topic,
message, params);}
    public void info(LogTopic topic, Throwable th, String message, Object... params)
{log(LogLevel.INFO, topic, th, message, params);}

    public void trace(String message, Object... params) {log(LogLevel.TRACE, message, params);}
    public void trace(Throwable th, String message, Object... params) {log(LogLevel.TRACE, th,
message, params);}
    public void trace(LogTopic topic, String message, Object... params) {log(LogLevel.TRACE, topic,
message, params);}
    public void trace(LogTopic topic, Throwable th, String message, Object... params)
{log(LogLevel.TRACE, topic, th, message, params);}

    public void debug(String message, Object... params) {log(LogLevel.DEBUG, message, params);}
    public void debug(Throwable th, String message, Object... params) {log(LogLevel.DEBUG, th,
message, params);}
    public void debug(LogTopic topic, String message, Object... params) {log(LogLevel.DEBUG, topic,
message, params);}
    public void debug(LogTopic topic, Throwable th, String message, Object... params)
{log(LogLevel.DEBUG, topic, th, message, params);}


    //-------------------------------------------------actual log methods (private)----------------------------------

    private void log(LogLevel logLvl, String message, Object[] params) {
        log(logLvl, defTopic, null, message, params);
    }
```

```
    private void log(LogLevel logLvl, Throwable th, String message, Object[] params) {
        log(logLvl, defTopic, th, message, params);
    }

    private void log(LogLevel logLvl, LogTopic topic, String message, Object[] params) {
        log(logLvl, topic, null, message, params);
    }

    private void log(LogLevel logLvl, LogTopic topic, Throwable throwable, String message, Object[]
params) {
        if (willLog(logLvl, topic)) {
         if (!checkParamsFormat(params)) {
             error(Logging.LOGGING, new LoggerException("invalid parameters format"), "Logging method
called with wrong parameter format", "params", params);
             return;
         }
        if (params.length == 1) {
            if (params[0] instanceof Properties) {
                Properties p = (Properties)params[0];
                List newParams = new ArrayList();
                for (String name : p.names()) {
                    newParams.add(name);
                    newParams.add(p.get(name));
                }
                params = newParams.toArray();
            }
            if (params[0] instanceof Map) {
                Map m = (Map)params[0];
                List newParams = new ArrayList();
                for (Object key : m.keySet()) {
                    newParams.add(key.toString());
                    newParams.add(m.get(key));
                }
                params = newParams.toArray();
            }
        }
        Vector<Object> currentParams = paramsToVector(params);
        Vector<Object> allParams = joinParams(currentParams, defParams);
        LogEngine.log(Thread.currentThread().getId(), logLvl, topic, throwable, message, allParams);
        }
    }

    //-----------------------------------other private methods----------------------------------------

    protected void setDefaultParameters(Object[] params) throws LoggerException {
        if (isImmutable) throw new LoggerException("Cant set attributes on the shared default instance of
Logger");
        if (!checkParamsFormat(params)) throw new LoggerException("Incorrect format of parameters;
has to be name1, value1, name2, value2, etc");
        this.defParams = paramsToVector(params);
    }



    protected boolean checkParamsFormat(Object[] params) {
        if (params.length == 0) return true;
        if (params.length == 1 && (params[0] instanceof Properties || params[0] instanceof Map)) return
true;
        boolean check = true;
        if (params.length % 2 != 0) check = false;
```

```
        for (int i = 1; i < params.length; i += 2) {
            if (!(params[i - 1] instanceof String)) check = false;
        }
        return check;
    }

    protected static Vector<Object> paramsToVector(Object[] params) {
        Vector<Object> result = new Vector<Object>();
        for (Object o : params) {
            result.add(o);
        }
        return result;
    }

    protected static String paramsToString(Object[] params) {
        if (params == null) return "null";
        StringBuffer buffer = new StringBuffer();
        for (Object o : params) {
            if (o == null) buffer.append("null");
            else buffer.append(o.toString());
        }
        return buffer.toString();
    }

    protected Vector<Object> joinParams(Vector<Object> params1, Vector<Object> params2) {
        Vector<Object> result = new Vector<Object>();
        if (params1 == null) result.addAll(params2);
        else if (params2 == null) result.addAll(params1);
        else {
            result.addAll(params1);
            result.addAll(params2);
        }
        return result;
    }



    //------------------------------------ classes --------------------------------------------


    /**
     *
     */
    private class loggerRule {
        String message;
        long topic;
        int lvl;
        Date date;
        String sourceClass;
        String sourceMethod;
        String sourceLine;
        Log log;
    }


}
```

```
package MyApp.log;

import java.util.List;
import org.dom4j.Element;

/**
 * Interface defining its members as capable of returning their representation in XML format for
purposes of logging
 *
 */
public interface Loggable {

    /**
     *returns a XML representation of the object, with its root element being the given paramName
     *the "stack" parameter serves to avoid loops - if you need to call another Loggable.toLogXML
     *or ObjectToXMLConvertor.convert within the code pass it the stack list with your current object
     *added to it
     **/
    public Element toLogXML(String paramName, List<Object> stack);

}
```

V LogEngine

```
package MyApp.log;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileFilter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FilenameFilter;
import java.io.IOException;
import java.io.PrintStream;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.TimeZone;
import java.util.Timer;
import java.util.TimerTask;
import java.util.Vector;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;
import MyApp.Utils;


/**
 *
 *
 */
public class LogEngine {


    static final long MAX_SHUTDOWN_WAIT = 10000; //how long the shutdown hook waits (max) for all logger threads to finish their work; low setting can prevent last few messages from being logged, high setting can cause long wait times if there is a problem preventing the logging (like inacessible log file)
    static final int BLOCK_LOGGING_THREAD_FOR_MAX = 500;
    static final Logger.LogLevel BLOCK_LOGGING_THREAD_LVL = Logger.LogLevel.DEBUG; //set to make logger block whenever queues reach a level where they would no longer log a given request;
    //setting to null will means never block, setting to debug means always block when needed, setting to for example warn means block only for warn, error or fatal

    long queueSize_greenAlert = 600000;
    long queueSize_yellowAlert = 800000;
    long queueSize_redAlert = 1000000;

    static final File defLogPath = new File(".");
    private static LogEngine inst;
    Map<LogTopic, Log> topicsToLogs;
    Map<String, Log> logfilesToLogs;
    Map<LogTopic, List<Appender>> topicSpecificAppenders;
    Map<Log, List<Appender>> logSpecificAppenders;
    Map<LogTopic, Logger.LogLevel> logLevels;
```

```
    Logger.LogLevel generalDefLvl;
    Calendar cal;
    List<Appender> genericAppenders;
    long serialBase;
    Long serialOffset;//Long so I can use toString();
    String serialBaseStr;
    long charactersInQueue = 0;
    static boolean ready = false;
    File logPath;
    AlertStatus currentAlertStatus = AlertStatus.OK;
    Object appendersLock;
    static PrintStream consoleOutStream = System.out;
    static PrintStream consoleErrStream = System.err;
    static Timer logFilePackager;
    static List<Log> awaitingShutdown;

    static int packAfter = 3;
    static int deleteAfter = 14;

    enum AlertStatus {

        OK, GREEN, YELLOW, RED;
    }



    /** Creates a new instance of LogEngine, starts a timer taking care of packaging logfiles */
    private LogEngine() {
        topicsToLogs = new HashMap<LogTopic, Log>();
        logfilesToLogs = new HashMap<String, Log>();
        topicSpecificAppenders = new HashMap<LogTopic, List<Appender>>();
        logSpecificAppenders = new HashMap<Log, List<Appender>>();
        logLevels = new HashMap<LogTopic, Logger.LogLevel>();
        generalDefLvl = Logger.LogLevel.DEBUG;
        genericAppenders = new ArrayList<Appender>();
        serialBase = 0L;
        serialOffset = 0L;

        appendersLock = new Object();
        logPath = defLogPath;
        cal = Calendar.getInstance(TimeZone.getTimeZone("UTC"));
        //Shutdown Hook
        Thread hook = new Thread() {
            public void run() {
                getInst().insertIntoLog(Thread.currentThread().getId(), Logger.LogLevel.INFO,
Logger.Desktop.MAIN_APP_FRAME, null, "Application shutting down", new ArrayList<Object>());
                shutdown();
            }
        };
        Runtime.getRuntime().addShutdownHook(hook);
        setAlertLevels(10, 12, 15);
    }


    //--------------------  Static  ----------------------------------------------------
    /**
     *Prepares the logging system for usage or re-creates it if log dir was changed
     **/
    @SuppressWarnings("static-access")
```

```
public static void createLogs() {
    if (inst == null) {
        inst = new LogEngine();
    } else {
        inst.topicsToLogs = new HashMap<LogTopic, Log>();
        inst.logfilesToLogs = new HashMap<String, Log>();
    }
    inst.treeWalker(Logger.class, new Vector<Logger.Params>());

    logFilePackager = new Timer();
    TimerTask packageFiles = new TimerTask() {

        public void run() {
            packOldFiles();
        }
    };
    logFilePackager.scheduleAtFixedRate(packageFiles, 0L, 3600000L);

    ready = true;
}


public static void setLogPackingDelay(int packLogsAfter, int deleteLogsAfter) {
    packAfter = packLogsAfter;
    deleteAfter = deleteLogsAfter;
}


/**
 * Find out whether a log request with given Log Level and topic
 * @param logLvl
 * @param topic
 * @return
 */
protected static boolean willLog(Logger.LogLevel logLvl, LogTopic topic) {
    return logLvl.compareTo(getLogLevel(topic)) <= 0;
}

protected static void setLogDirectory(String directory) {
    if (directory == null) {
        getInst().logPath = defLogPath;
    } else {
        getInst().logPath = new File(directory);
        createLogs();
    }
}

protected static File getLogDirectory() {
    return getInst().logPath;
}

/**
 * Get all Appenders for given topic
 * @param LogTopic topic
 * @param Log log
 * @return List<Appender>
 */
protected static List<Appender> getAppendersForTopic(LogTopic topic, Log log) {
    List<LogTopic> topics = new ArrayList<LogTopic>();
    topics.add(topic);
```

```
    return getAppendersForTopics(topics, log);
}

/**
 * Get all generic appenders
 * @return List<Appender>
 */
protected static List<Appender> getGenericAppenders() {
    List<Appender> result = new ArrayList<Appender>();
        for (Appender appender : getInst().genericAppenders) {
            if (!result.contains(appender)) {
                result.add(appender);
            }
        }
    return result;
}

protected static Logger.LogLevel getLogLevel(LogTopic topic) {
    if (topic != null && getInst().logLevels.get(topic) != null) {
        return getInst().logLevels.get(topic);
    } else {
        return getInst().generalDefLvl;
    }
}

protected static synchronized void redirectSystemOut() {
        if (consoleOutStream.equals(System.out)) {
            addAppender(new ConsoleAppender(consoleOutStream), null);
            System.setOut(new PrintStream(new LoggerStream(Logger.Desktop.SYSTEM_OUT), true));
        }
}
protected static synchronized void redirectSystemErr() {
        if (consoleErrStream.equals(System.err))
            System.setErr(new PrintStream(new LoggerStream(Logger.Desktop.SYSTEM_ERR), true));
}

protected static void setLogLevel(Logger.LogLevel logLevel, LogTopic topic) {
    if (topic != null) {
        getInst().logLevels.put(topic, logLevel);
    } else {
        getInst().generalDefLvl = logLevel;
    }
}

/**
 *  Get all appenders connected to the given Log instance
 * @param Log log
 * @return List<Appender>
 */
protected static List<Appender> getAppendersForLog(Log log) {
        List<Appender> result = getGenericAppenders();
        try {
            for (Appender appender : getInst().logSpecificAppenders.get(log)) {
                if (!result.contains(appender)) {
                    result.add(appender);
                }
            }
        }
        catch (NullPointerException ex){
```

```
            //other way to do it would be: if (getInst().logSpecificAppenders.contains(log)) -- but that
wouldnt be thread-safe
        }
        return result;
    }

    /**
     * Get all appenders connected to the given topics and to the given Log instance
     * @param topics List<LogTopic> - can be null - in that case only generic appenders and those
connnmected to given log are be returned
     * @param log Log - can be null - in that case only generic appenders and those connnmected to
given topics are be returned
     * @return
     */
    protected static List<Appender> getAppendersForTopics(List<LogTopic> topics, Log log) {
        synchronized (getInst().appendersLock) {
            List<Appender> result = getAppendersForLog(log);
            for (LogTopic topic : topics) {
                if (getInst().topicSpecificAppenders.containsKey(topic)) {
                    for (Appender appender : getInst().topicSpecificAppenders.get(topic)) {
                        if (!result.contains(appender)) {
                            result.add(appender);
                        }
                    }
                }
            }
            return result;
        }
    }

    protected static void addAppenderForLog(Appender appender, Log log) {
        if (!getInst().logSpecificAppenders.containsKey(log)) {
            getInst().logSpecificAppenders.put(log, new ArrayList());
        }
        if (!getInst().logSpecificAppenders.get(log).contains(appender)) {
            getInst().logSpecificAppenders.get(log).add(appender);
        }
    }

    protected static void addAppender(Appender appender, List<LogTopic> topics) {
        synchronized (getInst().appendersLock) {
            if (topics == null && !getInst().genericAppenders.contains(appender)) {
                getInst().genericAppenders.add(appender);
            }
            if (topics != null) {
                for (LogTopic topic : topics) {
                    if (!getInst().topicSpecificAppenders.get(topic).contains(appender)) {
                        getInst().topicSpecificAppenders.get(topic).add(appender);
                    }
                }
            }
        }
    }

    protected static void removeAppender(Appender appender, List<LogTopic> topics) {
        if (topics == null && getInst().genericAppenders.contains(appender)) {
            Iterator i = getInst().genericAppenders.iterator();
            while (i.hasNext()) {
                if (i.next().equals(appender)) {
                    i.remove();
```

```
            }
        }
    }
    if (topics != null) {
        for (LogTopic topic : topics) {
            if (getInst().topicSpecificAppenders.get(topic).contains(appender)) {
                Iterator i = getInst().topicSpecificAppenders.get(topic).iterator();
                while (i.hasNext()) {
                    if (i.next().equals(appender)) {
                        i.remove();
                    }
                }
            }
        }
    }
}

/**
 * sets logger queues sum size limits as percent of total heap space
 * the relation is only approximate and should not be set too high (max tens of percent)
 * @param greenPercent
 * @param yellowPercent
 * @param redPercent
 */
protected void setAlertLevels(int greenPercent, int yellowPercent, int redPercent) {
    if (greenPercent < 100 &&  greenPercent > 0 && yellowPercent < 100 &&  yellowPercent > 0 &&
redPercent < 100 &&  redPercent > 0) {
        long maxMem = Runtime.getRuntime().maxMemory();
        if (maxMem == Long.MAX_VALUE)
            maxMem = 209715200; //if unlimited, set to 200MB
        queueSize_greenAlert = maxMem * greenPercent / 100 / 2;
        queueSize_yellowAlert = maxMem * yellowPercent / 100 / 2;
        queueSize_redAlert = maxMem * redPercent / 100 / 2;
    // the last division by 2 is a rough estimate of string size - 2 bytes for every character
    //- since queue lengths are an esitmate of the sum of lengths of strings inside all logger queues at
a given moment
    }
    else throw new IllegalArgumentException("all parameters must be between 0 and 100");
}

protected static void log(long threadId, Logger.LogLevel level, LogTopic topic, Throwable throwable,
String message, Vector<Object> params) {
      getInst().insertIntoLog(threadId, level, topic, throwable, message, params);
}

/**
 * Create a new log request and put it into the queue belonging to the log that contains this requests
topic
 * @param threadId
 * @param Logger.LogLevel level
 * @param LogTopic topic
 * @param Throwable throwable
 * @param String message
 * @param List<Object>params
 */
private synchronized void insertIntoLog(long threadId, Logger.LogLevel level, LogTopic topic,
Throwable throwable, String message, List<Object> params) {
    if (level.compareTo(getLogLevel(topic)) <= 0) {
        LogRequest line;
```

```
            line = new LogRequest(getNextSerial(), new Date(), threadId, level, topic, throwable, message,
params);
            if (alertFilter(level, line.getSize())) {
                topicsToLogs.get(topic).addToLog(line);
            } else {
                if (BLOCK_LOGGING_THREAD_LVL != null &&
BLOCK_LOGGING_THREAD_LVL.compareTo(level) <= 0) {
                    line.blockingStarted();
                    int blockedFor = 0;
                    while (!alertFilter(level, line.getSize()) && blockedFor >
BLOCK_LOGGING_THREAD_FOR_MAX) {
                        try {
                            Thread.currentThread().sleep(50);
                            blockedFor += 50;
                        } catch (InterruptedException ex) {
                        }
                    }
                    line.blockingFinished();
                    topicsToLogs.get(topic).addToLog(line);
                }
            }
        }
    }

    /**
     * Called by the shutdown hook, tell all Logs to shutdown, and wait until all of them confirm
     * they have shut down, or until a fixed time limit expires
     */
    protected void shutdown() {
        consoleOutStream.println("shutdown hook activated");
        logFilePackager.cancel();
        awaitingShutdown = new ArrayList();
        Long waitingTime = 0L;
        for (Log log : logfilesToLogs.values()) {
            awaitingShutdown.add(log);
            log.shutdown();
        }
        while (awaitingShutdown.size() > 0 && waitingTime < MAX_SHUTDOWN_WAIT) {
        try {
            Thread.currentThread().sleep(20);
            waitingTime += 20;
        } catch (InterruptedException ex) {
        }
        }
    }

    protected static void reportShutdown(Log log) {
        if (awaitingShutdown != null && awaitingShutdown.contains(log)) awaitingShutdown.remove(log);
    }

    /**
     * returns an instance of LogEngine if it exists, otherwise sets one up
     * (it sets the log directory according to property MyApp.log.path,
     * if that doesnt exist it tries user home, if that is for some reason undefined
     * it will set current working dir)
     * @return
     */
    protected static synchronized LogEngine getInst() {
        if (inst == null) {
            inst = new LogEngine();
```

```
            if (System.getProperties().getProperty("MyApp.log.path") != null) {
                File logDir = new File(System.getProperties().getProperty("MyApp.log.path"));
                inst.logPath = new File(System.getProperties().getProperty("MyApp.log.path"));
            } else if (System.getProperty("user.home") != null) {
                inst.logPath = new File(System.getProperty("user.home") + File.separator + ".octopus" +
File.separator + "logs" + File.separator);
            } else {
                inst.logPath = new File(".");
            }
            inst.createLogs();
        }
        return inst;
    }

    /**
     *Return all topics asociated with a given filename or null if the LogEngine wasnt initialized yet
     **/
    protected static List<LogTopic> getTopicsForFilename(String filename) {
        if (ready) {
            List<LogTopic> result = new ArrayList<LogTopic>();
            Log log = getInst().logfilesToLogs.get(filename);
            for (LogTopic topic : getInst().topicsToLogs.keySet()) {
                if (getInst().topicsToLogs.get(topic) == log) {
                    result.add(topic);
                }
            }
            return result;
        }
        return null;
    }


    protected String timeToSerialBase(Long timestamp) {
            cal.setTimeInMillis(timestamp);
            StringBuffer buff = new StringBuffer();
            buff.append(cal.get(cal.YEAR));
            buff.append(Utils.leadZero(2, cal.get(cal.MONTH) + 1));
            buff.append(Utils.leadZero(2, cal.get(cal.DAY_OF_MONTH)));
            buff.append(Utils.leadZero(2, cal.get(cal.HOUR_OF_DAY)));
            buff.append(Utils.leadZero(2, cal.get(cal.MINUTE)));
            buff.append(Utils.leadZero(2, cal.get(cal.SECOND)));
            buff.append(Utils.leadZero(3, cal.get(cal.MILLISECOND)));
            return buff.toString();
    }

    protected AlertStatus modifyQueueLengthAndCheckStatus(int delta) {
        charactersInQueue += (long) delta;
        AlertStatus newAlertStatus = AlertStatus.OK;
        if (getInst().charactersInQueue > queueSize_redAlert) {
            newAlertStatus = AlertStatus.RED;
        } else if (getInst().charactersInQueue > queueSize_yellowAlert) {
            newAlertStatus = AlertStatus.YELLOW;
        } else if (getInst().charactersInQueue > queueSize_greenAlert) {
            newAlertStatus = AlertStatus.GREEN;
        }
        if (getInst().currentAlertStatus.compareTo(newAlertStatus) != 0) {
            alertStatusChange(newAlertStatus, currentAlertStatus);
        }
        currentAlertStatus = newAlertStatus;
```

```
        return newAlertStatus;
    }


    //---------------------------------- private ----------------------------------------------


    /**
     * Create a new unique serial number based on current time and a suffix of 3 numbers making any
     * 2 serials created at the same milisecond unique;
     */
    private String getNextSerial() {
        Date now = new Date();
        if (now.getTime() == serialBase) {
            serialOffset++;
        } else {
            serialBase = now.getTime();
            serialOffset = 0L;
            serialBaseStr = timeToSerialBase(now.getTime());
        }
        return serialBaseStr + Utils.leadZero(3, serialOffset.toString());
    }


    private boolean alertFilter(Logger.LogLevel level, int requestLength) {
        AlertStatus status = modifyQueueLengthAndCheckStatus(0);
        boolean result =
                ((status == LogEngine.AlertStatus.OK) ||
                (status == LogEngine.AlertStatus.GREEN && level.compareTo(Logger.LogLevel.TRACE) <=
0) ||
                (status == LogEngine.AlertStatus.YELLOW && level.compareTo(Logger.LogLevel.WARN)
<= 0) ||
                (status == LogEngine.AlertStatus.RED && level.compareTo(Logger.LogLevel.FATAL) ==
0));
        if (result && requestLength > 0) {
            charactersInQueue += requestLength;
        }
        return result;
    }


    private void alertStatusChange(AlertStatus status, AlertStatus oldStatus) {
        Logger.LogLevel level = Logger.LogLevel.ERROR;
        String message = "";
        if (status.compareTo(oldStatus) > 0) {
            if (status == AlertStatus.GREEN) {
                message = "LOGGER QUEUES ARE GETTING TOO BIG; STOPPED LOGGING ALL
DEBUG AND TRACE";
            }
            if (status == AlertStatus.YELLOW) {
                message = "LOGGER QUEUES SIZE IS GETTING CRITICAL; STOPPED LOGGING ALL
EXCEPT FATAL";
            }
            if (status == AlertStatus.RED) {
                message = "LOGGER QUEUES SIZE CRITICAL; DUMPING ALL QEUES";
                level = Logger.LogLevel.FATAL;
                dumpQueues();
            }
```

```
        } else {
            level = Logger.LogLevel.ERROR;
            if (status == AlertStatus.OK) {
                message = "LOGGER QUEUES ARE BACK TO OK";
            }
            if (status == AlertStatus.GREEN) {
                message = "LOGGER QUEUES SHRINKING, ALLOWING ALL LOGGING EXCEPT
DEBUG AND TRACE";
            }
        }
        //this is a direct Log call - use of this technique is strongly discouraged! (this is a special case to
ensure correct placement of these messages in the logging queues)
        Vector params = new Vector<Object>();
        params.add("maxMemory");
        params.add(Runtime.getRuntime().maxMemory());
        params.add("freeMemory");
        params.add(Runtime.getRuntime().freeMemory());
        topicsToLogs.get(Logger.Logging.LOGGING).addToLog(new LogRequest(getNextSerial(), new
Date(), Thread.currentThread().getId(), level, Logger.Logging.LOGGING, null, message, params));
    }

    private void dumpQueues() {
        for (Log log : logfilesToLogs.values()) {
            log.dump();
        }
    }
    //------------------  Private  ------------------------------------------------
    /**
     * Walk trough the LogTopic tree inside the Logger class using recursion, creating the coresponding
logs
     * @param enumClass
     * @param history
     */
    private void treeWalker(Class enumClass, Vector<Logger.Params> history) {
        //    try {
        Vector<Logger.Params> newHistory = history;
        if (enumClass.isEnum() && LogTopic.class.isAssignableFrom(enumClass)) {
            for (Object topicObj : enumClass.getEnumConstants()) {
                try {
                    LogTopic topic = (LogTopic) topicObj;
                    initLog(topic, history);
                } catch (ClassCastException e) {
                    consoleErrStream.println("Class cast exception in LogEngine treewalker method when
processing class" + enumClass.getCanonicalName());
                }
            }
        }
        for (Class subClass : enumClass.getDeclaredClasses()) {
            treeWalker(subClass, newHistory);
        }
    //    } catch (IllegalAccessException ex) {
    //        ex.printStackTrace();
    //    } catch (InstantiationException ex) {
    //        ex.printStackTrace();
    //    }
    }

    /**
     *If there is a subdirectory under the corrent log directory that corresponds to a date 3 days ago
     * pack all .log files found there into separate .zip files
```

```
     */
    private static void packOldFiles() {
        final SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
        final Date packFrom = new Date(new Date().getTime() - 86400000L * packAfter);
        final Date deleteFrom = new Date(new Date().getTime() - 86400000L * deleteAfter);
        File dir = LogEngine.getLogDirectory();
        FilenameFilter logFileFilter = new FilenameFilter() {

            public boolean accept(File dir, String name) {
                return (name.endsWith(".log"));
            }
        };
        FileFilter logDirFilter = new FileFilter() {

            public boolean accept(File pathname) {
                try {
                    if (pathname.isDirectory() && format.parse(pathname.getName()).getTime() <
packFrom.getTime()) {
                        return true;
                    }
                } catch (ParseException ex) {
                }
                return false;
            }
        };

        if (dir.isDirectory()) {
            for (File subDir : dir.listFiles(logDirFilter)) {
                {
                    try {
                        if (format.parse(subDir.getName()).getTime() < deleteFrom.getTime()) {
                            deleteDir(subDir);
                        } else {
                            if (format.parse(subDir.getName()).getTime() < packFrom.getTime()) {
                            for (File file : subDir.listFiles(logFileFilter)) {
                                byte[] buf = new byte[1024];
                                String outFilename = file.getAbsolutePath().replace(".log", ".zip");
                                try {
                                    ZipOutputStream out = new ZipOutputStream(new
FileOutputStream(outFilename));
                                    FileInputStream in = new FileInputStream(file);
                                    try {
                                        out.putNextEntry(new ZipEntry(file.getName()));
                                        int len;
                                        while ((len = in.read(buf)) > 0) {
                                            out.write(buf, 0, len);
                                        }
                                    } catch (IOException e) {
                                        Logger.getDefaultLogger().info(e, "Could not package old logfile", "logfile",
file.getName());
                                    } finally {
                                        try {
                                            in.close();
                                            out.closeEntry();
                                            out.close();
                                        } catch (IOException ex) {
                                        }
                                        file.delete();
                                    }
                                } catch (FileNotFoundException ex) {
```

```
                    }
                }
                }
            }
        } catch (ParseException ex) {
        //cannot happen
        } finally {

        }
    }
  }
}


private static void deleteDir(File dir) {
    for (File f : dir.listFiles()) {
        if (f.isDirectory())
            deleteDir(f);
        else
            f.delete();
    }
    dir.delete();
}

/**
 * Create a new Log, set it up and place it into all of LogEngines maps
 * @param topic
 * @param history
 */
private void initLog(LogTopic topic, Vector<Logger.Params> history) {
    Logger.Params params = null;
    if (topic.getParams().filename != null) {
        params = topic.getParams();
    } else {
        for (int i = 0; i < history.size(); i++) {
            if (history.get(i).filename != null) {
                params = history.get(i);
            }
        }
    }
    if (params == null) {
        params = new Logger.Params("generalLog");
    }

    if (logfilesToLogs.containsKey(params.filename)) {
        topicsToLogs.put(topic, logfilesToLogs.get(params.filename));
    // System.out.println("topic" + topic.toString() + " linked to log " + params.filename);
    } else {
        Log log = new Log(params);
        topicsToLogs.put(topic, log);
        logfilesToLogs.put(params.filename, log);
    // System.out.println("topic" + topic.toString() + " linked to log " + params.filename);
    }



}
```

```
    private static class LoggerStream extends ByteArrayOutputStream {
    Logger log;

        public LoggerStream(LogTopic topic) {
            super();
            log = Logger.getLoggerForTopic(topic);
        }

        public void flush() throws IOException {
            try {
                super.flush();
                String line = this.toString();
                super.reset();
                if (line.length() == 0 || line.trim().length() == 0) {
                    return;
                }
                log.info(line);
                super.reset();
            } catch (Exception e) {
                        log.error("Standard out flush failed", e);
            }
        }
    }

   //------------------------------ public ---------------------------------



    /**
     * A data class that lives in the log request queues, storing all the information about the log
     * request made by the Logger class; Is handed over to Appenders for processing
     */
    public class LogRequest {

        private String serial;
        private Date date;
        private Long threadId;
        private Logger.LogLevel logLvl;
        private LogTopic topic;
        private Throwable throwable;
        private String message;
        private Logger.LogLevel level;
        private Map<String, String> paramsMap;
        private SimpleDateFormat formater = new SimpleDateFormat("HH:mm:ss");
        private Calendar cal;
        private String attributeValue;
        private Date blockingStart;
        private Long blockingDur = null;


        public LogRequest(String serial, Date date, Long threadId, Logger.LogLevel logLvl, LogTopic
topic, Throwable throwable, String message, List<Object> params) {
            level = logLvl;
            this.serial = serial;
            this.date = date;
            this.threadId = threadId;
            this.logLvl = logLvl;
            this.topic = topic;
            this.message = message;
```

```
        this.throwable = throwable;
        paramsMap = new HashMap<String, String>();
        cal = Calendar.getInstance(TimeZone.getTimeZone("UTC"));
        attributeValue = null;
        String attribute = topic.getParams().attribute;
        if (params.size() > 1) {
            for (int i = 1; i < params.size(); i += 2) {
                if (params.get(i - 1) != null) {
                    String paramXML = (ObjectToXMLConvertor.convert(params.get(i),
(String)params.get(i - 1)));
                    paramsMap.put(params.get(i - 1).toString(), paramXML);
                }
                if (params.get(i - 1) != null && params.get(i) != null && params.get(i - 1).equals(attribute))
{
                    attributeValue = params.get(i).toString();
                }
            }
        }
    }

     /**
     * get the standard entry inserted into logfiles for this LogRequest; appenders may choose to
disregard this method
     * and process the LogRequests fields to generate an entry in suitable format; however, use this
method where possible to avoid confusion
     * @return String - the standard log file entry
     */
    public String getLine() {
        StringBuffer line = new StringBuffer("");
        line.append(escape(serial));
        line.append("|");
        line.append(escape(threadId.toString()));
        line.append("|");
        line.append(escape(level.toString()));
        line.append("|");
        String debug = topic.getClass().getName().replace("MyApp.log.Logger$", "") + "." +
topic.toString();
        debug = escape(debug);
        //line.append(escape(topic.getClass().getCanonicalName().replace("MyApp.log.Logger.", "") +
"." + topic.toString()));
        line.append(debug);
        line.append("|");
        line.append(escape(message));
        line.append("|");
        if (throwable == null) {
            line.append("|");
        } else {
            Throwable th = throwable;
            while (th != null) {
                line.append(th.getClass().getName());
                line.append("(");
                line.append(th.getMessage());
                line.append(")");
                line.append("--");
                for (StackTraceElement e : th.getStackTrace()) {
                    line.append("  ");
                    line.append(e.toString());
                    line.append("\n");
                }
                th = th.getCause();
```

```
                if (th != null) {
                    line.append("CAUSED BY:--");
                }
            }
            line.append("|");

        }
        for (Entry<String, String> entry : paramsMap.entrySet()) {
            line.append(entry.getValue());
        }
        if (blockingDur != null) {
            line.append("<Log_Block_Duration>");
            line.append(blockingDur);
            line.append("</Log_Block_Duration>");
        }
        return line.toString();
    }

    /**
     * Get some rough estimate of this LogRequests size
     * @return estimated size
     */
    public int getSize() {
        int length = 50; //estimate of the small non-string parts of the LogRequest
        length += message.length();
        for (Entry<String, String> entry : paramsMap.entrySet()) {
            length += entry.getKey().length();
            length += entry.getValue().length();
        }
        //TO DO: Add some sort of estimate for Throwable
        return length;
    }

    private String escape(String string) {
        String result = string.replace("&", "&amp;");
        result = result.replace("|", "&vbar;");
        return result;
    }

    private void blockingStarted() {
        blockingStart = new Date();
    }

    private void blockingFinished() {
        blockingDur = new Date().getTime() - blockingStart.getTime();
    }



/*------------------------------------- GETTERS ----------------------------------------

     /**
     * Appenders should never change the Throwable obtained by calling this method
     * @return
     */
    public Throwable getThrowable() { return throwable; }

        public Logger.LogLevel getLogLevel() { return logLvl; }

        public Date getDate() { return date; }
```

```
        public String getMessage() { return message; }

        public String getSerial() { return serial; }

        public Long getThreadId() {return threadId; }

        public Map<String, String> getParamsMap() {return paramsMap; }

        public LogTopic getTopic() { return topic; }

        public String getAttributeValue() { return attributeValue; }
    }
}
```

VI Log

```
package MyApp.log;

import java.io.BufferedOutputStream;
import java.io.File;
import java.util.concurrent.ConcurrentLinkedQueue;

/**
 *
 *
 */
public class Log {

    String filename;
    File file;
    String attribute;
    BufferedOutputStream bOS;
    ConcurrentLinkedQueue logRequestQueue;
    LoggerThread queueProcessorThread = null;
    boolean flushMode = false;
    Logger.LogLevel defLvl;
    LogEngine.AlertStatus logAlertStatus = LogEngine.AlertStatus.OK;
    LogEngine engine;

    /**
     * Creates a new instance of Log
     */
    public Log() {
    //fake
    }

    public Log(Logger.Params params) {
            this.filename = params.filename;
            this.defLvl = params.defLvl;
            this.attribute = params.attribute;
            engine = LogEngine.getInst();
            //to be used
            try {
                Appender appender;
                appender = (Appender) params.appender.newInstance();
                appender.setFilename(params.filename);
                LogEngine.addAppenderForLog(appender, this);

            } catch (InstantiationException ex) {
            //  Logger.getDefaultLogger().error(Logger.Logging.LOGGING, ex, "error while creating log
appender", "AppenderClass", params.appender);
            } catch (IllegalAccessException ex) {
            // Logger.getDefaultLogger().error(Logger.Logging.LOGGING, ex, "error while creating log
appender", "AppenderClass", params.appender);
            }
            logRequestQueue = new ConcurrentLinkedQueue();

    }

    public void addToLog(LogEngine.LogRequest line) {
        logRequestQueue.offer(line);
        if (queueProcessorThread == null || queueProcessorThread.isAlive() == false) {
            queueProcessorThread = new LoggerThread("Logger " + filename, this);
```

```
            queueProcessorThread.start();
        }
    }

    public void dump() {
            flushMode = true;
    }

    /**
     * Shutdown this log; if the LoggerThread is running pass it the shutdown command, otherwise report
'shutdown completed' to the LogEngine
     */
    public void shutdown() {
        if (queueProcessorThread != null && queueProcessorThread.isAlive())
queueProcessorThread.shutdown();
        else LogEngine.reportShutdown(this);
    }




    class LoggerThread extends Thread {
        boolean shutdown = false;
        Log log;

        public LoggerThread(String name, Log log) {
            super(name);
            this.log = log;
        }

        public void run() {
            LogEngine.LogRequest request;
            boolean terminated = false;
            boolean success = false;
            while (!terminated) {
                request = (LogEngine.LogRequest) logRequestQueue.peek();
                if (request != null) {
                    if (!flushMode) {
                        success = false;
                        boolean allAppendersSucceeded = true;
                        for (Appender appender : LogEngine.getAppendersForTopic(request.getTopic(), log)) {
                            boolean ok = appender.append(request, request.getAttributeValue());
                            if (ok) {
                                if (appender.isPrimary()) success = true;
                            } else {
                                allAppendersSucceeded = false;
                            }
                        }
                        if (allAppendersSucceeded) success = true; //if all appenders logged, its ok even if
none of them was primary
                    }
                    if (success || flushMode) { //if at least one appender succeeded in writing, take the request
from the queue
                        try {
                            request = (LogEngine.LogRequest) logRequestQueue.poll();
                            if (request != null && !flushMode) {
                                engine.modifyQueueLengthAndCheckStatus(-1 * request.getSize());
                            }
                        } finally {
```

```
                    //the try is here to make SURE the counter moves if and only if the request is taken
from the queue
                    }
                } else {//if no appender succeeded, leave request in queue and wait 200ms
                    try {
                        Thread.currentThread().sleep(200);
                    } catch (InterruptedException ex) {
                        LogEngine.consoleOutStream.println("logger thread " + filename + "
INTERRUPTED");
                    }
                }
            } else {//if queue is empty, tell appenders they can rest for now and wait 200ms before
looking again or commence shutdown if shutdown flag is set
                for (Appender appender : LogEngine.getAppendersForLog(log)) {
                    if (shutdown) {
                        if (logRequestQueue.peek() == null) {
                            appender.shutdown();
                            terminated = true;
                        }
                    } else {
                        appender.rest();
                    }
                }
                flushMode = false;
                try {
                    Thread.currentThread().sleep(200);
                } catch (InterruptedException ex) {
                        LogEngine.consoleOutStream.println("logger thread " + filename + "
INTERRUPTED");
                }
            }
        }
        LogEngine.reportShutdown(log);
    }

    public void shutdown() {
        boolean b = (logRequestQueue.peek() == null);
        shutdown = true;
    }
  }
}
```

VII FileAppender

```
package MyApp.log;

import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 *
 *
 */
public class FileAppender implements Appender {
    static int END_OF_LINE = 30;
    static List<Object[]> allWritersInSystem = new ArrayList();

    File file;
    File dir;
    Map<String, File> subFiles;
    Map<File, BufferedOutputStream> writers;
    String filename;
    Date date;
    Long initDay;
    SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd"+ File.separator);
    File path = LogEngine.getLogDirectory();
    String filenameSuffix = ".log";
    Logger log = Logger.getLoggerForTopic(Logger.Logging.LOGGING);
    boolean resting = false;

    /**
     * Creates a new instance of FileAppender
     */

    public FileAppender() {
    }

    public void setFilename(String filename) {
        this.filename = filename;
        date = new Date();
        initDay = date.getTime()  / 86400000;
        dir = new File(path.getAbsolutePath() + File.separator + formatter.format(date));
        subFiles = new HashMap<String, File>();
        writers = new HashMap<File, BufferedOutputStream>();
    }

    public void rotate() {
        for (BufferedOutputStream bos : writers.values()) {
            try {
                bos.close();
            } catch (IOException ex) {
                //
```

```
            }
        }
        file = null; //to force next append() to create it
        setFilename(filename);
    }


    public boolean append(LogEngine.LogRequest request, String attribute) {
        if (resting) {
            wake();
        }
        try {
            Long today = new Date().getTime() / 86400000;
            if (!today.equals(initDay)) { //TO DO:
                rotate();
                initDay = today;
            }
            if (!dir.isDirectory()) {
                boolean result = dir.mkdirs();
            }
            if (file == null || !file.getName().equals(filename + filenameSuffix)) {
                file = new File(path.getAbsolutePath() + File.separator + formatter.format(date) + filename +
filenameSuffix);
                try {
                    if (writers.get(file) != null)
                        writers.get(file).close();
                    BufferedOutputStream writer = new BufferedOutputStream(new FileOutputStream(file,
true));
                    writers.put(file, writer);
                    Object[] debugEntry = new Object[3];
                    debugEntry[0] = this;
                    debugEntry[1] = file;
                    debugEntry[2] = writer;
                    allWritersInSystem.add(debugEntry);
                } catch (IOException ex) {

                }
            }
            BufferedOutputStream bw = writers.get(file);
            if (attribute != null) {
                attribute = makeFSCompatible(attribute);
                if (subFiles.keySet().contains(attribute)) {
                    bw = writers.get(subFiles.get(attribute));
                } else {
                    File newSubFile = new File(path.getAbsolutePath() + File.separator +
formatter.format(date) + filename + "_" + attribute + filenameSuffix);
                    try {
                        if (!newSubFile.isFile()) {
                            newSubFile.createNewFile();
                        }
                        BufferedOutputStream newWriter = new BufferedOutputStream(new
FileOutputStream(newSubFile, true));
                        subFiles.put(attribute, newSubFile);
                        writers.put(newSubFile, newWriter);
                        Object[] debugEntry = new Object[3];
                        debugEntry[0] = this;
                        debugEntry[1] = newSubFile;
                        debugEntry[2] = newWriter;
                        allWritersInSystem.add(debugEntry);
                        bw = newWriter;
```

```
                } catch (IOException ex) {
                }
            }
        }
        String line = request.getLine();
        line.replace("\u001E", "\u001C");//write the line, replace char 30 - no escaping since there is no
use for this character when reading the logs anyway
        line += ("\n");
        bw.write(line.getBytes("UTF-8"));
        bw.write(END_OF_LINE); //the character used by Log Reader as line separator
        bw.flush();

        return true;
      } catch (IOException ex) {
        handleFailedAttempt(ex);
      }
      return false;
    }



    public void rest() {
      if (!resting) {
      for (BufferedOutputStream bw : writers.values()) {
        closeBW(bw);
      }
      resting = true;
      }
    }

      public boolean isPrimary() {
      return true;
    }

    public void shutdown() {
      for (BufferedOutputStream writer : writers.values()) {
        try {
          writer.close();
        } catch (IOException ex) {
          //well cant do much about that during shutdown
        }
      }
    }

    private String makeFSCompatible(String attribute) {
      StringBuffer buffer = new StringBuffer();
      for (Character ch : attribute.toCharArray()) {
        if (Character.isLetterOrDigit(ch) || ch == '_' || ch == '-') {
          buffer.append(ch);
        } else {
          buffer.append("_");
        }
      }
      return buffer.toString();
    }


    private void wake() {
      for (File f : writers.keySet()) {
        try {
```

```
                if (writers.get(f) != null)
                    writers.get(f).close();
            }
            catch (IOException ex) {

            }
            try {
                writers.put(f, new BufferedOutputStream(new FileOutputStream(f, true)));
            } catch (FileNotFoundException ex) {

            }
        }
    }


    private void closeBW(BufferedOutputStream bw) {
              try {
            bw.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }



    private void handleFailedAttempt(IOException ex) {

    }

}
```

VIII ObjectToXMLConvertor

```
package MyApp.log;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import org.dom4j.DocumentFactory;
import org.dom4j.Element;

public class ObjectToXMLConvertor {

    /** Creates a new instance of ObjecttoXMLConvertor */
    public ObjectToXMLConvertor() {
    }

    public static String convert(Object obj, String paramName) {
            return convertToElement(obj, paramName, new ArrayList<Object>()).asXML();
    }

    private static Element convertToElement(Object obj, String paramName, List<Object> stack) {
            stack.add(obj);
            Element e = DocumentFactory.getInstance().createElement(paramName);
            if (obj == null) {
            e.setText("(null)");
        }
            else if (obj instanceof Integer || obj instanceof Long || obj instanceof Double
            || obj instanceof Short || obj instanceof Character || obj instanceof CharSequence) {
            e.setText(obj.toString());
        }
            else if (obj instanceof Boolean) {
                if ((Boolean)obj) e.setText("true");
                else e.setText("false");
            }
            else if (obj instanceof Iterable) return IterabletoXML(obj, e, stack);
            else if (obj instanceof Loggable) return ((Loggable)obj).toLogXML(paramName, stack);
            else e.setText(obj.toString());
            stack.remove(obj);
            return e;
    }

    private static Element IterabletoXML(Object obj, Element e, List<Object> stack) {
        Iterator iter = ((Iterable)obj).iterator();
        int i = 0;
        while (iter.hasNext()) {
            Object next = iter.next();
            if (stack.contains(next)) {
                Element newElem = DocumentFactory.getInstance().createElement(e.getName() + "-" + i);
                newElem.setText(next.toString());
                e.add(newElem);
            }
            else e.add(convertToElement(next, e.getName() + "-" + i, stack));
            i++;
        }
        return e;
    }
}
```