TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B 2612 – Elektrotechnika a informatika
Studijní obor: 2612R011 – Elektronické informační a řídicí systémy

Efektivní logování procesů v prostředí Sun Java

Effective logging of processes in the Sun Java environment

Bakalářská práce

Autor: **Robert Jevčič**

Vedoucí práce: **ing. Igor Kopetschke**

Liberec 29.05.2009

Zde bude originál zadání BP

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce.

Datum: 29.05.2009

Podpis:

Abstrakt

Bakalářská práce se zabývá otázkou, jak a co nejefektivněji logovat v aplikacích postavených na Java2 platformě společnosti Sun Microsystems. Nežli se dostane k samotné otázce logování, doporučuje postup při vývoji aplikací, popisuje jejich životní cyklus a obecně pohlíží na chyby, jež mohou v aplikaci nastat. Dále se snaží vysvětlit pojem logování, k čemu ho lze použít a proč je nezbytně nutné při vývoji rozsáhlejších aplikací.

Následuje rešerše logovacích nástrojů. Nástroje jsou popsány a rozděleny z hlediska funkčnosti. Z funkčního hlediska rozděluje tato práce logovací API do dvou skupin. Do první skupiny patří nástroje, které slouží pouze k logování. Druhou skupinou nástrojů jsou takové, které slouží jako nadstavba skupiny první. Jejich pomocí lze dostát nezávislosti použitého logovacího frameworku ze skupiny první.

Zbytek práce je zaměřen na knihovnu Log4j od společnosti APPACHE. Jedná se o nejvýkonnější a nejpropracovanější nástroj, který je v tuto dobu na trhu. Zřejmě i z tohoto důvodu se těší největší oblibě u většiny vývojářských týmů. Ve zbytku práce je popsána práce s Log4j. Popis je podrobný do té míry, aby se dala práce použít jako ucelený učební text, doplněný dostačujícím počtem praktických ukázek.

Poslední částí je miniaplikace, jež implementuje knihovnu Log4j. Generuje různé druhy výstupů na různá zařízení. Zároveň se zabývá možností strojového zpracování logů, které nebyly zformátovány pomocí XMLLayoutu.

Abstract

Bachelor thesis deals with question, how most effectively to log events in applications, based on Java2 platform of Sun Microsystems. Before logging, there is described development and lifetime of applications in generally. There are described errors, which can happen. That there is thought of logging, why is it so important and necessary in development and service of application.

Next part starts with recherché of logging tools. Logging tools are described and divided in the two main groups. First group contains really logging API's. In second are frameworks, which are wrappers of logging tools from first group. The Second group makes application independent on chosen logging API.

Remaining part is specialized on package Log4j from The Apache Software Foundation. Log4j is best performed and most popular logging toolkit. There is described working with this library.

Final part of this thesis is concentrated in small application, which is trying to parse and represent logging outputs, that was not formatted by XMLLayout.

1 Životní cyklus aplikace

1.1 Analýza

Mnoho odlišných cest může vést k vytvoření kvalitní aplikace, tj. aplikace, jejíž výsledný vzhled a chování budou přesně odpovídat požadavkům zákazníka. Prvním předpokladem k dosažení tohoto nemalého cíle, nebo alespoň k jeho maximálnímu přiblížení, je podrobná analýza. Na jejím počátku by měl být kladen důraz nejen na zjišťování požadavků, ale zároveň by se mělo zjistit, zda se požadavky rovnají skutečné potřebě zákazníka. Po vzájemné dohodě mezi zúčastněnými stranami, tedy zákazníkem a dodavatelem, je možné vypracovat a nabídnout zákazníkovi návrh vhodného řešení. Tento návrh bude v jeho konečné podobě obsahovat podrobný popis aplikace pomocí prostředků k tomu určeným a další specifikace (volba operačního systému, databáze, programovacích prostředků, harmonogram zavedení systému, …).

1.2 Implementace

Samotný vývoj aplikace probíhá na základě analýzy. Je zapotřebí dodržovat jak popsané specifikace, tak i termíny.

1.3 Testování

Na počátku životního cyklu aplikace bude provedena analýza testů. Analýza testů definuje riziková místa z pohledu výsledné kvality aplikace, jakými prostředky, v jakém rozsahu, nad kterými prvky aplikace a kdy bude ověřování a vyhodnocení jakosti provedeno. V analýze testů se dále stanovují ukazatele, metrika a hladiny dosažení kvality (akceptační kritéria). Tato analýza je postupně v průběhu životního cyklu zpřesňována do plánu testů tak, jak se zpřesňují informace o návrhu řešení.

Plán testů je po dokončení návrhu aplikace rozpracován do podrobných testovacích scénářů, jsou připraveny testovací skripty, ověřovací sady dat a jsou připraveny veškeré nástroje a zázemí pro ověřování kvality. Poté se provede ověření správnosti zvolených testovacích postupů a zdrojů a ověří se shoda s požadavky analýzy a plánu testů. Samotné ověřování kvality produktu probíhá souběžně s vývojem. Vždy po dokončení dílčí části produktu bude v postupných krocích provedeno ověření nových prvků (komponentové testy), poté se ověří jejich spolupráce a integrace s jinými již ověřenými částmi produktu (integrační testy) a nakonec se provedou kompletní testy

celého systému (systémové testy). Tímto způsobem zařazení testů vedle vývoje jsou minimalizovány časové prodlevy mezi dokončením vývoje produktu a jeho nasazením do provozu.

1.4 Zavedení do provozu

Po nasazení do provozního prostředí jsou pak ve spolupráci se zákazníkem prováděny akceptační testy a poté systém uveden do provozu.

1.5 Servis a správa aplikace

Po uvedení aplikace do provozu může být zajišťována jak záruční a pozáruční služba, tak zákaznická podpora.

2 Chyby aplikace

2.1 Funkční chyby

Funkční chybou nazýváme takový stav, kdy se chování aplikace či jejích jednotlivých komponent neztotožňuje s požadavky na něj kladenými (chyba logiky v programu). Jsou to chyby, které nelze pomocí programovacích prostředků nijak zaznamenat.

2.2 Kompilační chyby

Kompilační chyba nastává tehdy, není-li dodržena správná syntaxe daného programovacího jazyku. Java kompilátory (překladače) používají při kompilaci rozsáhlou a přísnou kontrolu syntaktických chyb, s pomocí které může být velké množství chyb odhaleno již při vývoji aplikace. Překladač jazyka Java hlásí i jiné chyby než syntaktické, například použití neinicializované proměnné a podobně.

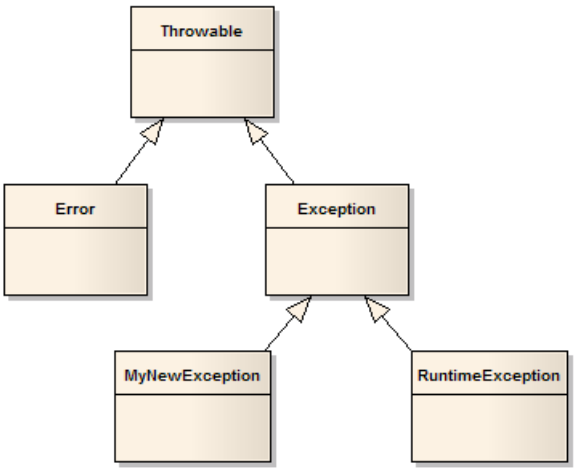
2.3 Běhové chyby

Běhové chyby vznikají až za běhu aplikace. Mezi ně patří například dělení nulou, odmocňování záporného čísla, překročení rozsahu pole, překročení rozsahu typu (celočíselného typu), práce s doposud nevytvořeným objektem, pokus o otevření neexistujícího diskového souboru pro čtení a podobně.

2.4 Výjimky

Pro ošetření běhových chyb využívá Java mechanizmu výjimek. Výjimka je generována tehdy, dojde-li v programu právě k běhové chybě. Mechanizmus výjimek je jedním z velmi silných bezpečnostních opatření jazyku Java. Již na úrovni překladače nutí programátora k ošetření některých chybových stavů, jež by mohly nastat.

Jazyk Java rozlišuje tři základní druhy výjimek `Error`, `RuntimeException` a `Exception`. Právě u poslední z uvedených, tedy Exception, nutí překladač jazyku Java programátora k jejímu ošetření. Reakce programátora na zbylé dva typy výjimek je dobrovolná. Základní hierarchie výjimek je zobrazena pomocí diagramu tříd na Obr. 1.



Obr. 1: Diagram tříd – základní hierarchie výjimek

2.4.1 Třída Error

Třída `Error` a třídy z ní odvozené představují závažné chyby aplikace, které se mohou vyskytnout ve virtuálním stroji (JVM). Tyto chyby obvykle neošetřujeme. V případě jejich výskytu program skončí s příslušnou chybovou hláškou. Toto chování můžeme demonstrovat např. na výjimce typu OutOfMemoryError, kdy se JVM snaží využít více paměťových prostředků, než má povoleno (v tomto případě už zřejmě nebude vhodné vytvářet další objekty v paměti, kde pro ně beztak již není místa). Tyto výjimky patří mezi nekontrolované výjimky (unchecked exceptions).

2.4.2 Třída RuntimeException

Třída `RuntimeException` a její podtřídy popisují výjimky, které mohou nastat kdekoliv v programu – jsou známy též pod označením asynchronní výjimky. Je čistě volbou programátora, zda na ně bude reagovat či nikoliv, překladač nás v tomto případě k ošetření nenutí (patří mezi nekontrolované výjimky).

Mezi výjimkami tohoto typu uvedu dva příklady, které postačí pro představu a lze se s nimi během programování běžně setkat. Jsou jimi `NullPointerException` a `IndexOutOfBoundsException`. Výjimka typu `NullPointerException` je vyvolána v případě, v němž se snažíme pracovat s referenční proměnnou nějaké třídy (volat na ní metody, …), avšak proměnná se neodkazuje na platný objekt. Výjimky typu `IndexOutOfBoundsException` se mohou přihodit při práci s poli, kdy se snažíme přistoupit k prvku pole, který je mimo rozsah daného pole.

2.4.3 Třída Exception

Výjimky této třídy a jejích potomků patří mezi kontrolované výjimky (checked exceptions). JVM nás při psaní kódu nutí tyto výjimky ošetřovat. Bez ošetření části kódu, v níž k vyvolání výjimky dochází, dojde při pokusu o kompilaci k chybě.

Patří do skupiny synchronních výjimek. Na rozdíl od asynchronních výjimek se nemohou vyskytovat kdekoliv v kódu. Vyskytují se v souvislosti s voláním určitých metod. Jedná se např. o metody, které pracují se vstupy či výstupy. Chceme-li naprogramovat novou výjimku a požadujeme, aby byla při použití ošetřena, podědíme tuto novou třídu právě od třídy `Exception`.

2.4.4 Ošetřování výjimek

Je na programátorovi samotném (závisí to i na architektuře aplikace), jakým z možných postupů ošetří kód, který vyžaduje zpracování výjimky. Může výjimku zachytit a kompletně zpracovat pomocí `try-catch` bloku. Nechce-li v daném místě programového kódu výjimku zpracovávat, může předat informaci o jejím výskytu o úroveň výše, tedy volající metodě (pomocí klíčového slova `throws`). Dalším způsobem ošetření výjimky jest kombinace obou předchozích. V takovém případě částečně či kompletně ošetříme výjimku v metodě a zároveň poskytneme informaci o jejím výskytu nadřazené metodě.

Příklad kompletního ošetření výjimky (výjimek):

```
import org.apache.log4j.Logger;
public class MyClass
{
  private static final Logger log = Logger.getLogger(MyClass.class);
  /**
   * Returns integer value >= 0 in case of success. If error occurred
   * returns negative integer value.
   * @return integer number.
   */
  public int readSomething() {
    int number = 0;
    try {
      // kód, který může obsahovat vyvolání výjimky
    } catch( NumberFormatException e) {
        log.error(e.getMessage());
        return -1;
    } catch( ArithmeticException e) {
        log.error(e.getMessage());
        return -2;
    }
    return number;
  }
}
```

Příklad předání výjimky o úroveň výše (vyhození):

```
public static void doSomething() throws Exception {
      // deklarace těla metody
}
```

3 Logování a trasování

3.1 Úvod

Během celého životního cyklu aplikace, ať už je to během jejího vývoje nebo ve stádiu, kdy je aplikace nasazena a servisována, je velice důležité a užitečné mít možnost analyzovat vnitřní chování a chod. V každém stádiu jejího života potřebujeme mít možnost co nejrychleji najít a odladit případnou chybu v aplikaci, anebo zjistit podrobnosti, které vedly k pádu aplikace či její částečné nefunkčnosti.

Při vývoji lze poměrně dobře ladit aplikaci pomocí debuggeru. Debugger je softwarový nástroj, jenž se používá pro nalézání programátorských chyb ve zdrojovém kódu aplikace. Za chodu aplikace umožňuje „krok po kroku“ procházet zdrojové kódy právě probíhajících procesů, zkoumat stav těchto procesů: obsah paměti, hodnoty na zásobníku, objektové závislosti, a z nich usuzovat, proč k chybě v programu dochází. Ladění chyb pomocí debuggeru je poměrně časově náročné a jistou měrou zvyšuje nároky na systémové zdroje. Jeho činnosti lze poměrně dobře využít při implementaci aplikace. Většina dnes dostupných vývojových prostředí (ať pro jazyk Java či jiný) takový debugger obsahuje. Jeho absence by při vývoji aplikace velice znepříjemnila život nejednomu programátorovi. Jsou však situace, kdy je použití debuggeru nevhodné, nebo dokonce nemožné. V takovém případě potřebujeme zapojit do hry jiný nástroj.

Dostaneme-li se do situace, kdy nelze použít debuggeru – např. po nasazení aplikace do provozu, je zapotřebí zajistit výpis stavů proměnných, objektů a chod programu všeobecně na nějaké médium (logování). Takového nástroje samozřejmě využíváme i během vývoje, současně s použitím debuggeru. Na použití takového nástroje je samozřejmě myšleno již při vývoji aplikace. Je nasnadě dopředu si rozmyslet míru podrobnosti a formát informací, které budeme chtít pro jejich pozdější použití uchovávat. Taková otázka pak povede i k volbě řešení.

3.2 Výběr logovacího API

Výběr logovacího nástroje by se měl řídit jak požadavky na jeho skutečnou potřebu, ale také s ohledem na vynaložené úsilí programátorů. V dnešní době existuje mnoho užitečných či méně užitečných nástrojů, které lze k tomuto účelu využít. Není tedy zapotřebí utrácet peníze za ne zrovna levný čas programátorů při vytváření vlastního logovacího API. Existuje jich celá řada a jejich využití je dokonce zdarma.

Naučit se s některým z nich pracovat zabere v porovnání s napsáním takové aplikace minimum času.

3.3 Výhody a nevýhody logování

3.3.1 Výhody logování

- Logování může vygenerovat detailní informace o chodu aplikace
- Je-li logování jednou přidáno do projektu, již nevyžaduje dalšího zásahu
- Aplikační logy mohou být ukládány a studovány v pozdější době
- Jsou-li logy dostatečně podrobné a řádně zformátovány, lze je použít při auditu
- Zaznamenání chyb, jež se nezobrazují uživatelům aplikace, může pomoci při odstraňování potíží
- Logování může sloužit pro ladění chyb tam, kde nelze použít debugger

3.3.2 Nevýhody logování

- Zvýšení časové náročnosti generováním a zaznamenáváním zpráv
- Zvětšení objemu kódu (odhaduj se, že logování tvoří okolo 4% kódu)
- V případě nevhodného formátování a mnohoslovnosti může být zhoršena jejich čitelnost a použití
- Přidání logovacího kódu může zhoršit čitelnost kódu
- Může klást na vývojáře značné úsilí, není-li zavedeno již na začátku vývoje aplikace a dopisuje se dodatečně

3.3.3 Výhody vs. nevýhody

Domnívám se, že při správném použití dobře navrženého logovacího API výhody předčí nevýhody. Ba naopak, se jeho nepoužitím při tvorbě rozsáhlejšího projektu můžeme dříve či později dostat do nemalých potíží a aplikace se stane neudržitelnou.

4 Logovací frameworky platformy Java

Logování je dnes běžnou záležitostí pro většinu vývojových týmů. V průběhu času bylo vytvořeno několik frameworků, pomocí nichž se zjednodušil a standardizoval proces logování v platformě Java.

4.1 Model logování

Funkčnost většiny z frameworků si je v základním principu podobná. Logování událostí je rozděleno do tří hlavních částí. Zachycení události (zprávy), její zformátování a zapsání na příslušný výstup. V této kapitole se budu snažit podrobně popsat vlastnosti jednotlivých API a rozdíly mezi nimi.

Další základní vlastností je, na jakou činnost jsou balíky přednostně určeny. Balíky Log4j (`org.apache.log4j`) a Java Logging API (`java.util.logging`) jsou v podstatě plnohodnotnými logovacími nástroji. Další z popisovaných API – Jakarta Commons Logging a SLF4J – jsou spíše jakousi nadstavbou (wrapperem) logovacího frameworku, která slouží k dosažení nezávislosti aplikace na konkrétním API.

Základní princip frameworků Log4j a Java Logging lze popsat v rámci tohoto odstavce. V popisu základní funkčnosti bude dostačující uvést rozdíl v pojmenování některých tříd, rozdíly další popíši zvlášť v odstavci 4.6.1. Jak jsem již uvedl dříve, popis funkcionality logování lze rozdělit do tří skupin, jímž odpovídají tři komponenty (třídy, rozhraní). Jsou jimi `Logger`, `Appender` (`Handler`) a `Layout` (`Formatter`). V závorkách jsou uvedeny názvy, jež používá Java Logging Framework, mimo ně Log4j.

Komponenta Logger je odpovědná za odchycení události (zprávy) určité úrovně a její předání do logovacího frameworku. Tato zpráva je dále zformátována pomocí komponenty `Layout` (`Formatter`). O samotné zaznamenání do logu se postará komponenta `Appender`. V praxi to funguje tak, že jednotlivým loggerům přiřadíme jeden či více appenderů. Těmto appenderům se nastaví požadovaný layout.

Další informace o principech logování popíši v následujících bodech a pak se budu věnovat podrobně popisu jednoho z nich, a to Log4j. Rozhodl jsem se tak z důvodů, že je framework Log4j více používaným a poskytuje širší škálu možností.

4.2 Log4j API

Log4j je knihovna nabízená pod open source licencí. Byla vyvinuta jako podprojekt projektu Apache Software Foundation's Logging Services. Jejím základem byla logovací knihovna, vyvíjená firmou IBM na konci devadesátých let minulého století. První verze byla představena v roce 1999.

Jak jsem uvedl již v předchozím textu, architektura Log4j je postavena na třech základních komponentách: `Logger`, `Appender` a `Layout`. Tento koncept dovoluje vývojářům logovat zprávy podle jejich typu a priortity, kontrolovat kam a v jakém tvaru budou zapsány. Loggery jsou objekty, které aplikace zavolá jako první při zahájení logování zprávy. Je-li loggeru předána zpráva k logování, logger vygeneruje objekt třídy `LoggingEvent`, který obalí (wrapne) danou zprávu. Loggery pak tyto objekty předávají jejich asociovaným appenderům. Appendery posílají informace obsažené v objektech třídy `LoggingEvent` na specifikovaný výstup. Uvedení příkladu pro názornost: `ConsoleAppender` zapíše informace do výstupního proudu System.out (standardní výstup), FileAppender je zapíše do souboru. Před odesláním informací, obsažených v `LoggingEvent` objektu, využijí některé appendery layoutu pro vytvoření textové reprezentace informace v požadovaném formátu.

V log4j mají objekty typu `LoggingEvent` (události, zprávy) přidělenou úroveň, jež určuje jejich prioritu. Implicitní úrovně (seřazeny od nejvyšší k nejnižší prioritě) jsou `OFF`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG` a `ALL`. Loggery a appendery mají též přidělenou úroveň. Tyto vykonají svou práci a zalogují pouze v případě, že požadavky na logování (tedy události) jsou úrovně vyšší či rovné úrovni nastavené v loggerech a appenderech. Vyvolám-li například požadavek na logování úrovně `DEBUG` a je-li appenderu nastavena úroveň `INFO`, událost zalogována nebude.

Konstruktor třídy `Logger` obsahuje parametr `name`, jehož hodnotu nahraje do členské proměnné `name` při vytváření instance. Každý objekt typu `Logger` má tedy své jméno. Log4j řadí instance třídy `Logger` do stromové struktury dle jejich jmen stejným způsobem, jako jsou organizovány balíčky v jazyku Java. Jak je uvedeno v dokumentaci Log4j, logger je předkem (ancestor) jiného loggeru, začíná-li jméno potomka (descendant) jeho jménem následovaným tečkou. Logger je nazýván rodičem (parent) potomka (child), nejsou-li mezi ním a potomkem (descendant) žádní předci (ancestors). Vzhledem k tomu, že jsem pro anglické výrazy „descendant“ a „child“ nenalezl adekvátní překlad, jenž by se lišil, použil jsem v závorkách termíny originální.

Snad jen bychom mohli potomka (descendant), nazvat jakýmsi „vzdáleným potomkem“. Pro názornost můžeme uvést příklad, kdy logger pojmenovaný jako "a.b" je přímým potomkem loggeru "a". Logger "a.b.b" je potomkem loggeru "a.b" a prapotomkem loggeru "a". Nemá-li logger explicitně nastavenou úroveň, tak používá úroveň svého nejbližšího předka, kterému byla úroveň přiřazena. Loggery dědí také appendery, ačkoliv mohou být nakonfigurovány tak, aby používali pouze appendery, které mají nastaveny přímo.

Je-li logger požádán k zalogování zprávy, tak nejprve zjistí, zda požadavek splňuje podmínku pro zalogování (úroveň požadavku >= úroveň loggeru). Pokud je podmínka splněna, vytvoří z dané zprávy objekt typu `LoggingEvent` a pošle ho svým appenderům. Appendery událost zformátují a pošlou ji na svůj výstup (výstupy).

4.3 Java Logging API (JLA)

Balík `java.util.logging`, který společnost Sun Micosystems představila v roce 2002 jako součást balíku J2SE (verze 1.4), je implementován dle specifikace JSR 47. Tento balík je velice podobný Log4j – používá podobného konceptu, akorát některé z komponent, jak již bylo uvedeno dříve v této práci, nazývá vlastními jmény. `Appender` je `Handler`, `Layout` je `Formatter` a `LoggingEvent` nazývá jako `LogRecord`. Java Logging API používá úrovně stejně jako Log4j. Rozdíl je akorát v počtu defaultních úrovní, kdy má Log4j o jednu méně (tedy osm oproti devíti). JLA používá stejné hierarchie pro řazení loggerů jako Log4j. Loggery také dědí vlastnosti od svých rodičovských (předkovských) loggerů. Rozhodne-li se vývojář, který umí pracovat s Log4j používat JLA, jediné co bude potřebovat je slovník, který bude přiřazovat původní názvy komponent Log4j k názvům komponent JLA.

4.4 Jakarta Commons Logging (JCL)

Tento projekt funguje jako nadstavba logovacího toolkitu. Při použití tohoto API nám je umožněno použít jakéhokoliv podporovaného logovacího toolkitu, a to beze změny v kódu aplikace. Mezi podporované logovací nástroje patří Log4j, Avalon LogKit a Java Logging API. JCL poskytuje pouze jakési přemostění pro psaní zpráv. Jednou z velkých výhod takového přístupu je přenositelnost aplikace, či jejích částí (například nějakého balíku) a jejich opětovné využití v rámci jiných aplikací, bez potřeby úpravy logovacího kódu aplikace. Nevím, do jaké míry dochází k využití tohoto

API, spíše bude záležet na politice té které firmy a míře nadšení dělat aplikace a kód univerzálními.

4.5 Simple Logging Facade for Java (SLF4J)

SLF4J framework byl vytvořen vývojářem Ceki Gülcü, jakožto vylepšení již tak velmi úspěšného a hojně využívaného logovacího nástroje Log4j. Požadavek na vytvoření tohoto nástroje byl vyvolán kritikou Log4j po výkonnostní stránce. Dalším údajným důvodem by měla být jistá kritika Jakarta Commons Logging frameworku.

Byla vynechána logovací úroveň `FATAL` na základě toho, že by v logovacím nástroji nemělo být místo pro rozhodování, kdy má být aplikace ukončena. Tím pádem už není rozdílu mezi úrovněmi `FATAL` a `ERROR`.

Logování je podobné frameworku JCL, kdy jsou logy vytvářeny též pomocí třídy `LoggerFactory`. Tento postup (nebo spíše jeho forma) samozřejmě není ani nijak zvlášť vzdálen logování v ostatních logovacích API.

Logovací metody jsou přetíženy, mohou akceptovat jednu, dvě či více hodnot. Výskyty uzavřených složených závorek {} v logovaných zprávách, jsou pak nahrazeny hodnotami z parametrů metody. Navíc tento jednodušší způsob poskytuje výkonnostní zlepšení. Je-li úroveň logované hlášky menší, než nastavená, nedochází k volání časově náročných toString() metod. Bude-li v následujícím příkladu nastavena úroveň logování na `INFO`, proběhne spodní řádek rychleji.

```
private static final Logger LOG =
LoggerFactory.getLogger(MyClass.class);

LOG.debug("V aplikaci je " + count + " uzivatelu: " +userAccountList);
LOG.debug("V aplikaci je {} uzivatelu: {}", count, userAccountList);
```

4.6 Log4j a Java Logging API podrobně

4.6.1 Funkční rozdíly

Přestože jsou tyto dva frameworky velice podobné, je mezi nimi jistý rozdíl. Lze ho vyjádřit rčením „Cokoliv Java Logging API dokáže, umí Log4J také – a umí toho ještě více.“. Nejvíce se liší v implementacích appenderů/handlerů, layoutů/formatterů a možnostech konfigurace. Java Logging API obsahuje čtyři konkrétní implementace handlerů, kdežto Log4j obsahuje sedmnáct implementací appenderů. Java Logging API handlery jsou použitelné pro základní logování – poskytují možnost zapisovat do

bufferu, konzole, socketu a souboru. Na druhé straně Log4j definuje appendery, které poskytují možnosti logování na výstupy všech možných rozmanitých typů (od zapisování do logů, systémových logů – Unix, Windows, přes posílání zpráv e-mailem, až po ukládání do databáze).

Java Logging API má pouze dva formattery `SimpleFormatter` a `XMLFormatter`. Log4J obsahuje jejich obdoby v podobě appenderů a má další dva navíc. Stejně tak je tomu i v případě layoutů/formatterů.

Rozdíly v možnostech použití výše popsaných logovacích frameworků, jsou uvedeny v následující tabulce.

API	Podporované úrovně	Standardní appenders (handlers)	Standardní layouty (formattery)
Log4J	OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, ALL	AsyncAppender, ConsoleAppender, DailyRollingFileAppender, ExternallyRolledFileAppender, FileAppender, JDBCAppender, JMSAppender, LF5Appender, NTEventLogAppender, SMTPAppender, NullAppender, RollingFileAppender, SocketAppender, SocketHubAppender, SyslogAppender, TelnetAppender, WriterAppender	DateLayout, HTMLLayout, PatternLayout, SimpleLayout, XMLLayout
Java Logging API	OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL	Závisí na použitém frameworku, defaultně je použita Sun JVM implementace: MemmoryHandler, ConsoleHandler, FileHandler, SocketHandler	SimpleFormatter, XMLFormatter
Jakarta Commons Logging	OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE, ALL	Závisí na použitém frameworku.	Závisí na použitém frameworku.
SLF4J	OFF, ERROR, WARN, INFO, DEBUG, TRACE, ALL	Závisí na použitém frameworku.	Závisí na použitém frameworku.

Tabulka 1: Rozdíly funkčnosti jednotlivých testovacích frameworků

4.6.2 Nastavení logování pomocí konfiguračních souborů

Log4j i Java Logging API (JLA) mohou být nastaveny pomocí konfiguračních souborů. Log4j poskytuje širší rozsah možností při použití konfiguračního souboru, nežli je tomu u JLA. JLA může být též konfigurován pomocí properties souboru. Použijeme-li však J2SE 5.0 či starší, konfigurace handlerů se váže ke třídě a nikoliv k instanci třídy. Tento neduh se projevuje tím, že dojde ke ztrátě informací o nastavení z konfiguračního souboru (například vytvoříme-li více instancí třídy `FileHandler`, nebudeme jim moci přidělit různé soubory, do kterých by mohly zapisovat). Tohoto nastavení lze samozřejmě i zde dosáhnout, nikoli však pomocí konfiguračního souboru. Takovouto změnu bychom mohli jednoduše provést, ale pouze v kódu aplikace.

Log4j může být konfigurováno jak pomocí properties souboru, tak i XML souboru a appendery můžou být konfigurovány na instanční bázi. Přesně to samé lze říci a layoutech.

Hlavní výhodou nastavení pomocí konfiguračních souborů je schopnost měnit vlastnosti logování naší aplikace, bez nutnosti rekompilace zdrojových kódů a nasazování nové verze aplikace (často je taková změna pouze dočasná, kdy po odstranění problému nastavujeme vlastnosti logování na původní hodnoty).

4.7 Výběr logovacího frameworku

Log4j poskytuje funkcionality, které JLA postrádá, ačkoliv JLA Log4j dohání. JLA může být v případě potřeby rozšířeno do podoby Log4j – lze napsat více handlerů, naimplementovat obdobu `PatternLayoutu` a zdokonalit konfigurační mechanizmus. Otázkou zůstává, není-li takováto cesta zbytečně trnitá.

Podobná důležitá rozhodnutí, mezi něž volba logovacího frameworku jistě patří, běžně dělají projektoví architekti či zkušení programátoři. Avšak zodpovězením následujících tří otázek, můžeme dojít k možnému řešení:

Otázka 1:

Využijeme rozmanitosti různých výstupů, na které je možné pomocí Log4j logovat (databáze, e-mail), nebo nám postačí běžné logování do souboru?

Otázka 2:

Obejdeme se bez možnosti plně ovlivnit formát logované zprávy, tedy využití `PatternLayoutu` Log4j?

Otázka 3:

Využijeme možnost měnit komplexní nastavení logování v naší aplikaci po té, co byla nasazena na produkční prostředí (Log4j)?

Odpovíme-li alespoň jednou kladně pro Log4j, bude rozumné tento logovací toolkit použít. Vzhledem ke skutečnosti, že jsou tyto dva frameworky velice podobné i co se složitosti týče a Log4j nabízí více, nahrává to právě Log4j.

5 Log4j podrobně

5.1 Úrovně logování

Třída `org.apache.log4j.Level` definuje následující úrovně logování: `OFF`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`, `ALL`. Třída `Level` je potomkem třídy `org.apache.log4j.Priority`, ve které jsou nadefinovány hodnoty priority jednotlivých úrovní. Hodnoty priorit lze k úrovním přiřadit analogicky dle názvů.

org.apache.log4j.Priority		
public static final int	ALL_INT	-2147483648
public static final int	DEBUG_INT	10000
public static final int	ERROR_INT	40000
public static final int	FATAL_INT	50000
public static final int	INFO_INT	20000
public static final int	OFF_INT	2147483647
public static final int	WARN_INT	30000

Tabulka 2: Přehled priorit definovaných v třídě Priority

org.apache.log4j.Level		
public static final int	TRACE_INT	5000

Tabulka 3: Přehled priorit definovaných v třídě Level

Mechanizmus vyhodnocování, zda se zpráva dané úrovně zaloguje, jsem vysvětlil již v bodu 4.2. Dalším důležitým aspektem, který ještě vysvětlen nebyl, je přidělení událostí do správné úrovně.

Přiřazení logovaných událostí k jejich úrovni:

`TRACE`

Události této úrovně jsou ještě podrobnějšími, nežli v úrovni `DEBUG`. V praxi se tato úroveň příliš nepoužívá.

`DEBUG`

Události zaznamenávané v této úrovni jsou velmi podrobné a lze jich využít pro debugování aplikace (trasování, výpis proměnných, …). Po nasazení aplikace do provozu je tato úroveň většinou vypnuta, zapíná se na dobu nezbytně nutnou v případě poruchy aplikace.

`INFO`

Události této úrovně jsou „hrubé“ a mají informativní charakter. Lze je využít pro trasování či zaznamenávání zajímavých událostí v aplikaci.

`WARN`

V této úrovni zaznamenáváme události, ve kterých se program dostává do neočekávaných či nežádoucích stavů, jež by mohly vést např. ke špatné funkci aplikace.

`ERROR`

Mezi události úrovně `ERROR` patří takové chybové stavy, jejichž ošetřením lze pokračovat v běhu aplikace. Mezi tyto události patří i výjimky (vzhledem k velkým nárokům na zpracování by se neměli používat pro ošetření běžného chodu aplikace).

`FATAL`

Mezi události úrovně `FATAL` patří vážné chyby, které pravděpodobně povedou k pádu aplikace (třída `Error`).

Pro přehlednost přikládám tabulku, která graficky znázorňuje, které události se zalogují při nastavených úrovních loggeru. V políčkách označených zelenou barvou, se zprávy zalogují, v červených nikoliv.

		Zaloguje zprávu úrovně				
		DEBUG	INFO	ERROR	WARN	FATAL
úroveň loggeru	DEBUG					
	INFO					
	ERROR					
	WARN					
	FATAL					
	ALL					
	OFF					

Tabulka 4: Logování zprávy dané úrovně dle nastavené úrovně loggeru

5.2 Třída Logger

Třída `org.apache.log4j.Logger` je centrální třídou balíku Log4j. Většina logovacích operací, vyjímaje konfiguraci, je prováděna skrze tuto třídu. Její funkci jsem popsal v odstavcích 4.1 a 4.2.

Uvedu zde názornou ukázku, kdy nenastavujeme logování pomocí konfiguračního souboru, ale ve zdrojovém kódu aplikace:

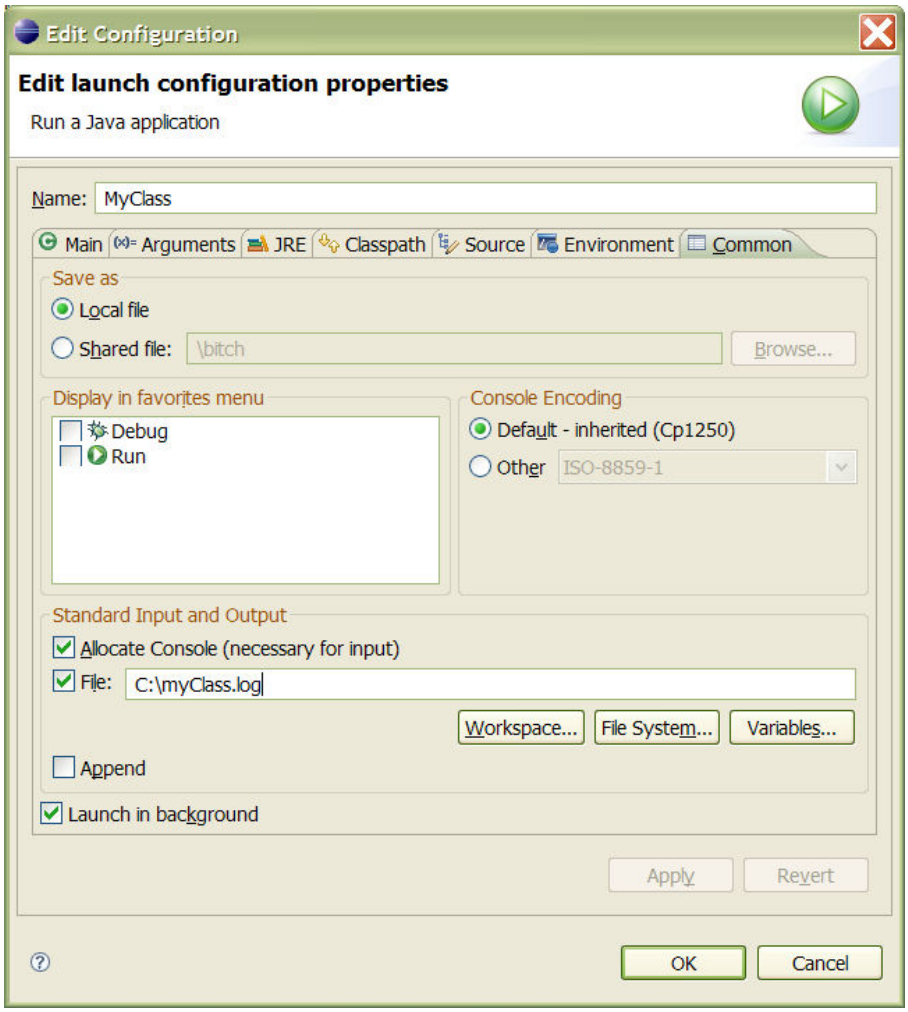
```
import java.io.FileOutputStream;

import org.apache.log4j.HTMLLayout;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;
import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.WriterAppender;

public class MyClass {
  static Logger logger = Logger.getLogger(MyClass.class);

  public static void main(String args[]) {
    // vypis do standardniho vystupu
    String pattern = "[%d{dd.MM.yyyy HH:mm:ss,SSS}] [%5p] [%l] %m%n";
    PatternLayout patternLayout = new PatternLayout(pattern);
    ConsoleAppender consoleAppender
      = new ConsoleAppender(patternLayout);

    //vypis html do souboru
    HTMLLayout htmlLayout = new HTMLLayout();

    WriterAppender writerAppender = null;
    try {
       FileOutputStream output = new FileOutputStream("output.html");
       writerAppender = new WriterAppender(htmlLayout,output);
    } catch(Exception e) {
      logger.error(e.getMessage());
    }

    logger.addAppender(consoleAppender);
    logger.addAppender(writerAppender);
    logger.setLevel((Level) Level.DEBUG);

    logger.debug("DEBUG message");
    logger.info("INFO message");
    logger.warn("WARN message");
    logger.error("ERROR message");
    logger.fatal("FATAL message");
   }
}
```

Před spuštěním aplikace je vhodné nastavit přesměrování standardního výstupu viz. Obr. 3.

Log session start time Mon Jun 01 05:58:46 CEST 2009

Time	Thread	Level	Category	Message
0	main	DEBUG	MyClass	DEBUG message
0	main	INFO	MyClass	INFO message
0	main	WARN	MyClass	WARN message
15	main	ERROR	MyClass	ERROR message
15	main	FATAL	MyClass	FATAL message

Obr. 2 Log soubor output.html zobrazený ve webovém prohlížeči



Obr. 3 Vývojové prostředí Eclipse, přesměrování standardního výstupu

Soubor 1: Log soubor output.html:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Log4J Log Messages</title>
<style type="text/css">
<!--
body, table {font-family: arial,sans-serif; font-size: x-small;}
th {background: #336699; color: #FFFFFF; text-align: left;}
-->
</style>
</head>
<body bgcolor="#FFFFFF" topmargin="6" leftmargin="6">
<hr size="1" noshade>
Log session start time Mon Jun 01 05:44:48 CEST 2009<br>
<br>
<table cellspacing="0" cellpadding="4" border="1"
bordercolor="#224466" width="100%">
<tr>
<th>Time</th>
<th>Thread</th>
<th>Level</th>
<th>Category</th>
```

```
<th>Message</th>
</tr>

<tr>
<td>0</td>
<td title="main thread">main</td>
<td title="Level"><font color="#339933">DEBUG</font></td>
<td title="MyClass category">MyClass</td>
<td title="Message">DEBUG message</td>
</tr>

<tr>
<td>47</td>
<td title="main thread">main</td>
<td title="Level">INFO</td>
<td title="MyClass category">MyClass</td>
<td title="Message">INFO message</td>
</tr>

<tr>
<td>47</td>
<td title="main thread">main</td>
<td title="Level"><font
color="#993300"><strong>WARN</strong></font></td>
<td title="MyClass category">MyClass</td>
<td title="Message">WARN message</td>
</tr>

<tr>
<td>47</td>
<td title="main thread">main</td>
<td title="Level"><font
color="#993300"><strong>ERROR</strong></font></td>
<td title="MyClass category">MyClass</td>
<td title="Message">ERROR message</td>
</tr>

<tr>
<td>47</td>
<td title="main thread">main</td>
<td title="Level"><font
color="#993300"><strong>FATAL</strong></font></td>
<td title="MyClass category">MyClass</td>
<td title="Message">FATAL message</td>
</tr>
```

Soubor 2: Standardní vystup (MyClass.log):

[29.05.2009 05:58:46,125] [DEBUG] [MyClass.main(MyClass.java:34)] DEBUG message

[29.05.2009 05:58:46,125] [INFO] [MyClass.main(MyClass.java:35)] INFO message

[29.05.2009 05:58:46,125] [WARN] [MyClass.main(MyClass.java:36)] WARN message

[29.05.2009 05:58:46,140] [ERROR] [MyClass.main(MyClass.java:37)] ERROR message

[29.05.2009 05:58:46,140] [FATAL] [MyClass.main(MyClass.java:38)] FATAL message

Log soubory obsahují dle očekávání hlášky všech logovaných úrovní, jelikož úroveň loggeru byla nastavena na `DEBUG`.

5.3 Rozhraní Appender

Rozhraní `org.apache.log4j.Appender`. Toto rozhraní implementují všechny appendery (tedy přesněji řečeno jsou poděděny od abstraktní třídy `org.apache.log4j.AppenderSkeleton`, která toto rozhraní implementuje. Popis obecné funkce appenderů byl popsán v odstavcích 4.1 a 4.2. Jen krátce zopakuji, že objekty tohoto typu slouží pro určení výstupní destinace pro logované události.

5.3.1 Třída AppenderSkeleton

Abstraktní třída, jež je rodičovská pro ostatní appendery v tomto balíku. Třída poskytuje kód pro běžnou funkci, jako je podpora pro treshold filtrování a podpora pro obecné filtry.

5.3.2 Třída AsyncAdapter

Tento adapter dovoluje logovat události asynchronně. Sbírá veškeré události, které mu byly zaslány a po naplnění bufferu je odešle všem appenderům, jež jsou na něj napojeny. Pro obsluhu událostí, které udržuje v bufferu, používá separátní vlákno.

Tento appender lze použít pouze za předpokladu, že bude nastaven konfiguračním skriptem, a obsah tohoto souboru bude načten pomocí třídy `DOMConfigurator`.

5.3.3 Třída ConsoleAppender

`ConsoleAppender` zapisuje logované události na standardní výstup `System.out` či chybový výstup `Systém.err` s použitím specifikovaného layoutu. Tento appender patří mezi nejběžněji používané.

5.3.4 Třída FileAppender

Tento appender slouží pro zápis logů do souboru. Jeho přímými potomky jsou třídy `RollingFileAppender` a `DailyRollingFileAppender`. Tento appender a zvláště jeho potomci, patří též mezi hojně využívané.

5.3.5 Třída RollingFileAppender

Tento appender je rozšířením předchozího. Ukládá do souboru, který je automaticky zálohován, dosáhne-li nastavené velikosti. Nastavit lze i počet zálohovaných souborů. Defaultně je nastaven na jeden soubor. Nastavíme-li maximální počet souborů na 0, nebude vytvářet zálohy souboru a po naplnění souboru do jeho maximální velikosti, bude soubor ořezáván.

5.3.6 ExternallyRolledFileAppender

Tento appender je potomkem třídy `RollingFileAppender`. Naslouchá na socketu na specifikovaném portu (property Port). Čeká na zprávu typu "RollOver". Je-li přijata zpráva tohoto typu, je podléhající log soubor překlopen a zpět (do procesu, který překlopení inicializoval) je zaslána potvrzující zpráva.

Výhodou jest nezávislost na použitém operačním systému, rychlost a spolehlivost. Nevýhodou je, že takováto inicializace pro překlopení log souboru, není žádným způsobem autentifikována – při použití na produkčním prostředí se doporučuje vytvořit nějako formu ochrany, aby nedocházelo k nežádoucím překlopením log souborů.

5.3.7 Třída DailyRollingFileAppender

DailyRollingFileAppender je potomkem třídy FileAppender. Podléhající log soubor je zálohován s frekvencí zadanou uživatelem. Plán zálohování je specifikován pomocí volby DatePattern. Tento pattern může používat konvence třídy `SimpleDateFormat`.

Příklad: soubor je nastaven na /foo/bar.log a DatePattern na `'.'yyyy-MM-dd`, dne 11.12.2009 o půlnoci se logovací soubor `/foo/bar.log` nakopíruje do adresáře `/foo/bar.log.2001-12-11` a logování pro 12.12. bude pokračovat v `/foo/bar.log` až do jeho dalšího překlopení následující den.

DatePattern	Čas překlopení
'.'yyyy-MM	Překlopení na začátku každého měsíce.
'.'yyyy-ww	Překlopení v první den každého týdne. Tento den záleží na lokalizaci.
'.'yyyy-MM-dd	Překlopení každý den o půlnoci.
'.'yyyy-MM-dd-a	Překlopení každý den o půlnoci a o poledni.
'.'yyyy-MM-dd-HH	Překlopení na sklonku každé hodiny
'.'yyyy-MM-dd-HH-mm	Překlopení na začátku každé minuty.

Tabulka 5: Přehled použitelných patternů

5.3.8 Třída JDBCAppender

Tento appender posílá logované události do databáze. Každá událost je uložena do `ArrayList` bufferu. Jakmile je buffer naplněn, každá logovaná událost je vložena do konfigurovatelného SQL příkazu a ten je vykonán. Mezi konfigurovatelné vlastnosti patří: velikost bufferu, db URL, uživatel a heslo – lze je nakonfigurovat standardní cestou Log4j. Metoda `setSql(String sql)` nastavuje SQL příkaz, jenž použit. Tento příkaz je poslán do layoutu PatternLayout (nezáleže na tom, zda byl vytvořen automaticky či ručně). SQL příkaz může obsahovat veškeré Conversion Patterns.

5.3.9 Třída JMSAppender

Java Message Service (JMS) API je standard pro zasílání zpráv, který povoluje aplikačním komponentám založeným na platformě Java 2 a J2EE vytvářet, posílat, příjmat a číst zprávy (mezi sebou).

Tento jednoduchý appender publikuje události jako JMS témata (slouží k řízení toku od vydavatelů k odběratelům). Události jsou serializovány a přenášeny jako JMS zprávy typu `ObjectMessage`.

5.3.10 Třída LF5Appender

Tento appender loguje události do swingové logovací konzole.

5.3.11 Třída NTEventLogAppender

Loguje do logu systémových událostí operačního systému Windows.

5.3.12 Třída NullAppender

Tento appender neloguje na žádné zařízení.

5.3.13 Třída SMTPAppender

Appender SMTPAppender posílá e-mail, dojde-li ke specifické logované události, typicky nastanou-li chyby úrovně `ERROR` či vyšší (`FATAL`). Počet událostí, doručených e-mailem se nastaví pomocí volby BufferSize. Appender drží maximálně BufferSize posledních logovaných událostí v cyklickém bufferu. Tento způsob zajišťuje, že míra paměťových požadavků zůstává na rozumné míře, přestože je logování např. v nepřetržitém provozu a zasílání zpráv o událostech je stále funkční.

5.3.14 Třída SocketAppender

Tento appender odesílá logované události na vzdálený logovací server. Appender nepoužívá layouty. Posílají na stranu serveru serializované `LoggingEvent` objekty.

Vzdálené logování používá protokolu TCP. V důsledku toho, je-li server dostupný, dorazí na něj logované události. Pokud dojde k pádu serveru, logovací požadavky jsou jednoduše zahozeny. Pokud poté dojde k naběhnutí serveru, přenos pokračuje posledními nedoručenými událostmi. Čas mezi pádem serveru a opětovným připojením, minimalizuje vlákno connector, které se periodicky pokouší na server připojit.

Logované události jsou automaticky zaznamenávány nativní TCP implementací. Je-li linka k serveru pomalejší, než rychlost produkce událostí, klient může pouze zvýšit network rate.

5.3.15 Třída SocketHubAppender

Chová se podobně jako `SocketAppender`. Rozdíl je v připojení na vzdálený logovací server. SocketHubAppender se chová ke vzdáleným serverům jako ke klientům. Může udržovat více než jedno připojení. Přijme-li logovanou událost je předána celé, v tu chvíli připojené skupině logovacích serverů. Tyto vzdálené servery se jednoduše připojí k `SocketHubAppenderu` jako hosté (není zapotřebí žádné další úpravy v nastavení).

5.3.16 Třída SyslogAppender

Tento appender lze využít k posílání zpráv na vzdálený syslog server.

5.3.17 Třída TelnetAppender

Appender, který se specializuje na zápis do socketů, jež jsou určeny pouze ke čtení.

5.3.18 Třída WriterAppender

Zapisuje logované události na `Writer` či `OutputStream`.

5.4 Třída Layout

Objekty třídy `Layout` složí k formátování logovaných zpráv. Formát logovacího souboru je velmi důležitý. Jeho volbě by se měla věnovat zvýšená pozornost. Rozhodnutí, který formát vybrat, bude záviset též na způsobu, jakým budeme v budoucnosti logy analyzovat. Formát, který bude vhodný pro prohlížení pouhý okem, nebude vhodný ke strojovému zpracování a naopak. Log4j nabízí 4 třídy, pomocí nichž můžeme logované události a zprávy formátovat.

5.4.1 Třída SimpleLayout

`SimpleLayout` je nejjednodušším z uvedených layoutů. Skládá se z úrovně logované události, následované pomlčkou a samotné zprávy.
Příklad:

DEBUG – zpráva aplikace

5.4.2 Třída HTMLLayout

HtmlLayout generuje složitější strukturu logovaného souboru než předchozí. Události formátuje do HTML tabulky. Obsahují-li události znaky jiné než ASCII, může dojít k porušení struktury generované tabulky. Příklad výstupu tohoto layoutu lze vidět v bodě 5.2 (Soubor 1, str.26).

5.4.3 Pattern Layout

Flexibilní layout, konfigurovatelný pomocí textového patternu. Cílem této třídy je zformátování objektu typu LoggingEvent a vrácení výsledku jakožto textového řetězce. Výsledky závisí na volbě conversion pattern (konverzní vzor). Conversion pattern je poskládán ze znaků, jež se nazývají conversion specifiers (konverzní specifikátory). Conversion pattern je možné doplnit vlastním textem.

Každý konverzní specifikátor začíná znakem %, který je následován volitelnými modifikátory formátu a konverzními znaky (např. %-5p). Není zde žádný explicitní separátor, jenž by separoval text od konverzního vzoru. Pattern parser pozná, kdy dospěl na konec konverzního specifikátoru nebo čte konverzní charakter

Popis konverzních znaků:

C	Vypíše celé jméno (včetně cesty) třídy, jež vyvolala logovací požadavek. Tento konverzní specifikátor může být volitelně následován kladným celým číslem ve složených závorkách C{1}. Číslo v závorkách určuje počet úrovní, se kterým se jméno třídy vygeneruje. Je-li plně kvalifikované jméno třídy *cz.tul.Trida*, pak C{i} vrací: i=1 *Trida*, i=2 *tul.Trida*, i>=3 *cz.tul.Trida*. Pro i<=0 dojde k chybě.
d	Vypíše čas logované události. Za specifikátorem **d** může být ve složených závorkách upřesněn jeho formát. Není-li formát uveden, použije se defaultní ISO8601 (`2009-05-29 21:18:20,109`). Popis upřesňujících specifikátorů: d = den v měsíci, D = den v roce, m = minuty, M = měsíc, s = sekundy, S = tisíciny sekund, h = hodiny ve formátu 12h, H = hodiny ve formátu 24h. Tyto lze v {} libovolně kombinovat. Například %d{HH:mm:ss,SSS} nebo %d{dd MMM yyyy HH:mm:ss,SSS}. Syntaxe je stejná jako u SimpleDateFormat. Dalším způsobem je umístit do {} jeden z následujících stringů: "ABSOLUTE", "DATE", "ISO8601".
F	Vygeneruje jméno celého souboru dané třídy i s příponou .java.
l	Vypíše informace o místě, odkud byla logovací metoda zavolána. Tyto informace závisí na implementaci JVM. Běžně se skládají z plně kvalifikovaného jména třídy a čísla řádku. Výstup může vypadat následovně: `pckg.asd.MyClass.main(MyClass.java:46)`.
L	Vypíše číslo řádku, kde byla zavolána logovací metoda
m	Vypíše zprávu, která je asociována s logovanou událostí.
M	Vypíše jméno metody, z níž byla zavolána logovací funkce.
n	Vypíše platformově závislý znak (znaky) pro nový řádek. V patternu lze namísto tohoto konverzního znaku použít klasických "\n" nebo "\r\n".
p	Úroveň logované události.
r	Čas v ms od doby vytvoření instance layoutu do doby, kdy byla vytvořena logovaná událost.

t	Vypíše jméno vlákna, v němž byla událost zalogována.
x	Vypíše NDC (nested diagnostic context) asociovaný s vláknem, v němž byla vygenerována logovaná událost.
X	Vypíše MDC (mapped diagnostic context) asociovaný s vláknem, v němž byla vygenerována logovaná událost. Tento konverzní znak musí být následován kličem k mapě uvnitř {}. Hodnota uvnitř MDC korespondující s klíčem bude výstupem (%X{clientNumber}).
%	Vypsání % se provádí pomocí sekvence %%.

Pomocí formátovacích modifikátorů je možné měnit minimální a maximální šířku pole a zarovnání. Tyto volitelné modifikátory se umisťují mezi % a konverzní znak. Probereme je směrem z leva.

- Prvním volitelným modifikátorem je zarovnání textu. Je-li znaménko - za %, je text zarovnán vlevo, není-li použit, tak vpravo.
- Dále může obsahovat celé číslo, které určuje minimální délku textu. Je-li text delší, bude doplněn mezerami z druhé strany, než na kterou je zarovnán. V případě, že je text delší než nastavené minimum, zobrazí se celý.
- Posledním volitelným modifikátorem je maximum. Definuje se celým číslem, kterému předchází tečka. Je-li text delší než toto maximum, ořízne se.

Příklad konverzního vzoru:
%-20.30m – text zprávy události je zarovnán doleva. Jeho minimální délka je 20 znaků. V případě, že bude zpráva kratší, bude text doplněn mezerami, a to zprava. Maximum zprávy je nastaveno na 30 znaků.

5.4.4 XML Layout

Výstup tohoto layoutu se skládá pouze z elementů log4j:event, jenž jsou definovány v log4j.dtd. Tento výstup tedy není plnohodnotným XML. Takto zaznamenané události mohou vypadat následovně:

```
<log4j:event logger="asd.MyClass" timestamp="1243893514421"
level="INFO" thread="main">
      <log4j:message><![CDATA[INFO message]]></log4j:message>
</log4j:event>
```

Hlavní výhodou `XMLLayoutu` je pevně daná struktura generovaného souboru, díky které lze záznamy snadno (a hlavně spolehlivě) strojově zpracovat.

5.5 Nastavení log4j

Logování lze nastavit dvěma základními způsoby, které popíši v následujících dvou odstavcích.

5.5.1 Nastavení ve zdrojovém kódu aplikace

Prvním a méně vhodným z nich, je nastavení logování ve zdrojovém kódu aplikace. Tento způsob není dostatečně pružný. Budeme-li chtít měnit nastavení, musíme udělat změny v kódu. Poté projekt znovu vyexportovat a nasadit novou verzi.

5.5.2 Nastavení v konfiguračním souboru

Lepším způsobem je nastavení logování pomocí konfiguračního souboru. V aplikaci pak stačí navíc jeden příkaz, jenž nastavení z tohoto souboru načte a logování běží.

Jestliže zdůrazňujeme, jak užitečné je mít možnost měnit nastavení z vnějšku aplikace, nesmíme opomenout umístění konfiguračního souboru. Výhodné bude soubor vyjmout z hierarchie aplikace. Není sice až zas takovým problémem vyexportovaný projekt editovat (WAR, EAR, JAR), avšak této možnosti se nám povětšinou nedostane. Často situace vypadá tak, že se vyexportovaný projekt nasadí na testovací prostředí. Po otestovaní jde tentýž soubor na prostředí produkční, a to přesně v takovém stavu, v jakém byl prostředí testovacím.

Log4j podporuje dva druhy konfiguračních souborů – properties a XML. V praxi více používaným, je properties soubor. Jeho oblíbenost tkví v jednodušší struktuře a lepší čitelnosti.

Ukázka XML souboru:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
```

```
  <appender name="ConsoleAppender"
            class="org.apache.log4j.ConsoleAppender">
      <layout class="org.apache.log4j.SimpleLayout"/>
  </appender>

  <root>
      <priority value ="debug" />
      <appender-ref ref="ConsoleAppender"/>
  </root>
</log4j:configuration>
```

Totéž nastavení pomocí properties souboru:

```
log4j.rootLogger=DEBUG, rootLogFile
log4j.appender.rootLogFile=org.apache.log4j.ConsoleAppender
log4j.appender.rootLogFile.layout=org.apache.log4j.SimpleLayout
```

Více ukázek nastavení Log4j lze nalézt na CD v adresáři aplikace.

5.6 Zpracování logů

Logy lze prohlížet za pomocí různých softwarových nástrojů. Volba nástroje pro prohlížení závisí na layoutu, který má appender nastaven.

Pro `SimpleLayout` a `PatternLayout` je nejlepší volbou textový editor. Výstup zformátovaný pomocí `HTMLLayoutu` lze pohodlně studovat ve webovém prohlížeči. Pro XMLLayout má Log4j vlastní velmi vydařený grafický nástroj CHAINSAW. Události zobrazí v přehledné tabulce:

ID	Timestamp	Level	Logger	Message	Thread	applicat...	hostname
52	2009-06-03 06:07:14,156		org.apache.log4j.chainsaw.ChainsawCyclicBufferTableModel	request to sort col=7	Chainsaw-WorkerThread	log	chainsaw
1	2009-06-03 06:05:41,125		org.apache.log4j.chainsaw.LogUI	Using '' for auto-configuration	AWT-EventQueue-0	log	chainsaw
2	2009-06-03 06:05:41,500		org.apache.log4j.chainsaw.help.HelpManager	Could not find any local JavaDocs, you might want...	AWT-EventQueue-0	log	chainsaw
3	2009-06-03 06:05:41,859		org.apache.log4j.chainsaw.plugins.PluginClassLoaderFactory	pluginDirectory cannot be null, and it must exist a...	AWT-EventQueue-0	log	chainsaw
4	2009-06-03 06:05:41,890		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Located known Receiver class org.apache.log4j.n...	AWT-EventQueue-0	log	chainsaw
5	2009-06-03 06:05:41,890		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Located known Receiver class org.apache.log4j.n...	AWT-EventQueue-0	log	chainsaw
6	2009-06-03 06:05:41,890		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Located known Receiver class org.apache.log4j.n...	AWT-EventQueue-0	log	chainsaw
7	2009-06-03 06:05:41,890		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Located known Receiver class org.apache.log4j.n...	AWT-EventQueue-0	log	chainsaw
8	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Located known Receiver class org.apache.log4j.n...	AWT-EventQueue-0	log	chainsaw
9	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Failed to locate Receiver class:org.apache.log4j.n...	AWT-EventQueue-0	log	chainsaw
10	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Failed to locate Receiver class:org.apache.log4j.d...	AWT-EventQueue-0	log	chainsaw
11	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Failed to locate Receiver class:org.apache.log4j.d...	AWT-EventQueue-0	log	chainsaw
12	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Located known Receiver class org.apache.log4j.v...	AWT-EventQueue-0	log	chainsaw
13	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Failed to locate Receiver class:org.apache.log4j.x...	AWT-EventQueue-0	log	chainsaw
14	2009-06-03 06:05:41,906		org.apache.log4j.chainsaw.receivers.ReceiversHelper	Failed to locate Receiver class:org.apache.log4j.c...	AWT-EventQueue-0	log	chainsaw
15	2009-06-03 06:05:42,671		org.apache.log4j.chainsaw.ApplicationPreferenceModelPanel	Can't find new GTK L&F, might be Windows, or <...	AWT-EventQueue-0	log	chainsaw
16	2009-06-03 06:05:42,703		org.apache.log4j.chainsaw.osx.OSXIntegration	OSXIntegration.init() called	AWT-EventQueue-0	log	chainsaw
17	2009-06-03 06:05:42,703		org.apache.log4j.chainsaw.osx.OSXIntegration	Not OSX, ignoring...	AWT-EventQueue-0	log	chainsaw
18	2009-06-03 06:05:42,765		org.apache.log4j.chainsaw.messages.MessageCenter	Zeroconf started!	Chainsaw-WorkerThread	log	chainsaw
19	2009-06-03 06:05:42,859		org.apache.log4j.chainsaw.zeroconf.ZeroConfPlugin	menu 'Connect to' was NOT added because the '...	Chainsaw-WorkerThread	log	chainsaw
20	2009-06-03 06:05:42,875		org.apache.log4j.chainsaw.messages.MessageCenter	Looks like ZeroConf stuff is available... WooHoo!	Chainsaw-WorkerThread	log	chainsaw
21	2009-06-03 06:05:43,156		org.apache.log4j.chainsaw.LogUI	waiting for initialization to complete	Chainsaw-WorkerThread	log	chainsaw

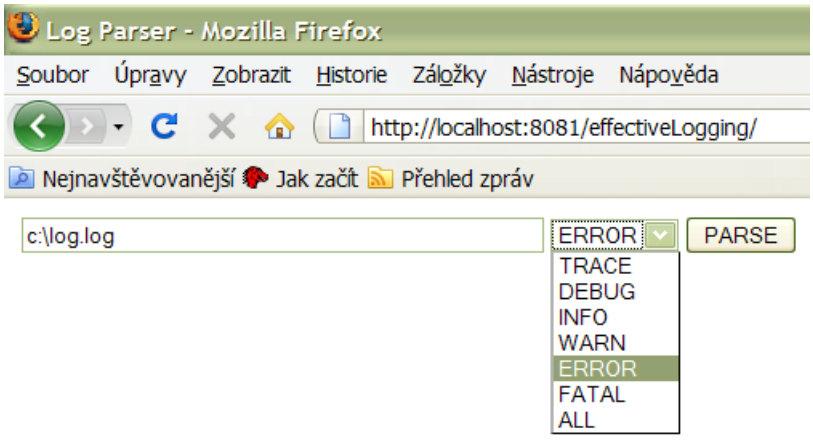
Tabulka 6: Ukázka z aplikace Chainsaw

6 Aplikace

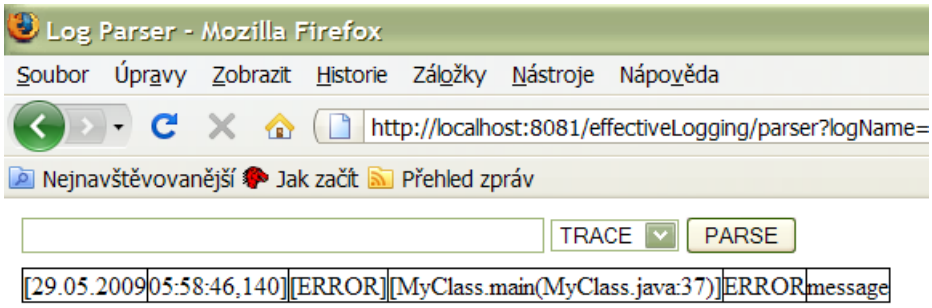
Vzhledem ke špatnému časovému odhadu nezbylo mnoho místa pro vytvoření aspoň trochu zajímavější aplikace. Přesto jsem se snažil dostát zadání a zadání z části implementovat. Rozhodl jsem se pro jednoduchou webovou aplikaci, používající technologii servetů.

Aplikace loguje pomocí knihovny Log4j. Přestože není nijak rozsáhlá, loguje události dle úrovně, do které spadají (z pohledu efektivního logování). Část aplikace, jež parsuje a zobrazuje logy, formátované `SimpleLayoutem` a `PatternLayoutem`, není dotažena do uspokojivého stavu. Avšak poskytuje alespoň názorný důkaz, že tyto (zejména složitější `PatternLayout`) nejsou vhodné ke strojovému zpracování. Tuto část aplikace bych při dostatku času zřejmě řešil tak, že mezi vstupy pro parsování by byl i konverzní vzor.

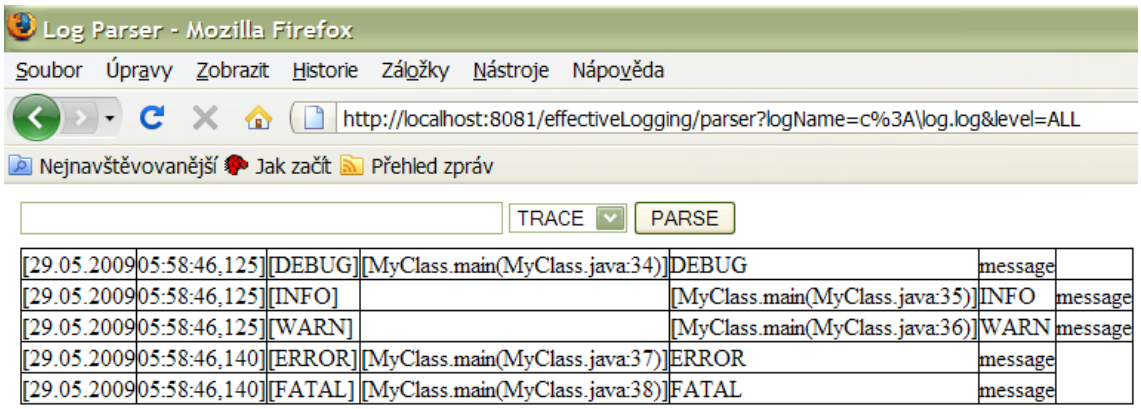
Screenshoty aplikace:



Obr. 4 Zadání umístění logu, váběr zobrazované úrovně



Obr. 5 Výsledek při výběru úrovně ERROR (tato hodnota není držena paměti, zobrazená TRACE je implicitní)



Obr. 6 Výsledek při výběru úrovně ALL

Aplikace funguje tak, že zobrazí události vybrané úrovně (logika je tedy jiná, než při zapisování do logu).

Pro vývoj aplikace bylo použito vývojového prostředí Eclipse a serveru Apache Tomcat 6.0. Programovací prostředky, použité knihovny a WAR aplikace budou přiloženy na CD.

7 Závěr

Vyhodnocování výsledků jsem se snažil věnovat průběžně a tedy mnou zajímavá zjištění, již byla popsána v předchozích kapitolách. Původně jsem se chtěl ještě zmínit o možnostech AOP (aspektově orientované programování), jehož lze zajímavým způsobem využít při logovaní v aplikacích.

Jinak se domnívám, že jsem z převážné části dostál požadavkům zadání a doufám, že potencionálnímu čtenáři nabídne nejen ucelený přehled o možnostech logování v aplikacích napsaných v jazyku Java, ale že z textu pochopí i jak pracovat s nástrojem Log4j.

8 Použitá literatura

Tištěné zdroje:

- HEROUT, P., *Učebnice jazyka JAVA*. 2005. ISBN 80-7232-115-3

Internetové zdroje:

- Sun developer network. URL: java.sun.com.
- Manuál log4j. URL: logging.apache.org/log4j/1.2/manual.html.
- Nastavení log4j. URL: docs.google.com/Doc?id=ddhf5h22_56gqgv47gz.
- Barry Ruzek. Effective Java Exceptions.
  URL: moai.tistory.com/attachment/dk180000000000.pdf.
- Petr Brabec. Vlastnosti technologie Java.
  URL: http://www.publish.cz/pczone/?id=147 .
- Roman Pichlík. Úvod do problematiky výjimek.
  URL: http://interval.cz/clanky/java-a-vyjimky-uvod-do-problematiky-vyjimek.
- Algoritmizace a programování. URL:
  http://www.fm.tul.cz/prg/ALP2/Prednasky/ALP2_T03_BalikyVyjimky.pdf.
- Google. Example Log4j Configuration File.
  URL: http://www.java2s.com/Code/Java/Language-Basics/Examplelog4jConfigurationFile.htm.
- Saveen. Using Log4j in Spring – Effective Logging.
  URL: http://javaandjava.blogspot.com/2008/12/using-log4j-in-spring-effective-logging.html.
- Rose Thornton. Logging in Java applications.
  URL: http://www.developer.com/java/other/article.php/1404951.
- Debugger. URL: http://cs.wikipedia.org/wiki/Debugger.
- Ceki Gülcü. Think again efore adopting the commonc-logging API.
  URL: http://www.qos.ch/logging/thinkAgain.jsp.
- Java Logging Framework.
  URL: http://en.wikipedia.org/wiki/Java_Logging_Frameworks.

- Joe Mcnamara. Log4j vs. java.util.logging. URL: http://java.sys-con.com/node/48541.
- Sun Microsystems. Java logging overview. URL: http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html
- Rostislav Behan. Log4j Logovací API pro aplikační programátory. URL: http://nb.vse.cz/~ZELENYJ/it380/eseje/xbehr02/index.htm.
- Pavel Klobasa. Logujeme s přehledem. URL: http://vyvojari.oxyonline.cz/log4j-logujeme-s-prehledem.
- Log4j. http://en.wikipedia.org/wiki/Log4J.