VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Framework pro analýzu logů Java aplikací

Framework for Java application log files

2014 Martin Bayer

Zadání bakalářské práce

Jiří Dvorský

Ukázka sazby diplomové nebo bakalářské práce

Diploma Thesis Typesetting Demo

+++

Podpis vedoucího katedry

+++

Podpis děkana fakulty

Tohle je druhá strana zadání diplomové práce.

Konec strany

Tohle je třetí strana zadání diplomové práce.

Konec strany

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

Zde vložte text dohodnutého omezení přístupu k Vaší práci, chránící například firemní know-how. Zde vložte text dohodnutého omezení přístupu k Vaší práce, chránící například firemní know-how. A zavazujete se, že

1. o práci nikomu neřeknete,
2. po obhajobě na ni zapomenete a
3. budete popírat její existenci.

A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. A ještě jeden důležitý odstavec. Konec textu dohodnutého omezení přístupu k Vaší práci.

V Ostravě 16. dubna 2009 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 16. dubna 2009 .....

Rád bych na tomto místě poděkovala všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Abstrakt

Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt. Tohle je nějaký abstrakt.

Klíčová slova: typografie, LaTeX, diplomová práce

Abstract

This is English abstract. This is English abstract. This is English abstract. This is English abstract. This is English abstract. This is English abstract.

Keywords: typography, LaTeX, master thesis

Seznam použitých zkratek a symbolů

RCP	–	Rich client platform

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Vývoj softwaru ve 21. století se stává oblastí trhu, kde se čím dál více projevuje konkurenční boj společností o získávání zakázek a jejich úspěšné dokončení s co největší marží. Samotné vypracování zadání, jeho prvotní implementace, testování a nasazení na prostředí zákazníka však nejsou hlavními částmi životního cyklu aplikací. Z důvodu snížených investic do nových projektů se do popředí dostává část, kdy se aplikace udržuje a spravuje tak, aby i dříve vyrobené programy mohly úspěšně plnit požadavky uživatelů na funkčnost, bezpečnost i podporu uživatelských operací. K dosažení tohoto cíle byly vyvinuty frameworky, které jsou používány k záznamu činnosti softwaru v čase tak, aby byly případné chyby lépe identifikovatelné, analytitk a programátor měl co nejvíce informací použitelných k nalezení příčiny a její nápravě. Tímto postupem se může časová náročnost opravy rapidně snížit, což vede přímo k úspoře zdrojů potřebných ke správě systému.

Počet aplikací, které fungují na straně zákazníka i několik desítek let je nezanedbatelný. Se snižováním rozpočtů se bude tento počet pravděpodobně ještě zvětšovat. Proto se budou rozšiřovat i potřeby výrobců software ve vztahu k rychlejší a efektivnější správě aplikací. Aby toho mohlo býti dosáhnuto, je třeba rozšiřovat funkcionalitu týkající se logování tak, aby bylo dostatečně efektivní a zároveň, aby nezahlcovalo vývojáře zbytečnými nebo nepřehlednými informacemi. I při dodržení těchto zásad budou logy středních a velkých aplikací obsahovat deseti tisíce záznamů v případě serverových aplikací a tisíce záznamů v případě aplikací klientských. Jedná se o poměrně velké množství informací, které musí vývojář zpracovat. Většinu prováděných operací vedoucích ke snížení počtu dat potřebných pro analýzu, jako je vyhledávání, filtrování a seskupování užitečných informací, lze automatizovat.

Cílem mé diplomové práce bude návrh a implementace frameworku, který bude sloužit právě k automatizaci běžných úkonů při zpracování log záznamů z aplikací. Primárně se zaměřím na aplikace napsané v jazyku Java a používající k logování jeden z běžných frameworků. Aplikace bude rozšiřitelná tak, aby si byl každý analytik nebo vývojář schopen vyrobit vlastní nový kus funkcionality ve formě plug-inu a ten nainstalovat do aplikace. Plug-iny budou moci býti koncentrované na sdíleném úložišti, čímž bude docíleno aktuálnosti funkcionality napříč všemi uživateli v rámci vývojového týmu. Struktura pluginu bude jednoduchá, aby i méně zkušený programátor mohl vytvořit svůj plug-in a používat ho. K tvorbě frameworku použiji technologii Eclipse 4, která sama o sobě podporuje potřebnou architekturu.

V teoretické části diplomové práce se budu věnovat popisu samotné tvorby log záznamů. Rozeberu motivaci k použití logování včetně existujících přístupů k její realizaci. Dále se budu zabývat popisem tří frameworků používaných k zajištění logování Java aplikací, jejich srovnání a důkladnějšímu popisu nejsofistikovanějšího z nich.

Dále představím technologie použité při implementaci frameworku. V krátkosti zmíním historii i předpokládaný vývoj použité platformy. Vysvětlím její hlavní vlastnosti a přednosti pro vývoj rozšiřitelné architektury. Zastavím se u frameworku použitého pro tvorbu uživatelského rozhraní a vyzdvihnu jeho negativa a pozitiva.

V části praktické popíši hlavní principy a myšlenky celého vytvářeného frameworku včetně použitého klíčového návrhového vzoru. Vysvětlím jednotlivé vrstvy architektury aplikace. Dále se budu věnovat návrhu a realizaci prvků uživatelského rozhraní, vlastních i generických komponent.

Jednotlivé funkční jednotky(procesory) celého frameworku je možno rozdělit do několika skupin. Vyjasním použití těchto skupin i důvod rozdělení včetně popisu základních komponent, jejich konfigurace a možností použití. Pro vztahy mezi jednotkami základních skupin existují logická pravidla, jejichž dodržení je přímo implementováno tak, aby uživatel nebyl schopen tato pravidla porušit. Na krátkých příkladech vyobrazím důvody omezení závislostí. Další funkcionalitou podporující funkci jednotlivých plug-inů se budu zabývat v další části. Samostatně vysvětlím i zpracování vstupních souborů, především přístup, pomocí kterého jsem problém řešil.

V části týkající se implementace uvedu také postup, pomocí kterého bude možno vytvořit samostatný procesor použitelný v aplikaci. Součástí bude i instalace a případná aktualizace plug-inu pomocí použité platformy.

Celou funkcionalitu navrženého frameworku následně otestuji na modelové situaci. Provedu implementaci a instalaci přídavných procesorů. Uvedu výsledky jednotlivých částí procesu analýzy logů, délku průběhu a srovnání s manuálním způsobem analýzy. Celý proces zpracovávání podrobím performance testování nástrojem Yourkit Java Profiler.

V poslední části diplomové práce se zamyslím nad dalším vývojem užitečné funkcionality, která by mohla být později dotvořena a zmíním nástroje, které jsem při vývoji používal, důvody i způsoby jejich použití.

2 Popis způsobu logování a používané frameworky

Logování používané v dnešních serverových i klientských aplikacích není ve většině případů implementováno samotnými vývojáři aplikace. Výhodou této oblasti je, že již existují řešení, která jsou odladěna, důkladně testována a především je brán ohled na zdroje tak, aby nedocházelo ke zbytečnému zpomalení chodu aplikací.Právě výkon je jednou z hlavních výhod frameworků implementujících logování před způsobem používaným často vývojáři při ladění funkcionality, jímž je výpis do konzole.

2.1 Logování v aplikacích

Logovací frameworky dovolují programátorům logovat jakékoliv informace na různých úrovních, které by měly logicky odpovídat zpracovávané informaci. Jedná-li se například o chybu, je vhodné použít úroveň "ERROR"apod. Najít rovnováhu mezi množstvím a užitečností zaznamenaných událostí je velmi podstatné z důvodu přehlednosti vytvořených záznamů. Úroveň zaznamenávání údajů je třeba používat i u operací, které nastávají často. Je možno pro ně vytvořit zvláštní soubory, do kterých budou zaznamenány jen některé aktivity.

2.2 Proč se logování používá

Zaznamenávání údajů o běhu aplikace, vyvolaných událostech i chybách je užitečné u velkých a překvapivě i u malých aplikací. Pokud je dobře použito, zjednodušuje analýzu chybného chování, událostí vyvolaných uživatelem nebo změn dat v čase. Usnadňuje vývoj nové funkcionality a ladění nedostatků, které nebyly včas odhaleny během testování. Pokud je zákazníkem nalezena chyba, kterou není jednoduché zopakovat z důvodu rozdílnosti dat nebo prostředí, na kterém jsou aplikace spuštěny, je rozšíření logování jednou z mála a zároveň nejúčinnějších způsobů, jak chybu v chování odhalit a následně opravit.

V některých případech se mezi jednotlivými verzemi softwaru objevují tzv. regresní chyby, vznikající implementací nové funkcionality nebo opravami jiných chyb. V tomto případě bývá užitečné porovnání zaznamenaných informací mezi předešlou a aktuální verzi. Můžeme takto zjistit, zda byly volány příslušné metody, k jakým změnám dat došlo apod. Motivací pro používání logování je tedy to, aby bylo možné analyzovat chyby a události v aplikaci rychle a účelně tam, kde se skutečně nacházejí.

2.3 Jak se logování používá

Pro obecné použití logování se dají aplikovat některé postupy vycházející ze zkušeností s vývojem a údržbou softwaru v minulosti. Po nasazení softwaru u zákazníka se postupně odhalují problematické části, u kterých je potřeba logování rozšířit. Pro vytvoření instance loggeru se v aplikacích nejčastěji používá návrhový vzor Factory. Tento způsob je nejflexibilnější možný, protože dovoluje měnit framework použitý k logování beze změn ve zdrojovém kódu. V samotné aplikaci už se poté volá nad danou instancí metoda, pomocí

které bude záznam vytvořen. Obvykle jsou tyto metody pojmenovány podle požadované úrovně logování.

V aplikacích se logují různé informace na různých úrovních. Je velmi užitečné znát nejpoužívanější úrovně a jejich použití:

TRACE - jsou zde obsaženy především detailní informace o stavu objektu nebo aplikace. Většinou se jedná o delší záznam. Záznamy by mělý být obsaženy pouze v log souborech.

DEBUG - detailní informace o běhu aplikace. Je užitečné používat, pokud vývojáři potřebují vědět, jaké objekty byly vytvořeny, které metody volány atd. Neměly by být zapisováni jinam než do souborů.

INFO - podstatné informace o běhu aplikace. Spouštění aplikace jejich služeb, vypínání, připojení externích zařízení apod. Obvykle jsou tyto záznamy vypisovány do konzole, nemělo by jich tedy být zbytečně moc.

WARNING - pomocí této úrovně se zaznamenávají události, které sice přímo nevedou k chybám aplikace, ale dané chování není standardní. Dále se zde mohou objevit informace o používání zastaralých knihoven apod.

ERROR - chyby, které způsobují chybný běh aplikace nebo nepředpokládané podmínky pro její běh.

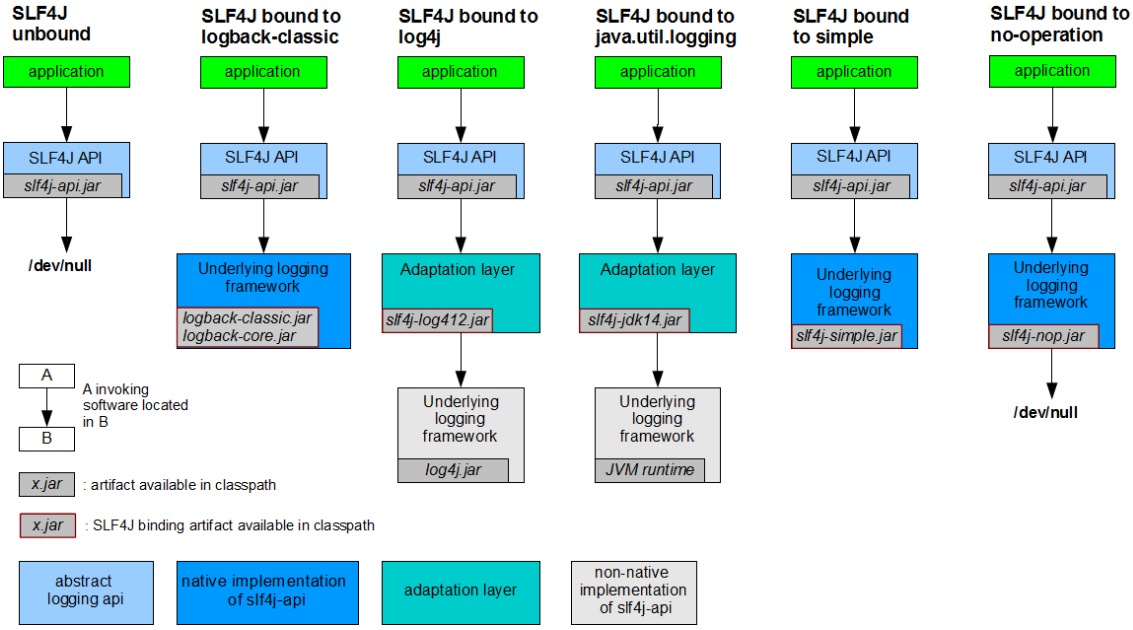
FATAL - jiné chyby, které obvykle vedou k předčasnému vypnutí aplikace. Jedná se o nejzávažnější chyby a měly by být vypsány na konzoli.

2.4 Používané frameworky

Mezi nejznámější frameworky používané pro záznam událostí v Java aplikacích se řadí Log4J, Logback vycházející z Log4J, Java Logging API z balíku java.util.logging, Apache Commons Logging and SLF4J. Mezi jednotlivými frameworky jsou v některých oblastech rozdíly. V dnešní době se však již jedná o rozdíly spíše nepatrné. Např. konfigurace Log4J a Logback jsou prakticky totožné. Velkým pomocníkem pro práci se zmíněnými frameworky je SLF4J umožňující jednoduchou náhradu jednoho API druhým bez nutnosti změny implementace.

2.5 SLF4J

Jedná se o abstrakci různých logovacích frameworků. Využívá se zde návrhový vzor Fasáda. Tento způsob návrhu dovoluje programátorovi zvolit si nástroj pro logování v kterékoliv části vývojového cyklu software a jeho jednoduchou záměnu. Jmenovitě dovoluje použití nástrojů log4j, java.util.logging, Simple logging a NOP. Projekt Logback podporuje SLF4J nativně. Výhodou použití knihovny SLF4J je to, že při potřebě logování stačí vývojáři vytvořit objekt Logger pomocí třídy LoggerFactory z balíku org.slf4j a zavoláním její statické metody getLogger. Argumentem je buď řetězec nebo instance



Obrázek 1: Použití SLF4J s logovacími frameworky (převzato z [1])

class. V obou případech slouží argument k pojmenování vytvořené instance Logger z balíku org.slf4j. Toto pojmenování se poté používá v konkrétní implementaci pro umožnění volby úrovně logování podle umístění v hierarchii. Implementace Logger z SLF4J API neumožňuje logování na úrovni Fatal, protože tato úroveň byla podle zkušeností vyhodnocena jako redundantní s úrovní Error. Developer však může pro specifické chování vytvořit vlastní úroveň logování pomocí třídy org.slf4j.Marker.

Jak již bylo zmíněno, jedinou nativní implementací SLF4J je projekt Logback. Pro zprovoznění ostatních implementací (log4j, java util logging, apache commons logging) je nutné použít adaptéry, které jsou ve formě JAR souborů dostupné na stránkách projektu SLF4J. Vazby mezi jednotlivými knihovnami jsou blíže vysvětleny na obrázku. 1

Podmínkou pro použití SLF4J je umístění příslušných souborů na classpath. Základem je soubor slf4j-api.jar, který obsahuje kompletní implementaci fasády. Další nutné soubory jsou odvozeny od konkrétního použitého frameworku. Pokud je použit Logback, stačí mít na classpath umístěny soubory logback-classic.jar a logback-core.jar. V případě jiných implementací je nutno použít jak adaptér zmíněný dříve, tak soubory potřebné k používání samotné implementace. O další operace, jako načtení konfigurace, inicializaci instancí implementující org.slf4j.Logger mají již na starost jednotlivé konkrétní knihovny. Všechny kroky běhu SLF4J v aplikaci jsou:

- zprovoznění žádané implementace logování umístěním příslušných souborů na classpath
- zhodnocení, zda navázání SLF4J s implementací bylo úspěšné, pokud ne, je použita defaultní prázdná implementace NOP

- použití implementace k logování

Součástí projektu SLF4J je i nástroj SLF4J Migrator, který slouží k jednoduchému a rychlému nahrazení statické implementace Log4j, apache commons logging nebo java util logging implementací používající SLF4J. I když u tohoto způsobu existují určité limitace vzhledem k nahrazovaných implementacím, stále se jedná o značné urychlení procesu. Použití nástroje Migrator je vysvětleno na stránkách SLF4J[1].

2.6 Logback framework

Pro podrobnější popis jsem vybral framework Logback. Jedním z důvodů je jeho kompatibilita s SLF4J. Hlavní třída ch.qos.logback.classic.Logger implementuje přímo org.slf4j.Logger, takze zde nedochází ke zbytečné režii ať již se jedná o paměť či procesor. Implementace Logback se skládá ze tří hlavních komponent jimiž jsou logback-core, logback-classic a logback-access. Jak již bylo zmíněno dříve, v samotném programu nejsou nutné žádné reference na Logback. SLF4J ověří, zda se na classpath nacházejí alespoň logback-classic a logback-core a použije je. Komponenta logback-classic rozšiřuje logback-core a implementuje SLF4J. Hlavními třídami Logbacku jsou Logger, Appender a Layout.

Logger je součástí komponenty logback-classic. Vytvoření instance Logger je nastíněno v ukázce 1

```
LoggerFactory.getLogger(getClass());
```

Výpis 1: Vytvoření instance Logger

Pro každou instanci Logger je možné nastavit, která úroveň bude logována. Pojmenování Loggeru umožňuje použití hierarchie tak, že pokud není úroveň logování pro některý Logger nastavena, Logback se pokusí použít nastavení předka. Hierarchičnost se aplikuje způsobem shodným s pojmenováním balíčků a tříd Javě. Například logger pojmenovaný 'cz.martinbayer.logger' je předkem 'cz.martinbayer.logger.Class'. Bude mít tedy logger pojmenovaný 'cz.martinbayer.logger' nastavenu prioritu ERROR a zároveň 'cz.martinbayer.logger.Class' nebude mít nastavenu žádnou úroveň, bude použití nastavení předka, tedy ERROR. Mezi jednotlivými úrovněmi logování taktéž platí určité závislosti zaručující, že pokud je povoleno logování s nějakou 'prioritou', je zároveň automaticky umožněno logovat události s vyšší prioritou. Pokud je kupříkladu pro logger povoleno logování úrovně WARN, potom je automaticky dovoleno logovat události na úrovni ERROR. Použití TRACE, INFO a DEBUG bude zakázáno a neprojeví se na výstupu[3].

Poznámka 2.1 Priorita jednotlivých úrovní: TRACE <DEBUG <INFO <WARN <ERROR

Appender je součástí komponenty logback-core a jeho hlavním úkolem je zajištění přístupu frameworku do zvolené cílové destinace. Aktuálně existují Appendery pro velké množství výstupů. Jedná se o výstupy na konzoli nebo do souboru, také je možné provádět logování na vzdálený server přes technologií soketů. Významným pro logování také bývá použití databázových lokací (MySQL, PostgreSQL, Oracle atd.). Ke každému

loggeru může být připojeno více appenderů. Často se jedná o výstup na konzoli a do souboru zároveň. Vztahy appenderů a loggerů jsou odvozeny z hierarchie loggerů. Některé appendery zajišťují i složitější logiku nad vytvářením a spravováním logovaných záznamů. Jeden z nich se jmenuje `RollingFileAppender`, který podle nastavení v konfiguraci umožňuje cyklické zapisování souborů. Pokud dosáhne hlavní logovaný soubor určité velikosti, jeho obsah je zkopírován do souboru se stejným názvem a číslem na konci. Doba, po jakou se mají záznamy archivovat je součástí konfigurace. Používá se přístup, kdy je kontrolováno stáří záznamů a po určité době jsou jisté záznamy smazány, nebo se kontroluje pouze počet vytvořených log souborů a pokud se má vytvořit další, nejstarší z nich je smazán a u všech je inkrementována číselná část názvu výstupního souboru[4].

Implementace rozhraní Layout se starají o transformaci příchozí události na řetězec. Události v komponentě logback-classic jsou pouze typu `ch.qos.logback.classic.api.ILoggingEvent`. Každý vývojář je tedy schopen vytvořit si svůj vlastní layout. Nejznámějším a nejpoužívanějším layoutem je PatternLayout, který je založen na konceptu příkazu `printf()` jazyka C. Schéma layoutu je složeno ze specifikátorů, které obsahují literály a výrazy určujících formátování. Každý specifikátor začíná znakem

V prvních dvou krocích framework rozhoduje, zda bude událost zapsána do logu nebo jestli bude ignorována. První krok obsahuje vyfiltrování nežádoucích událostí. Logback má již implementovány některé filtry. Patří mezi ně například GEventEvaluator zajišťující filtrování na základě podmínky napsané v jazyce Groovy, filtr TresholdFilter pouze zakazuje logování pro události na nižší úrovni než je definována argumentem `<level></level>` nebo LevelFilter umožňuje specifikaci chování přímo pro určitou úroveň logovaného záznamu. Vývojáři mohou implementovat vlastní filtry. Stačí vytvořit třídu, která rozšiřuje abstraktní třídu Filter a implementovat její metodu `decide()`. Pokud je použit filtr, je při rozhodování zavolána metoda `decide`. Ta vrací jednu z hodnot enumerátoru FilterReply. Zde patří hodnoty `ACCEPT`, `DENY` a `NEUTRAL`. Konfigurace může obsahovat soubor filtrů, které se řetězcově řadí a jsou postupně vyhodnocovány. Pokud je filtrem vrácena hodnota DENY, logování pro aktuální událost je zablokováno. Hodnota NEUTRAL způsobí, že je zpracovávání aktuálním filtrem přerušeno a vyhodnocení je předáno dalšímu filtru v pořadí. Hodnota ACCEPT způsobí, že je filtrování skončeno a neprovádí se ani druhý krok první části procesu. Poslední možná hodnota, DENY, přeruší celý proces logování. Druhou podmínkou pro zalogování eventy je logování na stejné nebo větší úrovni pro aktuální pojmenovaný logger. Pokud podmínka nevyhovuje, je logování přerušeno.

Pokud má být událost zalogována, je vytvořen objekt `LoggingEvent` obsahující všechna data pro vytvoření záznamu. Instance `LoggingEvent` obsahuje následující data:

- jméno loggeru, na kterém byla událost vyvolána
- úroveň, pro kterou byla událost vytvořena
- samotná zpráva předána jako argument při vyvolání události
- text případné chyby předané při konstrukci
- čas, kdy byl požadavek na zapsání události vytvořen

- název vlákna, ve kterém byla zpráva zaznamenána
- MDC(Mapped Diagnostic Context) - může obsahovat libovolná data. Je obsažena v balíku `org.slf4j`. Příklad použití je uveden ve výpisu 2. Pokud je pro aktuální appender nastaven vzor `<Pattern>%X{first} %X{last} - %m%n<Pattern>`, je výsledek následující:

```
Prvni Druhy debug1
Prvni Druhy debug2
```

```
MDC.put("first","Prvni"); /* do kontextu je nastavena hodnota "Prvni" pod klicem " first " */
MDC.put("second","Druhy"); /* to stejne pro "second" */
logger.debug("debug1"); /* je zalogovana zprava "debug1" na urovni DEBUG */
logger.debug("debug2"); /* je zalogovana zprava "debug2" na urovni DEBUG */
MDC.remove("first"); /* smazani kontextu */
MDC.remove("second");
```

Výpis 2: Ukázka použití MDC

Po vytvoření `LoggingEvent` objektu je vyvolána metoda `doAppend` na všech vhodných Appenderech. Ta se postará o formátování podle definované konfigurace náležející aktuálnímu appenderu a následné umístění, vytištění nebo zaslání zformátované Eventy na příslušnou lokaci.

2.7 Výkon

Výkon frameworku zajišťující logování je velmi podstatný. Pokud je logování vypnuto (level loggeru je nastaven na OFF), výkonostní náročnost je minimální. V tomto případě se jedná pouze o zavolání metody a porovnání hodnoty typu `Integer`. V ostatních případech obsahuje každý logger informaci o úrovni logování, kterou má použít i v případě, že ji nemá přímo konfigurovánu. Tato informace je inicializována již při spuštění, takže pozdější rozhodování je velmi rychlé. Udávaná hodnota pro formátování a zapsání události do lokálního souboru je pro procesor Pentium 3.6GHz kolem 10 mikrosekund[3].

3 Použité technologie

Implementace zadání diplomové práce je provedena nad platformou Eclipse, konkrétně Eclipse 4. Principy této technologie jsou postaveny na specifikaci OSGI. Při implementaci je obvykle použita také technologie Equinox, která je implementací OSGI.

3.1 OSGI a Equinox

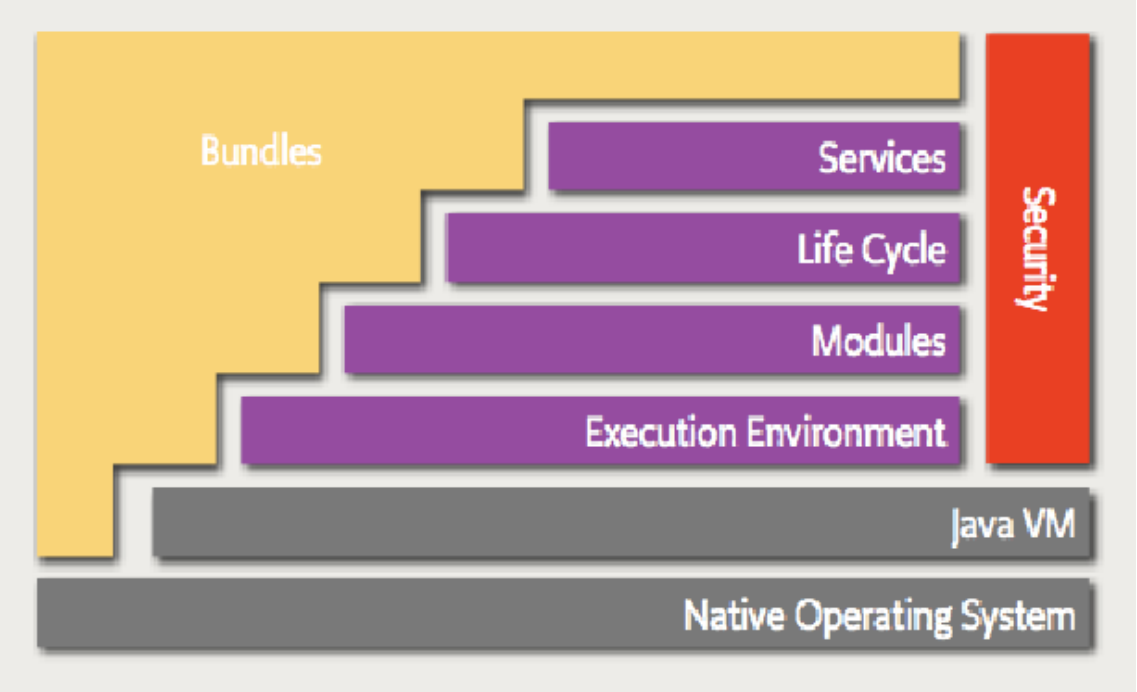
Jedním z hlavních parametrů technologie Eclipse je použití spolupracujících komponent, z kterých jsou výsledné aplikace tvořeny. K popisu chování komponent je v Eclipse použita specifikace OSGI.

OSGI je soubor specifikací, které definují dynamický systém komponent pro programovací jazyk Java. Koncept OSGI dovoluje skládání aplikací z více znovupoužitelných komponent. Pro komunikaci mezi komponentami se používají tzv. `services`. Díky této vlastnosti není nutné, aby spolupracující komponenty věděly cokoliv o použitých implementacích. Specifikace OSGI tedy dovoluje snadné rozdělení systému do logických celků a tento typ architektury dovoluje budovat rozsáhlejší systémy při zachování minimální komplexnosti (pokud je specifikace vhodně použita). Aby mohly komponenty spolupracovat, stačí, když pro ostatní komponenty vystaví služby, které poskytují a které potřebují ke svému běhu.

Jednotlivé komponenty vytvořené vývojářem se nazývají `bundles`. Může v nich být definováno kompletní chování, logika a případně i uživatelské rozhraní části systému. Tato komponenta je tvořena jednoduchým JAR souborem. Rozdíl oproti klasickému JAR souboru je právě ve viditelnosti jednotlivých funkcionalit pro ostatní komponenty systému. Zatímco při použití JAR souboru mohou ostatní komponenty použít jakékoliv veřejné nebo chráněné třídy a metody, OSGI definuje pro bundle vrstvu omezující viditelnost komponenty. Vrstva se nazývá `modul`. Z toho tedy vyplývá, že pokud má některá jiná komponenta používat daný bundle, musí jeho modul explicitně dovolit(exportovat) funkcionalitu, která má být viditelná. Konkrétně se dají exportovat balíky. Na druhou stranu, pokud daná komponenta potřebuje využívat služby jiné komponenty, musí specifikovat, co přesně chce použít. Její modul tedy importuje potřebné balíky. Návrh architektury je zřejmý z obrázku2

Použití služeb(Services) umožňuje využití sofistikovanějšího přístupu pří získávání instancí bez aplikace postupu často implementovaného v Javě, kterým jsou Factories. Řešením tohoto problému je použití tzv. `OSGI service registry`. Pokud nějaký bundle vytvoří objekt, může ho do tohoto registru služeb vložit pod klíčem reprezentovaný rozhraním (např. IMyService), které daný objekt implementuje. Pod stejným rozhraním mohou být registrovány různé implementace více komponent. Pokud jiná komponenta potřebuje získat některou z těchto instancí, může ji použít pomocí rozhraní použitého jako klíče. Při registraci více implementací daného rozhraní je možno použít `properties` k vyhledání požadované konkrétní implementace. V kostce je chování následujíc[6]:

- jedna komponenta zaregistruje své služby pomocí daného rozhraní (např. IMyService)



Obrázek 2: Architektura OSGI (převzato z [6])

- pokud druhá komponenta potřebuje některou z instancí IMyService, může ji získat z OSGI service registry
- druhá komponenta může dále naslouchat událostem vyvolaným první komponentou

Díky tomuto přístupu je možno jednotlivé komponenty instalovat, spouštět i odinstalovávat za běhu aplikace. Odinstalování je samozřejmě možné pouze tehdy, pokud není daný modul právě používán. Vzhledem k tomu, že OSGI je pouze předpis a specifikace modulárního chování Java aplikace, existuje několik jejich implementací. Z opensource jsou to například Knoplerfish OSGI, Apache Felix nebo Equinox.

Eclipse Equinox je implementace OSGI tvořící základ Eclipse aplikací. Equinox specifikace definuje OSGI bundles jako plug-iny. Rozšiřuje architekturu OSGI o tzv. extension points. Při implementaci v Eclipse je tak možno použít jak koncept služeb, tak i koncept extension points[7].

3.2 Eclipse

Eclipse je platforma dovolující implementaci modulárních Java aplikací s využitím specifikace OSGI, její implementace Equinox a definicí použití uživatelského rozhraní. Eclipse je open source projekt, jehož počátky sahají do roku 2001. Komunita projektu zaštiťuje více než dvěstě dalších projektů poskytujících produkty pro různé fáze softwarového

vývoje. Nejznámějším z nich je Eclipse IDE, které je nejrozšířenějším vývojovým prostředím Java vývojářů. Vývoj Eclipse je řízen neziskovou organizací *Eclipse Foundation*. Ta se stará především o podporu open source komunity a zajištění standardů u projektů vyvíjených touto komunitou. Všechny projekty vyvíjené pod dohledem Eclipse Foundation jsou vydávány pod licencí *EPS-Eclipse Public License*.

V roce 2004 byla vydána verze Eclipse 3.0. Ta poprvé podporovala použití Eclipse platformy k vývoji samostatných aplikací. Tyto aplikace jsou pojmenovány jako Eclipse RCP[rich client platform] aplikace. Eclipse platforma je základem pro aplikace vyvíjené společnostmi jako IBM nebo Google, což dokazuje flexibilnost i pokračující rozvoj tohoto frameworku. Jeho modulárnost umožňuje budování systémů založených na znovupoužitelných komponentách. V pozadí celého projektu stojí velká komunita vývojářů i uživatelů poskytujících podporu při problémech s vývojem aplikací pomocí Eclipse frameworku.

Základem eclipse je *Eclipse platform projekt* poskytující jádro frameworku a služby, pomocí kterých jsou Eclipse aplikace implementovány. Zároveň poskytuje prostředí, v kterém jsou aplikace spouštěny a spravovány. Základní myšlenkou je zpřístupnění způsobu, kterým je možno jednoduše a rychle vytvářet nové nástroje a aplikace.

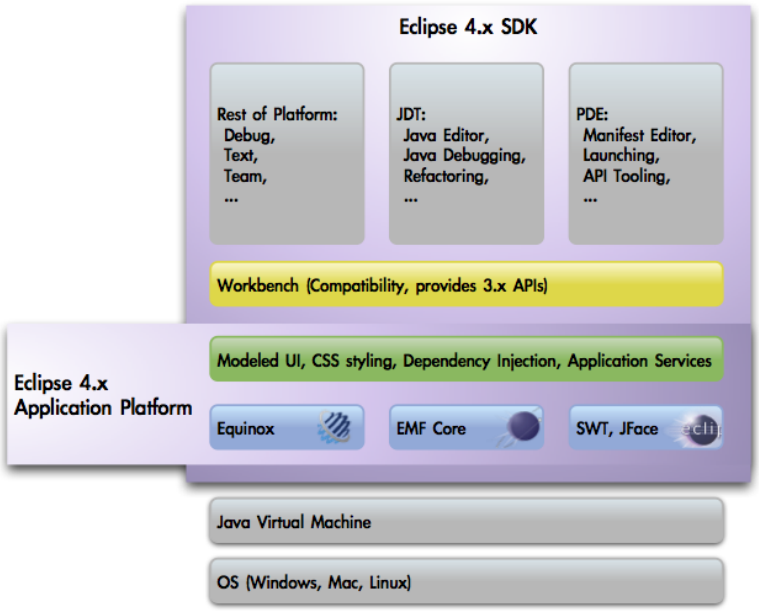
3.3 E4

Eclipse 4 je nová verze eclipse platformy používající nové sadu nových technologií poskytujících ještě flexibilnější vývoj eclipse plug-inů a aplikací. Při vývoji frameworku ve verzi 4 bylo využito možnosti zdokonalit stávající části a přepracovat části způsobující problémy. Oproti Eclipse 3.x jsou zavedeny některé nové koncepty:

- Pro popis Eclipse aplikace je nyní použita struktura nazývající se Aplikační model (Application model)
- Tento model může být modifikován jak při vývoji tak i za běhu aplikace
- Model je možno také rozšiřovat
- Je zavedena podpora Dependency Injection
- Vzhled grafických prvků Eclipse aplikace může být přizpůsoben použitím CSS
- Aplikační model je oddělen od samotného frameworku použitého při implementaci uživatelského rozhraní

Eclipse 4 obsahuje vrstvu, která dovoluje spuštění plug-inů napsaných pod verzí Eclipse 3.x beze změn. Kompletní architektura SDK pro Eclipse 4 je patrna na obrazku3.

Eclipse 4 byl vyvinut v rámci projektu *e4*. Jedná se o inkubátor zajišťující projekty vedoucí k vývoji eclipse platformy. Projekt zavedl do Eclipse platformy některé nové technologie, které byly přeneseny zpětně i do jádra celého frameworku. Pro vývoj aplikací nad platformou Eclipse 4 jsou používány nástroje vyvinuty v rámci projektu *Eclipse e4 tooling*, které nejsou součástí Eclipse 4 platformy. Většina základních konstrukcí nabízených technologií Eclipse 4 byly použity i při implementaci diplomové práce.



Obrázek 3: SDK Eclipse 4.x (převzato z [**?**])

3.3.1 Dependency Injection

Dependency Injection(DI) v Eclipse 4 zjednodušuje přístup ke globálním singleton proměnným, ke kterým se v Eclipse 3 přistupovalo pomocí statických metod. Anotace "@Inject"označuje v Eclipse 4 konstruktor, metodu nebo proměnnou dostupnou pro DependencyInjection. Obecně zle DependencyInjection použít pro všechny komponenty obsažené v aplikačním modelu. Pomocí metod třídy `ContextInjectionFactory` lze vložit DI kontext i do ostatních objektů. Příklad použití DI je předveden v ukázce 3.

```
/* metoda create je implementovana v tride, ktera je soucasti modelu */
@Inject
public void init (MApplication app){
    /* inicializace promenne message pojmenovane "messageToBeInjected" a jeji vlozeni do
        kontextu */
    String message = "Say hello to Eclipse 4";
    IEclipseContext ctx = app.getContext(); /* ziskani DI kontextu z instance MApplication, ktera
        je automaticky injectovana z nadrazeneho kontextu*/
    ctx.set("messageToBeInjected", message);
}

@PostContextCreate /* metoda oznacena touto anotaci je volana po vytvoreni DI kontextu pro
    aktualni tridu a jako jeji parametry jsou automaticky pouzity promenne obsazene v DI
    kontextu */
public void create(Composite parent, @Optional@Named(value="messageToBeInjected")String
    message){
    /* parent je rodicovska komponenta nutna pro vytvoreni SWT komponent - v DI kontextu je
        dostupna automaticky */
```

```
    /* message je promenna vlozena do kontextu pred zavolanim metody create. napr. pomoci
       predchozi metody init() */
    /* anotace @Optional zajistuje, ze pokud nebyla hodnota pojmenovane promenne "
       messageToBeInjected" jeste vlozena do kontextu, bude jeji hodnota NULL, bez Optional
       anotace by byla frameworkem zobrazena vyjimka */
}
```

Výpis 3: Ukázka Dependency Injection v E4

3.3.2 Logování

Možnost logování je ve třídách aplikačního modelu dovolena pomocí instance třídy Logger z pluginu `org.eclipse.e4.core.services`. Je možné ji využít s použitím dependency injection 4.

```
@Inject Logger logger;
```

Výpis 4: Vložení Logger pomocí DI

3.3.3 Aplikační model

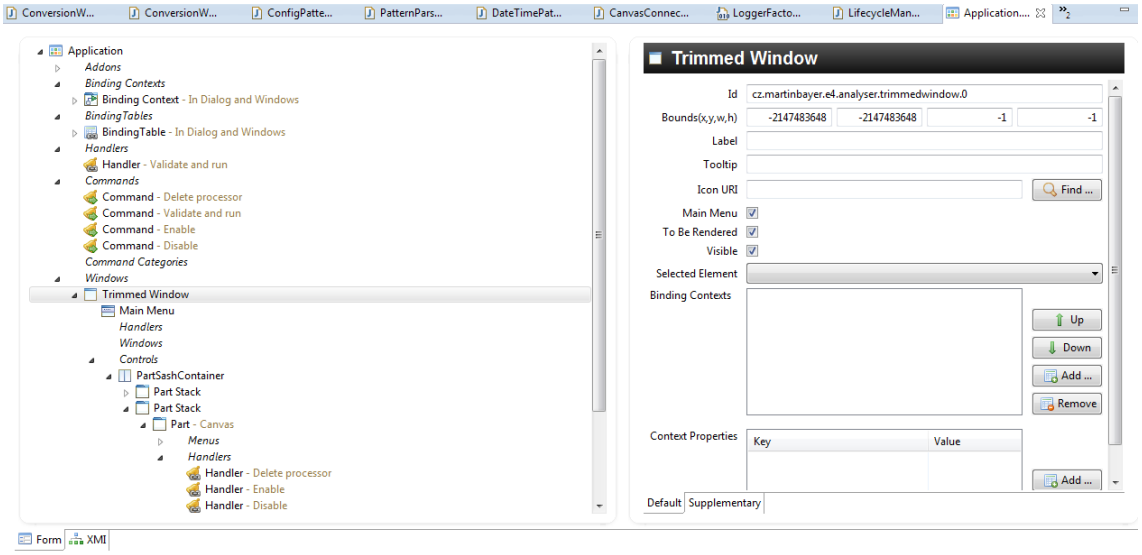
Aplikační model v Eclipse 4 popisuje strukturu aplikace. Jsou v něm obsaženy vizuální komponenty i některé funkční prvky (handlers, commands). Většina objektů modelu je hierarchicky seřazena. Obsah jednotlivých komponent není definován v aplikačním modelu, ale v naprogramovaných třídách. V modelu je možné definovat vlastnosti objektů v něm obsažených. Jedná se především o identifikátory, popisky, velikosti. Model je specifikován v souboru `Application.e4xmi`. Ten je nutný pro spuštění RCP aplikace na platformě Eclipse 4. Při úpravě a vývoji aplikačního modelu jsou užitečné E4 tools, protože nabízí nástroje k ulehčení práce s aplikačním modelem, který je ve skutečnosti XMI soubor. Úprava souboru aplikačního modelu tedy vypadá jako na obrázku 4.

3.3.4 Handlery

Handlerem se v Eclipse 4 nazývá třída registrovaná v aplikačním modelu. Handler je spouštěn při uživatelských akcích spouštěných stisknutím tlačítek, zvolením položek menu apod. Handler běžně implementuje dvě metody označené anotacemi `@Execute` a `@CanExecute`. První z nich specifikuje chování prováděné po spuštění handleru. Druhá na základě podmínek určuje, zda je možno handler spustit či nikoliv. Protože je handler součástí aplikačního modelu, jeho parametry může být jakákoliv proměnná definovaná v aplikačním kontextu. Je zde opět použita Dependency Injection[8].

3.4 SWT a JFace

SWT (Standard Widget Toolkit) a JFace jsou knihovny pro tvorbu uživatelského rozhraní v Eclipse. SWT definuje grafické prvky zvané *widgets*, dále pak umožňují jejich rozmístění pomocí *layout managerů*. Implementace SWT podporuje vykreslování svých komponent



Obrázek 4: Úprava aplikačního modelu

na platformách Windows, Linux, Mac OS a dalších. Pokud je to možné, využívá SWT nativní položky (widgety) daného operačního systému pomocí *Java Native Interface-JNI*. JNI je framework, který dovoluje Java aplikaci běžící v JVM volat a používat nativní aplikace a knihovny vytvořené v jiném programovacím jazyce jako C++ nebo Assembler.

Je tedy jasné, že aplikace vytvořená pomocí SWT bude velice podobná ostatním aplikacím běžícím na určité platformě. V tomto ohledu je srovnatelné s AWT. SWT má však implementováno více komponent (např. tabulky). Pokud potřebná grafická komponenta není na dané platformě dostupná, SWT ji emuluje[9].

JFace je nástroj poskytující pomocné nástroje a třídy k podpoře tvorby komponent grafického rozhraní, které by bylo jinak zdlouhavé. JFace umožňuje programátorovi soustředit se na vývoj specifické funkcionality místo řešení běžných problémů grafických prvků. Vyžívá, ale nepřepisuje komponenty SWT, pouze zajišťuje jejich efektivnější využití[10]. JFace implementuje tzv. *DataBinding*, což je systém automatické validace a synchronizace hodnot mezi objekty. Nejčastěji se používá pro provázání hodnot zobrazeným na grafickém rozhraní a hodnotami v modelu aplikace. JFace obsahuje implementaci databinding funkcionality pro komponenty SWT, JFace a JavaBeans[11].

4 E4Logsis

Aplikace implementovaná jako diplomová práce se nazývá E4Logsis. Jedná se totiž o nástroj určený k analýze logů postavený na platformě Eclipse 4. Eclipse framework byl zvolen kvůli potřebě modulárního přístupu, který je u Eclipse 4 aplikací splněn použitím OSGI specifikace.

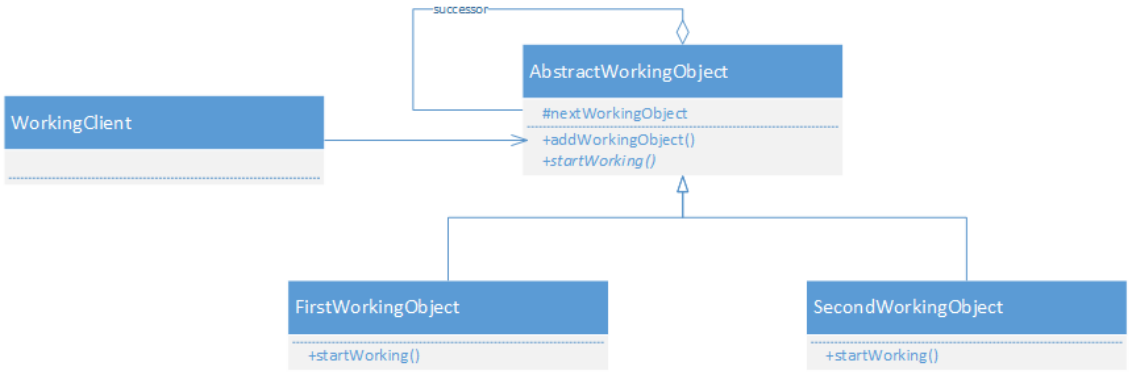
4.1 Hlavní principy a myšlenky

Analýza chování aplikací nasazených u zákazníka je zásadním problémem zasahující všechny vývojáře jak komerčních, tak i open-source aplikací a projektů. Správným přístupem k této problematice není implementace jednorázového řešení pro danou aplikaci, neboť většinou neumožňuje snadnou konfiguraci a přizpůsobení pro použití s jinými aplikacemi. Základní myšlenkou při tvorbě aplikace E4Logsis je vytvoření frameworku umožňujícího tvorbu nástrojů na analýzu logů aplikací. Nejedná se tedy o hotový software, který by byl okamžitě použitelný. Jedná se o rámec, na jehož základě je možné provádět různé činnosti nad záznamy chodu aplikací i jinými soubory. V aplikaci jsou definovány pouze nejzákladnější části, které mohou být při analýze logů potřebné. Ostatní komponenty si doprogramuje vývojář sám přesně podle svých potřeb. Pokud je tedy nástroj použit například v rámci vývojového nebo analytického týmu, s postupem času se možná variabilita aplikace stane tak velkou, že již bude možno vypracovávat scénáře analýzy pouze s již vytvořenými komponentami. Návrh a implementace jednotlivých komponent musí být jednoduchá, aby ji zvládl jakýkoliv Java programátor, který nutně nemusí znát použití Eclipse technologie. Návrh implementace frameworku počítá s využitím návrhového vzoru *Chain of responsibility*.

Aplikace E4Logsis implementuje dvě hlavní součásti. Jsou jimi paleta komponent a plátno. Při realizaci scénáře pro analýzu vybere vývojář, které komponenty chce použít a seřadí je do řetězce na plátně. Jednotlivé části později podle potřeby propojí. Pokud žádná z komponent nevyhovuje danému účelu, může ji programátor snadno implementovat, sdílet s ostatními a instalovat. Pokud již daná komponenta existuje a je vytvořena její nová verze, nainstalovaná komponenta je aktualizována. Pokud může každý vývojář v týmu pracovat na komponentách aplikace E4Logsis, je vhodné mít pro jejich zdrojové kódy zařízený samostatný repositář právě z důvodu lepší a jednodušší správy vytvořených komponent.

4.2 Návrhový vzor Chain of responsibility a jeho použití

Jak již bylo zmíněno, modulární implementace systému jednotlivých komponent využívá myšlenku návrhového vzoru Chain of responsibility. Ten, jak již název napovídá, řadí objekty do řetězce. Objekty si později při spuštění předávají odpovědnost za provedenou akci. V základu zná každý objekt svého následníka, na kterém zavolá příslušnou metodu. Všechny třídy v řetězci musí být potomkem stejného předka, kterým je obvykle abstraktní třída nebo rozhraní definující abstraktní veřejnou metodu. Všechny prvky v řetězci je možno velice snadno vložit pouze pomocí prvního objektu. Například pomocí



Obrázek 5: Návrhový vzor Chain of responsibility

metody `add(ChainItem item)`. Pokud má první prvek definován svého následníka, zavolá na něm opět metodu `add`. Tato operace probíhá až do doby, než je nalezena komponenta bez definovaného následníka. Spuštění procesu je zajištěno jen pro první objekt v řetězci, zbývající jsou vždy volány předcházejícím objektem. Často bývá využíváno podmínek, kdy některé ze zřetězených komponent jen spustí vykonávání operace na dalším objektu, neboť vstupní parametry neodpovídaly spuštění aktuální funkcionality[13]. Třídní diagram návrhového vzoru je vyobrazen na obrázku5.

4.3 Řídící komponenta

Druhou podstatou frameworku E4Logsis je jeho řídící komponenta(Canvas Objects Manager). Ta má na starost korektní přidávání komponent na plochu a zajišťuje jejich korektní spojení pomocí konektorů. Zároveň je schopna vrátit z Eclipse contextu všechny dostupné instance procesorů, které prvními v řetězci. Klasický Chain of responsibility návrhový vzor je tedy rozšířen o možnost, mít na jedné úrovni více komponent.

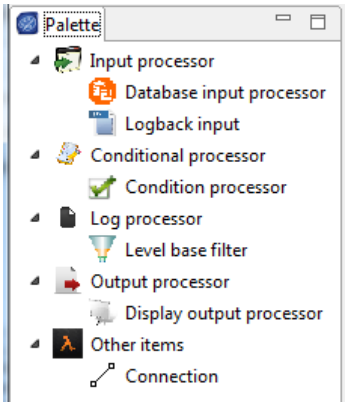
4.4 Architektura

4.5 Grafické rozhraní

Všechny komponenty jsou vytvořeny pomocí SWT nebo JFace. Jak bylo zmíněno dříve, okno aplikace je rozděleno do dvou hlavních částí. První z nich je paleta obsahující komponenty, neboli business procesory vykonávající určitý druh činnosti. Druhou částí je plátno (canvas), na které jsou jednotlivé procesory umisťovány a spojovány do logických řetězců tak, aby mohly vykonávat potřebnou činnost.

4.5.1 Paleta s procesory

Paleta je vytvořena jako jedna část, v aplikačním modelu zvaná jako Part. Její implementace je provedena ve třídě PalettePart. Jedná se o jednoduchou třídu, která zajišťuje pouze



Obrázek 6: Paleta s procesory

dvě operace. Hlavní, zde použitou, grafickou komponentou je zde `TreeViewer` z balíku `org.eclipse.jface.viewers`. Aby bylo možné vykreslit položky jednotlivých procesorů, je nutné nastavit pro TreeViewer tzv. *content provider* a *label provider*. Content provider definuje vztahy rodič-potomek mezi jednotlivými položkami. V případě procesorů se jedná o jednoduchou vazby, kdy jsou položky rozděleny do několika skupin podle určení použití. Label provider implementuje grafické zobrazení procesoru v paletě. Vykreslena je jak ikona procesoru, jeho název. Jako data pro paletu, tedy instance jednotlivých procesorů, jsou použity plug-iny zaregistrované jako služba pod rozhraním `IProcessorItemWrapper`. Vzhled palety vidíme na obrázku6.

4.5.2 Canvas

Druhou, nejdůležitější, částí grafického rozhraní je plocha, neboli canvas(plátno). Canvas je implementován jako nekonečná plocha, existuje zde proto podpora posunu v horizontálním a vertikálním směru. Procesory vybrané v paletě je možné umístit kliknutím myši na canvas. Pomocí položky *Connection* jsou jednotlivé procesory propojeny. Platí zde určitá pravidla tak, aby uživatel nemohl volit nevalidní scénáře. Tyto podmínky závisí na typech spojených procesorů. Objekty přidávané na plátno jsou vlastní grafické komponenty vytvořené rozšířením třídy `org.eclipse.swt.widgets.Composite`. Položka představující procesor se skládá z pozadí, ikony přiřazené procesoru při jeho implementaci a textové komponenty definující jméno procesoru na ploše. Procesor podporuje *Drag'n'Drop* posouvání. Taktéž komponenta znázorňující propojení procesorů je rozšířením třídy Composite. Jsou zde implementovány posuny koncových bodů přímky podle posunu procesorů spojených touto komponentou.

Na procesor i konektor je navázáno kontextové menu, které je zobrazeno pravým kliknutím myši. Nabízí se zde možnosti smazání komponenty, její zakázání nebo povolení. V případě smazání procesoru jsou smazány všechno konektory, které s ním mají společný bod. Konektor je mazán samostatně. Do všech zmíněných operací se zapojuje *Canvas*

Object Manager starající se o korektní správu logiky procesorů i konektorů, jejich přidávání i odstraňování. V manageru jsou generována výchozí unikátní jména procesorů.

5 Reference

[1] SLF4J user manual. QUALITY OPEN SOFTWARE. *Simple Logging Facade for Java (SLF4J)* [online]. 2005, 2014-03-31 [cit. 2014-04-07]. Dostupné z: http://www.slf4j.org/manual.html

[2] Chapter 2: Architecture. QUALITY OPEN SOFTWARE. *Logback Project* [online]. 2008 [cit. 2014-04-07]. Dostupné z: http://logback.qos.ch/manual/architecture.html

[3] Chapter 2: Architecture. QUALITY OPEN SOFTWARE. *Logback Project* [online]. 2008 [cit. 2014-04-07]. Dostupné z: http://logback.qos.ch/manual/architecture.html

[4] Chapter 4: Appenders. QUALITY OPEN SOFTWARE. *Logback Project* [online]. 2008 [cit. 2014-04-07]. Dostupné z: http://logback.qos.ch/manual/architecture.html

[5] Chapter 6: Layouts. QUALITY OPEN SOFTWARE. *Logback Project* [online]. 2008 [cit. 2014-04-07]. Dostupné z: http://logback.qos.ch/manual/architecture.html

[6] The OSGi Architecture. OSGI ALLIANCE. *OSGi Alliance* [online]. 1999, 2013-12-16 [cit. 2014-04-07]. Dostupné z: http://www.osgi.org/Technology/WhatIsOSGi

[7] OSGi Modularity - Tutorial. VOGEL, Lars. VOGELLA. *Vogella* [online]. 2007, 2013-06-10 [cit. 2014-04-07]. Dostupné z: http://www.vogella.com/tutorials/OSGi/article.html

[8] VOGEL, Lars. *Eclipse 4 application development: Eclipse RCP based on Eclipse 4.2 and e4* [online]. Leipzig: [Vogel/a], c2012, xx, 408 s. [cit. 2014-04-07]. Wizard wand series. ISBN 978-394-3747-034. Dostupné z: http://www.vogella.com/books/eclipsercp.html

[9] SWT - Tutorial. VOGEL, Lars. VOGELLA. *Http://www.vogella.com/* [online]. 2010, 2013-10-15 [cit. 2014-04-07]. Dostupné z: http://www.vogella.com/tutorials/SWT/article.html

[10] The JFace UI framework. *Eclipse documentation* [online]. 2007 [cit. 2014-04-07]. Dostupné z: http://help.eclipse.org/helios/index.jsp?topic=

[11] JFace Data Binding. THE ECLIPSE FOUNDATION. *Wiki.eclipse.org* [online]. 2005, 2013-08-14 [cit. 2014-04-07]. Dostupné z: https://wiki.eclipse.org/JFace_Data_Binding

[12] Eclipse4. THE ECLIPSE FOUNDATION. *Wiki.eclipse.org* [online]. 2010, 2014-03-07 [cit. 2014-04-07]. Dostupné z: https://wiki.eclipse.org/Eclipse4

[13] Chain of Responsibility. *Design Patterns* [online]. 2001 [cit. 2014-04-07]. Dostupné z: http://www.oodesign.com/chain-of-responsibility-pattern.html