VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Framework pro analýzu logů Java aplikací

Framework for Java Log Analysis

2014 Martin Bayer

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Bayer**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Framework pro analýzu logů Java aplikací
Framework for Java Log Analysis

Zásady pro vypracování:

Cílem diplomové práce je vytvoření frameworku nad platformou Eclipse. Framework bude zajišťovat základní operace pro analýzu logů s možností dodatečných rozšíření tak, aby byla výsledná aplikace uzpůsobená pro použití s konkrétním softwarem vyvíjeným v Javě.
Framework bude zajišťovat obecné řešení pro přístup ke zdrojům logů (textové soubory, databáze, XML soubory). Implicitně bude schopen pracovat se základními frameworky, které se používají pro logování Java aplikací (Log4j, Logback, Java Logging API) a jejich konfiguracemi.

1. Analyzujte a popište používané techniky pro vytváření logů v dnešních aplikacích.
2. Navrhněte možná řešení obecné analýzy logů a zvolte nejvhodnější z nich.
3. Navrhněte architekturu frameworku tak, aby byly pokryty základní operace nad logovanými záznamy včetně správy zdrojů při analýze záznamů větších velikostí a optimalizace rychlosti zpracování.
4. Implementujte řešení pro použití s konkrétním systémem včetně vhodného uživatelského rozhraní.
5. Proveďte testy a zhodnoťte přínos implementované aplikace.
6. Zvažte náročnost změny architektury tak, aby byl výsledný produkt spustitelný jako tenký klient ve webovém prohlížeči, případně na mobilním zařízení.

Seznam doporučené odborné literatury:

[1] Vogel, L., Eclipse 4 Application Development: The complete guide to Eclipse 4 RCP development, 2012, ISBN 978-3943747034
[2] Schildt, H., Java, A Beginner's Guide, 5th Edition, McGraw-Hill Osborne Media; 5 edition, 2011, ISBN 0071606327
[3] Schildt, H., Java The Complete Reference, 8th Edition, McGraw-Hill Osborne Media; 8 edition, 2011, ISBN 0070435928
[4] Horstmann, C. S, Core Java Volume I--Fundamentals (9th Edition), Prentice Hall; 9 edition, 2012, ISBN 0137081898

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Ing. Michal Krumnikl**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2014 .....

Rád bych na tomto místě poděkoval všem, kteří mi byli v mé práci nápomocní. Především vedoucímu mé diplomové práce Mgr. Ing. Michalu Krumniklovi za trpělivost a ochotu.

Abstrakt

Diplomová práce se věnuje tvorbě aplikace určené k analýze log záznamů. Je vypracována v programovacím jazyce Java na platformě Eclipse 4. Hlavní myšlenkou je její rozšiřitelnost pomocí pluginů. První část se věnuje popisu možných způsobů logování a nástrojů používaných při vývoji Java aplikací. Ve druhé části jsou popsány specifika platformy Eclipse 4 a dalších použitých technologií. Poté následuje vysvětlení hlavních principů implementované aplikace. Poslední část je zaměřena na testování při použití v reálných situacích.

Klíčová slova: analýza log souborů, Eclipse 4 RCP, Java, plug-in, SLF4J, Logback, Log4J

Abstract

The diploma thesis deals with the implementation of the application for the log file analysis. It is created using the Java programming language and Eclipse 4 platform. The main idea is to allow the application to be extended via plug-ins. The first part focuses on the description of possible ways of logging and tools used during the application development. In the second part, the specifications of Eclipse 4 platform and other used technologies are described. This is followed by the main principles used for the log analysis application development and theirs explanation. The last part deals with the tests of the application in the real situations.

Keywords: log files analysis, Eclipse 4 RCP, Java, plug-in, SLF4J, Logback, Log4J

Seznam použitých zkratek a symbolů

RCP	–	Rich client platform
SLF4J	–	Simple Logging Facade for Java
API	–	Application programming interface
NOP	–	no operation logger
JAR	–	Java Archive
OSGI	–	Open Service Gateway initiative
SWT	–	Standard Widget Toolkit
JNI	–	Java Native Interface
AWT	–	Abstract Window Toolkit
IDE	–	Integrated development environment
DI	–	Dependency injection
SDK	–	Software development kit
CSS	–	Cascading Style Sheets
RAP	–	Remote application platform
GEF	–	Graphical Editing Framework
XML	–	Extensible Markup Language
XSLT	–	Extensible Stylesheet Language Transformations
JAXB	–	Java Architecture for XML Binding
HTML	–	HyperText Markup Language
XMI	–	XML Metadata Interchange
JVM	–	Java virtual machine
FOP	–	Formatting Objects Processor

Obsah

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, analytik 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 není zanedbatelný. 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 dosaženo, 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 E4Logsis, 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 pluginu a ten nainstalovat do aplikace. Pluginy budou moci být koncentrovány 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 plugin 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 frameworků používaných k zajištění logování Java aplikací 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 pluginů 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 pluginu 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.

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 zapisovány 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.

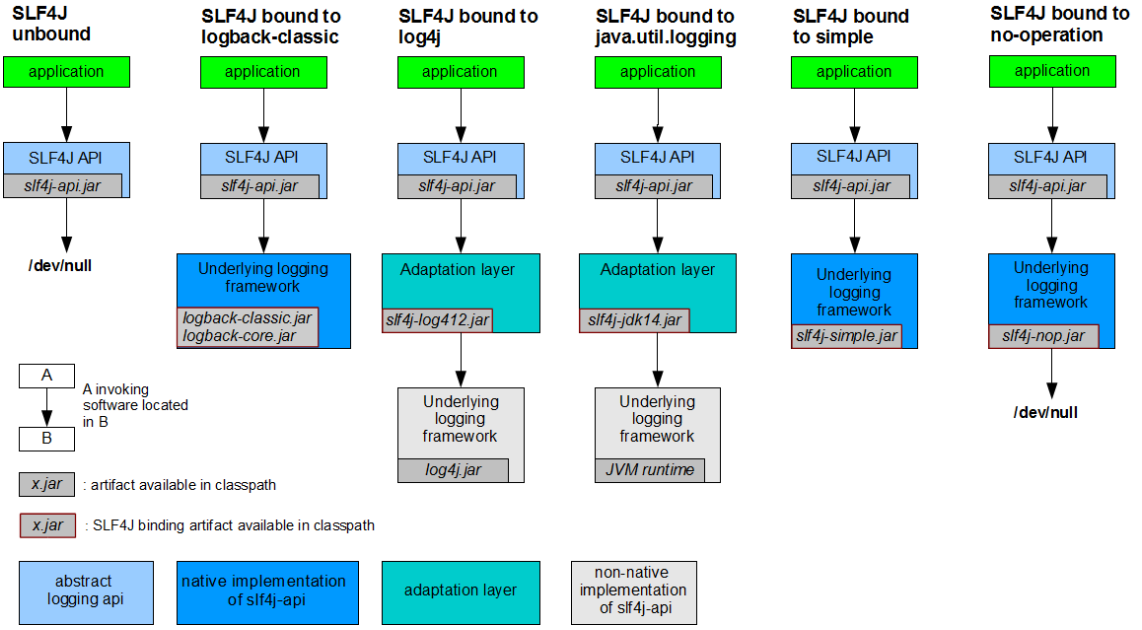
FATAL - jiné chyby, které obvykle vedou k předčasnému vypnutí aplikace. Jedná se o nejzávažnější chyby a měly by být vypsány na konzoli.

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 vytvářené aplikace.

2.5 SLF4J

Jedná se o abstrakci různých logovacích frameworků. Využívá se zde návrhový vzor Fasáda (Facade). 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. 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 výchozí 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 nahrazovaným 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`, takže 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 v programovacím jazyce Java[1]. Například logger pojmenovaný `cz.martinbayer.logger` je předkem `cz.martinbayer.logger.Class`. Pokud bude mít tedy logger pojmenovaný `cz.martinbayer.logger` nastavenu prioritu *ERROR* a zároveň `cz.martinbayer.logger.Class` nebude mít nastavenu žádnou úroveň, bude použito 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].

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 použitím technologie soketů. Významným pro logování bývá také použití databázových lokací (MySQL, PostgreSQL, Oracle atd.). Ke každému loggeru

[1]Priorita jednotlivých úrovní: TRACE <DEBUG <INFO <WARN <ERROR

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 % a je následován volitelnými formátovacími parametry, konverzním slovem a volitelnými parametry ve složených závorkách. Podle konverzního slova rozhoduje PatternLayout, jaká informace obsažená v události bude na té dané pozici zapsána. Kompletní workflow tvorby záznamu v log souboru pomocí Logbacku je následující [5].

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 `\textlessPattern\textgreater\%X\{first\}\%X\{last\}-\%m\%n\textlessPattern\textgreater`, 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 Nástroje použité při implementaci

Při implementaci aplikace jsem použil několik nástrojů zjednodušujících samotný vývoj, správu změn nebo tvorbu poznámek. U nejpoužívanějších z nich nyní krátce zmíním jejich účel a přínos:

- Java 7[2] - aplikace je spouštěna v rámci frameworku e4 na platformě Java 7. Právě z důvodu jednoduché implementace lambda funkcí bude ihned po vydání Eclipse IDE 4.4. změněno použití Javy 7 na Javu 8.
- Eclipse for RCP and RAP developers[3] - je verzí Eclipse IDE určenou pro vývojáře Eclipse pluginů a aplikací. Ve výchozím stavu obsahuje všechny potřebné součásti pro vývoj RCP aplikací a pluginů.
- e4tools[4] - rozšíření Eclipse IDE o rozsáhlejší podporu vývoje Eclipse 4 RCP aplikací
- Hosted redmine[5] - jedná se o volně dostupnou službu poskytovanou firmou BitBot[6]. Jedná se o systém vytváření a spravování problémů a nových funkcionalit. Hlavním přínosem je přehlednost a možnost přímého propojení s SVN serverem
- GitHub[7] - pro účely správy verzí jsem při vývoji aplikace použil verzovací systém Git. Ten umožňuje založení lokálního repositáře a následnou synchronizaci změn na server, kterým je právě github.
- MindMup[8] - je aplikace spustitelná z webového prohlížeče Chrome. Její funkcionalita umožňuje tvorbu jednoduchých myšlenkových map a jejich ukládání v rámci Google dokumentů. Je to jeden z nejjednodušších způsobů, jak tyto mapy synchronizovat mezi více počítači a archivovat je.

[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[3] https://www.eclipse.org/downloads/packages/eclipse-rcp-and-rap-developers/keplersr2
[4] http://download.eclipse.org/e4/downloads/
[5] http://hostedredmine.com/
[6] http://www.bitbot.com.au/
[7] https://github.com/
[8] http://www.mindmup.com/

4 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.

4.1 OSGI a Equinox

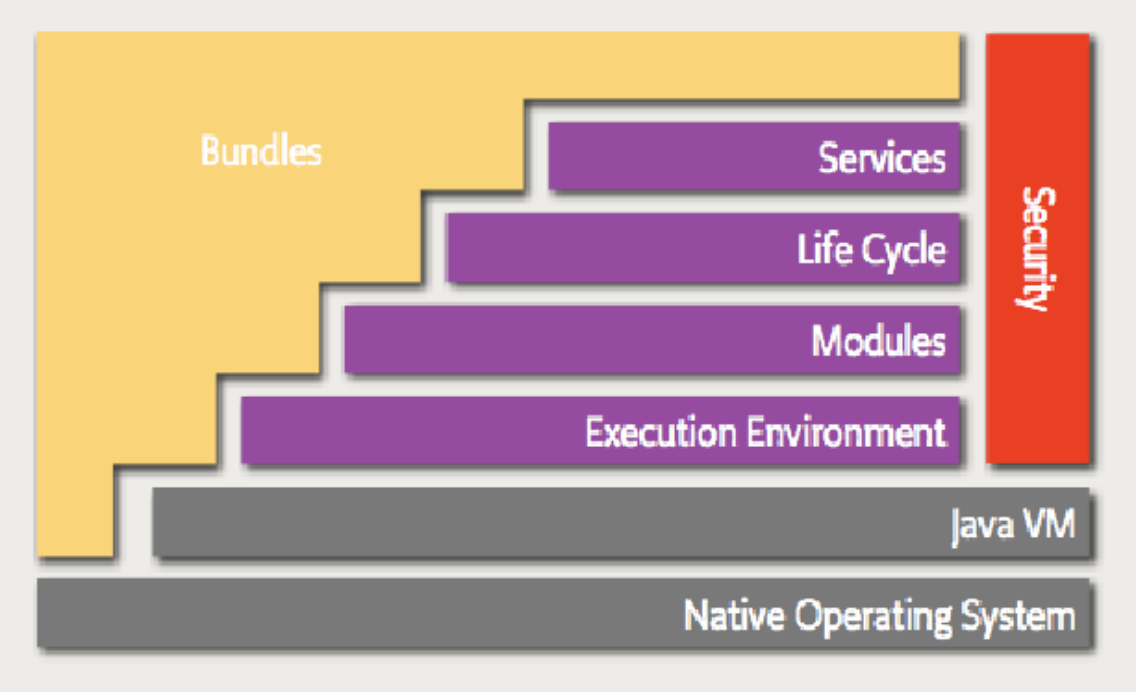
Jedním z hlavních parametrů technologie Eclipse je použití spolupracujících komponent, z kterých jsou výsledné aplikace tvořeny. K popisu chování komponent je v Eclipse použita specifikace OSGI.

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 obrázku 2.

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 [6]. V kostce je chování následující:

- 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 jeho 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 pluginy. Rozšiřuje architekturu OSGI o tzv. extension points. Při implementaci v Eclipse je tak možno použít jak koncept služeb, tak i koncept extension points [7].

4.2 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 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, ve kterém jsou aplikace spouštěny a spravovány. Základní myšlenkou je zpřístupnění způsobu, kterým je možno jednoduše a rychle vytvářet nové nástroje a aplikace.

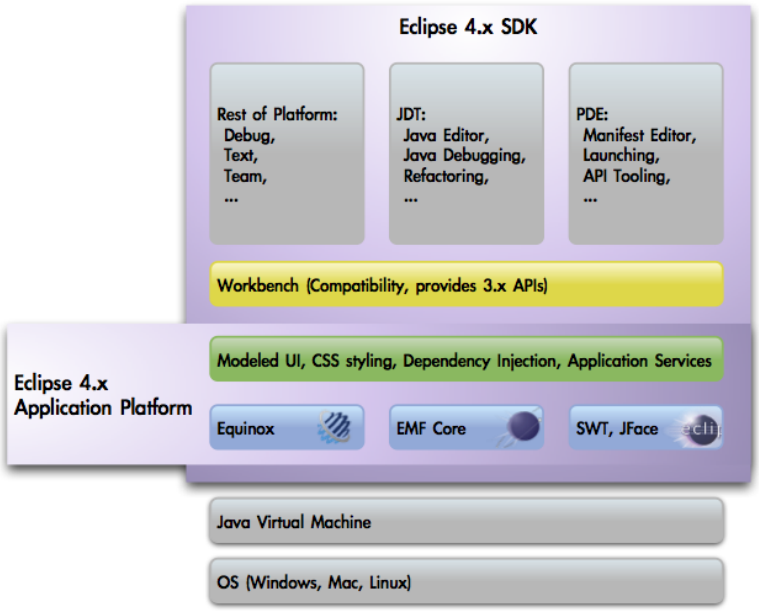
4.3 E4

Eclipse 4 je nová verze eclipse platformy používající sadu nových technologií poskytujících ještě flexibilnější vývoj eclipse pluginů 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 pluginů 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 byla použita i při implementaci diplomové práce.



Obrázek 3: SDK Eclipse 4.x (převzato z [12])

4.3.1 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 lze 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

4.3.2 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

4.3.3 Aplikační model

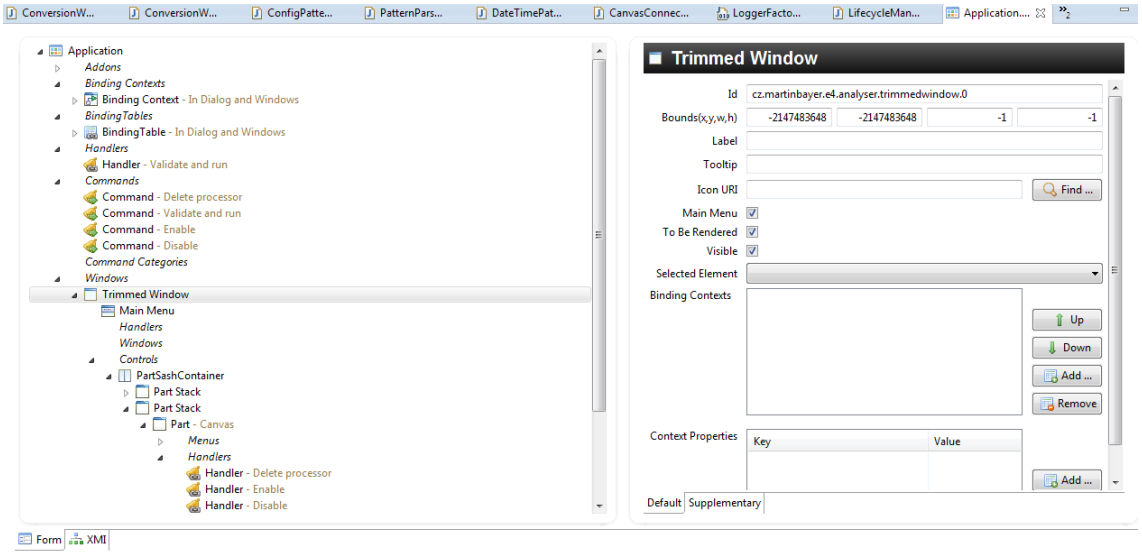
Aplikační model v Eclipse 4 popisuje strukturu aplikace. Jsou v něm obsaženy vizuální komponenty i některé funkční prvky (handlers, commands). Většina objektů modelu je hierarchicky seřazena. Obsah jednotlivých komponent není definován v aplikačním modelu, ale v naprogramovaných třídách. V modelu je 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.

4.3.4 Handlery

Handlerem se v Eclipse 4 nazývá třída registrovaná v aplikačním modelu. Handler je spouštěn při uživatelských akcích po stisknutí tlačítka, 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].

4.4 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. Využívá, 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 zobrazených na grafickém rozhraní s hodnotami v modelu aplikace. JFace obsahuje implementaci databinding funkcionality pro komponenty SWT, JFace a JavaBeans [11].

5 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.

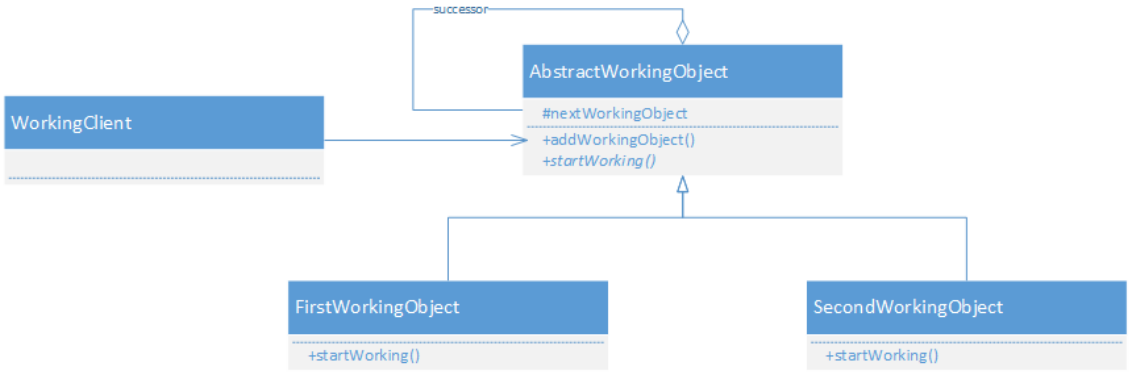
5.1 Hlavní principy a myšlenky

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.

5.2 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 obrázku 5.

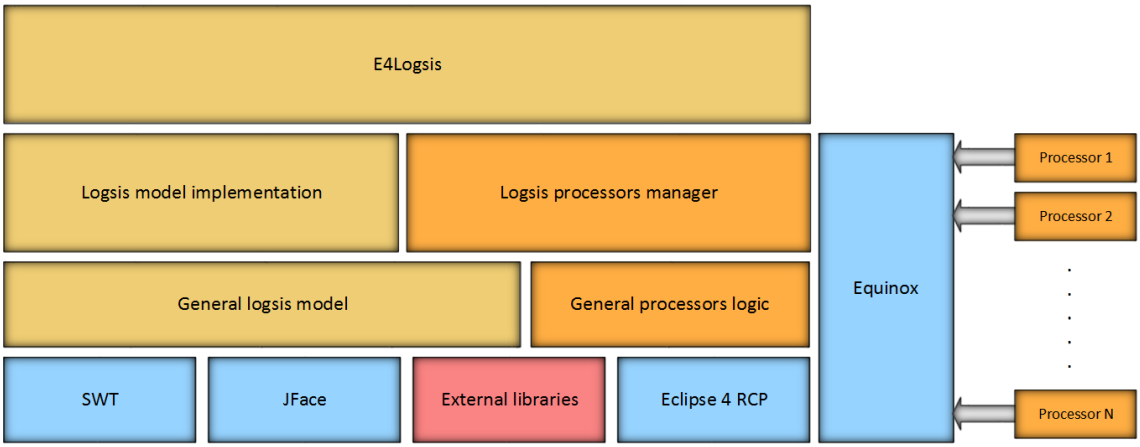
5.3 Řídící komponenta

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 jsou prvními v řetězci. Klasický Chain of responsibility návrhový vzor je tedy rozšířen o možnost mít na jedné úrovni více komponent.

5.4 Architektura

Jak již bylo popsáno dříve, implementace je postavena na technologiích JFace, SWT a hlavní komponentou je Eclipse 4 RCP. Další komponenty jsou zobrazeny na obrázku 6. Jednotlivé části architektury jsou implementovány jako Eclipse pluginy, avšak na všech, vyjímaje procesory, je přímo závislá funkcionalita E4Logsis aplikace. Funkce a funkcionalita použitých bloků je následující:

- External libraries - obsahuje knihovny třetích stran. Mezi ně patří například Apache knihovny FOP, common-beanutils, common-io, commons-validator a další. Použity jsou především nadřazenými vrstvami ke zjednodušení běžných úkonů jakými je klonování objektů, exportování výsledků pomocí XSL transformace apod.
- General logsis model a General processors logic - jsou zde definovány jednotlivé typy procesorů jako abstraktní třídy a jejich základní implementace. Dále je zde abstraktní



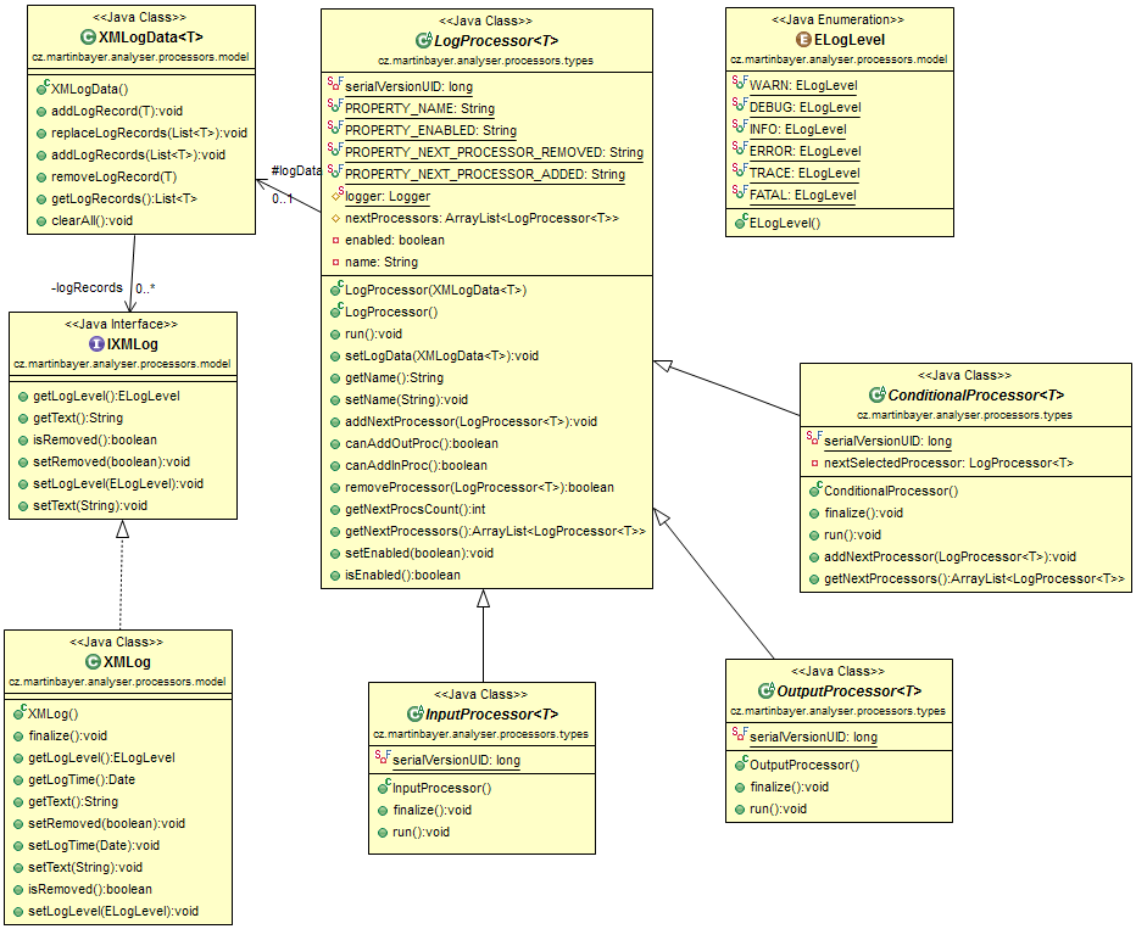
Obrázek 6: Architektura aplikace E4Logsis

popis datové struktury kolekce dat vzniklých z reálných záznamů. Vztahy mezi jednotlivými implementacemi procesorů a daty jsou jasné z třídního diagramu 7. Struktura a rozdíl ve funkčnosti jednotlivých typů procesorů bude popsána později.

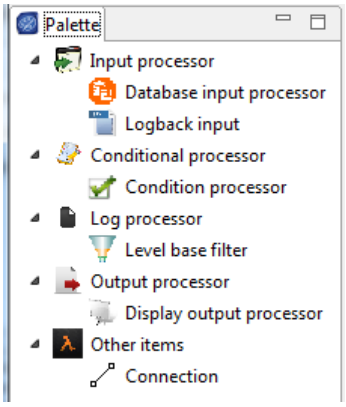
- Logsis model implementation - jedná se o konkrétní implementaci objektu představující jednotku dat ekvivalentní k jednomu záznamu v reálných log záznamech. Tento objekt je poté jednotlivými procesory používán při zpracování.
- Logsis processors manager - při spouštění aplikace inicializuje procesory z nainstalovaných pluginů. Jejich instance poté poskytuje nadřazené vrstvě. Při inicializaci kolekce procesorů zároveň uchovává reference na nainstalované pluginy z důvodu pozdější deserializace objektů.
- Equinox - tato implementace OSGI zajišťuje instalaci a aktualizaci pluginů (procesorů).
- E4Logsis - využívá ostatní komponenty k vytváření scénářů tvořených pro účely analýzy log záznamů.

5.5 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.



Obrázek 7: Architektura aplikace E4Logsis



Obrázek 8: Paleta s procesory

5.5.1 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 dvě operace. Hlavní, zde použitou, grafickou komponentou je `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 jednoduché vazby, kdy jsou položky rozděleny do několika skupin podle určení použití. Label provider implementuje grafické zobrazení procesoru v paletě. Vykreslena je jak ikona procesoru, tak jeho název. Jako data pro paletu, tedy instance jednotlivých procesorů, jsou použity pluginy zaregistrované jako služba pod rozhraním `IProcessorItemWrapper`. Vzhled palety vidíme na obrázku 8.

5.5.2 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 všechny konektory, které s ním mají společný bod. Konektor je mazán samostatně. Povolení a zakázání zvoleného procesoru je podstatné pro celou logiku použití programu. Pokud uživatel potřebuje dočasně použít na místě existujícího procesoru jiný, může dosavadní procesor zakázat, vytvořit objekt nového a propojit se sousedními procesory. Do všech zmíněných operací se zapojuje `CanvasObjectManager` starající se o korektní správu logiky procesorů i konektorů, jejich přidávání i odstraňování. V manageru jsou generována výchozí unikátní jména procesorů.

5.6 Business procesory

Procesory vykonávající činnost nad zpracovávanými záznamy jsou základem celé aplikace. V následující části budou popsány jejich typy a předpokládaná funkcionalita, vztahy mezi procesory a další operace mající vztah k procesorům a prováděné během používání.

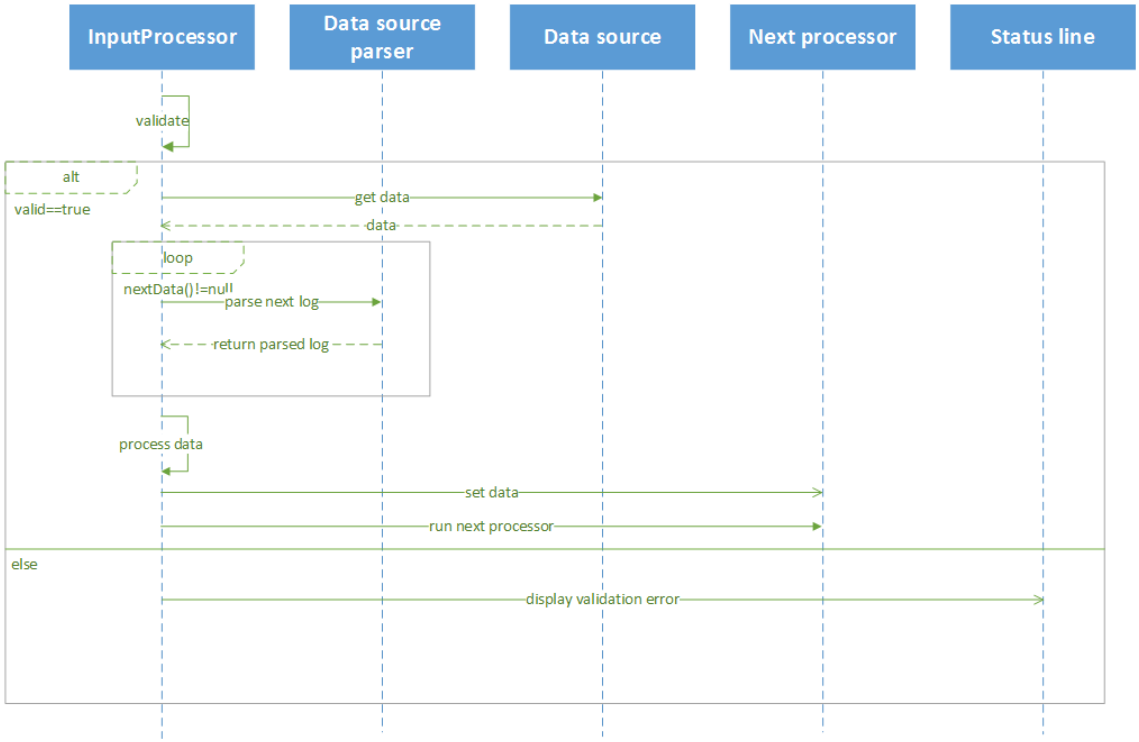
5.6.1 Typy procesorů

Jednotlivé business procesory jsou rozděleny do čtyř skupin podle svého určení. V abstraktních třídách je implementována příslušná funkcionalita tak, aby byla splněna posloupnost požadovaných operací. Zároveň umožňují kontrolu počtu příslušných povolených procesorů jak vstupních, tak výstupních.

5.6.1.1 Vstupní procesory

Vstupní procesor se stará o čtení, získávání dat pro analýzu. Jeho součástí jsou tři základní metody. První z nich se obsluhuje čtení dat z daného zdroje a jejich konverzi do kolekce objektů typu definovaného v komponentě *Logsis model implementation*. Další metoda umožňuje implementaci prvotních operací nad získanými daty. Vstupní komponenty jako jediné inicializují kolekci dat, které jsou pak dále předávány referencí, aby nebyla zvyšována hardwarová náročnost operací. Použití vstupního procesoru je omezeno tím, že není dovoleno, aby do něj vstupovala jakákoliv propojení. Sekvence kroků provedených při běhu vstupního procesoru je zobrazena na diagramu 9.

Pokud se jedná o vstupní procesor získávající data ze souborů vytvořených pomocí frameworků pro logování, je třeba k jejich načtení použít regulárních výrazů. Z testování vyplynulo, že pro velké textové soubory není vhodné použít jen regulární výrazy na vyhledání a extrahování kompletních záznamů. Pokud je záznam obsažen, je zpracování velmi rychlé, avšak pokud aktuální kus řetězce nevyhovuje danému regulárnímu výrazu, je zpracování pro větší kusy textu velmi pomalé. Regulární výraz totiž nemůže být pomocí konfigurace Logback ani Log4J nijak délkově omezen, prohledává proto celý soubor. Z tohoto důvodu je výhodnější použít řešení šité na míru každému jednotlivému způsobu logování a příslušnému typu log souboru. Zpravidla je pro jednu aplikaci používáno logování stejného vzoru, proto není implementace vzhledem k přínosům nikterak zbytečná. Je například výhodnější rozdělit nejdříve vstupní soubor podle regulárního výrazu do jednotlivých záznamů a až poté z nich vyčíst potřebná data. Později bude popsán systém čtení záznamů pro konkrétní aplikace používající framework Logback a Log4j.



Obrázek 9: Vstupní procesor - sekvenční diagram

Ve výchozím stavu jsou v aplikaci E4Logsis nainstalovány dva vstupní procesory. Jeden z nich provádí rozbor aplikačních dat zaznamenaných s pomocí frameworku Logback. Druhý pak čte soubory zapsané s využitím frameworku Log4J. Z důvodu zmíněného v předchozím odstavci je však technologie, pomocí které byla data zaznamenána nepodstatná. První částí funkce procesoru je čtení zdrojových dat.

Ke čtení souborů je použita třída `FileChannel`, která používá přímý přístup k souboru. Pomocí dalších tříd, kterými jsou `ByteBuffer` a `CharBuffer` se část záznamu uloží do instance třídy `StringBuilder`. Pokud je podle definovaných pravidel nalezen kompletní log záznam, je příslušná část překopírována do druhé instance `StringBuilder` a z původní je smazána. Druhá instance je poté zpracována ve druhém vláknu, které slouží ke kompletnímu parsování na jednotlivé atributy. Ty jsou uloženy do objektu typu vytvořeného v modulu *Logsis model implementation*. Celý proces je implementován jako *semafor*[9]. Způsob čtení souboru je implementován v pluginu `cz.martinbayer.logparser`.

Protože probíhá čtení souborů oběma procesory analogicky, vyžaduje jejich spuštění i stejné vstupní parametry. Vstupní parametry se dají navolit v dialogovém okně. Soubory k analýze se určují buď výběrem složky nebo jednotlivých souborů. Při výběru složky jsou zobrazeny všechny obsažené typy souborů (podle přípony). Uživatel poté výběrem ze zobrazených přípon omezuje počet vstupních dat. Složky s log soubory obvykle obsahují staré záznamy s příponami doplněnými např. pořadím (input.log, input.log.1 apod.). Pokud je tedy potřeba analyzovat jen nejnovější soubory, stačí označit příponu `*.log`. Na dialogu jsou zobrazeny pomocné informace obsahující výčet zvolených souborů. Dále pak jejich počet a celkovou velikost. Z výkonnostních důvodů nejsou data čtena při každém spuštění procesu, ale jen při prvním. Pokud je třeba na základě činnosti procesoru smazat některé záznamy, jsou příslušná data jen označena jako smazaná tak, aby se s nimi nadále nepracovalo. Pokud je poté proces spuštěn znovu, je indikátor smazání dat změněn na výchozí hodnotu a data se opět jeví jako nově načtená. Tato data jsou platná pouze pro příslušný vstupní procesor. Pokud je při dalším spuštění použit jiný, jsou data přečtena znovu podle konfigurace příslušného procesoru. Zjednodušeně lze říci, že pokud je procesor použit několikrát za sebou, není nutné znovu číst a analyzovat data. Toto chování lze změnit pomocí zrušení možnosti *Use previous data* v Menu *Configuration*.

5.6.1.2 Výstupní procesory

Výstupní procesory definují způsob zobrazení nebo uložení výsledků analýzy. Procesor je validní v případě, že neobsahuje žádná výstupní propojení. Ve výchozím stavu jsou v aplikaci E4Logsis nainstalovány tři výstupní procesory. Jeden z nich zobrazuje výsledky na monitoru pomocí JFace komponenty `TableViewer`. Druhý exportuje data jako HTML stránku pomocí připravené XSLT šablony. A třetí ukládá výsledky do XML souboru.

Procesory ukládající výsledky do HTML, resp. XML souboru vyžadují zadání jen jednoho parametru, kterým je cesta k ukládanému souboru. Při exportu do XML je využito funkcionality implementované v balíku `javax.xml.bind` (JAXB). Nejdříve jsou

[9] jedná se o systém synchronizace vláken, kdy první vlákno provádí nějakou činnost a druhé čeká na její dokončení. Druhé vlákno však neblokuje vlákno první. To může pokračovat ve své činnosti okamžitě poté co upozorní druhé vlákno o změně stavu (např. přečtení dat)

jednotlivé záznamy obaleny instancí třídy označené anotacemi z balíku `javax.xml.bind.annotation`. To umožňuje jednoduché vypsání takto upravených dat do XML souboru pomocí třídy `Marshaller` z balíku `javax.xml.bind`. Ve výchozím stavu se názvy tagů rovnají názvům jednotlivých proměnných. Pokud je však nutné použít ve výsledném XML jiný název pro příslušný tag, dá se tato hodnota změnit pomocí anotace `@XmlElement`. Mezi další používané anotace patří `@XmlRootElement` - označuje proměnnou, pro niž je vytvořen kořenový uzel a `@XmlType` - používá se při definici jména jedné komponenty představující objekt, pořadí argumentů atd.

U druhého způsobu umožňujícího export a zobrazení dat ve formě HTML stránky je použito XML transformace. Konkrétně se jedná o transformaci pomocí XSL šablony. Ta obsahuje definici jednoduché HTML stránky společně s tagy knihovny XSLT. Ty definují, kde má být hodnota tagu z XML souboru umístěna. Šablona je součástí procesoru. XML schéma nutné pro vytvoření výsledného souboru je uloženo pouze v paměti. Transformace XSL šablony a XML schématu do HTML souboru je provedeno pomocí třídy `Transformer` z balíku `javax.xml.transform`.

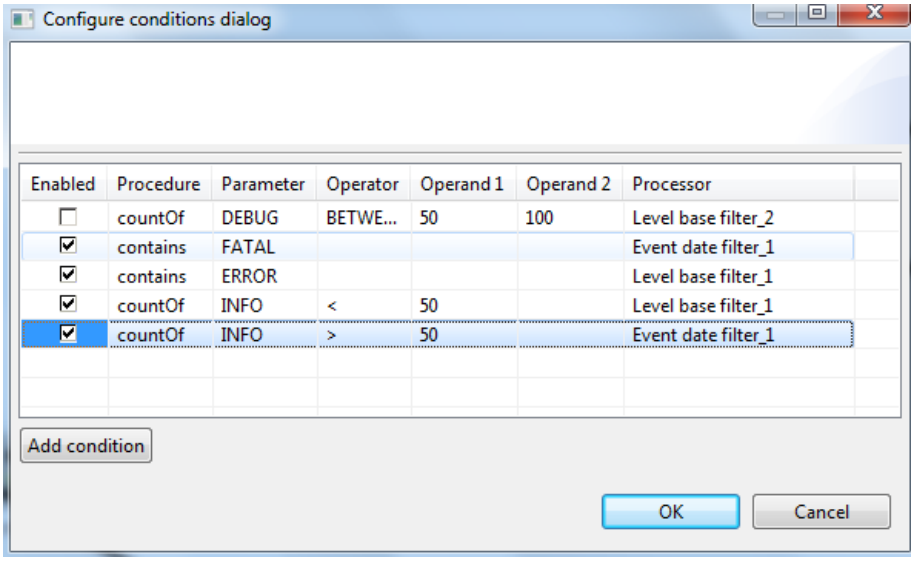
Poslední výstupní procesor zobrazuje data na monitoru. Jedná se o dialogové okno, jejímž základem je JFace komponenta `TableViewer`. V každém řádku tabulky jsou zobrazena všechna pole datového typu znázorňujícího analyzovaný záznam. Řazení může být prováděno nad sloupci s hodnotami časových údajů a úrovněmi logování. Pole obsahující delší texty jsou z důvodu přehlednosti implementovány tak, aby jejich hodnotu bylo možné zobrazit ve zvláštním dialogovém okně. Zobrazení dat na monitoru je vhodné zejména pro data menšího rozsahu. Je tedy používáno při vyhledávání určitých dat v obsahu zpráv.

Činnost výstupních procesorů probíhá ve dvou fázích. V první z nich je většinou implementován dodatečný proces nad výslednými daty. Druhá se stará o vytvoření výstupu na monitor nebo do souboru podle použitého procesoru.

5.6.1.3 Podmínkové procesory

Podmínkový procesor je jediný, který je validní, pokud z něho vychází i více spojení do následujících procesorů. Na základě splnitelnosti definovaných podmínek však musí být na konci procesu vybrán právě jeden následující procesor, který bude spuštěn. Pokud nevyhovuje ani jedna podmínka, je zpracovávání přerušeno a je zobrazena chybová hláška.

Ve výchozím stavu je v E4Logsis aplikaci nainstalován jeden podmínkový procesor. Ten má závislost na modul definující funkce. Každá funkce představuje podmínku, jejíž splnění rozhoduje o dalším průběhu zpracování dat. V rámci konfigurace procesoru je možné zvolit si aktuálně z funkcí *contains* a *countOf*. Obě tyto funkce pracují s argumenty typu úrovně logování. Jednotlivé podmínky lze v dialogovém okně deaktivovat. Taková podmínka je poté při rozhodování přeskočena a pokračuje se zpracováním následující. Řádek s podmínkou je při vyhodnocování přeskočen i v případě, že pro něho není definován cílový procesor. Ukázka konfigurace jednotlivých podmínek je předvedena na obrázku 10.



Obrázek 10: Vstupní procesor - sekvenční diagram

5.6.1.4 Filtry

Poslední skupinou procesorů jsou filtry. Ty na základě konfigurace označují příslušná data jako smazaná. Tím je omezen počet datových záznamů a jsou zobrazeny jen ty potřebné. V základní konfiguraci jsou v aplikaci E4Logsis tři základní filtry.

Prvním filtrem jsou data mazána na základě jejich úrovní logování. Na dialogu nastavení je možno vybrat úrovně, které budou při běhu aplikace smazány. Druhou možností je filtrování podle data a času zaznamenání události. Vstupními parametry jsou počáteční a koncové datum a čas. Tyto proměnné je možné konfigurovat na dialogu nastavení. Při běhu programu jsou smazány všechny záznamy, jejichž čas vytvoření nespadá do definovaného rozmezí.

Třetí filtr kontroluje obsah proměnných *message*[10] a *error*[11]. Po dvojitém kliknutí na procesor je zobrazeno dialogové okno pro specifikaci nastavení chování procesoru. Nejdříve je nutné definovat prohledávanou proměnnou. K tomu slouží dvě tlačítka označená *Error* a *Message*. Pokud není vybráno ani jedno z nich, nejsou data filtrována. V poli pro zadání vyhledávaného řetězce je možno použít znaménko '+' k oddělení jednotlivých výrazů. Poslední konfigurační komponentou je tlačítko 'All words' určující, zda musí být v daném poli obsažena všechna slova v daném výrazu.

5.6.2 Vztahy mezi procesory

Jak již bylo zmíněno, mezi jednotlivými procesory (resp. druhy procesorů) a jejich vztahy existují některá pravidla, jejichž nedodržení nedovolí úspěšnou validaci a spuštění definovaného scénáře. Aby tato pravidla neomezovala uživatele v práci, má na sobě každý

[10] obsahuje kompletní obsah úspěšně zpracovaných zpráv
[11] obsahuje zaznamenané informace o chybě

procesor definovánu operaci jeho aktivace a deaktivace. Možnosti mohou být zobrazeny použitím pravého tlačítka myši. Pokud je procesor deaktivován, jeví se všem ostatním komponentám jako procesor smazaný, ani validace tyto procesory nezohledňuje. Základními pravidly pro vztahy mezi procesory před spuštěním procesu jsou:

1. V okamžiku spuštění musí být aktivní pouze a právě jeden vstupní procesor
2. Z každého procesoru, kromě podmínkového, může vycházet právě jedno spojení
3. Do vstupního procesoru nesmí vcházet žádné spojení
4. Z výstupního procesoru nesmí vycházet žádné spojení

5.6.3 Validace scénáře

Proces validace scénáře slouží k ověření platnosti podmínek zmíněných v předchozím bodu. Nejdříve je ověřována platnost prvního pravidla. Po jejím splnění je získán odkaz na validní vstupní procesor a ten je použit jako bod, od kterého jsou validovány všechny následující procesory. Pokud je v řetězci komponent zařazena podmínka, jsou ověřovány všechny cesty z ní vedoucí. Z toho vyplývá, že se nikdy nevalidují procesory, které jsou neaktivní nebo neleží na cestě vedoucí z jediného vstupního procesoru. Pokud je při validaci nalezen problém se splnitelností některého z pravidel, je zobrazeno dialogové okno s textovým vysvětlením chyby.

5.7 Uložení projektu

Jednotlivé scénáře jsou v aplikaci nazvány jako *projekty*. K ulehčení práce uživatele dovoluje aplikace ukládání a otevírání již vytvořených projektů. Projekty jsou ukládány jako soubory s příponou `*.e4ls`. Základem procesu vytvoření obrazu aktuálního scénáře je serializace. Konkrétně se jedná o zapsání objektů pomocí třídy `ObjectOutputStream`. Z toho důvodu musí být všechny ukládané objekty serializovatelné. Pro uživatele to znamená, že pokud bude vytvářet nové třídy, jejichž obejkty budou součástí ukládaných procesorů, musí implementovat rozhraní `Serializable`. Jelikož aplikace přistupuje k instancím procesorů přes služby a nejsou pro ni tedy viditelné konkrétní třídy implementující veřejná rozhraní, vzniká při otevírání projektů a deserializaci problém s určením těchto, pro aplikaci skrytých, typů. Aby byla možná korektní deserializace, je nutné udržovat reference na použité pluginy s procesory. Ve vlastní implementaci metody `resolveClass` třídy rozšiřující `ObjectInputStream` je poté možné načítat potřebné třídy pro deserializovaný procesor. Ukázka této metody je ve výpisu kódu 5. Z důvodu úspory zdrojů při serializaci a následné deserializaci nejsou ukládány kompletní objekty. Sníží se tím také počet tříd, u kterých by bylo nutné implementovat rozhraní `Serializable`. Proměnné, které jsou vytvořeny při běhu samotné aplikace a není je třeba ukládat, jsou označeny klíčovým slovem *transient*. Pro větší objekty, které jsou představovány především položkami na Canvasu, jsou implementovány speciální třídy, které do své struktury uloží z příslušných objektů jen nejnutnější parametry. U procesorů

se jedná o jednoznačný identifikátor, umístění na canvasu, uživatelské nastavení včetně jména procesoru a téměř kompletní data nutná pro práci logické části procesoru. Nikdy nejsou ukládána zpracovávaná data, která by mohla působit výkonnostní potíže. Pro propojení mezi procesory je ukládán jednoznačný identifikátor, identifikátory procesorů, které spojuje a umístění počátečního a koncového bodu přímky na canvasu.

```
for (Bundle b : bundles) {
    try {
        clazz = b.loadClass(cname);
        /* exception not thrown so clazz was found */
        return clazz;
    } catch (ClassNotFoundException e) {
        /* ignore exception and check another bundle or throw the exception if there is no more
            bundles to check */
    }
}
throw new ClassNotFoundException("Unable to find a class locally nor in bundles");
```

Výpis 5: Načítání soukromých tříd jiných pluginů

5.8 Rozšiřování aplikace

Logika celé E4Logsis aplikace spočívá v instalaci přídatných vytvořených procesorů. Ty jsou vytvořeny jako pluginy, které je nutné zařadit do *feature project*[12]. Tato *feature* je poté umístěna na aktualizační stránku (*update site*), odkud je při spuštění aplikace stažena a její pluginy jsou instalovány, popř. aktualizovány (pokud již jsou v daný okamžik nainstalovány).

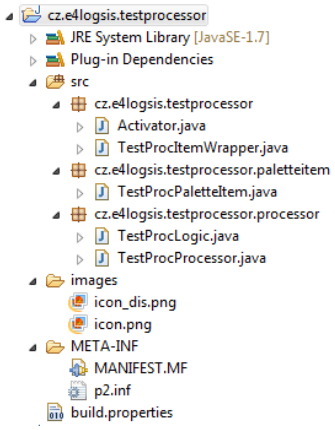
Aby bylo možné provést implementaci procesoru, je nutné splňovat několik prerekvizit. V první řadě je potřeba mít nainstalované Eclipse IDE s podporou vývoje pluginů. Základními komponentami, jejichž reference procesor obsahuje jsou pluginy `cz.martinbayer.analyser`, `cz.martinbayer.analyser.impl`, `cz.martinbayer.analyser.processor` a `cz.martinbayer.utils`. První knihovna obsahuje obecnou implementaci procesorů včetně popisu druhů procesorů a vztahů mezi nimi. V druhé se nachází pouze konkrétní implementace datového typu představující analyzovaný záznam. V `utils` knihovně se nachází obecná funkcionalita usnadňující práci s datovými typy, SWT a JFace komponentami apod. Její použití tedy není povinné. Pokud máme dostupné všechny potřebné knihovny, je postup vytvoření nového procesoru následující.

V prvním kroku vytvoříme v Eclipse IDE nový plugin projekt[13] s názvem `cz.e4logsis.testprocessor`. Pro vývoj bude použita implementace Equinox, musí být vzgenerován i aktivátor[14]. Ke správné funkcionalitě je potřebné vytvoření souborové struktury podobné té na obrázku 11. Třída `Activator` implementující rozhraní `org.osgi.framework.BundleActivator` je použita k registraci třídy `TestProcItemWrapper` pomocí rozhraní `cz.martinbayer.analyser`.

[12] jedná se o projekt, který sdružuje více pluginů tvořících určitý logický celek

[13] File->New->Other->Plug-in development->Plug-in project

[14] Třída implementující rozhraní `BundleActivator`. V *MANIFEST.MF* souboru je definována pomocí klíče `Bundle-Activator`



Obrázek 11: Souborová struktura nového procesoru

`processors.IProcessorItemWrapper`. Je nejvhodnějším místem pro implementaci jednoduchých statických metod poskytujících potřebné objekty. Většina těchto tříd v E4Logsis aplikaci bude obsahovat metodu `getEclipseContext()` s návratovou hodnotou typu `org.eclipse.e4.core.contexts.IEclipseContext`. Další vhodnou metodou je implementace poskytující objekt typu `org.eclipse.e4.core.services.log.Logger` jednoduše získanou z dostupného kontextu[15].

Třída `TestProcItemWrapper` je klíčovou součástí implementace procesoru. Jejím účelem je vytváření a vracení instancí tříd implementující rozhraní `IProcessorLogic` a `IProcessorsPaletteItem`. Dále je v `TestProcItemWrapper` třídě implementována metoda `setContext(IEclipseContextctx)`. Ta je označená anotací `@Inject` a kontext je zde vložen pomocí Dependency Injection při spouštění aplikace. Poslední metodou, obsluhující otevření dialogu nastavení (pokud je implementován), je metoda `mouseDoubleClicked(MouseEvente)`. Ta reaguje na dvojité kliknutí myši na procesor umístěný na canvasu.

Instance třídy `TestProcPaletteItem` popisuje vzhled komponenty umístěné na canvasu. Jsou zde implementovány metody pro získání ikony aktivovaného i deaktivovaného procesoru. Součástí je specifikace názvu procesoru zobrazená pro položku procesoru na paletě. Aby nebylo nutné v každé implementaci znovu vytvářet obsah metod pro získávání instancí ikon, je implementována třída `cz.martinbayer.analyser.processors.BasicProcessorPaletteItem`, která zajišťuje výchozí přístup k ikonám procesoru. Stačí tedy, aby byla tato třída rozšířena implementací metody `getLabel()` vracející popis procesoru pro paletu a aby byly definovány proměnné specifikující cesty k souborům s ikonami.

`TestProcLogic` slouží jen jako továrna vytvářející objekty typu `TestProcProcessor`. Ten již implementuje samotnou funkcionalitu prováděnou nad jednotlivými daty. Testovací procesor rozšiřuje `OutputProcessor`, proto implementuje metody `process`, `createOutput` a `showResult`. Předpokládaným výsledkem činnosti těchto metod je

[15]Instance objektu vracená metodou getEclipseContext()

konečné zpracování výsledných dat, vytvoření výstupu a zobrazení informací o ukončení procesu.

Za povšimnutí stojí soubor *p2.inf*. V něm je definována proměnná *service.name* s hodnotou `IProcessorItemWrapper`. Tato hodnota je použita při prvním použití pluginu.

5.9 Instalace procesoru

Pokud je implementace pluginu kompletní, je třeba umožnit jeho instalaci do E4Logsis aplikace. Správná funkcionalita nainstalovaných pluginů je zaručena korektní konfigurací souboru *build.properties*. Instalace pomocí komponent projektu *p2* umožňuje instalovat jen *features*. Z toho důvodu musí být nejdříve vytvořen *feature projekt*[16]. Pro ten je poté definován plugin, který se stane součástí feature. Nejjednodušším způsobem poskytnutí nové funkcionality externím aplikacím, je vytvoření aktualizační stránky[17], na kterou se dají jednotlivé *features* umístit. Schéme update site se vytváří v Eclipse IDE pomocí položky *Update Site Project* ve skupině *Plug-in Development*. Jedním ze souborů vytvořených z šablony je *site.xml*. V něm se definují komponenty, které chceme instalovat. Tlačítkem *Build* na záložce *Site Map* je vyexportována *feature* a obsažené pluginy. V sestavené stránce jsou vytvořeny složky *features* a *plugins* obsahující všechny potřebné JAR soubory představující pluginy a feature.

Instalace, popř. aktualizace pluginů zveřejněných na stránce probíhá během zapínání aplikace. K jejímu vykonání jsou použity především komponenty z balíku `org.eclipse.equinox.p2`. Při prvním spuštění aplikace je zobrazeno dialogové okno vyzývající uživatele k zadání adresy stránky s pluginy. Pokud se jedná o korektní lokaci s instalovatelnými pluginy a jsou nalezeny nové pluginy nebo již nainstalované pluginy s vyšším číslem verze, je uživatel vyzván k potvrzení instalace dostupných aktualizací. Pokud nejsou žádné nové komponenty nalezeny, není žádný dialog zobrazen a program je spuštěn. Instalace nových procesorů vyžaduje restart aplikace. O tom je uživatel informován dalším dialogovým oknem. Pokud není restartování umožněno, nové pluginy není možno používat. Jsou zobrazeny při dalším spuštění aplikace.

5.10 Aktualizace procesoru

V případě změny implementace některé části nainstalovaného procesoru může být tato komponenta aktualizována. Stačí k tomu inkrementovat číslo verze pluginu, znovu ho přidat do projektu definujícího stránku s aktualizacemi, tuto stránku znovu vyexportovat a nasadit. Aktualizace poté probíhá stejným způsobem jako instalace. P2 framework zjistí, že komponenta je již nainstalována a že je číslo verze nižší než verze publikována na *update site*. Sám tedy zvolí místo procesu instalace aktualizaci. Po aktualizaci je opět potřebný restart aplikace E4Logsis.

[16] New>Other>Plug-in development>Feature project
[17] update site

6 Testování uživatelského scénáře

V následující části bude provedeno testování na dvou scénářích z praxe[18]. Testování bude důkladně popsáno a budou zhodnoceny přínosy aplikace. Aplikací, vytvářející záznamy je komplexní systém pro správu zdrojů používaný největšími, především americkými, energetickými společnostmi. V systému figuruje celkem sedm aplikací. Analyzovaná data budou tvořena v jednom případě zprávami zaznamenanými při komunikaci mezi jednotlivými aplikacemi. V dalším případě budou analyzována data jedné z aplikací obsluhující konečnou interakci posádek pracujících v terénu se zbývajícími součástmi systému.

Celý systém je závislý na posílání zpráv mezi systémy. Jedná se o kritické místo, které v případě nesprávného chování v konečném důsledku způsobuje nedoručení dat terénním pracovníkům, ztrátu dat, konfigurací atd. Jedna z aplikací, nazvaná *Monitor*, obstarává přeposílání většiny zpráv mezi aplikacemi. Při zpracování tyto zprávy analyzuje, získává z nich data a tyto poté ukládá do databázových zdrojů příslušných aplikací. Právě při analýze zpráv dochází k častým chybám způsobeným špatným pořadím zpráv, nekompletními daty nebo kvůli chybám v jejich validaci. V případě výskytu chyby je obvykle zaznamenána výjimka a zpracování zprávy je přerušeno. V takovém případě je nutné najít konkrétní příčinu selhání, analyzovat a poté opravit, popř. zaslat analýzu týmu spravujícímu aplikaci, která chybu způsobuje. Nejdříve bude tedy provedena analýza chyby způsobující nedoručení zpráv o naplánovaném úkolu pracovní jednotce.

6.1 Scénář 1

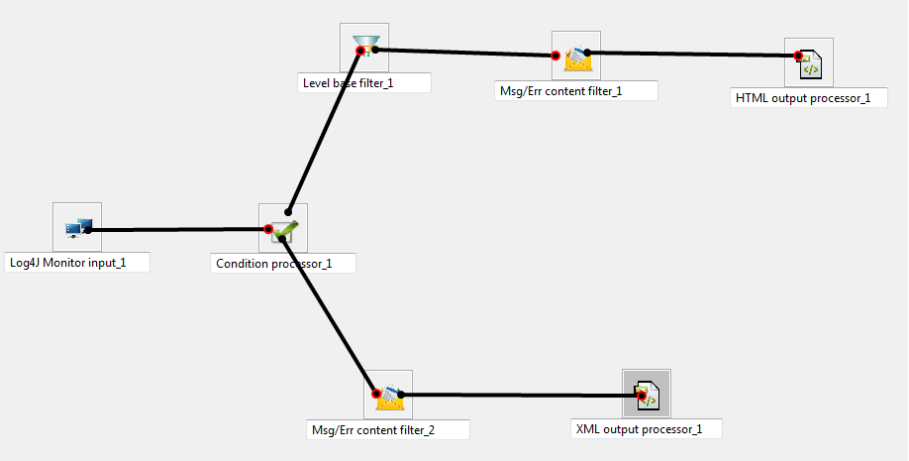
V systému pro vytváření úkolů byla vytvořena nová položka. Ta je poté odeslána do systému plánujícího práci. Odtud jsou v požadovaný okamžik odeslány zprávy specifikující úkol do mobilního systému, se kterým pracuje některá z terénních pracovních jednotek. Informace o úkolu však nejsou jednotce po odeslání dostupné. Není přesně známo, jaké typy zpráv byly použity k přenosu informací. Jedinou jistou informací je identifikátor úkolu - 1539619. Data potřebná pro analýzu chyby byla zaslána zákazníkem. Z toho důvodu není jasné, kdy přesně k chybě došlo.

6.1.1 Sestavení scénáře v E4Logsis

Vzhledem k poskytnutým informacím je zřejmé, že bude nutné zjistit, zda některé záznamy obsahují daný identifikátor. Záznamy dodané zákazníkem jsou soubory vytvořené aplikací *Monitor*. Z chybného chování se dá usuzovat, že je dostačující prohledat hodnoty proměnných představující zprávu a případnou chybu. Pokud budou obsaženy záznamy s úrovní *ERROR*, budou kvůli jasnějšímu výsledku zobrazeny jen záznamy logované s úrovní *ERROR*, *TRACE* a *DEBUG*. Pokud bude počet nalezených záznamů větší než dvacet, bude pro export použit XML soubor. V opačném případě budou vyfiltrovaná data zobrazena na HTML stránce.

Pro analýzu dodaných záznamů budou podle zadání použity následující procesory:

[18] z důvodů otestování analýzy logů vytvořených frameworky log4j a logback



Obrázek 12: Schéma umístění komponent - Scénář 1

- Log4J Monitor input - bude číst záznamy poskytnuté zákazníkem a převádět je do formy zpracovatelné ostatními procesory
- Condition processor - zjistí, zda jsou v záznamech obsaženy záznamy s úrovní *ERROR*
- Level base filter - bude použit, pokud byly v záznamech nalezeny některé vytvořené na úrovni *ERROR*. Budou vyfiltrovány jen záznamy s úrovněmi *ERROR*, *TRACE* a *DEBUG*
- Msg/Err content filter - na canvasu budou umístěny dvě instance filtru. Bude vyhledávat řetězec '1539619'. Pokud byla podmínka přítomnosti záznamů s úrovní ERROR splněna, bude prohledávat pouze vyfiltrované záznamy. V opačném případě bude prohledávat všechny záznamy
- XML output procesor - bude obsahovat všechny záznamy obsahující vyhledávaný řetězec
- HTML output processor - jeho výstupem budou všechny záznamy nalezené mezi záznamy s úrovní *ERROR*, *TRACE* nebo *DEBUG*

Po vytvoření procesorů na canvasu a jejich propojení je nutné jednotlivé procesory konfigurovat. Struktura umístění a propojení procesorů je zobrazena na obrázku 12. Pokud jsou všechny procesory korektně nastaveny, je možné spustit proces. U prvního spuštění nad danými daty se dá očekávat, že jejich analýza bude nějakou dobu trvat, neboť bude analyzováno asi 350MB dat. Při dalších spuštěních jsou již data načtena a fáze parsování není znovu vykonávána.

Po spuštění procesu pomocí tlačítka *Validate and run* se na monitoru zobrazí dialog s oznámením o probíhajícím procesu. V testovacím případě trvala analýza souborů 4m39s. Výhodou je, že pokud by bylo potřeba upravit scénář bez změny vstupního procesoru,

data se znovu nečtou. Zobrazení jiných dat v jiných formátech tak již vyžaduje jen zlomek času. Nové spuštění testovacího scénáře zabralo jen 2s času. Výsledkem je HTML stránka zobrazující pět chybových záznamů týkajících se námi hledaného úkolu. Z výsledku je patrné, že chyba nastala z důvodu duplikátu identifikátoru pracovního úkolu. V praxi to znamená, že je analýza chyby hotová. Nyní je nutné zaslat její výsledky týmu zodpovědnému za správu aplikace vytvářejících úkoly. Ideální je vytvoření XML souboru se zaznamenanými neúspěšně zpracovanými zprávami. Vzhledem k tomu, že aktuálně jsou data uložena v HTML souboru, stačí změnit cílovou destinaci propojení směřujícího do HTML procesoru tak, aby se jejím cílem stal XML procesor. Po novém spuštění procesu je vyexportován XML soubor.

Při absenci nástroje sloužícího k analýze záznamů aplikací by bylo nutné manuálně provést několik kroků. Prvním z nich je prohledání všech zaslaných souborů některým z programů přímo určených k vyhledávání nebo pomocí funkcionality obsažené v jiném softwaru. Pokud bychom řetězec '1539619' hledali v rámci aplikace *Unreal Commander*[19], trvalo by zobrazení souborů obsahujících řetězec jen několik málo vteřin. Horší by však bylo prohledávání, analýza a následné kopírování zpráv do externího zdroje kvůli další práci na problému. V tomto případě by byla časová náročnost podobná použití aplikace E4Logsis. Rozdíl se však bude zvětšovat, pokud budou data nalezena ve více souborech najednou nebo jich bude více. V tom případě by bylo největším problémem vyhledat postupně v souborech požadované výskyty a vykopírovat je do zvláštního souboru. Již na takto jednoduchém příkladu je patrná konečná úspora času a zdrojů při korektním použití aplikace E4Logsis.

6.2 Scénář 2

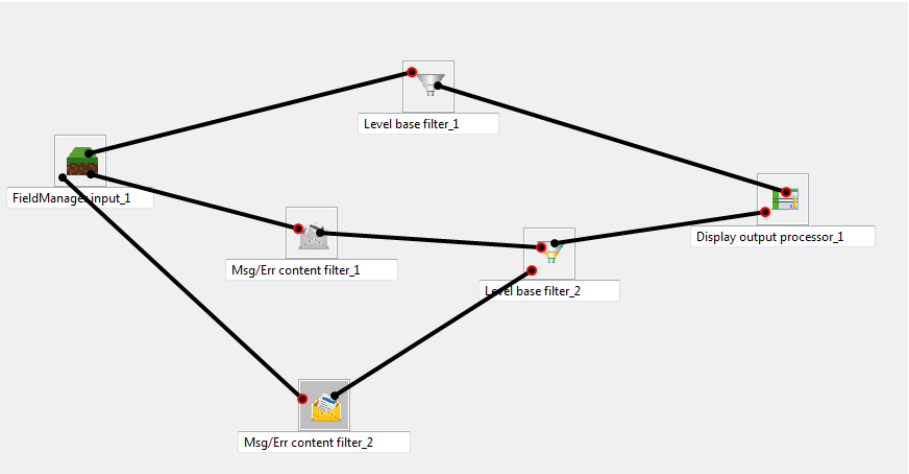
V klientské aplikaci dochází k chybě zvané *resynchronizace*. Zpravidla se jedná o chybu vazeb mezi objekty uloženými v databázi, překročení maximální velikosti sloupce tabulky apod. Chyba je uživateli oznámena otevřením dialogu se zprávou o chybě dat.

6.2.1 Sestavení scénáře v E4Logsis

Jediná známá informace je, že došlo k resynchronizační chybě, která měla za následek restart klientské aplikace. K dispozici poskytl zákazník čerstvé klientské logy pouze z daného dne. Tím se sníží náročnost celkové analýzy, neboť dat je řádově méně než u prvního scénáře. Vzhledem k malému množství informací poskytnutých zákazníkem bude vhodné nejdříve najít některé základní informace ohledně chyby. Protože se data ze souborů načítají jen jednou, může být v počátku vytvořen jen jednoduchý scénář a ten později rozšířen bez zbytečné časové náročnosti. Pro rychlejší analýzu použijeme výstup na monitor s možností filtrování a řazení získaných dat.

V prvním kroku byly vyfiltrovány všechny záznamy úrovně *ERROR*. Ve výsledcích zobrazených v dialogovém okně se dá nyní snadno dohledat, který objekt způsobil chybu. K této operaci byly použity procesory *FieldManager input* (čtení záznamů ze souborů), *Level*

[19] http://www.x-diesel.com/



Obrázek 13: Schéma umístění komponent - Scénář 2

base filtr (vyfiltrování ERROR záznamů) a *Display output* procesor (zobrazení výsledků na monitoru).

Každý objekt je označen jednoznačným identifikátorem, který je vhodný při další analýze. V následujícím kroku byl do scénáře přidán procesor *Msg/Err content* filtr. Ten bude navazovat na vstupní procesor a jeho výstupní konektor bude připojen do procesoru *Level base* filtr. Pomocí *Msg/Err content* filtru budou prohledány všechny záznamy a jejich obsah proměnných reprezentující zprávu a výjimku. Vyhledávaným řetězcem bude v tomto případě identifikátor objektu získaný v předchozím kroku. Úrovňový filtr tentokrát vyfiltruje záznamy vytvořené s úrovní *ERROR*, *DEBUG* a *INFO*. Po spuštění a úspěšném dokončení procesu již lze dedukovat přibližnou příčinu chyby. Je patrné, že chybu způsobila aktualizace objektu, kdy jeden z jeho povinných atributů byl aktualizován hodnotou *NULL*.

Konkrétní příčinu je nyní možno odhalit novým prohledáváním s rozšířením *Msg/Err content* filteru o typ objektu povinného sloupce chybového objektu. Z dalšího výpisu je již jasné, že ze strany serveru přišel na klienta požadavek ke smazání objektu, na něhož se odkazuje jiný objekt. V reálu by byly vyžádány i serverové logy zákaznického serveru.

Všechny tři kroky mohou být v aplikaci E4Logsis vytvořeny najednou díky možnosti zakázání některých procesorů. Konfigurace procesorů na canvasu by mohla vypadat přibližně jako na obrázku 13.

7 Možnosti budoucího vývoje

Vývoj aplikace není vypracováním diplomové práce u konce. V plánu jsou další úkoly, které by mohly být vypracovány, aby se aplikace stala více variabilní.

Jedním z těchto plánů je zabudování plánovaných úkolů tak, aby mohla být aplikace nasazena např. na serveru, kde by podle určeného rozvrhu analyzovala a zaznamenávala data běžících aplikací. Mohl by být implementován vstupní procesor, který by reagoval na přetočení log souborů a započal by analýzu dalšího kompletního log souboru.

Dalšího vylepšení by mohlo být dosaženo přidáním podmínkového procesoru, který by umožňoval uživateli vložení vlastní procedury. S vydáním Javy 8 se zde nabízí možnost použití jednoduchých lambda funkcí. Mohl by být také vytvořen separátní projekt, který by sdružoval vývoj procedur a funkcí pro podmínkové procesory nad daným datovým typem. Poté by jeden z pluginů představující podmínkový procesor sám využíval funkce připravené také jako pluginy. Ty by bylo možné instalovat a aktualizovat podobným způsobem jako procesory.

Stejný způsob by mohl být použit i u vstupních procesorů, kde by byla implementována funkcionalita dovolující pomocí specifikovaného regulárního výrazu analyzovat jakákoliv jednoduchá data bez nutnosti vytvoření vlastního procesoru. Bude zde však vždy existovat nebezpečí, že bude zpracování extrémně pomalé (zvláště pro větší objemy dat).

Bude implementován Eclipse plug-in dovolující vytvoření kostry implementace procesoru jako nového projektu.

Zjednodušením hlavních grafických částí programu (canvas a na něm umístěné položky) by mohlo být jejich přepracování tak, aby byly použity komponenty projektu GEF, resp. GEF4[20]. Ten existuje a funguje pro platformu Eclipse 3.x. Verze umožňující vývoj na platformě Eclipse 4 prozatím není dokončena. Hlavní výhodou by mělo být nativní použití SWT komponent, jejichž ekvivalent bude existovat i pro implementaci pomocí RAP.

Aplikace bude rozšířena o možnost použití více vstupních procesorů v rámci jednoho procesu. V některých případech je vhodné analyzovat data z více zdrojů najednou. Bude vytvořena nová komponenta unifikující data z více zdrojů do jedné společné kolekce. Procesory budou číst data v oddělených vláknech.

Užitečnou funkcionalitou by mohla být implementace vyhledávacího vstupního procesoru, který nepřevádí všechna data na objekty, ale pracuje pouze s objekty, jejichž tělo např. obsahuje předdefinovanou hodnotu. Systém by fungoval tak, že by byla nakonfigurovaná hodnota nejdříve nalezena ve vstupních souborech. Při každém nálezu by procesor teprve převedl reálný objekt z log souboru na objekt aplikace E4Logsis. Tímto způsobem by bylo mnohonásobně urychleno vstupní zpracování souborů, avšak na úkor neúplnosti zpracovávaných dat.

Na výstupních procesorech by bylo možno nastavit, zda se mají aktuálně připravená data zapamatovat i při zpracovávání nových souborů. Tím by se dalo dosáhnout funkcio-

[20] https://wiki.eclipse.org/GEF/GEF4

nality, kdy by aplikace četla data postupně z více zdrojů, avšak všechna by byla dostupná v konkrétním výstupním procesoru.

7.1 Podpora spouštění aplikace v prohlížeči

Jednou z alternativ, jak by mohl být program používán, je jeho nasazení na serveru a ovládání přes webový prohlížeč. V rámci eclipse platformy ve verzi 3 bylo možné spouštět desktopové aplikace i ve webovém prohlížeči. K tomu slouží RAP[21] platforma vyvíjená na základě platformy RCP. Ke správné funkčnosti bylo potřeba provést v programu, v závislosti na složitosti softwaru, více či méně zásadní změny. Hlavními změnami však bylo použití jiných závislostí např. místo balíku `org.eclipse.ui` bylo nutné použít `org.eclipse.rap.ui`. To bohužel pro verzi Eclipse 4 není prozatím možné. Příprava RAP pro eclipse 4 je sice v procesu, ale stabilní verze by měla podle plánu vyjít až 25. června 2014[22] společně s vydáním stabilní verze Eclipse 4.4 Luna. Problémem však stále bude vlastní implementace grafických komponent canvas a processor, pro něž v RAP neexistuje žádná alternativa. Aby tedy aplikace E4Logsis fungovala v prohlížeči, bylo by třeba implementovat RAP verzi těchto dvou komponent.

[21] Remote Application Platform (původně Richa Ajax Platform)
[22] https://wiki.eclipse.org/Luna/Simultaneous_Release_Plan

8 Závěr

V diplomové práci jsem se věnoval implementaci softwaru E4Logsis sloužícího k analýze záznamů Java aplikací. Program byl vytvořen nad platformou Eclipse 4 RCP, což zajišťuje jeho jednoduchou rozšiřitelnost o dodatečné moduly. Zásadní myšlenkou vývoje je možnost instalace business procesorů pracujících nad vstupními daty tak, aby bylo dosaženo potřebné funkcionality. Pokud je v rámci vývoje aplikace nutné řešit její chyby v rámci celého týmu vývojářů, mohou být procesory zveřejňovány tak, aby vznikala jakási jejich databáze, s jejíž pomocí se stává samotná analýza postupem času snadnější a komfortnější.

Oproti zadání, ve kterém byl navržen rozbor souborů na základě konfigurace logovacího frameworku, jsem tuto funkcionalitu změnil v řešení 'na míru' především z výkonnostních důvodů. V původním řešení jsem počítal se čtením celých vstupních souborů pomocí regulárních výrazů. Celý proces byl velice rychlý, pokud existovaly záznamy omezené délky odpovídající složeným regulárním výrazům. Jakmile však délka zprávy vzrostla, doba rozboru se neúměrně zvyšovala. V mém případě trvalo zjištění, že zprávu o velikosti asi 800kB není možné pomocí předepsaného regulárního výrazu rozdělit, asi 4,5 minuty. Zvolil jsem proto způsob, kdy je potřeba definovat méně obecné procesory s předdefinovanými regulárními výrazy. Vstupní data jsou nejdříve rozdělena podle jednoduchého regulárního výrazu (např. časový údaj na začátku řádku) do jednotlivých zpráv. Z těch jsou následně přečteny ostatní vlastnosti zprávy. Celý proces je enormně rychlejší než předchozí způsob právě díky předem známé délce jednotlivých záznamů. V mém případě došlo ke snížení časové náročnosti z devíti minut na 25 vteřin. Změna způsobu čtení záznamů s sebou přináší i jedno pozitivum, neboť mohou být čtena vstupní data bez ohledu na jejich původ.

V aplikaci jsou implementovány procesory pro základní práci včetně implementace mechanismu dovolujícího instalovat další komponenty. Čtení zdrojů je implementováno připojením kanálu přímo k danému souboru. Jedná se tak o nejrychlejší a nejúčinnější způsob.

Grafické rozhraní je implementováno pomocí SWT a JFace frameworků. U dvou grafických komponent tvořících část zobrazení jednotlivých procesorů na obrazovce je použita vlastní implementace třídy `Composite`. Ukládání a otevírání projektů je prováděno serializací resp. deserializací za použití tříd `ObjectOutputStream` a `ObjectInputStream`. Všechny instance objektů použité při činnosti procesoru musí být serializovatelné nebo nesmí být zařazeny do ukládaných položek (objekty, jenž se dají rekonstruovat po načtení serializovaných procesorů).

Úspěšnost implementace je nakonec otestována použitím aplikace E4Logsis na dvou praktických příkladech. Následuje srovnání časové náročnosti s dosavadním manuálním procesem, jehož závěrem je, že použití aplikace s nainstalovanými výchozími procesory značně usnadňuje proces analýzy chyb. Dá se také očekávat, že s přibývajícím počtem dalších procesorů se bude účinnost, rychlost a komfort používání dále zvyšovat.

Plánuji další rozvoj a údržbu programu závislou na rozsahu reálného použití. Po dalším otestování funkcionality v praxi bych software rád nabídl jako Open Source program pro analýzu logů. V tom případě bude nutné vypracovat podrobnou dokumentaci tak, aby bylo pro uživatele použití co nejsnadnější a nejpřímočařejší.

9 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], 2012, 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. *Vogella* [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=

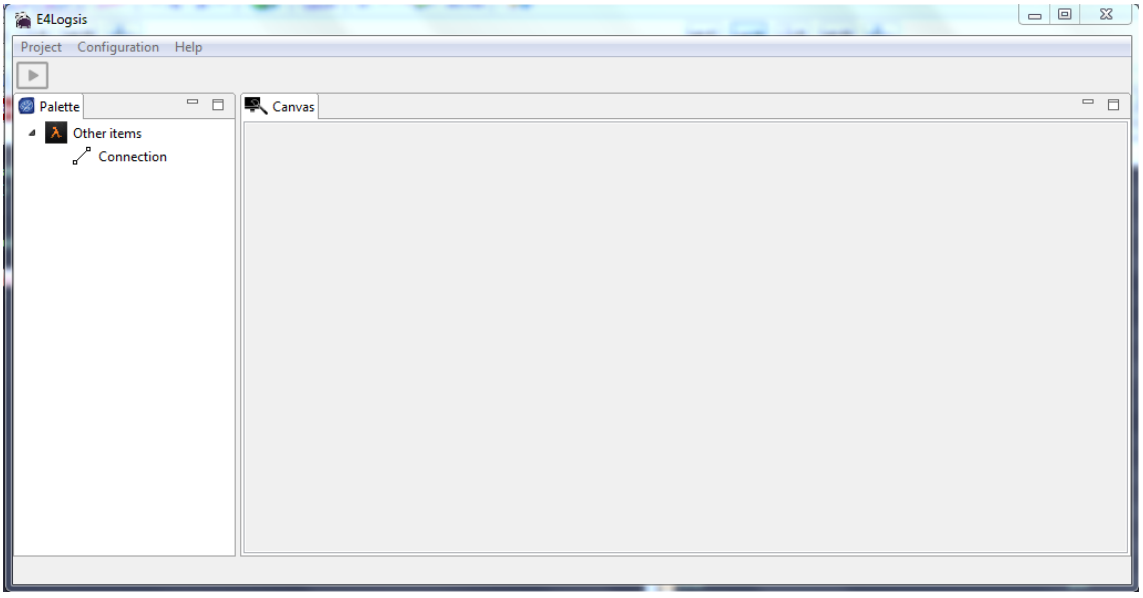
[11] JFace Data Binding. THE ECLIPSE FOUNDATION. *Wiki.eclipse.org* [online]. 2005, 2013-08-14 [cit. 2014-04-07]. Dostupné z: https://wiki.eclipse.org/JFace_Data_Binding

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

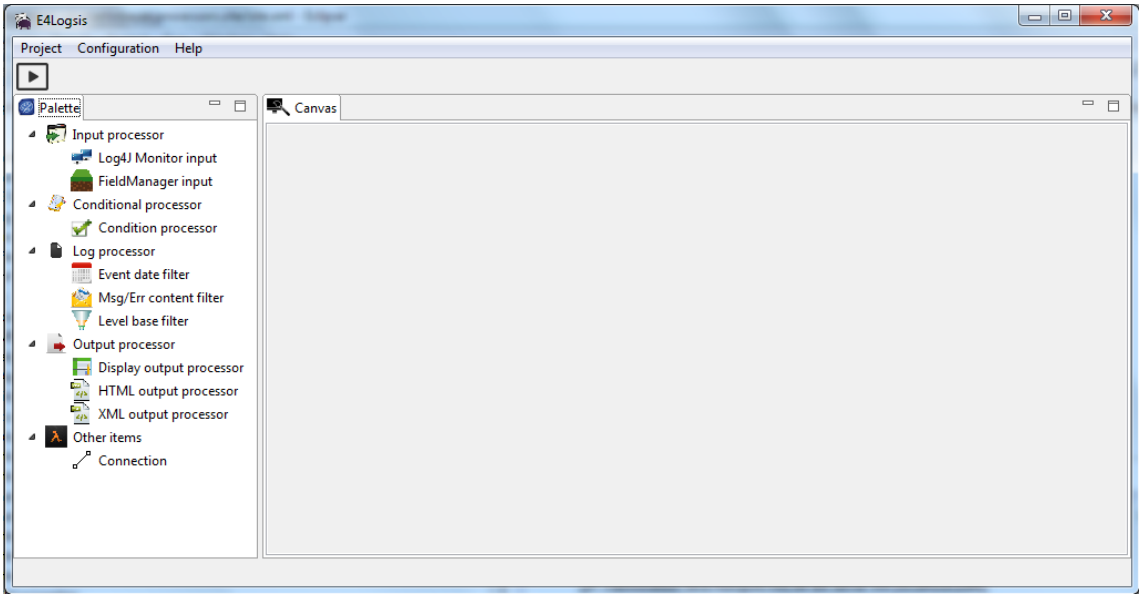
[13] Chain of Responsibility. *Design Patterns* [online]. 2001 [cit. 2014-04-07]. Dostupné z: http://www.oodesign.com/chain-of-responsibility-pattern.html

A Ukázky prvků uživatelského rozhraní aplikace E4Logsis

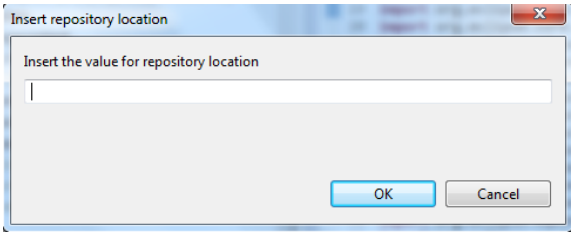
V příloze jsou zobrazeny příklady uživatelského rozhraní včetně potřebných dialogů sloužících ke konfiguraci jak procesorů (pluginů), tak celé aplikace. Z jednotlivý obrázků lze získat náhled na reálné použití aplikace.



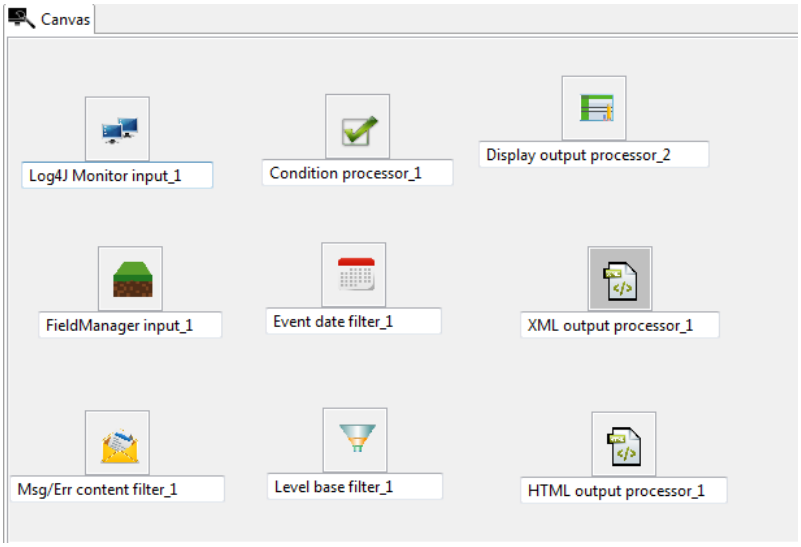
Obrázek 1: Aplikace bez nainstalovaných procesorů



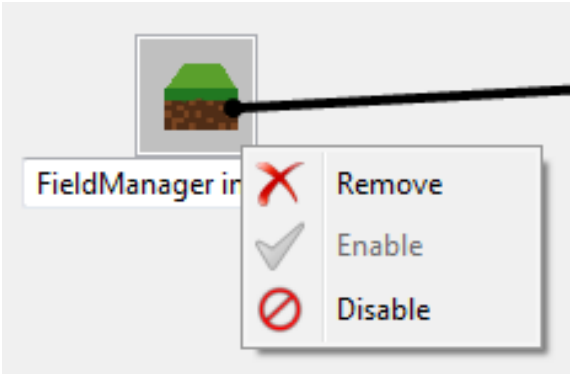
Obrázek 2: Aplikace s nainstalovanými procesory



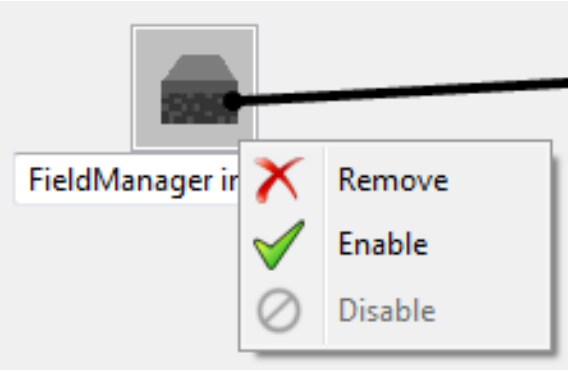
Obrázek 3: Konfigurace lokace s instalovatelnými pluginy



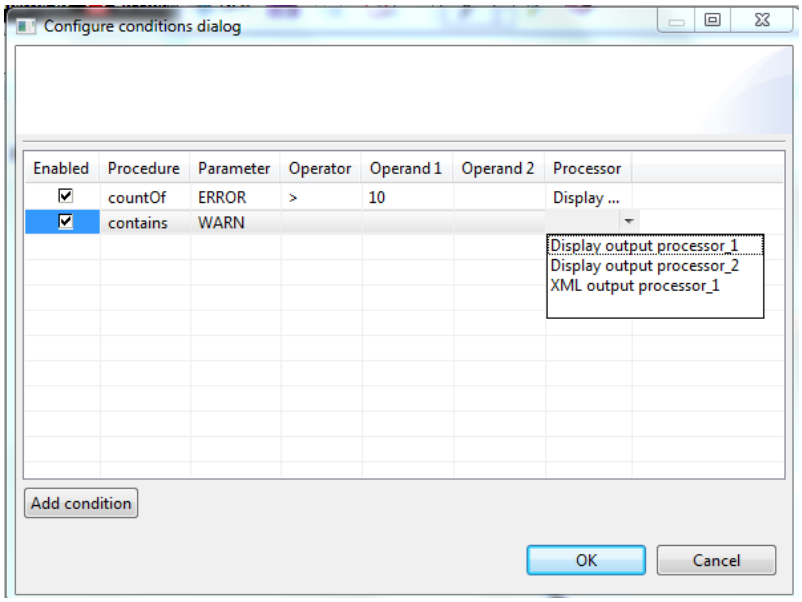
Obrázek 4: Canvas s umístěnými procesory



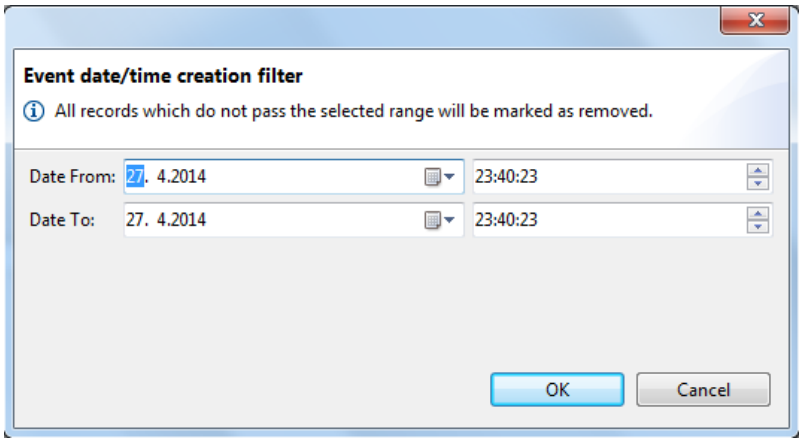
Obrázek 5: Kontextové menu aktuálně povoleného procesoru



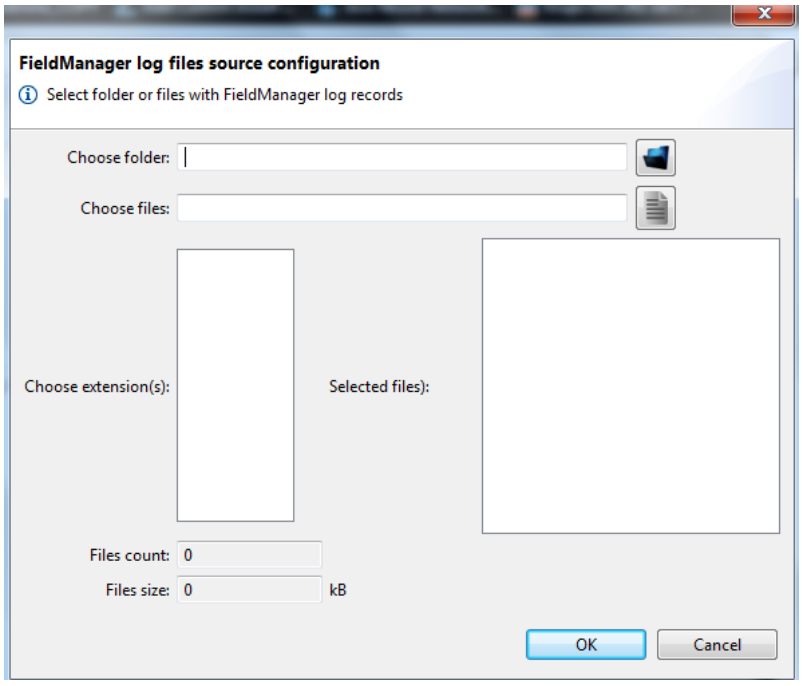
Obrázek 6: Kontextové menu aktuálně zakázaného procesoru



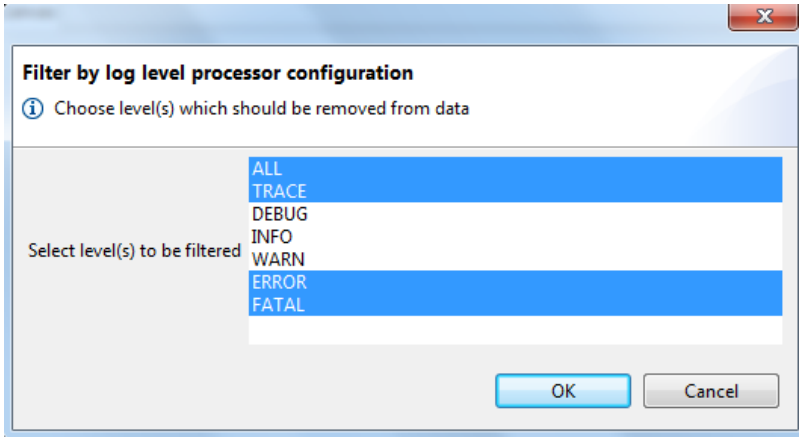
Obrázek 7: Konfigurace podmínkového procesoru



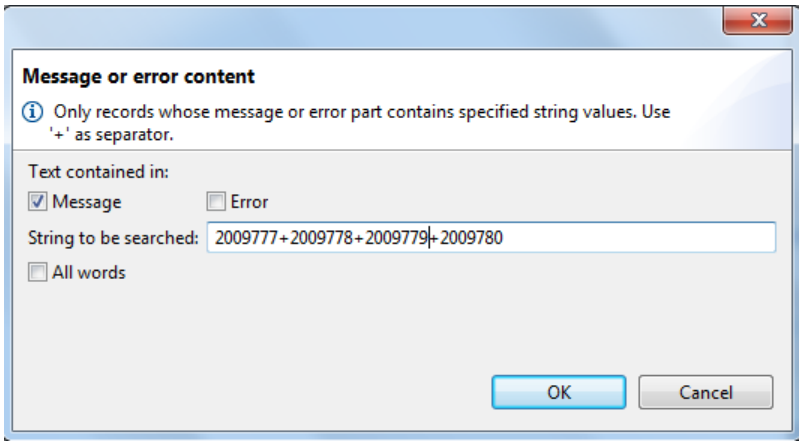
Obrázek 8: Konfigurace časového filtru



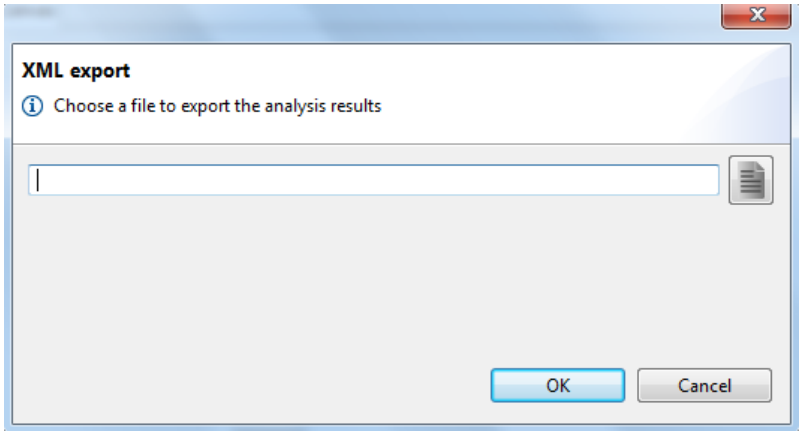
Obrázek 9: Konfigurace jednoho z vstupních procesorů



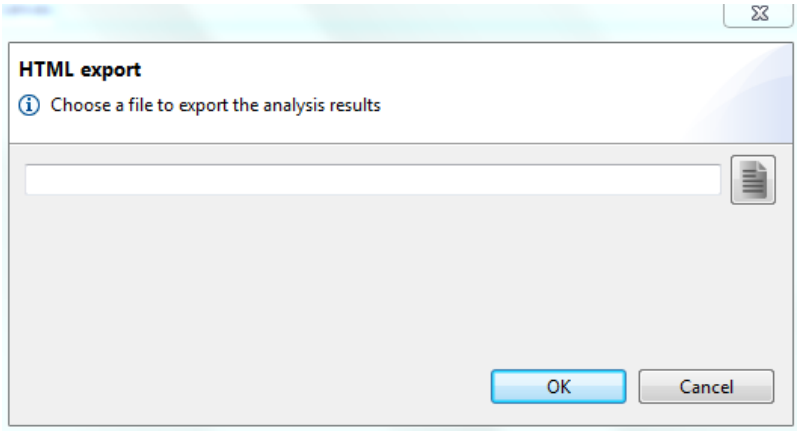
Obrázek 10: Konfigurace úrovňového filtru



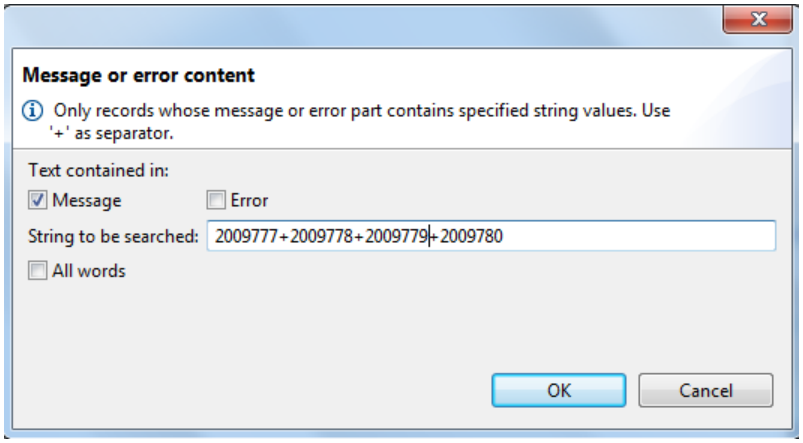
Obrázek 11: Filtr na základě obsahu zprávy



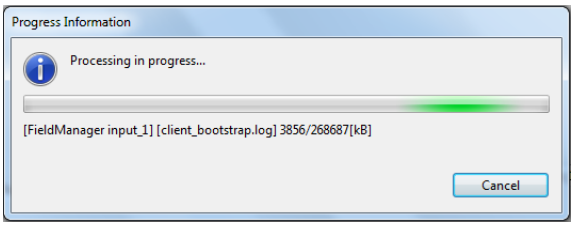
Obrázek 12: Konfigurace výstupního procesoru pro XML



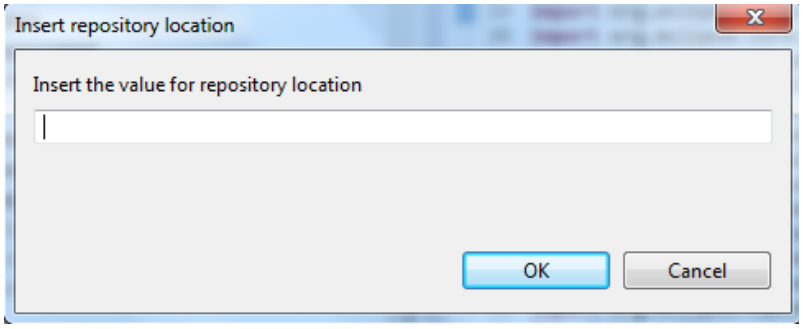
Obrázek 13: Konfigurace výstupního procesoru pro HTML



Obrázek 14: Filtr na základě obsahu zprávy



Obrázek 15: Znázornění probíhajícího procesu



Obrázek 16: Konfigurace lokace s instalovatelnými pluginy