Platforma průmyslové spolupráce

CZ.1.07/2.4.00/17.0041

Název

Logging Statements Rewrite Tool for Java Language

Popis a využití

- best practice pro tvorbu logovacích zpráv
- výuka: pokročilá Java

Jazyk textu

- anglický

Autor (autoři)

- Michal Tóth

Oficiální stránka projektu:

- http://lasaris.fi.muni.cz/pps

Dostupnost výukových materiálů a nástrojů online:

- http://lasaris.fi.muni.cz/pps/study-materials-and-tools

Contents

1 Introduction

Nowadays technology era full of information, there is always an urgent need to collect correct information in the shortest possible time. Similar tendency applies to software applications, as well. We want to know what has happened at the moment X and why part Y has failed. That is why applications during their runtime generate some kind of control information about their past and current state, and what have they done, or what are they going to do in their next steps. This information is generally called *logging messages* or *events*.

These events have been part of our work since the beginning okof creation of computer programs. Sometimes they are misused in a form of a quick substitution for debuggers. It is always a great feeling, when you see and you know, what exactly your application is doing at the moment, as well as how it deals with planned or unplanned situations. On the other hand, application can generate a lot of redundant information, which is not needed at the moment. Therefore we have to move them into some kind of *buckets* and prioritize them. These priorities are mostly knows as *levels*. This might seem to be a good solution, yet there are still problems with a lot of finding and searching for exact thing, which can be metaphorically compared with a looking for a needle in haystack. For people, log information about time and circumstances of the events, expressed by means of natural language, sentences, has always been satisfying. However, for searching in it by computers, it was a very hard task to do.

In situations, when one company had well-structured natural language logs of their product and another company had them structured as well, yet in a different way, cooperation between two products of the above mentioned enterprises was highly complicated. In this case, it becomes impossible for both products to process logs simultaneously. That is the reason, why structured logging came in eventually. It stresses the importance of readability of log information for computer, rather then for humans. Natural language logs are generalized to the most vital information and the rest is discarded. Also, the need for cloud based logging system is alarming as well, because of excessive exchange of redundant information and overflowing of networks with unusable data.

This is when the NGMON Logger, a structured logging framework appears and offers a solution. Although, NGMON Logger has to be able to understand application. Hence application has to be rewritten to NGMON Logger's syntax, when we need to use structured logging. The goal of this thesis, is to design and implement a tool for rewriting of Java application's logging framework. This logging framework

would not run in any Java Virtual Machine, so technically, it would be a set of text files, which will have to meet the needs of NGMON Logger.

The thesis is organized as follows. Chapter 2, *Logging systems*, presents an overview of the most important logging frameworks, which have been used in Java language in recent years. Chapter 3, *Overview of used tools for implementation*, introduces tools, which have been found appropriate during the search for possible methods of static code translation and creating of *LogTranslator* application. Chapter 4, *Implementation of changing logging system*, describes the process of the search for correct solution for rewriting problem, defined at the beginning of chapter, and the way it affected our selection of tools, used during the research. We also present our implemented *LogTranslator* solution in this chapter. In the Chapter 5, *Testing LogTranslator on Apache Hadoop*, we define what needs to be done in order to incorporate *LogTranslator* into your Java application. We also show some excerpts from example run of LogTranslator. In the end of the chapter, a brief overview is provided of performance of the Apache Hadoop using NGMON Logger, instead of default logging frameworks. In the last chapter, *Conclusion*, we summarize the overall findings of the thesis.

2 Logging systems

Logging is a fundamental part of any application. It is not commonly used directly by customer, but it is vital for supporting and further developing and maintenance of an application. It would be highly time consuming to search for errors in an application without knowing what is happening within it. That is why some part of coding is reserved for logging of states and actions, performed by application.

Mainly, when developing an application, we should focus on what information we should log. Events, such as start, stop, restart of module, various security information, resource management, performed actions like http requests, responds, queries, triggered actions and errors, which could lead to possible problems in the application. For all mentioned events, we should use some kind of logging mechanism, which stores these messages into a file or just outputs them to standard output stream. Such mechanism could be a purely custom application, simple outputting information using *printing* support of any programming language, or use a proper logging framework.

Why should we use logging framework? Why is it not sufficient to use plain simple *System.out.println()*, *System.out.format()* or in case of errors *System.err.println()*? To name just a few advantages in favor of logging frameworks:

- ability to control granularity of logs from verbose level to completely disable logging output,
- possibility to conveniently change log output destination (file, console, remote socket, JMS, GUI components, ...),
- keep singular and manageable structure of logs throughout the whole application,
- ease of log reading and association with appropriate location and context of application,
- possibility to completely change logging framework without changing application log statements.

Generally, any log message should contain answers for questions such as who, when, where, what and have a result of performed what action. In the following sections, we talk solely about the Java programming language and tools available in this language.

2.1 Java logging systems

In the world of Java programming language, there are many different logging systems. Beginning with simple specification - Java Logging API, which is presented in Java Standard Edition since version 1.4, through the popular frameworks such as Apache Commons Logging, Apache Log4j to newer frameworks, which take into consideration the feedback accumulated over the years, about what works, what does not and what should be done better, and also adjusting to the needs of developers and demands of technology advancement. Observation shows that approximately 4% of code is dedicated to logging[1], so it would make sense to choose logging framework wisely to your needs. In the following sections, we provide a brief overview of Java Logging API, Apache Commons Logging, Apache Log4j projects, Logback and Slf4j logging frameworks.

2.1.1 Java Logging API

One of the first logging systems for Java was officially specified in JSR-047 *Java*^{TM} *Logging API Specification* [1] with final release in 2002 for Java version 1.4. Core functionality of this logging API is provided by classes and interfaces in *java.util.logging* package to support maintaining and servicing software for customers[2]. Specification describes key functionality elements, as well as targeted log use cases, which led to creation of specification itself. Following key objects of Java Logging API are:

Logger - hierarchically organized entities for an application or a specific system to log messages by making individual logging calls.

LogRecord - passes logging requests between logging framework and log handlers.

Handler - exports logRecord object to its destination - memory, output stream, console, file, socket, etc.

Level - set of standard logging levels for controlling application logs output.

Filter - provides more fine-grained control of logging, than levels. Each handler and logger can have their own associated filter.

Formatter - provides support for formatting a logRecord object and converting it to the string.

As stated before, Java Logging API has been created with respect to some specific use cases. In our case of log usage, it means that we want to be able to identify, what problem and where has occurred by going through the logs. There was a significant demand for such *problem diagnosis* feature, performed by various user groups, namely:

---

1. According to `http://logging.apache.org/log4j/1.2/manual.html#Configuration`
2. `http://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html`

End users and system administrators - simple logging of common problems, which can be fixed or tracked locally - security, configuration or resource running errors

Field service engineers - generally such engineer needs more complex and verbose logging then system administrator, to find specific subsystem error and possibly even fix it on site

Development organization - after field service engineer finds an error on site and passes the verbose logs to its development organization, it would also like to fix particular error in their system for all customers, so they should have easily traceable and very detailed logs for fixing it

Developers - during development phase of a project it is always very convenient for developers, apart from various debugging and profiling tools, to have some extra information (logs), generated either by developed project or another cooperating system

Java Logging API supports internationalization, remote access, serialization and security matters with main requirement principle, stating that *"untrusted code should not be able to change the logging configuration"*[3].

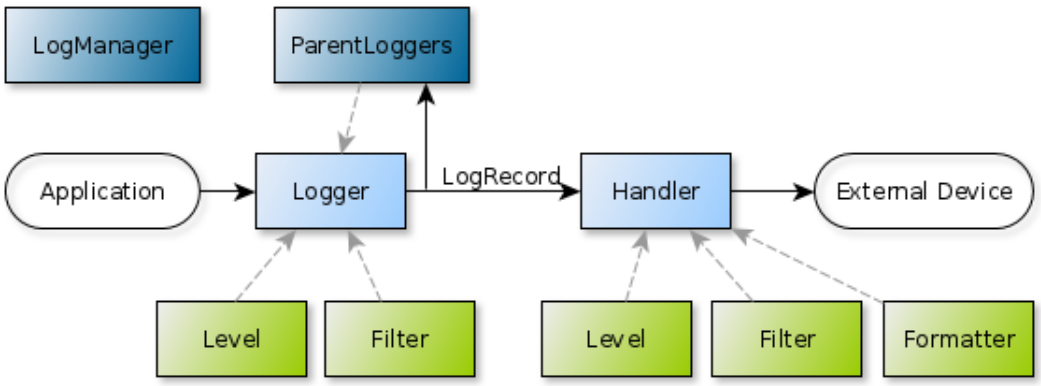

Figure 2.1: Structure model of Java Logging API.

In Figure 2.1 we can see the model structure of Java Logging API. We will describe the process [2] of how logs are handled and modified since their creation until sending them to some *output device*. This process is used almost in any other

3. Java Logging API overview, Section 1.14 Security on [3]

logging framework with some minor or major modifications.

In the beginning, global LogManager is created which subsequently creates and manages all the Logger objects. Those are main objects from user perspective, on which he can call logging methods. Loggers are hierarchically organized in tree structure, beginning with root node as empty string and typically continuing with child nodes as package and module names[4]. Level associates log event level with Logger handling given object. User can also assign Filter object to particular Logger for more fine-grained control of logs. LogRecords are created by Logger and forwarded to at least one accessible Handler. Handler can have associated Level, Filter and Formatter object. First two have the very same meaning as with Logger. The third, formatter, is responsible for converting a log event to the string output and later can be formatted by pattern defined as an XML or in a custom configuration file. Final step is exporting the formatted log event to appropriate device by Handler. Supported output devices by Java Logging API[5] are console, stream, file, socket or memory.

2.1.2 Apache Commons Logging

Being light-weight and having a high level of abstraction for use of independent logging toolkits, are two main goals of Apache Commons Logging, abbreviated as JCL (Jakarta Commons Logging)[3]. Having single abstract layer of *Log* and *LogFactory* interface, JCL allows user to choose from various logging implementations. JCL offers user the ability to plug-in Log4j, Avalon LogKit, Java Logging API implementation or any other logging system by simple adding of *commons-logging.jar* package to the classpath. In some special cases it is also needed to provide an extra information to *commons-logging.properties* file as well. In a rare situation when JCL is not able to pick up any logging library, it will silently fallback to default simple logging wrapper. Apache Commons Logging standard distribution contains following jar packages:

commons-logging.jar - contains JCL API, default LogFactory and Log implementation for Log4J, Avalon LogKit, JDK 1.4 and pre-JDK 1.4 logging implementation. In most cases, adding only this jar package to classpath should be sufficient to successfully run JCL.

commons-logging-api.jar - this jar package is intended for use in projects, where one has to recompile commons-logging using alternative Java environment

4. Logger hierarchy follows the unwritten naming rule of giving names based on your company reversed website url, project name, modules, sub-modules, etc ...
5. Implementation is in *java.util.logging* package since Java 1.4 version.

and is unable to compile with optional libraries provided by Apache release of common-logging. This package contains minimum possible dependencies.

commons-logging-adapters.jar - includes only adapters to third-party log system implementations. It can not be used on its own, because it does not contain core commons-logging framework.

Log message priority levels follows specifications set by JSR-47[6]. Apache Commons Logging framework offers following priority levels[7], ordered by severity according their user guide [3]:

fatal - Severe errors that cause premature termination. Expect these to be immediately visible on a status console.

error - Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console.

warn - Use of deprecated APIs, poor use of API, 'almost' errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console.

info - Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep their number to minimum.

debug - detailed information on the flow through the system. Expect these to be written to logs only.

trace - more detailed information. Expect these to be written to logs only.

All of these methods have their counterpart in *is<Level>Enabled()* to save log processing overhead by skipping the evaluation of log. This is very useful in cases where log contains many variables which have to be resolved or various types of concatenating expressions and strings have to be made by compiler[8]. Another way of saving disk and processing, is by setting the level of log statement and effectively limit the log verbosity by specifying lower boundary of levels. As log levels are ordered[9], by setting level to *warn*, only levels higher then *warn* will be processed, those are *error* and *fatal* levels.

---

6. JSR 47: Logging API Specification - described in 2.1.1
7. We can consider these explanations as guideline for almost any logging system, using priority levels for logs.
8. This used to be a problem, but since Java 1.5 release, compiler can cope with string concatenation very easily by changing String's "+" concatenation to StringBuilder's append method [4].
9. Typical order of log level is: trace < debug < info < warn < error < fatal.

Internationalization or *National Language Support* used by commons logging framework is supported via message file and message keys in logs. Also, they recommend to use internationalized messages for fatal, error, warn and info levels of log messages, to be easily understood by any system operator.
JCL is designed to offer creation of additional extensions of *Log* interface, particularly for implementation of new logging systems and *LogFactory* interface to provide strategies for discovering underlying logging frameworks.

By having a quick look at download section[10] of commons-logging, it is quite easy to see, that there is no advance in development of version 1.1 since release in 2006. The only visible progress is maintenance offering which seems to happen in middle of 2013 for version 1.1.3.

2.1.3 Apache Log4j

Another very successful and popular Java logging framework under Apache's wings is *Log4j*. Initial release written for Java 1.3 dates back to 1999 [5]. Current version 1.2.17 is maintenance only, as there has been no development releases from 2007 according to release history[11]. Originating from *E.U. SEMPER* project as tracing API and after many enhancements, iterations and evolution over time it became logging package for Java. It is ported to other languages like C, C#, Perl, Python, Eiffel.

Log4j is using the same concept as other logging frameworks - inserting log statements into source code. Provides precise context about running application for easy debugging and serves as auditing tool as well, because logging statements usually tends to stay in source code during whole lifecycle of application. Main focus had been put on simplicity, reliability, extensibility and fastness of logging framework during design. Simplicity has shown in the three main components *logger*, *appender* and *layout*. By working together they enable developer to set a given type, a level and a formatting of logging messages as well as how and where should log be reported to. Log4j supports same logging levels as Apache Commons Logging, listed in Section 2.1.2. There is also a support to define your own levels, but it is discouraged to do so.

*Logger*s are named entities, following hierarchical structure to selectively enable or disable logs. Having one root logger and many child nodes named after software components to control mentioned functionality with tied logging levels to loggers.

---

10. Download archive of Apache Commons Logging `http://archive.apache.org/dist/commons/logging/binaries/`
11. log4j release history `http://logging.apache.org/log4j/1.2/changes-report.html`

Appender in log4j's terminology is the output destination of log. Destination can be any of console, file, GUI component, remote socket server, JMS, NT Event Loggers and remote UNIX Syslog daemon. Logger can have more then one appender to append simultaneously to different locations. It is also possible to use asynchronous appender.

Customizing log's output format is used as often as choosing log's destination output. Formatting the log request according to user's demands is done by *layout*, associated with appender. Layout's class *PatternLayout* formats appropriately log using similar specification as C's function *printf*.

Configuring of Log4j for application usage could be made in two ways. First, using an xml or a Java property file and second in a less flexible way - fully programatically. Framework does not have special default initialization, which works as is. Specially Log4j has no default appender. Default initialization procedure looks for *log4j.configuration* system property, then falls back to default *log4j.properties* configuration file and loads resources from there. If there is no resource configuration found, then Log4j aborts default initialization procedure.

Performance issues are covered similarly as in Apache Commons logging by fencing log statements by *is<Level>Enabled()* methods. Great optimization came to log output formatting by layout module. Same effort has been put to appender modules. Typical cost of whole logging process is between 100 to 300 microseconds [5].

Its manageable usage - selectively changing logging output, turning logging on and off or simply changing output format of already finished application by configuring only one file is powerful feature and by minimizing performance cost it became pretty straightforward to use log4j in many projects. According to log4j's wiki page[12] development of log4j 1.3 has ended because of too many compatibility and maintenance issues with newer JDKs in preference to log4j 2.0. Log4j will have support for existing applications and for JDK 1.5 and lower.

Log4j 2 and Logback logging frameworks try to use and improve log4j based on new demands and ideas. We will have a look on them in following chapters.

2.1.4 Logback Project

Logback is conceptually very similar to its ancestor log4j, but brings many improvements over log4j. A main developer of Logback is the same one as popular log4j framework had. Logback's architecture is generic for various use cases. Designed of three main modules *logback-core*, *logback-classic* and *logback-access*. Logback-

12. Log4j wiki page `http://wiki.apache.org/logging-log4j/`

classic module can be thought of as advanced log4j. By natively implementing slf4j API, it is very easy to switch between logging frameworks. Integration with servlet containers like Tomcat or Jetty is handled by logback-access module. Finally, as name says logback-core provides main support for previously named two modules [6]. There is also another specific module[13] - *logback-audit*. It has a significant importance for storing long-term business event logs for auditing purposes.

Between the list of main benefits over older log4j belongs ten times faster implementation, heavy test suite to guarantee overall logback's stability, mentioned implementation of slf4j API, automatic reloading of configuration file upon modification, graceful recovery from I/O failures, automatic compression and deletion of old log archives, ability to write into single log file by multiple appenders, interactive event logging and access viewer, special mode for log event granularity and debugging purposes and last, but not least, the possibility to use *SiftingAppender*, which can log into separate user files defined by which user is running the application.

2.1.5 Apache log4j 2

As mentioned in previous chapter, original log4j project has been transferred to maintenance only, as developers had changed their focus towards developing log4j 2.0. Apart from main issue with compatibility with JDK 1.6+, there are improvements in asynchronous logging performance, multiple API support, automatic reload of configuration file upon modification without losing log events and plugin support for configuration of Appenders, Layouts, Filters, Lookups or Pattern Converters [7] over its ancestor *log4j1*. Project is still developed under open source license. Some of the advancements over Log4j and Logback:

- Significant improvement of lock-free asynchronous loggers using *LMAX Disruptor*[14] library. Ten times higher throughput in multi-threaded applications and much lower latency then log4j1 or Logback can be seen in Figure 2.2 taken from [7] section 12.1.5.1 Logging Throughput.
- Concurrent support of locking threads on low level, prevents occurring deadlocks.
- Ability to pass log event from layout to any Appender, not just to OutputStream, thanks to Layout's return type of byte array instead of String.

13. Actually it is an open source project on its own hosted by website `http://audit.qos.ch/`
14. LMAX Disruptor: A High Performance Inter-Thread Messaging Library - Nowadays considered to have the most effective way of sending messages between threads using ring buffer instead of queues. It was open-sourced in 2011. More information can be found in reference technical paper `http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf`

- Layout format can be more various then fixed form of log4j or Logback.
- Filtering of log events before they are passed to Logger module.

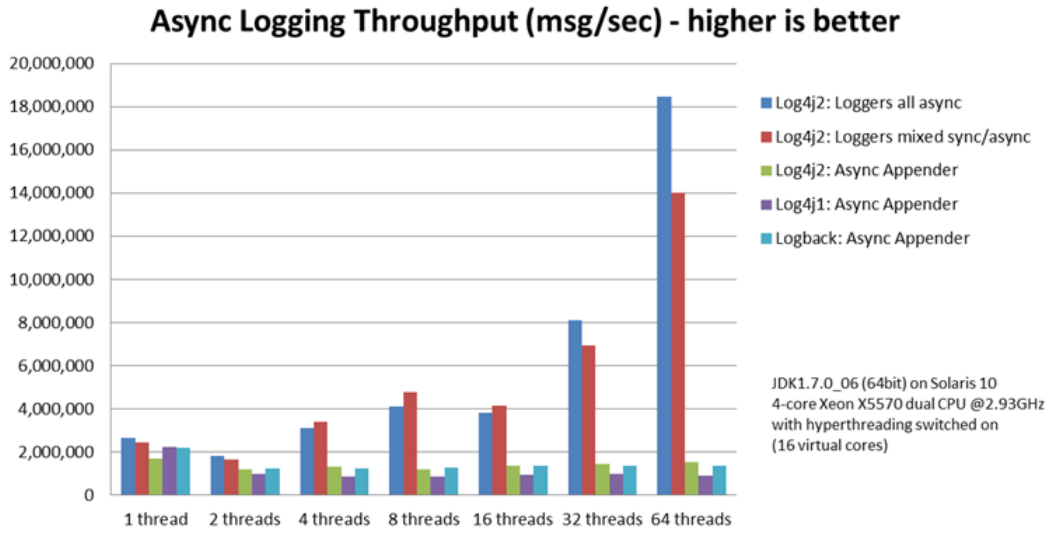

Figure 2.2: Log4j2 asynchronous logging throughput comparison chart with other logging frameworks.

Log4j 2 does not have backward API compatibility with log4j, instead it offers adapter implementation. One can also use log4j 2 using adapter for Apache Commons Logging or Slf4j framework to migrate his current logging framework.

2.1.6 Simple Logging Facade for Java

The Simple Logging Façade for Java (Slf4j) is an abstraction layer for various logging frameworks and provides an end user with the option to simply change logging framework implementation during deployment of application [9]. Officially supported frameworks are Java Logging API, Log4j version 1 and 2, Logback and JCL. Other frameworks can be supported by custom plugins.

To use slf4j for your application, you just have to add *slf4j-api-version.jar* as only mandatory package on classpath and optional slf4j binding for your desired logging framework to use. Only one binding can be used at a time, as you can use only one logging framework. In case of switching to another logging framework, only change needed to do is to switch appropriate binding in classpath of slf4j. Here is a list of supported bindings[15]:

15. In listing there is omitted version number of Java package for sake of simplicity, for example

slf4j-log4j.jar - binding for an older log4j version 1.2

slf4j-jdk14.jar - binding for java.util.logging or also known as Java Logging API

slf4j-nop.jar - default no-operation binding, silently discarding all log events

slf4j-simple.jar - outputting log events to standard error output stream, but only those with level higher then info

slf4j-jcl.jar - binding for Jakarta Commons Logging

logback-classic.jar - native implementation of binding for Logback means no additional computational and memory overhead

Figure2.3 distinctly depicts main idea and high level structure of how slf4j handles different bindings. If you choose not to use or slf4j does not find any binding on classpath, the default *slf4j-nop.jar* - no-operation module will be used and all logs will be discarded. This is a default behavior since version 1.6.0 and instead of throwing a *NoClassDefFoundError*, slf4j will warn you with single line warning message. Slf4j interfaces and adapters are very simple and understandable. They do not use class loading nor access class loaders for binding discovery.

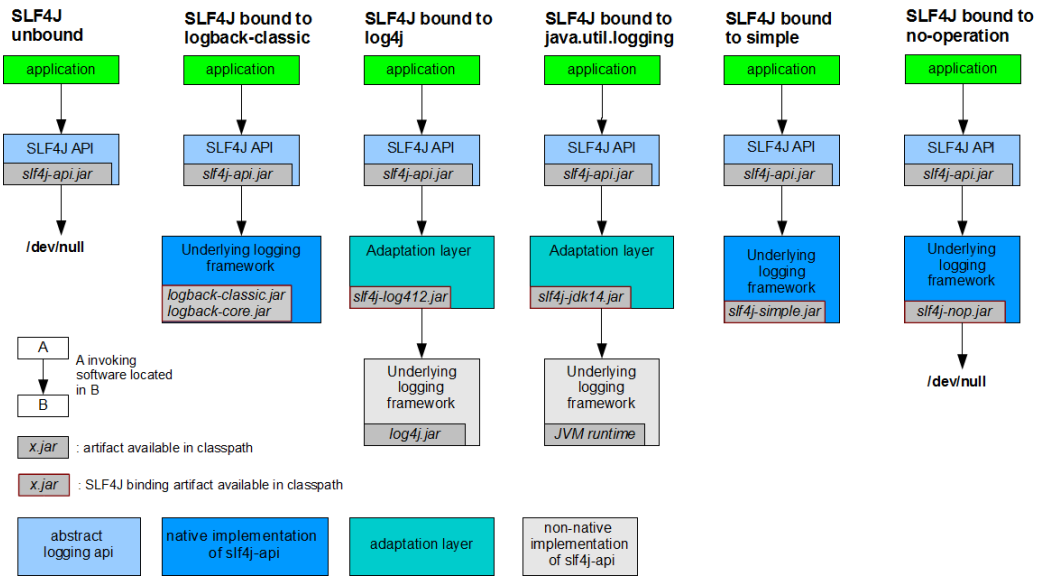

Figure 2.3: Various usage bindings for slf4j framework. Image taken from slf4j's manual[9].

slf4j-jcl-1.7.5.jar to *slf4j-jcl.jar*

One of many interesting features is capability of slf4j to bridge logging frameworks and bring them under control of one framework. That means, if project uses for example log4j and JCL framework, then slf4j can redirect log4j and JCL calls to slf4j and use later on like native slf4j calls[16] Better explanation provide Figure 2.4. For such cases one should use *slf4j-migrator* tool.

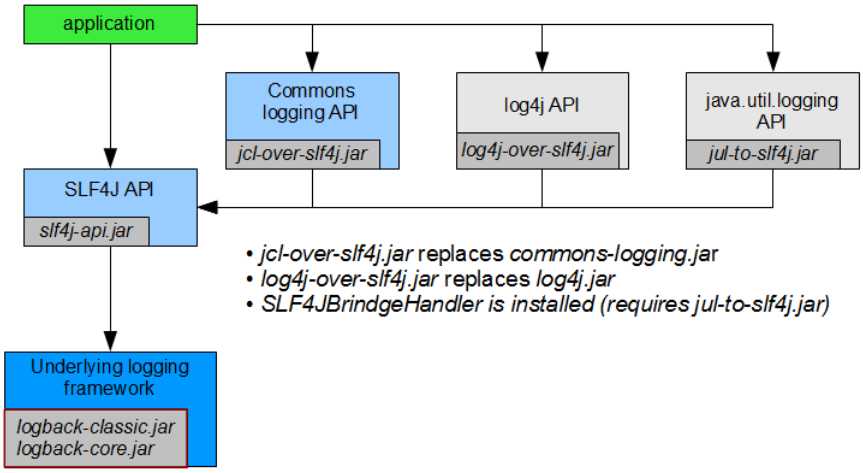

Figure 2.4: Slf4j bound to logback-classic with redirection of JCL, log4j and java.util.logging. Image taken from *Bridging legacy APIs* `http://www.slf4j.org/legacy.html`.

Slf4j also supports *Mapped Diagnostic Context*. In general, MDC is a key-value based map filled with application specific logging pairs. They can be inserted by framework into logging messages for future message filtering or action triggering. These extra logging information is very useful, specially in client-server applications, where log events can be easily associated with specific client by unique parameters. If underlying logging framework has support for MDC, slf4j will delegate these logging pairs to them.

2.2 Diagnostic and Audit logging

By the intended usage og log information, we can basically divide them into two main categories. The first category is termed as an audit logging and second, diagnostic logging. To clear things up, audit logging is a kind of logging, which developer

16. More information is available on project web page explaining bridging `http://www.slf4j.org/legacy.html`.

or system administrator does not need much for their work. On contrary, such information is vital for security reasons, auditing purposes and any customer, whose application is connected with money operation and high fluctuation of potential and contemporary customers like banks, insurance, government or electronic shop systems. Every interaction made with system should be logged and stored for future, in case something bad might have happened. That means, storing who, when and from where someone logged in and out of system and what transactions has he made during his visit are very important to know and log. If system is really crucial such user action transaction should not execute successfully, in case of inability to properly create and store appropriate transaction log.

On the other hand, diagnostic logging is not so important for customer (as service provider), but it is for developer and system administrator, mainly to better understand the flow of the application. Being able to figure out where is what problem and fix it quickly is one of the main goals for maintaining high availability and security of application.

Both of these categories can become very large in terms of disk usage during period of time. Of course, not every company chose to log everything and each having different policy of log rotation and storage. But if company has to store such large data for a longer period of time and also being able to search in them, it can become a complicated and significant task to tackle. Also collecting this data to analyzing unit is another aspect of this problem. Logging mechanisms currently used in logging frameworks to output to files or devices are becoming an obsolete solution, so we should look for new ways of handling these information.

2.3 Structured logging as opposed to natural language logging

Natural language logging is what we use in almost any logging framework today. All of the frameworks presented in Chapter 2 use from bigger part natural language in their logging statements. They give free will to developer to write anything in logging statement as there is no restriction on format and content because whole log will be converted to string after all. Of course, developers can use strict format of their log statements in projects to imitate *structured* logs after internal agreement, but it is mostly only the temporary solution to problem. Following snippets of code in Listing 2.1might be very familiar for reader.

Structured logging is one of possible solutions for problem, described in the last paragraph of Section 2.2. Structured logging in its very basic form is a mechanism of sending a structured message in a JSON or an XML format with given log information to central log storage. This message is created from log statement which form is designed to be read (parsed) by computers, rather then humans. Such sent

```
LOG.warn("Hadoop command-line option parsing not performed. " +
         "Implement the Tool interface and execute your" +
         "application with ToolRunner to remedy this.");
...
LOG.debug("default FileSystem: " + jtFs.getUri());
...
LOG.warn("No job jar file set. User classes may not be found."+
         "See Job or Job#setJar(String).");
...
LOG.debug("Configuring job " + jobId + " with " + submitJobDir+
         " as the submit dir");
...
```

Listing 2.1: Use of natural language in log statements. Example has been taken from Hadoop project - *org.apache.hadoop.mapreduce.JobSubmitter.java* file.

log is termed as event in structured logging terminology. Events can be send to various devices over conventional transport mechanisms. When event is delivered to its final destination, log storage or processor, it is conducted for further event analysis. Analysis of event is much easier, as processor already knows what data is held in obtained message and there is no need to figure them out using various parsing mechanisms for processing of natural language formatted logs.

Writing structured log statements is not a very popular choice for developers. In most cases it takes less time to write similar log statement in natural language. Syntax of structured statements is similar to following examples in Listing 2.2.

It seems like JSON format might be used more in future, because it is much faster in the means of time processing and easier to read for human then an XML file containing same data. On contrary its simple data types can be limiting in some use cases where XML would be favored. Following Listing 2.3 depicts examples of two already processed log events in JSON format, prepared for deliverance.

2.3.1 Common Event Expression

According to authors of log analysis study [10] conducted in 2002, there are few major problems reoccurring and preventing of better utilization of logs. These problems were summed up after deep examination of log records from web and digital library systems and namely they are:

```
val logItem = Json.obj(
      "@source" -> instanceID,
      "@fields" -> fields,
      "@timestamp" -> dtFormat.print(now),
      "@source_host" -> "mn.local",
      "@source_path" -> sourcePath,
      "@message" -> msg,
      "@type" -> eventType
    )
...
var Logger = require('testApp');
var log = new Logger({name: 'hello' /*, ... */});
log.info("Hello %s", "world");
```

Listing 2.2: Example of structured log statements used in source code.

```
{  "_index" : "logstash-2013.12.07",
   "_type" : "play",
   "_source" : { "@source":"test-log",
      "@fields": { "instanceID":"test-log",
         "request":"GET /test-log/index.html",
         "user-agent":"Mozilla/5.0",
         "httpCode":200,
         "duration_ms":123},
      "@timestamp":"2013-12-07T13:12:56.123Z",
      "@source_host":"local",
      "@source_path":"GET /test-log/index.html",
      "@message":"Index displayed",
      "@type":"INFO" }
 }
 ...
{  "name":"hello",
   "hostname":"hamster.local",
   "pid":3421,
   "level":20,
   "msg":"Hello world",
   "time":"2013-12-28T17:25:37.050Z"}
```

Listing 2.3: Examples of structured log messages in JSON format from logstash

ambiguity - semantic of log format entries is not proper to fully distinguish between different ones,

complexity of logs - analysis of bigger number of logs proved to be challenging,

disorganization - poor structure and organization of logs,

incompatibility - various forms of log formats thwarts possibility to use same analysis tools,

incompleteness - some of the key information were missing to fully understand particular log,

inflexibility - too much of system specific information to be used on other nodes or computers,

verboseness - overwhelm of redundant dumped system log information.

Collective will to eliminate mentioned issues and need for unified event representation and classification across multiple applications led to creation of specification - *Common Event Expression*[11] a unified event language for interoperability. Developed by cooperation of industry companies, end user groups, individual experts and U.S. government organizations. By unifying events, end users do not have to worry about writing their own adapters and log management becomes much easier to handle, as all events generated by event data producer are classified and have structure defined by vocabulary and common syntax.

Lifecycle of one event can be divided into six fundamental steps depicted in Figure 2.5 and defined as:

Describe - description of event is based upon given requirements

Encode - event is encoded into record

Send - record is transported to its destination

Receive - receiving of transmitted record by log storage or analyze processor

Decode - decoding of obtained record into event

Analyze - event is analyzed and compared with requirements which may lead into further issue investigation

The CEE standard describes a data structure for multiple profiles. A CEE Profile consists of two main components - a field dictionary and an event taxonomy. A field dictionary specifies field names with description and their associated names to use with this field. For example *"time"* as field name would have associated names *"timestamp"*, *"date"*, *"occurred"*. Event taxonomy is a vocabulary of event tags to provide consistent classification over whole organization and various applications. As vocabulary can be used an enumerated set of fields from field dictionary. CEE Profile use two primary profiles. First, function profile, can be applied on all applications

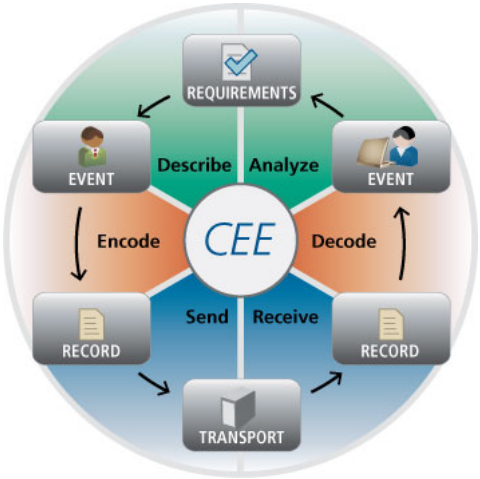

Figure 2.5: Event lifecycle in CEE standard using three main fragments - Requirements, Events and Records. Image taken from *About CEE - Archive* `http://cee.mitre.org/about/`.

of one type to have common log mechanism. Other, product profile, can define set of logging rules on one specific product.

To obtain higher goal, having same structure of events across multiple applications and organizations, CEE provides common, shared vocabulary and taxonomy as *Core profile*. This profile would need to be adopted by other communities and organizations to benefit from same field dictionary and taxonomy regardless of source of log event. In such case, the consumer would not need to worry about different event structure and could freely expect same event data across all applications sharing the same CEE profile.

Sadly, between years 2012 - 2013 sponsorship for Common Event Expression has been halted. Subsequently MITRE has stopped all their work on CEE standard. Although ideas still live in fedora-hosted projects Lumberjack[17] and ELAPI[18] - Event Logging API. Lumberjack focuses on updating and enhancing event log architecture as well as to improve creation and standardizing the content of event logs by implementing concepts and specification defined by Common Event Expression.

ELAPI provides an abstract layer for storing destination, may it be a file, syslog or database, for changing it to more advanced logging facility without need of touching existing logging application. It creates logs, which are easy to manage and are of various structure and complexity in contrary of having a one fixed log structure.

17. Lumberjack's project website `https://fedorahosted.org/lumberjack/`
18. ELAPI project website `https://fedorahosted.org/ELAPI/`

Posses the ability to represent the same data for human and computer by formatting obtained data appropriately in the most pleasant way and secures reliability of log storage, so important log events will not be lost, by defining a fail-order list of support destinations.

As a side-note Python natively supports structured logging and may output these logs in JSON format. Another interesting logging framework currently under development, operating with structured logs is experimental project New Generation Monitoring system.

2.4 New Generation Monitoring Logger

2.4.1 Overview of NGMON Logger

After presenting nowadays state of various logging frameworks, needs and problems around logging, community in Masaryk's University have concluded that there should be an answer for these questions by creating its own logging framework. New generation monitoring Logger (NGMON) system is an experimental logging monitoring project under heavy development by community led by Daniel Tovarňák. It aims to be lightweight, extensible and interoperable in terms of functionality and to provide single access point for multiple simultaneous tenants accessing local and remote monitoring information of applications in its scope. It is secured on multiple levels by fine grained access control list, write-protected by encryption of AES logarithm on database level. Having small memory and processing usage and maintaining high write throughput by having structured monitoring events object as main element is endorsed asset as well. Overall architecture overview of data flow of NGMON is depicted on Figure 2.6 taken from [12]. So far, NGMON is in Java language only, but further languages will be supported in future. All information about NGMON has been acquired from [12], [13] and [14] publications.

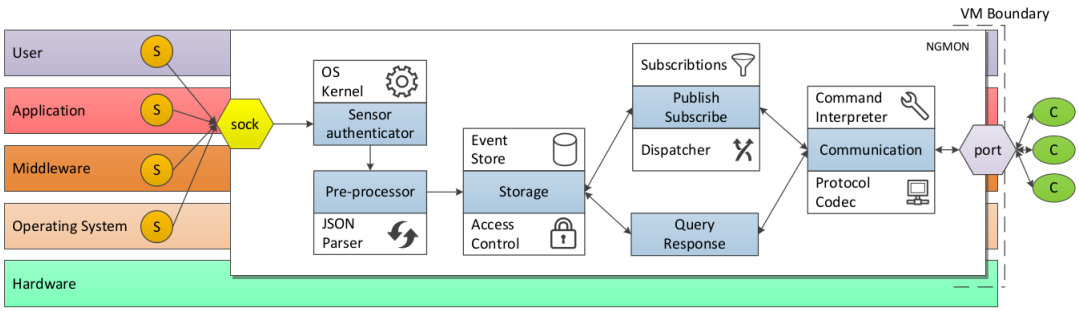

Figure 2.6: Data flow architecture of New Generation Monitoring.

By monitoring event we consider any notification, log or measurement. Notification is a message of various importance and usually needs further investigation. Typically, it is sent by email to administrator. Log can be any kind of message originating from arbitrary subsystem of monitoring entity and is usually sent into file or device as message in natural language[19]. Measurement is a quantitative information or metric of entity, but mostly it is used and computed by internal protocols. These three categories, event objects, are represented as typed data object with defined structure in JSON format with the same destination. All of them should be collected in a unified way and represented in the similar format, despite being a different in nature, but still considered as event objects. It is done this way, mainly because of minimization of an already highly complex and diverse events today. Data format of event object should be of following properties:

1. Standardized
2. Self-describing
3. Based on extensible schema
4. Structured
5. Compact

Most of these properties have already been mentioned in the CEE Section 2.3.1 describing standard, so it seems that NGMON is the right path. Structure of event objects is defined in one of two possible schemas. Either in predefined fixed or custom type schema, specific to particular event needs. Schema support part of Common Event Expression standard mentioned earlier, in particular the support of JSON format encoding. CEE Log Syntax can be helpful in providing five event data format properties listed in previous enumeration. Typical event generated into JSON by producer has structure similar to one depicted in Listing 2.6. NGMON's events can be transported also in XML or YAML format if needed, but prioritized is JSON data format. Monitoring events can be delivered in two means between sensor, the event producer, and the event consumer. Notifications and publish-subscribe[20] interactions are performed by producer via push model. Query based response initiated by consumer uses pull model. Communication is provided by custom SSL encrypted TCP protocol, where JSON event object is in a single body of frame. Synchronous and asynchronous transfer mode of data is supported for concurrent access of multiple tenants.

19. As stated before in Chapter 2.3
20. Publish-subscribe is a messaging pattern, where senders - publishers create a content and have no idea about who receives - subscribes for their messages. So subscriber receives only messages he is interested in - subscribed for.

```
1 private final static Logger LOG = Logger.getLogger(MyClass.
    class.getName());
2
3 if (LOG.isFatalEnabled()) {
4         LOG.fatal("Fatal error occured! " + error + " happend
              on host " + address.getHostAddress() + " at " +
              System.currentTimeMillis());
5 }
6 ...
7 if (LOG.isWarnEnabled()) {
8         LOG.warn("User " + user + " has exceedded his quota
              limit: " + limit + " by " + exceeding);
9 }
```

Listing 2.4: Example of log statement used in most logging frameworks today.

2.4.2 NGMON's logging system

What is significant about NGMON is its generation of event objects by custom log statement syntax. Typically as seen in Java Commons Logging, Java Logging API, Log4j, Slf4j, Logback or similar frameworks, logging statement is still in more-less traditional syntax, which was defined in Java Logging API in 2004. Since then no major breakthrough happened in the world of logging frameworks, despite massive cloud computation advancements, multitenant usage and processing of such grid data. NGMON tries to blaze out its own new way in this area, by creating its own logging system, called Java Event Logger. JEL should satisfy needs proposed earlier in Listing 2.4.1 and still maintain some degree of user friendliness.

NGMON's API is based on Logback project and performance of this logging framework should be comparable with frameworks using unstructured loggers. Log statements similar semantically or intentionally similar by design are grouped tightly together by using same namespace. Namespace could be reused across different projects and/or companies and save some amount of work for developers by using already defined statements.

One can compare traditional syntax depicted in Listing 2.4 with NGMON's depicted in Listing 2.5 using simple log definition and statements.

Log statements in Listing 2.5 on line 8 would generate the JSON log event depicted in Listing 2.6.

Developers have conducted two performance experiments to test their ideas. The first experiment was about testing performance of log statement execution,

```
Logger DAEMON_NS = Logger.getLogger(NS_Daemon.class,
                new JSONWriter());

if (LOG.isFatalEnabled()) {
        DAEMON_NS.error(error, address.getHostAddress(),
                System.currentTimeMillis()).tag(clazz).fatal();
}
...
if (LOG.isWarnEnabled()) {
        DAEMON_NS.user_exceed_quota(user, limit, exceeded)
                .tag(QuotaChecker.class).warn();
}
```

Listing 2.5: Example of log statement used in NGMON logging framework.

```
{'Event':{
    'id':16086,
    'occurrenceTime':'2014-01-15T12:05:27.567Z',
    'hostname':'machine2.mydomain.cz',
    'type':'org.myorg.myapp.quotachecker.user_exceeded_quota',
    'application':'My Application',
    'process':'myapp',
    'processId':27881,
    'severity':2,
    'http://myapp.myorg.org/v0.1/events.jsch':{
        'user': 'tester',
        'limit': 1024,
        'exceeded': 150
    }
}}
```

Listing 2.6: Example of NGMON generated structured log.

serialization and appending data for file or syslog between unstructured natural language statements and traditional statements using Logback opposing structured statements using NGMON's Java Event Logger. JEL proved to be on similar level as statements in natural language. Logback was faster by 30 - 40%, and expectably winning test. The second undertaken experiment tested performance of extracting and processing of obtained log event. Events were of two different kinds, represented as natural language and parsed by regular expressions, and structured events queried by JSON processor. A set of regular expressions was used to match various types of events, which proved to be significantly slower then simple JSON query. Even with unrealistic scenario, when regular expression was used as first item in set, querying was faster by 25%. Results and detailed information can be seen in published paper [14].

2.5 Summary of various logging systems

This whole chapter was a somewhat brief history and overview of generally most available and used logging frameworks for Java language that proved to work and satisfy need of given project at present time of development and usage. Ranging from Java Logging API through Log4j to Logback and structured logging standards and frameworks, spanning over 20 years or development and we are still unsure of whether it is the best possible way to use monitoring events like we did in past and do nowadays. Some people [15] are negative about progress towards structured logging frameworks and consider them to be most probably waste of time. On the other hand, they see various problems with present state, but do not offer any solutions. There are few technical blog posts about proposal and general best practices for structured logging from individuals and project groups like Splunk [21], Logstash[22] or Bunyan[23] for popular Javascript web platform Node.js.

With growing number of cloud computational usage, applications processing huge amounts of data, demands for clear, proper and not overly-verbose monitoring information have rapidly rose up. As we are unaware of new possibilities of processing these large amounts of data, leaning towards structured logging frameworks seems like a good way to at least do some progress in present state of logging frameworks which are getting rusty. If this would also initiate new ideas of how we perceive, could monitor and process data in more suitable way, it might be also a great achievement in our opinion.

When we want to test NGMON Logger properly with many applications and spread this logging framework, we need more applications to use NGMON. To achieve this goal, application's present logging statements have to be changed into suitable format. NGMON introduces new semantics and syntax for logging statements which we presented in Listing 2.5. This thesis shows four attempts of how to manually and later semi-automatically change Java application's default logging framework to NGMON's. They all will be presented in successive chapters.

21. Splunk products specializes in collecting, indexing, exploring, analyzing and visualizing the data. `http://dev.splunk.com/view/logging-best-practices/SP-CAAADP6`
22. Logstash is a tool for managing events and logs. After initial collection it parses, analyzes, stores and makes logs searchable `http://www.logstash.net/`
23. Bunyan project website `https://github.com/trentm/node-bunyan`

3 Overview of the tools, used for implementation

3.1 Used Tools

In this chapter we mention all tools we have worked with, in chronological order. When they were found as an interesting choice for us and were actually used, while trying to implement them in the best possible way of rewriting current logging framework in application to fit NGMON's syntax.

Starting with gathering all logs from Apache Hadoop project using Intellij IDEA's *search and export to file* function and manually rewriting the code, following with simple script in python by matching strings from the same IDEA's exported search text file and frivolous search and replace in source code as pure text file, formatted in Java language, to some more advanced approaches like tinkering with Java Compiler API and changing internal objects of compiler. Final and probably the most elegant solution is done using ANTLR framework for rewriting of tokens in syntax trees generated by ANTLR[1], when parsing the java input files.

3.1.1 Git

Git is a very popular and powerful open source version control system used in many projects, particularly in many open source projects, as it was also originally designed for these kind of work. Gaining on popularity due to its easiness of use, fastness, decentralization and rich functionality [16]. We also chose to use git and as a project home for NGMON Logger, we have selected GitHub, a free project hosting repository website. LogTranslator git repository can be found on url[2] containing all our developed source codes.

3.1.2 Sublime Text

Multipurpose sophisticated text editor, which functionality is extendable by vast number of plugins, varying from different needs of contributors, makes Sublime Text[3] a very efficient, yet still easy to use text editor. As there was no need for specialized Integrated Development Environment for our Python prototyping, we used sublime text editor in conjunction with Python plugins for our first and naïve log statement rewriting approach.

---

1. ANother Tool for Language Recognition `http://www.antlr.org/`
2. LogTranslator git repository `https://github.com/michalxo/LogTranslator`
3. Official Sublime Text website `http://www.sublimetext.com`

3.1.3 IntelliJ IDEA

IntelliJ's Integrated Development Environment IDEA[4] is designed for programming in Java, specially for projects profiting from Java Enterprise Edition standards usage. IDEA supports many enterprise frameworks like Spring, Enterprise Java Beans, Google Web Toolkit, Struts, Google App Engine and offers many other advantages for fast development of various applications.

At first we used its structural *search and export to text file* function as input for prototyping application using *simple and naive* 4.1 python script for changing logs. Later it was used as should be with Java Compiler API in hand with Java Pluggable Annotation Processor and finally with ANTLR as final decision for building log rewriting tool - LogTranslator and testing purposes.

This IDE is considered to be one of the best for Java development, but on the other hand, it is not free of charge[5]. For our project, we have been using *Ultimate* version because of Apache Maven and easy integration with bigger, multi-moduled project - Apache Hadoop, which is written in compliance with Java Enterprise Edition standards.

3.1.4 Apache Maven

Maven[6] is what a project engineers would call *Project Management Tool* or by majority of its users, *build tool* [18] to build deployable artifacts from source code. Maven was originally designed to simplify building process of Jakarta Turbine project from several projects with different Ant[7] build files. Apache took over Maven and developed it in to what we use today - build multiple projects together and publish or share project information for next reuse [17]. Maven is capable of doing many things that Ant can not do in a convenient and straightforward way or is simply incapable, because it is more of build-only tool. We can safely consider Maven to be a platform, rather then a tool, because it does not only simplify build management, but also encourages a common interface between developers and software projects.

Capability of many different tasks over source code and project itself is another domain for Maven. Tasks are ranging from basic build tools like preprocessing, compiling, testing, packaging, installing through deploying and cleaning to running various reports, source code and release maintenance or generating websites.

More formally, Maven is a project management tool, which encloses a Project

4. IntelliJ IDEA's website address `http://www.jetbrains.com/idea`
5. There is a possibility for free of charge usage of Ultimate edition (fully supports Java EE), if project is open source and IntelliJ's developers approve your request.
6. Official Apache Maven project's website `http://maven.apache.org/`
7. Apache Ant (*Another Neat Tool*) is similar to *make* build tool, but it is mostly suited to Java language and platform. As a main building file serves `build.xml`.

Object Model[8], lifecycle of project, dependency management system and execution of various actions via plugins during specific lifecycle phases of project. Project containing multiple artifacts, each having its own *pom module* is a standard situation for Maven, where it applies its cross-cutting logic from a set of shared plugins.

Maven can easily grow from tiny build and distribute tool to monstrous size, handling every operation needed by versatile demands of huge project. To preserve its functionality for any platform of choice, there is a simple concept hardwired with Maven - *convention over configuration*. That means, developers are not required to create build process themselves or set up every configuration detail. Also application can safely assume some reasonable system default settings, so everything should work straight away, without the need of fiddling things out[9]. As a result of following this concept, using framework defaults, it is much faster to setup and execute project.

Following list displays some of directory default values, where `$basedir` variable points to projects main directory.

${basedir}/src/main/java - source code of all Java classes (except tests)

${basedir}/src/main/resources - various configuration and property files, sometimes in `META-INF` folder like `persistence.xml`, `beans.xml`, `applicationContext.xml`, `log4j-conf.xml`

${basedir}/src/main/webapp - designed for web application archive content - for example it contains folder `WEB-INF` with configuration `web.xml` and various property files,

${basedir}/src/test - contains source code of all Java test classes

${basedir}/target - stores compiled classes and built deployable packages

Maven is able to perform various tasks during specific lifecycle phase of project. Just for illustration, there are in total thirty phases[10] depending on type of packaging, during which you can interact with project through defined goals and native or custom-made plugins. Thanks to all of mentioned characteristics, there is no doubt, that Maven has naturally evolved into a non-written standard in Java SE and EE for production and academic use.

---

8. POM - a well-formed `XML` file `pom.xml` for given project or artifact, which describes structure, dependencies as well as what and when should be done by Maven plugins with common project(s).
9. Albeit you can always change location of source code and target as well as can be overridden almost any specific behavior.
10. Apache Maven 3.0.4 has following phases: validate, initialize, generate-sources, process-sources, generate-resources, process-resources, compile, process-classes, generate-test-sources, process-test-sources, generate-test-resources, process-test-resources, test-compile, process-test-classes, test, prepare-package, package, pre-integration-test, integration-test, post-integration-test, verify, install, deploy, pre-site, site, post-site, site-deploy, pre-clean, clean, post-clean.

LogTranslator project uses Maven as a project management tool and works with Maven projects. Generation of source code is performed into one of main directories - specifically into *${basedir}/src/main/java/* directory.

3.2 Language processing tool - ANTLR

ANTLR (ANother Tool for Language Recognition)[11] is a powerful parser generator for reading, processing, executing or translating of structured text or binary files. It is widely used in production and academical environment for various language specific operations. *Twitter's* search engine, Hadoop related projects (languages) *Hive* and *Pig*, Oracle *SQL Developer IDE*, *Netbeans IDE's C++ parser* depend on ANTLR, also *Hibernate's HQL language* for object-relation mapping is written in ANTLR.[22]

Those were just a few of the companies that highly depend on this successful open-source project, which has recently released version 4. Apart from mentioned usages, one can create his own property file reader or JSON parser, regular expression matcher, code converter (also translator) or some any other language case-specific tool. In our case, we used translator feature of ANTLR, which generates parse tree or abstract syntax tree, walkers, listeners and translators based on visitor pattern (see Section 4.4.4).

It is worth mentioning, that there is only Java implementation of ANTLR[12], but one can create tool for any language or purpose he needs to.

3.3 Chosen application - Apache Hadoop

As an example application for our all testing purposes with swapped logging system from default's to ours and ideal for cloud oriented logging system was chosen Apache Hadoop[13] project. Lately this project has become very popular and its production usage is booming because of its main properties designed specially for cloud oriented problems. Hadoop is chosen often because of its massive scalability, high reliability, availability and as fault-tolerant distributed oriented system for huge data storage and processing on big data sets spread across multiple clusters ranging from single to thousands of machines.
[21]

Apache Hadoop project consists of four main packages (projects):

11. Official ANTLR website `http://www.antlr.org`
12. Officially supported by author Terrence Parr. He does not plan in near future to invest his time to reimplement ANTLR to different programming language.
13. Apache Hadoop website address `http://hadoop.apache.org`

Hadoop Common - foundation package provides filesystem, operating system abstraction layer and tools needed for cooperation of different projects,

Hadoop Distributed Filesystem (HDFS) - a distributed file system designed for running on commodity hardware. It is highly fault-tolerant and is designed to be used on low-cost machines. HDFS also provides high throughput and besides that, it is still similar to any other existing distributed file systems.

Hadoop MapReduce - implementation of a programming model for large scale data processing based on Google's ideas.

Hadoop YARN - a resource-management platform, responsible for managing compute resources in clusters and using them for scheduling of users' applications.

other - many Hadoop related projects like Ambari, Avro, Cassandra, HBase, Hive, Pig, Zookeeper provide auxiliary support[14].

Hadoop project has been chosen as LogTranslator's example prototyping application for various log usage collecting, because it is a big project consisting of approximately 500 classes with log usages, has multiple contributors and uses Maven building system. As we have found out, Hadoop uses three different logging frameworks, Log4j, Slf4j and Apache Commons Logging. Multiple contributors mean a lot of coding styles (preferences), so we could see much of various code usage in original log methods, which we could handle. On the other hand, the building process of Hadoop module is daunting and can easily fail.

14. We have not used any of those projects for log translation purposes.

4 Implementation of changing logging system

When we first wanted to fill up NGMON logging framework with testing data, there was no "real" application using it. We were going to take an existing well-known and widely used application and turn its logging framework into NGMON's instead of actual one. It might be either Java logging util, Slf4j, Log4j, Apache Commons Logging, Log4j2 or any other non-NGMON logging framework. To successfully achieve this goal, we had to change the following features in the given application:

- change related log imports to NGMON's and add new ones,
- change log definitions to NGMON's,
- change checker methods like *LOG.isDebugEnabled()* into custom "dummy" logger always returning true,
- turn original log statements into new NGMON's syntax with method names as event names,
- parse all variables and their appropriate types if possible[1],
- generate appropriate log namespaces which would contain relevant events

There were many challenges, but one stood out, which was mentioned above. To parse all variables reference from within and out of given class, and use them to create new log methods in NGMON syntax. This syntax is very different from what we have seen so far in logging frameworks. To make our point clear, please see Listings 2.4 and 2.5. This means, that we have to change for example following log statement in natural language

```
LOG.debug("Update nonSequentialWriteInMemory by " + count +
                " new value:"+nonSequentialWriteInMemory);
```

into the following statement, done in a more structured way of NGMON syntax

```
LOG.update_nonsequentialwriteinmemory_by_new_value(count,
          nonSequentialWriteInMemory).tag("write-in").debug();
```

1. Although, we are able to get almost all of variable data types, NGMON uses only types supporting JSON format - Number, String, Boolean, Array, Object, null. In our LogTranslator application we operate with Java primitive types and String by explicit type-casting to them.

We have taken three different steps which lead us to fourth, final and probably the best solution so far. Those four steps were:

1. Manual rewriting - completely manual rewriting and creating of all log declarations, statements, imports, variables. Described in detail in Chapter 4.1
2. Python prototyped script - simple script, which helped us to understand what is necessary to do in order to translate application's logging framework. However, it was unmanageable as the "main project" in future. More information is in Chapter 4.2.
3. Pluggable Annotation Processor - use of Java Compiler and Pluggable Annotation Processor to trigger custom code during compile time and modify compiler's internal symbols table and rewrite source code during actual compilation. Details are in Chapter 4.3.
4. ANTLR project - by using language recognition tool, we do not have to deal with compiler internals, on contrary it is just another application which does all the hard for us, so we can focus on our problem. Further reading is in Chapter 4.4.

It is worth to note, that we can not use any kind of Java *Reflection Utils*, because we are not translating code in runtime, which would normally be in Java Virtual Machine, thus we have to translate code with other tools and our input is only text file, which *accidentally* has *.java* suffix. In the following sections we talk a bit more about all of these steps we had taken.

4.1 Naïve approach using structured text

The very first contact with what had to be done was during this first phase. We have extracted all the *LOG.<method>()* calls from IDEA and stored them into text file. It is important to mention, that we were interested in all log calls, except those, used in tests. Therefore we would count and change only useful and executive log calls from application.

```
~/hadoop-common/src/main/java/org/apache/hadoop/http  (13 usages found)
   HttpServer.java  (13 usages found)
       HttpServer(String, String, int, boolean, Configuration, AccessControlList, Connector, ...)  (1
            usage found)
           (281: 9) LOG.info("adding path spec: " + path);
       addJerseyResourcePackage(String, String)  (1 usage found)
           (430: 5) LOG.info("addJerseyResourcePackage: packageName=" + packageName
       addFilter(String, String, Map<String, String>)  (2 usages found)
           (504: 5) LOG.info("Added filter " + name + " (class=" + classname
       addGlobalFilter(String, String, Map<String, String>)  (1 usage found)
           (526: 5) LOG.info("Added global filter '" + name + "' (class=" + classname + ")");
       start()  (3 usages found)
           (684: 9) LOG.info("Jetty bound to port " + listener.getLocalPort());
           (687: 9) LOG.info("HttpServer.start() threw a non Bind IOException", ex);
           (690: 9) LOG.info("HttpServer.start() threw a MultiException", ex);
```

```
stop()  (4 usages found)
    (774: 7) LOG.error("Error while stopping listener for webapp"
    (784: 7) LOG.error("Error while destroying the SSLFactory"
    (794: 7) LOG.error("Error while stopping web app context for webapp "
    (801: 7) LOG.error("Error while stopping web server for webapp "
```

Listing 4.1: Excerpt of IDEA's search for *LOG.<method>()* usage output showing logs used in HttpServer.java class of Apache Hadoop project.

In our case Apache Hadoop had 4718 calls on log object in 801 files, which met these conditions. Example of IDEA's output can be seen in Listing 4.1 That is quite a big number to process them manually. Just to see how long it would take, we have changed a few dozens of log methods and got an average time of 1 minute and 27 seconds per log change. For simpler logs, changing syntax took around 47 seconds, and for more complex syntaxes, this time interval was longer, coming up to 2 minutes. In addition to that, we counted 10 seconds to create new method into appropriate NGMON's namespace. Also, we had to change imports, log definition and "log checker" methods.

Assuming that each one of 801 files has its own log definition and imports, changing Hadoop's logging framework into NGMON, would have taken around 466325 seconds which is roughly $129,5$ hours. This number was calculated using the formula in Listing 4.2 and values were from Table 4.1.

Change or create	abbrev.	Time needed [seconds]
Appropriate imports	imp	10
Log definition	def	10
Log checker method	chec	5
Log method	log	87
Create method in NGMON's namespace	ns	10

Table 4.1: Measured time in seconds needed to manually transform single appropriate logging statements to NGMON's needs.

Let's assume that following statements are true. Every log checker[2] method has in general half the count of number of logs. In the best scenario, every log should have appropriate checker method, but generally, in real projects this is not true at all. Also some logs can be the same as others, but can have different variables with the same data type, so we can ignore another generation of the same namespace log method. Assume that this number is between $5 - 15\%$ and very project specific, so let's stick with 10% as an average value.

2. By log checker methods we consider for example *LOG.isInfoEnabled()* method call.

$$
\begin{aligned}
chec\_logs &= all\_logs \div 2 \\
total\_time &= number\_of\_files \cdot (imp + def) + \\
&\quad chec\_logs \cdot chec + \\
&\quad all\_logs \cdot log + all\_logs \cdot 0.9 \cdot ns
\end{aligned}
$$

Listing 4.2: Rough estimation formula for conversion to NGMON's project

According to this estimation, one has to work for 130 hours, what is 17 working days, just to rewrite whole Apache Hadoop manually. And this is only rough estimation, which does not include human errors, checking for already "generated" methods in namespace. Above all, one has to be fully focused on task for whole time not to make mistakes, which is obviously physically impossible. Imagine we would find out in the middle of translation process, that we could not cope with Hadoop and were forced to switch to other application. This whole manual work seemed like a naïve idea, so we quickly moved away from this solution and moved to more automatized solution.

4.2 Python prototyping application

As a first semi-automatized solution we came up with a short script in Python, which would do the work we expect to transform arbitrary Java logging framework to NGMON's.

Script starts by looking into IDEA generated file, which has exactly the same structure as in previous solution in Listing 4.1, and parses one log method after another. Each of these log methods are distinguished by path to Java file containing it and even by exact position (line : column). One little problem was, that log call were not complete, missing another lines of code, when log method has been spread over multiple lines, so whole log method had to be "reparsed" again. Extracted variables and content of log call could be easily used to generate new log method for NGMON's namespace and replace given log in the original Java source code file. Very soon we hit a barrier with two major problems. First problem was inability to easily get types of variables and use them properly, as well as too complicated statements used in logs thanks to rich properties of Java language. Second was from long term standpoint - harder code maintenance and complicated string operations.

Soon we have started to consider more advanced approaches to think about source code and its parsing. We got into simple programs which can create *Syntax Trees* from source code files. Namely, it was through Eclipse Java development tools (JDT), which provided API to access Java source code via abstract syntax tree[3] or

3. Eclipse's introduction to Abstract Syntax Tree and code manipulation - `http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`

Java Model. While half of the work had been done, we tried some simple examples of how to connect Java and Python programs together and even call methods from one language and work in other[4]. Results looked really interestingly, but they put us on the wrong line. We were keen to go back and not to try to reinvent the wheel, we left this half-finished approaches and moved into some more serious solutions which were proved to work.

While it might seem that this approach gave us almost nothing new, it was entirely true. We have found out, that fiddling with Abstract or any other Syntax trees are the correct way to rewrite source code without doing all the manual work. Latest source code of log translation python script can be found on github[5], in one of the commits in history, as we have completely devoted repository to Java LogTranslator application only.

4.3 Pluggable annotation processor and Java Compiler API

Coming to the more advanced solutions, we have decided to attempt a combined use of compiler and annotation processor, which will be plugged into compilation process and forced to be used during annotation processor phase, before compilation of source codes themselves. First, let's have a look at those two interesting tools which are present in Java 1.6 by default.

4.3.1 Pluggable Annotation Processing API - JSR 269

In Java version 1.6 there has been added functionality for standardized way of compiling Java source code files from Java program itself. This support came from standard JSR-199[6]. Implementation is in *javax.tools* package, specifically we speak about *JavaCompiler* interface.

On top of this Compiler standard is seamlessly integrated Pluggable Annotation Processing API, specified in JSR-269[7]. These two specifications, Java compiler and extending annotation processor, allow us to interact with source code by changing internal structure of java compiler. Implementation of annotations processor is also in *javax* package, with api defined in *javax.mirror* namespace. This JSR allows us to write our own annotation processors, which could be plugged into compilation

4. Py4j project is capable of calling Java objects from python code. More information can be gained from `http://py4j.sourceforge.net/`
5. Python LogTranslator link pointing to the specific commit in history - `https://github.com/michalxo/LogTranslator/blob/12509d5315a3487fa06972057f11fdfd41a55b80/Log-change/logchanger.py`.
6. JSR-199 specification `https://jcp.org/en/jsr/detail?id=199`
7. JSR-269 specification `https://www.jcp.org/en/jsr/detail?id=269`

process. We would call annotation processor by simply passing *-processor MyAnnotationProcessor* argument to javac command. Such custom annotation processor appears to be highly effective, when working with the state of compiler and is capable of even rewriting syntax trees, which is what we currently need. In the next section we would like to give a short presentation of compilation process in Java, which is an important part of this rewriting method.

4.3.2 A short introduction to Java compilation process

In this section, we would like to present a general concept of compilation of Java source code files as described in OpenJDK documentation [19]. The compilation process is controlled by *JavaCompiler* class and consists of three main parts. Process is also depicted on Figure 4.1.

1. Parse and Enter - Source files are read and parsed into syntax trees. All externally visible definitions are entered into the compiler's symbol tables.
2. Process Annotations - All appropriate annotation processors are called. If any annotation processor generate a new source or class file, the compilation is restarted, until no new files are created.
3. Analyze and Generate - Syntax trees created by the parser are analyzed and translated into class files. During the course of the analysis, references to additional classes may be found. The compiler checks the source and class path for these classes; if they are found on the source path, those files will be compiled as well, but without annotation processing.

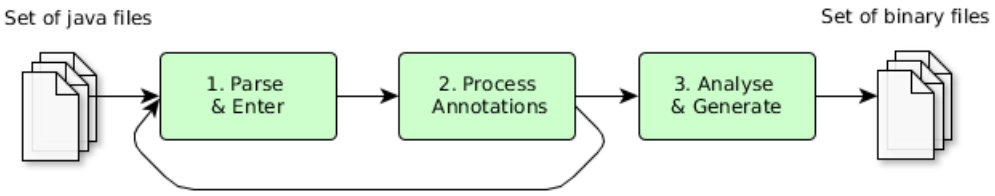

Figure 4.1: Java compilation process

During *Parse and Enter* phase, source files are read by Scanner and converted into a stream of tokens. Parser reads these tokens and creates appropriate abstract syntax trees using TreeMaker from subtypes of JCTree class as tree nodes. Each such tree is visited and all encountered symbols are entered into symbol's table. If there are some unknown symbol references, appropriate trees has to be analyzed.

JavacProcessingEnvironment is taking control from compilation process and accordingly updates JavaCompiler object if needed, during *Annotation processing* phase. JavacProcessingEnvironment enters, parses and invokes relevant annotation processors for given classes until all source files are completed.

At the end of this phase, the (un)modified JavaCompiler object is returned to compilation process and now actual *Analyze and Generate* phase begins. By analyzing parsed syntax trees, class files are generated. Many references can be found to other needed classes during analyzing process, which should be defined on class path or source path. This whole tree analyzing and generation process is using series of visitors in Section 4.4.4. Once a class is successfully written out as a class file, references to symbols and trees are nulled out, to allow garbage collector to clean up used memory.

4.3.3 Outcome of experimenting with Pluggable Annotation Processor

We were experimenting with this technique and found literature on *How to abuse JSR269 for AST rewriting* [20], which looked suitable at first sight for rewriting and generating source code files. We would create our own annotation processor, plug it into compilation process, which would run before third phase of compilation phase and rewrite all the log statements, definitions, imports, variables and prepare all the needed classes and methods to generate for us.

Everything seemed quite reasonably, but on the other hand, we did not want to dive too deep into compiler internals, manage and overwrite symbols table or use Java compiler specific code, which could be changed in the new version of Java. That would lead to errors and eventually cause bigger problems in future and unusability of application. Hence this method was a dead-end for us, and we had to go one step back to abstract syntax trees. It might not seem like a greatly invested time, but we have learnt a lot and could leave this experiment as well. We decided to try and have a look on a different solution then what *abusing of annotation processors to rewriting abstract syntax trees* offered.

4.4 Approach using ANTLR language tool

Our latest approach was to use ANTLR - *ANother Tool for Language Recognition*, which we have thought of at the very beginning, but were considering it as a big, overpowered project which would be hard to handle in every possible aspect. After some time, the opposite turned out to be true. What ANTLR offered us was all we needed and still had a lot more capabilities which we have not even came across. In the following sections, we will explain what is ANTLR, how it works and how we used it in our *LogTranslator* project.

4.4.1 How ANTLR works

To be able to work with ANTLR, we have to supply it with two inputs. First is a grammar file, which defines the language we want ANTLR to speak. Second is the file, we would like ANTLR to traverse and process. In best case, it really should be in the same language as supplied in grammar, so ANTLR can understand what it reads.

Formal rules of language are defined in a file with suffix *'.g4'*, known as grammar. It is primary input for ANTLR Java main program, depicted in Figure 4.2. Lexer, tokens and parser are then generated from these given grammar rules, so patterns could be matched by tokens and *parse tree* data structures built by parser.

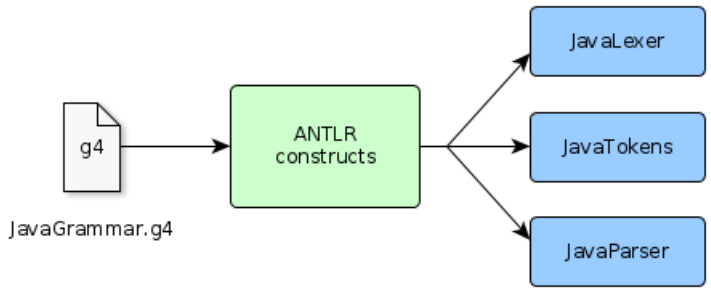

Figure 4.2: First run of ANTLR. Generation of lexer, tokens and parser from provided grammar file input are performed by ANTLR.

Parsing, formally known as syntax analysis, is a two phase action. During first phase, lexer through process called lexical analysis or *"tokenizing"* groups characters from input, into chunk of meaningful words or symbols known as *tokens*. Lexer also groups related tokens to token types like identifier, float, string, expression or any other type, while parser is interested only in token type.

Each token consists of at least two parts:

- token type - token classes from grammar ID, INT, STRING, EXPRESSION, Method, DeclarationField or other type,
- matched text - actual value of token depending on given token type,
- other - properties about matched text like line start, end, column start, end, children and parent tokens, ...

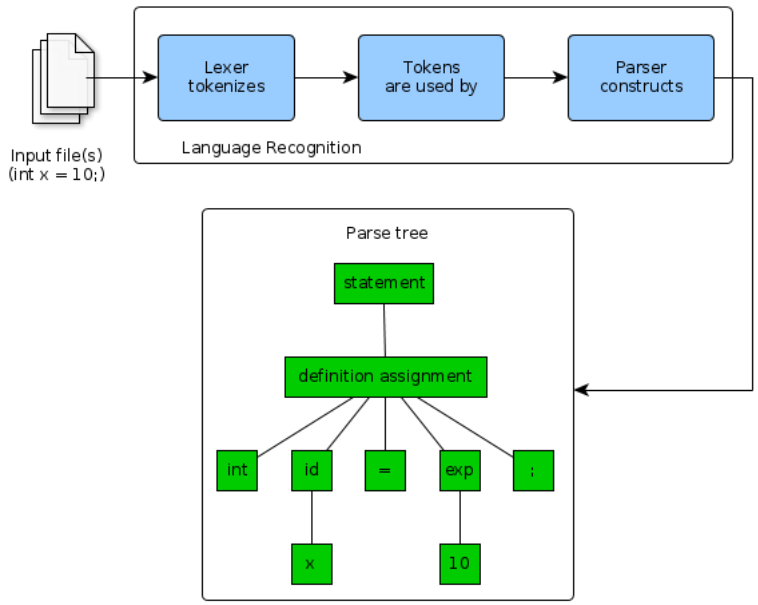

Figure 4.3: Second phase of ANTLR's run. Language recognizing process is using generated lexer, tokens and parser to create parse tree from provided input file(s).

In the second phase of syntactical analysis, parser takes tokens from lexer and recognizes the structure of sentence. As a result of recognition is built a data structure which represents this sentence - parse tree, (syntax tree) in ANTLR by default.

In the Figure 4.3 you can see an example of how ANTLR recognizes input file, based on provided grammar from Figure 4.2 and use of generated lexer and parser. Simple input as *int x = 10;* will force ANTLR to create a depicted green parse tree with appropriate grammar rules.

What interests us the most is ANTLR's ability to modify information in each node, while traversing it either using listener or visitor pattern. We will speak more about those patterns in following sections as well as about parse trees and examples made by ANTLR's visualization tool - *grun*.

4.4.2 Parse tree

A parse tree[8] describes how a parser recognized an input sentence. They record the sequence of rules a parser applied as well as the tokens it matches. They are used

8. Parse trees are sometimes called Syntax trees.

mainly in text rewriting scenarios, syntax highlighting and error-checking [23].

Generated parse tree is very tightly connected to appropriate input grammar as all tokens originate their name from actual rule name in grammar. The root node is named after grammar's starting point. Child nodes of root have their name after possible root's child elements and so on. In other words, in each subtree, the root node is named after specific grammar rule [22].

Parse trees are easy to build and thanks to their regularity, ANTLR can automatically build them. On the other hand, parse trees are really hard to walk and use for transformation. They contain a lot of noise because in their nodes are matched rules found by parser. Parse trees are also very sensitive to changes in the grammar.

To get more clear understanding of parse trees, let's have look at following example. In Listing 4.3 you can see a snippet of Java grammar[9] of root node.

```
1  // Starting point for parsing a Java language file
2  compilationUnit
3      :  packageDeclaration? importDeclaration*
4      |   typeDeclaration* EOF
5      ;
6
7  packageDeclaration
8      :   annotation* 'package' qualifiedName ';'
9      ;
10
11 importDeclaration
12     :   'import' 'static'? qualifiedName ('.' '*')? ';'
13     ;
14
15 typeDeclaration
16     :   classOrInterfaceModifier* classDeclaration
17     |   classOrInterfaceModifier* enumDeclaration
18     |   classOrInterfaceModifier* interfaceDeclaration
19     |   classOrInterfaceModifier* annotationTypeDeclaration
20     |   ';'
21     ;
```

Listing 4.3: ANTLR's snippet of Java grammar.

Following code in Listing 4.4 depicts an example of simple Java source code and

9. Grammatical rules for Java 1.7, were obtained from `https://github.com/antlr/grammars-v4/tree/master/java`

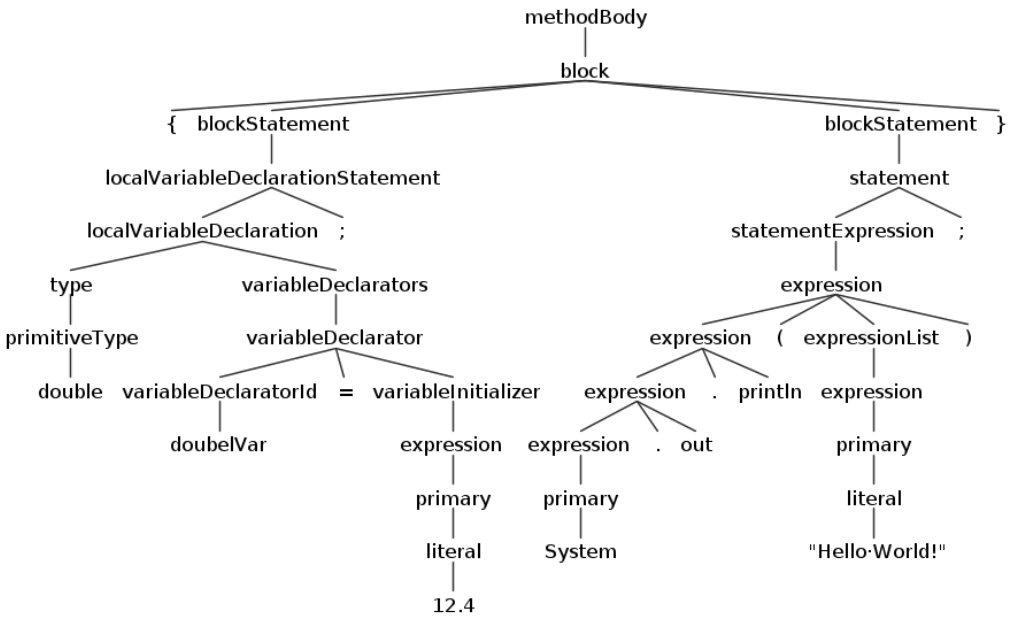

Figure 4.4: Parse Tree of shown Java code in Listing 4.4, specifically lines 5 and 6.

its appropriate *method body* rule part (lines number 5 to 7) of built parse tree in Figure 4.4 by ANTLR. You can see whole parse tree on Figure A.2 in appendix. A typical log statement is in Appendix A.1 and a nice example of leftmost derivation left-to-right grammar parser is in Appendix A.3.

```
1 public class App {
2         public static void main(String[] args) {
3             double doubleVar = 12.4;
4             System.out.println("Hello World!");
5         }
6 }
```

Listing 4.4: Java source code Example.

4.4.3 Abstract syntax tree

Because of too much information in parse tree nodes and sensitivity to grammar change, internally ANTLR uses Abstract syntax trees[10] as intermediate representa-

10. Also sometimes called as syntax tree or parse tree.

tion. AST is a tree representation, that captures only the most important information from input - all input tokens and the appropriate input structure. Interior nodes are either operators or operations, instead of rule names.

In order to be proper *intermediate representation*, used by language tools, Abstract Syntax Tree should have following properties:

- Density - they should not have unnecessary nodes,
- Convenience - should be easy to walk,
- Meaningfulness - put stress on operators, operands, and the relationship between them rather than artifacts from the grammar [23].

Simple example of AST can be seen in Figure 4.5, depicting simple statement $num = 3 + 7 \cdot y$. As you can see, they do not contain any rule names from grammar. Not even parenthesis, semicolons or other syntactic sugar, which are redundant for tree processing. Only operators and operations. Also, they are very easy to walk and process, due to the mentioned properties and being a linked list structure.

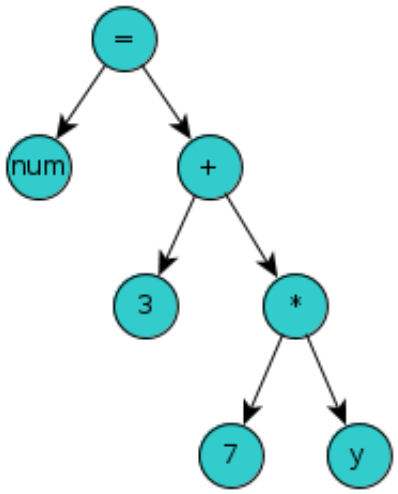

Figure 4.5: Simple example of Abstract Syntax Tree with following input $num = 3 + 7 \cdot y$.

4.4.4 Walking the tree

When we speak about walking or traversing a tree, we think about visiting a node in tree and performing an operation on it. Order of visiting nodes is of course important as it affects operations and final result. We could write our own depth first

search algorithm, visit all nodes and perform in each node an action, what we have to do. Or we could reuse one of the already implemented tree-walking mechanisms offered by ANTLR. We could choose to use visitor or listener.

By default, ANTLR creates listener tree walker. Listener interface responds to events triggered by the built-in tree walker. Such events are entering and exiting specific rule node in syntax tree by ANTLR's walker class *ParseTreeWalker*. For example, events are listener's implementation of methods like *enterEveryRule()*, *exitCompilationUnit()*, *enterPackageDeclaration()*, *exitFieldDeclaration()* and so on. To make use of events, we have to extend ANTLR's generated listener for given grammar, which is a subclass of *ParseTreeListener* class, by our own listener and override specific methods (events) which we are interesting in.

In Figure 4.6 you can see how ANTLR's listener walks the tree. Direction is counterclockwise, because of *Adaptive LL(*)*[11] parser, depicted by blue arrow in figure. By yellow lightnings we have tried to show where *ParseTreeWalker* initiates appropriate rulenode enter or exit event.

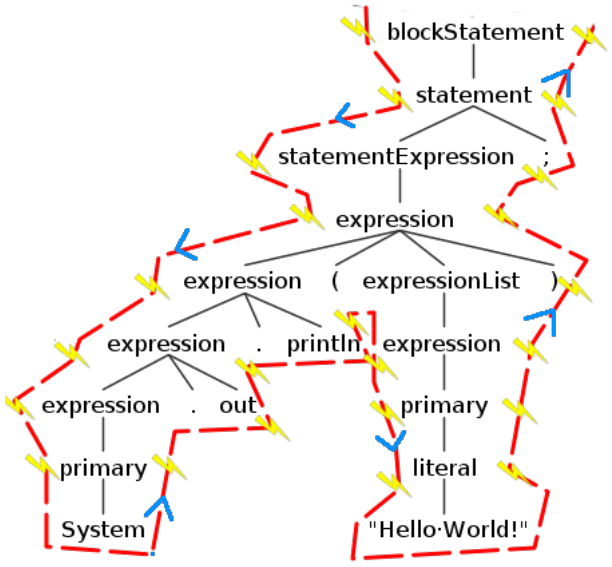

Figure 4.6: Depiction of walking tree by listener.

When using the listener pattern mechanism, we do not have to write our own tree walker. On the other hand, we are unable to control the order of visited nodes.

11. Adaptive Left to right, Leftmost derivation, using non restricted, infinite tokens for lookahead. More information about ALL(*) grammar parser can be found on `http://www.antlr.org/papers/allstar-techreport.pdf`.

In this case, we should use visitor pattern to walk the tree. We ask ANTLR to generate visitor interface from grammar by passing *-visitor* argument to first run of ANTLR, when creates appropriate lexer, parser and tokens. Generated visitor creates one *visit* method per rule. One can explicitly say which node you want to visit by overriding appropriate visit method. As a nice addition, one does not have to wait until walker visits all nodes. They could be safely skipped over.[22]

Now when we are aware of the foundations of the work with ANTLR language recognizing tool, we are finally prepared to create LogTranslator project.

4.4.5 LogTranslator project

With experiences gained from previous not so successful attempts, we have moved to implementation of a translation program in Java version 1.7 using ANTLR's linguistic abilities of version 4 and in compliance with Maven project specifications. Our *example* application full of logs to transform was Apache Hadoop. As a side effect, during implementation of LogTranslator we have noticed, that Hadoop uses not one, but three separate logging systems. At the time of writing this thesis, we support log4j, log4j2, slf4j and apache commons logging frameworks. Java utils logging should be next to implement, and user can easily add another logging framework when needed. LogTranslator works only with Maven project, but it should not be so hard to add capability to translate non-maven projects as well. In Figure 4.7 you can see what LogTranslator does with given project directory input, containing at least one logging statement.

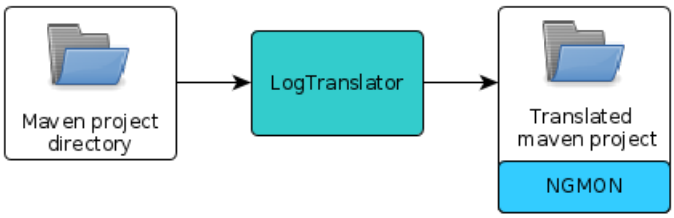

Figure 4.7: A "big picture" of what LogTranslator does with given Maven project input directory.

As we have mentioned earlier in the first list in Chapter 4 of what is needed to be done for application to work with NGMON's logging framework, we have accordingly broke down project into separate steps. That means, LogTranslator is divided into eight consecutive logical steps. If any of the first four steps fails, whole application fails. We could think of the project as a bigger script written in Java, which is easy

to manage, build and update in future, due to capabilities of Maven.

1. Initialize user settings and initiate search for Java log files in given project folder, which contain at least one non-test log method call.
2. Generate namespaces from gathered Java source code files containing log calls.
3. Parse each file using ANTLR; gather all referenced variables, change imports, log definition and log checker methods and log statements calls.
4. Generate new NGMON's methods and replacements for original log method calls from available information. Also generate GoMatch[12] patterns from log methods.
5. Prepare NGMON's namespaces and fill them with newly generated log methods using StringTemplate tool.
6. Write all rewritten Java source code files and prepared NGMON's namespace files on hard drive.
7. Create dummy checker class LogGlobal and default NGMON's SimpleLogger class, using Log4j2. Put them into target's Maven project directory with generated namespaces as new Maven module.
8. Show information about changed logs, modified and generated source codes paths.

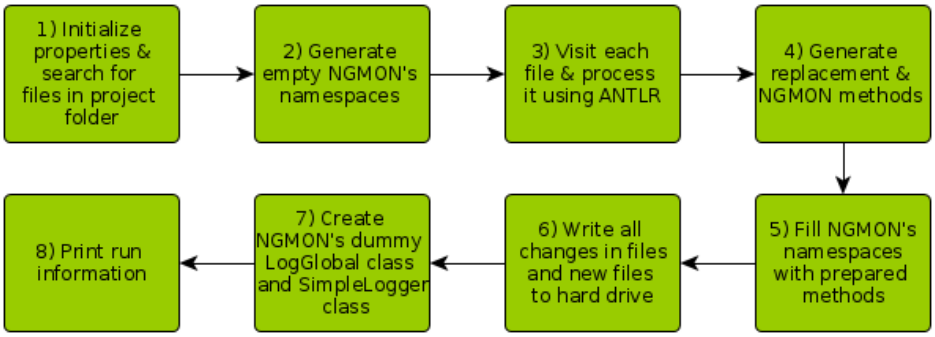

Figure 4.8: Diagram of LogTranslator process divided into consecutive steps.

LogTranslator (LT) is started and all phases are directed by *TranslatorStarter* class. At the beginning of run, *TranslatorStarter* initializes user settings from *log-*

12. GoMatch is a pattern matching project, which focuses on matching log events generated from various applications. More information can be found on `https://github.com/halafi/gomatch/wiki`

translator.property file. These settings hold information about location of project directory to be translated, how NGMON should generate new log methods, namespaces, various imports and log definitions, where to put generated namespaces[13], and cover possible future changes in NGMON itself.

After initialization, search for Java files containing at least one *log* occurrence in Java file in a given project home directory is started. All test related files are skipped. We are interested in application logs only. From gathered files from finished search, a set of empty NGMON namespaces is generated. They will serve for filling in with newly generated methods from found log events (methods) during following phase.

Each found file is then passed to ANTLR for language processing. This part is considered to be the heart of whole LT. We are looking for various LogFactory and Log imports and change them and append new NGMON one's. From now, we are aware of which logging framework has been used in processing file class. We also modify log definition and change all log checker methods to be called on *LogGlobal* object instead of original object. In case when there is no log definition, nor declaration in this class, we simply fall back to use of *FailsafeLogger*, which acts like any logging framework. When we finally come across logging statement, we parse all strings in it, from which will be generated NGMON's log event (method) name, its level and all numbers, variables, methods, new Java statements or any used objects in it. From this information we can prepare new NGMON log event and create replacement log. During that, we also store all possible information about all variables referenced in this class. Focusing mainly on getting its data type and position, and current line number in the file.

We should note, that LT does not take scopes into account, which might be a little problem in the future when NGMON will become more evolved. But as NGMON uses only primitive data types and string, we can safely typecast any object to string data type, so at the end, we will not lose any information whatsoever. We also take into account method calls and do a bit of a backtracking to find out its return type and initiate another run of ANTLR for this or another file parsing. This phase also rewrites original log method call by new - NGMON's replacement log event. Class which does all of this is named as *LogTranslator* to underline its importance.

This *LogTranslator* class copes well with log methods containing statements like creation of new object, throwing new exceptions, mathematical expressions, calling methods from this or other class, typecasting, simple boolean expressions

13. By default generated namespace files are placed into *${project_home}/src/main/java/log_events* directory.

and partially with ternary operators.

When every found file containing *log* statement is successfully parsed and modified by ANTLR, all generated NGMON log events which belong to this namespace are written into prepared namespaces class. In Listing 4.5 you can see an example of a prepared namespace.

```
package log_events.org.apache.hadoop;

import org.ngmon.logger.core.AbstractNamespace;

public class HdfsNamespace extends AbstractNamespace {
        <log event methods>
}
```

Listing 4.5: Example of NGMON's empty HdfsNamespace.

When all namespaces are filled with NGMON's new log events, they all are written on hard drive at once. *TranslatorStarter* initiates generation of dummy *LogGlobal* class, which substitutes all *isDebugEnabled(), isWarnEnabled(),..* calls and always returns true. NGMON can easily filter these levels on its own when is asked to. Another step is to create *SimpleLogger* class, which is a bridge between underlying NGMON's logging framework and NGMON itself. In LT by default, we have chose log4j2 to be the underlying logging framework. Both these files are written to appropriate locations in *$project_home/Logtranslator/src/main/java/* with compliance to Maven standards. By creating a new Maven module in target's application directory, we can easily build and incorporate newly translated LogTranslator project into target's application by modifying *pom.xml* file(s). Generation of new namespaces, *LogGlobal* and *SimpleLogger* files is done using StringTemplate[14] engine. StringTemplate is a great tool for generation of source code or any formatted text. It also can be found in the internals of ANTLR for code generation.

Final step is to show some basic information about run of LT, such as number of processed files, paths to generated namespaces, support and information files. Other information about run can be seen in LT's *logs* directory in *LogTranslator.log* file. Interestingly enough, this log file is generated by NGMON, as LT itself uses NGMON as default logging framework with Log4j2 backing it up.

From each modified log method we also generate a pattern, which is in compliance with GoMatch project's syntax. These collected patterns are used in GoMatch, to

14. StringTemplate project belongs to the same author as ANTLR has, Terrence Parr. Official website is `http://www.stringtemplate.org/`.

match log events from target's application. They are further evaluated in comparison with standard logging frameworks, NGMON Logger and regular expressions. More detailed output of LogTranslator's run can be found in Appendix A.1.

5 Testing LogTranslator with Apache Hadoop

5.1 Incorporating NGMON logging system into project

To be able to use NGMON logging system, you have to do the following things in your project[1]:

1. Make sure, that you have successfully installed *ngmon-logger-java* project, available from `https://github.com/ngmon/ngmon-logger-java` github repository.
2. Add NGMON Logger project as dependency into your Maven project - *pom.xml* file.
3. Create custom Logger class which will extend NGMON bridging Logger class between NGMON and underlying application. In our LogTranslator project, we chose Log4j2 as default underlying logging framework and SimpleLogger, which passes all log information events directly to file output.
4. Create Log instance from LoggerFactory which will use your Logger implementation or use SimpleLogger.
5. Use method calls on Log instance from already predefined namespaces for given application or defined in your own new namespace which will log events to your needs.
   Syntax for NGMON Logger log is mentioned at the beginning of Chapter 4. It is important to create logging method in appropriate namespace which would be named as event-name and have exactly the same formal parameters as particular log event call.

The aim of LogTranslator is to do the steps 3. - 5. automatically. User has to perform first two steps manually in order to successfully use NGMON logging framework. These two steps are easy to do, but not trivial for someone, who is not familiar with Maven building tool.

5.2 Applying LogTranslator to Apache Hadoop project

By running the LogTranslator on subproject of Apache Hadoop - *MapReduce*, we changed around 850 log method calls in MapReduce and Common Project. Because of multimoduled nature of Apache Hadoop, we had to manually add into parent's Maven project dependency on LogTranslator project. Therefore, sub-modules

1. We assume, that project is using Maven build management system.

of Hadoop could correctly pick up needed dependencies on methods defined in abstract namespaces. First encountered problem was the rare and unexpected usage of complicated log methods or logger itself used in conjunction with other objects ruining the compatibility. These issues had to be resolved manually.

In mentioned MapReduce project, we had to fix few non-conventional usages of log object, which caused minor problems. These issues were mainly connected with Hadoop's customized logging and reporting framework. There were 2 unknown methods for NGMON. The problem was, they were extending other class and logger object has been defined in parent's class, so they were in different NGMON's namespace. These methods had to be manually moved to correct namespace and LogTranslator project recompiled.

Common-core project had also similar problems and sometimes missing proper imports or log declaration. Despite the fact, that *common-core* is a really big project, there were around 60 problems directly and non-directly connected with the translation of log methods. In few cases, we had to completely comment out customized log settings or add null as argument for log object in *IOutils* method call, to be able to successfully compile Apache Hadoop. All issues directly connected with translation of another almost 900 log methods could be easily resolved in less then half an hour of manual fixing and rebuilding project. Non-direct issues took longer time. Apache Hadoop project has a habit of tying up loggers from production code classes and test classes. Therefore we had to manually break this connection, as at the beginning of LogTranslator project agreed not to translate testing methods at all. By using sophisticated IDE like IDEA in our case and partially compiling Maven sub-modules, we could relatively easily overcome this problem.

After sorting out these issues, we were able to successfully compile and run Hadoop Project with translated *MapReduce* and *common-core* sub-project.

In Listing 5.1, we can see some excerpts of logs from MapReduce's teragen example job. We have ran two same jobs each using different logging framework. First run was with original logging framework, second was with NGMON Logger and translated log statements. Original log is always above NGMON's translated log. We can easily spot the difference and compare mutual counterpart. We should note, that both logs were written by the same log4j2 logging framework's formatting pattern. That is why the time stamp and the level of both logs look the same.

```
2014-05-05 15:07:48,908 DEBUG org.apache.hadoop.metrics2.impl.MetricsSystemImpl - UgiMetrics, User and
     group related metrics

2014-05-05 15:09:10,441 DEBUG Log4jLogger - {"Event":{"tags":[],"type":"finalnamefinalde","level":10000,
     "_":{"schema":"log_events.org.apache.hadoop.Metrics2Namespace","finalName":"UgiMetrics","finalDesc
     ":"User and group related metrics"}}}


2014-05-05 15:07:49,073 DEBUG org.apache.hadoop.security.UserGroupInformation - UGI loginUser:mtoth (
     auth:SIMPLE)

2014-05-05 15:09:10,897 DEBUG Log4jLogger - {"Event":{"tags":[],"type":"privilegedaction_from","level"
     :10000,"_":{"schema":"log_events.org.apache.hadoop.SecurityNamespace","testingGroups":"mtoth (auth
     :SIMPLE)","where":"org.apache.hadoop.mapreduce.Job.connect(Job.java:1253)"}}}


2014-05-05 15:07:50,830 INFO  org.apache.hadoop.mapreduce.Job - Job job_local1482522238_0001 running in
     uber mode : false

2014-05-05 15:09:12,614 DEBUG Log4jLogger - {"Event":{"tags":["methodCall"],"type":"
     job_running_uber_mode","level":20000,"_":{"schema":"log_events.org.apache.hadoop.
     MapreduceNamespace","jobId":"job_local1281492327_0001","isUber":false}}}


2014-05-05 15:07:50,836 INFO  org.apache.hadoop.mapreduce.Job - Job job_local1482522238_0001 completed
     successfully

2014-05-05 15:09:12,620 DEBUG Log4jLogger - {"Event":{"tags":[],"type":"job_completed_successfully","
     level":20000,"_":{"schema":"log_events.org.apache.hadoop.MapreduceNamespace","jobId":"
     job_local1281492327_0001"}}}
```

Listing 5.1: Example of two MapReduce's teragen example job runs. First using original logging framework, second with NGMON Logger.

5.3 Performance testing of translated application

This chapter compares the performance of Apache Hadoop with the original logging framework and Apache Hadoop with NGMON translated logging system. We conducted 4 simple tests using example jobs from Hadoop's Mapreduce subproject. Each of the tests was run 5 times on Hadoop with default logging framework, with NGMON logging framework using default Log4j2's file appender and third run using asynchronous appender. As a timing device we used GNU/Linux command *time*, focusing on *real* output value, which clocks *elapsed real time (in seconds)*. Tests were conducted on a GNU/Linux Ubuntu 14.04 server running on AMD Opteron Processor 4284 quad-core with 8GB of RAM memory.

First example *wordcount*, expects a text file input and gathers the count of each unique word found in it. We have used 31MB big text file, which has been concatenated from various books, obtained from Project Gutenberg[2] website. Results showed, that examples ran with NGMON logger were slower by 7.93%, making it in average 1.6066 seconds longer then the same test without NGMON logger. Average non-NGMON example run for 18.6392 second and with NGMON 20.2458 seconds.

2. Project Gutenberg's website `http://www.gutenberg.org/ebooks/`

When NGMON uses asynchronous appender, results were even worse by 10.47% making it almost whole 2 seconds. On Chart 5.1 and Table 5.1 you can see the individual results gained from this test.

WordCount	1	2	3	4	5	Average	Diff
Hadoop	18.062	19.044	18.04	19.044	19.006	18.639	-
NGMON	20.526	20.16	20.13	20.181	20.232	20.245	1.606
NGMON Async	20.887	20.537	20.509	20.511	20.516	20.592	1.952

Table 5.1: Apache Hadoop's Wordcount example test run with Hadoop's default logging framework, NGMON logger and NGMON logger using asynchronous appender.

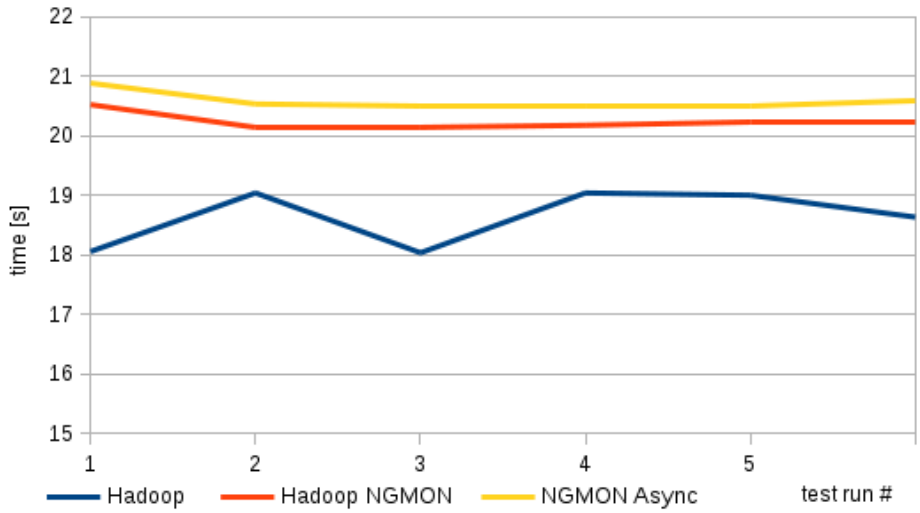

Figure 5.1: Apache Hadoop's Wordcount example test

As for the second test from Mapreduce's example jobs, *the approximation of* π *value* by quasi Monte-Carlo method has been chosen. Test used 10 maps and 10 000 000 000 samples for approximation. Results were more positive for NGMON logger this time, in comparison with gained results from first test. NGMON logger was slowing down the application by 3.67% which was around 14.5 seconds by average. When using asynchronous appender, Hadoop's job was slower by another 2.1 seconds. Jobs with NGMON were slower by 3.8% and 4.3% respectively. Average execution time spent by example job without NGMON logger was 380 seconds comparing to 394.5 seconds using NGMON logger. You can see test runs on Figure 5.2

and Table 5.2.

Pi (Monte-Carlo)	1	2	3	4	5	Average	Diff
Hadoop	378.82	380.86	383.87	378.72	377.79	380.01	-
Hadoop NGMON	397.80	395.61	392.77	392.84	393.63	394.53	14.517
NGMON Async	403.34	395.64	397.75	393.84	392.68	396.65	16.637

Table 5.2: Apache Hadoop's Pi approximation example test run with Hadoop's default logging framework, translated NGMON logger and NGMON logger using asynchronous appender.

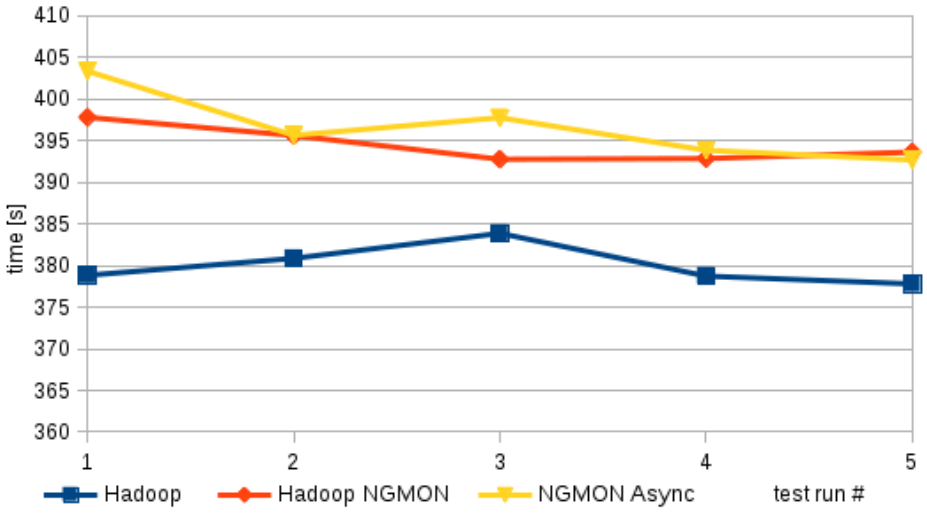

Figure 5.2: Apache Hadoop's Wordcount example test

Teragen and *Terasort* example jobs were chosen as third and fourth test. Teragen generates data of given size of rows, where one row is 100 bytes big. Generated output data are then handed to Terasort test, which sorts these random data. There is also a third Mapreduce's example job - *Teravalidate*, which validates the sorted data, but we have not used it for our testing purposes of NGMON Logger. We tested Teragen when creating 50000000 rows of data, a 5 GB file.

Our results show, that when using NGMON, generation of 5 GB file was a bit slower. but not as significantly as in previous tests. In both cases, generation was slower by 4.3%, which makes 4.3 seconds then when using default logging frame-

work. Average times of running Teragen job were 100.25 for non-NGMON job and 104.5 seconds for Hadoop with NGMON. In Figure 5.3 and Table 5.3 you can see an individual results of generating data files test.

Teragen	1	2	3	4	5	Average	Diff
Hadoop	100.28	100.041	99.96	100.05	100.92	100.25	-
Hadoop NGMON	102.00	111.09	100.99	102.62	106.23	104.591	4.338
NGMON Async	107.74	104.05	102.16	102.91	105.90	104.55	4.304

Table 5.3: Apache Hadoop's Teragen example test run, generating 1GB file with Hadoop's default logging framework, translated NGMON logger and NGMON logger using asynchronous appender.

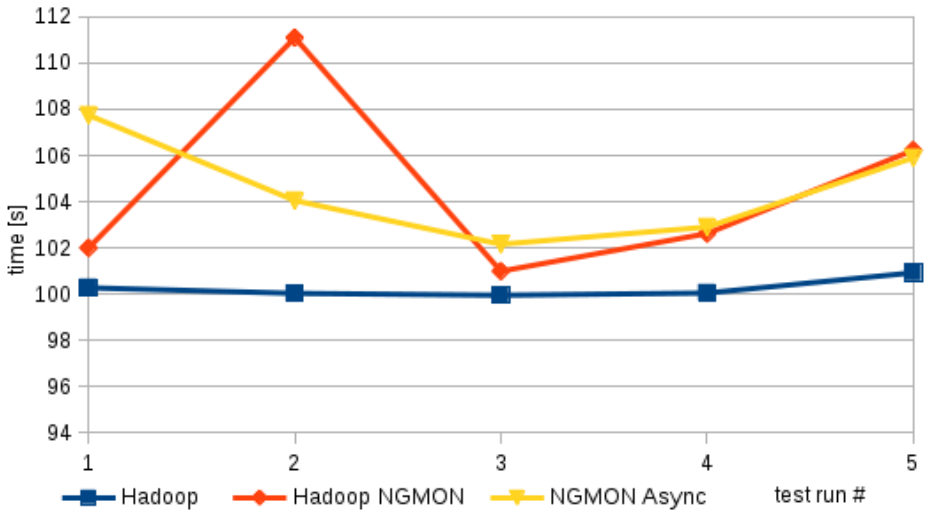

Figure 5.3: Apache Hadoop's Teragen example test

As our last, fourth test, we fed Terasort with 1 GB file and let it sort this input file. Average times of sorting were 55.7 for default NGMON logger and 67.2 seconds for NGMON using asynchronous appender, versus 57.3 seconds of non-translated Hadoop job. That means, job execution of translated Hadoop application using NGMON Logger was 2.76% slower for non-asynchronous appender and significantly slower by 20.6% for asynchronous appender. You can see the test results on Table 5.3 and Figure 5.4

Terasort	1	2	3	4	5	Average	Diff
Hadoop	76.62	54.987	47.787	49.287	50.015	55.739	-
Hadoop NGMON	67.691	53.882	52.597	58.958	53.489	57.323	1.584
NGMON Async	99.621	70.574	58.703	55.59	51.66	67.229	11.490

Table 5.4: Apache Hadoop's Terasor example test run sorting 1GB data file with Hadoop's default logging framework, translated NGMON logger and NGMON logger using asynchronous appender.

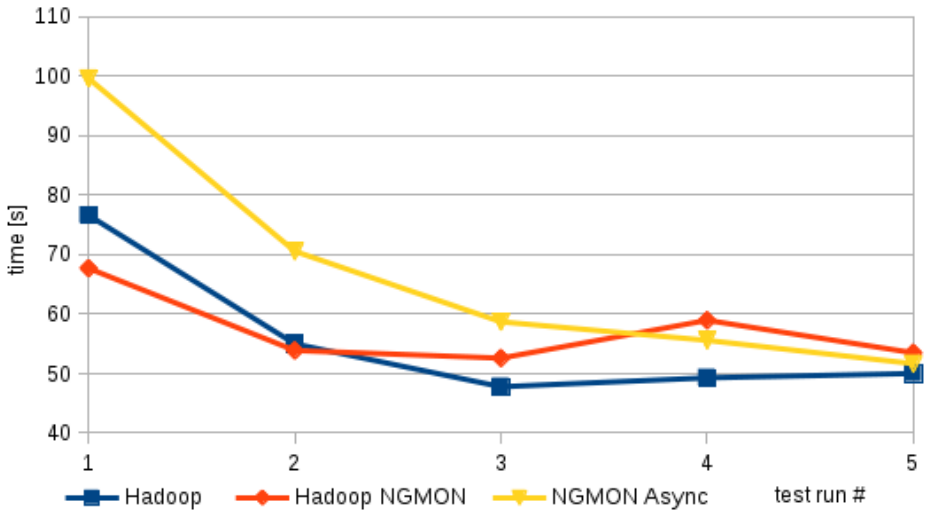

Figure 5.4: Apache Hadoop's Terasort example test

We think, that some of these results should not be blind-fully trusted as Apache Hadoop uses its own logging framework, which might lead to not as objective results. We also can see, that cached data could result in another test quickness, despite our efforts to make tests as isolated as possible. We recommend further testing on application(s), which does use only native logging systems like those, mentioned in Chapter 2.1. From test execution results between NGMON's two appenders, we can clearly see, that using default log4j2's appender is generally faster or at least as fast as using asynchronous appender. Apart from Hadoop's custom logging system, our results were as we had expected. It should logically take a bit longer to process inserted log to JSON and write it out, than not doing this operation at all.

6 Conclusion

On our way to find and implement the most manageable and suitable solution to translate the log method statements, there were three stages. Collecting information about currently used logging frameworks, looking for possibilities and experiences with various code rewriting tools, as well as implementing and testing the semi-automatically translated application.

Chapter 2 provides a short introduction into the evolution of Java logging frameworks. Starting from the first proposal of *JSR 47: Logging API Specification* and implementation in *java.util.logging* package of Java 1.4, through advanced frameworks like Log4j, Apache Commons Logging, Slf4j or Log4j2. That was followed by discussion about benefits of the structured logging over the natural language logging, with presenting a Common Event Expression and New Generation Monitoring Logger project.

In the further chapters we consider options of how to rewrite source code, which is not running in Java Virtual Machine[1]. Possible options were completely manual rewriting or semi-automated rewriting of source code combined with hand rewriting or rewriting by means of simple automated script. More advanced options were manipulating the structure of a syntax tree with the help of Eclipse Java Development Tools or Java Compiler API in conjunction with Pluggable Annotation Processors. Finally, we chose the *ANother Tool for Language Recognition* tool, which allowed us to implement whole log statements rewriting application and make it easily extendable.

The second half of thesis displays the usage of implemented application on Apache Hadoop. We have successfully proved, that we are able to change some parts[2] of Java application, which is using Maven management and build tool. Regrettably, it has to be admitted, that for the moment, it does not seem possible to fully automatically convert whole application to NGMON Logger's framework without performing some minor alterations to original source code. Especially in cases, when application is complex, which means that it consists of classes, reusing loggers from other classes, which are defined in different packages, or loggers used also in tests classes, or in cases, when log statement expressions are very complex and use some badly designed constructions, it is needed to fix these problems manually. Also, new problems arise when target application has its own logging framework, which

1. That is the reason, why we have not mentioned anything about Reflection API.
2. Whole application can be also successfully translated to NGMON Logger's statements syntax.

is hard to work with. However as we have said before, LogTranslator can convert around 4700 log statements in about 15 seconds, which is a great achievement. On the contrary, sometimes manual work has to be done, in order to successfully compile and run target application.

Both original Apache Hadoop and translated Apache Hadoop using NGMON Logger have been tested and compared, just to confirm our hypothesis, that translated application has to run for a longer time period, due to inserting of individual logs into NGMON log processor and outputting it in JSON format. Although, this has to be admitted, that it might be useful to similarly test a different application, which does not use any customized logging framework as Apache Hadoop does. In our opinion, the results would be more evident and transparent.

We believe, that goals set in the beginning of the thesis have been fulfilled and the designed LogTranslator application can be successfully used and further improved, if needed. LogTranslator can be extended by supporting Java Logging API, various new expressions in log methods, which we did not handle, since Apache Hadoop did not contain them. After all, natural language is very rich, and human's ingenuity is never-ending as well.

Bibliography

[1] *Java^{TM} Logging Overview*, `http://docs.oracle.com/javase/7/docs/technotes/guides/logging/overview.html`, Updated November 26 2001, Oracle and/or its affiliates, [cited 15.11.2013].

[2] Chua Hock-Chuan, *Java Programming Java Logging Framework*. `http://www.ntu.edu.sg/home/ehchua/programming/java/JavaLogging.html`, November 2012, [cited 15.10.2013].

[3] *Apache Commons Logging*, `http://commons.apache.org/proper/commons-logging/guide.html`, Apache Software Foundation, The, [cited 15.11.2013].

[4] *Optimization: Don't do it... The compiler will!*. `http://javamoods.blogspot.cz/2010/02/optimization-dont-do-it-compiler-will.html`, February 2, 2010, [cited 17.11.2013].

[5] Ceki Gülcü, *Short introduction to log4j*. `http://logging.apache.org/log4j/1.2/manual.html`, March 2002, [cited 15.10.2013].

[6] *Logback Project*. `http://logback.qos.ch`, Quality Open Software, [cited 8.12.2013].

[7] *Apache Log4j 2, User's Guide*. `http://logging.apache.org/log4j/2.x/log4j-users-guide.pdf`, Apache Software Foundation, The, version 2.0-beta9, [cited 28.11.2013].

[8] Martin Thompson, Dave Farley, Michael Barke, Patricia Gee, Andrew Stewart, *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*. `http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf`, May 2011.

[9] Ceki Gülcü et. al, *Simple Logging Facade for Java (SLF4J)*. `http://www.slf4j.org/manual.html`, Quality Open Software, [cited 10.12.2013].

[10] M. GonÇalves, M. Luo, R. Shen, M. Ali, and E. Fox, "An xml log standard and tool for digital library logging analysis," in Research and Advanced Technology for Digital Libraries (M. Agosti and C. Thanos, eds.), vol. 2458 of Lecture Notes in Computer Science, 2002.

[11] *Common Event Expression*, `http://cee.mitre.org/`. The MITRE Corporation, [cited 4.1.2013].

[12] Tovarňák, Daniel, Tomáš Pitner, *Towards Multi-Tenant and Interoperable Monitoring of Virtual Machines in Cloud*. In Andrei Voronkov, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen Watt, Daniela Zaharie. *Proceedings of 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing.*, Los Alamitos (CA): IEEE Computer Society, 2012. s. 436-442, ISBN 978-1-4673-5026-6.

[13] Tovarňák, Daniel, Tomáš Pitner, *Distributed Event-driven Model for Intelligent Monitoring of Cloud Datacenters*. In Zavoral, Filip and Jung, Jason J. and Badica, Costin. *Intelligent Distributed Computing VII. Cham: Springer International Publishing Switzerland*, 2013. ISBN 978-3-319-01570-5.

[14] Tovarňák, Daniel, Tomáš Pitner, Andrea Vašeková, Svatopluk Novák, *Structured and Interoperable Logging for the Cloud Computing Era: The Pitfalls and Benefits*. In Proceedings of 6th IEEE/ACM International Conference on Utility and Cloud Computing, 2013.

[15] Miloslav Trmač, *Do Not Believe in Structured Logging*. online `http://carolina.mff.cuni.cz/~trmac/blog/2011/structured-logging`, [cited 10.1.2013].

[16] Scott Chacon, *PRO Git*. online `http://git-scm.com/book`, [cited 4.11.2013].

[17] Authors from tutorialspoint.com, *Apache Maven Tutorial*. online `http://www.tutorialspoint.com/maven`, [cited 9.11.2013].

[18] Tim O'Brien, John Casey, Brian Fox, Bruce Snyder, Jason Van Zyl, Eric Redmond, *Maven: The Definitive Guide*. Online version, Sonatype, Inc., Palo Alto, CA 94301, 2006 – 2008.

[19] Oracle Corporation, *Compilation Overview*. online `http://openjdk.java.net/groups/compiler/doc/compilation-overview/`, [cited 5.4.2014].

[20] David Erni, Adrian Kuhn, *The Hacker's Guide to Javac*. online `http://scg.unibe.ch/archive/projects/Erni08b.pdf`, University of Bern, March 2008.

[21] Tom White, *Hadoop: The Definitive Guide*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st Edition, 2009.

[22] Terrence Parr, *Definitive ANTLR 4 Reference, The*. Pragmatic Bookshelf, 2nd Edition, 2012.

[23] Terrence Parr, *Language Implementation Patterns*. Pragmatic Bookshelf, P1.0 printing, December 2009.

[24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st Edition, November 1994.

A Supplement

A.1 Appendix A

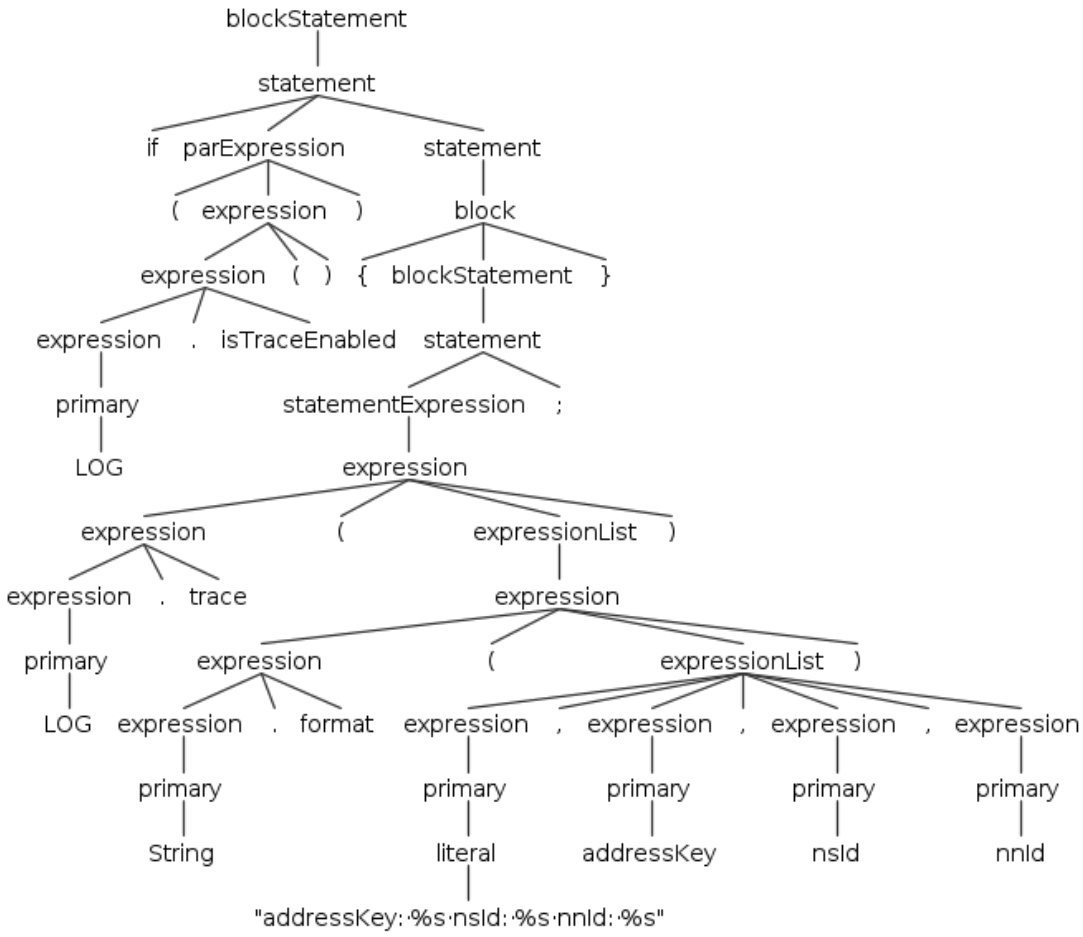

Figure A.1: Typical log method statements generated by ANTLR, which have to be translated into NGMON's syntax.

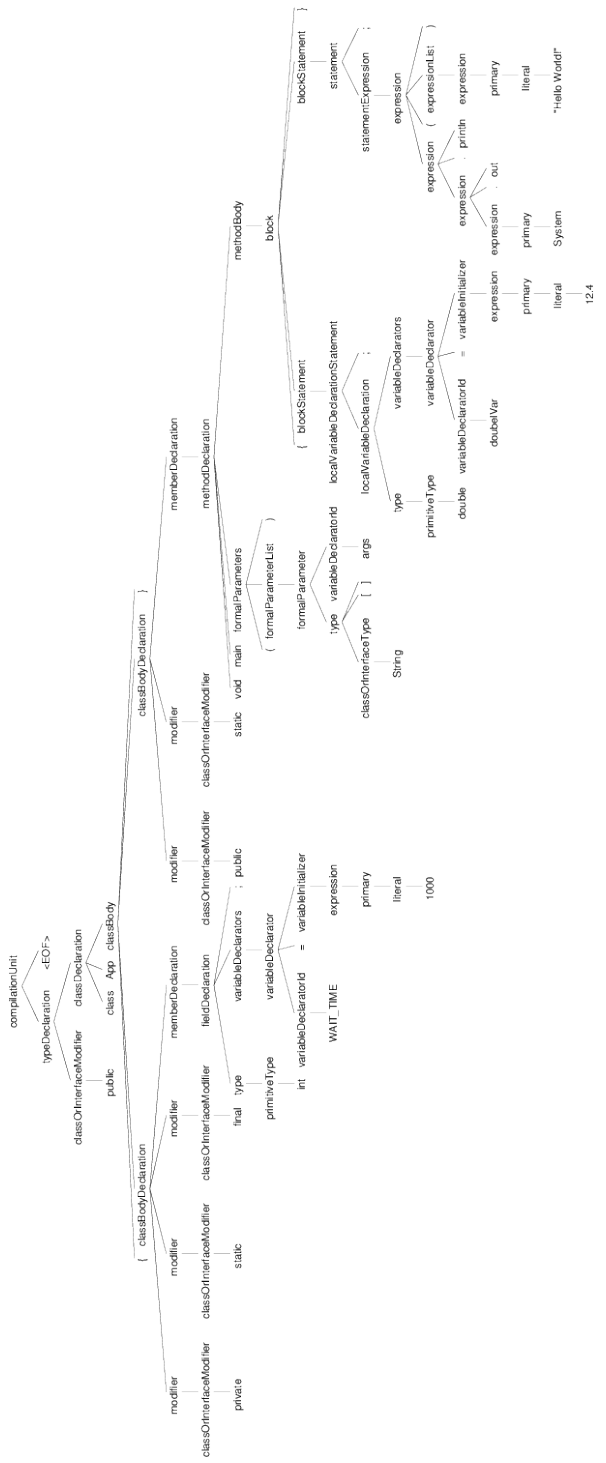

Figure A.2: ANTLR's parsed tree of App.java source code 4.4

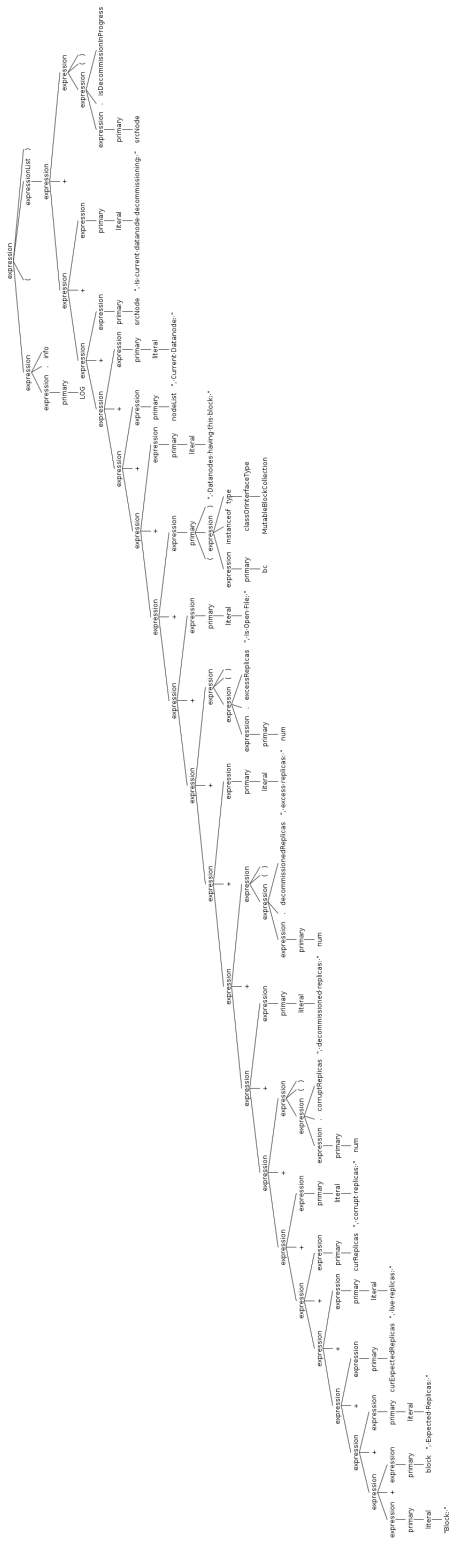

Figure A.3: A fine example of leftmost derivation in left-to-right grammar parser - ALL(*), which uses ANTLR by default. For full scale image see picture on the following link `https://github.com/michalxo/LogTranslator/blob/master/antlr4_left_recursion.png`.

A.2 Appendix B

Example of LogTranslator's output run. On the first line is default log method found in Java class. Second line is NGMON Logger's substitution method for original log method. On the third line is generated NGMON Logger's method declaration for NGMON's log method. Fourth line contains generated Go-Match pattern and last fifth line is path to given file.

```
LOG.warn("Inprogress znode "+child+" refers to a ledger which is empty. This occurs when the NN"+"
     crashes after opening a segment, but before writing the"+" OP_START_LOG_SEGMENT op. It is safe to
     delete."+" MetaData ["+l.toString()+"]")
LOG.inprogress_znode_refers_ledger_which_emp(child, String.valueOf(l.toString())).tag("methodCall").warn
     ()
public AbstractNamespace inprogress_znode_refers_ledger_which_emp(String child, String lMethodCall)
org.apache.hadoop.contribnamespace.inprogress_znode_refers_ledger_which_emp##Inprogress znode %{STRING:
     child} refers to a ledger which is empty. This occurs when the NN crashes after opening a segment,
      but before writing the OP_START_LOG_SEGMENT op. It is safe to delete. MetaData [%{STRING:
     lMethodCall}]
/home/mtoth/tmp/rewritting/hadoop-common-pure/hadoop-hdfs-project/hadoop-hdfs/src/contrib/bkjournal/src/
     main/java/org/apache/hadoop/contrib/bkjournal/BookKeeperJournalManager.java

LOG.error("Interrupted while purging "+l,ie)
LOG.interrupted_while_purging(l.toString(), ie.toString()).error()
public AbstractNamespace interrupted_while_purging(String l, String Exception)
org.apache.hadoop.contribnamespace.interrupted_while_purging##Interrupted while purging %{STRING:l} %{
     STRING:Exception}
/home/mtoth/tmp/rewritting/hadoop-common-pure/hadoop-hdfs-project/hadoop-hdfs/src/contrib/bkjournal/src/
     main/java/org/apache/hadoop/contrib/bkjournal/BookKeeperJournalManager.java

LOG.error("Couldn't delete ledger from bookkeeper",bke)
LOG.couldnt_delete_ledger_from_bookkeeper(bke.toString()).error()
public AbstractNamespace couldnt_delete_ledger_from_bookkeeper(String Exception)
org.apache.hadoop.contribnamespace.couldnt_delete_ledger_from_bookkeeper##Couldn't delete ledger from
     bookkeeper %{STRING:Exception}
/home/mtoth/tmp/rewritting/hadoop-common-pure/hadoop-hdfs-project/hadoop-hdfs/src/contrib/bkjournal/src/
     main/java/org/apache/hadoop/contrib/bkjournal/BookKeeperJournalManager.java

LOG.error("Error deleting ledger entry in zookeeper",ke)
LOG.error_deleting_ledger_entry_zookeeper(ke.toString()).error()
public AbstractNamespace error_deleting_ledger_entry_zookeeper(String Exception)
org.apache.hadoop.contribnamespace.error_deleting_ledger_entry_zookeeper##Error deleting ledger entry in
      zookeeper %{STRING:Exception}
/home/mtoth/tmp/rewritting/hadoop-common-pure/hadoop-hdfs-project/hadoop-hdfs/src/contrib/bkjournal/src/
     main/java/org/apache/hadoop/contrib/bkjournal/BookKeeperJournalManager.java

LOG.warn("ZNode: "+legderMetadataPath+" might have finalized and deleted."+" So ignoring NoNodeException
     .")
LOG.znode_might_have_finalized_and_deleted_i(legderMetadataPath).warn()
public AbstractNamespace znode_might_have_finalized_and_deleted_i(String legderMetadataPath)
org.apache.hadoop.contribnamespace.znode_might_have_finalized_and_deleted_i##ZNode: %{STRING:
     legderMetadataPath} might have finalized and deleted. So ignoring NoNodeException.
/home/mtoth/tmp/rewritting/hadoop-common-pure/hadoop-hdfs-project/hadoop-hdfs/src/contrib/bkjournal/src/
     main/java/org/apache/hadoop/contrib/bkjournal/BookKeeperJournalManager.java

LOG.trace("Setting maxTxId to "+maxTxId)
LOG.setting_maxtxid(maxTxId).trace()
public AbstractNamespace setting_maxtxid(long maxTxId)
org.apache.hadoop.contribnamespace.setting_maxtxid##Setting maxTxId to %{LONG:maxTxId}
/home/mtoth/tmp/rewritting/hadoop-common-pure/hadoop-hdfs-project/hadoop-hdfs/src/contrib/bkjournal/src/
     main/java/org/apache/hadoop/contrib/bkjournal/MaxTxId.java
```

Listing A.1: Excerpt from LogTranslator's run output, which shows orginal log statement method and its appropriately generated counterparts.

List of Figures