Bachelor Thesis

Title

Impact of Logging Configuration Changes in Source Code and Project Configuration File

Supervisor
Professor Katsuro Inoue

Author
Liang Qiu

2022/2/8

Department of Information and Computer Sciences
School of Engineering Science, Osaka University

Reiwa 3rd Bachelor Thesis

Impact of Logging Configuration Changes in Source Code and Project Configuration File

Liang Qiu

Abstract

Logs are essential component of software programs and widely used by software developers. There are two ways to do logging configurations, the one in source code and another in project configuration file, playing an important role on the practicality, constancy, and stability of logging. Although recent research has been conducted to understand current practice on logging statements in source code and importance of project configurations file, but no existing research focuses on the impact of logging configuration changes in source code and project configuration file. To fill this gap of lacking studying collective effect of project configuration file and logging statements in source code, in this paper, we conduct an exploratory study for aiming to investigate the impact on the logging output of changes in logging statements in source code and project configuration file. The subject of the survey is the open-source project ActiveMQ which is written in Java using the log4j library. We confirm that the logging statements output status are more affected by project configuration file change than by logging statements change in source code. Developers change the log level in source code or the threshold in project configuration file to change the log statements output status.

Keywords

Logging statements
Log level
Project configuration file

Contents

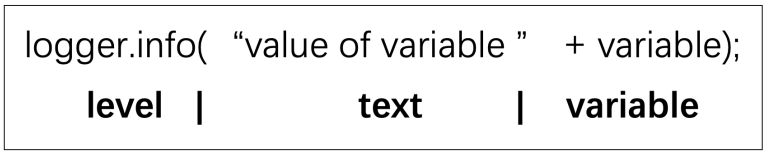

Figure 1: An example of a logging statement

1 Introduction

Log is an essential component of software programs and widely used by software developers for recording valuable run-time variable information and error messages during program execution [7]. The logging components deal with the demand from the application code and output the logging information to the specific targets. If the logs are not properly output, it will be difficult to find the cause of the problem when it occurs. On the other hand, proper log output makes it easier to identify the cause of the system problem. Using run-time information, we can handle failure diagnosis and make it easier to understand the program, and so on. A logging statement, typically consists a log level to specify the severity of the logged event, a textual part indicating the event, and one or more variables [3][4][13]. An example of a logging statement is shown in Figure 1.

However, appropriate logging is difficult to reach in practice. Both logging too little and logging too much is undesirable [3]. Too detailed logging may create unsustainable overhead for the system while too rough log may miss critical information. So we can say that it is not a trivial task to set up a properly designed logging component, considering the huge kinds of logging requests. To address the trade-off, popular logging libraries such as Apache Log4j [2] provide log levels to control the number of log messages recorded on the disk [9]. The logging libraries can simplify the writing of the logging component, but inserting log requests into the application code requires a fair amount of planning and effort.

There are two ways to do logging configurations, the one in source code and another in project configuration file, playing an important role on the practicality, constancy, and stability of logging. Prior researches find that many problems exist in the practice of writing logging statements, such as missing failure information [12], improper logging level [13], and duplicated log message [8]. Besides writing logging statements in source code, setting project logging configurations is also highly crucial for output essential log

messages. Previous studies find that about 32.4% of loggers are configured to control the logging activities of external libraries [15]. Although recent research has been conducted to understand current practice on logging statements in source code and importance of project configurations file, but no existing research focuses on the relationship of logging statements in source code and project configurations file.

To fill this gap of lacking studying collective effect of project configuration file and logging statements in source code, in this paper, we conduct an exploratory study for aiming to investigate the impact of changes in logging statements in source code and project configuration file on the logging output. The subject of the survey is the open-source project ActiveMQ[1] which is written in Java using the log4j library. Our preliminary study confirms that there are enormous logging statements in source code through the project ActiveMQ. The log level of these logging statements change frequently over the development of the project ActiveMQ which consists with the prior study showing that developers take great effort on maintaining and updating log level over the lifetime of a project. We explore the changes in development history by tracing the development history to clarify the impact of logging configuration changes in source code and project configuration file. From our study we find that the log statements output status are more affected by project configuration file change than by log level change in source code. With the result, we can argue that developers should be more careful when changing project configuration file.

Paper organization. The remainder of the paper is organized as follows. Section 2 introduce the prior study about logging and some background knowledge about logging. Section 3 describes the studied open source project and performs an empirical study on the log level distribution and track the change of log level and configuration file threshold in the studied project. Section 4 describes the research questions we are concerned about and the approached that we used to answer these questions. Section 5 presents the results of our case study. Section 6 discusses threats to the validity of our findings. Finally, Section 7 draws conclusions.

2 Background

In this section, we discuss the existing study about logging, log level, and logging configuration. Also, we introduce the specification of the log4j library which we use as the study case in this research.

2.1 Prior study on logging

Prior research on logging statements has been conducted from three main perspectives.

What to log for. The rich information in logs is essential for many tasks in the software development, such as failure diagnosis [6][11], performance analysis [5], and user behavior analysis [14].

How to log. The logging component receives logging requests from application source code and outputs the formatted logging information to the specified destinations [15].

Where to put log statements. To avoid logging too little or too much, developers need to proper decision on where to log in their logging practices during development. Prior study divided the place of developer log into 5 categories, *Assertion-check logging*, *Return-value-check logging*, *Exception logging*, *Logic-branch logging* , *Observing-point logging*. The category of observing-point logging has the highest number of logged snippets among the five categories [3].

2.2 Logging mechanism

Log levels are beneficial for both developers and users to determine the suitable quantity of logs to output during the running of the program. With log levels, developers and users can allow the output of logs for crucial events (e.g., errors), while block logs for less important events (e.g., debug events). Developers and users use the mechanism of to trade-off the enormous information in logs. Common logging libraries such as Apache Log4j [2] typically has six log levels, including trace, debug, info, warn, error, and fatal. The most verbose log level is “trace” and the least verbose log level is “fatal”. The log level of logging statements to be output during running is controlled by the users. For instance, a user determines the log level to output at the “info” level and “info” will be the threshold. Only the logging statements with the “info” level or with a log level that is less verbose than “info” (“warn”, “error” and “fatal”) would output. The following code snippet shows an illustrative example for the logging mechanism with setting log level in source code.

Listing 1: The logging mechanism with setting log level in source code

```
//Logging code
import org.apache.log4j.Logger;
public class log4jTest{
        public void logging(){
                BasicConfigurator.configure();
                //default log level:info
                Logger logger = Logger.getLogger(log4jTest.class);
                logger.info("access log")
                logger.trace("will not output")
        }
}
//Generated Log
INFO log4jTest  - access log
```

To help manage these log statements without the need to modify them manually, the libraries also provide logging configurations, which can allow users to control logging behaviors not embedded within their code but from an external configuration file. The following code snippet shows an illustrative example for the logging mechanism with setting threshold in configuration file.

Listing 2: The logging mechanism with setting threshold in configuration file

```
//Logging code
import org.apache.log4j.Logger;
public class log4jTest{
        public void logging(){
                // no need to embed within code
                Logger logger = Logger.getLogger(log4jTest.class);
                logger.info("access log")
                logger.trace("will output this time")
        }
}

//Project configuration
log4j.rootLogger = trace, console
log4j.console.appender = org.apache.log4j.ConsoleAppender
log4j.appender.console.layout = org.apache.log4j.SimpleLayout

//Generated Log
INFO log4jTest  - access log
TRACE log4jTest - will output this time
```

2.3 Prior study on log level and logging configuration

Prior research finds that developers often struggle with choosing an appropriate log level for each log statement and take great effort on maintaining and updating log level over the lifetime of a project [13]. Li et al. leverage ordinal regression models to automatically suggest the most appropriate level for each newly-added logging statement [7]. Zhi et al. conduct a research on how logging configurations are used and revealed lots of findings about current practice of project configurations files [15]. However, there is no existing research on the collective impact of log level in source code and project configuration file.

Table 1: AN OVERVIEW OF THE STUDIED SYSTEM

System	LOC[1]	NOC[2]	NOF[3]	NOCOF[4]	Studied develop. history
ActiveMQ	459K	10.9K	5.2K	42	2005-12 to 2022-01

[1] LOC refers to the lines of code.

[2] NOC refers to the number of commits.

[3] NOF refers to the number of files.

[4] NOPCF refers to the number of project configuration files.

3 Preliminary Study

Before studying the impact of logging configuration changes in source code and project configuration file, we conduct a preliminary study in order to identify the conditions of the characteristics of the studied system. This preliminary study includes the overview of the studied system and the distribution of log level in source code and thresholds in project configuration file. Also, we track the changes of log level in source code and threshold in project configuration file to get the number of added log statements and the number of deleted log statements in each commit.

3.1 An Overview of the Studied System

Studied System. In this study, we focused on Apache ActiveMQ[1] project. The Java source code for these programs is massively using log4j logging library, which is the subject of this research. Table 1 shows an overview of the system.

3.2 Log Level Distribution in Source code and Project Configuration File Thresholds Distribution

As Algorithm 1 and Alogirthm 2 shows, we study the distribution of log level in source code and project configuration file thresholds.

Log Level Distribution in Source Code

We traversed all files of the target project. For the file with the suffix name“ .java”, we traversed each line of the file to determine if it contained LOG.info/warn/debug... or log.info/warn/debug... to get the number of each log level in java source code.

Project Configuration File Thresholds Distribution

For the project configuration file, we iterate through the file named “log4j.properties”.

Table 2: The Distribution of Log Levels in Source Code and Configuration File Thresholds

	fatal	**error**	**warn**	**info**	**debug**	**trace**
project configuration file thresholds	0	0	15	13	8	6
log levels in source code	0	513	479	3617	1336	368

Algorithm 1: Distribution of log level in source code and project configuration file thresholds

Input: project_filepath

Output: number of log level in source code

```
for file in filepath do
    if file.suffix equals '.java' then
        for line in file do
            if LOG.level or log.level in line then
                number of the log level ++
                //level is one of info, warn, debug, trace, error, fatal
            end
        end
    end
end
Output the number of log level;
```

In this studied system, all threshold-setting sentences start with "log4j.looger.org.apache" or "log4j.rootLogger". So we check the line containing "log4j.logger.org.apache" or "log4j.rootLogger" to determine if it contains INFO/WARN/DEBUG/TRACE/ERROR/FATAL to get the number of thresholds.

As result, we clarified the distribution of source code log levels and configuration file thresholds in the studied system. Table 2 shows the distribution of the log levels in source code and project configuration file thresholds in the studied system.

Algorithm 2: Distribution of project configuration file thresholds

Input: filepath

Output: number of project configutaion file thresholds

```
for file in filepath do
    if file.name equals 'log4j.properties' then
        for line in file do
            if 'log4j.logger.org.apache' or 'log4j.rootLogger' in line and '#' not in
             line then
                if threshold in line then
                    number of the threshold ++
                    //threshold is one of INFO, WARN, DEBUG, TRACE, ERROR,
                     FATAL
                end
            end
        end
    end
end
Output the number of thresholds;
```

3.3 Changes of Log Level in Source Code and Project Configuration File Threshold

As Table 2 shows, we found that there are enormous log statements in this study case. We want to figure out how many of them have ever been modified, that is, the log level in source code changes. So, we use *PyDriller* [10] which is a Python framework that helps developers in analyzing Git repositories to extract information about modified files. Specifically, we track the changed log levels in source code of the modified files on each commit to figure out the changes of log level in source code and proeject configuration file threshold. With *PyDriller*, we can get the diff parsed in a dictionary containing the added and deleted lines. The dictionary has 2 keys: “added” and “deleted”, each containing a list of Tuple (int, str) corresponding to (number of line in the file, actual line). As Algorithm 3 shows, we can traverse the whole dictionary with “added” and “deleted” as the key to get the number of added logs and the number of deleted logs respectively in each commit. Table 3 shows the result.

Table 3: The change of the log level in source code

	fatal	**error**	**warn**	**info**	**debug**	**trace**
number of added logs	15	1467	1448	6173	3271	870
number of deleted logs	15	954	969	2556	1935	502

From the results of the Table 3, we can see that developers change the log level frequently, which means that changing the log level is very costly for development. This is also consistent with the findings of previous study.

Algorithm 3: Number of added logs and deleted logs

Input: project_url

Output: number of added logs and deleted logs

```
for commit in the commit history of project_url do
    for file in the modified files of the commit do
        for line in the values of the dictionary corresponding the key 'added' of the
         file do
            if Log.level or log.level in line then
                number of the added log level ++
                //level is one of info, warn, debug, trace, error, fatal
            end
            if log4.logger.org.apache.activemq in line and threshold in line then
                number of the added threshold ++
                //threshold is one of INFO, WARN, DEBUG, TRACE, ERROR,
                 FATAL
            end
        end
        for line in the values of the dictionary corresponding the key 'deleted' of the
         file do
            if Log.level or log.level in line then
                number of the deleted log level ++
                //level is one of info, warn, debug, trace, error, fatal
            end
            if log4.logger.org.apache.activemq in line and threshold in line then
                number of the deleted threshold ++
                //threshold is one of INFO, WARN, DEBUG, TRACE, ERROR,
                 FATAL
            end
        end
    end
end
Output the number of added logs, deleted logs, added thresholds, deleted
 thresholds;
```

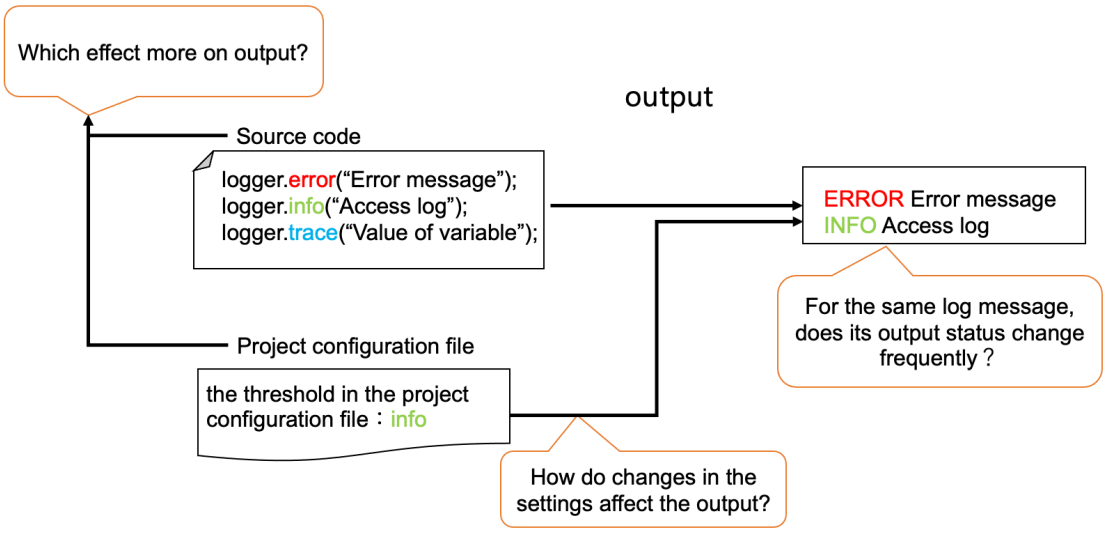

Figure 2: Issues of logging

4 Survey Outline

Even if the level of the log output statement or the threshold in the project configuration file is changed, there are cases where the final output does not change. As Figure 2 shows, both the log level in the source code and the threshold in project configuration file affect the output of the log, however we don't know how each change actually affects the output status of log statements. In this study, we would like to clarify the impact of logging configuration changes in source code and project configuration file. In particular, how the changes of log level in source code and threshold in project configuration file jointly determines the output is the subject we are most interested in studying here.

We use *PyDriller* to tracks changes in terms of commit, and analyse the impact of these changes on the log message output.

In each commit, some logs are completely deleted, some new logs are added, some logs are modified, and some logs remain unchanged. We use a dictionary to store these information. The keys are log messages, and the values are their log levels. With *PyDriller*, we can know whether the file is added, deleted, or modified for each commit. Each time a new log statement is added, we record its log level and log message as a pair on the dictionary. Each time a log statement is deleted, we remove the information about it from the dictionary. If it is modified, we modify the corresponding key and value. In this way, we can dynamically study the overall change process of the log. The Figure 3 shows the

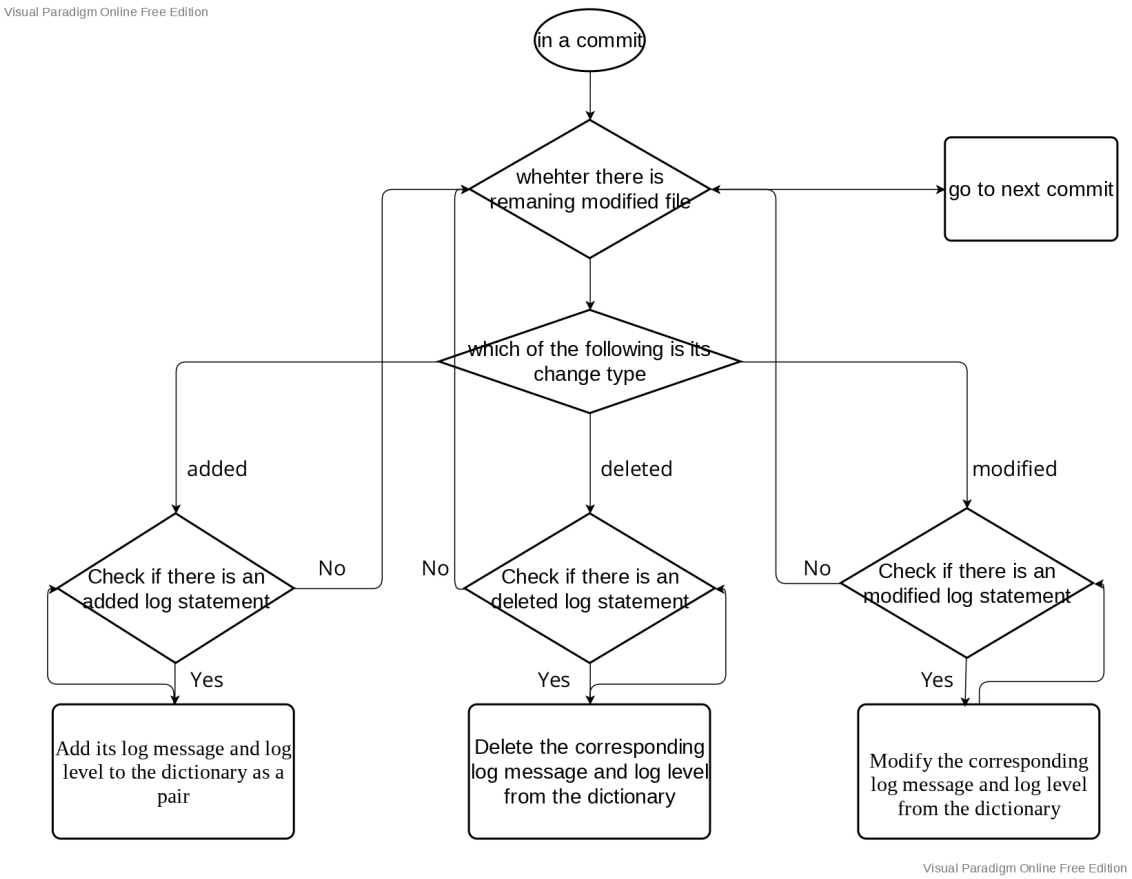

Figure 3: Flowchart_dynamic

flow chart of the entire dynamic process.

In order to study the effect of log changes on the output at a more detailed level, we set up the following research items.

4.1 RQ1: How does changing the log level and threshold of configuration file affect the output of the log?

Motivation. As we found in the preliminary study, the log levels and threshold of configuration file are frequently modified in a project development. In this research question, we want to figure out how these changes of log levels impact the output of the log.

Approach. Specifically, we will examine two detailed questions described below. We will calculate the answers to these two detailed questions, as well as analyze the meaning behind these results. Below, we describe the two detailed questions.

1:How many log output statements never change their log level out of all the log output statements?

For this question, we will count how many log statements have changed their log level, and then subtract these from the total number of log statements to get the number (percentage) that have not changed. In the dynamic analysis above, we track the modified log statement. If the log message has not changed and its log level or configuration file's threshold has changed, we count the change as a valid result. Note that if there is a log message that corresponds to a log level that changes more than once, we only count one of them as a valid result. For one modified log statement, if its log message changes, we will not treat it as a valid result even if its log level or configuration file's threshold has changed, because we will treat it as a new log statement once the log message has changed. We will give the answer to two questions on this issue.

1. In terms of each log statement, the percentage of log statements which both the log level in the source code and the threshold in the configuration file are unchanged.

2. In terms of each commit, the percentage of commits where there are no changes in the log level in the source code and the threshold in the configuration file.

2:How many log output statements change their log levels which have no effect on output?

As we described earlier, for a log statement, it is not only the log level that affects the output but also the configuration file threshold. For this problem, we not only track each log level change, but also check the configuration file threshold changes. As shown in Figure 4 for example, case 1 shows the change which the log level change has no effect on the output. Case 2 and case 3 show the change with effect on output. The change of log level in case 2 makes the log message that was to be output will no longer be output. And the change of log level in case 3 make the log message that was not to be output will be output.

4.2 RQ2: When the the log message output status is changed, what type of file is being changed, source code file, configuration file or both?

Motivation. In this research question, we figure out whether both log level and configuration file thresholds change have a significant impact on the output status change. If many of the changes in output status of the log message caused by changing both factors at the same time, then we could argue that comparing with the existing studies which only

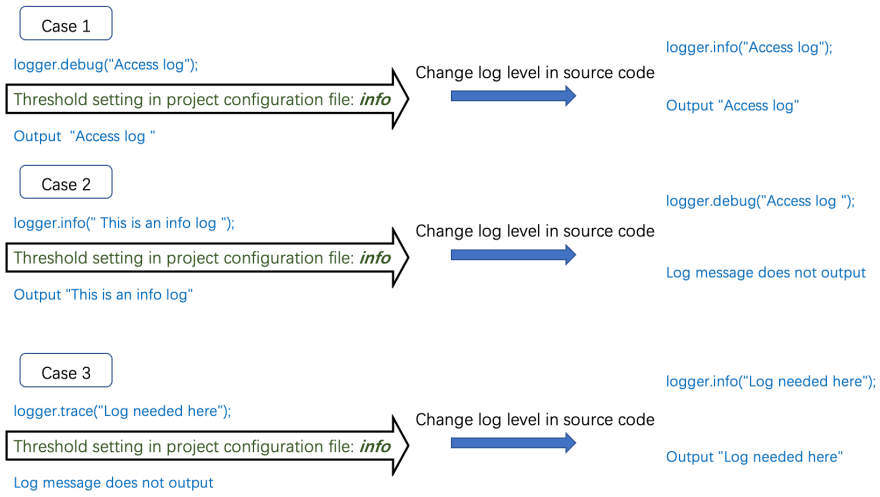

Figure 4: Case of log level change with and without effect

one of them has been studied intensively, both should be considered for further studies. If not, we could argue that we should focus more on the one with more changes.

Approach. In each commit, we check each log statement whether it will be output by its log level and its configuration file threshold. If its log level is lower than its configuration file threshold, then its log message will be filtered out and it will not be output. We use a dictionary to store these messages. Key is the log message, value is 1 or 0, where 1 means it will be output and 0 means it will not be output. We keep updating this dictionary on a per-commit basis, and when we have the same log message with opposite output, we check whether the log level has changed, or the configuration file threshold has changed, or both.

4.3 RQ3: For the same log message, does its output status change frequently?

Motivation. In this research question, we want to figure out whether there are many log statements that whether it is output or not change frequently or these changes of output are concentrated in a small number of logs.

Approach. In RQ2, we use a dictionary to save each log message and whether it is output. In this research question, we need to keep track of the number of changes on the log statement's output status. When log statement's output status changes, not only do we need to invert its output status(1 to 0, or 0 to 1) but also need to record how many times has it changed. We track their change history (e.g. from error $\rightarrow$ info $\rightarrow$ error, or

error → info → warn...)

Algorithm 4 and Algorithm 5 shows the entire analysis process. Algorithm 4 output the changes of the log messages' threshold and the changes of output status via configuration file threshold change. Algorithm 5 output the the changes of the log messages' log level, the changes of the output status via log level change, and the history of the log messages' log level.

Algorithm 4: Analyse process1

```
for commit in the commit history of project_url do
    for file in the modified files of the commit do
        if it is a configuration file then
            if it's change type equals 'ADD' then
                if added statement is a log statement then
                    1) get the threshold of this configuration file.
                    2) update the log message's threshold in the java file which has
                       the same prefix with the configuration file.
                    3) update the matter of whether the log message output or not
                       in the java file which has the same prefix with the configuration
                       file
                end
            end
            if it's change type equals 'MODIFY' then
                if modified statement is a log statement then
                    get the threshold of this configuration file.
                    if the threshold has changed then
                        1) update the log message's threshold in the java file which has
                           the same prefix with the configuration file.
                        2) record these changes
                        3) check if the matter of whether the log message output or not
                           has changed in the java file which has the same prefix with
                           the configuration file. If so, record these changes.
                    end
                end
            end
        end
    end
end
```

Algorithm 5: Analyse process2

```
for commit in the commit history of project_url do
    for file in the modified files of the commit do
        if it is a java file then
            if it's change type equals 'ADD' then
                if added statement is a log statement then
                    1) Set log message - log level pair, log message's history, log message's
                       output status (default:output).
                    2) Check if the log statement controlled by a configuration file. If so,
                       set the log message threshold consistent with the configuration file
                       threshold.
                    3) Check if the log message has a threshold. If so, set the log
                       message's output status corresponding to the log level and threshold.
                end
            end
            if it's change type equals 'MODIFY' then
                if modified statement is a log statement then
                    if the log message of the statement has been paired then
                        if the log level now differ from before then
                            1) update the log message - log level pair and record these changes
                               and update the log message's history: old log level + ' ' + new
                               log level
                            2) Check if the log message has a threshold. If so, set the log
                               message's output status corresponding to the threshold and the
                               new log level.
                        end
                    end
                    if the log message of the statement has not been paired then
                        Set log message - log level pair, log message's history.
                    end
                end
            end
        end
    end
end
```

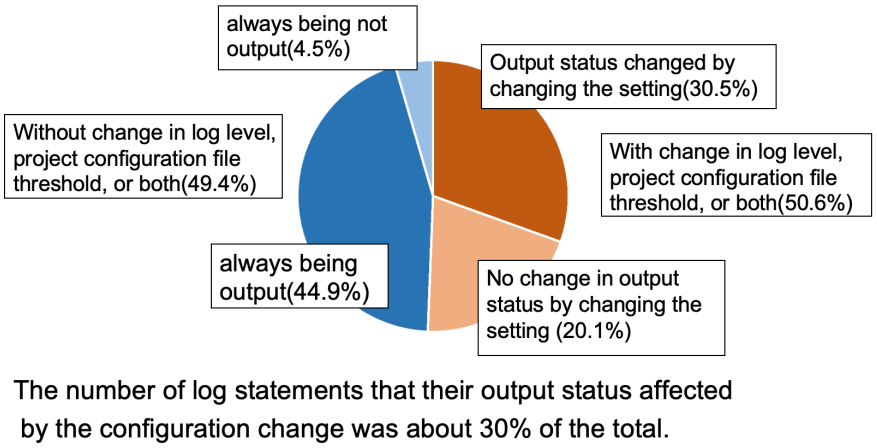

Figure 5: Log statements with and without logging configuration change

5 Results and Discussions

5.1 RQ1: Result

5.1.1 In the case of log statement unit

As Figure 5 show, there are 3,080 log statements in total. No log level and project configuration file threshold has changed in 49.4% (1521) log statements. And in these 49.4% (1521) log statements, 44.9% (1382) are always being output, and 4.5% (139) are always being not output. Log level , configuration file threshold, or both has changed in 50.6%(1559) log statements. In this case, 30.5% (939) of log statements changed their output status while 20.1% (620) has not changed. From the results above, we can know that about 30% log statements changed their output status in the process of the project evolution.

5.1.2 In the case of commit unit

As Figure 6 show, there are 10,966 commits in total. Changes in log level in source code and threshold in project configuration file occur in 2.7% of them. In those commits, 23.9% of them have output status change by changing the setting, 76.1% of them have no change in output status by changing the setting. From the results above, we can know that about 24% commits have output status change by changing the log level in source code and threshold in project configuration file.

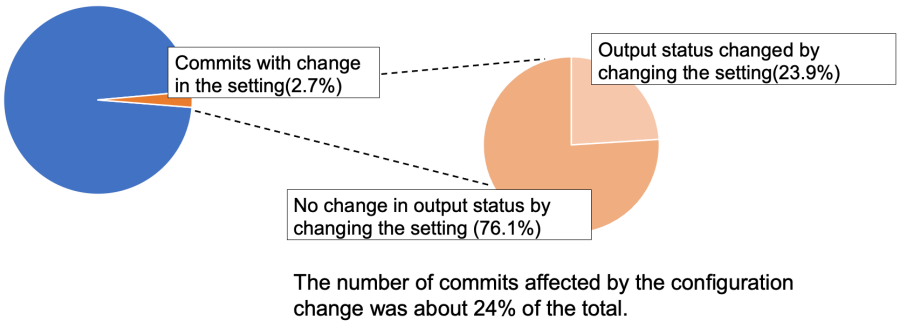

Figure 6: The number of commits with and without change in output status by logging configuration change

5.2 RQ2: Result

5.2.1 In the case of log statement unit

As Figure 7 shows, 3.7%(35) log statements has changed their output status by the change of source code log level. 96.3%(90) log statements has changed their output status by the change of configuration file threshold. From the results above, we can know that log output statements often change their output due to changes in the threshold in project configuration file.

5.2.2 In the case of commit unit

As Figure 8 shows, 34.3%(24) of the commits changed only the log level of the source code, 64.3%(45) of them changed only the configuration file threshold, and 1.4%(1) of them changed both of them when there are log statements which changed their output status in each commit. From the commit unit perspective, log output statements also often change their output due to changes in the threshold in project configuration file.

5.3 RQ3: Result

As Figure 9 shows, 19.2% log statements changed their log level in source code more than once. Their log levels are commuting without change in the threshold in project configuration file. The result shows that for about 20% logging statements, their log level are changed frequently which means developers change these log levels with the intention

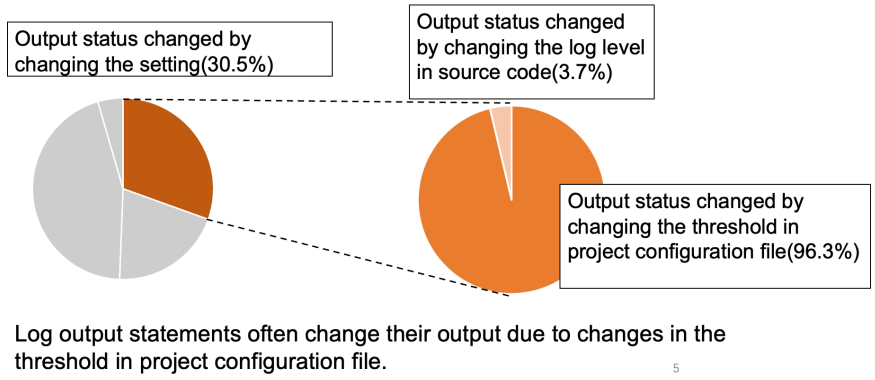

Figure 7: File changed when log statements output status change in the case of log statement unit

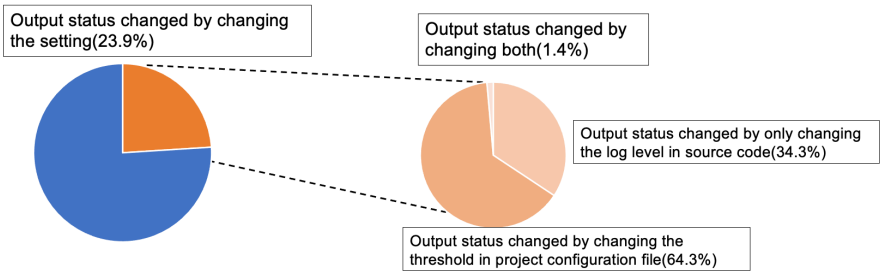

Figure 8: File changed when log statements output status change in each commit

on changing the output.

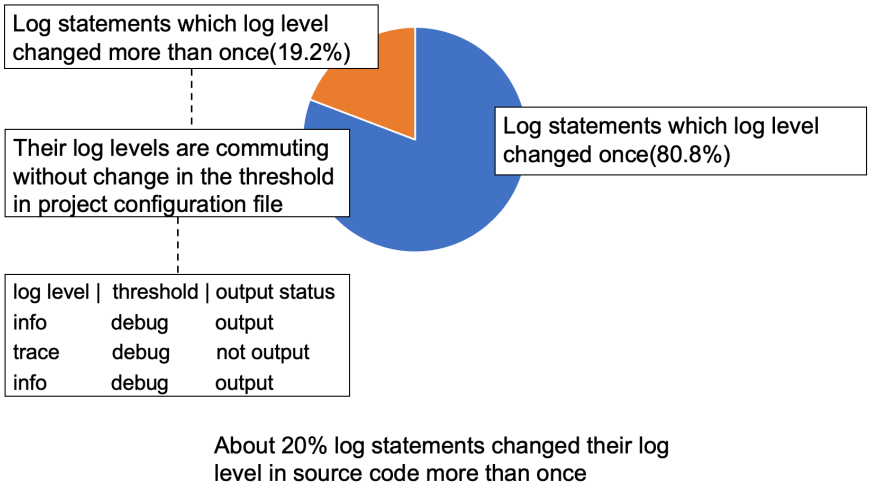

Figure 9: Log statements with once and more than once log level change in source code

6 Threats to Validity

External Validity Generality of our results are considered as the external threat to validity in this research. We choose only one open source project ActiveMQ as our study case in this research. But other projects may use use other kinds of logging libraries and different rules, this time the results may not apply to other projects. For example, other projects may have more than more than 6 gradations in their library. Further study in different domains and sizes project can benefit our study. Also, since we choose only one project written in java as our study case, we have not conducted research on projects in other languages, the results may not apply to the projects in other languages. Conducting this study on a larger number of items in a variety of other languages would make the results more generalizable.

Internal Validity In this paper, we used the thresholds in the project configuration file. However, whether these thresholds can be completely trusted is a remaining issue for us to figure out. For example, if there is case that the threshold in repository is "info" but changing to "debug" temporarily when developers want to debug. Further study on issue like this can make our research more credible.

Construct Validity This paper investigate the impact of logging configuration changes in source code and project configuration file. As result, project configuration file play a more important role in the change of log statements output status. Future work should conduct studies figuring out whether the project configuration file change by developer change the output status of log statements which are not supposed to be changed. This could give developer suggestions when they want to change the configuration file. We expect future work to expand this study.

7 Conclusion

Prior studies shows that developers are always struggling with choosing the an appropriate log level for each log statement and take great effort on maintaining and updating log level over the lifetime of a project. There are prior research on recommendation on choosing log level and revealed lots of findings about current practice of project configurations files. But no existing research focuses on the collective impact of log level in source code and project configuration file. In this paper, we conduct the study which focus on the log statements output status change. Some of the key findings of our study are as follows:

1) Log output statements affected by project configuration file changes were about 30% of the total in the case of log statement unit, and 24% in the case of commit unit.

2) The log statements output status are more affected by logging configuration change than by logging code change.

3) There are logs of cases that for one log statement, developers change its log level in source code or the threshold in project configuration file to change the log statements output status.

Acknowledgement

First and foremost, I would like to give my sincere gratitude to Professor Katsuro Inoue for giving me the opportunity to work with him. He provided advice and important direction to my thesis work.

Second, I would like to thank Associate Professor Makoto Matsushita in Department of Computer Science, Osaka University for his professional instructions and advice.

Also, I would like to present my thanks to Assistant Professor Tetsuya Kanda for great instructions throughout the process of writing the thesis, improving the outline and the argumentation, and correcting the grammatical errors. His insightful comments on every draft, which provide me with many enlightening ideas, have inspired me to a great extent

What is more, special thanks to Mr Kazumasa Shimari, my tutor who, with extraordinary patience and consistent encouragement, gave me great help by providing me with necessary materials, advice of great value and inspiration of new ideas. It is his suggestions that draw my attention to a number of deficiencies and make many things clearer. With out his strong support, this thesis could not been the present form.

I would like to express my gratitude to all members of Department of Computer Science for their guidance. Thanks are also due to many friends in Department of Computer Science, especially students in Inoue Laboratory.

References

[1] Apache. activemq. `https://github.com/apache/activemq`.

[2] Apache. Log4j. `https://logging.apache.org/log4j/2.x/`.

[3] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 24–33, 2014.

[4] Ceki Gulcu. The complete log4j manual. In *Quality Open Software*, 2003.

[5] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 145–155, 2012.

[6] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, p. 60–70, 2018.

[7] Heng Li, Weiyi Shang, and Ahmed E. Hassan. Which log level should developers choose for a new logging statement? *Empirical Software Engineering*, Vol. 22, No. 4, pp. 1684–1716, 2017.

[8] Zhenhao Li. Characterizing and detecting duplicate logging code smells. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 147–149, 2019.

[9] Tsuyoshi Mizouchi, Kazumasa Shimari, Takashi Ishio, and Katsuro Inoue. Padla: A dynamic log level adapter using online phase detection. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 135–138, 2019.

[10] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pp. 908–911, 2018.

[11] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, p. 117–132, 2009.

[12] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, p. 293–306, 2012.

[13] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 102–112, 2012.

[14] Sha Zhao, Julian Ramos, Jianrong Tao, Ziwen Jiang, Shijian Li, Zhaohui Wu, Gang Pan, and Anind K. Dey. Discovering different kinds of smartphone users through their application usage behaviors. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, p. 498–509, 2016.

[15] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. An exploratory study of logging configuration practice in java. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 459–469, 2019.