Studying the Use of Java Logging Utilities in the Wild

Boyuan Chen
York University
Toronto, Canada
chenfsd@cse.yorku.ca

Zhen Ming (Jack) Jiang
York University
Toronto, Canada
zmjiang@cse.yorku.ca

ABSTRACT

Software logging is widely used in practice. Logs have been used for a variety of purposes like debugging, monitoring, security compliance, and business analytics. Instead of directly invoking the standard output functions, developers usually prefer to use logging utilities (LUs) (e.g., `SLF4J`), which provide additional functionalities like thread-safety and verbosity level support, to instrument their source code. Many of the previous research works on software logging are focused on the log printing code. There are very few works studying the use of LUs, although new LUs are constantly being introduced by companies and researchers. In this paper, we conducted a large-scale empirical study on the use of Java LUs in the wild. We analyzed the use of 3, 856 LUs from 11, 194 projects in GitHub and found that many projects have complex usage patterns for LUs. For example, 75.8% of the *large*-sized projects have implemented their own LUs in their projects. More than 50% of these projects use at least three LUs. We conducted further qualitative studies to better understand and characterize the complex use of LUs. Our findings show that different LUs are used for a variety of reasons (e.g., internationalization of the log messages). Some projects develop their own LUs to satisfy project-specific logging needs (e.g., defining the logging format). Multiple uses of LUs in one project are pretty common for *large* and *very large*-sized projects mainly for context like enabling and configuring the logging behavior for the imported packages. Interviewing with 13 industrial developers showed that our findings are also generally true for industrial projects and are considered as very helpful for them to better configure and manage the logging behavior for their projects. The findings and the implications presented in this paper will be useful for developers and researchers who are interested in developing and maintaining LUs.

KEYWORDS

empirical software engineering, logging code, logging practices;

ACM Reference Format:
Boyuan Chen and Zhen Ming (Jack) Jiang. 2020. Studying the Use of Java Logging Utilities in the Wild. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380408

1 INTRODUCTION

Execution logs (a.k.a., logs) have been used widely in practice for a variety of purposes (e.g., system monitoring [53, 62], failure diagnosis [79, 80], and business analytics [6, 41]). Logs are generated during runtime by the output statements that developers insert into the source code. Instead of directly invoking the standard output functions like `System.out.print`, developers prefer to instrument their systems using logging utilities (LUs) (e.g., `SLF4J` [64] for Java and `spdlog` [65] for C++) due to additional functionalities like thread-safety (synchronized logging in multi-threaded systems), data archival configuration (automated rotation of the log files), and verbosity levels (controlling the amount of logs outputted).

Unlike many of the software engineering tasks (e.g., code refactoring [15] and release management [28]), there are no well-defined guidelines for software logging. Recently, there have been many research works devoted to the area of where-to-log (deciding the appropriate logging points) [11, 16, 80, 85, 87], what-to-log (providing sufficient execution context in the logging code) [25, 38, 82], and how-to-log (developing and maintaining high quality logging code) [9, 10, 39, 40, 81]. However, all of these works focus on improving the quality of log printing code (e.g., `LOG.info("User " + username + " authenticated")`). There are only two research works on the migration [33] and the configuration [86] of the LUs. It is important to study the use of LUs due to these three reasons:
Measuring the Adoptions of the LUs: Although there are already many LUs available (e.g., [1, 42, 43, 64]), new LUs are continuously introduced by companies (e.g., Flogger from Google [19]) and researchers (e.g., NanoLog [78] and Log++ [48]). It is not clear whether these LUs are adopted and used in the wild.
Understanding the Complex Use of the LUs: Incorporating multiple LUs in one project may cause various issues during compilation [45, 66], deployment [50, 68], and runtime [26, 27]). However, many software systems still use more than one LUs in their projects [76]. For example, `Hadoop`, which is a very well-maintained popular open source Big Data platform, contains not only six external LUs (`Apache Commons Logging`, `java.util.logging`, `Log4j 1.x`, `Log4j 2`, `SLF4J`, `Jetty logging`) and implements their own LU in their project. `IntelliJ Idea`, which is a very popular IDE, uses 12 LUs in their project. It is important to study this phenomena in order to suggest best practices for the system developers and to identify future directions for the LU designers and researchers.
Assessing the Impact of LUs on the Logging Code: On one hand, the structure of logging code is a result of adopting certain LUs. For example, it is generally recommended to specify the logging needs as rules via Aspect Oriented-Programming (AOP) constructs to improve system modularity [35] and use centralized logging [30] to support internationalization (a.k.a., outputting the log messages in different human lanaguages). On the other hand, extra care are needed in order to cope with the complex use of LUs

in one project. For example, many projects nowadays reuse existing functionalities by importing third-party packages, which also use LUs. However, there is no empirical study to assess the impact of the logging code due to the use of the LUs.

In this paper, we have performed a large-scale empirical study on the use of Java LUs in the wild. We focus on Java, because it is currently ranked as the most popular programming language in the world based on the TIOBE index [73]. Many popular software systems (e.g., Android-based mobile applications [54, 69], IDEs [13, 29], web servers [3, 59], and Big Data platforms [2, 24]) are implemented in Java and logging is prelevant in these systems [8, 83]. We have examined 11,194 open source Java-based projects in GitHub, which uses 3,856 LUs. We have uncovered four important findings and implications. To ensure the usefulness and generalizability of our findings from open source systems, we also interviewed 13 industrial developers on the use of LUs. We summarize our main contributions and findings as follows:

- This is the first empirical study on the complex use of LUs in the wild.
- 3,856 LUs are currently being used by 11,194 projects in GitHub. The number of used LUs increases as the systems grow bigger, as larger-sized projects usually reuse existing functionalities by importing third-party packages, which also use LUs. Many projects also implement their own LUs to satisfy project-specific needs. Our findings raise developers' awareness on the complexity of the LUs in their systems as well as the need to manage these LUs to better monitor and debug systems' runtime behavior.
- In addition to the quantitative study, we have also conducted a qualitative study to understand and characterize the rationales behind the complex use of LUs for different projects. Such results can guide researchers to propose more general logging solutions for various logging context.
- To support independent verification or further research on the use of Java LUs, we have provided a replication package [70] in this paper. This package can be useful for other researchers who are interested in studying and improving the logging practices.

Paper Organization. The rest of this paper is organized as follows. Section 2 provides an overview of our approach and describes our studied projects. Section 3 and 4 quantitatively and qualitatively study the use of the LUs in the wild. Section 5 explains our interview process and describes our observations. Section 6 discusses the significance of our findings and presents some future work. Section 7 describes the related works. Section 8 explains the threats to validity and Section 9 concludes the paper.

2 OVERVIEW

In this section, we will provide an overview of our approach to empirically studying the use of Java LUs in the wild and describe our studied projects.

2.1 Our Approach

We followed a mixed-methods approach characterized by a sequential explanatory strategy [12] to analyze the use of LUs. We first extracted the source code from popular Java-based GitHub projects (Section 2). Then we performed a quantitative study on measuring the degree of adoption of different LUs as well as comparing the number of used LUs across different projects (Section 3). We also tracked the number of projects which also implement their own LUs. Afterwards, we performed a qualitative study (Section 4) by manually studying the use of LUs among different projects. Since our study is performed on open source projects, we also cross-validated our findings by interviewing 13 experienced developers who work on commerical systems (Section 5).

2.2 Studied Projects

To study the use of Java LUs in the wild, we focused on analyzing Java-based projects from GitHub. GitHub is currently the largest code hosting site with more than 100 million projects as of April 2019 [17]. We built a local GHTorrent [20] database from the MySQL dump. This database, which was last updated on 2019-06-01, contains the meta information of a project such as the corresponding GitHub URL, the project name, and the main programming language(s). We extracted a list of GitHub URLs for all the Java-based projects for post-processing.

We further filtered the list of GitHub projects to avoid potential perils in our analysis [34]. One GitHub project may be forked or cloned by others, whose source code can be identical or very similar to the orignal one. This would introduce noise into our study. Hence, we only selected projects which are neither a fork nor a clone of other projects. Furthermore, the number of stars that a project has indicates its popularity. Similar to prior works [5, 36, 58], we further filterd the list of projects to ensure they have at least 30 or more stars. We ended up with 25, 611 Java projects. We downloaded the soure code for the most recent releases of these projects by invoking the GitHub APIs.

3 QUANTITATIVE STUDY

In this section, we first analyzed the selected projects to identify the list of LUs that are used. Then we measured the degree of LUs which are adopted and used. Finally, we compared the use of LUs among different projects.

3.1 Identifying LUs in Each Project

We developed a heuristic-based technique to identify LUs in each project. First, we excluded Java files, which were either in a `test` folder or containing the keyword of "test" in their file names, as they are probably not related to the core features of the projects under study. Then we used JDT [31] to parse the remaining Java files to identify a list of imported statements and function invocations for each Java class. We made sure these imported classes are used by checking if there is one or more methods been invoked in that class. We further filtered the list of import statements in each Java class, so that only packages or Java classes whose names contain patterns like "logging", "logger" or ".log." were kept. These packages or classes are considered as LUs. For logging in Aspect-Oriented Programming (AOP), we first identified Java files with import statements that contain "aspectj". Since `AspectJ` can do more than logging, we subsequently parsed the Java files to check whether they use any of the LUs identified by the above rules.

If a project used more than one LUs and their package names were similar, we merged them as one LU. For example, if the following two LUs, `Foo.Bar.Baz.ConsoleLogger` and `Foo.Bar.Baz.FileLogger`, were identified in one project, we merged them into one LU (`Foo.Bar.Baz`).

To verify the correctness of our heuristic-based technique, we randomly sampled 441 files and manually examined the identified LUs. Our technique yield an precision of 96%. Some of the Java classes are misclassified as LUs as the identified log-related Java classes did not implement log printing functions. For example, `LoggerProvider.java` in the `Ninja` framework [46] was not considered as an LU, since this class did not provide any log printing functions other than a utility function that returns an `SLF4J` logging object.

3.2 Measuring the Adoptions of LUs

We further classified the size of these projects into the following five bins based on their number of Java classes using the definitions from [47]. Table 1 shows the results. For example, there are a total of 761 *large*-sized projects, whose number of Java classes is between 1,000 and 5,000. Among them, 728 projects (a.k.a., 95.7%) adopt one or more LUs. There is a clear trend of increasing adoptions of LUs as the size of the projects get bigger.

For the subsequent studies, we removed all the projects which did not use any LUs, since they are not relevant to this paper. We also excluded the projects whose size was *very small*, as many of them are not considered as useful projects (e.g., trial projects for self-studying, and collections of code snippets). After filtering, there were 11,194 Java projects remaining, which used 3,856 distinct LUs.

Table 1: Measuring the Adoptions of LUs Among the Java-based GitHub projects.

Bins (# of classes)		Total # of Projects	% of Projects Used LUs
Very Small	[0, 20)	11,390	43.2%
Small	[20, 100)	7,781	72.5%
Medium	[100, 1000)	5,576	84.3%
Large	[1000, 5000)	761	95.7%
Very Large	[5000, ∞)	103	97.1%
Total		25,611	63.1%

3.3 Comparing the Use of LUs Among Projects

Depending on where these LUs were implemented, we further classified the list of used LUs in each project as *External LU (ELU)*s or *Internal LU (ILU)*s. It is an ELU, if the implementation of this LU is not inside the project under study. Otherwise, it is an ILU. For example, `Hadoop` uses five different ELUs: `Apache Commons Logging`, `java.util.logging`, `Log4j 1.x`, `Log4j 2`, `SLF4J`, `Jetty logging` and implements its own ILU, which is mainly used for auditing purposes. Table 2 shows the results of the percentage of projects that use ELUs, ILUs, or both. For example, among all the 728 Large-sized projects that used LUs, 95.3% of them are ELUs, 75.8% of them implemented their own ILUs in their projects, and

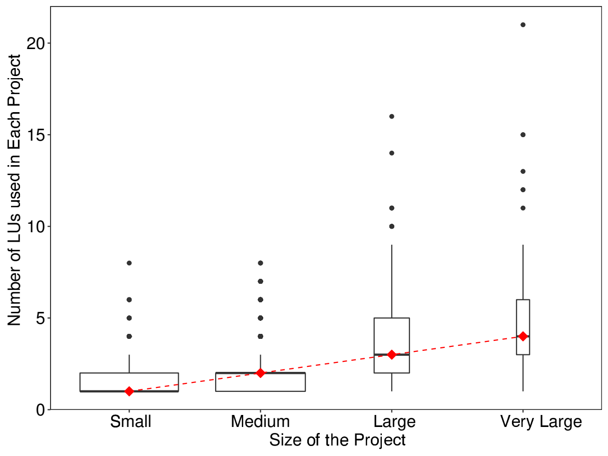

Figure 1: The Distritbutions of the Number of Adopted LUs in Each Project Grouped by the Size of the Projects.

71.2% used both types of LUs. Overall, there are 866 ELUs used by 11,194 projects. 26.7% of the studied projects have implemented their own ILUs.

The percentage of projects that adopted ELUs and ILUs shown in the second and the third columns increases as the size of projects increases. When comparing the LUs within the same bins, there are always more projects using ELUs than ILUs. However, almost all of the *very large*-sized projects implemented their own LUs. The combined use of ILUs and ELUs also increase dramatically as project sizes become *large* or *very large*.

Table 2: Comparing the Complexity of the Uses of LUs Among Projects.

Project Size	% of Projects Using (*ELU*	*ILU*	*Both*)
Small	(97.2%	12.7%	9.9%)
Medium	(95.3%	34.6%	29.9%)
Large	(95.3%	75.8%	71.2%)
Very Large	(97.0%	92.0%	89.0%)
Total	(96.3%	26.7%	23.0%)

We further examined the number of LUs that were used in each project. For each project, we counted the number of LUs, which included both ELUs and ILUs. We grouped projects based on their sizes and visualized their distributions using boxplots in Figure 1. The width of the boxplot is proportional to the number of projects in that group. Since there are much more *small*-sized projects in our study, it is the widest. The red dashed line connects the median points for each bin. The median usage for the *small*-sized projects is one. It increases as the size of the project increases. For the *very large*-sized projects, the median number of LUs used is four. The project that adopts the biggest number of LUs is within the group of *very large*-sized projects. It is an enterprise web platform ([57]), in which 21 LUs are used!

We conducted a Kruskal-Wallis test [37] to statistically check if their distributions are identical. The *p-value* was smaller than 0.05, which indicated statistically signficant differences among the distributions of LUs across different project sizes. This clearly demonstrates the complexity of the use of LUs in our studied projects.

The findings in our quantitative studies motivated us to conduct the subsequent qualitative studies (Section 4) to figure out the rationales behind them.

Findings: (1) There are 3, 856 LUs, of which 866 are ELUs, being used by over 11, 000 projects. (2) 96.3% of the studied projects adopt at least one ELU and 26.7% adopt at least one ILU. (3) As the project size becomes larger, more LUs will be integrated into the project. This is especially the case for *large* and *very large*-sized projects.
Implications: The LUs are used widely among different Java projects. However, as the size of the project increases, the complexitity of the use of LUs also increases. Multiple LUs are used in many of the *medium* to *very large*-sized projects. This is contradictive to the common understanding of developers [71, 74, 76, 77] and researchers [33], as only a handful of popular Java LUs (e.g., `Log4j 1.x`, `Log4j 2`, and `SLF4J`) are compared and discussed.

4 QUALITATIVE STUDY

In this section, we conducted a qualitative study on the use of the LUs. We manually examined their source code in order to answer the following RQs:

- **RQ1: What are the external LUs being used in the wild?** As shown in Section 3, there were 866 ELUs being used by 11,194 projects. The goal of this RQ is to characterize the rationales of different ELUs.
- **RQ2: Why do developers implement ILUs in their projects?** 26.7% of the studied projects have implemented their own internal LUs. This is especially the case for the *large* or *very large*-sized projects, in which 75.8% and 92.0% of them contain ILUs. The goal of this RQ is to extract the project-specific logging needs by studying the use of ILUs.
- **RQ3: How are multiple LUs used in the wild?** Many of the studied projects, especially for the *large* or *very large*-sized projects, use multiple LUs. The goal of this RQ is to characterize the usage context associated with using multiple LUs in one project.

4.1 RQ1: What are the external LUs being used in the wild?

In this RQ, we will characterize the rationales behind the use of individual ELUs. For each ELU, we calculated the number of projects that adopted it. We sorted ELUs by their popularities (a.k.a., the number of projects that used them) and selected the top-100 most popular ELUs, which are used by 95% of the studied projects.

For each of these 100 ELUs, we manually examined the online documentation, release notes, and blog posts for these ELUs to understand their features and capabilities. Then we studied the source code of the projects which used them to extract their usage context. Our findings are summarized in Figure 2. Generally, there are four reasons behind the use of ELUs: (1) General-purpose logging, (2) LU interactions, (3) Internationalization, and (4) Modularization.

4.1.1 General-purpose Logging. *General-purpose logging* refers to the most common logging usage context, which requires: (1) *verbosity levels*, which are to control the amount of log message outputted; (2) *configurations*, which are used to specify various options and policies to format and output the logs; and (3) *thread safety*, which is to ensure the logs are recorded in sequence for a multithreaded system.

Among the top-10 most popular LUs, 8 are for general-purpose logging. Six ELUs (`SLF4J`, `Log4j 1.x`, `Log4j 2`, `java.util.logging`, `Apache Commons Logging`, and `LogBack`) are considered as general-purpose LUs for desktop or server-based systems and two ELUs (`android.util.Log` and Timber [72]) are general-purpose LUs for Android-based systems. As shown in Figure 2, a log printing function from a general-purpose ELU generally consists of four parts: the logging object (`LOG`), the verbosity level (`debug`), the static texts describing the logging context (`"parsing File"`), and the dynamic contents revealing the runtime information (`file`).

Among all the desktop or server-based ELUs, `SLF4J` is the most popular ELU due to its improved performance and compatibility with other LUs [71, 74, 76]. There are many Android projects (4, 477) in our study. Most of them are *small*-sized projects that used the default ELU from Android SDK (i.e., `android.util.Log`).

4.1.2 LU Interactions. *LU interactions* concerns about configurating and controlling the logging behavior for the imported packages. Many Java-based projects are built on top of existing packages, many of which contain ILUs. In order to configure and control the logging behavior for one imported package, developers have to invoke the logging APIs from this package. For example, 161 projects use the `okhttp3` [52] to send and receive data from network. In order to enable logging for the `okhttp3` package, developers of the `bitcoin-wallet` project [7], which is a Bitcoin payment app for Android, invokes the `addInterceptor` method from `okhttp3`'s ILU as shown in Figure 2. 83 out of the top-100 ELUs in this study are used for this reason. Hence, *LU interactions* is the main reason why there are so many ELUs used in the wild.

4.1.3 Internationalization. *Internationalization* concerns about adapting the logs to other human languages (e.g., German or Chinese). In order to support internationalization, developers first need to create a centralized file, which stores a list of pre-defined log message templates. Each of the log message template comprises four parts: the verbosity level, the parameterized string, the logging methods, and the message ID.

The third row of Figure 2 shows one such example. This code snipeet is from `Hibernate-ORM`, which is a popular Object-Relational Mapping (ORM) framework. Inside the centralized file, `CoreMessage Logger.java`, there are two annotations (`@LogMessage` and `@Message`) for each logging method and the message ID. `@LogMessage` contains a key/value pair, which defines the verbosity level (`level = WARN`) of this log message. `@Message` provides the parameterized string, which contains the static texts (`I/O reported cached file could not be found`) and placeholders for parameters `%s`. The message ID is an integer, that can be used to uniquely identify this log message. The logging method which implements this log message is `cachedFileNotFound`. It takes in two parameters: the file path and the error message. In order to output this log message, the `cachedFileNotFound` method needs to be invoked. The code

Rationales	% of Top-100 ELUs	% of Projects	Top 2 LUs (where applicable)	Code Examples
General-Purpose	12%	89.9%	android.util.Log (1) SLF4J(2)	`static final Logger LOG = LoggerFactory.getLogger(Configuration.class);` `...` `LOG.debug("parsing File " + file);` Coniguration.java (Hadoop, using SLF4J)
LU Interactions	83%	14.3%	okhttp3.logging.HttpLoggingInterceptor (8) io.netty.handler.logging.LoggingHandler (11)	`HttpLoggingInterceptor loggingInterceptor = new HttpLoggingInterceptor();` `...` `final OkHttpClient.Builder httpClientBuilder = new OkHttpClient.Builder();` `...` `httpClientBuilder.addInterceptor(loggingInterceptor);` Constants.java (bitcoin-wallet, using okhttp3.logging.HttpLoggingInterceptor)
Internationalization	4%	1.0%	JBoss Logging (15) org.glassfish.jersey.logging (29)	`@LogMessage(level = WARN)` `@Message(value = "I/O reported cached file could not be found : %s : %s", id = 23)` `void cachedFileNotFound(String path, FileNotFoundException error);` CoreMessageLogger.java (Hibernate-ORM, using JBoss Logging) `...` `catch ( FileNotFoundException e ) {` `log.cachedFileNotFound( serFile.getName(), e );` `}` CacheableFileXmlSource.java (Hibernate-ORM, using JBoss Logging)
Modularization	1%	1.7%	AOP Logging (10)	`@Around("execution(* org.unitedinternet.cosmo.service.ContentService.getRootItem(..)) &&" + "args(user)")` `public Object checkGetRootItem(ProceedingJoinPoint pjp, User user)` `throws Throwable {` `LOG.debug("in checkGetRootItem(user)");` `...` SecurityAdvice.java (cosmo using AOP Logging) `public class StandardContentService implements ContentService {` `...` `public HomeCollectionItem getRootItem(User user) ...` StandardContentService.java (cosmo using AOP Logging)

Figure 2: The Rationales Behind the Use of Top-100 Most Used ELUs.

snippet in `CacheableFileXmlSource.java` shows one example of how this log message can be invoked during runtime. It is used within a catch block to handle an exception.

To translate the log messages into different human languages, developers need to create a translation property file and define internationalized labels for each log message. For example, the property file `log_en_FR.properties` would contain all the French labels for all of the above English log messages.

The most common ELU for *Internationalization* is JBoss Logging [30]. Although there are three other ELUs, which also support this usage context, they are just wrappers of the JBoss Logging.

4.1.4 Modularization. Modularization concerns about improving the modularity of the logging code. Logging code is a cross-cutting concern, as it inter-mixes with the feature code. Only one ELU, AOP-based logging, is used for this reason. AOP is a programming paradigm, which is designed to improve modularity by reducing the amount of cross-cutting concerns [35]. Logging is considered as one of its common use cases.

In order to perform AOP-based logging, developers need to provide rules through aspect files. A typical aspect file consists of pointcuts and advice. A pointcut is to define the point of execution where the cross-cutting concern (e.g. logging) needs to be applied. An advice is the additional code (e.g. logging code) instrumented.

Figure 2 shows a real-world example from `cosmo`, a calendar server that implements CalDAV protocol. The file `SecurityAdvice.java` is the aspect file. The instrumented points (pointcuts) are defined through the annotation. In this example, the annotation `@Around` means both the beginning and the end of the methods will be instrumented. The value within the brackets specify the instrumented methods. In this example, any methods with the name `getRootItem` within package `org.unitedinternet.cosmo.service.ContentService` and parameter type as `user` will be instrumented. The instrumented code (advice) is defined via method `checkGetRootItem`. It contains a log printing statement with the message `in checkGetRootItem` and the user name. The code snippet in `StandardContentService.java` shows how a log message is actually generated. This class implements the interface `ContenService`, which is within the specified package. It contains a method `getRootItem`, of which the parameter is `user`. Therefore, this method qualifies the pre-defined instrumented rule above. Hence, during runtime, the above logging statment will be executed while entering and exiting this method. In general, only 2% of our studied projects adopted AOP-based logging. This matches with previous empirical studies [4, 51], which indicated the very limited use of AOP in the wild.

Findings: There are four main reasons behind the use of ELUs: (1) General-purpose logging, (2) LU interactions, (3) Internationalization, and (4) Modularization. Majority of the projects use ELU for *general-purpose logging*. 83 out of the top-100 mostly used ELUs are used for *LU interactions*. Some ELUs provide unique features like *internationalization* and *modularization*, but they are only used by a very small number of projects.

Implications: Many ELUs are used, as developers need to enable and configure the logging behavior for the imported packages. It is important to manage these LUs to better monitor and debug systems' runtime behavior. An existing study [23] shows that log configurations are one of the main source of errors for Java-based projects. Unfortunately, only one tool [86] is developed to detect invalid loggers in the configuration files. To effectively monitor and debug systems' behavior, developers need to figure out how to configure the logging behavior for the imported packages and how to interact these LUs with the LUs in their own projects. Hence, techniques to recover the logging architecture and conduct automatic configuration are needed.

4.2 RQ2: Why do developers implement ILUs in their projects?

It is not clear why many projects have still implemented their ILUs even there are 866 ELUs available in the wild. To investigate the rationales behind this, out of 2, 990 projects which have implemented ILUs, we randomly selected 341 of them for close manual examination. This corresponds to a confidence level of 95% with a confidence interval of ±5%. We used the stratified sampling technique [21] to ensure representative samples are selected from projects of different sizes. The portion of the sampled projects within different sized projects is equal to the relative weight of the total number of the projects with that size. For example, there are a total of 721 *small*-sized projects which contain ILUs. Hence 82 ($\frac{721}{2990} \times 341$) *small*-sized projects are selected.

For each selected project, we carefully studied how their ILUs are being used by reading through the source code. We also examined the relevant commit logs, issue reports, and pull requests [22] to look for the rationales on why ILUs are implemented. In the end, we have identified three project-specific logging needs as shown in Figure 3: (1) Defining the logging format, (2) Compatibility with other LUs, and (3) Ease of configuration and dependency management.

4.2.1 Defining the Logging Format. Log messages are generally loosely structured, which contain free formed texts. Many projects define their own format of the log messages, so that they can be automatically parsed and analyzed. For example, the `Hadoop` project introduces audit logging in order to satisfy the security compliance requirements. The format of the auditing logs vary across different `Hadoop` components. For example, in `hadoop-common` component shown in Figure 3, an interface (`KMSAuditLogger`) is first defined with a method `logAuditEvent`, which defines the auditing methods to be invoked. To facilitate code reuse, `SimpleKMSAuditLogger` implements this interface by using the adapter pattern. It implements the `logAuditEvent` method by invoking the `info` method from an `SLF4J` logger object. The `logAuditEvent` method contains a switch statement, in which depending on the actual event, different audit logs will be outputted. The resulting audit logs are much more structured compared to the regular loosely defined log messages. Other ILUs like `Cassandra`'s `StatusLogger` also fall into this case.

4.2.2 Compatibility with other LUs. Many of the studied projects can be packaged and used by other projects. For reusable packages, it is preferred to be flexible and compatible with different ELUs, as developers always want to use the most up-to-date LUs in their projects. Since LU migrations require high manual efforts and are error-prone [33], developers usually implement their ILUs to separate the coupling of their logging code with the LUs.

`Vert.x` [75] is a popular tool-kit for building reactive applications on the JVM. It receives more than 10,000 stars on GitHub. It suppports four general purpose Java ELUs: `java.util.logging`, `Log4j 1.x`, `Log4j 2`, and `SLF4J`. This functionality is realized by implementing the strategy design pattern to unify the APIs among these four ELUs. These four ELUs do not share the same set of verbosity levels. Therefore, the ILU needs to provide a unified interface (`LogDelegate`) for their logging methods. The `LogDelegate` interface defines a set of common logging APIs (e.g., `info`, `error`). Separate implementation classes are introduced to wrap around the four popular general purpose ELUs: `java.util.logging`, `Log4j 1.x`, `Log4j 2`, and `SLF4J`. Figure 3 shows a code snippet for `JULLogDelegate`, which adapts the functionalities of `java.util.logging` to the common interface defined by `LogDelegate`. To implement the `error` method, it calls the `log` method along with the verbosity level `SEVERE` and the variable `message` from `java.util.logging`.

4.2.3 Ease of Configuration and Dependency Management. One of the common errors associated with logging is the configuration of LUs [23]. The main cause of this is due to complex dependency structures [26, 27, 45, 50, 66, 68]. Hence, about 29.3% of the ILUs are implemented to ease the configuration and dependency management issues. They are usually built from the ground up using only the standard JDK libraries and are not dependent on any ELUs to minimize the effort to manage dependencies. Figure 3 shows one example. This code snippet is from `Slick2D`, which is a 2D Java game library. This ILU is designed to be lightweighted and easy to use. It logs the complete error information with timestamps to the console.

Findings: There are three project specific needs, which lead to the implementation of ILUs: (1) defining the logging format, (2) compatibility with other LUs, and (3) ease of configuration and dependency management. Among these three needs, *defining the logging format* is the most common one. 55.3% of the sampled projects implemented ILUs for this specific need. As the size of the projects grow larger, more projects implement ILUs to satisfy the needs of *compatability with other LUs*.

Implications: Although more than 20% of the sampled projects implemented their own ILUs from the ground up. These ILUs are mainly used internally for debugging purposes. They are usually not intended to be used by other projects, since they might not satisfy the general logging needs (e.g., not thread-safe). Additional tools and techniques need to be developed to automatically warn developers who imported packages which implement ILUs.

Rationales	% of Projects	Code Examples

Defining the Logging Format	Small (64.6%)
	Medium (53.5%)
	Large (47.6%)
	Very Large (72.7%)
		`import org.slf4j.Logger;`
		`import org.slf4j.LoggerFactory;`
		`...`
		`class SimpleKMSAuditLogger implements KMSAuditLogger {`
		`  final private Logger auditLog = LoggerFactory.getLogger(KMS_LOGGER_NAME);`
		`  public void logAuditEvent(final OpStatus status, final AuditEvent event) {`
		`    switch (status) {`
		`      case OK:`
		`      auditLog.info("{}[op={}, key={}, user={}, accessCount={}, interval={}ms] {}",`
		`      status, event.getOp(), event.getKeyName(), event.getUser(),`
		`      event.getAccessCount().get(),`
		`      (event.getEndTime() - event.getStartTime()), event.getExtraMsg());`
		`      break;`
		SimpleKMSAuditLogger.java (Hadoop)

Compatibility with other LUs	Small (7.3%)
	Medium (18.4%)
	Large (25.4%)
	Very Large (27.3%)
		`public interface LogDelegate {`
		`...`
		`  void error(Object message);`
		`}`
		LogDelegate.java (vert.x)
		`public class JULLogDelegate implements LogDelegate {`
		`  private final java.util.logging.Logger logger;`
		`  JULLogDelegate(final String name) {`
		`    logger = java.util.logging.Logger.getLogger(name);`
		`  }`
		`  ...`
		`  public void error(final Object message) {`
		`    log(Level.SEVERE, message);`
		`  }`
		JULLogDelegate.java (vert.x)

Ease of Configuration and Dependency Management	Small (24.4%)
	Medium (30.3%)
	Large (33.3%)
	Very Large (27.3%)
		`public class DefaultLogSystem implements LogSystem {`
		`  public static PrintStream out = System.out;`
		`  ..`
		`  public void error(Throwable e) {`
		`    out.println(new Date()+" ERROR:" +e.getMessage());`
		`    e.printStackTrace(out);`
		`  }`
		`  ...`
		DefaultLogSystem.java (Slick2D)

Figure 3: The Rationales Behind why ILUs are Implemented.

4.3 RQ3: How are multiple LUs used in the wild?

The goal of this RQ is to understand and characterize the uses of multiple LUs in one project. We sorted our studied projects by the number of LUs that are used. Then we selected the top-100 projects, which used the most LUs, for close manual examinations. The number of LUs for these projects ranges from 7 (Hadoop) to 21 (`Liferay-portal`). Figure 4 summarizes our findings. There are generally four usage context behind the use of multiple LUs: (1) Interaction with LUs from the imported packages; (2) Managing the logging contents; (3) Formatting logging messages across different components; and (4) Developer convenience. As the project sizes grow bigger, the usage context of multiple LUs also become more complex. Note that there is no *small*-sized projects among them, as they generally use much fewer LUs.

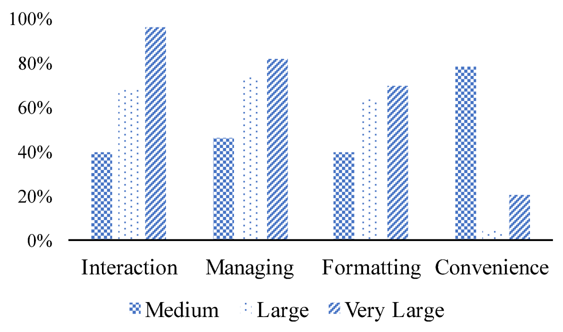

Figure 4: The Usage Context Behind the Top-100 Projects, which Contain the most LUs.

```
pipeline.addLast("logger", new LoggingHandler(LogLevel.INFO));
```

(a) InboundConnectionInitiator.java(Cassandra, using Netty LoggingHandler)

```
KieRuntimeLogger logger=KieServices.Factory.get().
      getLoggers().newFileLogger(session,"log/correlation");
```

(b) CorrelationExample.java (OpenNMS, using KieRuntimeLogger)

Figure 5: An Example of using Multiple LUs for *interaction with LUs from the imported packages.*

```
import org.apache.maven.plugin.logging.Log;
...
public File resolveById(String id, Log log)
                     throws MojoFailureException {
...
  log.debug("Resolving artifact " + id + " from "
                               + projectRepositories);
```

Dependency31Helper.java (karaf)

Figure 6: An Example of using Multiple LUs for *managing the logging contents.*

4.3.1 Interaction with LUs from the imported packages. Many Java-based software systems use third party packages, which also contain LUs. In order to have full observability of the resulting systems, it is important to enable logging across all the components. Figure 5 shows two different examples on how to configure or enable logging for the imported packages. However, due to the API variabilitiy of these LUs, different techniques are needed in order to enable or configure their logging behavior. For example, `Netty` uses event-based programming. To enable the logging of the `Netty` package, the `Cassandra` developer needs to add the `LoggingHandler`. In order to enable logging for `Kie`, the `OpenNMS` developer needs to create a new logger to output the logs to a file. The more third party packages are used, the more interactions there are with the LUs from these imported packages. This is one of the main reasons why many *large* or *very Large*-sized projects use more LUs than the smaller sized projects.

4.3.2 Managing the logging contents. In addition to configuring and enabling the logging behavior for the imported packages, sometimes the studied projects also use the LUs from the imported packages for additional logging. This is mainly to ensure the relevant logging contents are stored in the same location. `Maven` is a popular tool to build and manage Java projects. The core of `Maven` consists of a set of plugins. Each plugin is reponsible for a particular functionality. For example, `clean`, which is used to clean up the build artificats, and `compiler`, which is used to compile Java sources, are two default plugins. During execution, the logs generated from these plugins will be redirected to the same storage location. If other projects intend to develop `Maven` plugins, they are advised to use the LUs from `Maven` to generate build related logs [49], so that `Maven`-related information can be aggregated to one centralized location, which is easy to analyze and archive. Figure 6 shows one such example. On one hand, `karaf` uses their own LU to record system-specific information. On other hand, it uses the LU from `Maven` to record build related logs.

4.3.3 Formatting logging messages across different components. In some cases, one LU may not satisfy all the logging needs for one project. For example, as shown in Figure 3, `Hadoop` uses `SLF4J` for generating execution logs, which are used for debugging and monitoring purposes. It also uses ILU to generate audit logs to satisfy security requirements.

```
for (InstanceStatus status : instanceStatusList) {
  if (INSTANCE_RUNNING !=
        status.getInstanceState().getCode()){
    LOGGER.debug("Instances are up but not
                all of them are in running state.");
    return false;
  }
 }
```

ASGroupStatusCheckerTask.java (cloudbreak)

Figure 7: An Example of using Multiple LUs for *developer convenience.*

4.3.4 Developer convenience. Some of the top-100 projects use AOP-based logging. However, the developers also include logging code, which is instrumented using general-purpose ELUs. This is mainly due to developer convenience. For example, `cloudbreak` uses AOP-based logging, but it also uses SLF4J. Figure 7 shows one such example. Since checking the instance state is a very localized concern, it is much faster to instrument with the general purpose LU than AOP-based logging. Similar cases also apply to LUs, which are used for internationalization. For example, `Hibernate` uses the JBoss logging to support internationalization. However, instead of putting the log messages into the centralized file, the developer chose to place their log message along with the feature code for debugging purposes.

Findings: There are four usage contexts behind the use of multiple LUs in one project: (1) interaction with LUs from the imported packages; (2) managing the logging contents; (3) formatting logging messages across different components; and (4) developer convenience. Except for *developer convenience*, the percentage of these usage contexts increases as the project sizes increase.
Implications: Logging is considered as a cross-cutting concern, as the logging code scatters across the entire system and intermixes with the feature code. To cope with this challenge, AOP is introduced. However, AOP cannot be used to satisfy the current logging needs, due to their inconvenience on specifying localized logging context. More importantly, for large-scale projects which use many third party packages, it is necessary to enable and configure the logging behavior for these imported packages in order to gain full observability of the entire systems. The management of the logging behavior for these packages is rather complex and introduces another form of cross-cutting concerns. Further research is urgently needed to develop tools or techniques to automatically manage and modularize such concerns.

5 INTERVIEWS

Although we have analyzed 425 Java-based projects, our study focuses on Java-based open-source projects in GitHub. In order to assess the generalizability of our findings, we conducted a semi-structured inteview with 13 experienced industrial developers. We decided to conduct semi-structured interviews instead of surveys so that we can interact with the participants in a more flexible way. For example, we have prepared a set of questions before-hand. During the interview, we sometime asked follow-up questions based on participants' answers on the fly [32].

5.1 Participants

Similar to [60], the participants are from the authors' personal contacts. All of them are working at large-scale software companies. The development experience of the participants ranges from one to eight years. The types of projects that participants are working on vary from server-side projects (9), frameworks (3), and client application (1). All these projects have been widely used by millions of users worldwide. The programming languages that the participants use daily include Java, PHP, C++, C#, Python, and Go.

5.2 Findings

All of the participants have inserted, deleted, updated logging code in their development activities. It reaffirms that software logging is a pervasive practice [8].

5.2.1 Cross-validation of our findings. We presented our findings on the rationales behind the use of ELUs and ILUs, as well as the usage context for multiple LUs. Then we asked the participants: (1) whether their projects adopt one or more of the studied LUs; (2) whether our characterized usage context and the logging needs would be useful for them; and (3) if there are any additional information to add or comment on.

Some participants mentioned that they did adopt one or more of the studied LUs in this paper. However, the rationales (e.g., JBoss-style logging) is a bit different. All participants felt that the findings and code examples from this study can help them better configure and manage the logging behavior for their projects.

- **ELUs**: All participants acknowledged that the general-purpose logging is the most commonly used logging need. However, they also acknowledge the challenges on co-evolving the logging code with the rapidly changed feature code, as the logging code is inter-mixed with the feature code. Although the participants aggreed that AOP-based logging is a great idea, only two of the interviewed participants used AOP-based logging in their projects. This is mainly due to (1) high learning curve of a different programming paradigm, (2) difficult to translate logging concerns into rules (a.k.a., advice), and (3) lack of automated support for legacy logging code. Compared to open-source projects, there is a much higher portion of industrial projects (7/13) that adopted the LUs that centralize the log messages (a.k.a., JBoss-style logging). One participant mentioned that in addition to internationalization, the centralized logging-style can: (1) improve the accuracy and the speed of the log processing task [88]; and (2) attach additional meta information (e.g., issue resolution strategies) with the message ID for better field support.
- **ILUs**: The participants mentioned that the choice of LUs were decided by the software architects or senior developers. Once the LUs were set up, only under very rare cases that they will migrate or modify LUs. None of them have directly modified the existing LUs. We asked the participants what type of LUs were used in their projects. Three of them said they directly use open source LUs such as `Log4j` [42] (Java) and `log4net` [44] (C#). The rest of them use ILUs included in their project. The rationales behind implementing ILUs are similar to our findings from the open source projects.
- **The use of Multiple LUs**: The participants are surprised about the use of multiple LUs. Five of them did not realize the need to do this until we presented our findings. They complained about the challenges on debugging and monitoring their projects, which use many third-party packages. By leveraging our findings and code examples, they can enable and manage the logging behavior from these imported packages, which will greatly improve the observability of the overall systems.

5.2.2 Beyond Logging. Five participants mentioned that they have integrated tracing tools (e.g., opentracing [55]) or Application Performance Monitoring (APM) tools (sentry [61], Google firebase [14], etc.) into their systems. They applied these tools to replace some of the logging functionalities such as crash reporting and collection of the profiling data. Unlike logging, most of these tools do not need developers to arbitrarily write code to record the desired information. For example, one participant mentioned that they adopted Spring-sleuth [67]. It is a distributed tracing tool which can automatically record the interactions between multiple Sprint Boot microservices. These tools pre-instrument output statements to record information such as RPC calls and stack traces, and they need minimum configuration efforts. The participants also mentioned that although these tools are useful, they still cannot completely replace the needs for software logging in their projects.

6 DISCUSSIONS

Logs have been used extensively in practice. Instead of invoking output functions like System.out.println, developers prefer to use logging utilities (LU) to instrument their systems. This paper categorizes the complex use of LUs in the wild and presents multiple implications which are useful for developers and researchers. In this section, we will discuss the significance of our findings and present some future work.

6.1 Complex Use of LUs

There are generally four reasons to use LUs: (1) general-purpose logging, (2) LU interactions with imported packages, (3) internationalization of the log messages, and (4) modularization of the logging code. Larger projects tend to use multiple LUs in one project to satisfy one or more of the following usage context: (1) interaction with LUs from various imported packages, (2) managing the logging contents, (3) formatting logging messages across different components, and (4) developer convenience. Many projects also implement their own LUs to satisfy project specific needs like defining the logging format, compatibility, and ease of configuration and dependency management. Such findings would be useful for developers and researchers who are interested in developing and maintaining LUs:

- Our findings raise developers' awareness that their systems can contain multiple LUs due to the imported packages. It is important to manage these LUs to better monitor and debug systems' runtime behavior.

- To effectively monitor and debug systems' behavior, developers need to figure out how to configure the logging behavior for the imported packages and how to interact these LUs with the LUs in their own projects. Hence, techniques to recover the logging architecture are needed.
- To ensure QoS, best practices and anti-patterns for managing LUs are needed, particularly in the following areas: (1) selecting the appropriate default verbosity levels for LUs for good observability with low overhead; and (2) correlating logging information across different LUs in one project.
- In addition to common LUs (e.g., Log4j or SLF4J), many projects also use additional ELUs or implement ILUs. These LUs are project-dependent and require implementation/-maintenance effort. The findings in the paper can guide researchers to propose more general techniques to satisfy such logging needs.

6.2 Beyond LUs

Certain findings in this paper may be applicable to other utilities. It is important to study the use of different utilities in the wild due to: (a) multiple utilities (e.g., database/testing frameworks and ad libraries) with same/similar functionalities can be used in one project; and (b) cross-cutting concerns (e.g., logging, analytics and security) are implemented not only in the projects' code but also in the imported packages. Effectively managing such concerns is an open problem, which is important for development and maintenance of such systems [18, 84].

7 RELATED WORK

In this section, we discuss two realted research areas.

7.1 Empirical Studies on Logging Practices

There are no well-established logging guidelines in neither industrial [16, 56] nor open source systems [8, 63, 81]. Hence, it is important to derive best practices and common mistakes by studying the current logging practices. Shang et al. [63] found that the amount of logging code is correlated with the amount of post-release bugs. Kabinna et al. [33] studied the logging library migrations for 33 Apache-based Java projects. They found that migrating logging libraries requires high manual efforts and is error-prone. Zhi et al. [86] conducted an exploratory study on the configuration aspects of the LUs.

Our work is different from the above studies in the following three aspects: (1) this paper is the first study on the use of LUs, which includes both ELUs (a.k.a., logging libraries) as well as ILUs. (2) The scale of our study is much bigger than the previous ones, as we have studied over 11,194 Java-based GitHub projects, which uses 3,856 LUs. (3) Different from all of the above works, which are mainly quantitative studies, we have performed both the quantitative and the qualitative studies on the use of Java-based LUs.

7.2 Improving the Quality of the Logging Code

This area can be further divided into three parts: (1) where-to-log, (2) what-to-log, and (3) how-to-log.

- **where-to-log** is related to the problem of selecting appropriate logging points in the source code. Fu et al. [16] proposed a data-mining based approach to automatically extract the important attributes which affect the locations of the logging points. Zhu et al. [87] proposed a machine-learning based technique to learn common logging points based on the code structures. Yuan et al. [80] proposed a program-analysis based method to add logging points for debugging purposes. Ding et al. [11] proposed a constraint solving based method to select the optimal logging points which incur minimum performance overhead while keeping the maxmium runtime information. Zhao et al. [85] came up with a tool Log20, which automatically places the logging points under certain overhead threshold.
- **what-to-log** is related to the problem of providing complete runtime information in the logging code. Yuan et al. [82] proposed a program analysis based approach to add additional variables into existing logging statements so that more complete execution paths can be recovered, thus improving the diagnosability. He et al. [25] characterized the static texts inside logging statements for 10 Java and 7 C# projects. They provided an NLP-based description generation tool to automatically generate static texts in logging statements.
- **how-to-log** is related to the problem of designing and maintaining high quality logging code. Yuan et al. [81] partially studied the inconsistent verbosity level problem through a clone based approach. Li et al. learned from code changes in history to predict just-in-time logging code changes [39] and to suggest the most approapriate verbosity levels [38] through machine learning-based approaches. Chen et al. [9] summarized six types of anti-patterns within logging statements and proposed a tool to automatically detect them. Li et al. [40] proposed an automated tool to detect duplicate logging code smells. Chen et al. [10] proposed an approach to extract the Logging-Code-Issue-Introducing changes.

Our study lies within the category of **how-to-log**, but differs with the above works in the sense that our focus is on the LUs instead of the log printing code.

8 THREATS TO VALIDITY

In this section, we will discuss the threats to validity.

8.1 External Validity

We conducted our study only on 11,194 Java-based open source projects. We cross-validated our findings by interviewing with industrial developers. Although our approach can be easily adapted to another study on the use of LUs, our finding may not be generalizable to projects written in other programming languages.

8.2 Internal Validity

In our study, we removed all of the *very small*-sized projects, since only a small fraction of them use LUs and most of them are trial projects for self-studying and collections of code snippets. We also removed forked projects, since they are likely duplications of the base ones. We also removed unpopular projects which contain few stars. This practice is similar to many of the previous empirical studies on GitHub-based proejcts [5, 36, 58].

8.3 Construct Validity

In our study, we used a heuristics-based approach to count the adoptions of LUs. We made our best efforts to avoid bringing in any false positives. Through manual verification of the randomly selected samples, our approach yields an precision of 96%.

9 CONCLUSION

In this paper, we conducted a large-scale empirical study on the use of Java LUs in the wild. We studied 11, 194 Java-based projects in the GitHub. These projects use 3, 856 LUs. Our findings suggest that the complexity of the use of LUs increases as the project size increases. Although many projects only use LUs for general-purpose logging, the actual logging needs vary from project to project. The main reason behind multiple use of LUs is to enable or manage the logging behavior of the imported packages. Many projects choose to implement their own ILU mainly for defining the project-specific format of their logging code. The findings and the implications presented in this paper will be useful for LU designers and researchers as well as system developers.

REFERENCES

[1] Apache Commons Logging. 2019. https://commons.apache.org/proper/commons-logging/. Last accessed: 07/23/2019.

[2] Apache Hadoop. 2019. https://hadoop.apache.org/. Last accessed: 08/22/2019.

[3] Apache Tomcat. 2019. http://tomcat.apache.org/. Last accessed: 08/22/2019.

[4] Sven Apel, Don Batory, Sven Apel, and Don Batory. 2008. How AspectJ is used: An analysis of eleven Aspectj programs. *Journal of Object Technology (JOT)* (2008).

[5] Lingfeng Bao, Xin Xia, David Lo, and Gail C. Murphy. 2019. A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects. *IEEE Transactions on Software Engineering (TSE)* (2019).

[6] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The Bones of the System: A Case Study of Logging and Telemetry at Microsoft. In *Companion Proceedings of the 38th International Conference on Software Engineering (ICSE-C), 2016.*

[7] Bitcoin Wallet. 2019. Bitcoin Wallet app for your Android device. https://github.com/bitcoin-wallet/bitcoin-wallet. Last accessed: 08/23/2019.

[8] Boyuan Chen and Zhen Ming (Jack) Jiang. 2016. Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation. *Empirical Software Engineering (EMSE)* (2016).

[9] Boyuan Chen and Zhen Ming (Jack) Jiang. 2017. Characterizing and Detecting Anti-patterns in the Logging Code. In *Proceedings of the 39th International Conference on Software Engineering (ICSE), 2017.*

[10] Boyuan Chen and Zhen Ming (Jack) Jiang. 2019. Extracting and studying the Logging-Code-Issue- Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering (EMSE)* (2019).

[11] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-aware Logging Mechanism for Performance Diagnosis. In *Proceedings of the 2015 Usenix Annual Technical Conference (ATC), 2015.*

[12] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. *Selecting Empirical Methods for Software Engineering Research.*

[13] Eclipse Java IDE. 2019. . https://www.eclipse.org/ide/. Last accessed: 08/22/2019.

[14] Firebase. 2019. A mobile and web application development platform . https://firebase.google.com/. Last accessed: 08/23/2019.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc.

[16] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE-C), 2014.*

[17] GitHub features. 2019. https://github.com/features. Last accessed: 07/29/2019.

[18] Mathieu Goeminne and Tom Mens. 2015. Towards a survival analysis of database framework usage in Java projects. In *Proceedings of 31st International Conference on Software Maintenance and Evolution (ICSME), 2015.*

[19] Google Flogger. 2019. A Fluent Logging API for Java. https://github.com/google/flogger. Last accessed: 07/23/2019.

[20] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), 2013.*

[21] Jiawei Han. 2005. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[22] Ahmed E. Hassan and Richard C. Holt. 2004. Using Development History Sticky Notes to Understand Software Architecture. In *Proceedings of 12th International Workshop on Program Comprehension (IWPC), 2004,.*

[23] Mehran Hassani, Weiyi Shang, Emad Shihab, and Nikolaos Tsantalis. 2018. Studying and detecting log-related issues. *Empirical Software Engineering (EMSE)* (2018).

[24] HBase. 2018. Apache HBase. https://hbase.apache.org. Last accessed: 01/29/2018.

[25] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. Characterizing the Natural Language Descriptions in Software Logging Statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), 2018.*

[26] How to enable logging in dubbo? 2019. https://blog.csdn.net/JDream314/article/details/44620767. Last accessed: 08/21/2019.

[27] How to enable logging in Jetty. 2019. https://stackoverflow.com/questions/25786592/how-to-enable-logging-in-jettyt. Last accessed: 08/21/2019.

[28] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional.

[29] IntelliJ IDEA. 2019. https://www.jetbrains.com/idea/. Last accessed: 08/22/2019.

[30] JBoss Logging. 2019. https://developer.jboss.org/wiki/JBossLoggingTooling. Last accessed: 07/16/2019.

[31] JDT Java Development Tools. 2019. https://eclipse.org/jdt/. Last accessed: 07/23/2019.

[32] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 35th International Conference on Software Engineering (ICSE), 2013.*

[33] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging Library Migrations: A Case Study for the Apache Software Foundation Projects. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR), 2016.*

[34] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR), 2014.*

[35] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. *Aspect-oriented programming.*

[36] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A Large Scale Study of Multiple Programming Languages and Code Quality. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016.*

[37] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association (JASA)* (1952).

[38] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which Log Level Should Developers Choose for a New Logging Statement? *Empirical Software Engineering (EMSE)* (2017).

[39] Heng Li, Weiyi Shang, Ying Zou, and Ahmed E. Hassan. 2017. Towards just-in-time suggestions for log changes. *Empirical Software Engineering (EMSE)* (2017).

[40] Zhenhao Li, Tse-Hsun Chen, Jinqiu Yang, and Weiyi Shang. 2019. DLfinder: characterizing and detecting duplicate logging code smells. In *Proceedings of the 41st International Conference on Software Engineering (ICSE), 2019.*

[41] Jimmy Lin and Dmitriy Ryaboy. 2012. Scaling big data mining infrastructure: the twitter experience. *SIGKDD Explorations* 14, 2 (2012), 6–19.

[42] Log4J. 2019. A logging library for Java. http://logging.apache.org/log4j/1.2. Last accessed: 07/23/2019.

[43] LOG4J 2. 2019. Apache Log4j 2. http://logging.apache.org/log4j/2.x. Last accessed: 07/23/2019.

[44] log4net. 2019. log4net A logging library for .NET. https://logging.apache.org/log4net/. Last accessed: 07/23/2019.

[45] Logging dependency conflicts. 2019. https://cloud.tencent.com/developer/ask/121135. Last accessed: 08/21/2019.

[46] LogProvider in Ninja. 2019. https://github.com/ninjaframework/ninja. Last accessed: 08/22/2019.

[47] Cristina Lopes and Joel Ossher. 2015. How Scale Affects Structure in Java Programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015.*

[48] Mark Marron. 2018. Log++ Logging for a Cloud-native World. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS), 2018.*

[49] Maven plugin development guide. 2019. http://maven.apache.org/guides/plugin/guide-java-plugin-development.html. Last accessed: 08/23/2019.

[50] Multiple logging implementations found in Spring Boot. 2019. https://stackoverflow.com/questions/52911393/multiple-logging-implementations-

found-in-spring-boot. Last accessed: 08/21/2019.

[51] Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le Traon. 2013. Usage and Testability of AOP: An Empirical Study of AspectJ. *Information and Software Technology (IST)* (2013).

[52] OkHttp - an HTTP client for Android, Kotlin, and Java. 2019. https://github.com/square/okhttp. Last accessed: 08/23/2019.

[53] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Communications of ACM* (2012).

[54] Omni-Notes: note-taking application for Android. 2019. https://github.com/federicoiosue/Omni-Notes. Last accessed: 08/21/2019.

[55] Opentracing: vendor-neutral APIs and instrumentation for distributed tracing. 2019. https://opentracing.io/. Last accessed: 07/23/2019.

[56] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry Practices and Event Logging: Assessment of a Critical Software Development Process. In *Companion Proceedings of the 37th International Conference on Software Engineering (ICSE-C), 2015*.

[57] Liferay Portal. 2019. Liferay Portal - an open source enterprise web platform. https://github.com/liferay/liferay-portal. Last accessed: 08/22/2019.

[58] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. 2017. A Large-scale Study of Programming Languages and Code Quality in GitHub. *Communications of the ACM* (2017).

[59] Red Hat Wildfly. 2019. https://wildfly.org/. Last accessed: 08/22/2019.

[60] Mohammed Sayagh, Noureddine Kerzazi, Bram Adams, and Fabio Petrillo. 2018. Software Configuration Engineering in Practice: Interviews, Survey, and Systematic Literature Review. *IEEE Transactions on Software Engineering (TSE)* (2018).

[61] Sentry - cross-platform application monitoring, with a focus on error reporting. 2019. https://sentry.io. Last accessed: 08/23/2019.

[62] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. 2014. An exploratory study of the evolution of communicated information about the execution of large software systems. *Journal of Software: Evolution and Process (JSEP)* (2014).

[63] Weiyi Shang, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering (EMSE)* (2015).

[64] Simple Logging Facade for Java (SLF4J). 2019. https://www.slf4j.org/. Last accessed: 08/12/2019.

[65] spdlog - Very fast, header-only/compiled, C++ logging library. 2019. https://github.com/gabime/spdlog. Last accessed: 07/23/2019.

[66] Spring Boot pull request 4341. 2019. https://github.com/spring-projects/spring-boot/issues/4341. Last accessed: 08/21/2019.

[67] Spring Cloud Sleuth. 2019. https://spring.io/projects/spring-cloud-sleuth. Last accessed: 08/23/2019.

[68] StackOverflow: Disable Logback in SpringBoot. 2019. https://stackoverflow.com/questions/23984009/disable-logback-in-springboot/23991715. Last accessed: 08/01/2019.

[69] Telegram - a cloud-based instant messaging service. 2019. https://github.com/DrKLO/Telegram. Last accessed: 08/21/2019.

[70] The replication package. 2020. https://www.eecs.yorku.ca/~chenfsd/resources/icse2020_replication.zip. Last accessed: 01/26/2020.

[71] The State of Logging in Java. 2019. https://stackify.com/logging-java/. Last accessed: 08/21/2019.

[72] Timber - a logger with a small, extensible API which provides utility on top of Android's normal Log class. 2019. https://github.com/JakeWharton/timber. Last accessed: 08/23/2019.

[73] TIOBE Index for August 2019. 2019. https://www.tiobe.com/tiobe-index/. Last accessed: 07/23/2019.

[74] Ultimate Guide to Logging. 2019. https://www.loggly.com/ultimate-guide/java-logging-basics/. Last accessed: 08/21/2019.

[75] Vert.x - a tool-kit for building reactive applications on the JVM. 2019. https://github.com/eclipse-vertx/vert.x. Last accessed: 08/23/2019.

[76] What's Up with Logging in Java? 2019. https://stackoverflow.com/questions/354837/whats-up-with-logging-in-java. Last accessed: 07/23/2019.

[77] Why not use java.util.logging? 2019. https://stackoverflow.com/questions/11359187/why-not-use-java-util-logging. Last accessed: 07/23/2019.

[78] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC), 2018*.

[79] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2010*.

[80] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI), 2012*.

[81] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-source Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012*.

[82] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011*.

[83] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun (Peter) Chen. 2019. Studying the characteristics of logging practices in mobile apps: a case study on F-Droid. *Empirical Software Engineering (EMSE)* (2019).

[84] Ahmed Zerouali and Tom Mens. 2017. Analyzing the evolution of testing library usage in open source Java projects. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017*.

[85] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), 2017*.

[86] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, , and Tao Xie. 2019. An Exploratory Study of Logging Configuration Practice in Java. In *In Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019*.

[87] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE), 2015*.

[88] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu. 2019. Tools and benchmarks for automated log parsing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, (ICSE-SEIP), 2019*.