Table of Contents

Research Articles

1 Recommending Relevant Open Source Projects on GitHub using a Collaborative-Filtering Technique
Mohamed Guendouz, GeCoDe Laboratory, Tahar Moulay University of Saida, Saida, Algeria
Abdelmalek Amine, GeCoDe Laboratory, Tahar Moulay University of Saida, Saida, Algeria
Reda Mohamed Hamou, GeCoDe Laboratory, Tahar Moulay University of Saida, Saida, Algeria

17 Should Open-Source Technology be used in Design Education?
Tom Page, Loughborough University, Loughborough, UK

31 Open Source Developer Layer Assessment: Open Onion
Aminat Abiola Showole, University of Abuja, Abuja, Nigeria

49 Two Level Empirical Study of Logging Statements in Open Source Java Projects
Sangeeta Lal, Jaypee Institute of Information Technology, Noida, India
Neetu Sardana, Jaypee Institute of Information Technology, Noida, India
Ashish Sureka, ABB Corporate Research Center, Bangalore, India

Copyright
The International Journal of Open Source Software and Processes (IJOSSP) (ISSN 1942-3926; eISSN 1942-3934), Copyright © 2015 IGI Global. All rights, including translation into other languages reserved by the publisher. No part of this journal may be reproduced or used in any form or by any means without written permission from the publisher, except for noncommercial, educational use including classroom teaching purposes. Product or company names used in this journal are for identification purposes only. Inclusion of the names of the products or companies does not indicate a claim of ownership by IGI Global of the trademark or registered trademark. The views expressed in this journal are those of the authors but not necessarily of IGI Global.

The International Journal of Open Source Software and Processes is indexed or listed in the following: ACM Digital Library; Bacon’s Media Directory; Cabell’s Directories; Compendex (Elsevier Engineering Index); DBLP; GetCited; Google Scholar; INSPEC; JournalTOCs; MediaFinder; Norwegian Social Science Data Services (NSD); SCOPUS; The Index of Information Systems Journals; The Standard Periodical Directory; Ulrich’s Periodicals Directory
Two Level Empirical Study of Logging Statements in Open Source Java Projects

Sangeeta Lal, Jaypee Institute of Information Technology, Noida, India
Neetu Sardana, Jaypee Institute of Information Technology, Noida, India
Ashish Sureka, ABB Corporate Research Center, Bangalore, India

ABSTRACT

Log statements present in source code provide important information to the software developers because they are useful in various software development activities. Most of the previous studies on logging analysis and prediction provide insights and results after analyzing only a few code constructs. In this paper, the authors perform an in-depth and large-scale analysis of logging code constructs at two levels. They answer nine research questions related to statistical and content analysis. Statistical analysis at file level reveals that fewer files consist of log statements but logged files have a greater complexity than that of non-logged files. Results show that a positive correlation exists between size and logging count of the logged files. Statistical analysis on catch-blocks show that try-blocks associated with logged catch-blocks have greater complexity than non-logged catch-blocks and the logging ratio of an exception type is project specific. Content-based analysis of catch-blocks reveals the presence of different topics in try-blocks associated with logged and non-logged catch-blocks.

Keywords: Debugging, Dirichlet Allocation (LDA), Empirical Software Engineering and Measurement, Latent, Logging, Source Code Analysis, Source Code Metrics, Tracing

INTRODUCTION

Logging is an important software development practice that is used to record important program execution points in the source code. The recorded log generated from program execution provides important information to the software developers at the time of debugging. Fu et al. (2014) conducted a survey of Microsoft developers, asking them their opinion on source code logging. Results of the survey showed that 96 percent of the developers consider logging statements the primary source of information for problem diagnosis. In many scenarios, logging is the only information available to the software developers for debugging because the same execution...
environment is unavailable (which makes bug regeneration difficult) or the same user input is unavailable (because of security and privacy concerns) (Yuan et al., 2012). Yuan et al. (2012) showed in their characterization study that the bug reports consisting of logging statements get fixed 2.2 times faster compared to the bug reports not consisting of any logging statements. Logging statements are not only useful in debugging, but they are also useful in many other applications, such as anomaly detection (Fu et al., 2009), performance problem diagnosis (Nagaraj et al., 2012), and workload modeling (Sharma et al., 2011).

Logging statements are important, but they have an inherent cost and benefit tradeoff (Fu et al., 2014). A large number of logging statements can affect system performance because logging is an I/O-intensive activity, whereas an insufficient number of logging statements can miss important debugging information and can lessen the benefits of logging. Hence, developers need to avoid both excessive and insufficient logging. However, previous research and studies show that developers often face difficulty in optimal logging, that is, identifying which code construct to log in the source code (Fu et al., 2014; Zhu et al., 2015). It happens because of lack of training and the domain experience required for optimal logging. For example, Shang et al. (2015) reported an incident of a user from a Hadoop project complaining about less logging of catch-blocks. Recently the software engineering research community has conducted studies to understand the logging practices of software developers in order to build tools and techniques to help with automated logging. The current studies provide limited characterization study or conduct analysis on fewer code constructs. There are gaps in previous studies, as they do not analyze all the code constructs in detail, which this study aims to fill.

The work presented in this chapter is the first large-scale, in-depth, and focused study of logged and non-logged code constructs at multiple levels. High-level (source code files) and low-level (catch-blocks) analysis were conducted to identify relationships between code constructs and logging characteristics. This two-level analysis can reveal interesting insights for logging prediction tools. A case study was performed on three large, open-source Java projects: Apache Tomcat (Apache Tomcat, n.d.), CloudStack (Apache CloudStack, n. d.), and Hadoop (Page, n. d.). Following are the main research dimensions (RDs) considered in this work:

**RD1. Statistical Analysis of Source Code files**: The authors performed statistical analysis of logged and non-logged source code files. They analyzed the complexity, distribution, etc., of files.

**RD2. Statistical Analysis of Catch-Blocks**: The authors performed statistical analysis of logged and non-logged catch-blocks. They analyzed the complexity, logging ratio, contribution of an exception type in logged catch-blocks, etc.

**RD3. Content-based Analysis of Catch-Blocks**: The authors performed content-based analysis of contextual information present in the try-blocks associated with logged and non-logged catch-blocks. They used the Latent Dirichlet Allocation (LDA) topic modeling technique for content analysis.

**RELATED WORK AND RESEARCH CONTRIBUTION**

This section presents the closely related work and the novel research contributions of the study presented in this chapter in context to existing work. The authors categorize the related work in three dimensions: 1) improving source code logging, 2) uses of logging statements in other applications, and 3) applications of LDA in topic identification.
Improving Source Code Logging

Fu et al. (2014) empirically analyzed logging practices of software developers on two industrial systems. They addressed three research questions in their study: first, finding code snippets that were logged frequently; second, identifying the distinguishing characteristics of logged and non-logged code constructs; and third, building a tool for logging prediction. They analyzed 100 randomly chosen logging statements and identified the most frequently logged code construct types. They performed detailed analysis of return value check and exception snippets. They computed the logging ratio of each unique exception type and reported that the majority of the exception types falls in the range of a medium logging ratio (i.e., 10 percent to 90 percent). They analyzed 70 non-logged catch-blocks and identified the main reasons for not inserting a logging statement in the catch-block. They reported the correlations among the presence of some specific keywords that affect the logging decision such as “delete,” “remove,” “get,” etc. The machine learning–based tool proposed by Fu et al., which used contextual information from the code, gave an F-score of 80 percent to 90 percent. This shows that contextual information can be an important factor when making logging decisions. This study extends the characterization study performed by Fu et al. on many dimensions. First, the study performed by Fu et al. presents results on the basis of manual analysis of only a few code constructs, whereas in this work the authors present their analysis using much larger code constructs. Second, the authors extended their study by answering many more research questions at two levels. Third, the authors analyzed open-source Java project, whereas they analyzed closed-source C# projects. Hence, the results in this chapter can be reproduced by the software engineering research community.

Zhu et al. (2015) extended the study performed by Fu et al. (2014) by using more features for logging prediction on exception type and return value check. However, this study does not provide details of any statistical analysis of the features used for model creation. For example, they used lines of code of try-blocks as a feature for catch-block logging prediction, but did not provide any statistical evidence about it. The authors’ analysis of physical lines of code (SLOC) of try-blocks on three projects reveals that SLOC of try-blocks may not be a good feature for catch-block logging prediction, as SLOC of try-blocks associated with logged catch-blocks is not always higher than that of non-logged catch-blocks. Lal et al. (2016) proposed machine learning based tool for logging prediction on if-blocks and catch-blocks. They also provide only average values of the features used for prediction.

Yuan et al.’s (2012) work involved empirically analyzing modifications to log messages. They reported many interesting findings from their empirical analysis performed on four large open-source projects. Yuan et al. (2012) reported that 18 percent of all the committed revisions modify logging code, and 26 percent of the time developers modify the verbosity level of the logging code as an afterthought. Forty-five percent and 27 percent of the time developers modify the text and variable of the log messages, respectively, to incorporate changes in the execution information. Based on these findings, they proposed a simple code clone-based technique to find inconsistent verbosity levels in the source code. Kabinna et al.’s (2016) work on predicting the stability of logging statements used features from three different domains: context, developer, and content. In contrast to these studies, our work focuses on finding distinguishing features of logged and non-logged code constructs.

Uses of Logging Statements in Other Applications

Logging statements have been found useful in various software development tasks (Mariani & Pastore, 2008; Nagaraj et al., 2012; Shang et al., 2015; Xu et al., 2009; Yuan et al., 2010). Shang
et al. (2015) used logging statements present in a file to predict defects. Shang et al. (2015) proposed various product and process metrics using logging statements to predict post-release defects in software. Nagaraj et al. (2012) used good and bad logs of the system to detect performance issues. Nagaraj et al. (2012) also developed a tool, DISTALYZER, that helps developers find components responsible for poor system performance. Xu et al.’s (2009) work involved mining console logs from distributed systems at Google. They used logging information to find anomalies in the system. The authors verified anomalies were detected at the time when the system raised performance-related issues. They reported that performance issues are raised at the same time when anomalies are detected in system. Yuan et al. (2010) proposed a technique for finding the root cause of the failures by using logging information. They developed a tool, SherLog, that can use logs to find information about failed runs. SherLog can find important information about failures without requiring any re-execution of the code. All these studies focused on using log information in other applications such as finding root causes and performance issue detection. In contrast to these studies, the work described in this chapter focuses on the comparison between logged and non-logged code constructs at two levels.

LDA Applications in Topic Identification

LDA is a popular topic modeling technique (Blei et al., 2003). It has been utilized widely in various software engineering applications to discover meaningful topics (Barua et al., 2012; Pagano & Maalej, 2013; Thomas et al., 2014; Tian et al., 2009). Tian et al. (2009) used LDA for software categorization. They proposed a system that can learn topic models from the identifier and comments present in the source code and can categorize software into one of the 43 programming languages such as C, C++, Java, PHP, Perl, etc. Thomas et al. (2014) used LDA topic models for software evolution analysis. Results reported by them show that topic models are effective in discovering actual code changes. Pagano et al. (2013) used LDA to study blogging behavior of committers and noncommitters. Results showed that committers’ blogs consist of topics related to features and domain concepts, and 15 percent of the time blogs consist of topics related to source code. In contrast, blogs of noncommitters consist of topics related to conferences, events, configuration, and deployment. Barua et al. (2012) used LDA on a StackOverflow questions and answers dataset in order to discovery the most popular topics among the developer community. Results showed that a wide variety of topics are present in the developer discussions. They also showed that topics related to Web and mobile application development are gaining popularity compared to other topics. All these previous studies show the effectiveness of LDA topic models in the software engineering applications, and hence the authors of this chapter choose LDA for topic analysis. However, in contrast to these studies, the authors used LDA for topic identification in logging and nonlogging code constructs. To the best of their knowledge, LDA has never been used for topic modeling in this context.

Research Contribution

In the context of existing work, this chapter makes the following novel and unique research contributions:

1. The authors perform high-level (source code files) and low-level (catch-block) analysis of logged and non-logged code constructs. Multilevel analysis can provide useful insights and can help in identifying distinguishing features for logging prediction at various levels.
2. The authors perform statistical analysis of high-level and low-level code constructs. Statistical analysis can be beneficial in identifying properties of logging code constructs across the projects.
3. The authors perform content-based analysis of low-level code constructs and show that semantic analysis techniques such as LDA can be useful in discovering topics of logged and non-logged code constructs.
4. The authors present results of their empirical analysis on three large open-source Java projects: Apache Tomcat, CloudStack, and Hadoop.

**RESEARCH DIMENSIONS AND RESEARCH QUESTIONS**

Table 1 shows three main research dimensions (RDs) and respective research questions (RQs) considered in this work. Following is a brief description of each RD and respective RQs:

**RD1. Statistical Analysis of Source Code Files:** In RD1, the authors answer three main research questions related to the statistical properties of logged and non-logged files. Statistical analysis is important because it provides insights about the logged and non-logged code constructs without looking at the semantics of the code. The first and second RQs compute the percentage of logged files and their average SLOCs. The third RQ computes the correlation between file SLOC and respective logging count.

**RD2. Statistical Analysis of Catch-Blocks:** In RD2, the authors answer five research questions related to the statistical properties of logged and non-logged catch-blocks. The fourth research question compares the complexities of the try-blocks associated with logged and non-logged catch-blocks to investigate whether complexities of try-blocks have any effect on the corresponding catch-block logging decision or not. The fifth and seventh RQs compute the logging ratio of all the exception types and the top 20 exception types in all three projects. The sixth RQ computes the contribution of an exception type in total catch-blocks and total logged exception types. The eighth RQ investigates whether logged and non-logged catch-blocks can co-exist.

<table>
<thead>
<tr>
<th>Research Dimension</th>
<th>Research Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistical analysis of high-level code constructs</td>
<td>1. Is distribution of the logged files skewed?</td>
</tr>
<tr>
<td>(source code files)</td>
<td>2. Do logged files have greater complexity compared to that of non-logged files?</td>
</tr>
<tr>
<td></td>
<td>3. Is there a positive correlation between file complexity and log statement count?</td>
</tr>
<tr>
<td>Statistical analysis of low-level code constructs</td>
<td>4. Do try-blocks associated with logged catch-blocks have greater complexity compared to that of non-logged catch-blocks?</td>
</tr>
<tr>
<td>(catch-blocks)</td>
<td>5. What is the logging ratio of different exception types?</td>
</tr>
<tr>
<td></td>
<td>6. Is the exception type contribution the same in total catch-blocks as well as in total logged catch-blocks?</td>
</tr>
<tr>
<td></td>
<td>7. Are the top 20 exception types and their respective logging ratios the same in all three projects?</td>
</tr>
<tr>
<td></td>
<td>8. Can logged and non-logged catch-blocks co-exist?</td>
</tr>
<tr>
<td>Content-based analysis of low-level code constructs</td>
<td>9. Do try-blocks associated with logged and non-logged catch-blocks have different topics?</td>
</tr>
<tr>
<td>(catch-blocks)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Details of individual research questions addressed in each research dimension
RD3. Content-based Analysis of Catch-Blocks: In RD3, the authors use an LDA-based topic modeling technique on the contextual information present in the try-blocks associated with logged and non-logged catch-blocks. They hypothesize that the contextual information present in the try-blocks can reveal important information for the corresponding catch-block logging.

RESEARCH METHOD AND EXPERIMENTAL DATASET

This section presents the research methodology and experimental dataset details (refer to Figure 1). The research method consists of two phases: dataset selection and dataset preparation.

Dataset Selection Phase

In this phase, the authors selected open-source projects on which to conduct their experiments. Following are the list of properties and essential criteria which were taken into account while selecting the three open-source projects for the analysis:

1. **Type: Open Source:** The authors conducted their study on open-source software projects so that the work can be replicated and used for benchmarking and comparison.
2. **Programing Language: Java:** The authors selected a Java-based project for the study because Java is one of the most used programing languages (Kim, 2016; Krill, 2016).
3. **Logging Framework: Log4J:** The authors used Java projects utilizing the Log4J (Goers, n.d.) framework for logging. They targeted projects using the Log4J framework only because this is one of the widely used frameworks for Java logging.
4. **Number of Java Files: More than 1,000:** The authors set this threshold so that they can draw statistically significant conclusions.
5. **Number of Catch-Blocks: More than 1,000:** The authors set this threshold so that they can draw statistically significant conclusions.

![Figure 1. Research method followed in this study](image-url)
Experimental Dataset Details

The authors selected three projects for their empirical study based on the criteria defined for the dataset selection phase: Apache Tomcat, CloudStack, and Hadoop. All three projects are long-lived Java projects with a development history of ~7 to 17 years. Table 2 shows the SLOC of all three projects. SLOC are computed using the LocMetrics tool (LocMetrics, n.d.). Apache Tomcat, CloudStack and Hadoop have been previously used by the research community for logging and other studies (Kabinna et al., 2016; Lal & Sureka, 2016; Lal et al., 2016; Shang et al., 2015; Zimmermann et al., 2009). Following are the details of each project.

Apache Tomcat: Apache Tomcat is open-source software developed under the umbrella of the Apache Software Foundation (Apache Tomcat, n.d). It is a Web server that implements many Java EE specifications like Java Servlet, Java EL, Java Sever Pages, and WebSocket. Logging is important in Apache Tomcat Web server; it has its own LogManager implementation; and it also supports private per-application logging configurations (Crossley, n. d.; Team, n. d.).

CloudStack: CloudStack is open-source software developed by the Apache Software Foundation (Apache CloudStack, n.d). It provides public, private, and hybrid cloud solutions. It also provides a highly available and scalable Infrastructure as a Service (IaaS) cloud computing platform for deployment and management of networks of virtual machines. It provides support for many hypervisors such as VMware, KVM and Xen Cloud Platform (XCP). CloudStack provides large amounts of log entries, and for a CloudStack administrator investigating errors in the logs is an inevitable task (Kosinski, 2013).

Hadoop: Hadoop is also developed by the Apache Software Foundation (Page, n. d). It is a framework that enables distributed processing of large datasets. It is scalable from a single server to multiple machines. The Apache Hadoop library is designed to detect and handle application-layer failures. Hadoop is one of the most widely used software platforms, and various tools have been developed to monitor the status of the Hadoop using generated logs (Shang et al., 2015; Rabkin & Katz, 2010).

Dataset Preparation Phase

In this step, the authors extract logging statements and target code constructs from the source code. Following are the details of the data preparation.

Table 2. Experimental dataset details

<table>
<thead>
<tr>
<th>Project</th>
<th>Apache Tomcat</th>
<th>CloudStack</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>8.0.9</td>
<td>4.3.0</td>
<td>2.7.1</td>
</tr>
<tr>
<td>Logging Library</td>
<td>Log4j</td>
<td>Log4j</td>
<td>Log4j</td>
</tr>
<tr>
<td>Java File</td>
<td>2,037</td>
<td>5,351</td>
<td>6,332</td>
</tr>
<tr>
<td>SLOC</td>
<td>276,209</td>
<td>1,142,970</td>
<td>951,629</td>
</tr>
<tr>
<td>Log Line Count</td>
<td>2,703</td>
<td>10,428</td>
<td>10,108</td>
</tr>
<tr>
<td>Total Catch Blocks</td>
<td>3,325</td>
<td>12,591</td>
<td>7,947</td>
</tr>
<tr>
<td>Logged Catch Blocks</td>
<td>887 (27%)</td>
<td>2,790 (22.16%)</td>
<td>2,078 (26.15%)</td>
</tr>
<tr>
<td>Distinct Exception Types</td>
<td>120</td>
<td>163</td>
<td>265</td>
</tr>
</tbody>
</table>
Files: The authors extracted all the high-level (source code files) code constructs from the source code. They focused only on Java files in this work and removed other types of files such as CSS and XML. Table 2 shows statistics on the number of Java files extracted from each of the projects. For example, the Apache Tomcat project consists of 2,037 Java files, whereas the CloudStack project consists of 5,351 Java files. The authors extracted logging statements from each file (refer to Table 3). They marked a file as “logged” if it consisted of at least one logging statement; otherwise, it was marked as “non-logged.”

Catch-Blocks: Next the authors extract all the catch-blocks from the source code. They extracted all the catch-blocks from the Java files using the Eclipse Java source code parsing library (Beaton, n.d). However, a single try-block can have multiple catch-blocks. In such cases the authors considered all catch-blocks belonging to a single try-block as a separate instance. Figure 2 shows an illustrative example of separate instance creation. The authors marked a catch-block as “logged” if it consisted of at least one logging statement.

Table 3. The count (%) of logged files in the total files. It also shows the average SLOC of logged and non-logged files. LFC: Logged File Count; AS: Average SLOC; LF: Logged Files; NLF: Non-logged Files

<table>
<thead>
<tr>
<th>Project</th>
<th>Total Files</th>
<th>LFC (%)</th>
<th>AS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat</td>
<td>2,037</td>
<td>365 (17.9%)</td>
<td>260.04</td>
</tr>
<tr>
<td>CloudStack</td>
<td>5,351</td>
<td>798 (14.9%)</td>
<td>290.81</td>
</tr>
<tr>
<td>Hadoop</td>
<td>6,332</td>
<td>1414 (22.3%)</td>
<td>254</td>
</tr>
</tbody>
</table>

Figure 2. Catch-block instance creation from try-blocks

```
try
{  return m.invoke(base,(Object[])null);
}
catch (InvocationTargetException e)  
{  Throwable cause=e.getCause();
    ...  
}
catch(Exception e )
{
    throw new ELException(e);
}
```

```
try
{  return m.invoke(base,(Object[])null);
}
catch(Exception e )
{
    throw new ELException(e);
}
```
shows that the experimental dataset consists of 3,325, 12,591, and 7,947 catch-blocks in Apache Tomcat, CloudStack, and Hadoop, respectively. It also shows that 27 percent, 26.15 percent, and 22 percent of the catch-blocks are logged in Apache Tomcat, Hadoop, and CloudStack, respectively.

**Logging Lines:** All three projects used in the empirical study are Java and Log4J based projects. However, the authors observed several inconsistencies in logging statement formats and hence created 26 regular expressions to extract all the logging statements. The authors observed two semantically different types of logging: first, in which the logging level is explicitly mentioned (for example, Type 1 and Type 2 logging statements in Listing 1) and second, in which the logging level is not mentioned explicitly (for example, Type 4 and Type 5 in Listing 1). The authors also observed several inconsistencies in the uses of the log levels. For example, Listing 1 shows three different ways in which the log level “warn” is used in different datasets (refer to Type 1, Type 2, and Type 3 in Listing 1).

**Listing 1. Example of logging statements taken from the dataset**

```java
/*Type 1: (Taken from Hadoop)*/
LOG.warn(AuthenticationToken ignored: + ex.getMessage());
/*Type 2: (Taken from Hadoop)*/
logWarningWhenAuxServiceThrowExceptions(service, AuxServicesEvent. APPLICATION_INIT, th);
/*Type 3: (Taken from Apache Tomcat)*/
Logger.getLogger(getLoggerName(getHost(),url)).log(Level.WARNING,"Unable to determine web application context.xml " + docBase, e);
/*Type 4: (Taken from Apache Tomcat)*/
log("Error closing redirector: " + ioe.getMessage(),Project.MSG_ERR);
/*Type 5: (Taken from Apache Tomcat)*/
project.log(wrong object reference + refId + - + pref.get-Class());
```

**STATISTICAL ANALYSIS ON HIGH-LEVEL CODE CONSTRUCTS**

The following subsections present the work on characterizing high-level code constructs (source code files). The authors answer research questions related to the distribution and complexity of logged files. They also analyze correlations between the logging count of a file and its SLOC.

**RQ 1:** Is the Distribution of the Logged Files Skewed?

The authors counted the number of files that consisted of at least one logging statement. Table 3 shows that only 14.9 percent, 17.9 percent, and 22.3 percent of files consisted of logging statements in Apache Tomcat, CloudStack, and Hadoop, respectively. This result shows that distribution of files containing logging statements is highly skewed, that is, less than 23 percent of files consist of logging statements. The authors believe that understanding the characteristics of source code for files that do not contain any logging statements can provide useful insights.
for logging prediction tools, as the tool does not need to predict logging in the files, given that there is no history of logging statements.

The distribution of files containing and not containing log statements is skewed as only ≈14 percent to 22.3 percent of files contain logging statements.

**RQ 2: Do Logged Files Have Greater Complexity Compared to Non-logged Files?**

This subsection presents a comparison of the complexity of the logged and non-logged files. The authors measured the complexity of a file using its SLOC. To compute SLOC, they removed all the blank lines, package statements, import statements, and comments from the file. They also removed lines containing only ‘{‘ or ‘}’. Table 3 shows the values of average SLOC of logged and non-logged files for all three projects. The table also shows that for the Apache Tomcat project, the average SLOC value of logged and non-logged files is 260.04 and 69.37. Results show a similar trend for other two projects, that is, the average SLOC of logged files is higher than that of non-logged files. Average values provide useful statistics, but they lack significant details about the actual distribution. Hence the authors drew a box-and-whisker plot for the SLOC of logged and non-logged files. The graph in Figure 3 shows that the median SLOC values of logged files is higher that of non-logged files for all three projects. For example, the median SLOC value for logged files in the CloudStack project is 114.5, whereas the median SLOC value for non-logged files in the CloudStack project is 46.0. Figure 3 also shows a higher interquartile range for logged files, which shows a higher spread of SLOC values in logged files compared to that of non-logged files. The results presented in this subsection lead to many more questions regarding the analysis of more complex metrics (such as object-oriented metrics; Thwin & Quah 2005) of files to get a deeper understanding about the relation between file complexity and logging.

Logged files have greater complexity and spread (measured using SLOC of a file) as compared to that of non-logged files.

**RQ 3: Is There a Positive Correlation between File Complexity and Log Statement Count?**

The box-and-whiskers plot of the previous subsection shows that files with higher SLOC (i.e., higher complexity) are more likely to contain logging statements. Hence the authors hy-
pothesize that there exists a positive correlation between file SLOC and its log statement count, that is, the higher the SLOC, the higher the log statement count of the file. To test this hypothesis, the authors created a scatter plot between file SLOC and the respective log statement count. Scatter plots are one of the simplest yet powerful methods to visualize correlations between two variables. The authors created two scatter plots: the first scatter plot was between the SLOC of all the files in the database and respective log statement count, and the second scatter plot was between the SLOC only of logged files and respective log statement counts. The authors also computed the Pearson correlation between file SLOC and log statement count (Welcome to Statistics, n.d.). They obtained correlation values of 0.58, 0.76, and 0.67 for Apache Tomcat, CloudStack, and Hadoop, respectively, which shows that a positive correlation exists between SLOC and log statement count of a file (refer to Figure 5). However, it is interesting to observe the correlation value between file SLOC and logging count decreases after the addition of non-logged files (refer to Figure 4). The authors observed the presence of three (one in each project) very large, non-logged files in all three projects. Figure 4 shows these three files, marked using a red circle. Manual analysis reveals that these three files are tool-generated files and hence do not consist of any log statements. Table 4 gives details about the files and the tool used to generate these files. The experimental results presented in this subsection show a positive correlation between file SLOC and log statement count. The authors believe that these findings

Figure 4. Scatter plot showing correlation between SLOC of the files and respective logging counts: (a) Apache Tomcat; (b) CloudStack; (c) Hadoop

Figure 5. Scatter plot showing correlation between SLOC of only logged files and respective logging counts: (a) Apache Tomcat; (b) CloudStack; (c) Hadoop
A positive correlation exists between the SLOC of logged files and the logging count.

STATISTICAL ANALYSIS ON LOW-LEVEL CODE CONSTRUCTS

The following subsections work on characterizing low-level code constructs (catch-blocks). The authors answer research questions related to complexity, logging ratio distribution, and whether logged and non-logged catch-blocks can exist together.

RQ 4: Is the Complexity of Try-Blocks Associated with Logged Catch-Blocks Greater than That of Non-logged Catch-Blocks?

The authors compared the complexity of the try-blocks associated with logged and non-logged catch-blocks. They wanted to analyze whether the complexity of a try-block acts a parameter when deciding to log corresponding catch-blocks or not. In this work, they considered three parameters to measure the complexity of a try-block: size of the try-block (SLOC count), operator count of the try-block and method call count.

Comparing SLOC of Try-Blocks Associated with Logged and Non-Logged Catch-Blocks

The authors computed SLOC of the corresponding try-blocks associated with logged and non-logged catch-blocks. They computed SLOC using the same method described in a previous section. Listing 2 shows an example of a try-block from the Apache Tomcat project. The SLOC value of the try-block shown in Listing 2 is 2. Figure 6 shows box-and-whisker plots revealing the dispersion and skewness in SLOC for logged and non-logged try-blocks across three projects. The graph in Figure 6 reveals that the median and the third-quartile values for logged try-blocks are more than the corresponding values for non-logged try-blocks in Apache Tomcat and Hadoop. For example, the third quartile and median for logged try-blocks in the Apache Tomcat project is 7.0 and 2.0, respectively, whereas the third quartile and median for non-logged try-blocks in the Apache Tomcat project is 2.0 and 1.0, respectively. However, for the CloudStack project, the authors observed that the third quartile for logged try-blocks is higher than the third quartile for the non-logged try-blocks but the median value is smaller. The box plots in Figure 6 also reveal that the interquartile range (width of the box: Q3 – Q1) for logged try-blocks is higher than those of non-logged try-blocks, indicating a higher spread.

<table>
<thead>
<tr>
<th>Project</th>
<th>File Name</th>
<th>File SLOC</th>
<th>Log Count</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat</td>
<td>ELParser.java</td>
<td>2,272</td>
<td>0</td>
<td>Auto-generated using JJTree and JavaCC</td>
</tr>
<tr>
<td>CloudStack</td>
<td>AmazonEC2 Stub.java</td>
<td>250,323</td>
<td>0</td>
<td>Auto-generated using WSDL</td>
</tr>
<tr>
<td>Hadoop</td>
<td>Hamlet.java</td>
<td>19,431</td>
<td>0</td>
<td>Auto-generated using HamletGen</td>
</tr>
</tbody>
</table>

Table 4. Details of the three large non-logged files

A positive correlation exists between the SLOC of logged files and the logging count.
Listing 2. Example of a try-block taken from the Apache Tomcat project

```java
try{
    lc=new LoginContext(getLoginConfigName());
    lc.login();
}catch(LoginException e)
{
    log.error(sm.getString(spnegoAuthenticator.
    serviceLoginFail),e);
    response.sendRedirect(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    return false;
}

Try-LOC: 2
Operator Count: 7 (()()() =)
Method Call Count: 2 (getLoginConfigName, login)
Catch Exception: LoginException
```

Comparing Operator Count of Try-Blocks Associated with Logged and Non-logged Catch-Blocks

Counting the total number of operators in a program has been widely used as a metric to measure the complexity of given source code. The Halstead metric for computing program complexity is based on counting the total and distinct numbers of operators and operands in the source code (Virtual Machinery, n.d.). The authors created a list of 19 arithmetic operators (=, *, +, -, %, !, (, ), [, ], &, ?;::>, <, ^, ~, /) that perform normal mathematical operations such as add, subtract, multiplication, division, and modulo. They counted the number of operators (from the list of 19) in the try-block linked to logged and non-logged catch-blocks. The box plots in Figure 7 reveal that the third-quartile values for logged try-blocks (28, 47, and 18) are greater than the corresponding values (9, 45, and 12) for non-logged try-blocks in Apache Tomcat, CloudStack, and Hadoop. The median values for Apache Tomcat indicate that logged try-blocks have greater complexity in terms of operator count. The authors observed that the median value for logged try-blocks and non-logged try-blocks for the Hadoop project is the same. They believe that the lines-of-code metric is correlated to the number-of-operators metric, and hence they observe similar trends for both measures.
Comparing the Method Call Count of Try-Blocks Associated with Logged and Non-logged Catch-Blocks

The Halstead complexity measure computes program complexity based on several factors, such as the number of distinct operators and operands, as well as the total number of operators and operands. Predefined library function and user-defined function calls are considered operators according to the Halstead complexity metric. A large number of methods (equivalent to operators) within a try-block increases both cognitive complexity and testing complexity. Listing 2 shows an example of a try-block with two executable statements, both of which are function calls (getLoginConfigName() and login()). There can be two try-blocks with the same number of executable statements but a different number of function calls, and hence the complexity measure based on method call count is different from the complexity measure based on lines of code, as well as the complexity measure based on total number of operators. The authors computed the number of function calls for every try-block in the source code dataset. Figure 8 shows the box plots for the number of methods for the three projects in the experimental dataset. It reveals that the third-quartile value for the logged try-block is higher than the corresponding values for the non-logged try-block for the Apache Tomcat and Hadoop projects. For example, the median and third-quartile value for Apache Tomcat is 2.0 and 6.0, respectively, for the logged try-block.

Figure 8. Comparison of method call count of try-blocks associated with logged and non-logged catch-blocks: (a) Apache Tomcat; (b) CloudStack; (c) Hadoop
which is higher than the median and third-quartile value of 1.0 and 2.0, respectively, for the non-logged try-block. The authors observed that for the CloudStack project, the third-quartile value is the same for both logged and non-logged try-blocks.

These results show that the complexity of try-blocks associated with logged catch-blocks is not always greater than that of non-logged catch-blocks. Hence, the complexity of try-blocks should be used cautiously as a feature for logging prediction.

Try-blocks associated with logged catch-blocks have greater complexity than that of non-logged catch-blocks for the Apache Tomcat and Hadoop projects.

**RQ 5:** What Is the Logging Ratio Trend of the Various Exception Types?

This subsection statistically analyzes the logging ratio (LR) of distinct exception types in all three projects. The logging ratio of each exception type is computed using Equation 1 (refer to Table 5 for details about the acronyms used in this equation). The logging ratio metric is defined and used earlier by Fu et al. (2014) for analysis of exception types on C# projects. The logging ratio of an exception type shows the percentage of its logged catch-blocks (TLCBi) to its total number of catch-blocks (TCBi). For example, the “ChannelException” exception type in the Tomcat dataset has 25 catch-blocks, out of which 15 are logged. Hence, the logging ratio of ChannelException exception type i.e., LR_{ChannelException} is 60 percent. Figure 9 shows the histogram of the logging ratio of distinct exception types for all three projects. In Figure 9, the x-axis

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Catch-Blocks in the Dataset</td>
<td>TCB_{DT}</td>
</tr>
<tr>
<td>Total Logged Catch-Blocks in the Dataset</td>
<td>TLCB_{DT}</td>
</tr>
<tr>
<td>Total Catch-Block of i^{th} Exception Type</td>
<td>TCB_{i}</td>
</tr>
<tr>
<td>Total Logged Catch-Blocks of i^{th} Exception Type</td>
<td>TLCB_{i}</td>
</tr>
<tr>
<td>Exception Type Ratio (Catch Count) of i^{th}</td>
<td>ERCC_{i}</td>
</tr>
<tr>
<td>Exception Type Ratio (Log Count) of i^{th} Exception Type</td>
<td>ERLC_{i}</td>
</tr>
<tr>
<td>Logging Ratio of i^{th} Exception Type</td>
<td>LR_{i}</td>
</tr>
</tbody>
</table>

**Figure 9.** Logging ratio of all three projects: (a) Apache Tomcat; (b) CloudStack; (c) Hadoop
shows the range of the logging ratio (with an interval of 10 percent) and the y-axis shows the percentage of the distinct exception types falling in that range. On top of each bar of the histogram, the distinct exception types falling in that logging ratio range are plotted and counted. For example, Figure 9a shows that 47 exception types in the Apache Tomcat project have a logging ratio between 0 percent and 10 percent. Fu et al. (2014) reported in their study that the majority of exception types belong to either a very high (>=90%) or low (<=10%) logging ratio range. Although they computed the results on C# projects and results of a Java project can differ, the authors observed results similar to Fu et al. (2014) with Java projects.

\[ LR_i = \frac{\text{TLCB}_i \times 100}{\text{TCB}_i} \]  

(1)

The majority of the exception types in the Java project belong to either a very high (>=90%) or very low (<=10%) logging ratio.

**RQ 6:** Is the Exception Type Contribution the Same in Total Catch-Blocks and in Total Logged Catch-Blocks?

This subsection measures the contribution of each exception type in total catch-blocks as well as in total logged catch-blocks. The authors define two metrics: Exception Type Ratio (Catch Count) [ERCC] and Exception Type Ratio (Log Count) [ERLC] for the same; refer to Equation 2 and Equation 3 for details (refer to Table 5 to get details on the acronyms used in these equations). ERCC defines the percentage of contribution of a particular exception type in total catch-blocks whereas ERLC defines it for logged catch blocks. ERCC computes the percentage of total catch-blocks of an exception type (TCB) to total catch-blocks in the dataset (TCBD), whereas the ERLC metric computes the percentage of total logged catch-blocks of an exception type (TLCB) to total logged catch-blocks in the dataset (TLCBD). For example, for the Apache Tomcat project we have TCB\text{DT} = 3325 and TLCB\text{DT} = 887. Now for ‘ChannelException’ exception type we have TCB\text{ChannelException} = 25 and TLCB\text{ChannelException} = 15. Hence, for ChannelException type we have value of ERCC=7.5 and ERLC=1.69. The motivation behind computing these two metrics is to find exception types that contribute a great deal to total catch-blocks and less to logged catch-blocks, or vice versa. For example, ChannelException type have comparatively less contribution in logged catch-blocks as compared to that in all catch-blocks because for it the value of ERCC metric is greater than ERLC metric. Early detection of such exception types can be beneficial to developers as well as logging prediction tools, because exception-specific rules can be created for such exception types. Figure 10 shows the histogram of ERCC metrics for all three projects. In Figure 10, the x-axis shows the ERCC range and the y-axis shows the sum of ERCC values of all the exception types falling in that ERCC range, that is, all the exception types falling in a particular group give an ERCC range and their ERCC value can be summed. On top of each bar is a count of unique exception types falling in that ERCC range. For example, Figure 10a shows that for the Apache Tomcat project 116 exception types have an ERCC metric value between 0 and 5 and sum of their ERCC values is 35, that is, 116 exception types together constitute 35 percent of total catch-blocks. Figure 10a also shows that two exception types in the Apache Tomcat project have ERCC values between 20 and 25 and together they constitute 42 percent of total catch-blocks. Figure 11 shows the histogram for ERLC metrics for all three
projects. In Figure 11, the x-axis shows the ERCC range and the y-axis shows the sum of ERLC values of all the exception types falling in that ERCC range. The x-axis is the same in both graphs so the contribution of the same exception type in total catch-blocks (using the ERCC metric) and in total logged catch-blocks (using the ERLC metric) can be compared. Figure 11a shows that 116 exception types together constitute 38 percent of all logged catch-blocks, whereas these same 116 exception types constitute only 35 percent of total catch-blocks (refer to Figure 10a).

Figure 10b and Figure 11b show an interesting finding about four exception types (marked by arrows in the Figure) from the CloudStack project. These four exception types—ADBException, AxisFault, IllegalArgumentException, and XMLStreamException—have very large ERCC values (i.e., very large contribution in total catch-blocks), but very low ERLC values (i.e., much smaller contribution in the total logged catch-blocks). For example, the exception type “XMLStream” has 1,952 occurrences in the CloudStack project, but none of these occurrences are logged. Detection of such exception types can be beneficial for logging prediction tools, as the tool can learn the exception types that have such drastic differences in ERCC and ERLC values.

\[
ERCC_i = \frac{|TCB_i| * 100}{|TCB_{DT}|}
\]
Some exception types in the CloudStack project have very high ERCC values (i.e., very large contribution in total catch-blocks) but very low ERLC values (i.e., much less contribution in the total logged catch-blocks).

**RQ 7:** Are the Top 20 Exception Types and Their Respective Logging Ratios the Same in All Three Projects?

This section presents an analysis of the logging ratio of the top 20 most frequent exception types in all three projects. The authors plotted a pie chart showing the contribution of the top 20 most frequent exception types in total catch-blocks. Figure 12 shows that the top 20 most frequent exception types contribute to ≈80 percent to 88 percent of the total catch-blocks for all three projects. Hence, analyzing the top 20 exception types can be crucial for the logging prediction tools, as ≈80 percent of the time the tool will be making a prediction for one of these top 20 exception types. In addition to this, if a similar trend exists regarding the logging ratio of the top 20 exception types across the projects, then it can be beneficial for cross-project logging prediction.

**Table 6. Details of four exception types from the CloudStack project with very high ERCC but very low ERLC values. CCB: Count of Total Catch-Blocks; CLCB: Count of Total Logged Catch-Blocks; LR: Logging Ratio**

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>CCB</th>
<th>ERCC</th>
<th>CLCB</th>
<th>ERLC</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADB Exception</td>
<td>960</td>
<td>0.0762</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AxisFault</td>
<td>643</td>
<td>0.0511</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>1,971</td>
<td>0.1565</td>
<td>19</td>
<td>0.0068</td>
<td>0.0096</td>
</tr>
<tr>
<td>XMLStreamException</td>
<td>1,952</td>
<td>0.155</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\text{ERLC}_i = \frac{\left| T\text{LCB}_i \right| \times 100}{T\text{LCB}_{DT}} \quad (3)
\]

Some exception types in the CloudStack project have very high ERCC values (i.e., very large contribution in total catch-blocks) but very low ERLC values (i.e., much less contribution in the total logged catch-blocks).

**Figure 12. Pie chart of top 20 exception types, showing percentage in total contribution: (a) Apache Tomcat; (b) CloudStack; (c) Hadoop**
The authors wanted to answer two interesting research questions about the top 20 exception types: Are these top 20 exception types the same across all the projects, and do the top 20 exception types show common trends for logging ratios in all three projects? To answer these research questions, the authors computed the top 20 exception types, as well as their respective logging ratios (using Equation 1) for all three projects. The answer to the first question is “No.” Results show that only 6 exception types are common among the three projects in the top 20 exception type list. Table 7 shows details of these six common exception types (Exception, IOException, Throwable, InterruptedException, IllegalStateException, and IllegalArgumentException) in all the three projects. The Throwable class is the superclass of all errors and exceptions in the Java language. The Exception class and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch. Throwable and Exception are higher-level classes (Exception extends Throwable, which extends the root of the class hierarchy Object), with several subclasses defining specific exception types; hence, they are common. The authors believe that classes like InterruptedException are common because Apache Tomcat, CloudStack, and Hadoop extensively use multithreading, and InterruptedException is thrown when a thread is waiting, sleeping, or otherwise occupied and the thread is interrupted. The authors compared logging ratios of six common exceptions in all three projects. Table 7 shows no specific trend for logging ratios across the three projects. For example, the exception type “Exception” has a low logging ratio for the Apache Tomcat and Hadoop projects (i.e., 37.25 percent and 27.72 percent), whereas for the CloudStack project, it has a high logging ratio (i.e., 66.81 percent). The authors observed similar trend for other exception types. Thus, the answer to the second research question is also “No.” This indicates that the logging ratio of an exception type is project specific, and hence a cross-project defect-prediction technique might need more sophisticated features than logging ratio.

The most frequent exception types, as well as their respective logging ratios, are project specific.

RQ 8: Do Logged and Non-logged Catch-Blocks Coexist Together?

The Java programing language allows associating multiple catch-blocks (each with a different exception type) to a single try-block. In this subsection, the authors’ aim is to investigate whether a single try-block can have both logged and non-logged catch-blocks or not. This research question is important to answer, as many times catch-block logging prediction tools use features from try-blocks. If the frequency of such try-blocks is very high, then it can affect the

Table 7. Logging ratio details of 6 common exceptions in the top 20 exception type list of the three projects

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Apache Tomcat</th>
<th>CloudStack</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>37.25%</td>
<td>66.81%</td>
<td>27.72%</td>
</tr>
<tr>
<td>IOException</td>
<td>27.41%</td>
<td>54.69%</td>
<td>36.69%</td>
</tr>
<tr>
<td>Throwable</td>
<td>45.66%</td>
<td>72.24%</td>
<td>53.05%</td>
</tr>
<tr>
<td>InterruptedException</td>
<td>6.12%</td>
<td>25.15%</td>
<td>23.28%</td>
</tr>
<tr>
<td>ClassNotFoundException</td>
<td>32.14%</td>
<td>4.48%</td>
<td>6.49%</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>23.19%</td>
<td>0.96%</td>
<td>16.67%</td>
</tr>
</tbody>
</table>
performance of such machine learning–based catch-block logging prediction tools. To answer this research question, the authors computed the count of try-blocks with both logged and non-logged catch-blocks in all three projects. Table 8 shows that a very small percentage (i.e., ≈0.33 percent to 1.4 percent) of total try-blocks has both logged and non-logged catch-blocks.

**Content-based Analysis of Low-Level Code Constructs**

This section presents the experimental results of content-based analysis of low-level (catch-blocks) logged and non-logged code constructs. The authors applied LDA for content analysis. LDA is a popular topic modeling technique and has been used widely in the past for topic identification in the source code and in many other research areas (Thomas et al., 2014; Maskeri et al., 2008; Pagano & Maalej, 2013). The following subsections describe the steps of the LDA model creation, results of LDA topic modeling, and the authors’ observations from the obtained results.

**Preprocessing Steps for LDA Analysis**

The preprocessing steps for LDA model creation are as follows:

1. To identify the topics present in the logged and non-logged catch-blocks, the authors analyzed the contents of the try-blocks associated with logged and non-logged catch-blocks. They created corpus consisting of the content of try-blocks associated with logged and non-logged catch-blocks.
2. The authors performed preprocessing and removed all the English stop words, special characters, and operators. They removed English stop words such as “is,” “the,” and “of” from the analysis because they were mainly interested in identifying the core functionality of the code constructs that leads to logging. The authors then applied stemming on the obtained corpus. Stemming is useful in reducing inflected words to the same root words and hence helps in reducing the corpus size. The authors used the Python NLTK library for stop word removal and stemming (Natural Language Toolkit, n.d.).
3. The authors believe that words that occur in almost all the documents or that occur in very few documents may not be helpful in retrieving useful topics. Hence they removed all the words that occurred in 80 percent of the documents and in less than 2 percent of the documents.
4. The authors performed LDA for 10,000 runs because LDA gives better results when the number of iterations is increased. Previous studies in software engineering research have also used the same threshold value for LDA (Thomas et al., 2014).
5. The authors set the number of topics parameter for the LDA algorithm as 10.
6. The authors used a default value of other LDA parameters in the Python LDA library (Gensim, n.d.).

<table>
<thead>
<tr>
<th>Project</th>
<th>Apache Tomcat</th>
<th>CloudStack</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique try-blocks</td>
<td>2,914</td>
<td>9,899</td>
<td>7,171</td>
</tr>
<tr>
<td>Try-block with more than one catch-block</td>
<td>254</td>
<td>1,002</td>
<td>653</td>
</tr>
<tr>
<td>Try-block with mix of logged and non-logged catch-blocks</td>
<td>41 (1.40%)</td>
<td>31 (0.31%)</td>
<td>77 (1.07%)</td>
</tr>
</tbody>
</table>

A very small percentage of try-blocks have both logged and non-logged catch-blocks.
RQ 9: Do Try-Blocks Associated with Logged and Non-logged Catch-Blocks Have Different Topics?

Table 9 shows the result obtained by LDA topic modeling on try-blocks associated with both logged and non-logged catch-blocks. From this table, the authors observed that topics listed under try-blocks associated with logged and non-logged catch-blocks are different. Hence they randomly picked some of the topics from the logged and non-logged category and analyzed the differences in the associated code blocks. The authors drew the following interesting observations from this analysis:

1. They observed the “thread sleep” topic in the Apache Tomcat project. This topic is mentioned in the non-logged catch-block category. The authors further analyzed occurrences of “thread sleep” in the Apache Tomcat project. They observed that in 84 occurrences of “thread sleep,” it occurred 71 times in try-blocks associated with non-logged catch-blocks.
2. The authors observed the presence of a topic related to “socket” in both try-blocks associated with logged and non-logged catch-blocks. They analyzed all 43 try-blocks consisting of socket and wrapper words and found that in try-blocks associated with logged catch-blocks, the socket wrapper is mostly used for close or error functions, whereas for try-blocks associated with non-logged catch-blocks, the socket function is used for timeout operations. LDA is able to detect this difference, as shown in the Apache Tomcat project regarding topics 4 (logged catch-blocks) and 3 (non-logged catch-blocks).
3. The authors analyzed the “result stub (topic 1)” topic from the CloudStack project. They found 161 occurrences of try-blocks consisting of both words. They also noticed that catch-blocks associated with all 161 try-blocks are non-logged. LDA is able to detect this because the “request stub” topic is not present in logged catch-blocks.

The contextual information present in the try-blocks provides important information for the associated catch-block logging.

CONCLUSION AND FUTURE WORK

Source code logging is an important software development practice, and tools and techniques that can help software developers make optimal and strategic logging decisions can be benefi-

<table>
<thead>
<tr>
<th>Project</th>
<th>Logged Catch-Block</th>
<th>Word</th>
<th>Non-logged Catch-Block</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tomcat</td>
<td>1. channel file 2. method param. 3. context log 4. socket status</td>
<td>1. channel, get, file, new, stream 2. method, get, param, valu, type 3. context, get, log, null, host 4. socket, statu, get, wrapper</td>
<td>1. thread sleep 2. channel read 3. socket pool 4. connect pool</td>
<td></td>
</tr>
<tr>
<td>CloudStack</td>
<td>1. byte key 2. response value 3. network 4. vm host</td>
<td>1. key, byte, new, pair, string 2. respons, valu, string, name, equal 3. network, ip, host, string, conn 4. vm, host, cmd, answer, state</td>
<td>1. result stub 2. pram om 3. java lang 4. stmt</td>
<td></td>
</tr>
<tr>
<td>Hadoop</td>
<td>1. key id 2. assert 3. job conf 4. rm token</td>
<td>1. id, key, get, contain, info 2. request, fail, system, assert 3. job, get, name, map, conf 4. token, rm, get, except, new</td>
<td>1. user token 2. key 3. get response 4. file path</td>
<td></td>
</tr>
</tbody>
</table>

Table 9. Topics discovered in try-blocks associated with logged and non-logged catch-blocks
cial. Analysis of logged and non-logged code constructs can provide useful insights to improve current logging prediction tools. The work presented in this chapter is the first (to the best of the authors’ knowledge) large-scale, two-level, empirical study of logged and non-logged code constructs. The authors performed statistical and content-based analysis of source code files and catch-blocks from three large open-source Java projects. They answered nine research questions in this chapter. Following are the main research findings of this work:

- Fewer files consist of logging statements.
- Source code files with logging statements have a much larger average SLOC compared to those without logging statements.
- There is a positive correlation between the SLOC of logged files and their respective log statement counts.
- Try-blocks associated with logged catch-blocks have greater complexity than that of non-logged catch-blocks for the Apache Tomcat and Hadoop projects.
- Some exception types contribute greatly to total catch-blocks, whereas there is little or no contribution in total logged catch-blocks.
- The logging ratio of an exception type is project specific.
- The LDA-based topic modeling technique is effective in discovering topics of logged and non-logged code constructs.

The authors think that this work provides a future direction for two lines of work: statistical analysis and content-based analysis. Statistical analysis provides the ability to explore more deeply the features of logged code constructs. In this work, the authors analyzed a complexity metric (SLOC, operator count, etc.) with respect to logged and non-logged code constructs. However, many other source code metrics, such as inheritance depth, and object-oriented metrics need to be evaluated for deeper analysis of logged and non-logged files. Content-based research needs more exploration in terms of the topics present in the logged and non-logged code constructs. In this work, the authors used LDA for topic modeling, and the initial results are encouraging. However, deeper analysis of code constructs with respect to multiple semantic techniques such as LDA and Latent Semantic Indexing (LSI) is required for in-depth analysis of the topics present in logged and non-logged code constructs.

**THREATS TO VALIDITY**

**Number and Type of Project:** The authors selected Apache Tomcat, CloudStack, and Hadoop projects for the study. All three projects are open-source, Java-based projects. Other types of projects, such as closed source, or projects written in other languages (e.g., C#, Python) need to be evaluated. Overall, the authors cannot draw any general conclusion that is applicable to all software logging. They believe that this study provides insight about logging practices of open-source, Java-based projects.

**Quality of Ground Truth:** The authors assumed that logging statements inserted by software developers of Apache Tomcat, CloudStack, and Hadoop project are optimal. There is the possibility of errors or nonoptimal logging in the code by the developers, which can affect the results of the study. However, all three projects are long lived and are actively maintained; hence it is safe to assume that most of the code constructs have good (if not optimal) logging. The authors used 26 regular expressions to extract the logging statements from the source code. Manual analysis reveals that all the logging statements were extracted (to the best of the authors’ knowledge). However, there is still a possibility that the regular expressions missed some types of logging statements in the source code.
REFERENCES


Sangeeta Lal is an Assistant Professor in the department of computer science in Jaypee Institute of Information Technology, Noida, India. She has completed her masters from IIIT-D and currently pursuing her PhD from Jaypee Institute of Information Technology under the guidance of Dr. Neetu Sardana and Dr. Ashish Sureka. She has published papers in many national and international conferences. Her research interest is in mining software repositories. She has experience in working in various domains of software repositories such as source code, bug reports, version control system, and Q&A websites. Currently she is working on log statements analysis in Java code.

Neetu Sardana is an Assistant Professor in the department of computer science in Jaypee Institute of Information Technology, Noida, India. Her research interest is in the area of mining software repositories and social network analysis. She has supervised several M.Tech theses and B.Tech major projects. She has published papers in many national and international conferences and journal. She is currently serving as a PC member of IC3 conference which is held every year in JIIT-Noida.

Ashish Sureka is a Principal Researcher at ABB Corporate Research Center (India). He was a Faculty Member (at IIIT-Delhi) from July 2009 to October 2014 and a visiting researcher at Siemens Corporate Research from August 2014 - July 2015. His current research interests are in the area of Mining Software Repositories, Software Analytics, and Social Media Analytics.