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.
Enhancing Education for the Knowledge Society Era with Learning Ecosystems
www.igi-global.com/chapter/enhancing-education-for-the-knowledge-society-era-with-learning-ecosystems/168977?camid=4v1a

Collaboration in Open Source Domains: A Perspective on Usability
www.igi-global.com/article/collaboration-open-source-domains/38903?camid=4v1a