Where Do Developers Log? An Empirical Study on Logging Practices in Industry

Qiang Fu[1], Jieming Zhu[2], Wenlu Hu[3], Jian-Guang Lou[1], Rui Ding[1], Qingwei Lin[1], Dongmei Zhang[1], Tao Xie[4]

[1]Microsoft Research Asia, Beijing, China
{qifu,jlou,juding,qlin,dongmeiz}@microsoft.com

[2]The Chinese University of Hong Kong, HK, China
jmzhu@cuhk.edu.hk

[3]Carnegie Mellon University, PA, USA
wenlu@cmu.edu

[4]University of Illinois at Urbana-Champaign, IL, USA
taoxie@illinois.edu

ABSTRACT
System logs are widely used in various tasks of software system management. It is crucial to avoid logging too little or too much. To achieve so, developers need to make informed decisions on where to log and what to log in their logging practices during development. However, there exists no work on studying such logging practices in industry or helping developers make informed decisions. To fill this significant gap, in this paper, we systematically study the logging practices of developers in industry, with focus on *where* developers log. We obtain six valuable findings by conducting source code analysis on two large industrial systems (2.5M and 10.4M LOC, respectively) at Microsoft. We further validate these findings via a questionnaire survey with 54 experienced developers in Microsoft. In addition, our study demonstrates the high accuracy of up to 90% F-Score in predicting where to log.

Categories and Subject Descriptors
B.2.3 [**Arithmetic and Logic Structures**]: Reliability, Testing, and Fault-Tolerance – *diagnostics and error-checking*

General Terms
Reliability, Standardization

Keywords
Logging practice, automatic logging, developer survey

1. INTRODUCTION
Logging is a conventional programming practice that saves important runtime information by adding statements in source code as shown below:

Log (level, "logging message %s", variable);

A logging statement typically consists of a logging function and its parameters, including text messages, variables, and a verbosity level (e.g., *fatal*/*error*/*info*) to specify the severity of the logged event.

The importance of logging can be widely identified by the various usages of logs in software system management tasks, including anomaly detection [26, 27], error debugging [8], performance diagnosis [11, 24], workload modeling [17], system behavior understanding [28], etc. Additionally, its importance is also reflected by our survey results with 54 experienced developers in Microsoft (more details on the survey are in Section 2). Almost all the participants (96%) *strongly-agree/agree* that logging statements are important in system development and maintenance. About 96% of the participants think that logs are a primary source for problem diagnosis. Furthermore, 93% of the participants answer that they take good consideration in practice when adding a logging statement.

Given the importance of logging, it is crucial to avoid logging too little, which may miss necessary runtime information (e.g., error sites) needed for postmortem analysis. On the other hand, it is also crucial to avoid logging too much for three main reasons. First, logging incurs both system runtime overhead (e.g., CPU consumption and I/O operations) and storage cost. For instance, one of our studied software systems produces about 2 gigabytes of logs per machine per day on average. Second, arbitrarily placed logging statements likely generate a lot of trivial logs that may be redundant or useless, thus masking the truly important information in logs. Third, logging itself incurs cost of code development and maintenance.

To avoid both logging too little and logging too much, developers need to make informed decisions on where to log and what to log in their logging practices during development. However, to the best of our knowledge, there exists no work on studying such logging practices in industry (e.g., how such decisions are made in practice) or helping developers make informed decisions.

To fill this significant gap, in this paper, we systematically study the logging practices of developers in industry, with focus on *where* developers log. Where to log has a great impact on log quality because logging locations reveal execution code paths, which have been shown helpful in postmortem analysis [23]. This study helps understand the current logging practices of developers and serves as the first step towards improving the logging practices in industry.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591175

Our study considers developers' logging behaviors from the point of view of each logging statement and its logged code snippet. Such logged code snippet is a block of source code whose behavior the logging statement intends to log. In the study, we investigate where to log by studying whether to log for a code snippet, through the following three research questions:

RQ1: *What categories of code snippets are logged?* Specifically, we intend to investigate the logging statements and their logged code snippets to figure out the categories of the logged snippets.

RQ2: *What factors are considered for logging?* Specifically, what are characteristics of logged code snippets and unlogged code snippets, respectively? What factors do developers consider to determine whether to log or not?

RQ3: *Is it possible to automatically determine where to log?* In particular, is it potentially feasible to implement a logging-suggestion tool to assist developers to determine where to log?

Our study includes both source code analysis and a questionnaire survey with developers in Microsoft. In particular, we select two industrial software systems at Microsoft for study. These two systems have been deployed to serve users globally for a long time, and their logging instrumentation has been well tuned and refined.

Through detailed source code analysis on these software systems, we obtain a number of insightful findings regarding the logging practices of developers. Furthermore, we validate these findings with survey responses from 54 experienced developers in Microsoft. The results of our study show that there are generally five categories of logged snippets (*Finding* 1), in which about half of them are used to record unexpected situations (e.g., exceptions or function return errors), and the other half records normal execution information at critical execution points (*Finding* 2). Developers add logging statements to only a small percentage of situations related to exceptions or checked function calls (*Finding* 3), by considering factors such as exception types and context information (*Findings* 4~5). In addition, we evaluate the potential feasibility of predicting whether to log for a code snippet. The high prediction accuracy implies that predicting where to log is feasible (*Finding* 6).

In summary, our paper makes the following main contributions:

- We conduct an empirical study on logging practices in industry by both source code analysis for industrial software systems and a questionnaire survey with 54 experienced developers in Microsoft.
- We summarize five categories of logged snippets, including *assertion-check logging*, *return-value-check logging*, *exception logging*, *logic-branch logging*, and *observing-point logging*, covering all the scenarios of logging observed in our study.
- We characterize both logged and unlogged code snippets of catch blocks and return-value-check snippets, in which we find important factors on logging decisions, such as exception type and context factors.
- We demonstrate the potential feasibility of predicting where to log, which is valuable for further exploration.

The rest of this paper is organized as follows. Section 2 introduces our study methodology. Sections 3-5 discuss three research questions on what categories of code snippets are logged, what factors are considered for logging, and the feasibility of predicting where to log. Then Section 6 suggests some potential directions for improving logging practices. Sections 7 and 8 discuss threats to validity and related work, respectively, and finally Section 9 concludes this paper.

2. STUDY METHODOLOGY

In our study, we investigate two large industrial software systems, denoted as *System-A* and *System-B*, respectively, for the sake of confidentiality. Both of them are online service systems deployed globally in Microsoft data centers to serve a huge number of users. These online service systems provide rich features to enable diverse use scenarios, thus gaining high complexity. Table 1 presents the details of the two software systems. Both of them are selected due to their high popularity and long history of development. In addition, to allow the research community to reproduce or apply our study methodology on other systems, on our project website[1] , we release the detailed study materials and results of applying our study methodology on MonoDevelop, a large open-source project with 9 years of development history.

Table 1. Details of the studied software systems

Software systems	Description	LOC	# of logging statements	% logging statements
System-A	Online service	2.5M	23.5K	0.94%
System-B	Online service	10.4M	95.3K	0.92%

The development of the two systems was supposed to incorporate good logging practices for two reasons. First, these systems have been serving users worldwide for a long time, and their generated logs have successfully met the requirement of system testing, debugging, and operating. Second, our previous work [7] studied the logs of one of the systems, and proposed a solution for effectively diagnosing system failures by using the logs, indirectly reflecting the high quality of the logs.

Most parts of the two studied software systems are written in the C# programming language. While some other languages (e.g., ASP, C) exist in these systems, we focus on only the C# parts. C# is an object-oriented programming language that supports the exception-handling mechanism. In C#, the common practice of error handling is throwing exceptions instead of returning error codes. Thus, exceptions attract a lot of attention in our study.

We characterize the logging practices from both quantitative and qualitative aspects: statistical source code analysis and empirical developer survey.

Source Code Analysis. We first investigate the logging characteristics from the point of view of source code. To achieve this goal, we analyze the logging statements and their logged code snippets through both manual inspection and automatic analysis via a static analysis tool.

Because logging statements and their logged code snippets are quite diverse, we first define feasible criteria for categorizing logged snippets to make each category coherent for detailed in-depth category-specific analysis.

In order to identify factors that developers consider to determine where to log, we further characterize both logged and unlogged code snippets, especially for catch blocks, to reveal detailed log-

[1]http://research.microsoft.com/en-us/projects/loggingpractice/default.aspx

ging statistics. We further extract the identified logging factors as key features to predict where to log.

For our study, we develop automatic C# code analysis based on *Roslyn* [16], a static analysis tool released by Microsoft. By leveraging Roslyn, we conduct both syntax analysis and semantic analysis for source code.

Developer Survey. To further validate and elaborate our findings from source code analysis, we conduct a questionnaire survey, as well as some follow-up discussions via emails, with developers in Microsoft.

The questionnaire (available on our project website[1]) consists of 21 questions, which can be divided into four parts, including *background information of participants*, *importance of logging*, *current logging practices*, and *improving current practices*. While most of the questions have a list of choices, the participants can also write down their additional answers in a free-text form. Additionally, some open-ended questions are provided to enable the participants to add or elaborate their own ideas.

This questionnaire survey was conducted in August 2013, and we received 54 survey responses from developers with an average of 5.3 years of working experience at Microsoft. The participants work on various types of products including standalone desktop applications (9%), Web applications (20%), mobile applications (3%), software/Web services (59%), and some others (9%). When necessary, follow-up discussions were also held with participants via emails to help us understand their survey answers.

3. CATEGORIES OF LOGGED SNIPPETS

To understand the logging practices in terms of where to log, the first step is to investigate what categories of code snippets are usually logged (***RQ1***). We answer ***RQ1*** with two steps to make the categorization effort manageable: (1) manually categorizing a randomly sampled set of logging statements and their logged code snippets; (2) automatically classifying the full set of logging statements and their logged snippets into those categories via a static analysis tool.

3.1 Manual Categorization

We randomly sampled 100 logging statements from source code of System-A, and manually examined each logging statement and its logged code snippet to conduct categorization. Based on the syntax and structure of each logged snippet, we identified five categories. Table 2 presents the category names and the number of samples in each category.

Table 2. Categories from 100 sampled logged code snippets

Category		Samples	#Votes	% of votes
Unexpected situations	Assertion-check logging	19/100	27/54	50%
	Return-value-check logging	14/100	34/54	63%
	Exception logging	27/100	43/54	80%
Execution points	Logic-branch logging	16/100	36/54	67%
	Observing-point logging	24/100	44/54	81%

We next illustrate the details of each category as below:

(1) Assertion-check logging. In this category, developers use *Assert* (or similar functions) to perform assertion checking in the source code to identify errors. Failed assert statements automatically log the failure messages before execution termination. Example 1 in Figure 1 illustrates a real-world logging statement in this category to assert whether *site* is null. Out of the 100 samples in our study, 19 samples belong to this category.

```
/* Example 1: Assertion-check logging */
ULS.AssertTag(site != null, "site cannot be null");

/* Example 2: Return-value-check logging */
if (String.IsNullOrEmpty(tokenReference))
    ULS.SendTraceTag(ULSTraceLevel.Unexpected, "Missing token reference
value.");

/* Example 3: Exception logging */
try {
    RemoveOfflineAddressBooks();
}
catch(AccountUnauthorizedException e) {
    Logger.LogMessage("Removing failed with exception: {0}", e);
}

/* Example 4: Logic-branch logging */
if (instanceName.IsSqlExpressInstalled) {
    Tracer.TraceLogInfo("Detect sql express instance. No need to in-
stall.");
}
else {
    Tracer.TraceLogInfo("No sql express instance. Do fresh install.");
    res = SqlCleanInstall();
}

/* Example 5: Observing-point logging */
Tracer.TraceLogInfo("Creating the tab order for form {0}", base.Name);
```

Figure 1. Real-world examples of logging statements

(2) Return-value-check logging. In this category, logging statements are used to log potential function return errors after performing a return-value check. We find that incorrect return values of function calls (e.g., system/library calls) are widely used indicators of potential errors. It is a common practice to have a check on the return value of a function call, as illustrated by Example 2. By explicitly checking for contingencies using special return values (e.g., -1, *false*, *null,* and *empty*), developers can identify the unexpected errors (e.g., the null or empty token reference in Example 2) and log them accordingly. In our study, 14 samples belong to this category.

(3) Exception logging. In this category, developers log the exception context after an exception occurs (e.g., in a *catch* block or right before a *throw* statement). Exceptions are widely used mechanisms to capture errors in modern programming languages (e.g., C#, Java). Example 3 depicts a detailed example of *exception logging*, in which the exception thrown by function *RemoveOfflineAddressBooks* is captured in the try bock and then logged in its corresponding catch block. 27 samples in our study belong to this category.

(4) Logic-branch logging. In this category, logging statements are leveraged to record the runtime execution information at logic branch points, i.e., the code execution path. Logic branches in source code are typically generated by using a branch statement such as *if* or *switch*, which leads to different code execution paths. Log messages at critical branch points can help identify causally-related code execution paths and facilitate backward inference for root-cause identification of failures. Example 4 provides one of the 16 samples in this category, in which two branches are both logged to record the execution-path information.

(5) Observing-point logging. Except the above-mentioned categories of logged code snippets, we categorize all the other logged code snippets as *observing-point logging*. This category

Table 3. Categorization criteria

Category	Criteria
Assertion-check logging	The logging statement is triggered by an *Assert* statement.
Return-value-check logging	The logging statement is contained in a clause following a branch statement (e.g., *if*, *if-else*, *switch*), while one or more function return values are checked in the branch condition. In addition, the logging statement is not enclosed by any catch block within the clause.
Exception logging	The logging statement is contained either in a catch block or right before a throw statement.
Logic-branch logging	The logging statement is contained in a clause following a branch statement (e.g., *if*, *if-else*, *switch*), while the branch condition does not contain any check on a function return value.
Observing-point logging	All the other situations that exclude the above categories.

has various scenarios for logging. It may log at the entry/exit point of a function, record an important transaction, and report critical events (e.g., heartbeats) to ensure that the system is running as expected. We consider these logging points as observing points to observe and understand the runtime states of systems. In our study, 24 samples belong to this category.

In summary, we find that the first three categories of logged code snippets record *unexpected situations* that should not occur in normal executions, while the last two categories record normal execution information at critical *execution points*. In summary, two types of information are usually recorded:

- **Unexpected situations**. Assertion check, return-value check, and exceptions (i.e., the first three categories) are usually used to identify the unexpected situations, where the system potentially runs into an error. These points are typically logged, since the generated logs are greatly helpful in identifying error sites of the system when a failure happens.
- **Execution points**. Logic branch points and other observing points are informative execution points of the code flow. As a result, recording important execution information (e.g., execution path, system runtime states) by logging at these critical execution points can facilitate root-cause identification of an occurred failure.

In practice, when a failure occurs, developers usually identify the error site from the logs related to *unexpected situations*, and then trace back to identify the root cause of the failure based on the logs that record code execution path and states at a series of important *execution points*.

Survey results. To obtain developers' opinions about the identified categories, we have two questions in our developer survey.

First, we ask participants to tick the most common categories among the categories we have identified from the above manual categorization (via a multiple-choice question). The result is provided in Table 2 as "#Votes", which indicates the number of participants who consider the category as common. "% of votes" is the ratio between "#Votes" and the total number of participants. The top two with the highest votes are exception logging and observing-point logging, which suggests the great importance and ubiquity of recording exception context and runtime information of critical execution points.

Second, the participants are asked to list any additional categories not covered by our categorization. In the collected responses, some participants stated that calling an external component (e.g., RPC call or SQL request) is usually logged. Actually, we categorize this case into the observing-point logging. Another response is that entry/exit points of critical function calls are usually logged to record latency. Similarly, we also consider this case as observing-point logging. In fact, observing-point logging can be further divided into multiple sub-categories, which we plan to investigate in our future work.

Finding 1: There are five categories of logged code snippets, i.e., assertion-check logging, return-value-check logging, exception logging, logic-branch logging, and observing-point logging.

3.2 Automatic Categorization

In this subsection, we study the distribution of different categories of logged snippets in our studied software systems. For a logging statement, the categorization criteria are defined based on the syntax and structure of its logged code snippet. Table 3 shows the formal syntax definitions that describe how to automatically identify each category in source code.

For any given logging statement and its logged snippets, we examine whether they satisfy one of the five criteria one by one, from assertion-check logging to observing-point logging with the order in Table 3. Note that all these criteria are checked within the scope of the function that contains the logging statement. Once a logging statement and its logged snippet are judged as satisfying one criterion, the logged snippet is categorized into the corresponding category, and is not further checked against the remaining subsequent conditions.

Table 4. Categorization of logged snippets

Category	System-A	System-B
Assertion-check	5,476 (23%)	20,186 (21%)
Return-value-check	2,716 (12%)	8,959 (9%)
Exception	4,333 (18%)	8,399 (9%)
Subtotal: Unexpected situations	**12,525 (53%)**	**37,544 (39%)**
Logic-branch	3,807 (16%)	16,658 (18%)
Observing-point	7,170 (31%)	41,138 (43%)
Subtotal: Execution points	**10,977 (47%)**	**57,796 (61%)**
Total	**23,502**	**95,340**

Table 4 presents the categorization results. It is observed that all categories of logged snippets are pervasive in the two software systems. The category of *observing-point logging* has the highest number of logged snippets among the five categories, since this category contains various logging scenarios. In addition, almost half (39%~53%) of logged snippets are used to handle *unexpected situations* and record the needed information, while the other half

(47%~61%) record execution information at critical *execution points*, reflecting their importance in log analysis.

Finding 2: About half of the logged snippets are logged due to unexpected situations, while the other half are due to recording normal execution information at critical execution points.

In the subsequent sections, we focus on logging characteristics of unexpected situations, because they usually indicate error sites, and take up about half the logged snippets. Although logging at critical execution points is also important, especially for identifying root causes, it is not the focus of this paper. In fact, identifying root causes is a subsequent problem after finding error sites, which we leave for our future work.

4. FACTORS FOR LOGGING DECISION

Unexpected situations are often exposed under some typical patterns (i.e., assertion check, return-value check, and exception). Among these typical patterns, we mainly focus on two of them: catch blocks and return-value check. We do not study assertion-check snippets because all of the assertions are actually logged. For clarity, we denote code snippets of catch blocks and return-value-check snippets as *focused code snippets*.

Note that NOT every focused code snippet reveals an unexpected situation and is worth logging. For example, not all exceptions are unexpected. In many cases, exceptions are caught to indicate normal branch conditions, thus not being logged, as shown in Section 4.3.

We extract all the *focused code snippets* (i.e., catch blocks and return-value-check snippets) from source code, and further analyze what factors are considered for logging (***RQ2***) at certain focused code snippets, by characterizing both logged code snippets and unlogged code snippets, respectively. In more details, Section 4.1 discusses the overall logging statistics of focused code snippets. Section 4.2 and 4.3 present the in-depth analysis on logged and unlogged focused code snippets, respectively. Finally, Section 4.4 discusses some other factors such as contextual information.

4.1 Logging Statistics of Focused Code Snippets

We extract every catch block from the source code, and record its corresponding exception type. For instance, *AccountUnauthorizedException* is the exception type of the catch block in Example 3 of Figure 1. Especially, for those catch blocks with no explicitly specified exception types (i.e., no arguments in their catch statements), we denote their exception types as *System.Exception*. Then, we identify whether a catch block is logged by checking the existence of any logging statement in it. Similarly, for return-value-check code snippets, we identify all the function call sites in the source code, and record their function types. Note that we refer to a function type as a function prototype, e.g., *bool String.IsNullOrEmpty(string)* in Example 2 of Figure 1. We then check whether the return value of each function is checked in an *if* or *switch* statement. If checked, we further identify whether it is logged.

Table 5 shows the detailed statistics of catch blocks and return-value-check snippets. About 30%~42% of catch blocks are logged, while the logging ratio of checked function-call sites is only 8%~9%. The results show that only a small portion of focused code snippets are logged in practice.

We next study the detailed characteristics of logged/unlogged catch blocks and return-value-check snippets, respectively. However, we report the results for only catch blocks here due to the space limit, whereas the findings on return-value-check snippets are similar.

Table 5. Logging statistics of unexpected situations

Statistics		System-A	System-B
Catch block	Exception types	225	1657
	Catch blocks	7,582	21,656
	Logged catch blocks	3,222 (42%)	6,410 (30%)
Return value check	Function types	21,813	155,444
	Function call-sites	131,390	723,691
	Checked call-sites	34,464	104,167
	Logged call-sites	2,716 (8%)	8,959 (9%)

Finding 3: Only a small portion of focused code snippets are logged, including 30%~42% of the catch blocks and 8%~9% of the checked function-call sites. This observation (i.e., not all the focused code snippets need to be logged) calls for examination of the validity of the assumption made by Errlog [20].

4.2 Characterizing Logged Catch Blocks

To further understand the logging characteristics of catch blocks, we conduct an in-depth analysis on logged catch blocks with regard to exception types, considering that an exception type indicates one type of unexpected situations with specific semantic meanings. For a certain exception type, we count its corresponding number of catch blocks, and the number of logged catch blocks as well. Based on these numbers, we derive the logging ratio of each exception type, i.e., the number of logged catch blocks divided by the number of catch blocks in each exception type.

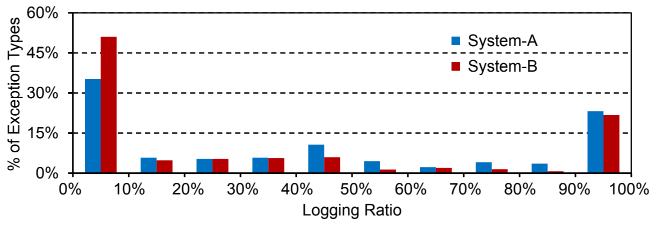


Figure 2. The distribution of exception types over logging ratio

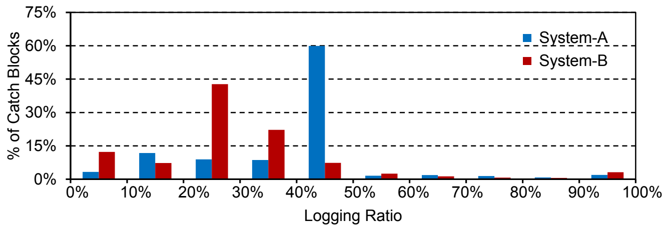


Figure 3. The distribution of catch blocks over logging ratio

Figure 2 illustrates the distribution of exception types with respect to the logging ratio at an interval of 10%. For better visualization, we normalize the number of exception types into the corresponding percentage out of the total number of exception types, as shown in the vertical axis. Note that all the numbers are calculated separately for the two software systems. As we can observe, a large percentage of exception types have either quite high (>90%)

or quite low (<10%) logging ratios. Specifically, an exception type with a high logging ratio means that developers log nearly every catch block of this exception type (e.g., *System.OutOfMemoryException,* and *Microsoft.BusinessData.Runtime.AuthenticationException* in Software-A). In contrast, an exception type with a low logging ratio indicates that the catch blocks with this exception type are rarely logged (e.g., *Microsoft.DuplicateValuesFoundException, System.EntryPointNotFoundException* in System-A). Consequently, these exception types have high correlation (i.e., either positive correlation or negative correlation) with the logging decisions of developers.

Additionally, we present the distribution of catch blocks across the logging ratio corresponding to each exception type in Figure 3. The vertical axis is also normalized to the percentage of catch blocks. It is shown that most of the catch blocks (including 91% in System-A and 82% in System-B) correspond to the exception types with a medium logging ratio, i.e., 10%~90% in our study. Only a small portion of catch blocks have exception types with quite high (>90%) or quite low (<10%) logging ratios. For example, in Software-A, 35% (79/225) of the exception types have low logging ratios, but the percentage of catch blocks with these exception types is only 3% (247/7,582).

These results indicate that although there are many exception types highly correlated with logging, they actually take up only a small portion of catch blocks. In other words, developers do not make the logging decisions merely based on the caught exception types for most of the catch blocks.

Survey results. To validate this result, we ask participants what factors they often consider to log in a catch block. The answers indicate that 57% of participants take the exception type as an important factor for logging, while some other factors are also considered (see Figure 5).

Finding 4: Many exception types have high correlations with developers' logging decisions in their catch blocks. However, the catch blocks corresponding to such exception types take up only a small portion of catch blocks. In other words, most catch blocks correspond to the exception types that are not highly correlated with developers' logging decisions.

4.3 Characterizing Unlogged Catch Blocks

In this section, we focus on those unlogged catch blocks, in order to investigate factors for not logging, i.e., potential reasons why developers do not log in a specific catch block.

According to the results in Table 5, the majority of catch blocks (e.g., 58% in System-A) are not logged in practice. We randomly select 70 unlogged catch blocks from System-A to characterize potential reasons of not logging.

We summarize the reasons by examining the related code snippets and understanding the code logic. In many cases, we are able to understand the code logic by reading only the specific function that contains the catch block. However, for some complicated samples, we need to further read the caller functions or ask for the help of code owners in order to understand the code logic.

Table 6 summarizes three categories of reasons, as well as their corresponding distribution over 70 samples. We next provide the description on each of these categories.

Table 6. Reasons of NOT logging in catch blocks

Reasons of not logging	Samples	% of samples	#Votes	% of votes
Logging decisions are made by subsequent operations	29/70	41%	34/54	63%
Exceptions are not critical	32/70	46%	7/54	13%
Exceptions are recoverable	9/70	13%	17/54	31%

```
/* Example 6: An exception used to determine logic branch*/
void AccountConfig(MONOAccount user, string propertyName) {
    ...
    bool userHasRights = true;
    try {
        user.DeleteAccountProperty(propertyName);
    }
    catch (UnauthorizedAccessException) {
        userHasRights = false;
    }
    if (userHasRights) {
        ...
    }
}
```

```
/* Example 7: An exception re-thrown */
try {
    Type t = Type.GetTypeFromID(guid);
    object instance = Activator.CreateInstance(t);
}
catch (Exception e) {
    throw new TestFailedException("Fail to create Com interface.\t: "
        + Tester.GetExceptionDetails(e));
}
```

```
/* Example 8: An exception recovered by retrying */
void DWAppOverride(...) {
   ...
   Uri UriNew = null;
   try {
      UriNew = new Uri(wApp);
   }
   catch (UriObjectFormatException) {
    // Assume http is the scheme and the URL param is the machine name
     if (UriNew == null) {
        try {
           UriNew = new Uri("http://" + wApp);
        }
     }
   }
}
```

Figure 4. Real-world examples of unlogged catch blocks

(1) Logging decision is made by subsequent operations. 29 samples are not logged because the catch blocks only execute some operations (e.g., setting properties/flags, re-throwing the exceptions or returning special values to their caller) to indicate exceptional states; while their subsequent operations determine whether to log for the exceptional states under certain context. Example 6 in Figure 4 provides an example of this category in which the catch block is used to catch the "*UnauthorizedAccessException*" exception. Then, the program sets the flag "*userHasRights*" to *false*, and then directs the execution to the subsequent logic branch. In Example 7, the caught *Exception e is* rethrown to its caller as a *TestFailedException*, and its callers determine whether to log the exceptions at a higher level.

(2) Exceptions are not critical. 32 samples are not logged because the caught exceptions do not have critical impacts on subsequent executions. For example, the execution on a code snippet continues normally to process the input requests even though an exception is thrown by a subtask.

(3) Exceptions are recoverable. 9 samples are not logged because the caught exceptions are recoverable, and the system executes the recovery actions to cope with these exceptions. According to our examination, we find two kinds of recovery actions: (a) the system retries the same or alternative operations for the same purpose until it succeeds or exceeds the maximal retry times (or time limit) and thus throws a new exception; (b) instead of retry, the system executes exception-handling operations, to bypass the failed operations. Example 8 illustrates an example of an exception recovered by retry. When the program fails to create an *Uri* object, it uses the default scheme *http* to create the object again.

Survey results. To obtain developers' opinions about the reasons of not logging, we ask participants to tick the most common ones based on the categories we have found from the above manual examination of unlogged snippets (via a multiple-choice question). The result is provided in Table 6 as "#Votes", which indicates the number of participants who consider the reason category as common. The first category has the highest votes, while the second category has the lowest votes, since it is actually difficult to identify whether the exception is critical, which largely depends on the domain knowledge of developers.

In addition, the participants are asked to write any additional reasons of not logging. No additional category of reasons is identified from their detailed answers.

Finding 5: The majority (58%~70% shown in Table 5) of catch blocks are not logged mainly because (1) passing the logging decision to subsequent operations, (2) exceptions are recoverable, and (3) exceptions are not critical.

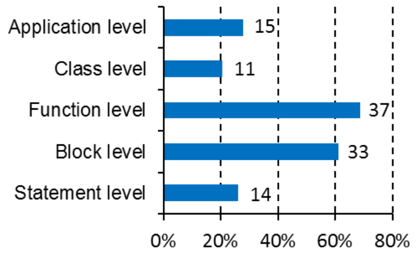
4.4 Other Factors for Logging Decision

From the preceding analysis, we observe that developers do not make a logging decision according to only the caught exception type (*Finding* 4). In addition, contextual information, such as whether the caught exceptions are critical or recoverable, is also taken into consideration (*Finding* 5). In other words, the decision to log for a code snippet is often highly related to the semantic functionality of the whole code snippet.

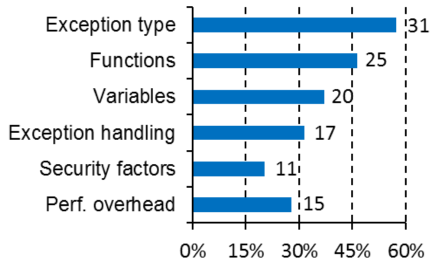
To validate this intuition, we conduct a case study by investigating 20 samples randomly sampled from the catch blocks with *FileNotFoundException* in System-A. The overall logging ratio for the catch blocks with *FileNotFoundException* is 35%. After examining the source code of these 20 code-snippet samples, we found that some contextual keywords are highly correlated to the decisions on whether to log a code snippet. For example, for try blocks containing the keyword "*delete*", the exceptions of *FileNotFoundException* are often not logged. The reason is that if the main task of a try block is to delete a file, it does not bring a negative effect for ignoring the message "the file does not exist" (i.e., *FileNotFoundException*). In our case study, there are also some other extracted keywords that are highly related to logging decisions, e.g., "*remove*" (20%), "*load*" (100%), "*get*" (100%), where the numbers denote their corresponding logging ratios.

Survey results. To further facilitate our understanding in how developers usually make logging decisions, we have two specific questions in our survey: what scope of source code and what factors do developers mostly consider to determine whether to log?

For the source-code scope, we present the survey results in Figure 5(a). The results show that most of the participants consider in the scope of function level (69%) and block level (61%). In other words, when encountering an exception, the function containing the exception (function level) or the corresponding try block (block level) is likely considered. Others such as the statement that throws the exception (statement level), the class containing the exception (class level), and the whole application logic (application level) are less considered by developers. With regard to the decision factors as shown in Figure 5(b), most participants consider the exception type (57%) and the function calls related to the exception (46%). In contrast, other factors, such as the related variables (37%), exception-handling operations (31%), security factors (20%), and performance overhead (28%), are less considered for logging decision.



(a) Scope



(b) Factors

Figure 5. Scope and factors considered for logging decision

5. AUTOMATIC LOGGING

Based on the findings described in Section 4, it is valuable to know whether it is possible to automatically predict where to log. If possible, such automatic prediction on where to log would greatly reduce the effort that developers spend on logging decision, and also improve their logging practices. In this section, we propose an automatic classification approach to predict whether to log for a code snippet, in order to demonstrate the potential feasibility of predicting where to log (***RQ3***).

This approach extracts the contextual information as features, and learns a classifier from the training code snippets. With the learned classifier, the approach predicts whether new code snippets need to be logged.

To achieve this goal, we need to first extract the contextual information related to logging decisions. Through our study, we find that the source code in our studied software systems is in good coding styles. Most of the functions, identifiers, etc., are well named using related semantic keywords in a good format. Consequently, in most cases, the functionality of a code snippet can be well understood based on the names of all the functions in this code snippet, the name of the function containing this snippet, the name of the class containing this snippet, and the keywords of the surrounding comment strings (if they exist). Therefore, we extract and segment these names and strings to a bag of words, and use them as features to learn a classifier for prediction of where to log.

More specifically, the approach is designed as a two-step procedure as follows.

Step1: Extracting contextual keywords. For each code snippet (i.e., try-catch block or return-value-check snippet), we denote the function that contains this snippet as its container function and the class containing this snippet as its container class. To extract the contextual keywords, we segment all names of the functions in each code snippet, as well as its container function name and its container class name, into a set of separate words. Specifically, for a catch block, we extract function names from the corresponding try block; while for a return-value-check snippet, we extract all the function names from the beginning of the container function to the statement of return-value check. As almost all the names used in our studied systems are well formatted (i.e., separating words using capital letters), we employ the following simple yet effective technique to address the word-segmentation problem: (a) We use *Roslyn* to transform the instance name to its prototype name. For example, the function name "*user.DeleteAccountProperty*" in Example 6 is transformed to "*MONOAccount.DeleteAccountProperty*". (b) We leverage the locations of upper-case letters and non-letter characters (e.g., dot or underscore) to split the name into words. Especially, for consecutive capital letters, we take them together as a single word (e.g., "*MONO*"). (c) Normalize each word to its lower-case format. In the preceding example, the function name is segmented and normalized into four words, i.e., "mono", "account", "delete" and "property". Meanwhile, we calculate the term frequency for each word. For example, in the above example, the frequency of the word "account" is 3 (twice in the function name in the try block and once in its container function). As a result, we obtain a bag of (contextual) words and their corresponding term frequency as features for each code snippet. Note that other sophisticated techniques (e.g., stemming, stopping-word removal) can also be used to further enhance the accuracy of contextual keyword extraction.

Step2: Learning logging classifier. Each code snippet is labeled as one of the two classes: *logged*/*unlogged*. As observed in Table 5, the logged code snippets are much fewer than the unlogged snippets (i.e., unbalanced classes). Therefore, we use the subsampling technique to address this problem. In other words, we randomly sample the unlogged code snippets to get an equal number with logged snippets. Then we feed these features with their labels into a decision-tree learner C4.5 [15] to learn a classifier. As a result, the logging status of each testing code snippet can be predicted as logged or unlogged by this classifier. Note that we extract only the function names and class names as features in our evaluation, since we find that the semantic functionality of a code snippet generally can be well denoted by their related functions, while all the others such as variables and parameters likely increase noises.

Result analysis. We employ the 10-fold cross-validation [29] to evaluate the accuracy of our prediction. Two groups of experiments are conducted, while one group uses the exception type (catch block)/function type (return-value-check snippet) as features, and the other group enriches these features with the extracted contextual keywords by following the preceding procedure. Table 7 provides the experimental results with respect to both catch blocks and return-value-check snippets in our studied software systems. Metrics including precision, recall and F-score are used to evaluate the prediction accuracy. As we can see, the second group of experiments outperforms the first group of experiments, which achieves high precision of 81.1%~90.2% and high recall of 80.8%~90.4%. The results show that, in contrast to using only type information (exception type/function type), by using a simple processing of contextual information, we are able to cover as many worth-logging points as before, and decrease false positives, leading to fewer noisy, unhelpful logs at runtime. The results further demonstrate that both the type information and the contextual information are useful for logging decision.

Table 7. Prediction results

Logging decision factors	Metrics	System-A		System-B	
		Catch block	Return-value-check	Catch block	Return-value-check
Type (Exception type /Function type)	Precision	0.700	0.792	0.614	0.860
	Recall	0.785	0.724	0.812	0.766
	F-Score	0.740	0.757	0.699	0.810
Type & Contextual information	Precision	0.902	0.870	0.811	0.882
	Recall	0.899	0.899	0.808	0.904
	F-Score	0.901	0.884	0.809	0.893

The experimental results demonstrate the potential feasibility of a logging-support tool that would enable automatic logging decision/suggestion if implemented. We believe that, with some powerful NLP techniques [9], it is possible to further improve the prediction accuracy by extracting significant features from the context and semantics of the code snippets. However, it is outside the scope of this paper and we leave it for future work.

Finding 6: A classifier learnt using type information and contextual information as features achieves good prediction accuracy on whether to log for a code snippet.

6. IMPROVING CURRENT PRACTICES

In our survey, we provide an open-ended question to ask participants to describe their needs or suggestions for improving current logging practices. By analyzing their valuable feedback, we next summarize a number of directions that deserve further exploration for improving current logging practices.

Automatic logging tool for developers. As mentioned in Section 1, logging is important in system development and maintenance. However, a great challenge faced by developers is making logging decision on where to log. Neither logging too much nor logging too little is desirable. Consequently, developers are in great need of automatic logging tool support. As one participant explicitly stated, "*...need to be more automatic for writing logs, instead of writing all by myself.*" In this regard, despite its simplicity, our result on logging prediction in Section 5.2 demonstrates the potential feasibility of automatic logging. Indeed, more research efforts are needed in future work.

On-demand logging in production. Traditionally, logging statements are statically inserted to source code, and print log messages at specific fixed program locations. However, this type of logging has drawbacks. First, it may generate too many useless logs, in which valuable logs may be obscured by the heavy noises. Second, fixed program locations for logging may miss valuable information necessary for investigation. One promising direction is to log on demand. That is, each program location for logging can be dynamically enabled (or disabled) to generate logs when a specific condition is satisfied (or not satisfied). For example, at a logging point of a Remote Procedure Call (RPC), only latency above a threshold value is symptomatic for logging as a performance anomaly, whereas latencies of normal calls can be ignored. Note that DTrace [4] can be a potential solution towards on-demand logging.

End-to-end tracing. Modern software systems are generally composed of various components, which may be deployed as distributed systems. As one participant stated, "*The logs we are using are still points in the timeline, not containing calling sequences, especially for asynchronous calls*." To address this problem, end-to-end tracing can provide a detailed picture of how a request was serviced through the whole system, and thus can assist in understanding the behaviors of a complex system.

Log filtering. "*I think we are doing too much logging on redundant stuff that is useless…*" With a system scaling up, more and more logs are produced, e.g., at a rate of about 50 gigabytes (around 120-200 million lines) per hour [10]. Hence, finding useful information under the huge volume of logs is referred to colloquially as "finding the needles in the haystack". Two solutions can be used for removing redundant logs: logging on-demand and filtering logs by employing post-processing.

Log categorization. "*Logs should be categorized by source, type, and function.*" Log categorization can help achieve better log understanding and postmortem analysis. Consequently, a good logging infrastructure should offer the ability of automatic categorization. One such good example is the Unified Logging System (ULS) in Microsoft, which supports automatic tagging of logs, such as event ID and request ID. With these automatically recorded tags, logs can be easily categorized with respect to an event type or a request. However, more of such similar infrastructure features are needed. For example, since we have more and more cross workloads, it would be quite helpful for troubleshooting if logs can be correlated to different workloads (i.e., request type).

Log analysis and visualization. Log analysis and visualization are an important step in log management. "*We need more powerful tool to view log*." Due to lacking deep knowledge of the system behaviors, system administrators can benefit from automatic tool support for log search, log analysis, and log visualization. However, because logs may usually be distributed in different machines and have a huge volume, how to query and visualize large-scale logs efficiently and effectively is challenging. Existing work, such as the commercial Splunk [18] tool, and open-source Logstash [6] and Kibana [5] tool, has provided initial solutions towards this goal.

7. THREATS TO VALIDITY

Threats to internal validity. Subjectiveness in the categorization of logged snippets is inevitable due to the large manual effort involved in both the empirical study and survey. In addition, there also might be human errors in collecting statistics, etc. These threats are mitigated by double-checking all manual work. We ensure that the results are individually verified and agreed upon by at least two authors. These threats could be further reduced by involving third-party people who have experiences on logging practices to verify our results.

Threats to external validity. The threats to external validity primarily include the degree to which the subject software systems are representative of true practice. Our study was conducted on two large industrial software systems written in C#. We believe that our findings on the logging practices from a leading software company such as Microsoft should be generalizable to many other industrial software systems. Future studies on more industrial software systems (along with open source systems) and their developers can help reduce such threats to external validity.

8. RELATED WORK

Log analysis. A large body of research work focuses on postmortem analysis of logs [12], which leverages techniques such as data mining, machine learning, and static analysis to analyze system logs. A wealth of useful information has been retrieved from logs, including event correlations [11, 25], resource usage [17], component dependency [13], and causal paths [22], to facilitate their diverse usage. The first important step towards effective log analysis is to ensure log quality. Our logging practice study aims to help achieve better logging, and thus can benefit the work on log analysis.

Logging practices. Despite great importance of logging, few efforts have been spent on studying logging practices. Yuan et al. have conducted prior work [20, 21, 22] on how to perform better logging. For example, their LogEnhancer tool [22] can automatically identify important variable values and insert them into the existing logging statements, in order to enrich the content recorded in logging statements. Their study conducted for their Errlog tool [20] investigates 250 real-world failure reports and summarizes a set of code patterns that suggest additional logging points to cover failure sites. However, their study considers only buggy code samples that caused field failures and were in need of being logged; it does not consider code samples that did not cause field failures (yet) but were still in need of being logged. In addition, their study does not include code samples that were not in need of being logged. Another piece of related work is a characteristic study [21] that investigates logging-statement modifications made by developers, by mining the revision histories of four open-source software projects. In contrast, our work differs from these studies in three main aspects. (1) *Studied software systems*. Industrial software systems are mainly used for our study, whereas all of the aforementioned work is conducted on open-source systems. (2) *Study methodologies*. Instead of studying revision histories [21] (e.g., code modifications) of logging statements or studying the limited samples of buggy code with field failures and in need of logging [20], we investigate logging practices by conducting source code analysis and an empirical survey study. (3) *Research problems*. Other than studying logging-statement modifications or potential logging improvement, we focus primarily on understanding developers' logging practices with respect to *where to log*.

Logging guidelines. There are no (or at least no standard) logging guidelines for developers. However, by searching the Internet, we found a number of technical blog posts [1, 2, 3, 14, 19] on logging tips. These posts are written by developers with deep domain expertise. For example, experiences of optimal logging at Google are discussed [14]. In general, these articles provide logging tips with regard to what to log, verbosity level, logging format, etc. Our work complements these high-level logging tips with comprehensive and detailed logging characteristics in development practices.

9. CONCLUSION

To avoid logging too little or too much, developers need to make informed decisions on where to log in their logging practices during development. However, there exists no work on studying such logging practices in industry or helping developers make informed decisions. To fill this significant gap, this paper presents our studies on logging practices of two large-scale online service systems at Microsoft. We focus on studying developers' logging practices with regard to *where to log*. We provide six valuable findings on the categories of logged code snippets, factors considered for logging decisions, and the feasibility of automatic logging. In addition, some potential directions for improving current logging practices are discussed, which are valuable for further exploration. This study facilitates better understanding the current logging practices of developers and serves as the first step towards improving the logging practices in industry.

10. REFERENCES

[1] 7 Good Rules to Log Exceptions. *http://codemonkeyism.co-m/7-good-rules-to-log-exceptions/*

[2] 7 More Good Tips on Logging. *http://codemonkeyism.com-/7-more-good-tips-on-logging/*

[3] 10 Tips for Proper Application Logging. *http://www.java-codegeeks.com/2011/01/10-tips-proper-application-logging.html*

[4] B. Gregg and J. Mauro. 2011. DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD. *Prentice Hall Press*.

[5] Kibana – make sense of a mountain of logs. *http://kibana.org*

[6] Logstash – open-source log management. *http://logstash.net/*

[7] R. Ding, J. J. Shen, Q. Fu, J. G. Lou, Q. Lin, D. Zhang, and T. Xie. 2012. Healing online service systems via mining historical issue repositories. In *Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pages 318-321.

[8] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. C. Hunt. 2009. Debugging in the (very) large: ten years of implementation and experience. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, pages 106-116.

[9] C. D. Manning, and H. Schütze. 2001. Foundations of statistical natural language processing. *The MIT Press*.

[10] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai. 2013. Towards fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 24(6):1245-1255.

[11] K. Nagaraj, C. Killian, and J. Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.

[12] A. J. Oliner, A. Ganapathi, and W. Xu. 2012. Advances and challenges in log analysis. *Communications of ACM (CACM)*, 55(2):55-61.

[13] A. J. Oliner and A. Aiken. 2011. Online detection of multi-component interactions in production systems. In *Proc. of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11)*, pages 49-60.

[14] Optimal Logging. *http://googletesting.blogspot.com/2013/-06/optimal-logging.html*

[15] J. R. Quinlan. 1993. C4.5: programs for machine learning. *Morgan Kaufmann Publishers*.

[16] Roslyn. *http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx*

[17] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. 2011. Modeling and synthesizing task placement constraints in Google compute clusters. In *Proc. of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*.

[18] Splunk – log management. *http://www.splunk.com/*

[19] The Art of Logging. *http://www.codeproject.com/Articles/-42354/The-Art-of-Logging*

[20] D. Yuan, S. Park, P. Huang, Y. Liu, M. Lee, Y. Zhou, and S. Savage. 2012. Be conservative: enhancing failure diagnosis with proactive logging. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, pages 293-306.

[21] D. Yuan, S. Park, and Y. Zhou. 2012. Characterizing logging practices in open-source software. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, pages 102-112.

[22] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transaction on Computer Systems (TOCS)*, 30(1).

[23] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. 2010. SherLog: error diagnosis by connecting clues from run-time logs. In *Proc. of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*, pages 143-154.

[24] R. Sambasivan, A. Zheng, M. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *Proc. of 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, pages 4-4.

[25] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, G. Lu. 2012. LogMaster: mining event correlations in logs of large-scale cluster systems. *In Proc. of IEEE 31st Symposium on Reliable Distributed Systems (SRDS'12)*, pages 71-80.

[26] Q. Fu, J.G. Lou, Y. Wang, and J. Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. of 9th IEEE International Conference on Data Mining (ICDM'09)*, pages 149-158.

[27] J.G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. 2010. Mining invariants from console logs for system problem detection. In *Proc. of 2010 USENIX Annual Technical Conference (ATC'10)*.

[28] Q. Fu, J.G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie. 2013. Contextual analysis of program logs for understanding system behaviors. In *Proc. of 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 397-400.

[29] S. Geisser. 1993. Predictive inference: an introduction. *Chapman and Hall*, New York, ISBN 0-412-03471-9.