Characterizing and Detecting Anti-patterns in the Logging Code

Boyuan Chen[1], Zhen Ming (Jack) Jiang[2]
Software Construction, AnaLytics and Evaluation (SCALE) Lab
York University, Toronto, Canada
{chenfsd[1], zmjiang[2]}@cse.yorku.ca

***Abstract*—Snippets of logging code are output statements (e.g., LOG.info or System.out.println) that developers insert into a software system. Although more logging code can provide more execution context of the system's behavior during runtime, it is undesirable to instrument the system with too much logging code due to maintenance overhead. Furthermore, excessive logging may cause unexpected side-effects like performance slow-down or high disk I/O bandwidth. Recent studies show that there are no well-defined coding guidelines for performing effective logging. Previous research on the logging code mainly tackles the problems of *where-to-log* and *what-to-log*. There are very few works trying to address the problem of *how-to-log* (developing and maintaining high-quality logging code).**

In this paper, we study the problem of *how-to-log* by characterizing and detecting the anti-patterns in the logging code. As the majority of the logging code is evolved together with the feature code, the remaining set of logging code changes usually contains the fixes to the anti-patterns. We have manually examined 352 pairs of independently changed logging code snippets from three well-maintenance open source systems: ActiveMQ, Hadoop and Maven. Our analysis has resulted in six different anti-patterns in the logging code. To demonstrate the value of our findings, we have encoded these anti-patterns into a static code analysis tool, LCAnalyzer. Case studies show that LCAnalyzer has an average recall of 95% and precision of 60% and can be used to automatically detect previously unknown anti-patterns in the source code. To gather feedback, we have filed 64 representative instances of the logging code anti-patterns from the most recent releases of ten open source software systems. Among them, 46 instances (72%) have already been accepted by their developers.

***Keywords*-anti-patterns; logging code; logging practices; empirical studies; software maintenance;**

I. Introduction

Logging is a common programming practice that developers use to record the runtime behavior of a software system. Logs have been used widely in industry for a variety of tasks (e.g., monitoring [1], debugging [2], remote issue resolution [3], test analysis [4], security and legal compliance [3], [5], and business decision making [6]). Logs are generated by the logging code that developers insert into the system. There are typically four components in a snippet of logging code: a logging object, a verbosity level, static texts and dynamic contents. Figure 1 shows an example. The logging object is `LOG`, the verbosity level is `debug`, the static texts are `Replaced Scanner Readers at row` and dynamic contents are `Bytes.toString(viableRow.getRow())`.

Unlike other aspects in the software development process (e.g., code refactoring [8] and release management [9]), recent empirical studies show that there are no well-established logging practices in industry [10], [11]. Developers usually need to rely on their common sense to perform their logging actions. In general, there are three challenges associated with establishing effective logging practices:

1) The problem of **where-to-log** is about deciding the appropriate logging points. Snippets of logging code can be inserted at various locations in the source code (e.g., inside the try & catch exception blocks, inside the condition blocks, etc.) to provide insights into the system's runtime behavior. However, excessive logging can bring additional maintenance overhead and cause performance slow-downs [12]. Hence, developers need to be selective when choosing the logging points.
2) The problem of **what-to-log** is about providing sufficient information in the logging code. The static texts provide a short description of the execution context and the dynamic contents indicate the current execute state. When composing a snippet of logging code, the static texts should be clear and easy to understand and the dynamic contents should be coherent and up-to-date [13], [14].
3) The problem of **how-to-log** is about developing and maintaining high quality logging code. Logging is a cross-cutting concern, as the logging code is scattered across the entire system and tangled with the feature code [15]. Although there are language extensions (e.g., AspectJ [16]) to support better modularization of the logging code, many industrial and open source systems still choose to inter-mix the logging code with the feature code [11], [17], [18]. Hence, it is difficult to

```
public void updateReaders() throws IOException {
    this.lock.writeLock().lock();
    try {
      …
      ViableRow viableRow = getNextViableRow();
      openReaders(viableRow.getRow());
      LOG.debug("Replaced Scanner Readers at row " +
        Bytes.toString(viableRow.getRow()));
    } finally {
      this.lock.writeLock().unlock();
    }
}
```

Fig. 1: An example of low quality logging code which caused crash of the HBase server due to a NullPointerException [7].

develop and maintain high quality logging code, while the system evolves.

Existing log characterization studies focus on addressing the challenges of "where-to-log" [10], [12], [19] and "what-to-log" [13], [14], [20]. There are very few works tackling the problem of "how-to-log" except partially in [18], where Yuan et al. developed a verbosity level checker to detect inconsistent verbosity levels. As there are already many lines of logging code in the open source and industrial systems, low quality logging code can hinder program understanding [14] and cause unexpected system failures [7]. Figure 1 shows a real-world bug in the logging code that caused the crash of the HBase system. Since the object `ViableRow` or the return value of the method call `viableRow.getRow()` can be null, HBase was crashed once the `NullPointerException` was thrown. Hence, in this paper, we study the problem of "how-to-log" by focusing on characterizing and detecting anti-patterns in the existing logging code. Similar to the design [21] and performance anti-patterns [22], we define *anti-patterns in the logging code* as recurrent mistakes which may hinder the understanding and maintainability of the logs.

In this paper, we have conducted a comprehensive study on characterizing anti-patterns in the logging code by manually going through more than six years of the logging code changes of three popular open source software systems (ActiveMQ, Hadoop and Maven). Our study has resulted in six anti-patterns in the logging code. To demonstrate the usefulness of our findings, we have developed a static analysis tool, called LCAnalyzer, which can automatically detect these anti-patterns. The contributions of this paper are:

1) This is the first systematic study on providing guidelines on developing and maintaining high quality logging code. Case studies show that the characterized six anti-patterns in the logging code are general and exist in the ten studied open source software systems.
2) Our static analysis tool, LCAnalyzer, yields an average recall of 95% and precision of 60% and can automatically reveal many previously unknown instances of the anti-patterns in the logging code.
3) To gather developers' perceptions on whether the anti-patterns are worth fixing, we have filed a total of 64 representative anti-pattern instances from the most recent releases of ten different open source software systems. So far, 46 instances (72%) have already been confirmed or fixed by their developers. This has demonstrated the importance and the impact of our work.
4) To support independent verification or replication of our study, we have provided a replication package [23] in this paper. This package, which contains the LCAnalyzer tool and the verified anti-pattern instances from real-world systems, can be useful for other researchers who are interested in studying the logging practices.

Paper Organization

The rest of the paper is organized as follows. Section II introduces the our process of characterizing the anti-patterns in the logging code. Section III explains the resulting anti-patterns and discusses our static analysis tool, LCAnalyzer. Section IV evaluates the performance of LCAnalyzer. Section V describes the results after applying LCAnalyzer on the most recent releases of ten different open source software systems and the initial developer feedback. Section VI presents the threats to validity. Section VII discusses the related work. Section VIII concludes this paper.

II. Our Process of Characterizing Anti-patterns in the Logging Code

We follow a grounded-theory fashion [24] to characterize anti-patterns in the logging code, since there are no prior works in this area. The majority of the logging code is changed together with the feature code for various software maintenance tasks (e.g., renaming of functions or class attributes, changing condition expressions, etc.) [17], [18]. The independent logging code changes are likely the fixes to the anti-patterns. Hence, in order to characterize the anti-patterns in the logging code, we focus on isolating and analyzing the logging code changes, which occur independently of the feature code changes. Figure 2 shows our process. First, we extract the fine-grained code changes from the historical code repositories. Second, we apply heuristics to automatically identify the extracted code changes which contain changes to the logging code. Then, we use program analysis techniques to automatically categorize the logging code changes into two types: (1) logging code changes due to co-changes in the feature code; and (2) independent logging code changes. Finally, we conduct manual analysis on the independent logging code changes to characterize the anti-patterns in the logging code.

In this paper, we will analyze the independent logging code changes from three popular open software systems: ActiveMQ, Hadoop and Maven as shown in Table I. These systems are from different application domains: ActiveMQ is a message broker middleware; Hadoop is a distributed BigData compute platform; and Maven is a client application used for build management and build automation. We pick these systems as our study subjects because of their popularity (used by millions of people worldwide) and rich development history (six to ten years). Each of the changes has been carefully peer-reviewed and discussed before they are accepted into the repository [25]. We have built a local mirror of the three systems using the online data dumps [26].

A. Extracting Fine-Grained Code Changes

First, we run J-REX [27] on the historical code repository of these three systems to extract the source code and the meta information of each commit (e.g., commit ID, commit logs, etc.). Different revisions of the same source code files are recorded separately. For example, the source code of the first and the second commits of *Foo.java* are recorded as *Foo_v1.java*, *Foo_v2.java*, respectively. Then we use ChangeDistiller (CD) [28] to extract the fine-grained source code changes between each pair of the adjacent revisions (e.g., code changes between *Foo_v1.java* and *Foo_v2.java*).

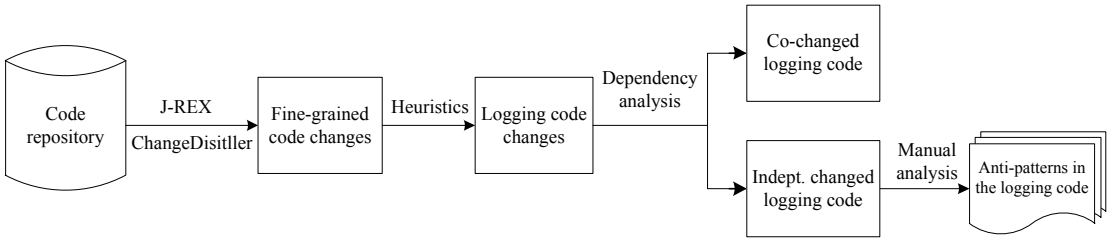

Fig. 2: Our process of characterizing anti-patterns in the logging code.

TABLE I: The three studied systems used in our characterization process.

System	Code history (begin, end)	Total revisions	Logging changes	Indep. logging changes
ActiveMQ	(12/02/2005, 10/19/2014)	9,677	2,757	571 (20.7%)
Hadoop	(01/06/2008, 10/20/2014)	25,944	10,077	2,843 (28.2%)
Maven	(12/15/2004, 11/01/2014)	29,362	3,164	943 (29.8%)

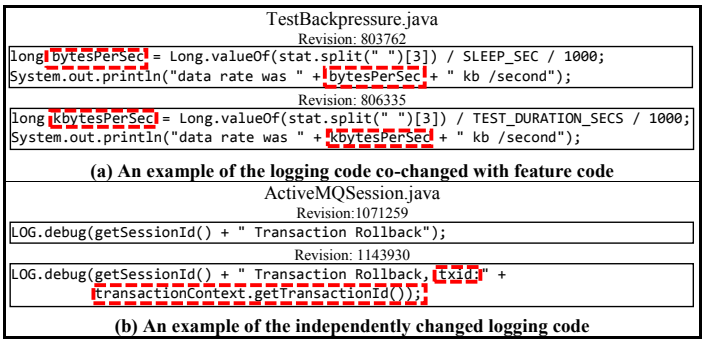

Fig. 3: An example of co-changed and independently changed logging code.

CD first parses the two file revisions into two Abstract Syntax Trees (ASTs), then compares them using the tree differencing algorithm. The output from CD is a list of fine-grained code changes (e.g., a method invocation in a function is updated or a class attribute is renamed).

B. Identifying Logging Code Changes

Similar to [10], [17], [18], [29], we apply heuristics to automatically identify the code revisions containing logging code changes. Our approach uses regular expressions to identify logging code using keywords (e.g., "`log`", "`trace`", "`debug`", etc.) in the code snippets. After the initial regular expression matching, the resulting dataset is further filtered to remove code snippets that contain mismatched words like "`login`", "`dialog`", etc. We then remove the logging code (e.g., the code snippets for logging object initializations) which do not generate logs. We achieve this by excluding code snippets that contain assignments ("="). The fourth column in Table I shows the total number of logging code changes for the three systems. For example, there are 2,757 snippets of logging code changes in ActiveMQ.

C. Categorizing Logging Code Changes

In general, there are two types of logging code changes: (1) changes in the logging code due to co-changes in the feature code; and (2) independently changed logging code. We have developed a program to automatically identify the co-changed logging code. Our program, which uses JDT [30], identifies the logging code and the feature code co-changes using program dependency analysis. We will explain our technique using a running example shown in Figure 3.

First, the program analyzes the changed feature code to identify the modified entities (e.g., updates to function `Foo` or renaming of local variable `bar`, etc.). For example, the variable `bytesPerSec` is updated to `kbytesPerSc` in Figure 3(a). Then, the program categorizes various changed components of the logging code. In Figure 3(a), only the dynamic content, variable `bytesPerSec`, is updated. Finally, the program tries to match the categorized changed components in the logging code to the modified entities in the feature code. If all the changed components in a snippet of logging code can be matched to the modified entities in the feature code, this snippet of logging code is considered as being co-changed with the feature code. For changes in the static texts, after filtering out the common words (e.g., "the", "a", etc.), we tokenize the changes into an array of words. If we can match all the changed words in the static texts and all the changed components in the dynamic contents to modified feature code entities, we consider this code snippet as co-changed logging code. For the logging code example show in Figure 3(a), as the only change in the logging code is a variable update and we can find the matching modified entity in the feature code, it is considered to be a snippet of co-changed logging code.

The remaining set of logging code changes are independently changed logging code, as one or multiple changed components cannot be matched with corresponding modified feature code entities. In Figure 3(b), method invocation `transactionContext.getTransactionId()` and static text "txid" are added to provide more execution context. As there is no corresponding feature code changes, it is a snippet of independently changed logging code. We have randomly sampled 377 instances of logging code changes

which corresponds to a 95% confidence level and $\pm 5\%$ confidence interval. Our method achieves a precision of 97%. The reason for the 3% misclassification is mainly due to some co-changed textual changes cannot matched exactly word-by-word to the modified changed entities. For example, the words "resizable" and "array" in the static texts cannot be exactly matched with the updated variable "resizeableArray". The last column in Table I shows the number of independently changed logging code and their percentage with respect to the total number of logging code changes. For example, there are only 2,843 instances of independently changed logging code in Hadoop. This corresponds to 28.2% ($\frac{2843}{10077} \times 100\%$) of the total logging code changes.

D. Manual Analysis

Our hypothesis is that such independently changed logging code is usually for addressing anti-patterns issues. To validate our hypothesis, we have manually gone through 352 pairs of independently changed logging code. This corresponds to a confidence level of 95% with a confidence interval of $\pm$ 5%. We use the stratified sampling technique [31] to ensure representative samples are selected and studied from each project. The portion of the sampled code snippets from each project is equal to the relative weight of the total number of independently changed logging code for that project. For example, there are 943 snippets of independently changed logging code in Maven out of a total of 4,357 from all three projects. Hence, 76 ($\frac{943}{4357} \times 352$) snippets are selected for Maven. For each of the selected code snippet, we have carefully compared the selected and the previous revisions to understand the rationales behind the logging code changes.

Table II shows the results of our manual analysis. It contains a total of nine reasons for independently changing the logging code. Each row in the table corresponds to one particular type of rationale. It contains the description and the number of instances found in the three studied systems. If we cannot find any instances of one rationale for that system, we indicate that cell as "-". The nine different rationales belong to two different categories: "what-to-log" and "how-to-log". As this dataset only contains the changes to the existing logging code, there are no logging code changes in the category of "where-to-log". There are a few instances in the row of "Others", as we cannot find the reasons for those changes. We have found four rationales of logging code changes in the category of "what-to-log". They take up more than 70% of the sampled code snippets. This shows the importance of studying the problem of "what-to-log" [20], [32]. Since this is not the focus of this paper, we will not further expand our analysis in this category.

There are five rationales in the category "how-to-log". Each corresponds to the different fixes to the anti-patterns in the logging code. Among these three systems, Hadoop has the largest number of instances in each rationale. This is not an indication of lower quality logging code in Hadoop, as the logs from Hadoop are actively being monitored and analyzed [33].

TABLE II: Our manual analysis results on the independently changed logging code.

Category	Rationale	Hadoop	ActiveMQ	Maven
What-to-log	Adding more context	81	27	31
	Clarifying/correcting contents	62	7	17
	Fixing typos in the static texts	14	2	11
	Removing redundant info	13	4	3
How-to-log	Checking nullable variables	12	-	-
	Removing object casting	3	-	-
	Correcting logging levels	3	-	2
	Refactoring logging code	33	4	7
	Changing output format	3	-	-
Others	-	6	2	5
Total	-	230	46	76

Rather, it is because the size of LOC and LLOC[1] in Hadoop is much bigger than ActiveMQ and Maven. In Section V, we have investigated in more details on the relation among LOC, LLOC and the number of anti-pattern instances. Log refactoring is a common rationale among all three systems. This shows that developers from all three systems are making an effort to improve the maintainability of their logging code.

To characterize the anti-patterns in the logging code, we have analyzed the code revisions before the independently changed logging code. We will describe the details of these anti-patterns in the next section.

III. Anti-patterns in the Logging Code

In the previous section, we have found five different rationales dedicated for fixing and improving the maintainability of the logging code ("How-to-log"). In this section, we will describe the anti-patterns in the logging code by studying the source code before these fixes. In general, as shown in Figure 4, there are five categories of anti-patterns in the logging code, corresponding to the five rationales that we have found. For each category of anti-patterns, we show an example code snippet extracted from real-world systems. There is one anti-pattern in each category, except in "Logging code smells", as there are two different types of logging code smells found in our manual analysis. Hence, in total, there are six anti-patterns in the logging code.

In sections III-A, III-B, III-C, III-D and III-E, we will describe the symptoms, the impacts, and the fixes of each anti-pattern. To ease explanation, we will use the code snippets shown in Figure 4 as our running examples. In section III-F, we will discuss about our logging code analysis tool, LCAnalyzer, which can automatically detect the six anti-patterns.

A. Nullable Objects

In the logging code, the dynamic contents are generated during runtime. However, in some cases, the objects used in the dynamic contents can be null. If not being careful, such snippet of logging code would cause a NullPointerException

[1]In this paper, LOC means "lines of code" and LLOC means "lines of logging code".

<table>
<tr><th>Name</th><th>Example</th></tr>
<tr><td>Nullable objects</td><td>
<pre>
if (proxy != null && invocationHandler != null && invocationHandler instanceof Closeable) {
  …
} else {
  LOG.error("Could not get invocation handler " + invocationHandler +
            " for proxy " + proxy + ", or invocation handler is not closeable.");
  …
}
</pre>
(a) RPC.java (Revision: 1177399)
</td></tr>
<tr><td>Explicit cast</td><td>
<pre>
DataNode.LOG.warn("Added missing block to memory " + (Block)diskBlockInfo);
</pre>
(b) FSDataset.java (Revision: 1202013)
</td></tr>
<tr><td>Wrong verbosity level</td><td>
<pre>
LOG.info("DEBUG --- Container FINISHED: " + containerId);
</pre>
(c) FifoScheduler.java (Revision:1178631)
</td></tr>
<tr><td rowspan="2">Logging code smells</td><td>
<u>Duplication with a method's definition (Dup1)</u>
<pre>
public String getRemoteName() {
  return channel.socket().getRemoteSocketAddress().toString();
}
…
public SocketTransceiver(SocketChannel channel) {
  this.channel = channel;
  LOG.info("open to -"+channel.socket().getRemoteSocketAddress());
}
</pre>
(d) SocketTransceiver.java (Revision:798646)
</td></tr>
<tr><td>
<u>Duplication with a local variable's definition (Dup2)</u>
<pre>
remoteAddress = s.getRemoteSocketAddress().toString();
…
LOG.warn("Invalid access token in request from "
        + s.getRemoteSocketAddress() + " for replacing block " + block);
</pre>
(e) DataXceiver.java (Revision: 802264)
</td></tr>
<tr><td>Malformed output</td><td>
<pre>
protected ClientScanner(final byte[][] columns, final byte [] startRow, …){
  …
  LOG.debug("Creating scanner over " + Bytes.toString(tableName) +
            " starting at key '" + startRow + "'");
  …
}
</pre>
(f) HTable.java (Revision: 656868)
</td></tr>
</table>

Fig. 4: Code snippet examples of anti-patterns in the logging code.

and cause the system to crash. In the else block of Figure 4(a), the object `proxy` can be null. Although this example will not cause a NullPointerException, the logging code is not informative regarding nullablity of `proxy`. The fix in this case is to check the nullability of `proxy` and handle the output differently.

B. Explicit Cast

Explicit casting informs the system to forcibly convert an object into a particular type. It might cause runtime type conversion errors and system crash. In Figure 4(b) `diskBlockInfo` was explicitly casted as the `Block` type. The fix is to remove the explicit cast and let the system decide during runtime the type of `diskBlockInfo`.

C. Wrong Verbosity Level

Many systems use verbosity level to control the types of information recorded into log files. For example, the log4j framework [34] provides multiple verbosity levels: FATAL, ERROR, WARN, INFO, DEBUG and TRACE. Each of these verbosity levels can be used for different software development activities. For example, if the verbosity level is set to be INFO, all the logs instrumented with INFO and higher levels (a.k.a., FATAL, ERROR, WARN) are printed whereas lower level logs (a.k.a., DEBUG and TRACE) are discarded. Although there are recommended guidelines on what types of information to record at each verbosity level [35], they are not strictly followed by developers. This anti-pattern may cause logging overhead and large volumes of redundant logs during log

analysis. In Figure 4(c), although the verbosity level is set to be INFO, the static texts suggest that this snippet of logging code is used for debugging purposes. The fix is to change the logging level to "DEBUG".

D. Logging Code Smells

Code smells are symptoms of bad design and implementation choices [8]. In addition to code smells, researchers define test smells to be poor design and implementation choices when developing test cases [36]. In a similar fashion as code smells and test smells, in this paper, we define logging code smells to be poor design and implementation choices when developing the logging code. As one snippet of effective logging code contains clear and easy to understand static texts, and coherent and up-to-date dynamic contents, the resulting logging code can be very long. Long logging snippets would hinder understanding and increase maintenance overhead. Hence, efforts are made to reduce the length of some long logging code snippets. Below we describe two particular anti-patterns:

- **Duplication with the Definition of Another Method (Dup1):** In Figure 4(d), the method call `channel.socket().getRemoteSocketAddress` is functionally equivalent as `getRemoteName()`. The fix is to replace this method call sequences with a shorter method call (`getRemoteName()`).
- **Duplication with the Definition of a Local Variable (Dup2):** In Figure 4(e), the local variable `remoteAddress` and the method call `s.getRemoteSocketAddress()` point to the same contents in memory. The fix here is to replace the method call sequences with the local variable `remoteAddress`.

The result after the change is functionally equivalent, but shorter logging code.

E. Malformed Output

Some objects do not have a human readable format defined. If they are printed directly, the logs can be malformed. In Figure 4(f), the variable `startRow` is a byte array, which does not have human readable format defined. The fix is to call `Bytes.toString()` method to properly format the variable `startRow` before printing.

F. LCAnalyzer

To evaluate the usefulness of our findings, we have implemented a tool called LCAnalyzer, which automatically scans the source code to detect the six aforementioned anti-patterns in the logging code. LCAnalyzer, which is a static code analyzer implemented using JDT [30], flags the anti-patterns in the logging code using ASTs. For example, to check whether one logging code snippet contains the Dup2 anti-pattern: we first identify the method which contains this logging code. Then, in this method we extract all the variable declaration statements and variable assignment statements before this logging code. If there are at least one method invocation sequences in this logging code snippet matched with one of the variable declarations or assignments, this code snippet will be flagged as containing the Dup2 anti-pattern.

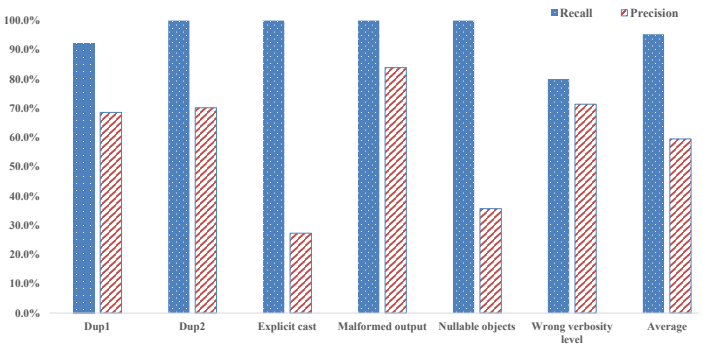

Fig. 5: The recall and precision of LCAnalyzer.

We have conducted two different case studies in this paper. In section IV, we evaluate the performance of LCAnalyzer. In section V, we have applied LCAnalyzer on the most recent releases of ten different open source software systems to evaluate the generalizability of our anti-patterns and to gather developer feedback.

IV. Evaluating the Performance of LCAnalyzer

Here we present our first case study, which is to evaluate the performance of LCAnalyzer. Sections IV-A and IV-B describe our process of evaluating the recall and the precision of LCAnalyzer, respectively.

A. Evaluating the Recall of LCAnalyzer

Constructing the Oracle Dataset: Unfortunately, there is no readily available oracle dataset which contains the verified instances of the anti-patterns in the logging code. To evaluate the performance of LCAnalyzer, we have built an oracle dataset by ourselves. The first author of this paper randomly selected a set of files from historical releases of three studied projects (ActiveMQ, Hadoop and Maven). Some of these files contain snippets of the verified anti-patterns (fixed by developers in the later revisions). In addition to these verified code snippets, the first author also manually went through every line of logging code in case there are any missing anti-patterns instances that are not addressed by the open source developers. The process was repeated until there are at least three verified instances of each type of anti-patterns. This process lasted for 2 weeks. Then, this set of files were handed over to a Master student (MSc1), who has no prior knowledge of logging code anti-patterns. Before MSc1 started examining the content of these files, he was only presented with the definitions of six anti-patterns of logging code. The anti-pattern detection algorithm and the goal of the experiment were not disclosed to him. The results between the first author and the MSc1 were compared and reconciled. The final resulting oracle dataset contains over 60 verified instances of anti-patterns.

Recall Results: We have applied LCAnalyzer on the oracle dataset. The blue dotted bars in Figure 5 show the recall results. In general, LCAnalyzer can detect 309 anti-pattern

```
LOG.info("Caught an AmazonClientException, which means the client encountered " +
         "a serious internal problem while trying to communicate with S3, " +
         "such as not being able to access the network.");
```

Fig. 6: An example code snippet that is a logging anti-pattern but LCAnalyzer fails to flag.

```
JobState oldState = getState();
try {
    getStateMachine().doTransition(event.getType(), event);
} catch (InvalidStateTransitonException e) {
  …
  LOG.info(jobId + "Job Transitioned from " + oldState + " to " + getState());
}
```

(a) Unexpected variable update

```
private void logEditsLocally(long firstTxId, int numTxns, byte[] data) {
  …
  editLog.logEdit(data.length, data);
  …
}
```

(b) Intentional logging of byte arrays

Fig. 7: Code snippets that LCAnalyzer mistakenly flags as logging anti-patterns.

instances, as the recalls among all types of anti-patterns are 80% and above. In particular, LCAnalyzer can detect all the anti-pattern instances in Dup2, Explicit cast, Malformed output, and Nullable objects.

In one logging code snippet, it includes a function call from one external library. As LCAnalyzer cannot extract the definition of that function, it misses two instances of Dup1. The reason for 80% recall in "Wrong verbosity level" is because LCAnalyzer uses an AST-based detection approach and it cannot understand the sentiment of the logging code. Figure 6 shows one such example. Although the verbosity level is INFO, the static texts ("`a serious internal problem`") suggest that the verbosity level should be WARN, ERROR or FATAL instead.

We have also calculated the average recall of LCAnalyzer, by averaging the recall values across six anti-patterns. The overall average recall for LCAnalyzer is 95%.

B. Evaluating the Precision of LCAnalyzer

To calculate the precision, we have manually examined all of our detected anti-pattern instances from our oracle dataset. The bars with red diagonal lines in Figure 5 show the precision results. There are two main reasons for the relative low precision for LCAnalyzer:

1) **Unexpected variable update**: Although some variables are defined in the same way as the method invocation sequences contained in the logging code, the values of these variables were modified. Figure 7(a) shows a falsely identified case of DUP2. `oldState` is defined as `getState()` at the beginning. However, it was later modified by the method sequences `getStateMachine().doTransition(event.getType(), event)`. Hence, `getState()` cannot be replaced by `oldState` in the logging code.
2) **Intentional logging:** Although some snippets of logging code contain the symptoms of the anti-patterns, developers perform these actions intentionally. Figure 7(b) shows an example of falsely identified instances of Malformed output. Although `data` is a byte array, developers intend to output binary data in this case.

We have also calculated the average precision of LCAnalyzer in a similar manner as the average recall. The average precision for LCAnalyzer is 60%.

TABLE III: The ten studied open source systems and their release information.

Domain	Name	Version	LOC	LLOC	AspectJ?
Server	CloudStack	4.9.0	572,461	11,433	Yes
	Hadoop	2.7.2	954,484	12,570	No
	HBase	1.2.2	433,709	7,466	No
	Tomcat	8.5.4	303,922	2,927	No
Client	ArgoUML	0.35.1	198,035	1,400	Yes
	jEdit	5.3.1	122,061	752	No
	Maven	3.3.9	78,525	400	No
Middleware/ framework	ActiveMQ	5.14.0	385,293	6,211	No
	Camel	2.17.3	803,039	7,369	Yes
	GWT	2.8.0RC2	756,374	1,550	No

The Case study on a verified dataset shows that our static logging code analysis tool, LCAnalyzer, can be used to successfully detect anti-pattern instances in the logging code. To further improve the performance of our tool, researchers can look into other techniques (e.g., data flow analysis or natural language processing) to encode and detect the anti-patterns.

V. Detecting Anti-patterns in the Open Source Software Systems

As the case study in the previous section shows satisfactory performance of LCAnalyzer, we apply this tool on the latest releases of ten different open source software systems. Our goal is to see how generalizable our characterized anti-patterns are. Table III shows the list of studied releases and their details. All these systems are actively maintained and used by millions of users worldwide. We have included the three studied systems in Section II, as we want to check whether the six anti-patterns still exist in their latest releases. In addition, we have also included other systems, especially systems not from Apache Software Foundation, to check the generalizability of our anti-patterns. Among these ten selected systems, four systems belong to the *Server* domain, three systems are from the *Client* domain and the remaining three systems are from the *Middleware/framework* domain.

The last two columns in Table III show the total lines of source code (LOC) and lines of logging code (LLOC) for each system. In general, the systems in the *Server* and the *Middleware/framework* domains contain more LOC and LLOC than systems in the *Client* domain. In addition, there is a strong correlation (a Spearman correlation value of 0.71) between LOC and LLOC indicating that larger code base implies more logging code. Three (CloudStack, ArgoUML and Camel) out of the ten studied systems also use AspectJ as part

of their logging solutions. However, these systems still contain hundreds or thousands of lines of logging code. There is no relation between the amount of logging code and whether the systems use AspectJ or not.

We have applied LCAnalyzer on the latest releases of the aforementioned ten systems. Our detection results are tabulated in Table IV. Each row corresponds to one system and each column refers to the number of instances for that anti-pattern. For example, Hadoop contains 8 instances of "Wrong verbosity level" which is 3.8% ($\frac{8}{210}*100\%$) of the total number of detected anti-patterns instances (210). In general, all ten systems contain anti-pattern instances. In particular, the two anti-patterns (Dup1 and Dup2) in the category of "Logging code smells" contain the biggest number of instances in nine out of ten systems (except Maven). Although developers have already addressed many of the past anti-pattern instances in Hadoop, ActiveMQ and Maven, there are still many instances in their latest releases.

The systems in the *Server* and *Middleware/Framework* domains contain more anti-pattern instances than systems in the *Client* domain. However, these systems also contain more lines of code. Hence, in order to investigate the relation between the amount of anti-pattern instances and the size of systems, we have calculated the Spearman correlation values between the total number of anti-pattern instances for each type of anti-pattern and LLOC. We denoted this as corr(LLOC, x) in the second last row of Table IV, in which "x" refers to a particular type of anti-pattern. For example, the correlation between LLOC and Dup1 is 0.87. This correlation value is bolded due to its statistical significance (a.k.a., p-value $<$ 0.05). Similar calculations are done between the anti-patterns and LOC. The results are shown in the last row of Table IV. There are medium to strong correlations between the number of anti-pattern instances and LLOC in five out of the six anti-patterns. This means that the larger the amount of logging code is, the harder it is to maintain. However, we do not see a clear connection between the anti-pattern instances and the overall system sizes (LOC).

Our characterized anti-patterns are general, as we can find their instances across various types of systems. There is a medium to strong correlation between the amount of logging code and the number of anti-pattern instances. As many industrial and open source systems contain large volumes of logging code [6], [10], [17], [18], this motivates the needs of further research into best practices of developing and maintaining high quality logging code.

Initial Feedback from Developers

We have selected 64 representative instances from ten open source systems mentioned above and filed online issue reports to gather developer feedback. So far, 46 instances (71.9%) have been accepted by their developers, 12 (18.7%) are under discussion, and 6 (9.4%) are rejected. There are two main reasons for the rejected instances:

```
/**
  * Dump out contents of $CWD and the
  * environment to stdout for debugging
  */
private void dumpOutDebugInfo() {
  …
  LOG.info("Dump debug output");
  …
}
```

Fig. 8: LCAnalyzer considers this snippet of logging code as an anti-pattern instance. But the Hadoop developer considered it as "intentional logging" and rejected the issue [37].

- *Intentional logging:* Figure 8 shows one such example from Hadoop. The INFO level logging code contains the `debug` message. However, the developer rejected this instance, as she indicated this was done intentionally ("*Not sure if this should change to debug level, since the function is called intentionally ...* ").
- *Developer's openness:* The developer of jEdit rejected all the filed anti-patterns instances due to his negative perceptions on the static analysis tools ("*Please do not submit code analysis tool results as bug.*").

72% of the reported instances have been accepted/confirmed by their open source developers. This has clearly demonstrated the importance and the value of this research.

VI. Threats to Validity

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

A. Internal Validity

We characterize the anti-patterns in the logging code by focusing on independently changed logging code, as these changes are likely the fixes to the existing logging code. Our approach may miss some instances of logging code fixes, as some of the co-changed logging code may be doing logging code fixing and feature code co-evolution in one commit. However, this might be a minor case, as previous studies [17], [18] show that most of the logging code co-changes are for co-evolution with feature code.

We have developed a dependency-based approach to automatically select independently changed logging code. If we can find corresponding modified entities in the feature code for all the changed components in the logging code in one code commit, this snippet of logging code is considered as co-changed with the feature code. Otherwise, it is a snippet of independently changed logging code. As our approach has yield a relatively high precision 97%, we believe that we have obtained most of the independently changed logging code.

B. External Validity

We have focused on characterizing and detecting logging anti-patterns in Java-based systems. Some of our derived findings and code anti-patterns may not be directly applicable to systems implemented in other programming languages.

TABLE IV: Our detection results on the latest release of ten open source software systems. The Spearman correlation numbers in the last two rows are shown in **bold** if they are statistically significant ($p < 0.05$).

Category	System	Wrong verbosity level	Dup1	Dup2	Nullable objects	Malformed outputs	Explicit cast	Total anti-patterns	Total log lines
Server	ClouStack	0 (0.0%)	125 (42.7%)	91 (31.1%)	17 (5.8%)	57 (19.5%)	3 (1.0%)	293(2.6%)	11,433
	Hadoop	8 (3.8%)	113 (53.8%)	59 (28.1%)	10 (4.8%)	5 (2.4%)	15 (7.1%)	210 (0.7%)	28,616
	HBase	0 (0.0%)	95 (61.7%)	36 (23.4%)	7 (4.5%)	15 (9.7%)	1 (0.6%)	154(2.1%)	7,466
	Tomcat	2 (6.5%)	8 (25.8%)	12 (38.7%)	3 (9.7%)	3 (9.7%)	3 (9.7%)	31(1.1%)	2,927
Client	ArgoUML	0 (0.0%)	4 (20.0%)	6 (30.0%)	8 (40.0%)	0 (0.0%)	2 (10%)	20(1.4%)	1,400
	jEdit	0 (0.0%)	2 (20.0%)	5 (50.0%)	3 (30.0%)	0 (0.0%)	0 (0.0%)	10 (1.3%)	752
	Maven	4 (28.6%)	8 (57.1%)	1 (7.1%)	1 (7.1%)	0 (0.0%)	0 (0.0%)	14 (3.5%)	400
Middleware/ Framework	ActiveMQ	4 (5.3%)	31 (41.3%)	26 (34.7%)	10 (13.3%)	3 (4.0%)	1 (1.3%)	75(1.2%)	6,211
	Camel	2 (2.7%)	21 (28.4%)	21 (28.4%)	5 (6.8%)	20 (27.0%)	5 (6.8%)	74(1.0%)	7,369
	GWT	4 (7.0%)	41 (71.9%)	11 (19.3%)	1 (1.8%)	0 (0.0%)	0 (0.0%)	57(3.7%)	1,550
corr(LLOC, x)	-	0.27	**0.87**	**0.90**	**0.78**	**0.65**	**0.71**	-	-
corr(LOC, x)	-	0.53	0.61	0.54	0.29	0.29	**0.70**	-	-

However, we feel that our history-based approach to characterize logging anti-patterns is generic and can be used to study the anti-patterns in the logging code developed in other programming languages (e.g., C or .NET).

We have characterized six different anti-patterns in the logging code by studying the historical changes in three popular open source systems: ActiveMQ, Hadoop and Maven in Apache Software Foundation. We have picked these systems due to the following two reasons: (1) they come from different application domains (*Middleware*, *Server* and *Client*) and (2) these three systems are actively maintained. All the committed source code has been carefully peer-reviewed [25]. We start our anti-pattern characterization process from Hadoop, and then move on to ActiveMQ and Maven. We have noticed that no additional anti-patterns have been identified in ActiveMQ and Maven. Furthermore, the case study in Section V shows that these anti-patterns also exist in many other non-Apache open source systems. This gives us some confidence in terms of the generalizablility of the anti-patterns. However, our catalog of anti-patterns may not be complete. We plan to address this problem by looking into other analysis approaches and studying more systems in the future.

C. Construct Validity

As there is no existing benchmarking dataset for anti-patterns in the logging code, we have built an oracle dataset ourselves to evaluate the performance of LCAnalyzer. The dataset, which has been compiled and verified by two different persons, contains the verified instances of anti-patterns in the logging code. These two persons include the first author of the paper and another Master student who has no prior knowledge of logging code anti-patterns. Our process of building the oracle dataset is similar to many other papers (e.g., [38], [39], [40]). However, we acknowledge that our oracle dataset may be incomplete (a.k.a., missing some anti-pattern instances).

VII. Related Work

There are three areas of research related to this paper: (1) empirical studies on the existing logging practices, (2) tools for understanding, developing and maintaining logging code, and (3) code smells and refactoring.

A. Empirical studies on the existing logging practices

Industry studies show that there are no well-defined best practices to guide developers on developing and maintaining the logging code [10], [11]. Hence, it is worthwhile to study the logging practices of existing systems and learn from them.

Yuan et al. [17], [18] conducted a quantitative study on the logging code of several large-scale open source software systems. They have found that developers are constantly making an effort to improve the quality of their logging code. Shang et al. [29] studied the relation between the spread of the logging code and system quality. They found that log related metrics (e.g., log density) were strong predictors of post release defects. Kabinna et al. [32] performed a quantitative study on the rationale of logging code changes. They built a data mining classifier to model the historical logging code changes. Their study showed that file ownership, developer experience, log density, and SLOC are important factors for deciding whether a snippet of logging code needs to be changed. Kabinna et al. [13] studied the migrations of logging libraries of several systems. They found that systems migrate their logging libraries to gain additional functionalities, to improvement maintainability, and to enhance performance. Over 70% of the migrated systems suffer from migration bugs afterwards.

Our work is different from the above works, as we focus on studying the rationales behind independently changed logging code. Our qualitative study on the logging code has resulted in six anti-patterns in the logging code, which can be used to detect and improve the quality of existing logging code.

B. Tools for better understanding, developing and maintaining logging code

We further divide this area of research into three categories:

- *Where-to-log* tackles the problem of where to place the logging points. Yuan et al. [2] proposed a program analysis-based approach to inferring additional logging

points to assist debugging. Fu et al. [10], [19] used a data mining-based approach to automatically identifying the important factors impacting the locations of the logging points. Ding et al. [12] used a constraint solving-based method to determine, during runtime, the optimal logging points which incur minimum performance overhead but maximum runtime information.

- *What-to-log* tackles the problem of adding sufficient runtime execution information. Yuan et al. [20] proposed a program analysis-based approach to suggesting additional variables to be added into the existing logging points to facilitate error diagnosis.
- *How-to-log* tackles the problem of developing and maintaining high quality logging code. Kiczales et al. [15] proposed Aspect Oriented Programming (AOP) to automatically develop and maintain the logging code. However, there are still many open source and industry systems which place the logging code along side with the feature code. In this aspect, only Yuan et al. partially studied this problem in [18]. They used a clone detection-based approach to automatically identifying inconsistent verbosity levels. Our paper is the first work which systematically studies the problem of "How-to-log" by characterizing and detecting the anti-patterns in the logging code.

C. Code smells and refactoring

Code smells are symptoms of bad design and implementation choices [8]. Code smell can increase change/fault proneness and decrease program understandability [41], [42]. There are various approaches to automatically detecting code smells in the source code (e.g., AST-based approach [40], [43]), history-based approach [39], [44] and text mining-based approach [45]).

In addition to code smells, researchers define test smells to be poor design and implementation choices when developing test cases [36]. Studies also show that test smells have a strong negative impact on program comprehension and software maintenance [46]. Van Rompaey et al. [47] proposed a metric-based technique to detect test smells.

In this paper, we have found that one of the top rationales of independently changed logging code is about log refactoring. Hence, we define the logging code smells as poor design and implementation choices when developing the logging code. We have proposed two symptoms of the logging code smells: Dup1 and Dup2. Both symptoms are to address the problems of: (1) duplication in logging code, and (2) long logging code. This paper is the first work which proposes the idea of logging code smells and their symptoms.

VIII. Conclusions and Future Work

Developers instrument their systems with logging code to gain insights about the systems' runtime behaviour. It is challenging to develop and maintain high quality logging code due to the lack of well-defined coding guidelines. In this paper, we have characterized six anti-patterns in the logging code by carefully studying the development history of three open source software systems from different application domains: ActiveMQ, Hadoop and Maven. To demonstrate the usefulness of our findings, we have developed LCAnalyzer, which statically scans through the source code searching for anti-pattern instances. Case studies show that LCAnalyzer, which has a high recall (95%) and a satisfactory precision (60%), can detect many anti-pattern instances in ten different open source software systems. We have filed a few selective instances to their issue tracking systems. So far, we have received very positive feedback from the Hadoop and the Tomcat developers.

Verifiability: We have provided a data package to support independent verification or replication of our study [23]. The package consists of three sets of data:

1) The **Characterization Dataset** contains the set of manually examined independently changed logging code snippets and our analysis results.
2) The **Oracle Dataset** contains the list of source code files and our verified instances of anti-patterns in the logging code.
3) The **Anti-pattern Instances Dataset** contains the list of anti-pattern instances detected by LCAnalyzer for the recent releases of ten open source software systems. In addition, the filed anti-pattern instances, the issue IDs and their current status are also included.

In the future, we want to improve the precision of LCAnalyzer by incorporating additional analysis techniques (e.g., data flow or natural language processing). In addition, we plan to expand our anti-pattern catalog of logging code by studying the development history of other systems (e.g., smartphone applications or industry systems). Finally, we also want to solicit more feedback from developers in terms of what are other good/bad logging code practices.

References

[1] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, "An exploratory study of the evolution of communicated information about the execution of large software systems," *Journal of Software: Evolution and Process*, 2014.

[2] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be Conservative: Enhancing Failure Diagnosis with Proactive Logging," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[3] A. Oliner, A. Ganapathi, and W. Xu, "Advances and Challenges in Log Analysis," *Communications of ACM*, 2012.

[4] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated Performance Analysis of Load Tests," in *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, 2009.

[5] "Summary of Sarbanes-Oxley Act of 2002," http://www.soxlaw.com/, Last accessed 08/26/2015.

[6] T. Barik, R. DeLine, S. Drucker, and D. Fisher, "The Bones of the System: A Case Study of Logging and Telemetry at Microsoft," in *Companion Proceedings of the 38th International Conference on Software Engineering)*, 2016.

[7] "HBASE-750: NPE caused by StoreFileScanner.updateReaders," https://issues.apache.org/jira/browse/HBASE-750/, Last accessed: 08/26/2016.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[10] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where Do Developers Log? An Empirical Study on Logging Practices in Industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014.

[11] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry Practices and Event Logging: Assessment of a Critical Software Development Process," in *Companion Proceedings of the 37th International Conference on Software Engineering*, 2015.

[12] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A Cost-aware Logging Mechanism for Performance Diagnosis," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2015.

[13] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging Library Migrations: A Case Study for the Apache Software Foundation Projects," in *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, 2016.

[14] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding Log Lines Using Development Knowledge," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, 1997.

[16] "The AspectJ Project," https://eclipse.org/aspectj/, Last accessed: 08/26/2016.

[17] B. Chen and Z. M. Jiang, "Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation," *Empirical Software Engineering*, 2016.

[18] D. Yuan, S. Park, and Y. Zhou, "Characterizing Logging Practices in Open-source Software," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[19] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to Log: Helping Developers Make Informed Logging Decisions," in *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[20] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[21] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[22] C. U. Smith and L. G. Williams, "Software performance antipatterns," in *Proceedings of the 2nd International Workshop on Software and Performance (WOSP)*, 2000.

[23] B. Chen and Z. M. Jiang, "The dataset for our study on characterizing and detecting anti-patterns in the logging code," http://nemo9cby.github.io/icse2017.html, Last accessed 08/26/2016.

[24] K. Charmaz, *Constructing grounded theory : a practical guide through qualitative analysis*. Sage Publications, 2006.

[25] P. C. Rigby, D. M. German, and M.-A. Storey, "Open Source Software Peer Review Practices: A Case Study of the Apache Server," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, 2008.

[26] "Dumps of the ASF Subversion repository," http://svn-dump.apache.org/, Last accessed 08/26/2016.

[27] W. Shang, Z. M. Jiang, B. Adams, and A. Hassan, "MapReduce as a general framework to support research in Mining Software Repositories (MSR)," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009.

[28] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering (TSE)*, 2007.

[29] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, 2015.

[30] "JDT Java Development Tools." https://eclipse.org/jdt/, Last accessed: 08/26/2016.

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

[32] S. Kabinna, W. Shang, C.-P. Bezemer, and A. E. Hassan, "Examining the Stability of Logging Statements," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[33] "Nagios Log Server - Monitor and Manage Your Log Data," https://exchange.nagios.org/directory/Plugins/Log-Files, Last accessed: 08/26/2016.

[34] "LOG4J a logging library for Java," http://logging.apache.org/log4j/1.2/, Last accessed: 08/26/2016.

[35] "Apache JCL Best Practices," http://commons.apache.org/proper/commons-logging/guide.html#JCL_Best_Practices, Last accessed: 08/26/2016.

[36] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring Test Code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001.

[37] "YARN-5506: Inconsistent logging content and logging level for distributed shell," https://issues.apache.org/jira/browse/YARN-5506, Last accessed: 08/26/2016.

[38] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling Code Clone Detection to Big-code," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

[39] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining Version Histories for Detecting Code Smells," *IEEE Transactions on Software Engineering (TSE)*, 2015.

[40] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering (TSE)*, 2010.

[41] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering (WCRE)*, 2009.

[42] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[43] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering (TSE)*, 2009.

[44] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Detecting bad smells in source code using change history information," in *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, 2013.

[45] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *IEEE Transactions on Software Engineering (TSE)*, 2014.

[46] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Software Engineering*, 2015.

[47] B. V. Rompaey, B. D. Bois, S. Demeyer, and M. Rieger, "On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test," *IEEE Transactions on Software Engineering*, 2007.