DLFinder: Characterizing and Detecting Duplicate Logging Code Smells

Zhenhao Li, Tse-Hsun (Peter) Chen, Jinqiu Yang and Weiyi Shang
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{l_zhenha, peterc, jinqiuy, shang}@encs.concordia.ca

***Abstract*—Developers rely on software logs for a wide variety of tasks, such as debugging, testing, program comprehension, verification, and performance analysis. Despite the importance of logs, prior studies show that there is no industrial standard on how to write logging statements. Recent research on logs often only considers the appropriateness of a log as an individual item (e.g., one single logging statement); while logs are typically analyzed in tandem. In this paper, we focus on studying duplicate logging statements, which are logging statements that have the same static text message. Such duplications in the text message are potential indications of logging code smells, which may affect developers' understanding of the dynamic view of the system. We manually studied over 3K duplicate logging statements and their surrounding code in four large-scale open source systems: Hadoop, CloudStack, ElasticSearch, and Cassandra. We uncovered five patterns of duplicate logging code smells. For each instance of the code smell, we further manually identify the problematic (i.e., require fixes) and justifiable (i.e., do not require fixes) cases. Then, we contact developers in order to verify our manual study result. We integrated our manual study result and developers' feedback into our automated static analysis tool, DLFinder, which automatically detects problematic duplicate logging code smells. We evaluated DLFinder on the four manually studied systems and two additional systems: Camel and Wicket. In total, combining the results of DLFinder and our manual analysis, we reported 82 problematic code smell instances to developers and all of them have been fixed.**

***Keywords*-log, code smell, duplicate log, static analysis, empirical study**

I. Introduction

Software logs are widely used in software systems to record system execution behaviors. Developers use the generated logs to assist in various tasks, such as debugging [18], [49], [51], testing [13], [15], [22], program comprehension [19], [41], system verification [6], [9], and performance analysis [14], [47]. A logging statement (i.e., code that generates a log) contains a static message, to-be-recorded variables, and log verbosity level. As an example, a logging statement may be written as *logger.error("Interrupted while waiting for fencing command: " + cmd);*. In this example, the static text message is *"Interrupted while waiting for fencing command: "*, and the dynamic message is from the variable *cmd*, which records the command that is being executed. The logging statement is at the *error* level, which is the level for recording failed operations [2].

Even though developers have been analyzing logs for decades [24], there exists no industrial standard on how to write logging statements [18], [35]. Prior studies often focus on recommending where logging statements should be added into the code (i.e., *where-to-log*) [53], [54], and what information should be added in logging statements (i.e., *what-to-log*) [36], [41], [51]. A few recent studies [12], [20] aim to detect potential problems in logging statements. However, these studies often only consider the appropriateness of one single logging statement as an individual item; while logs are typically analyzed in tandem [14], [51]. In other words, we consider that the appropriateness of a log is also influenced by other logs that are generated in system execution.

```
...
} catch (AlreadyClosedException closedException) {
     s_logger.warn("Connection to AMQP service is lost.");
} catch (ConnectException connectException) {
     s_logger.warn("Connection to AMQP service is lost.");
}
...
```

Fig. 1. An example of duplicate logging code smell that we detected in CloudStack. The duplicate logging statements in the two *catch* blocks contain insufficient information (e.g., no exception type or stack trace) to distinguish what may be the occurred exception.

In particular, an intuitive case of such influence is duplicate logs, i.e., multiple logs that have the same text message. Even though each log itself may be impeccable, duplicate logs may affect developers' understanding of the dynamic view of the system. For example, as shown in Figure 1, there are two logging statements in two different *catch* blocks, which are associated with the same *try* block. These two logging statements have the same static text message and do not include any other error-diagnostic information. Thus, developers cannot easily distinguish what is the occurred exception when analyzing the produced logs. Since developers rely on logs for debugging and program comprehension [41], such duplicate logging statements may negatively affect developers' activities in maintenance and quality assurance.

To help developers improve logging practices, in this paper, we focus on studying duplicate logging statements in the source code. We conducted a manual study on four large-scale open source systems, namely Hadoop, CloudStack, ElasticSearch, and Cassandra. We first used static analysis to identify all duplicate logging statements, which are defined as two or more logging statements that have the same static text message. We then manually studied all the (over 3K) identified duplicate logging statements and uncovered five

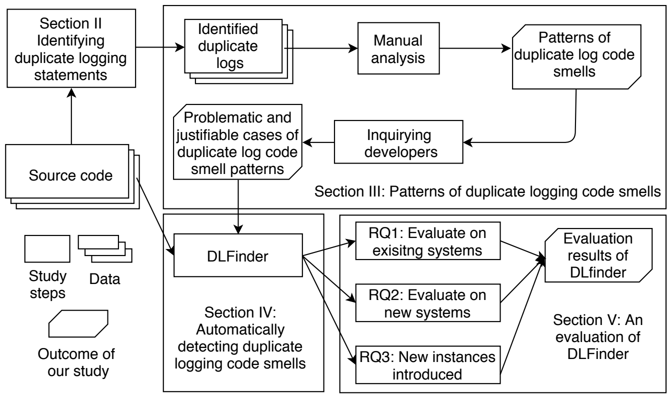

Section II Identifying duplicate logging statements
Identified duplicate logs
Manual analysis
Patterns of duplicate log code smells
Problematic and justifiable cases of duplicate log code smell patterns
Inquirying developers
Source code
Section III: Patterns of duplicate logging code smells
Study steps
Data
Outcome of our study
DLFinder
Section IV: Automatically detecting duplicate logging code smells
RQ1: Evaluate on exisitng systems
RQ2: Evaluate on new systems
RQ3: New instances introduced
Evaluation results of DLfinder
Section V: An evaluation of DLFinder

Fig. 2. The overall process of our study. The term "duplicate logging statements" is referred as "duplicate logs" for simplification.

patterns of *duplicate logging code smells*. We follow prior code smell studies [8], [17], and consider duplicate logging code smell as a "surface indication that usually corresponds to a deeper problem in the system". However, *not* all of the duplicate logging code smell are problematic and require fixes (i.e., *problematic duplicate logging code smells*). In particular, context (e.g., surrounding code and usage scenario of logging) may play an important role in identifying fixing opportunities. Hence, we further categorized duplicate logging code smells into *problematic* or *justifiable* cases. In addition to our manual analysis, we sought confirmation from developers on the manual analysis result: For the problematic duplicate logging code smells, we reported them to developers for fixing. For the justifiable ones, we communicated with developers for discussion (e.g., emails or posts on developers' forums).

We implemented a static analysis tool, DLFinder, to automatically detect *problematic* duplicate logging code smells. DLFinder leverages the findings from our manual study, including the uncovered patterns of duplicate logging code smells and the categorization on problematic and justifiable cases. We evaluated DLFinder on six systems: four are from the manual study and two are additional systems (Camel and Wicket). We also applied DLFinder on the updated versions of the four manually studied systems. The evaluation shows that the uncovered patterns of the duplicate logging code smells also exist in the two additional systems, and duplicate logging code smells may be introduced over time. An automated approach such as DLFinder can help developers avoid duplicate logging code smells as systems evolve.

In total, we reported 82 instances of duplicate logging code smell to developers and all the reported instances are fixed.

Figure 2 shows the overall process of this paper.

In summary, this paper makes the following contributions:

- We uncovered five patterns of duplicate logging code smells through an extensive manual study on over 3K duplicate logging statements.
- We presented a categorization of duplicate logging code smells (i.e., problematic or justifiable), based on both our manual assessment (i.e., studying the logging statement and its surrounding code) and developers' feedback.
- We proposed DLFinder, a static analysis tool that integrates our manual study result and developers' feedback to detect problematic duplicate logging code smells. We evaluated DLFinder for both the accuracy and generalization (i.e., on new systems and on the newer versions as systems evolve).
- We reported 82 instances of problematic duplicate logging code smells to developers (DLFinder is able to detect 72 of them), and all of the reported instances are fixed.

Paper organization. The rest of the paper is organized as follows. Section II describes how we prepare the data for manual study (i.e., duplicate logging statements) and the studied systems. Section III discusses the process and the results of our manual study, and also developers' feedback on our results. Section IV discusses the implementation details of DLFinder. Section V evaluates DLFinder for both the accuracy and generalization. Section VI discusses the threats to validity of our study. Section VII surveys related work. Finally, Section VIII concludes the paper.

II. Identifying Duplicate Logging Statements for Manual Study

In this section, we describe how we define duplicate logging statements and how we identify them for conducting a manual study. We also introduce the studied systems.

Definition and how to identify duplicate logging statements. We define duplicate logging statements as logging statements that have the identical static text messages. We focus on studying the log message because such semantic information is crucial for log understanding and system maintenance [41], [50]. As an example, the two following logging statements are considered duplicate: "*Unable to create a new ApplicationId in SubCluster*" + ***subClusterId.getId()***, and "*Unable to create a new ApplicationId in SubCluster*" + ***id***.

To prepare for a manual study, we identify duplicate logging statements by analyzing the source code with static analysis. In particular, the static text message of each logging statement is built by concatenating all the strings (i.e., constants and values of string variables) and abstractions of the non-string variables. We also extract information to support the manual analysis, such as the types of variables that are logged, and the log level (i.e., *fatal*, *error*, *warn*, *info*, *debug*, or *trace*). Log levels represent the verbosity level of the log and can be used to reduce logging overheads in production (e.g., only logging *info* level or above) [29], [50]. If two or more logging statements have the same static text message, they are identified as duplicate logging statements. We exclude logging statements with only one word in the static text message since those logging statements usually do not contain much static information, and are usually used to record the value of a dynamic variable during system execution.

Studied systems. We identify duplicate logging statements from four large-scale open source Java systems: Hadoop, CloudStack, ElasticSearch, and Cassandra, which are commonly used in prior studies for log-related research [12], [20], [28]. Table I shows the statistics of the studied systems. The

TABLE I
AN OVERVIEW OF THE STUDIED SYSTEMS IN OUR MANUAL STUDY.

System	Version	Release date	LOC	Num. of logs	Num. of dup. logs	Num. of dup. log sets	Med. words in dup. logs	Med. words in non-dup. logs
Hadoop	3.0.0	Nov. 2017	2.69M	5.3K	496 (9%)	217	6	6
CloudStack	4.9.3	Aug. 2017	1.18M	11.7K	2.3K (20%)	865	8	8
ElasticSearch	6.0.0	Nov. 2017	2.12M	1.7K	94 (6%)	40	6	7
Cassandra	3.11.1	Oct. 2017	358K	1.6K	113 (7%)	46	7	7

studied systems use the widely used Java logging libraries (e.g., Log4j [2] and SLF4J [3]). Hadoop is a distributed computing framework, which is composed of four subsystems: Hadoop Common, Hadoop Distributed File System, YARN, and MapReduce. CloudStack is a cloud computing platform, ElasticSearch is a distributed search engine, and Cassandra is a NoSQL database system. These systems belong to different domains and are well maintained. In our study, we study all Java source code files in the main branch of each system and exclude test files, since we are more interested in studying duplicate logging statements that may affect log understanding in production. In general, we find that there is a non-negligible number of duplicate logging statements in the studied systems (6% to 20%). The median number of words in the duplicate logging statements are similar to that of non-duplicate logging statements (i.e., both range from 6 to 8 words), which shows that they have a similar level of semantic information (in terms of the number of words).

III. PATTERNS OF DUPLICATE LOGGING CODE SMELLS

In this section, we conduct a manual study to uncover patterns of potential code smells that may be associated with duplicate logging statements (i.e., *duplicate logging code smells*). Similar to prior code smell studies [8], [17], we consider duplicate logging code smells as a *"surface indication that usually corresponds to a deeper problem in the system"*. Such duplicate logging code smells may be indications of logging problems that require fixes.

Furthermore, we categorize each code smell instance as either problematic (i.e., require fixes) or justifiable (i.e., do not require fixes), by understanding the surrounding code. Not every duplicate logging code smell is problematic. Intuitively, one needs to consider the code context to decide whether a code smell instance is problematic and requires fixes. As shown in prior studies [18], [28], [54], logging decisions, such as log messages and log levels, are often associated with the structure and semantics of the surrounding code. In addition to the manual analysis by the authors, we also ask for developers' feedback regarding both the problematic and justifiable cases. By providing a more detailed understanding of code smells, we may better assist developers to improve logging practices and inspire future research.

Manual study process. We conduct a manual study by analyzing all the duplicate logging statements identified from the studied systems. In total, we studied 1,168 sets of duplicate logging statements in the four studied systems (more than 3K logging statements in total; each set contains two or more logging statements with the same static message).

The process of our manual study involves five phases:

- Phase I: The first two authors manually studied 289 randomly sampled (based on 95% confidence level and 5% confidence interval [7]) sets of duplicate logging statements and the surrounding code to derive an initial list of duplicate logging code smell patterns. All disagreements were discussed until a consensus was reached.
- Phase II: The first two authors *independently* categorized *all* of the 1,168 sets of duplicate logging statements to the derived patterns in Phase I. We did not find any new patterns in this phase. The results of this phase have a Cohens kappa of 0.806, which is a substantial-level of agreement [31].
- Phase III: The first two authors discussed the categorization results obtained in Phase II. All disagreements were discussed until a consensus was reached.
- Phase IV: The first two authors further studied all logging code smell instances that belong to each pattern in order to identify justifiable cases of the logging code smell that may not need fixes. The instances that do not belong to the category of justifiable are considered potentially problematic and may require fixes.
- Phase V: We verified both the problematic instances of logging code smells and the justifiable ones with the developers by creating issue reports and pull requests, sending emails, or posting our findings on developers forums such as Stack Overflow. In particular, we reported every instance that we believe to be problematic (i.e., require fixes). We also reported a number of instances for each justifiable category.

Results. In total, we uncovered five patterns of duplicate logging code smells. Table II lists the uncovered code smell patterns and the corresponding examples. Table III shows the number of problematic code smell instances for each pattern. Below, we discuss each pattern according to the following template:

Description: A description of the pattern of duplicate logging code smell.

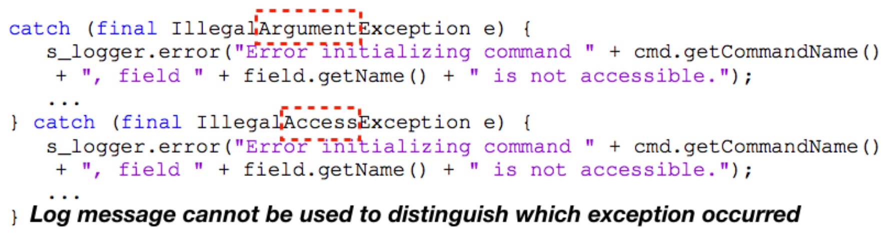
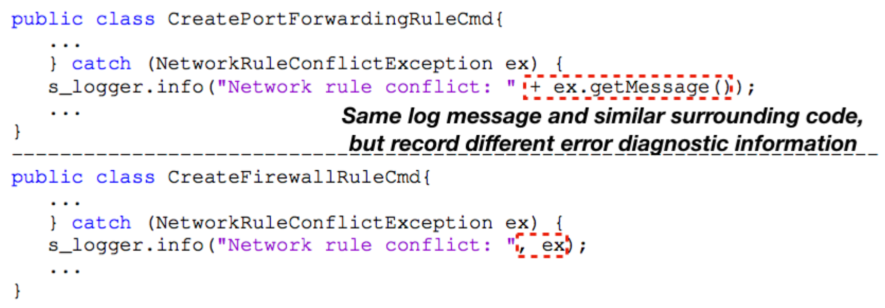
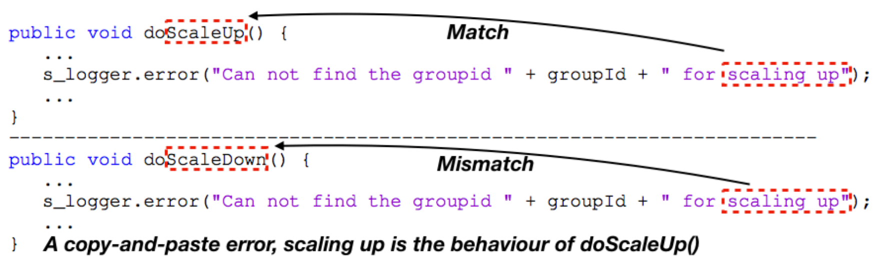
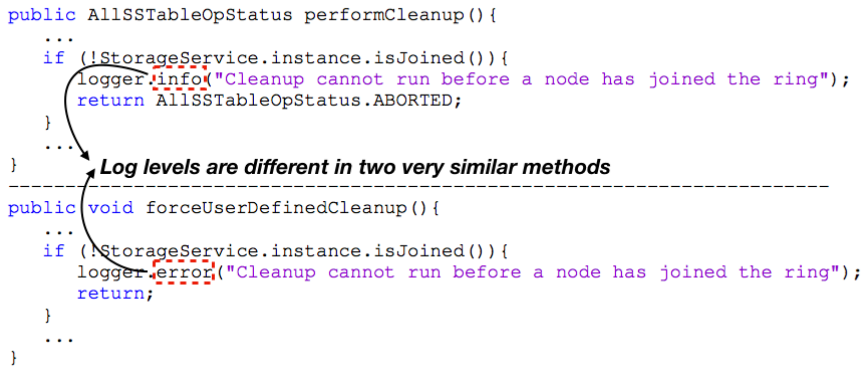
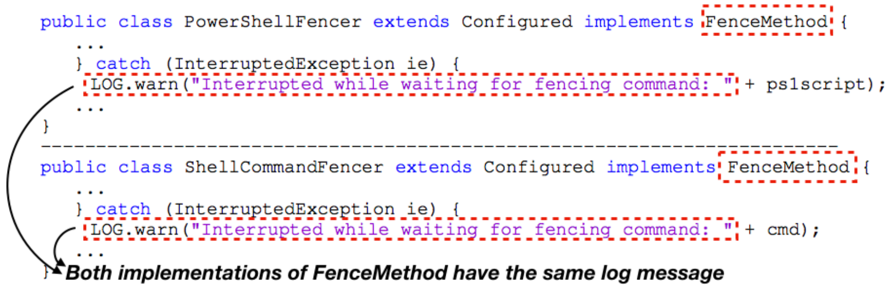
Example: An example of the pattern.

Code smell instances: Discussions on the code smell instances that we manually found. We also discuss the justifiable cases if we found any.

Developers' feedback: A summary of developers' feedback on both the problematic and justifiable cases.

Pattern 1: Inadequate information in catch blocks (IC).
Description. Developers usually rely on logs for error diagnostics when exceptions occur [48]. However, we find that sometimes, duplicate logging statements in different *catch*

TABLE II
PATTERNS OF DUPLICATE LOGGING CODE SMELLS AND CORRESPONDING EXAMPLES.

Name	Example
Inadequate information in catch blocks (IC)	
Inconsistent error-diagnostic information (IE)	
Log message mismatch (LM)	
Inconsistent log level (IL)	
Duplicate log in polymorphism (DP)	

Inadequate information in catch blocks (IC):

```
catch (final IllegalArgumentException e) {
   s_logger.error("Error initializing command " + cmd.getCommandName()
    + ", field " + field.getName() + " is not accessible.");
   ...
} catch (final IllegalAccessException e) {
   s_logger.error("Error initializing command " + cmd.getCommandName()
    + ", field " + field.getName() + " is not accessible.");
   ...
}
```

Log message cannot be used to distinguish which exception occurred

Inconsistent error-diagnostic information (IE):

```
public class CreatePortForwardingRuleCmd{
   ...
   } catch (NetworkRuleConflictException ex) {
   s_logger.info("Network rule conflict: " + ex.getMessage());
   ...
}
_________________________________________________________________________
public class CreateFirewallRuleCmd{
   ...
   } catch (NetworkRuleConflictException ex) {
   s_logger.info("Network rule conflict: ", ex);
   ...
}
```

Same log message and similar surrounding code, but record different error diagnostic information

Log message mismatch (LM):

```
public void doScaleUp() {
   ...
   s_logger.error("Can not find the groupid " + groupId + " for scaling up");
   ...
}
-------------------------------------------------------------------------
public void doScaleDown() {
   ...
   s_logger.error("Can not find the groupid " + groupId + " for scaling up");
   ...
}
```

Match

Mismatch

A copy-and-paste error, scaling up is the behaviour of doScaleUp()

Inconsistent log level (IL):

```
public AllSSTableOpStatus performCleanup(){
   ...
   if (!StorageService.instance.isJoined()){
      logger.info("Cleanup cannot run before a node has joined the ring");
      return AllSSTableOpStatus.ABORTED;
   }
   ...
}
-------------------------------------------------------------------------
public void forceUserDefinedCleanup(){
   ...
   if (!StorageService.instance.isJoined()){
      logger.error("Cleanup cannot run before a node has joined the ring");
      return;
   }
   ...
}
```

Log levels are different in two very similar methods

Duplicate log in polymorphism (DP):

```
public class PowerShellFencer extends Configured implements FenceMethod {
   ...
   } catch (InterruptedException ie) {
   LOG.warn("Interrupted while waiting for fencing command: " + ps1script);
   ...
}
-------------------------------------------------------------------------
public class ShellCommandFencer extends Configured implements FenceMethod {
   ...
   } catch (InterruptedException ie) {
   LOG.warn("Interrupted while waiting for fencing command: " + cmd);
   ...
}
```

Both implementations of FenceMethod have the same log message

TABLE III
NUMBER OF PROBLEMATIC INSTANCES VERIFIED BY OUR MANUAL STUDY AND DEVELOPERS' FEEDBACK (Prob.), AND TOTAL NUMBER OF INSTANCES (TOTAL) INCLUDING NON-PROBLEMATIC INSTANCES.

	IC		IE		LM		IL		DP	
	Prob.	Total	Prob.	Total	Prob.	Total	Prob.	Total	Prob.	Total
Hadoop	5	5	0	0	9	9	0	17	27	27
CloudStack	8	8	4	14	27	27	0	47	107	107
ElasticSearch	1	1	0	5	1	1	0	9	3	3
Cassandra	1	1	0	1	0	0	0	3	2	2
Total	15	15	4	20	37	37	0	76	139[1]	139

[1] Developers acknowledged the problem but we did not report all the instances, because systematic refactoring of DP would require supports from logging libraries.

blocks of the same *try* block may cause debugging difficulties since the logs fail to tell which exception occurred.
Example. As shown in Table II, in the `ParamProcessWorker` class in CloudStack, the *try* block contains two *catch* blocks; however, the log messages in these two *catch* blocks are identical. Since both the exception message and stack trace are not logged, once one of the two exceptions occurs, developers may encounter difficulties in finding the root causes and determining the occurred exception.
Code smell instances. After examining all the instances of IC, we find that all of them are potentially problematic that require fixes. For all the instances of IC, none of the exception type, exception message, and stack trace are logged.
Developers' feedback. We reported all the problematic instances of IC (15 instances) by using pull requests. All the pull requests were accepted by developers and the fixes were integrated to the studied systems. Developers agree that IC will cause confusion and insufficient information in the logs, which may increase the difficulties of error diagnostics.

Pattern 2: Inconsistent error-diagnostic information (IE).
Description. We find that sometimes duplicate logging statements for documenting exceptions may contain inconsistent error-diagnostic information (e.g., one logging statement records the stack trace and the other does not), even though the surrounding code is similar.
Example. As shown in Table II, the two classes `Create`**`PortForwarding`**`RuleCmd` and `Create`**`Firewall`**`RuleCmd` in CloudStack have similar functionalities. The two logging statements have the same static text message and are in methods with identical names (i.e., *create()*, not shown due to space restriction). The *create()* method in `Create`**`PortForwarding`**`RuleCmd` is about creating rules for port forwarding and the method in `Create`**`Firewall`**`RuleCmd` is about creating rules for firewalls. These two methods have very similar code structure and business logic. However, the two logging statements record different information: One records the stack trace information and the other one only records the exception message (i.e., *ex.getMessage()*). Since the two logging statements have similar context, the error-diagnostic information recorded by the logs may need to be consistent for the ease of debugging. We reported this example, which is now fixed to have consistent error-diagnostic information.

Code smell instances. As shown in Table III, we find 20 instances of IE, and four of them are considered problematic based on our understanding. From the remaining instances of IE, we find three justifiable cases that may not require fixes.
Justifiable case IE.1: Duplicate logging statements record general and specific exceptions. For 11/20 instances of IE, we find that the duplicate logging statements are in the *catch* blocks of different types of exception. In particular, one duplicate logging statement is in the *catch* block of a generic exception (i.e., the `Exception` class in Java) and the other one is in the *catch* block of a more specific exception (e.g., application-specific exceptions such as `CloudRuntimeException`). In all the 11 cases, we find that one log would record the stack trace for `Exception`, and the duplicate log would only record the type of the occurred exception (e.g., by calling *e.getMessage()*) for a more specific exception. The rationale may be that generic exceptions, once occurred, are often not expected by developers [48], so it is important that developers log more error-diagnostic information.
Justifiable case IE.2: Duplicate logging statements are in the same catch block for debugging purposes. For 3/20 instances of IE, we find that the duplicate logging statements are *in the same catch* block and developers' intention is to use a duplicate logging statement at debug level to record rich error-diagnostic information such as stack trace (and the log level of the other logging statement could be error). The extra logging statements at debug level help developers debug the occurred exception and reduces logging overhead in production [29] (i.e., logging statements at debug level are turned off).
Justifiable case IE.3: Having separate error-handling classes. For 2/20 instances, we find that the error-diagnostic information is handled by creating an object of an error-handling class. As an example from CloudStack:

```
public final class LibvirtCreateCommandWrapper {
   ...
      } catch (final CloudRuntimeException e) {
         s_logger.debug("Failed to create volume: " +
              e.toString());
         return new CreateAnswerErrorHandler(command, e);
      }
   ...
}

public class KVMStorageProcessor {
   ...
      } catch (final CloudRuntimeException e) {
         s_logger.debug("Failed to create volume: ", e);
         return new CopyCmdAnswerErrorHandler(e.toString());
      }
   ...

}
```

In this example, extra logging is added by using error-handling classes (i.e., `CreateAnswerErrorHandler` and `CopyCmdAnswerErrorHandler`) to complement the logging statements. As a consequence, the *actual* logged information is consistent in these two methods: One method records *e.toString()* in the logging statement and records the exception variable *e* through an error-handling class; the other method records *e* in the logging statement and records *e.toString()* through an error-handling class.

Developers' feedback. We reported all the instances of IE (four in total) that we consider problematic to developers as pull requests, all of which are accepted by developers. Moreover, we ask developers whether our conjecture was correct for each of the justifiable cases of IE. We received positive feedback that confirms our manual analysis on the justifiable cases.

Pattern 3: Log message mismatch (LM).

Description. We find that sometimes after developers copy and paste a piece of code to another method or class, they may forget to change the log message, thus resulting in duplicate logging statements that record inaccurate system behaviors.

Example. As an example, in Table II, the method *doScaleDown()* is a code clone of *doScaleUp()* with very similar code structure and minor syntactical differences. However, developers forgot to change the log message in *doScaleDown()*, after the code was copied from *doScaleUp()* (i.e., both log messages contain *scaling up*). Such instances of LM cause confusion when developers analyze the logs.

Code smell instances. We find that there are 37 instances of LM that are caused by copying-and-pasting the logging statement to new locations without proper modifications. For all the 37 instances, the log message contains words of incorrect class or method name that may cause confusion when analyzing logs.

Developers' feedback. Developers agree that the log messages in LM should be changed in order to correctly record the execution behavior (i.e., update the copy-and-pasted log message to contain the correct class/method name). We reported all the 37 instances of LM that we found through pull requests, and all of the reported instances are now fixed.

Pattern 4: Inconsistent log level (IL).

Description. Log levels (e.g., *fatal*, *error*, *info*, *debug*, or *trace*) allow developers to specify the verbosity of the log message and to reduce logging overhead when needed (e.g., *debug* is usually disabled in production) [29]. A prior study [50] shows that log level is frequently modified by developers in order to find the most adequate level. We find that there are duplicate logging statements that, even though the log messages are exactly the same, the log levels are different.

Example. In the IL example shown in Table II, the two methods, which are from the same class `CompactionManager`, have very similar functionality (i.e., both try to perform cleanup after compaction), but we find that the log levels are different in these two methods.

Code smell instances. We find three justifiable cases in IL that may be developers' intended behavior. We do not find problematic instances of IL after communicating with developers – Developers think the problematic instances identified by our manual analysis may not be problems.

Justifiable case IL.1: Duplicate logging statements are in the catch blocks of different types of exception. Similar to what we observed in IE, we find that for 8/76 instances, the log level for a more generic exception is usually more severe (e.g., *error* level for the generic Java `Exception` and *info* level for an application-specific exception). Generic exceptions may be more unexpected to developers [48], so developers may use a log level of higher verbosity (e.g., *error* level) to record exception messages.

Justifiable case IL.2: Duplicate logging statements are in different branches of the same method. There are 35/76 instances belong to this case. Below is an example from ElasticSearch, where a set of duplicate logging statements may occur in the same method but in different branches.

```
if (lifecycle.stoppedOrClosed()) {
    logger.trace("failed to send ping transport message",
        e);
} else {
    logger.warn("failed to send ping transport message",
        e);
}
```

In this case, developers already know the desired log level and intend to use different log levels due to the difference in execution (i.e., in the if-else block).

Justifiable case IL.3: Duplicate logging statements are followed by error-handling code. There are 18/76 instances that are observed to have such characteristics: In a set of duplicate logging statements, some statements have log levels of higher verbosity, and others have log levels of lower verbosity. However, the duplicate logging statement with lower verbosity log level is followed by additional error handling code (e.g., *throw a new Exception(e);*). Therefore, the error is handled elsewhere (i.e., the exception is re-thrown), and may be recorded at a higher-verbosity log level.

Developers' feedback. In all the instances of IL that we found, developers think that IL may not be a problem. In particular, developers agreed with our analysis on the justifiable cases. However, developers think the problematic instances of IL from our manual analysis may also not be problems. We concluded the following two types of feedback from developers on the "suspect" instances of IL (i.e., 15 problematic ones from our manual analysis out of the 76 instances of IL). The first type of developers' feedback argues the importance of semantics and usage scenario of logging in deciding the log level. A prior study [50] suggests that logging statements that appear in syntactically similar code, but with inconsistent log levels, are likely problematic. However, based on developers' feedback that we received, IL still may not be a concern, even if the duplicate logging statements reside in very similar code. A developer indicated that "conditions and messages are important but the *context* is even more important". As an example, both of the two methods may display messages to users. One method may be displaying the message to *local* users with a *debug* logging statement to record failure messages. The other method may be displaying the message to *remote* users with an *error* logging statement to record failure messages (problems related to remote procedure calls may be *more severe* in distributed systems). Hence, even if the code is syntactically similar, the log level has a reason to be different due to the different semantics and purposes of the code (i.e., referred to as different *contexts* in developers' responses). Future studies should consider both the syntactic structure and semantics of the code when suggesting log levels.

The second type of developers' feedback acknowledges the inconsistency. However, developers are reluctant to fix such inconsistencies since developers do not view them as concerns. For example, we reported the instance of IL that we discussed in Table II to the developer. The developer replied: "I think it should probably be an *ERROR* level, and I missed it in the review (could make an argument either way, I do not feel strongly that it should be *ERROR* level vs *INFO* level." Our opinions (i.e., from us and prior studies [29], [50]) differ from that of developers' regarding whether such inconsistencies are problematic. On one hand, whether an instance of IL is problematic or not can be subjective. This shows the importance of including perspectives from multiple parties (e.g., user studies, discussions with developers) in future studies of software logging practice. On the other hand, the discrepancy also indicates the need of establishing a guidance for logging practice and further even enforcing such standard.

Pattern 5: Duplicate logging statements in polymorphism (DP).

Description. Classes in object-oriented languages are expected to share similar functionality if they inherit the same parent class or if they implement the same interface (i.e., polymorphism). Since log messages record a higher level abstraction of the program [41], we find that even though there are no clones among a parent method and its overridden methods, such methods may still contain duplicate logging statements. Such duplicate logging statements may cause maintenance overhead. For example, when developers update one log message, he/she may forget to update the log message in all the other sibling classes. Inconsistent log messages may cause problems during log analysis [1], [20].

Example. As shown in Table II, the two classes (`PowerShellFencer` and `ShellCommandFencer`) in Hadoop both extend the same parent class and implement the same interface. These two classes share similar behaviors. The inherited methods in the two classes have the identical log message. However, as the system evolves, developers may not always remember to keep the log messages consistent in the two inherited methods, which may cause problems during system debugging, understanding, and analysis.

Code smell instances. We find that all the 139 instances of DP are potentially problematic that may be fixed by refactoring. In most of the instances, the parent class is an abstract class, and the duplicate logs exist in the overridden methods of the subclasses. We also find that in most cases, the overridden methods in the subclasses are very similar with minor differences (e.g., to provide some specialized functionality), which may be the reason that developers use duplicate logging statements.

Developers' feedback. Developers generally agree that DP is associated with logging code smells and specific refactoring techniques are needed. One developer comments that:

"You want to care about the logging part of your code base in the same way as you do for business-logic code (one can argue it is part of it), so salute DRY (do-not-repeat-yourself)."

Resolving DP often requires systematic refactoring. However, to the best of our knowledge, current Java logging frameworks, such as SLF4J and Log4j 2, do not support refactoring logging statements. The way to resolve DP is to ensure that the log message of the parent class can be reused by the subclasses, e.g., storing the log message in a static constant variable. We received similar suggestions from developers on how to refactor DP, such as "adding a method in the parent class that generates the error text for that case: *logger.error(notAccessible(field.getName()));*", or "creat[ing] your own Exception classes and put message details in them". However, we find that without supports from logging frameworks, even though developers acknowledged the issue of DP, they do not want to *manually* fix the code smells. Similar to some code smells studied in prior research [23], [42], developers may be reluctant to fix DP due to additional maintenance overheads but limited supports (i.e., need to manually fix hundreds of DP instances). *In short, logging frameworks should provide better support to developers in creating log "templates" that can be reused in different places in the code.*

We manually uncovered five patterns of duplicate logging code smells and six justifiable cases (for the IE and IL pattern) where the code smell instances may not need fixes. In total, our study helped developers fix 56 problematic duplicate logging code smells in the studied systems.

IV. DLFinder: Automatically Detecting Problematic Duplicate Logging Code Smells

The manual study uncovers five patterns of duplicate logging code smells and also provides guidance in identifying *problematic* logging code smells that require fixes. To help developers detect such problematic code smells and improve logging practices, we propose an automated approach, specifically a static analysis tool, called DLFinder. DLFinder uses abstract syntax tree (AST) analysis, data flow analysis, and text analysis. Below, we discuss how DLFinder detects each pattern of duplicate logging code smell.

Detecting inadequate information in catch blocks (IC). DLFinder first locates the *try-catch* blocks that contain duplicate logging statements. Specifically, DLFinder finds the *catch* blocks of the same try block that catch different types of exceptions, and these catch blocks contain the same duplicate logging statement. Then, DLFinder uses data flow analysis to analyze whether the handled exceptions in the *catch* blocks are logged (e.g., record the exception message). DLFinder detects an instance of IC if none of the logs in the *catch* blocks record either the stack trace or the exception message.

Detecting inconsistent error-diagnostic information (IE). DLFinder first identifies all the *catch* blocks that contain duplicate logging statements. Then, for each *catch* block, DLFinder uses data flow analysis to determine how the exception is logged by analyzing the usage of the exception variable in the logging statement. The logging statement records 1) the entire stack trace, 2) only the exception message, or 3) nothing at all. Then, DLFinder compares how the exception variable is used/recorded in each of the duplicate logging statements. DLFinder detects an instance of IE if a set of duplicate logging statements that appear in *catch* blocks has an inconsistent way of recording the exception variables (e.g., the log in one *catch* block records the entire stack trace, and the log in another *catch* block records only the exception message, while the two catch blocks handle the same type of exception). Note that for each instance of IE, the multiple *catch* blocks with duplicate logging statements in the same set may belong to different *try* blocks. In addition, DLFinder decides if an instance of IE belongs to one of the three justifiable cases (IE.1–IE.3). If so, the instance is marked as potentially justifiable and thus excluded by DLFinder.

Detecting log message mismatch (LM). LM is about having an incorrect method or class name in the log message (e.g., due to copy-and-paste errors). Hence, DLFinder analyzes the text in both the log message and the class-method name (i.e., concatenation of class name and method name) to detect LM by applying commonly used text analysis approaches [16]. DLFinder detects instances of LM using four steps: 1) For each logging statement, DLFinder splits class-method name into a set of words (i.e., *name set*) and splits log message into a set of words (i.e., *log set*) by leveraging naming conventions (e.g., camel cases) and converting the words to lower cases. 2) DLFinder applies stemming on all the words using Porter Stemmer [37]. 3) DLFinder removes stop words in the log message for each system. We find that there is a significant number of words that are generic across the log messages in a system (e.g., on, with, and process). Hence, we obtain the stop words by finding the top 50 most frequent words (each of our four studied systems has an average of 3,178 unique words in the static text messages) across all log messages in a system [46]. 4) For every logging statement, between the name set (i.e., from the class-method name) and its associated log set, DLFinder counts the number of common words shared by both sets. Afterward, DLFinder detects an instance of LM if the number of common words is inconsistent among the duplicate logging statements in one set.

For the LM example shown in Table II, the common words shared by the first pair (i.e., method *doScaleUp()* and its log) are "scale, up", while the common word shared by the second pair is "scale". Hence DLFinder detects an LM instance due to this inconsistency. The rationale is that the number of common words between the class-method name and the associated logging statement is subject to change if developers make copy-and-paste errors on logging statements (e.g., copy the logging statement in *doScaleUp()* to method *doScaleDown()*), but forget to update the log message to match with the new method name "doScaleDown". However, the number of common words will remain unchanged (i.e., no inconsistency) if the logging statement (after being pasted at a new location) is updated respectively.

Detecting inconsistent log level (IL). DLFinder detects an instance of IL if duplicate logging statements in one set (i.e., have the same static text message) have inconsistent log level. Furthermore, DLFinder checks whether an instance of IL belongs to one of the three justifiable cases (IL.1–IL.3). If so, the instance is marked as justifiable and DLFinder excludes this instance in the detection result.

Detecting duplicate logs in polymorphism (DP). DLFinder generates an object inheritance graph when statically analyzing the Java code. For each overridden method, DLFinder checks if there exist any duplicate logging statements in the corresponding method of the sibling and the parent class. If there exist such duplicate logging statements, DLFinder detects an instance of DP. Note that, based on the feedback that we received from developers (Section III), we do not expect developers to fix instances of DP. DP instances can be viewed more as technical debts [27] and our goal is to propose an approach to detect DP instances to raise developers' awareness regarding this issue.

V. An Evaluation of DLFinder

We evaluate our tool by answering three research questions.

RQ1: How well can DLFinder detect duplicate logging code smells in the four manually studied systems?

We applied DLFinder on the same versions of the systems that we used in our manual study (Section III). Since we obtain the ground truth (i.e., problematic code smells) in these four systems from our manual study, the goal of this RQ is to evaluate the detection accuracy of DLFinder. We calculated the precision and recall of DLFinder in detecting problematic duplicate logging code smells. Precision is the percentage of problematic code smell instances among all the detected instances, and recall is the percentage of problematic code smell instances that DLFinder is able to detect.

The first five rows of Table IV show the results of RQ1. Note that all the numbers in Table IV do not contain instances of justifiable cases since DLFinder focuses on detecting problematic ones and excludes the justifiable cases. For the patterns of IC, IE, and DP, DLFinder detects all the problematic instances of duplicate logging code smells (100% in recall) with a precision of 100%. For the IL pattern, since we do not find any problematic instances (as discussed in Section III), both of the columns of problematic instances in ground truth (*Pro.*) and detected (*T.Det.*) in Table IV are 0.

For the LM pattern, DLFinder achieves a recall of 83.8% (i.e., DLFinder detects 31/37 problematic LM instances). We manually investigate the six instances of LM that DLFinder cannot detect. We find that the problem is related to typos in the log message. For example, developers may write "mlockall" instead of "mLockAll". Hence, the text in the log message cannot be matched with the method name when we split the word using camel cases. The precision of detecting problematic LM instances is modest because, in many false positive cases, the log messages and class-method names are at different levels of abstraction: The log message describes a local code block while the class-method name describes the functionality of the entire method. For example, *encodePublicKey()* and *encodePrivateKey()* both contain the duplicate

TABLE IV
THE RESULTS OF DLFINDER IN RQ1 AND RQ2. IN EACH PATTERN, Pro. IS THE NUMBER OF PROBLEMATIC INSTANCES AS THE GROUND-TRUTH, Det. IS THE NUMBER OF INSTANCES DETECTED BY DLFINDER, AND T.Det. IS THE NUMBER OF true PROBLEMATIC INSTANCES DETECTED BY DLFINDER.

Research questions		IC			IE			LM			IL			DP		
		Pro.	T.Det.	Det.	Pro.	T.Det.	Det.	Pro.	T.Det.	Det.	Pro.	T.Det.	Det.	Pro.	T.Det.	Det.
RQ1: applying DLFinder on the same software versions as the manual study	**Hadoop**	5	5	5	0	0	0	9	7	44	0	0	1	27	27	27
	CloudStack	8	8	8	4	4	4	27	24	186	0	0	12	107	107	107
	ElasticSearch	1	1	1	0	0	0	1	0	15	0	0	0	3	3	3
	Cassandra	1	1	1	0	0	0	0	0	4	0	0	2	2	2	2
	Precision / Recall	100% / 100%			100% / 100%			12.4% / 83.8%			N/A			100% / 100%		
RQ2: applying DLFinder on new systems	**Camel**	1	1	1	0	0	0	14	10	95	0	0	3	29	29	29
	Wicket	1	1	1	0	0	0	1	1	4	0	0	0	1	1	1
	Precision / Recall	100% / 100%			- / -			11.1% / 73.3%			N/A			100% / 100%		
	Total	17	17	17	4	4	4	52	42	348	0	0	18	169	169	169

logging statement *"Unable to create KeyFactory"*. The duplicate logging statement describes a local code block that relates to usage of the *KeyFactory* class, which is different from the major functionalities of the two methods (i.e., as expressed by their class-method names). Nevertheless, DLFinder detects the LM instances with a high recall, and developers may quickly go through the results to identify the true positives (it took the first two authors less than 10 minutes on average to go through the LM result of each system to identify true positives). In the future, we plan to improve the precision of DLFinder by adopting a ranking mechanism based on estimating whether the log message and class-method name are at a similar level of abstraction (e.g., both describe the major functionality of a method). Moreover, we plan to find a better summarization of the surrounding code of the logging statements by utilizing the state-of-the-art research on code summarization [39].

RQ2: How well can DLFinder detect duplicate logging code smells in the additional systems?

The goal of this RQ is to study whether the uncovered patterns of duplicate logging code smells are generalizable to other systems. We applied DLFinder on two additional systems that are not included in the manual study in Section III: Camel 2.21.1 (released on Apr. 28, 2018) and Wicket 8.0.0 (released on May 16, 2018), which are both large-scale open source Java systems (1.7M and 380K LOC, respectively) for message routing and web application development. Similar to our manual study, the first two authors of this paper manually collect the problematic duplicate logging code smells in the additional systems, i.e., the ground-truth used for calculating the precision and recall of DLFinder. Note that the collected ground-truth of the two additional systems is only used in this evaluation, but not in designing the patterns in DLFinder. (There are also no new patterns found in this process.)

The sixth to eighth rows in Table IV show the results of the two additional systems. In total, we found 17 problematic code smell instances (DLFinder detected 13) in these two systems and all of them are fixed. Compared to the four systems in RQ1, DLFinder has similar precision and recall values in the two additional systems. Similar to what we found in RQ1, DLFinder cannot detect some LM instances due to typos in log message. DLFinder detected three instances of IL in Camel, however, with the manual investigation and getting developers' feedback, these IL instances are not problematic. Similar to what we discussed in Section III, the differences in the log level are related to having different semantics in the code.

TABLE V
THE RESULT OF RQ3: APPLYING DLFINDER TO THE NEWER VERSIONS OF THE STUDIED SYSTEMS. GAP. SHOWS THE DURATION OF TIME IN DAYS BETWEEN THE ORIGINAL (ORG.) AND THE NEWER RELEASE (NEW.)

	Releases Org., New.	Gap.	**IC**	**IE**	**LM**	**IL**	**DP**
Hadoop	3.0.0, 3.0.3	208	0	0	2	0	21
CloudStack	4.9.3, 4.11.1	297	5	0	2	0	0
ElasticSearch	6.0.0, 6.1.3	77	0	0	0	0	0
Cassandra	3.11.1, 3.11.3	294	0	0	0	0	1
Total	-	-	5	0	4	0	22

Different from a prior study [50], we found that all IL instances are not problematic in the six evaluated systems. Future studies are needed to investigate the effect of IL. DLFinder detects DP instances with 100% in recall and precision, however, we do not report them since developers are reluctant to fix them due to limited support from logging framework (as discussed in Section III). Nevertheless, the patterns of duplicate logging code smells that we uncovered can still be found in other systems.

RQ3: Are new code smell instances introduced over time?

We applied DLFinder on the latest versions of the four studied systems, i.e., Hadoop, CloudStack, ElasticSearch, and Cassandra. We then compared the results with the ones on previous versions. The gaps of days between the manually studied versions and the new versions vary from 77 days to 297 days. Table V shows that new instances of code smells are introduced during software evolution. These detected code smell instances are all problematic and are all reported and fixed except for DP. As mentioned in Section III and IV, our goal of detecting DP is to show developers the logging technical debt in their systems. In short, we found that duplicate logging code smells are introduced over time, and an automated approach such as DLFinder can help developers avoid duplicate logging code smells as the system evolves.

DLFinder is able to detect 72 problematic code smell instances in the four manually studied systems and two additional systems. The code smell patterns of DLFinder (i.e., uncovered from our manual study) also exist in new systems and DLFinder can detect new code smell instances that are introduced as systems evolve.

VI. THREATS TO VALIDITY

Internal validity. We define duplicate logging statements as two or more logging statements that have the same static text message. We were able to uncover five patterns of duplicate logging code smells and detect many code smell instances. However, logging statements with non-identical but similar static texts may also cause problems to developers. Future studies should consider different types of duplicate logging statements (e.g., logs with similar text messages). We conducted manual studies to uncover the patterns of code smells and study their potential impact. To avoid biases, two of the authors examine the data independently. For most of the cases the two authors reach an agreement. Any disagreement is discussed until a consensus is reached. In order to reduce the subjective bias from the authors, we have contacted the developers to confirm the uncovered patterns and their impact.

External validity. We conducted our study on four large-scale open source systems in different domains. We found that our uncovered patterns and the corresponding problematic and justifiable cases are common among the studied systems. However, our finding may not be generalizable to other systems. Hence, we studied whether the uncovered patterns exist in two other systems. We found that the patterns of code smells also exist in these two systems and we did not find any new code smell patterns in our manual verification. Our studied systems are all implemented in Java, so the results may not be generalizable to systems in other programming languages. Future studies should validate the generalizability of our findings in systems in other programming languages.

VII. Related Work

Empirical studies on logging practices. There are several studies on characterizing the logging practices in software systems [11], [18], [50]. Yuan *et al.* [50] conducted a quantitative characteristics study on log messages for large-scale open source C/C++ systems. Chen *et al.* [11] replicated the study by Yuan *et al.* [50] on Java open-source projects. Both of their studies found that log message is crucial for system understanding and maintenance. Fu *et al.* [18] studied where developers in Microsoft add logging statements in the code and summarized several typical logging strategies. They found that developers often add logs to check the returned value of a method. Different from prior studies, in this paper, we focus on manually understanding duplicate logging code smells. We also discuss potential approaches to detect and fix these code smells based on different contexts (i.e., surrounding code).

Improving logging practices. Zhao *et al.* [53] proposed a tool that determines how to optimally place logging statements given a performance overhead threshold. Zhu *et al.* [54] provided a tool for suggesting log placement using machine learning techniques. Yuan *et al.* [51] proposed an approach that can automatically insert additional variables into logging statements to enhance the error diagnostic information. Chen *et al.* [12] concluded five categories of logging anti-patterns from code changes, and implemented a tool to detect the anti-patterns. Hassani *et al.* [20] identified seven root-causes of the log-related issues from log-related bug reports. Compared to prior studies, we study logging code smells that may be caused by duplicate logs, with a goal to help developers improve logging code. The logging problems that we uncovered in this study are not discovered by prior work. We conducted an extensive manual study through obtaining a deep understanding on not only the logging statements but also the surrounding code, whereas prior studies usually only look at the problems that are related to the logging statement itself.

Code smells. Code smells can be indications of bad design and implementation choices, which may affect software systems' maintainability [5], [30], [43], [44], understandability [4], [10], and performance [45]. To mitigate the impact of code smells, studies have been proposed to detect code smells [21], [32]–[34], [40]. Duplicate code (or code clones) is a kind of code smells which may be caused by developers copying and pasting a piece of code from one place to another [38], [52]. Such code clones may indicate quality problems. There are many studies that focus on studying and detecting code clones [25], [26]. In this paper, we study duplicate logging code smells, which are not studied in prior duplicate code studies. In fact, in our manual study, we found that many duplicate logging statements may not be related to code clones. Future studies may further investigate the relationship between duplicate code and duplicate logging statements.

VIII. Conclusion

Duplicate logging statements may affect developers' understanding of the system execution. In this paper, we study over 3K duplicate logging code statements in four large-scale open source systems (Hadoop, CloudStack, ElasticSearch, and Cassandra). We uncover five patterns of duplicate logging code smells. Further, we assess the impact of each code smell and find not all are problematic and need fixes. In particular, we find six justifiable cases where the uncovered patterns of duplicate logging code smells may not be problematic. We received confirmation from developers on both the problematic and justifiable cases. Combining our manual analysis and developers' feedback, we developed a static analysis tool, DLFinder, which automatically detects problematic duplicate logging code smells. We applied DLFinder on the four manually studied systems and two additional systems. In total, we reported 82 problematic code smell instances in the six studied systems to developers and all of them are fixed. DLFinder successfully detects 72 out of the 82 instances. Our study highlights the importance of the context of the logging code, i.e., the nature of logging code is highly associated with both the structure and the functionality of the surrounding code. Future studies should consider the code context when providing guidance to logging practices. In addition, more advanced logging libraries are needed to help developers improve logging practice and to avoid logging code smells.

References

[1] "Changes to JobHistory makes it backward incompatible," https://issues.apache.org/jira/browse/HADOOP-4190, last checked April 4th 2018.

[2] "Log4j," http://logging.apache.org/log4j/2.x/.

[3] "Simple logging facade for Java (SLF4J)," http://www.slf4j.org, last checked Feb. 2018.

[4] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, Sept 2011, pp. 125–134.

[5] I. Ahmed, C. Brindescu, U. A. Mannan, C. Jensen, and A. Sarma, "An empirical examination of the relationship between code smells and merge conflicts," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17, 2017, pp. 58–67.

[6] H. Barringer, A. Groce, K. Havelund, and M. H. Smith, "Formal analysis of log files," *JACIC*, vol. 7, no. 11, pp. 365–390, 2010.

[7] S. Boslaugh and P. Watters, *Statistics in a Nutshell: A Desktop Quick Reference*, ser. In a Nutshell (O'Reilly). O'Reilly Media, 2008.

[8] D. Budgen, *Software Design*. Addison-Wesley, 2003.

[9] N. Busany and S. Maoz, "Behavioral log analysis with statistical guarantees," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 877–887.

[10] C. Chapman, P. Wang, and K. T. Stolee, "Exploring regular expression comprehension," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 405–416.

[11] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, Feb 2017.

[12] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 71–81.

[13] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 305–316.

[14] T.-H. Chen, W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, 2016, pp. 666–677.

[15] T.-H. Chen, M. D. Syer, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '17, 2017, pp. 243–252.

[16] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1843–1919, 2016.

[17] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. Addison-Wesley object technology series, 1999.

[18] 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 *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE-SEIP '14, 2014, pp. 24–33.

[19] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz, "An Industrial Case Study of Customizing Operational Profiles Using Log Compression," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 713–723.

[20] M. Hassani, W. Shang, E. Shihab, and N. Tsantalis, "Studying and detecting log-related issues," *Empirical Software Engineering*, 2018.

[21] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting and refactoring code smells in spreadsheet formulas," *Empirical Software Engineering*, vol. 20, no. 2, pp. 549–575, 2015.

[22] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of 24th International Conference on Software Maintenance (ICSM)*, 2008, pp. 307–316.

[23] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 672–681.

[24] 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*, ser. MSR '16, 2016, pp. 154–164.

[25] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[26] C. Kapser and M. W. Godfrey, "Cloning considered harmful," *Reverse Engineering, Working Conference on*, vol. 0, pp. 19–28, 2006.

[27] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, 2012.

[28] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, Jan 2018.

[29] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, Aug 2017.

[30] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, 2016, pp. 225–234.

[31] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia Medica*, vol. 22, no. 3, pp. 276–282, 2012.

[32] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Detection of embedded code smells in dynamic web applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012, pp. 282–285.

[33] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 268–278.

[34] C. Parnin, C. Görg, and O. Nnadi, "A catalogue of lightweight visualizations to support code smell inspection," in *Proceedings of the 4th ACM Symposium on Software Visualization*, ser. SoftVis '08, 2008, pp. 77–86.

[35] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *Proceedings of th 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 169–178.

[36] H. Pinjia, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd IEEE international conference on Automated software engineering*, 2018, pp. 1–11.

[37] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[38] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 72–81.

[39] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct 2017, pp. 112–122.

[40] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10, 2010, pp. 8:1–8:10.

[41] 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*, ser. ICSME '14, 2014, pp. 21–30.

[42] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '16, 2016, pp. 858–870.

[43] D. I. K. Sjberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyb, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, Aug 2013.

[44] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 403–414.

[45] X. Xiao, S. Han, C. Zhang, and D. Zhang, "Uncovering javascript performance code smells relevant to type mutations," in *Programming Languages and Systems*, X. Feng and S. Park, Eds., 2015, pp. 335–355.

[46] J. Yang and L. Tan, "SWordNet: Inferring semantically related words from software context," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1856–1886, 2014.

[47] K. Yao, G. B. d. Pdua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting logging locations for web-based systems performance monitoring," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, 2018, pp. 21–30.

[48] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, 2014, pp. 249–265.

[49] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 143–154.

[50] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *ICSE 2012: Proceedings of the 2012 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012, pp. 102–112.

[51] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *ASPLOS '11: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. Newport Beach, California, USA: ACM, 2011, pp. 3–14.

[52] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance*, vol. 23, no. 3, pp. 179–202, 2011.

[53] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, pp. 565–581.

[54] 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*, ser. ICSE '15, 2015, pp. 415–425.