Mining Monitoring Concerns Implementation in Java-Based Software Systems

Grigoreta Sofia Cojocar
Babeş-Bolyai University
Cluj-Napoca, Romania
grigo@cs.ubbcluj.ro

Adriana-Mihaela Guran
Babeş-Bolyai University
Cluj-Napoca, Romania
adriana@cs.ubbcluj.ro

ABSTRACT

In this paper we describe a new approach for automatic identification of monitoring concerns implementation in Java-based software systems. We also present the results obtained by using our approach on 21 Java-based systems, ranging from small to very large systems.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software evolution*;

KEYWORDS

monitoring concerns implementation, automatic identification

ACM Reference Format:
Grigoreta Sofia Cojocar and Adriana-Mihaela Guran. 2018. Mining Monitoring Concerns Implementation in Java-Based Software Systems. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering (NL4SE '18), November 4, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3283812.3283821

1 INTRODUCTION

For more than a decade researchers have tried to propose techniques for automatic crosscutting concerns identification, with the goal of refactoring. However, the results obtained are more like hints on where to start looking for them, and, in the end, the user still has to manually select the most appropriate results. One of the causes for these results is that the proposed approaches are more like "one-size-fits-all" solutions that try to identify all the crosscutting concerns that exist in a software system, without taking into consideration the particulars of each crosscutting concern. There are many different kinds of crosscutting concerns, like monitoring, security, transaction management, and each kind of concern has its particulars, and, maybe, it should be mined differently than other concerns. Still, the question *"Can crosscutting concerns be automatically identified from a system's code?"* remains. In order to answer this question, we focused on one kind of crosscutting concern, namely monitoring. Monitoring crosscutting concerns record the behaviour of a software system during development, testing and execution in its own environment. The monitoring concerns are usually classified in: logging, tracing and performance monitoring. *Logging* produces messages specific to the logic carried by a piece of code. *Tracing* produces messages for lower-level events such as: the entry or exit of a method, exception handling or object construction, and state modification. *Performance monitoring* measures the time taken by specific parts of the system to execute and/or the number of times a particular method is invoked.

We have started our research by analyzing how two monitoring concerns, namely *logging* and *tracing*, are implemented in order to gather information about the patterns used and their particulars. In this paper we present our original approach for automatic identification of *logging* and *tracing* monitoring concerns implementation in Java-based software system. This approach was developed based on the findings of our previous studies of how monitoring crosscutting concerns are implemented in object-oriented software systems.

The rest of the paper is structured as follows. Section 2 briefly presents the main results of our monitoring concerns studies. Our original approach is described in Section 3. Experimental results are given in Section 4. Similar approaches are described in Section 5, and further work is presented in Section 6.

2 MONITORING CONCERNS STUDY

In previous studies we have manually analyzed ten open-source software systems, developed in Java or C# in order to discover the pattern(s) used for monitoring concerns implementation [2, 3]. The most important results of the studies are:

(1) *Monitoring concerns are implemented using a logging toolkit.* The *logging* toolkit is responsible for recording the messages and for filtering only those messages specified in the configuration. The toolkit is either implemented in the analyzed software system or it is from a third-party.
(2) *Software systems usually use more than one logging toolkit for monitoring.* However some toolkits are used in just a couple of classes.
(3) *Different patterns are used for monitoring concerns implementation.* Even in the same system, monitoring concerns are implemented using different patterns. In our studies we have identified six different patterns, the three most used being:
   P1 - An object of the type used for recording messages from the logging toolkit, called *logger*, is declared as an attribute in the classes where monitoring must be performed. Then, each operation that needs to record monitoring messages calls the corresponding methods on this object.
   P2 - There is no declaration of a *logger* object as an attribute, but in each method where messages need to be recorded a local *logger* object is declared, and then used for recording.

NL4SE '18, November 4, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6055-5/18/11…$15.00
https://doi.org/10.1145/3283812.3283821

P3 - The *logger* attribute is inherited from a base class (there is no new declaration of a *logger* attribute), and this inherited attribute is used whenever needed.

From our studies we have determined that each analyzed system has a primary (or predominant) pattern used for monitoring concerns implementation and a few secondary ones. **P1** seems to be the primary pattern of choice for most systems, still there are exceptions [2]. For Java-based systems, pattern **P1** was used in more than 88% of the source files with monitoring for all analyzed systems. Also, in the case of **P1** pattern different styles were used for declaring the *logger* attribute, excluding the access modifier. Four styles were identified in the analyzed Java-based systems:

S1a - as a regular attribute (eg. `LoggerType loggerName`),
S1b - as a `static` attribute,
S1c - as a `final` attribute, and
S1d - as a `static` and `final` attribute.

S1d seems to be the preferred style for Java-based systems, but there are exceptions [2].

3 NEW APPROACH

Our new original approach is based on the results obtained in the previous studies. These results have shown that for Java-based systems the most used pattern for *logging* and *tracing* implementation is **P1** - the declaration of an attribute, usually as a `static` and/or `final` one, very often called *'log'* or *'logger'*, and then calling different methods on this attribute.

The approach tries to identify only *logging* and *tracing* monitoring concerns by analyzing the `static` and/or `final` attributes defined in a Java-based software system. It consists of automatically identifying the type(s) of the *logger* object, and then the classes in which this object is used. The challenging part of the approach is the automatic identification of the *logger* object' s type. Based on the results of our studies, in a software system there may be multiple types used for the *logger* object.

In order to automatically identify the type(s) of the *logger* object we perform two steps: *Computation* and *Filtering*.

Computation. For a Java-based software system S we compute the set $AttributesTypes(S) = \{T_1, T_2, ..., Tp\}$ corresponding to the types used for declaring the attributes in the system. Each element $T_i, i \in \{1, ..., p\}$ from the set contains the following data: $T_i = \langle TypeName_i, SFAttr_i, RegAttr_i, ClassesSFAttr_i, ClassesRegAttr_i, AttributesNames_i \rangle$ where:

- $TypeName_i$ is the fully qualified name of the type, eg. `int` or `java.lang.String`.
- $SFAttr_i$ is the number of times a `static` or `final` attribute of this type was declared in the system S.
- $RegAttr_i$ is the number of times a regular attribute of this type was declared in the system S. We consider an attribute as regular, if neither of the `static` or `final` modifiers were used for its declaration.
- $ClassesSFAttr_i$ is the set of classes in which at least one `static` or `final` attribute of this type was declared. $|ClassesSFAttr_i| \leq SFAttr_i$, where $|A|$ denotes the number of elements in the set A.
- $ClassesRegAttr_i$ is the set of classes in which at least one regular attribute of this type was declared. $|ClassesRegAttr_i| \leq RegAttr_i$.
- $AttributesNames_i$ is the set containing the `static` or `final` attributes' names and their frequency. $AttributesNames_i = \{(name_{i1}, freq_{i1}), ..., (name_{ik_i}, freq_{ik_i})\}$, where $1 \leq k_i \leq SFAttr_i$ and $\sum_{j=1}^{k_i} freq_{ij} = SFAttr_i$.

Filtering. After computing the set $AttributesTypes(S)$ we apply two filters in order to automatically determine the types used for *logging* or *tracing*. The first filter removes the following types: all Java primitive types (`byte`, `short`, `int`, `float`, `double`, `char`), any arrays of a primitive type (like `byte[]` or `int[][]`), all the types defined in `java.util` or `java.lang` packages (such as `java.lang.String`, `java.util.ArrayList`) but not the subpackages (types like `java.lang.reflect.Method` will not be removed), any arrays of a type defined in these two packages (eg. `java.lang.String[]`). It also removes the types that were used for declaring `static` or `final` attributes in less than 3 classes ($SFAttr < 3$). Monitoring concerns are crosscutting, meaning that they are implemented in many different classes. We consider that if a `static` or `final` attribute of the same type is defined in more than 3 classes than it can be considered as crosscutting.

Let $AttrTypes_{F_1}(S)$ be the set of types remaining after the first filter is applied. The second filter computes for each remaining type its probability of being a *logger* type, using Definition 3.1.

Definition 3.1. ***Probability of being a logger type.*** *Being given a type T_i used in the system S, $T_i \in AttributesTypes(S)$, we define the probability of being a logger type of T_i, $Plog(T_i)$, as*

$$Plog(T_i) = \frac{\sum_{j=1}^{k_i} freq_{ij} \cdot isLoggerName(name_{ij})}{SFAttr_i},$$

where isLoggerName is defined as:

$$isLoggerName(s) = \begin{cases} 1 & \text{if } 'log' \text{ is a substring of } s \\ 0 & \text{otherwise.} \end{cases}$$

The $isLoggerName(s)$ function performs a case insensitive comparison of *'log'* and s. The definition of $Plog$ is based on the observation from our studies that most of the names used for the *logger* attribute' name are 'log', 'logger', 'LOG', 'Logger' or the name contains the 'log' or the 'logger' substring.

The result of the second filter is the set $LogTypes(S)$ defined as follows:

$$LogTypes(S) = \{T_i \in AttrTypes_{F_1}(S) \mid Plog(T_i) >= 0.5\}.$$

In the results set, we select only the types whose probability of being a *logger* type is greater or equal to 0.5. If $|LogTypes(S)| = 0$ then we conclude that *tracing* or *logging* concerns are not implemented in the system S. If $|LogTypes(S)| > 0$, then all the types $T_i \in LogTypes(S)$ are used for *tracing* or *logging* monitoring concerns implementation, and the set $ClassSFAttr_i \cup ClassesRegAttr_i$ contains the classes in which they are implemented using pattern **P1**.

4 EXPERIMENTAL RESULTS

We have applied our new approach for automatic identification of *tracing* and *logging* implementation to 21 randomly chosen Java-based software systems, ranging from small systems with under 100 `.class` files analyzed to very large systems with thousands of

Table 1: Analyzed Software Systems

Software system	Version	Number of analyzed files
Seedstack	3.2.5	80
Openkoreantext	2.3.1	98
Dozermapper	6.4.0	269
Sonar-orchestrator	3.20.0.1708	347
Neuroph	2.94	382
JGap	3.6	447
Mars-sim	3.10	711
Spoon	7.0.0	1009
Atomix	3.0.0-rc8	1116
Atlas	5.1.6	1132
Ant	1.10.5	1141
Unboundid	4.0.7	1146
Maven-core	3.5.4	1167
JEdit	5.4	1355
Tomcat	9.0.10	1609
ArgoUML	0.34	2248
Hadoop	3.1.0	4442
Spring	5.0.8	4487
Hibernate	5.3.3.Final	4665
Drill-java-exec	1.14.0	5112
Hive-exec	3.1.0	21531

`.class` files analyzed, as presented in Table 1. The table also shows the version used for each analyzed system. The .jar files containing the analyzed .class files were almost all downloaded from Maven Repository and the files were analyzed using Soot [6].

The results obtained using our approach are presented in Table 2. For each analyzed system we give the number of results obtained, the value of *Plog* for each result, the type's name, the value of *SFAttr* and the *AttributesNames* for the corresponding type. If the *AttributesNames* contains '...' it means that the set contains other elements, but it was too large to be fully included in this table. As the results show our approach was able to automatically and correctly identify the types used for *logging* and *tracing* concerns (or their absence from the system) for 17 out of the 21 analyzed software systems. For 3 systems, namely Hadoop, Drill-java-exec and Hive-exec, the results were partially correct, and for 1 system, namely Mars-sim the results were incorrect. The approach was also able to automatically identify multiple *logger* types for 5 systems; for one of them, namely Hive-exec, identifying no less than 5 types used for *logging* or *tracing* implementation.

For each system whose results were only partially correct, the results set contains one or two types that are not used for monitoring implementation, but the *Plog* value was high enough to be included in the results set, due to the names used for declaring the attributes, i.e. the names contain the 'log' substring (e.g. 'networktopology', 'loggedInUgi', 'loginUser', and 'chronology'). For Mars-sim system, the results set contains just one type for the *logger* object, but the type was not correctly identified. The problem is, again, due to the *Plog* value, which is relatively high as some attributes' names contain the 'log' substring, i.e. 'val$dialog'.

False positives. The results obtained by the approach included a few false positives (types which were returned in the result sets, but are not used for monitoring concerns implementation), however their number is relatively low. From a total of 38 types returned by our approach for all the analyzed software systems, only 5 types were false positives (the ones emphasized in Table 2).

False negatives. None of the software systems used in our experiments have an apriori published list of types used for monitoring concerns implementation. We do not know if other types should have been included in the results set. An in-depth analysis of each system is required to determine if there are any false negatives.

5 RELATED WORK

In the beginning of 2000, many different techniques that tried to automatically identify all crosscutting concerns from a software system, called aspect mining techniques, were proposed. Monitoring concerns are crosscutting concerns, so all those techniques can be considered related to our work. However, those techniques do not automatically distinguish between different types of crosscutting concerns when they return the results, leaving this task to the user. Mens et al. provided an overview of the problems encountered by the proposed aspect mining techiques in [5]. Our approach focuses on only one kind of crosscutting concerns, and the user does not have to perform an additional selection after the results are returned.

In [1, 4] we proposed a similar approach for identifying the *logger* object type, but instead of computing a probability for each type and applying the second filter, we just sorted descending the possible types using the value of *SFAttr*. With that approach we were able to automatically identify one possible type for the *logger* object, as for most case studies it was the first ranked type. However that approach failed to automatically determine whether or not monitoring concerns were implemented in the system, and it also provided many false positives.

6 FURTHER WORK

Encouraged by the results obtained with this original approach, we will continue our work in the following directions:

- To use this approach on C#-based systems.
- To try to reduce the false positive results.
- To include in the classes resulting set, the ones where monitoring is implemented using patterns **P2** and **P3**.
- To study the implementation of other crosscutting concerns.

REFERENCES

[1] G.S. Cojocar and A.M. Guran. 2018. On Automatic Identification of Monitoring Concerns Implementation. In *Acta Electrotechnica et Informatica*, Vol. 3. to appear.
[2] G.S. Cojocar and A.M. Guran. 2018. A Study of Monitoring Crosscutting Concerns Implementation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18)*. ACM, NY, USA, 169–170.
[3] G. S. Cojocar. 2016. On Top-Down Aspect Mining for Monitoring Techniques Implementation. In *Proceedings of IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 249–254.
[4] G. S. Cojocar and A. M. Guran. 2017. On A Top Down Aspect Mining Approach for Monitoring Crosscutting Concerns Identification. In *Proceedings of IEEE 14th International Scientific Conference on Informatics (Informatics 2017)*. IEEE, 51–56.
[5] Kim Mens, Andy Kellens, and Jens Krinke. 2008. Pitfalls in Aspect Mining. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering (WCRE '08)*. IEEE Computer Society, Washington, DC, USA, 113–122.
[6] Soot [n. d.]. Soot: a Java Optimization Framework. ([n. d.]). http://www.sable.mcgill.ca/soot/.

Table 2: Software Systems' Results

System	Results	Plog	Type's Name	SFAttr	Attributes' names and frequency
Seedstack	1	1.0	org.slf4j.Logger	8	{ (LOGGER, 8) }
Openkoreantext	0	-	-	-	-
Dozermapper	1	1.00	org.slf4j.Logger	19	{(LOG, 7), (log, 12)}
Sonar-orchestrator	2	1.00 0.97	org.slf4j.Logger java.util.logging.Logger	12 39	{(LOG, 10), (LOGGER, 2)} {(logger, 22), (authLogger, 2), (stmtDT, 1), ...}
Neuroph	1	1.00	org.slf4j.Logger	9	{(LOG, 5), (LOGGER, 4)}
JGap	1	1.00	org.apache.log4j.Logger	10	{(LOGGER, 4), (log, 6)}
Mars-sim	1	0.60	*javax.swing.JDialog*	5	{*(val$dialog, 3), (val$frame, 1) ,...* }
Spoon	1	1.00	org.apache.log4j.Logger	3	{(LOGGER, 3)}
Atomix	1	0.98	org.slf4j.Logger	51	{(LOGGER, 33), (delegate, 1), (log, 17)}
Atlas	1	1.00	org.slf4j.Logger	138	{(logger, 138)}
Ant	0	-	-	-	-
Unboundid	1	1.00	java.util.logging.Handler	3	{(logHandler, 3)}
Maven-core	2	1.00 1.00	org.codehaus.plexus.logging.Logger org.eclipse.aether.spi.log.Logger	3 3	{(logger, 3)} {(LOGGER, 1), (logger, 2)}
JEdit	0	-	-	-	-
Tomcat	3	1.00 0.83 0.75	org.apache.juli.logging.Log org.apache.tomcat.util.log.UserDataHelper java.util.logging.Logger	164 6 4	{(log, 162), (logger, 2)} {(invalidCookieLog, 1), (userDataLog, 2), ...} {(logger, 1), (val$parent, 1), ...}
ArgoUML	1	1.00	org.apache.log4j.Logger	253	{(LOG, 253)}
Hadoop	4	1.00 1.00 1.00 1.00	org.slf4j.Logger org.apache.commons.logging.Log org.apache.hadoop.hdfs.server.namenode.FSEditLog *org.apache.hadoop.net.NetworkTopology*	321 100 3 3	{(LOG, 306), (LOGGER, 1), (blockLog, 2), ... } {(LOG, 92), (MetricsLog, 1), ...} {(editLog, 1), (log, 1), ...} {*(networktopology, 2), ...* }
Spring	1	1.00	org.apache.commons.logging.Log	233	{(logger, 229), (pageNotFoundLogger, 3), ... }
Hibernate	4	1.00 1.00 1.00 1.00	org.hibernate.internal.CoreMessageLogger org.jboss.logging.Logger org.hibernate.internal.EntityManagerMessageLogger org.hibernate.engine.jdbc.spi.SqlStatementLogger	245 171 10 8	{(LOG, 199), (log, 46)} {(LOG, 25), (LOGGER, 1), (log, 145)} {(log, 5), (MSG_LOGGER, 1), (LOG, 4)} {(sqlStatementLogger, 4), ...}
Drill-java-exec	2	0.98 0.66	org.slf4j.Logger *org.apache.hadoop.security.UserGroupInformation*	577 3	{(logger, 555), (tracer, 12), (log, 3), ... } {*(loggedInUgi, 1), (loginUser, 1)*}
Hive-exec	7	1.00 1.00 1.00 1.00 1.00 0.99 0.62	org.apache.commons.logging.Log java.util.logging.Logger org.apache.hadoop.hive.ql.log.PerfLogger *org.joda.time.chrono.BasicChronology* org.apache.hadoop.hive.ql.parse..repl.ReplLogger org.slf4j.Logger *org.joda.time.Chronology*	44 20 13 9 3 860 13	{(LOG, 44)} {(log, 4), (logger, 16)} {(perfLogger, 12), (PERF_LOGGER, 1)} {*(iChronology, 9)*} {(replLogger, 3)} {(LOG, 785), (LOGGER, 50), (logger, 4), ...} {*(chronology, 1), (iChronology, 7), ...*}