Poster: A Study Of Monitoring Crosscutting Concerns Implementation

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

The maintainability and understandability of a software system are affected by the way the system's concerns are implemented, especially if they are crosscutting concerns. In this paper we present a study of how monitoring crosscutting concerns are implemented in ten object-oriented software systems. The study's results are going to be used towards a new approach for automatic identification of monitoring concerns implementation.

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. Poster: A Study Of Monitoring Crosscutting Concerns Implementation. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195080

1 INTRODUCTION

For more than a decade researchers have proposed different techniques for (automatic) crosscutting concerns identification, but the obtained results are more like hints on where to start looking for them, and, in the end, the user still has to 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. But the question *"Can crosscutting concerns be automatically identified in the source code?"* still remains. In order to answer this question we should first analyze the existing software systems in order to gather information about how crosscutting concerns are implemented, about the patterns used, and their particulars. Based on these findings new approaches specific to each crosscutting concern could be developed, and better results could be obtained.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195080

In this paper we present the findings of our study of how monitoring crosscutting concerns are implemented in object-oriented software systems. Monitoring concerns record the behaviour of a software system during development, testing and execution in its own environment. We also discuss the challenges that need to be overcome in trying to develop an approach for their automatic identification in the source code.

2 STUDY

In order to determine how monitoring concerns are implemented, we have analyzed ten open-source software systems: five medium to large Java-based systems (ArgoUML, Jenkins, Tomcat v9, Spring Framework, and Hadoop) and five small to medium C#-based systems (Dache, EasyNetQ, Orleans, MassTransit, and Spring.NET).[1] For each system we wanted to discover the patterns used for monitoring implementation, and how much of the system is affected by these concerns. In Table 1 are shown the number of source files for each analyzed software system and the number of source files (and percentage) in which monitoring concerns were identified. During our analysis we have discovered the following:

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*, as shown in Table 1. Only one system, ArgoUML, used just one logging toolkit, and Hadoop, a very large system, used no less than 6 toolkits for monitoring, even though some of them were used just in a few source files.

3. *Different patterns are used for monitoring concerns implementation.* Even in the same system, monitoring concerns are implemented using different patterns. We have identified six different patterns in the analyzed software systems:

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.

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 to record monitoring messages.

[1]The accessing date and the download site of the analyzed source code for each software system are available at www.cs.ubbcluj.ro/~grigo/icse2018/systems.html.

Table 1: Software Systems' Analysis Results

Software system	Language	Source files	Files with monitoring	Logging toolkits	P1	P2	P3	P4	P5	P6	*Logger* declaration styles			
											S1a	*S1b*	*S1c*	*S1d*
Jenkins	Java	1296	209 (16.12%)	2	196	0	13	0	0	0	1	1	3	191
ArgoUML	Java	1922	324 (16.85%)	1	323	1	0	0	0	0	2	0	0	321
Tomcat v9	Java	2175	317 (14.57%)	2	288	2	26	0	1	0	3	20	27	238
Spring	Java	6243	431 (6.90%)	3	381	33	17	0	0	0	11	1	192	177
Hadoop	Java	7134	1547 (21.68%)	6	1526	9	11	0	1	0	9	105	16	1396
Dache	C#	66	7 (10.60%)	2	3	1	0	3	0	0	0	0	3	0
EasyNetQ	C#	381	52 (13.64%)	2	18	34	0	0	0	0	4	0	14	0
Orleans	C#	1062	272 (25.61%)	2	165	40	24	13	27	3	69	2	67	27
MassTransit	C#	1837	99 (5.38%)	5	85	14	0	0	0	0	1	0	22	62
Spring.NET	C#	2679	261(9.74%)	3	235	16	7	0	2	1	18	4	65	148

P4 - Only in C#-based systems, the *logger* is declared as a property of the class, using the specific syntax from C#, and then this property is used whenever monitoring messages need to be recorded. The property is used either only in the declaring class, or also in subclasses.

P5 - The *logger* object is sent as a parameter to the methods that need it. The callers of the methods are responsible for obtaining a reference to the *logger* object.

P6 - The class that needs to record monitoring messages defines a method for recording them, and then this method is called from all the other methods from the same class whenever necessary.

Table 1 shows the frequency of occurrence of these patterns in each analyzed software system. From these results we can conclude that each 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. For EasyNetQ the primary pattern used for monitoring implementation is **P2**. In Java-based systems, pattern **P1** is used in more than 88% of the source files with monitoring for all analyzed systems. In the C#-based systems the ratio is much lower for all analyzed systems. Only one system, Spring.NET uses this pattern in more than 90% of the source files with monitoring.

In the case of **P1** pattern different styles are used for declaring the *logger* attribute, excluding the access modifier. In the analyzed systems we have identified 4 possible styles: *S1a* - as a regular attribute (eg. `LoggerType loggerName`), *S1b* - as a `static` attribute, *S1c* - as a `final` attribute in Java, respectively `readonly` in C#, and *S1d* - as a `static` and `final` attribute in Java, respectively `static` and `readonly` in C#. The frequency of occurrence of each of these styles in the analyzed software systems is shown in Table 1. For the Java-based systems, the *S1d* style seems to be preferred, but there are exceptions (eg. Spring). For C# based systems, there is no style that seems to be preferred to the others.

3 CONCLUSIONS

The results of our study show that automatic identification of even one kind of crosscutting concern is still a challenging task. The way a crosscutting concern is implemented depends on different aspects: the programming paradigm and the programming language used, and, very important, on the programmers. Different persons use different styles for implementing a feature, even though they follow the same pattern. In the case of monitoring crosscutting concerns, the specifics that must be considered when developing an approach for automatic identification are:

The existence of more logging toolkits. The use of many logging toolkits influences the results of automatic identification, as toolkits that are used in only a couple of source files have a high probability of not being discovered.

The use of different patterns in the same system. All systems use more than one pattern for monitoring implementation, usually having a predominant pattern and one or more secondary ones. The presence of these secondary patterns may also affect the results of automatic identification, as some of them are not dependent on each other. For example pattern P3 is dependent on P1 and the classes using it could still be identified, but pattern P2 is independent of P1 and P3 and it could easily be overlooked. Also, pattern P5 is dependent on the existence of a *logger* object, that can be obtained using any of the P1, P2, P3 or P4 patterns.

Some patterns are programming language dependent. Pattern P4 is specific to C#-based systems. The approach should either target a particular programming language, or it should be configurable in order to include in their search, patterns specific to different programming languages.

For the same pattern, different implementation styles are used. For pattern **P1** at least 4 different styles were identified, but others could exist. There is no style preferred in all systems, meaning that there is no general style that the approach could look for in a system. A combination of *S1b*, *S1c*, and *S1d* was used in [1] in order to automatically discover the type of the logger object with promising results. The approach successfully identified the most used toolkit for ArgoUML, Tomcat v9 and Spring, but it did not discovered all logging tookits used from the software systems. More research needs to be done in this direction.

Further work needs to also be done in determining how using more than one pattern for monitoring concerns implementation in the same system influences their automatic identification.

REFERENCES

[1] 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.