On A Top Down Aspect Mining Approach for Monitoring Crosscutting Concerns Identification

Grigoreta-Sofia Cojocar
Department of Computer Science
Babeş-Bolyai University
1, M. Kogalniceanu Street
Cluj-Napoca, Romania
Email: grigo@cs.ubbcluj.ro

Adriana-Mihaela Guran
Department of Computer Science
Babeş-Bolyai University
1, M. Kogalniceanu Street
Cluj-Napoca, Romania
Email: adriana@cs.ubbcluj.ro

***Abstract*—*Aspect Mining* is a research domain that tries to identify crosscutting concerns implementation in software systems that were developed without using the aspect oriented paradigm. The goal is to refactor the implementation of these concerns in order to use aspects, and gain the benefits introduced by this paradigm. The aspect mining approaches proposed so far are all bottom-up approaches: starting from the source code of a software system they try to discover all the crosscutting concerns that exist in that system. In this paper we present a top-down aspect mining approach that we developed based on the observations gathered after analyzing how monitoring crosscutting concerns are implemented in three open source Java-based software systems. The approach aims to identify only the implementation of *logging* and *tracing* monitoring crosscutting concerns in Java-based software systems. We also present and discuss the results obtained by applying this approach on four open source Java software systems.**

I. Introduction

The ever increasing complexity of software systems makes designing and implementing them a difficult task. Software systems are usually composed of many different concerns. A concern is a specific requirement or consideration that must be addressed in order to satisfy the overall system. The concerns are classified in core concerns and crosscutting concerns. The core concerns capture the central functionality of a module, while crosscutting concerns capture system-level, peripheral requirements that cross multiple modules. Paradigms like procedural or object oriented programming provide good solutions for the design and implementation of core concerns, but they cannot deal properly with crosscutting concerns. Many different approaches have been proposed for the design and implementation of crosscutting concerns: subject oriented programming [1], composition filters [2], adaptive programming [3], generative programming [4], and aspect oriented programming (AOP) [5]. From these approaches, the aspect oriented programming approach has known the greatest success both in industry and academia.

For almost two decades researchers have tried to develop techniques and tools to (automatically) identify crosscutting concerns in software systems that were already developed without using AOP. This area of research is called *Aspect Mining*. The goal is to identify the crosscutting concerns, and then to refactor them to aspects, in order to obtain a system that can be easily understood, maintained and modified. In order to identify crosscutting concerns, the existing techniques try to discover one or both symptoms that appear when designing and implementing crosscutting concerns using the existing paradigms: code *scattering* and code *tangling*. Code scattering means that the code that implements a crosscutting concern is spread across the system, and code tangling means that the code that implements some concern is mixed with code from other (crosscutting) concerns.

The main contribution of this paper is to propose the first top-down aspect mining approach that tries to identify the implementation of two kinds of monitoring crosscutting concerns: *logging* and *tracing*. The approach does not aim to identify all the crosscutting concerns that exist in a software system, it only focuses on these two kinds of monitoring crosscutting concerns. We also present and discuss the results obtained by applying this approach to four real life open source Java-based software systems.

The rest of the paper is structured as follows. Section II presents an overview of the aspect mining techniques proposed so far. Section III first describes the two types of crosscutting concerns we are interested in, and then it presents the proposed approach for their automatic identification. In Section IV we present the four software systems on which we have applied the proposed approach and we discuss the results obtained. Conclusions and further work are given in Section V and Section VI, respectively.

II. Aspect Mining Techniques

The first approaches in aspect mining were query-based search techniques. The developer had to introduce a so-called seed (eg., a word, the name of a method or of a field) and the associated tool showed all the places where the seed was found. Very soon, researchers discovered that this approach to aspect mining has some important disadvantages: the user of the tool had to have an in-depth knowledge of the analyzed system, as he/she had to figure out the seed(s) to be introduced, and the large amount of time needed in order to filter the results displayed. There were many query based aspect mining tools proposed, like: Aspect Browser [6], The Aspect Mining Tool(AMT) [7], Feature Exploration and Analysis Tool(FEAT)

[8]. All these tools are performing the search in the source code of the mined system.

Since 2004 researchers have focused on developing aspect mining techniques that do not require an initial seed from the user. These techniques try to identify the crosscutting concerns starting just from some kind of system representation (the source code, the requirements documentation, some execution traces, etc.), and are called automated aspect mining techniques. Different approaches are used: clustering [9], [10], [11], clone detection techniques [12], [13], [14], metrics [15], association rules [16], formal concept analysis [17], [18], execution relations [19], [20], self organizing maps [21], and link analysis [22].

All the presently proposed automated aspect mining techniques try to discover all the crosscutting concerns that exist in the mined software system. The obtained results have shown that it is not so easy to develop an approach that can be used for discovering different types of crosscutting concerns. Consequently, the obtained results are not very accurate, and only some types of crosscutting concerns are discovered. If the techniques proposed in the beginning used very different approaches, the last ones are more an improvement of some of the previously proposed techniques. Even so, the results obtained by the new aspect mining techniques did not improve significantly. They obtained better results, but not much better. Also, practice has shown that not all crosscutting concerns can be refactored to aspects.

Mens et al. have conducted an analysis of the problems the proposed aspect mining techniques were encountering [23]. The main problems identified were: poor precision, poor recall, subjectivity, scalability, lack of empirical validation. The study was conducted in 2008 and since then the results obtained by the proposed aspect mining techniques did not improve much.

III. Top-Down Approach

In this section we describe our approach for identifying *logging* and *tracing* monitoring crosscutting concerns.

A. Monitoring Crosscutting Concerns

Monitoring concerns record the behaviour of a software system during development, testing and execution in its own environment. The most commonly used are: 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 and/or the number of times a particular method is invoked.

It is well-known that *tracing* and *performance monitoring* are better implemented using AOP. The AOP-based solution is clearly separated from the rest of the system, can be easily understood and maintained, and it can be easily plugged-in or plugged-out of the system. As for *logging* it is not clear yet if an AOP-based solution can be designed, and if it is better than the non-AOP one.

B. Our Approach

The proposed approach is based on the results obtained in a previous study where we have manually analyzed three Java-based software systems to determine if a pattern (or patterns) can be extracted for monitoring crosscutting concerns [24]. The obtained results have shown that the most similar to a pattern is the declaration of an attribute corresponding to the object used for recording the produced messages, and then calling different methods on it. This object is often called a *logger*. This attribute is in most cases a `static` and/or `final` one (for two of the analyzed systems the percentage is higher than 90%, and for all three case studies the percentage is higher than 60%).

Listing 1 shows a fragment from the source code of `AjpMessage` class from Tomcat v9 [25] (one of the analyzed software systems).

```
package org.apache.coyote.ajp;

import org.apache.juli.logging.*;
import org.apache.tomcat.util.res.StringManager;

public class AjpMessage {

  //The logger object
  private static final Log log =
    LogFactory.getLog(AjpMessage.class);

  // The string manager for this package.
  protected static final StringManager sm =
    StringManager.getManager(AjpMessage.class);

  //Write a MessageBytes out at the
  //current write position.
  public void appendBytes(MessageBytes mb) {
    if (mb == null) {
        log.error(sm.getString("ajpmessage.
          null"), new NullPointerException());
         appendInt(0);
         appendByte(0);
         return;
     }
     // other business logic code
     appendByteChunk(mb.getByteChunk());
  }

  //Write a ByteChunk out at the
  //current write position.
  public void appendByteChunk(ByteChunk bc) {
    if (bc == null) {
        log.error(sm.getString("ajpmessage.
          null"), new NullPointerException());
        appendInt(0);
        appendByte(0);
        return;
       }
     appendBytes(bc.getBytes(), bc.getStart(),
         bc.getLength());
  }

   //other attributes and methods
}
```

Listing 1. Fragment from AjpMessage class from Tomcat v9 [25].

The fragment includes the declaration of the *logger* object (named `log`) as a `static` and `final` attribute. The object is later used in some of the methods to record the corresponding messages. This pattern was identified in most of the classes that recorded messages. Still, there are also a few exceptions to this pattern: the *logger* object is declared in a base class and methods from subclasses only use it without declaring a new attribute, or there are a few cases in which the *logger* object is declared as a local variable in the methods that needed to record messages.

Based on these results we have developed a top-down aspect mining approach that tries to identify *logging* and *tracing* monitoring crosscutting concerns by analyzing the `static` or `final` attributes defined in a Java-based software systems. The approach consists of first automatically identifying the type of the *logger* object, and then the automatic identification of the affected classes (the classes in which the monitoring concerns are implemented). Having identified the classes, then we can (automatically) analyze them to determine whether the concerns can be (automatically) refactored to aspects.

Our approach for identifying the type of the *logger* object consists of the following steps:

1) **Instrumentation.** In order to determine the `static` or `final` attributes defined in a Java-based software system we need to automatically analyze the source code (`.java` files) or the bytecode (`.class` files) of the system. The existing libraries and frameworks that allow us to analyze them (like Soot [26] or Spoon [27]) require and/or use a classpath variable that must be properly set in order to be able to analyze the input (source code or bytecode). During this step we determine all the dependencies (usually other `.jar` files) of the system to be analyzed. This is the most time consuming step, as large software systems may dependent on many different libraries that must be identified by the user if there is no additional information present (such as a Gradle [28] or Maven [29] build file).
2) **Analysis.** After the completion of the first step, we automatically identify all the `static` or `final` attributes defined in the analyzed software system. During this step we gather the following information: the type of the attribute, the number of times this type was used for declaring a `static` or `final` attribute, and the number of distinct classes in which this kind of attributes were defined. We consider that the number of distinct classes in which `static` or `final` attributes of the same type were defined is important, as in the same class many different `static` or `final` attributes of the same type may be defined.
3) **Filtering.** From the results obtained at the previous step we remove 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[]`). During this step we also remove the types that were used for declaring `static` or `final` attributes in less than 3 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, otherwise it can be considered as coupling between the corresponding types (the type of the attribute(s) and the classes in which the `static` or `final` attribute(s) was (were) defined).
4) **Sorting.** The remaining `static` or `final` attributes' types are sorted descending by the number of declaring classes. The first n results will be presented to the user as possible results for the *logger* object's type. From our observations of the manually analyzed software systems the type should be among the first ranked results. The value of n can be decided by the user (or it could have a default value).

After identifying the type of the *logger* object, we consider that *logging* and *tracing* monitoring crosscutting concerns are implemented in all the classes having an attribute of this type. These classes are determined during the analysis of all the attributes defined in the software system, so no additional computation is needed.

IV. Study

A. Case Studies

In order to verify the applicability of the previously described approach we have used four open source Java-based software systems as case studies: Spoon, Tomcat v9, Spring Framework, and ArgoUML:

- **Spoon** is an open-source library that enables transformation and analysis of Java source code. It provides a metamodel where any kind of program element such as a class, a method, a field, a statement, etc. can be accessed for reading and/or modification. The code used for our analysis was downloaded from [30].
- **ArgoUML** is an open source UML modeling tool that includes support for all standard UML 1.4 diagrams. It runs on any Java platform. We have used version 0.34 for our analysis, and the source was downloaded from [31].
- **Apache Tomcat** is an open-source web container for Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies [32]. We analyzed version 9, the source code being downloaded from [33].
- **Spring Framework** is a modular framework that helps developing Java enterprise applications by providing a comprehensive programming and configuration model [34]. The developers focus on the application-level business logic, and the framework helps putting together the final system. The source code that we have used for our analysis was downloaded from [35].

Two of these case studies (Tomcat v9 and Spring Framework) were among the systems considered for our previous

TABLE I
CASE STUDIES

Case study	Number of classes and interfaces analyzed (CI)
Spoon	724
ArgoUML	2247
Tomcat v9	2502
Spring	5235

manual analysis, but the other two were not considered in our previous study from [24]. Table I presents the number of classes and interfaces analyzed for each case study.

For the automatic analysis step we have used Soot [26], a Java optimization framework that provides various representations for analyzing and transforming Java bytecode. In order to obtain the bytecode of the systems for which we only had the source code, we have first built the systems by following the instructions described on the corresponding websites.

B. Results

Table II presents the results obtained after executing the **Analysis** and **Filtering** steps of our approach. As the results show the number of `static` or `final` attributes defined in a large Java-based software systems is big. In all four case studies the number is greater than the number of classes and interfaces analyzed, and for ArgoUML and Tomcat the number is even bigger than the doubled number of classes and interfaces. However, if we consider only the attributes' type their number decreases significantly. In all cases, the number of different types is less than 25% of the number of `static` or `final` attributes. In three of the cases (ArgoUML, Tomcat V9, and Spring) the percentage of different types over the number of `static` or `final` attributes is even less than 16%, and in the case of ArgoUML it is even less than 8% meaning that only a small part of this information is necessary for the identification of the *logger* object type. The results also show that after the **Filtering** step more than 80% of the types are removed, meaning that they are either types which are commonly used for the business logic of a system (like the primitive types or the types from the `java.util` or `java.lang` packages) or they are not crosscutting (they were used in at most 2 different classes). For all four case studies the number of possible types for the *logger* object is less than 20% of the total number of types used for declaring `static` or `final` attributes, and it is less than 4% of the total number of `static` or `final` attributes defined in the system.

In Table III is presented a subset of the results obtained after executing the **Sorting** step for each case study. Column **DA** represents the number of times a `static` or `final` attribute of the corresponding type was declared, **DC** represents the number of classes that declare a `static` or `final` attribute of this type, **TDC** represents the number of classes that declare an attribute of this type (independently of the modifier used: `static`, `final` or none), **TDC/CI** represents the ratio of the number of declaring classes over the total number of classes and interfaces from the system, **CCC** represents the number of classes from the software system in which the concerns are implemented (the number is determined during the manual analysis of the system), and **ACC** represents the accuracy of our approach. The accuracy is considered to be the percentage of classes from the software system that are part of the crosscutting concerns implementation and were identified as such by our approach. For this study the accuracy is computed as the percentage of **TDC/CCC**.

For the three larger case studies (ArgoUML, Tomcat v9, and Spring) the type ranked at the first position is the type of the *logger* object, showing that this approach could be used for automatic identification of *logging* and *tracing* monitoring crosscutting concerns. For these three case studies the percentage of **TDC/CI** of the type ranked at the first position is also significantly greater than the percentage of the type ranked at the second position. The results also show that the type used for the *logger* object is different for all three case studies, meaning that it is dependent on the software system.

In the case of Spoon case study the top 3 ranked types do not include the type of the *logger* object. The most used type for declaring `static` or `final` attributes is `ReplacementVisitor` from `spoon.support.visitor.replace` package which was used in more than 10% of the classes from system. In this case the type of the *logger* object was ranked only at the position 15, being declared as a `static` or `final` attribute in only 4 classes, meaning less than 1% of the total number of classes. A more in-depth analysis has determined that the type was actually used in 5 classes, but in one class the *logger* object was not declared neither as a `static` nor a `final` attribute.

The study presented in [24] has also shown that there are software systems which use more than one type for the *logger* object. The manual analysis of two of the systems used in this study have shown that they use more than one type for the *logger* object: Tomcat v9 uses 2 types, and Spring Framework uses 3 types. For Tomcat case study the results obtained by our approach actually included both types, but the second type was ranked at the position 96 (from 136 possible positions), as shown in Table III. The second type was used only in 4 classes, and the *logger* object was declared as a `static` or `final` attribute in only 3 of the classes. In the case of Spring Framework, only one type was given among the possible results. The other two types were not included because they were used in at most 4 classes, and in some of these classes the *logger* object was not defined as a `static` or `final` attribute.

As shown in Table III, the accuracy of our approach is higher than 75% for all the three larger case studies, for two of them the accuracy being even higher than 90%. Even if for Spoon case study, where the type of the *logger* object is not among the top 3 ranked possible types, after choosing the correct type, the accuracy of our approach is 100%.

TABLE II
CASE STUDIES RESULTS - ANALYSIS AND FILTERING STEPS

Case study	Number of `static` or `final` attributes	Different types (DT)	Percentage of DT/attributes	Number of types after filtering (TF)	Percentage of TF/DT	Percentage of TF/attributes
Spoon	879	213	24.23%	35	16.43%	3.98%
ArgoUML	5374	394	7.33%	57	14.35%	1.06%
Tomcat v9	6630	733	11.05%	136	18.55%	2.05%
Spring	7776	1198	15.40%	214	17.86%	2.75%

TABLE III
CASE STUDIES RESULTS -SORTING STEP

Case study	Rank	Attribute's type	DA	DC	TDC	TDC/CI	CCC	ACC
ArgoUML	**1**	**org.apache.log4j.Logger**	**253**	**253**	**253**	**11.25%**	**324**	**78.08%**
	2	org.argouml.language.java.reveng.JavaParser	89	89	89	3.96%	-	-
	3	org.argouml.configuration.ConfigurationKey	93	23	24	1.06%	-	-
Tomcat v9	**1**	**org.apache.juli.logging.Log**	**274**	**274**	**276**	**11.03%**	**288**	**95.83%**
	2	org.apache.tomcat.util.res.StringManager	188	187	187	7.47%	-	-
	3	org.apache.catalina.tribes.util.StringManager	37	37	37	1.47%	-	-
	...	...	...	...	...	...	-	-
	96	**java.util.logging.Logger**	**4**	**3**	**4**	**0.15%**	**4**	**100%**
Spring	**1**	**org.apache.commons.logging.Log**	**357**	**355**	**365**	**6.97%**	**402**	**90.76%**
	2	java.lang.reflect.Method	117	76	94	1.79%	-	-
	3	org.springframework.orm.hibernate3.HibernateTemplate	39	39	40	0.76%	-	-
Spoon	1	spoon.support.visitor.replace.ReplacementVisitor	82	82	82	11.32%	-	-
	2	spoon.reflect.factory.Factory	7	7	22	3.03%	-	-
	3	spoon.reflect.visitor.chain.CtQueryImpl	7	7	7	0.96%	-	-
	...	...	...	...	...	...	-	-
	15	**org.apache.log4j.Logger**	**4**	**4**	**5**	**0.69%**	**5**	**100%**

V. CONCLUSIONS

The conclusions that can be drawn from the results obtained in Section IV are:

- The proposed approach can be used for the identification of *logging* and *tracing* monitoring crosscutting concerns. The results obtained for the three larger case studies have shown that the analysis of `static` or `final` attributes of a software system is a good starting point for the identification of the type of the *logger* object used for these crosscutting concerns implementation. The set of classes in which an attribute of this type is defined is also a good starting point for determining all the affected parts of the software systems. On these classes we can perform a more in-depth analysis in order to determine, for example, the methods in which the attribute is used and to also determine if the implementation can be refactored to aspects. As Table III shows the searching space is significantly reduced, less than 15% of the total number of classes and interfaces need to be considered for the in-depth analysis.
- This approach is *scalable*. As the results of the study have shown, the number of types considered as possible results are less than 6% of the total number of classes and interfaces. Even for large or very large software systems, the possible results for the *logger* object's type are reduced significantly. The time needed to obtain all the possible types is also small. It takes less than 3 seconds to obtain the possible types for any of our case studies. Identifying all the other parts of the concerns implementation may take longer, but it should still be an acceptable amount of type.
- The majority of the already proposed aspect mining techniques try to identify crosscutting concerns by analyzing the methods defined in a software system. However, the results obtained show that for *logging* and *tracing* monitoring crosscutting concerns, a different granularity provides more accurate results.
- The results of this approach can be considered as input for automatically refactoring the implementation of these monitoring crosscutting concerns into aspects. This approach can also be used to determine if the implementation can actually be refactored into aspects. In the analysis described in [24] we have determine that at least 25% of the messages recorded using the *logger* object are constructed using local variables. This kind of monitoring messages cannot be refactored to aspects.

We did not include in this paper a comparison of our approach with other already proposed approaches as it is difficult to compare them due to their different granularities. Also, they do not automatically separate the results based on the kind of crosscutting concerns, letting the user decide which results belongs to which crosscutting concerns.

VI. Further Work

In this paper we have presented a top-down approach for automatic identification of *logging* and *tracing* monitoring crosscutting concerns implementation. The approach analyzes the attributes defined in a Java-based software system in order to determine the *logger* object's type, and then determines all the affected classses. We have used the proposed approach on four open source Java-based software systems.

Further work will be done in the following directions:

- To apply the proposed approach on other open source (larger) case studies.
- To determine if the proposed approach can be used for software systems developed using other programming languages like C# or C++.
- To develop a plugin for a popular IDE like IntelliJ or Eclipse that will allow the automatic identification of *logging* and *tracing* monitoring crosscutting concerns.
- To determine if refactoring is possible for the analyzed software systems.
- To evaluate if the structure of the system would improve if refactoring to aspects is possible.

References

[1] W. Harrison and H. Ossher, "Subject-Oriented Programming: A Critique of Pure Objects," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93, New York, NY, USA, 1993, pp. 411–428.

[2] M. Aksit, "On the design of the object oriented language sina," Ph.D. dissertation, Department of Computer Science, University of Twente, The Netherlands, 1989.

[3] K. J. Lieberherr, "Component Enhancement: An Adaptive Reusability Mechanism for Groups of Collaborating Classes," in *Information Processing '92, 12th World Computer Congress*, J. van Leeuwen, Ed. Madrid, Spain: Elsevier, 1992, pp. 179–185.

[4] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings European Conference on Object-Oriented Programming*, vol. LNCS 1241. Springer-Verlag, 1997, pp. 220–242.

[6] W. G. Griswold, Y. Kato, and J. J. Yuan, "AspectBrowser: Tool Support for Managing Dispersed Aspects," UCSD, Tech. Rep. CS1999-0640, March 2000.

[7] J. Hannemann and G. Kiczales, "Overcoming the Prevalent Decomposition of Legacy Code," in *Advanced Separation of Concerns Workshop,at the International Conference on Software Engineering (ICSE)*, May 2001.

[8] M. P. Robillard and G. C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 406–416.

[9] D. Shepherd and L. Pollock, "Interfaces, Aspects, and Views," in *Proceedings of Linking Aspect Technology and Evolution Workshop(LATE 2005)*, March 2005.

[10] G. S. Moldovan and G. Serban, "Aspect Mining using a Vector-Space Model Based Clustering Approach," in *Proceedings of Linking Aspect Technology and Evolution (LATE) Workshop*. Bonn, Germany: AOSD'06, March, 20 2006, pp. 36–40.

[11] L. He and H. Bai, "Aspect Mining using Clustering and Association Rule Method," *International Journal of Computer Science and Network Security*, vol. 6, no. 2, pp. 247–251, February 2006.

[12] D. Shepherd, E. Gibson, and L. Pollock, "Design and Evaluation of an Automated Aspect Mining Tool," in *2004 International Conference on Software Engineering and Practice*. IEEE, June 2004, pp. 601–607.

[13] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[14] O. A. M. Morales, "Aspect Mining Using Clone Detection," Master's thesis, Delft University of Technology, The Netherlands, August 2004.

[15] M. Marin, A. van, Deursen, and L. Moonen, "Identifying Aspects Using Fan-in Analysis," in *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*. IEEE Computer Society, 2004, pp. 132–141.

[16] S. Vidal, E. S. Abait, C. Marcos, S. Casas, and J. A. Díaz Pace, "Aspect mining meets rule-based refactoring," in *Proceedings of the 1st Workshop on Linking Aspect Technology and Evolution*, ser. PLATE '09. New York, NY, USA: ACM, 2009, pp. 23–27.

[17] P. Tonella and M. Ceccato, "Aspect Mining through the Formal Concept Analysis of Execution Traces," in *Proceedings of the IEEE Eleventh Working Conference on Reverse Engineering (WCRE 2004)*, November 2004, pp. 112–121.

[18] T. Tourwé and K. Mens, "Mining Aspectual Views using Formal Concept Analysis," in *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop on (SCAM'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 97–106.

[19] S. Breu and J. Krinke, "Aspect Mining Using Event Traces," in *Proceedings of International Conference on Automated Software Engineering (ASE)*, 2004, pp. 310–315.

[20] J. Krinke and S. Breu, "Control-Flow-Graph-Based Aspect Mining," in *Workshop on Aspect Reverse Engineering (WARE)*, 2004.

[21] S. G. Maisikeli, "Aspect mining using self-organizing maps with method level dynamic software metrics as input vectors," Ph.D. dissertation, 2009.

[22] J. Huang, Y. Lu, and J. Yang, "Aspect mining using link analysis," in *Proceedings of the 2010 Fifth International Conference on Frontier of Computer Science and Technology*. IEEE Computer Society, 2010, pp. 312–317.

[23] K. Mens, A. Kellens, and J. Krinke, "Pitfalls in Aspect Mining," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, ser. WCRE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 113–122.

[24] G. S. Cojocar, "On Top-Down Aspect Mining for Monitoring Techniques Implementation," in *Proceedings of IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, 2016, pp. 249–254.

[25] "Apache Tomcat," http://tomcat.apache.org/.

[26] "Soot: a Java Optimization Framework," http://www.sable.mcgill.ca/soot/.

[27] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[28] "Gradle Build Tool," https://gradle.org/, Retrieved September, 2017.

[29] "Apache Maven Tool," https://maven.apache.org/, Retrieved September, 2017.

[30] "Spoon source code," https://github.com/INRIA/spoon, Retrieved July, 2017.

[31] "Argouml source code," http://argouml-downloads.tigris.org/nonav/argouml-0.34/ArgoUML-0.34-src.tar.gz, Retrieved March, 2016.

[32] "Apache tomcat," http://tomcat.apache.org/index.html, Retrieved January, 2016.

[33] "Apache tomcat v9," http://tomcat.apache.org/download-90.cgi, Retrieved January, 2016.

[34] "Spring framework," https://spring.io/, Retrieved January, 2016.

[35] "Spring framework source code," https://github.com/spring-projects/spring-framework, Retrieved January, 2016.