Aspect Mining meets Rule-based Refactoring

Santiago A. Vidal, Esteban S. Abait, Claudia Marcos
ISISTAN Research Institute, Faculty of Sciences, UNICEN University
Campus Universitario, Pje. Arroyo Seco
(B7001BBO) Tandil, Buenos Aires, Argentina
{svidal, cmarcos}@exa.unicen.edu.ar
eabait@alumnos.exa.unicen.edu.ar

Sandra Casas
UNPA University
Lis. De la Torre 1060
(CP 9400) Río Gallegos, Santa Cruz, Argentina
lis@uarg.unpa.edu.ar

J. Andrés Díaz Pace
Software Engineering Institute, Carnegie Mellon University
4500 Fifth Ave.,
Pittsburgh PA, 15232, USA
adiaz@sei.cmu.edu

ABSTRACT
Aspect-oriented software development allows the encapsulation of crosscutting concerns, achieving a better system modularization and, therefore, improving its maintenance. One important challenge is how to evolve an object-oriented system into an aspect-oriented one in such a way the system structure gets gradually improved. This paper describes a process to assist developers in the refactoring of object-oriented systems to aspects. To do so, we propose a tool approach that combines aspect mining techniques with a rule-base engine to apply refactorings.

1. INTRODUCTION
Aspect-oriented software development (AOSD) [13] is a paradigm that supports the encapsulation of the concerns that orthogonally crosscut the components of a system by means of aspects. These concerns are called crosscutting concerns (CCCs). CCCs cannot be easily modularized using traditional software engineering approaches (e.g., the object-oriented paradigm) to deal with the complexity and evolution of systems. Typical examples of CCCs are exception handling, logging and concurrency control.

For existing object-oriented systems to incorporate the benefits of AOSD, those systems are usually re-modularized into aspect-oriented systems. This leads to a need for techniques and tools that can help developers with the identification of crosscutting concerns, called *aspect mining* [12], and then with the refactoring of those concerns into aspects, called aspect refactoring [12]. Aspect mining enables the discovery of crosscutting concerns in the source code that can potentially become aspects (also known as *candidate aspects*). Aspect refactoring is the technique that accomplishes the necessary transformations in the code to turn the candidate aspects into aspectual code.

In this paper, we propose a comprehensive approach to perform the gradual evolution of an object-oriented system to an aspect-oriented one. This approach aims at assisting the developer in: performing the evolution process, automating many tasks involved in this process, taking advantage of precise aspect mining techniques, and applying different types of aspect refactorings.

We think that the migration from an OO system to an AO one improves the structure and quality of the software, and thus eases software evolution. Along this line, we believe that the provision of semi-automated support to help the developer to discover crosscutting concerns and to encapsulate them into aspects is really beneficial. A novelty of our approach is the use of dynamic analysis together with data mining techniques for identifying candidate aspects. Also, we present an aspect refactoring process based on existing types of refactorings which automates the major steps of the migration.

The rest of the paper is structured as follows. Section 2 describes the evolution process. Section 3 explains the details of the aspect mining approach. Section 4 describes the aspect refactoring support, and also how it is integrated with aspect mining. Finally, Section 5 presents some lessons learned and discussion.

2. THE APPROACH
The proposed approach consists of two main phases (see Figure 1): (i) aspect mining, and (ii) aspect refactoring. The first phase receives an object-oriented system (to be evolved) as input, and produces a number of candidate aspects as output. These aspects are identified by making a dynamic analysis of the system and applying association rules. The information of candidate aspects and the initial system's source code are then passed to the aspect refactoring phase. In this phase, different refactorings are evaluated and eventually applied to the code. As output, this second phase generates a new version of the system that contains aspect-oriented final code.

The whole approach is supported by an Eclipse-based prototype tool called AspectRT (Aspect Refactoring Tool). This tool helps developers to carry out the evolution process by automating parts of the tasks involved in each phase.

3. ASPECT MINING PHASE
Our aspect mining approach is based on the fact that it is possible to get the most relevant method associations from the system's

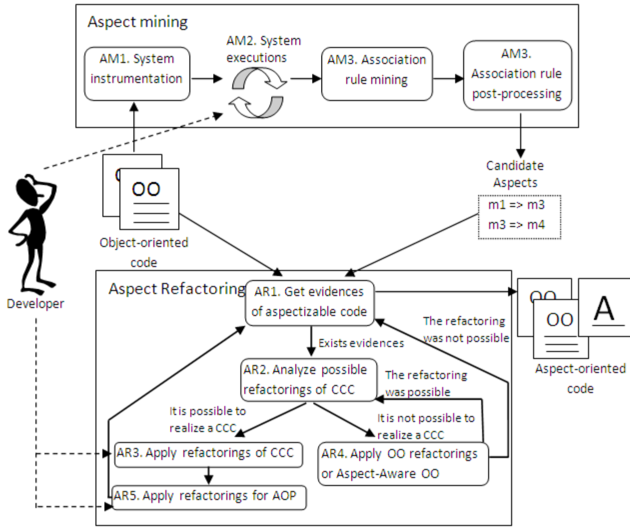

Figure 1. Integration of aspect mining with aspect refactoring

execution traces obtained using dynamic analysis [1]. This kind of associations gives developers valuable information about the behavior of the system and allows them to identify scattering symptoms.

The basic idea behind dynamic analysis algorithms is to observe run-time behaviors of software systems and extract information from the execution of the programs [4]. The approach described here is based on association rules [2]. It takes two pieces of information as input: execution traces and execution relations. The execution traces and relations are obtained by running the program under given scenarios. Each scenario can be seen as a instance of a use case [3]. The program trace is the sequence of method invocations during the execution of the program, and the execution relations registers the invocations from one method to another.

As an example, Figure 2 shows the use of the Observer design pattern [7] in a simple GUI application. The intent of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [7]. In this Observer implementation (initially presented in [10]), the Point class plays

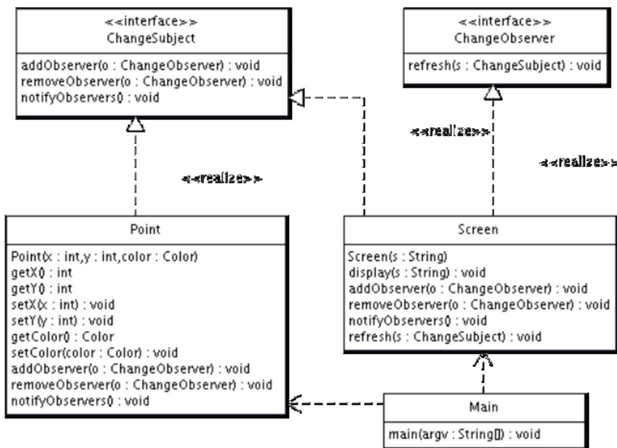

Figure 2. UML class diagram for the program under analysis

the subject role and the Screen plays the roles of both subject and observer (of Point and of itself). Here, we can run the scenario “the point changes its color” and obtain the trace and execution relations shown in Figure 3. The resulting trace contains the sequence of method invocations shown by the table atop of Figure 3. The execution relations for this trace are represented by the two columns at the bottom of that figure.

Change color trace

void observer.Point.addObserver(ChangeObserver);
void observer.Screen.addObserver(ChangeObserver);
void observer.Point.setColor(Color);void observer.Point.notifyObservers();
void observer.Screen.refresh(ChangeSubject);void observer.Screen.display(String);
void observer.Screen.notifyObservers();

Change color execution relations

Caller	Invoked
void observer.Point.setColor(Color)	void observer.Point.notifyObservers()
void observer.Point.notifyObservers()	void observer.Screen.refresh(ChangeSubject)
...	...
void observer.Screen.refresh(ChangeSubject)	void observer.Screen.display(String)
void observer.Screen.display(String)	void observer.Screen.notifyObservers()

Figure 3. Trace (atop) and execution relations (bottom).

The box at the top of Figure 1 depicts the steps of the proposed mining technique. The first and second steps (System instrumentation and System executions) correspond to the collection of runtime information about the system. The third step (Association rule mining) takes the set of traces as input and uses an association rule algorithm to find interesting associations among methods. The fourth step (Association rule post-processing) classifies rules in terms of scattering indicators, and removes redundant rules as well as rules with utility methods such as 'main' or 'run' [1]. Rules that cannot be classified are discarded.

3.1 Use of Association Rules

If each trace of the system under analysis is considered as a transaction T and the methods contained in all the traces as the set of items I, it is possible to get a dataset D from which a set of association rules can be generated. For example, the rules for the example shown in Table 1 will have the following form: Point.notifyObservers ⇒ Screen.refresh (support: 1.0, confidence: 1.0). The support value of the rule indicates the number of traces (transactions) in which both methods are present. In our example, the support value indicates that the two methods are present together in all the traces. On the other hand, the confidence value points out the stability of the method relation. Then, a confidence value of 1.0 means that each time the 'notifyObservers' method is called so is the 'refresh' method. For the proposed technique, the generated rules have only one item in its antecedent and one item on its consequent. We believe that these kinds of rules can be easily generated and even understood by developers.

In order to characterize the kind of rules that are interesting for the aspect mining process, let’s briefly show how the association rule algorithm works on the Observer example (Figure 2). Two scenarios are used to exercise the implementation: a) "a point changes its color", b) "a point changes its position". The AspectRT tool permits to obtain the execution traces and relations by means of a tracing aspect that registers all the method invocations and its relations.

Table 1. Final set of rules for the Observer pattern example.

Concern	Rule	Filter	Supp.	Conf.
Subject-Observer Mapping	Screen.addObserver ⇒ Point.addObserver	Name filter	1.0	1.0
Notification mechanism	Screen.notifyObservers ⇒ Point.notifyObservers	Name filter	1.0	1.0
Notification mechanism	Point.setColor ⇒ Point.notifyObservers	Recurrent consequent	0.5	1.0
Notification mechanism	Point.setX ⇒ Point.notifyObservers	Recurrent consequent	0.5	1.0
Update logic	Point.notifyObservers ⇒ Screen.refresh	Recurrent consequent	1.0	1.0
Update logic	Screen.notifyObservers ⇒ Screen.refresh	Recurrent consequent	1.0	1.0

When running the Apriori algorithm [2] over the traces with support value of 0.1 and confidence value of 0.1, it generated 70 rules. The resulting set of rules demonstrates the importance of the post-processing step, and furthermore, the need for classification filters that can provide more information for each rule. For example, rules that include methods like 'main' (rules 1 and 2), 'toString', 'hashCode' are not interesting for aspect mining purposes. Thus, a filter must remove rules that include those irrelevant methods. Redundant rules also must be removed from the final list of rules. For example, rules 5 and 6 show the same association between methods and have the same support and confidence value. Hence, another filter must remove the redundant rules.

3.2 Classification Filters

In AspectRT, the classification of the association rules is done by two filters. The first filter, called Naming Filter, looks for methods that have the same name and are called together whereas the second one, called Recurrent Consequent Filter, looks for rules that share the same consequent. The two filters are described below.

Naming Filter: Rules like Screen.addObserver ⇒ Point.addObserver (support: 1.0, confidence:1.0) could be indicators of a concern that is scattered over two different methods. This is not only because they share the same name (addObserver), but because they are present together in more than one execution trace (high confidence and support values). This means that both methods were called during the system execution for more than one scenario, thus both methods could correspond to the implementation of the same concern. This latter condition avoids many false positive that could arise if we only consider the syntactic nature of the method names. For this kind of filter, the confidence value says how semantically related both methods are, since the confidence indicates how many times the antecedent method is executed in conjunction with the consequent method. High confidence means a strong semantic relation between the involved methods.

The naming filter is simply defined as follows: given an association rule A ⇒ B, where A and B are methods, the name of A must be equal to the name of B.

Recurrent Consequent Filter: When two or more rules share the same consequent (for example, rules 3 and 4 of Table 1), the immediate assumption is that the method of the consequent is consistently invoked from the methods included in the antecedents of the rules. The method of the consequent could be implementing functionality that is required from various places of the system (like a 'log' method). Therefore, the existence of such method is an indicator of scattering symptom on the system.

The recurrent consequent filter is defined as follows: given an association rule A ⇒ B, where A and B are methods, the following conditions must hold:

- A and B must be in a execution relation where A is the invoker and B is the invoked method,
- B must be included as a consequent in another association rule C ⇒ B that also is in a execution relation where C is the invoker and B is the invoked method.

The application of these two filters along with the redundant rules and the irrelevant methods filters yield the rules shown in Table 1. The concern column of the table must be completed by the developer of the technique after manual investigation of each rule on the source code.

4. ASPECT REFACTORING PHASE

A variety of aspect refactorings have been proposed over the last years [7]. In this context, and in order to facilitate the evolution process, it is desirable to have tools able to support current and future refactorings. Our aspect refactoring approach is based on different kinds of aspect refactorings. Specifically, we use the following classification [9]:

- *Aspect-Aware OO Refactorings*: This includes those object-oriented refactorings which were extended and adapted to be used in the aspect-oriented paradigm. That is, this type of refactoring ensures that the OO refactorings correctly update the references to the AOP constructions. The Aspect-Aware OO refactorings have been discussed in [8] [11].
- *Refactorings for AOP constructs*: The refactorings grouped under this type have the property of being specifically oriented to elements of the aspect-oriented programming. Its objective is basically to improve the internal structure of aspects so that they are more legible and modifiable ([11] [15] [16]).
- *Refactorings of CCCs (Crosscutting concerns)*: The objective of this third group is to transform the crosscutting concerns in aspects. Regarding the elemental idea of the aspect-oriented paradigm, these refactorings group the different concerns that are dispersed throughout the code when modularizing them into an aspect ([14] [15]).

The proposed approach follows an iterative process that starts with an object-oriented code and evidences of "aspectizable" code. This evidence is actually provided by the candidate aspects resulting from the aspect mining approach presented in Section 3. Each cycle of the process produces a code refactoring by adding aspect-oriented code in AspectJ. For each piece of evidence that suggests aspectizable code in the system, we have to evaluate the application of one or more aspect refactorings that transform parts of the code into an aspect.

The main steps of the refactoring approach, as shown at the bottom of the Figure 1, are the following:

1. *Get evidences of aspectizable code*: This step recovers the code that has been identified as aspectizable by the aspect mining phase. That is, there is a description of OO code attributes, methods, classes, etc. that should be refactorized to encapsulate the crosscutting concerns into aspects. The connection with the aspect mining process is achieved through a XML file, which contains a list of candidate aspects with relevant data about those aspects.

2. *Analyze possible refactorings of CCCs*: This step examines the possibility of applying one refactoring of CCCs (or a group of them) to the target code. That is, a set of viable refactorings is selected. The reason for using CCC refactorings in this step is because the fragments of aspectizable code identified in the previous step contain crosscutting concerns that must be encapsulated into an aspect.

3. *Apply refactoring of CCCs*: The refactorings selected previously are executed, so that every crosscutting concern is extracted from the object-oriented code and inserted as an aspect. The code refactorings are applied automatically by the AspectRT tool. Eventually, the developer's intervention is necessary for some decisions, such as: the choice of an aspect in which a fragment of code will be encapsulated, the name of a new pointcut, etc.

4. *Apply OO refactorings or Aspect-Aware OO*: If it is not possible to apply any refactoring of CCCs, this step seeks to apply object-oriented refactorings and/or aspect-aware OO ones on the target code in order to restructure it and retry step 2. Sometimes, the identified code cannot be encapsulated directly into an aspect, and a previous OO refactoring is needed for the OO code to be adapted to the aspect refactoring pattern. For example, if the aspect refactoring Move Method from Class to Inter-type [15] is needed and the selected method contains logic that should stay in the class, the refactoring Extract Method [6] must be applied to the fragment of code that contains that logic.

5. *Apply refactoring for AOP constructs*: At last, this step tries to apply refactorings for AOP constructs to the aspect that has been modified in the application of refactorings of CCCs. Sometimes, when extracting a crosscutting concern, multiples refactorings are applied. So, the internal structure of the aspect that encapsulates the aspectizable code can need refactoring to improve its legibility and modularity, remove duplicate code, etc. For example, this situation may happen after repeatedly applying the aspect refactoring Extract Fragment into Advice [15]. The goal of Extract Fragment into Advice is to encapsulate a fragment of objective code into an aspect creating a new advice and a pointcut. Because of this refactoring, repeated points may appear in the new aspect. If so, the duplicate pointcuts are removed and the advice references are updated accordingly. This way, the approach ensures that not only the crosscutting concerns selected by the developer are encapsulated into an aspect, but also the internal structure of aspects is improved.

4.1 Identification of Refactorings

In order to identify the refactorings that can be applied to aspectizable code (AR1), we are currently using a rule-based paradigm [5]. The inference engine can identify code smells [16] from a class or structural patterns, and then infer a set of possible refactorings for the current context. When a set of aspect refactorings is identified, the tool informs the developer about it. The developer is responsible for accepting or refusing the refactoring of this code smell. The code smells supported by our tool have been grouped in three categories: tangling and scattering code, abstract class and inner class.

The structural patterns serve to delimit a subset of refactorings to be applied in an aspectizable code. These patterns use the information of the aspectizable code of the iteration, that is, they look whether the code is a method, a field, code inside a method, an inner class, etc. Based on this information, the engine can infer possible aspect refactorings to apply on the code.

The code smell and structural patterns are implemented in terms of simple rules like:

```
If (the aspectizable code is a method)
then (try these possible refactorings:
      Move Method from Class to Inter-type
      Extract Feature into Aspect)
```

For example, the first rule in Table 1 presents the method addObserver. During the iterations, the tool can identify a set of aspect refactorings to be applied. In this case, given that the aspectizable code is a method, it is possible to apply aspect refactorings like Move Method from Class to Inter-type and Extract Feature into Aspect [15]. Then, the developer must choose one of the refactorings. The Extract Feature into Aspect refactoring is appropriate, because a complete crosscutting concern needs to be encapsulated. When the developer chooses the aspect refactoring, the tool executes the changes on the source code. In the example of addObserver method, a new aspect called

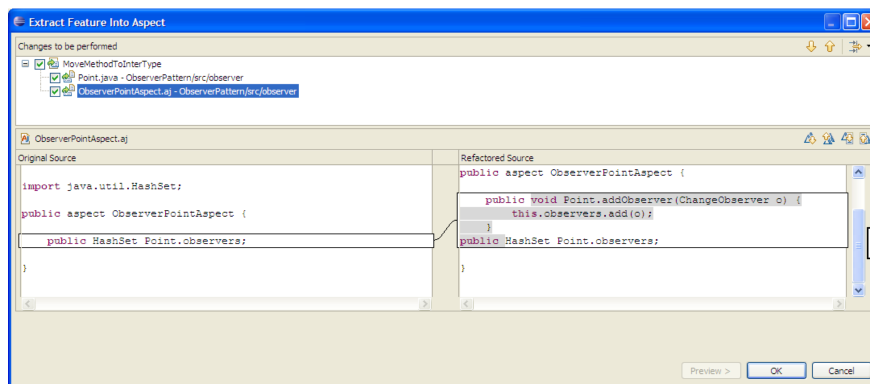

Figure 4. Wizard for applying Extract Feature Into Aspect refactoring.

ObserverPointAspect is created, and the variable observers and this method are encapsulated in the aspect. This situation is described in Figure 4.

Later, the process tries to execute the step 5. As the code related to this step is the target code, no automatic AOP refactoring identification is provided. For this reason, the analysis of fragments of code in which the refactoring can be applied is left to the developer. The tool assists the developer indicating which AOP refactorings are applicable to the selected fragment of code, and applying it automatically. In the example, such a refactoring is not necessary because the aspect's structure is very simple. Finally, the process goes back to step 1, in order to analyze the next candidate aspect.

5. DISCUSSION

This paper presents a proposal for a refactoring process that assists the evolution of an object-oriented system into an aspect-oriented system. We have developed a tool approach that combines an aspect mining technique and an aspect refactoring technique. On one hand, the aspect mining technique is based on dynamic analysis and association rules. The main advantages of the technique are: the automatic identification of scattering symptoms, and the generation of expressive rules describing the crosscutting. On the other hand, the aspect refactoring technique relies on a rule-based paradigm. The main advantages of this technique are: the integration of different kinds of refactorings, and the automation of the transformations.

At this moment, the process has been tested with three small case studies (one of them is the Observer pattern used in the paper) of about 100 lines of code (LoC) each one. The results have demonstrated the potentialities of the approach, reducing the coupling between object classes and the LoC of the involved classes (in about 40%), and increasing the modularity. However, some problems and open issues still remain. For instance, how to include automated identification of aspect code to be refactorized (Step 5 of AR) and how to give assistance to the developer for deciding which aspect refactoring should be selected (Step 2 of AR).

Also, we have started to investigate the use of 3D visualization techniques in AspectRT, when performing code explorations for refactorings. The goal of this feature is that of showing more effective visualizations to the developer about relations between packages, classes, aspects, methods and crosscutting concerns. This way, the developer has a high-level vision of the current system architecture and possible evolution paths for it. The developer can then decide which aspects should be created, resolve encapsulation issues for these aspects, and check the effects of possible refactorings on the system.

As future work, we will compare the proposed aspect mining technique with others dynamic approaches, in order to improve the existing knowledge on this kind of techniques. We also plan to define strategies and mechanisms to support a dynamic identification of AOP refactorings.

6. REFERENCES

[1] E.S. Abait, S.A. Vidal and C.A. Marcos. Dynamic Analysis and Association Rules for Aspects Identification. *II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2008)*, Campinas, Brasil, 2008.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *In Readings in Database Systems (3rd Ed.)*, pages 580-592, San Francisco, CA. Morgan Kaufmann Series In Data Management Systems. Morgan Kaufmann Publishers, 1998.

[3] G. Booch, J. Rumbaugh and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[4] S. Breu and J. Krinke. Aspect Mining Using Event Traces. *In Proceedings of the 19th IEEE international Conference on Automated Software Engineering*. Automated Software Engineering. IEEE Computer Society, 2004.

[5] S. Casas and C.A. Marcos. Exploración de Reglas de Inferencia para Automatizar la Refactorización Aspectual. *II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2008)*, Campinas, Brasil, 2008.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design patterns - Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[8] S. Hanenberg, C. Oberschulte and R. Unland. Refactoring of aspect-oriented software. *In 4th International Conf. on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, pages 19-35, Erfurt, Germany, 2003.

[9] J. Hannemann. Aspect-Oriented Refactoring: Classification and Challenges. *Workshop on Linking Aspect Technology and Evolution (LATE'06). 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, 2006.

[10] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. *In Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161-173. ACM Press, 2002.

[11] M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. *In Proc. of 4th AOSD Modeling With UML Workshop, UML'2003*, San Francisco, USA, 2003.

[12] A. Kellens and K. Mens. A survey of aspect mining tools and techniques. Technical Report 2005-08, INGI, UCL, Belgium, 2005.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin. Aspect-Oriented Programming. *In Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220-242, 1997.

[14] M. Marin, L. Moonen and A. Van Deursen. An approach to aspect refactoring based on crosscutting concern types. *In Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1-5, St. Louis, Missouri. ACM Press, 2005.

[15] M.P. Monteiro. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECSD-200401, Universidade do Minho, 2004.

[16] M.P. Monteiro and J.M. Fernandes. Towards a catalog of aspect-oriented refactorings. *In Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, Chicago, Illinois. ACM Press, 2005.