Volume 86, issue 6, June 2013
ISSN 0164-1212
ELSEVIER
The Journal of Systems and Software

The Journal of Systems and Software 86 (2013) 1482–1497

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Toward automated refactoring of crosscutting concerns into aspects

Santiago A. Vidal[a,b,*], Claudia A. Marcos[a,c]

[a] ISISTAN Research Institute, Faculty of Sciences, UNICEN University, Campus Universitario, B7001BBO Tandil, Buenos Aires, Argentina
[b] CONICET, Concejo Nacional de Investigaciones Científicas y Técnicas, Argentina
[c] CIC, Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina

A R T I C L E   I N F O

Article history:
Received 8 November 2011
Received in revised form
18 December 2012
Accepted 18 December 2012
Available online 28 December 2012

Keywords:
Software evolution
Separation of concerns
Aspect refactoring

A B S T R A C T

Aspect-oriented programing (AOP) improves the separation of concerns by encapsulating crosscutting concerns into aspects. Thus, aspect-oriented programing aims to better support the evolution of systems. Along this line, we have defined a process that assists the developer to refactor an object-oriented system into an aspect-oriented one. In this paper we propose the use of association rules and Markov models to improve the assistance in accomplishing some of the tasks of this process. Specifically, we use these techniques to help the developer in the task of encapsulating a fragment of aspectizable code into an aspect. This includes the choice of a fragment of aspectizable code to be encapsulated, the selection of a suitable aspect refactoring, and the analysis and application of additional restructurings when necessary. Our case study of the refactoring of a J2EE system shows that the use of the process reduces the intervention of the developer during the refactoring.

1. Introduction

The separation of concerns is an important issue in software engineering (Parnas, 1972). This becomes relevant when one tries to develop reusable, adaptable, extensible, and modifiable systems. Systems that have these quality attributes are easier to maintain and evolve than those system without them. A high degree of separation of concerns can be achieved with the adoption of object-oriented programing (OOP); however, there are some concerns that orthogonally crosscut the components of a system and whose encapsulation is almost unviable with this paradigm making system evolution difficult and costly (Eaddy et al., 2008). These kinds of concerns are called crosscutting concerns (CCCs) (Kiczales et al., 1997). Aspect-oriented programing (AOP) (Kiczales et al., 1997) complements OOP (as well as other paradigms) by providing mechanisms to construct these kinds of systems by encapsulating the CCCs into aspects. Thus, when the better separation of concerns provided, the more legible, maintainable, and evolvable system achieved (Garcia et al., 2005; Ferrari et al., 2010; Hanenberg et al., 2009). For example, when a CCC needs to be modified, the change is reduced to a single functional unit. For these reasons, in order to improve the evolution of OO systems it is useful to migrate them to aspect orientation (Garcia et al., 2005; Kiczales et al., 1997; Tonella and Ceccato, 2005; Mens and Tourwe, 2008). This can be achieved through the application of aspect refactorings (Kellens et al., 2007) based on information provided by aspect mining (Kellens et al., 2007) of those crosscutting concerns that may potentially become aspects (candidate aspects).

Taking into consideration the above, we have defined, in a previous work, a refactoring process (Vidal et al., 2009; Vidal and Marcos, 2009b) which has as input an object-oriented system and its identified candidate aspects, and it has as output an aspect-oriented system. It is an iterative process that restructures the code by applying aspect-oriented refactorings at each cycle of the iteration. The process has been implemented as a tool called AspectRT[1] (Aspect Refactoring Tool). While this process guides and assists the developer to encapsulate the CCCs of a system, some activities are done manually (i.e. without assistance) by the developer. These manual activities are time consuming for the developer and delay the process of migration. For example, the following activities, are manually accomplished by the developer:

1. The selection of a fragment of aspectizable code to be encapsulated based on information provided by the aspect mining process.
2. The selection of a suitable aspect refactoring to be applied on a fragment of aspectizable code.

* Corresponding author at: ISISTAN Research Institute, Faculty of Sciences, UNICEN University, Campus Universitario, B7001BBO Tandil, Buenos Aires, Argentina. Tel.: +54 2494 439840x42.
E-mail addresses: svidal@exa.unicen.edu.ar (S.A. Vidal), cmarcos@exa.unicen.edu.ar (C.A. Marcos).

[1] Available from http://sites.google.com/site/legacyandaop/Home/ar.

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.12.045

3. The execution of additional restructurings, after applying an aspect refactoring, when necessary. These additional restructurings are small changes in the code performed to preserve the behavior of the system or improve the resultant code.

We think that the use of tool assistance can reduce the time developers spend on this kind of activities. Along this line, in this work we propose assistance for these activities of the process through the use of association rules (Agrawal and Srikant, 1994) and Markov models (Rabiner, 1989).

Since we perform refactorings in a system using AspectRT, the way in which the developer applies restructurings to the system can be registered (e.g. the fragment of code selected, the refactoring applied, additional restructurings made, etc.). The context in which a refactoring is applied is also recorded. By context we mean the kind of fragment of code refactored (e.g. a statement, a method, a field, etc.) and information related to it such as visibility or other modifiers (e.g. concrete, abstract, etc.). Along this line, the Markov model is used to assist the developer while performing activities 1 and 3. The Markov model represents the interaction of the developer with the process by recording the order in which the developer performs refactoring activities. Then, using a Markovian algorithm that analyses the recorded interaction, the most probable action is proposed to the developer. These actions will be suggestions of a particular fragment of code to be refactored (activity 1) or the recommendation of additional restructurings after applying an aspect refactoring (activity 3).

In activity 2, the use of an association rule algorithm helps to select a suitable aspect refactoring to be applied to a fragment of aspectizable code by identifying similarities between context. The algorithm analyses a database of previous refactorings and the context in which they were applied in order to generate association rules. Then, when an aspect refactoring must be applied in a particular context, it is proposed based on the search of the rules that contain a similar context.

Along this line, the main contribution of this paper is providing assistance for a refactoring process that encapsulates CCCs into aspects by means of machine learning techniques. The experimental results of applying our approach to a J2EE system provide evidence of the benefits of assisting the developer during the process using artificial intelligence techniques.

The rest of this paper is structured as follows: Section 2 presents a brief introduction to AOP, Section 3 describes the aspect refactoring process and presents its strengths and weakness, Section 4 explains our proposal to use association rules and Markov models to automate the refactoring process, Section 5 discusses the algorithms configurations, Section 6 evaluates the approach by means of migrating a J2EE system, Section 7 presents the results of the migration, Section 8 assesses how the approach helps developers by conducting a controlled experiment, Section 9 presents the threats to validity of the case studies, Section 10 introduces some work in the area of aspect refactoring related to this research, and Section 11 presents the conclusions.

2. Aspect-oriented programing

Aspect-oriented programing (AOP) (Kiczales et al., 1997) is a programing paradigm that allows the encapsulation of those concerns that orthogonally crosscut the components of a system, called crosscutting concerns (CCCs), into a new component called an aspect. In this way, AOP increases software modularity and reduces the impact of change propagations when the systems are modified (Garcia et al., 2005).

As current programing languages, like Java or C++, do not have the necessary support for this separation of concerns, aspect-oriented languages and frameworks that support aspects have emerged. Some of the most popular of these, such as AspectJ[2] and Spring/AOP[3], support aspect orientation, supplying mechanisms to encapsulate the concerns. Generally, aspect-oriented programs are divided into two parts (Iwamoto and Zhao, 2003):

- **Base code** The classes, interfaces, and so on that define the basic functionality of the system are located in this code.
- **Aspect code** The aspects that encapsulate the crosscutting concerns are located in this code.

In these new languages and frameworks, we need to link the aspects with a statically (or dynamically) identifiable point in the computation of the base code in order to establish a relationship between them through a process called weaving. The identifiable points are well defined points in the system execution where the aspects will be executed. For example, in AspectJ this mechanism is called join point (Mens and Tourwe, 2008). Also, a set of join points can be specified through a notation called pointcut (Mens and Tourwe, 2008). For instance, the pointcut `call(void Point.setX(int))` captures all the calls to the method `setX(int)` of the class Point.

Also, other important mechanism is the advice declaration. An advice defines the code representing the CCC functionality that must be executed when a join point is activated. This mechanism can be executed before, after, or around the pointcut. For example, the advice:

```
after():                          call(void Point.setX(int)){
                                System.out.println(...);
```

is executed after the activation of the pointcut defined above and it prints a message.

3. Aspect refactoring process

In order to obtain a better separation of concerns through the use of AOP, we have proposed a process (Vidal and Marcos, 2009b) which assists the developer during the refactoring process of an object-oriented system into an aspect-oriented one. It is an iterative process which applies aspect refactorings in each cycle of the iteration. This aspect refactoring process has as input an object-oriented system and candidate aspects (previously identified by an aspect mining technique), and it has as output an aspect-oriented system. The candidate aspects are instances of a CCC. For each CCC there may be more than a candidate aspect.

In order to support the refactoring process, a tool called AspectRT was constructed. It is a plug-in for the Eclipse IDE[4] and it is integrated with AspectJ plug-in (AJDT).[5] AspectRT helps architects, designers, and developers to migrate object-oriented systems to aspect-oriented ones, providing a set of aspect refactorings. The tool is based on graphical wizards that assist the developer during the refactoring process. This tool allows the generation of AOP code to be used in AspectJ.

To encompass a wide range of situations that may occur during the refactoring process, different kinds of aspect refactorings are used: Aspect-aware OO refactorings, refactorings for AOP constructs and, refactorings of CCCs (Hannemann, 2006). This feature also ensures that crosscutting concerns are not only encapsulated into aspects, but that the internal structure of the aspects is also improved. The whole process, that we have proposed, relies on a set of steps (Abait et al., 2010; Vidal and Marcos, 2009b) to accomplish these goals. The purpose of these steps are three: encapsulating a CCC into an aspect, enabling the application of a *refactoring of CCC*

[2] http://www.eclipse.org/aspectj/.
[3] http://www.springsource.org/.
[4] http://www.eclipse.org/.
[5] http://www.eclipse.org/aspectj/.

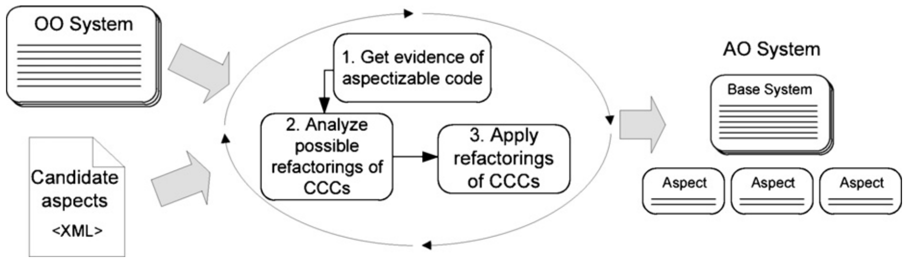

OO System
Candidate aspects <XML>
1. Get evidence of aspectizable code
2. Analyze possible refactorings of CCCs
3. Apply refactorings of CCCs
AO System
Base System
Aspect
Aspect
Aspect

Fig. 1. Refactoring process delineated.

over a CCC when the application of the refactoring is not possible, and improving the internal structure of aspects. In this paper we focus on those steps whose purpose is to analyze and improve the way in which the CCCs are encapsulated into aspects. As shown in Fig. 1, the steps are as follows:

1. **Get evidence of aspectizable code** A piece of aspectizable code is obtained from a list defined in the aspect mining stage. This list is an XML file that contains information related to the crosscutting concerns to be migrated. Each CCC contains one or more candidate aspects that can be spread among several language elements such as classes, interfaces, methods, fields, statements, implements declarations and inner classes (Fig. 2). During this step a language element of a candidate aspect is selected to be encapsulated into an aspect.
2. **Analyze possible refactorings of CCCs** This step selects a special kind of aspect refactoring, called refactoring of CCCs (Hannemann, 2006), whose goal is the encapsulation of a code fragment into an aspect. In order to determine a set of suitable aspect refactorings of CCCs (i.e. which restructuring will be applied to the code fragment), the analysis of a language element of the candidate aspect selected in the previous step must be conducted. The reason for this analysis is the fact that most of the aspect refactorings are applied over a well defined language element. However, since the process uses different kinds of aspect refactorings from different catalogs, usually there is more than one refactoring that can be applied to a language element. In these cases, the analysis must take into account the language element structure and the goals of the refactorings. For example, Monteiro's catalog (Monteiro et al., 2004, 2005) presents 4 refactorings whose target code is a class. If the language element refactored is of this kind, the developer must analyze the structure of the class and the goals of the aspect refactorings to decide which of them is the most suitable to encapsulate the class into an aspect.
3. **Apply refactoring of CCCs** The refactoring/s selected previously is/are applied, so that every crosscutting concern is extracted from the object-oriented code and inserted as an aspect. Once the target code and the refactoring to be applied are selected, the refactoring is automatically executed by AspectRT.

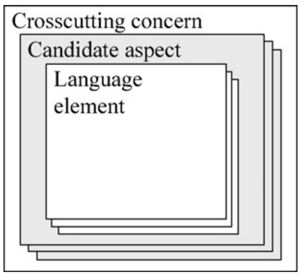


Fig. 2. Candidate aspects XML hierarchy.

We found some limitations in our previous work in reference to these steps. Specifically, we noticed a high intervention of the developer in several activities. For example, the selection of a piece of evidence of aspectizable code (Step 1) is accomplished manually by the developer using the AspectRT interface. Also, in Step 2, the analysis and selection of a suitable aspect refactoring of CCCs is accomplished by the developer by choosing from a menu of possible refactorings. In this regard, we previously presented an approach based on rules (Vidal and Marcos, 2009a) that restricts the possible refactorings to be applied. These were simple rules that for a given kind of language element returned a list of pre-computed possible refactorings. However, this technique was not precise enough to identify all the possible refactorings. Another limitation found was that, even though the aspect refactorings are applied automatically by the process (Step 3), on some occasions the developer's intervention is necessary for some decisions. Such interventions cannot be avoided because the developer needs to know precisely what changes are going to be made to the system. However, in these kinds of situations, in which the rate of the developer's intervention is high, the use of tool assistance can be useful.

Another situation in which the tool assistance can be used is when on some particular occasions, the application of an aspect refactoring is not sufficient to encapsulate a language element of a candidate aspect, with the result that additional restructurings must be performed manually. These situations occur because sometimes there is no refactoring to properly encapsulate a specific fragment of code into an aspect (or a specific refactoring is not implemented in the tool). In these cases, low complexity OO refactoring and aspect refactorings (e.g. *Extract Method* (Fowler, 1999), *Extract Fragment into Advice* (Monteiro et al., 2004),*Move Method from Class to Inter-type* (Monteiro et al., 2004)) are tried to encapsulate the fragment of code and then completing the extraction by additional manual restructurings. These additional restructurings can be proposed automatically by means of the analysis of previous interactions.

The common problem of the aforementioned limitations is that the developer must intervene too much during the refactoring process. These interventions cause a delay in the refactoring that consumes valuable time. In this paper we focus on the analysis of various techniques to solve these limitations by providing provide an assisted aspect refactoring process.

4. Assistance in the application of aspect refactorings

In order to improve the assistance during the identification and application of aspect refactorings in the process, we propose the use of artificial intelligence techniques. We use this kind of techniques because they can be useful for capturing the user's[6] knowledge when he/she is dealing with complex tasks, and later for

[6] The words developer and user are used indistinctly in this article.

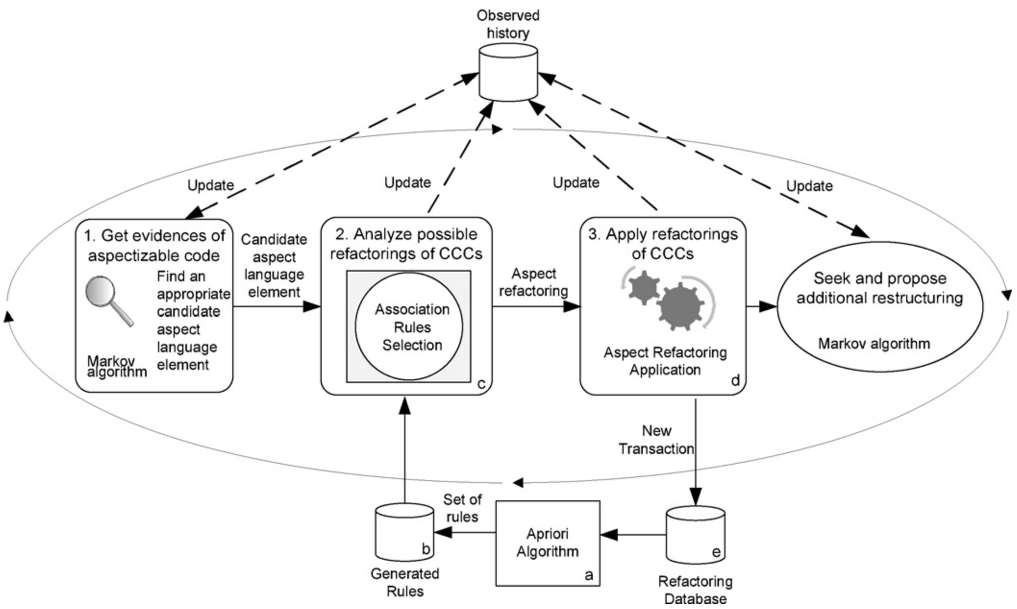

Observed history
Update
Update
Update
Update
1. Get evidences of aspectizable code
Find an appropriate candidate aspect language element
Markov algorithm
Candidate aspect language element
2. Analyze possible refactorings of CCCs
Association Rules Selection
c
Aspect refactoring
3. Apply refactorings of CCCs
Aspect Refactoring Application
d
Seek and propose additional restructuring
Markov algorithm
New Transaction
Set of rules
Apriori Algorithm
a
b
Generated Rules
e
Refactoring Database

Fig. 3. Automatic refactoring process.

recognizing the user intentions in similar cases. In this way, previous situations and experiences can be taken into account at the time of restructuring a new system. For example, in Step 2 of the process the aspect refactoring/s that should be applied to encapsulate a language element into an aspect must be determined. This is a complex task owing to the need for extensive knowledge of the goal of each aspect refactoring, limitations of use, mechanisms of application, and so on. Also, the manual selection and application of an aspect refactoring is time-consuming. For these reasons, we propose the tool assistance application using association rules (Agrawal and Srikant, 1994), during the identification of the aspect refactorings. Association rules contribute to the automatic recommendation of aspect refactorings by means of the analysis of previous experiences.

In Step 3, another complex situation emerges when the application of an aspect refactoring is not adequate to encapsulate a language element of a candidate aspect, causing additional manual restructurings. These changes are carried out to complete the encapsulation of a candidate aspect or to fix problems in the source code so as to allow correct compilation. When the application of an aspect refactoring is not sufficient to encapsulate a language element of a candidate aspect this situation can be solved in two ways. On the one hand, some specific aspect refactorings, as those presented by Laddad (2003a), can be used to completely encapsulate the candidate aspect. However, the application of a specific refactoring causes a scalability problem. That is to say, every time a new specific situation emerges, a specific refactoring must be implemented and integrated into the tool. On the other hand, simple aspect refactorings (as those presented by Monteiro et al. (2004, 2005)) can be used to encapsulate a candidate aspect and then apply manual changes in order to complete the encapsulation. Usually, these manual changes are simple and repeatable. For this reason, we propose the automatic identification and application of these changes using Markov models (Rabiner, 1989).

We also propose the use of Markov models to determine the order in which the aspectizable code should be refactored (Step 1). By using Markov we want to assist the developer during the process by selecting the aspectizable code to be refactored.

The use of these techniques in the identification and application of aspect refactorings in the refactoring process is shown in Fig. 3 and it is explained in detail in the following sections.

4.1. Aspect refactoring proposition through association rules

Association rules is an artificial intelligence technique widely used in data mining. It describes the relationships between the items in a set of relevant data. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items and $D = \{T_1, T_2, \ldots, T_N\}$ a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. An association rule is an implication of the form $A \Rightarrow B$, where $A \subset I$, $B \subset I$ and $A \cap B = \emptyset$ (Agrawal and Srikant, 1994). A is the antecedent of the rule and B is its consequent. The rule $A \Rightarrow B$ holds in the transaction set D with support s, where s is the percentage of transactions in D that contains $A \cup B$. That is, the support of this rule is the percentage of transactions containing all items from A and from B. The rule $A \Rightarrow B$ has confidence c in the transaction set D if c is the percentage of transactions in D containing B, from among the transactions that contain A. That is, from all the transactions containing A, c indicates the percentage of transactions that also contain B. In a more formal definition:

$$support(A \Rightarrow B) = \frac{\sigma(A \cup B)}{N}$$

$$confidence(A \Rightarrow B) = \frac{\sigma(A \cup B)}{\sigma(A)}$$

where

$\sigma(X) = |\{T_i | X \subseteq T_i, T_i \in D\}|$ and X is an itemset.

Rules that satisfy both a minimum support (*minsup*) threshold and a minimum confidence (*minconf*) threshold are called strong (Han and Kamber, 2000), and they are the output of an association rules algorithm.

We use association rules in order to assist the user in the identification of the aspect refactoring (or a set of them) which must be applied given a specific fragment of aspectizable code. To generate the rules that will be useful to propose aspect refactorings, the Apriori algorithm (Agrawal and Srikant, 1994) is adopted. As shown in Fig. 3, the assistance process consists of the followings stages:

(a) Generating the association rules based on the analysis of a database of refactoring information.
(b) Saving the rules into a database of generated rules.

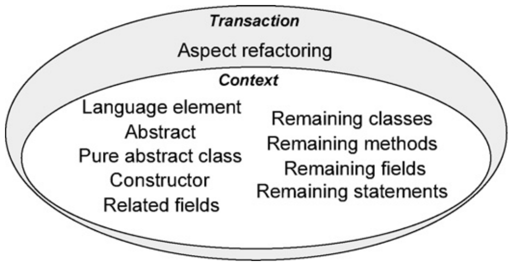


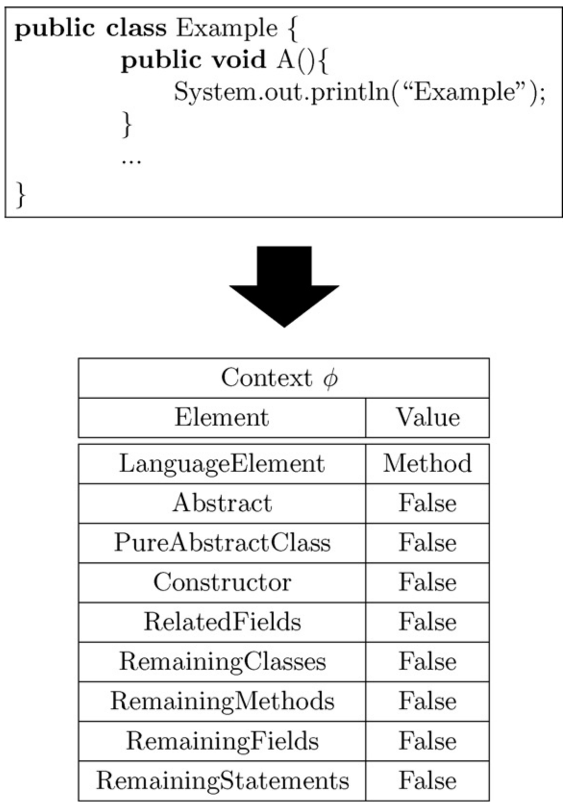
Fig. 4. Database transaction.

(c) Selecting automatically an appropriate aspect refactoring for a language element of a candidate aspect using the association rules.
(d) Applying the aspect refactoring.
(e) Saving to the refactoring database the information from the aspect refactoring that was applied.

Stage (a) is executed using the Apriori algorithm. This stage has as input a database of previous aspect refactoring experiences, and it has as output a set of association rules. The database contains information about the context in which a refactoring was applied in previous system's migrations. Specifically, the context that indicates each transaction is composed by 9 fields indicating the language element over which the refactoring was applied, data related with this language element (such as if it is abstract or concrete, or if it is a constructor method), and information connected with the remaining language elements to be migrated (e.g. the number of methods related with the candidate aspect that remains to be migrated). That is to say, the remaining language elements are those elements of the candidate aspect that have not yet been migrated (Fig. 4). In this way, the context not only contains information about the code that is refactored but also contains information about the order in which the user accomplishes this task. For example, consider a candidate aspect that is composed by a field and two methods. If the field is encapsulated into an aspect first, the remaining language elements will be the two methods.

In this first stage, the Apriori algorithm analyses the transactions of the database to find all the itemsets that satisfy a minimum support threshold. Later, it uses these itemsets to generate the rules that have a confidence greater than the minimum confidence. In our case the algorithm's itemsets are composed of elements from a transaction of the database (an aspect refactoring, a language element and information related to it, such as, if it is abstract or a constructor, etc.). An important issue in the implementation of the algorithm is how to filter out the unimportant and non-applicable rules. For this reason, we use item constraints (Srikant et al., 1997) to obtain only a subset of rules that are significant. Specifically, we look for rules having as an antecedent at least one language element and as a consequent an aspect refactoring. The reason of this constraint is that most of the aspect refactorings are applied over a well-defined language element. Once association rules have been generated, they are stored in a database as shown in stage (b).

Stage (c) is run in Step 2 of the refactoring process. When a language element of a candidate aspect is analyzed, a selection of an appropriate aspect refactoring is accomplished. This selection is based on the association rules generated in stage (a). In order to select a suitable aspect refactoring, a naive hierarchical selection process is used. When a language element of a candidate aspect is analyzed, the same fields of context information that compose the refactoring database are obtained. Later, this information can be compared with the resultant association rules. In this way, the goal of the hierarchical algorithm is to find a rule whose context (antecedent) is equal or as similar as possible to the context of



```
public class Example {
        public void A(){
            System.out.println("Example");
        }
        ...
}
```

Context ϕ	
Element	Value
LanguageElement	Method
Abstract	False
PureAbstractClass	False
Constructor	False
RelatedFields	False
RemainingClasses	False
RemainingMethods	False
RemainingFields	False
RemainingStatements	False

Fig. 5. Context obtaining from a language element of a candidate aspect.

the language element that is going to be refactored. When a rule is found, the consequent is suggested as aspect refactoring to be applied over the language element. This hierarchical selection process starts by obtaining the context data of the aspectizable code under analysis. Subsequently, the algorithm tries to find a suitable refactoring. With this goal in mind, the rules of a specific size for an antecedent are obtained. The beginning size is the full context size (that is, the most specific case), that is the rules whose antecedent is of the same size as the full context (in our case, size 9) are obtained. The next step implies iterating over these rules in order to determine if the antecedent of a rule *r* is contained in context data of the aspectizable code under analysis. When this happens, the consequent of the rule is proposed as aspect refactoring. If no rule is found, the algorithm tries to find a rule whose antecedent matches with a subset of the context. For example, as shown in Fig. 5, given method *A* and assuming there are no more classes, methods, fields or statements to be encapsulated, when method *A* is analyzed the context ϕ of size 9 is obtained. Then using the hierarchical naive algorithm a rule whose antecedent matched with the context is sought. If a rule is found, its consequent is proposed. If it is not, the algorithm tries again with the context subsets of size 8 and so on until the algorithm finds a rule or if a rule is not found the developer has to choose the refactoring to apply. In the event that a rule is found and an aspect refactoring is proposed, if the developer does not accept it, the selection process of stage (c) continues with the set of rules of lesser size. Considering the rules presented in Table 1 as the refactoring database and the context shown in Fig. 5 as a language element to be refactored, the aspect refactoring *Move Method from Class to Inter-type* is proposed when a context of size 4 is analyzed.

Stage (d) occurs in Step 3 of the refactoring process. If an aspect refactoring is found and accepted by the developer in stage (c), the process automatically applies the aspect refactoring in the third stage and the context data with the selected refactoring is saved in

Table 1
Example of rules.

Context size	Rules
1	languageElement.Field ⇒ refactoring.MoveFieldFromClassToIntertype languageElement.Method ⇒ refactoring.MoveMethodFromClassToIntertype
2	languageElement.Field, abstract.false ⇒ refactoring.MoveFieldFromClassToIntertype languageElement.Method, constructor.false ⇒ refactoring.MoveMethodFromClassToIntertype
3	languageElement.Field, abstract.false, constructor.false ⇒ refactoring.MoveFieldFromClassToIntertype languageElement.Method, abstract.false, constructor.false ⇒ refactoring.MoveMethodFromClassToIntertype
4	languageElement.Field, abstract.false, constructor.false, remainingFields.false ⇒ refactoring.MoveFieldFromClassToIntertype **languageElement.Method, abstract.false, constructor.false, remainingMethods.false ⇒ refactoring.MoveMethodFromClassToIntertype**

the refactoring database in stage (e). For example, in the case of the aforementioned method *A*, if the proposed refactoring is accepted by the developer, a transition that contains the context shown in Fig. 5 and the name of the refactoring is saved into the database.

4.2. *Language element identification and additional restructuring recommendation through Markov models*

A hidden Markov model (HMM) is a doubly stochastic process comprising an underlying stochastic process that is not directly observable but can only be visualized through another set of stochastic processes that produce the sequence of observations (Rabiner, 1989). A Markov model describes a process that goes through a sequence of discrete states. The model is called hidden because the state of the model at a time *t* is not observable directly. A HMM has the Markov assumption that is that given the present state, future states are independent of past states.

The main elements of an HMM are the following (Rabiner, 1989):

- N states that are denoted individually as $\{S_1, S_2, \ldots, S_N\}$.
- The actual state at time t is denoted as q_t.
- The state transition probability distribution is the matrix $A = \{a_{ij}\}$ where a_{ij} is the probability of the model transitioning from state i to state j.
- M distinct observable actions per state that are denoted individually as $V = \{v_1, v_2, \ldots, v_M\}$.
- The action probability distribution is the matrix $B = \{b_j(k)\}$ where $b_j(k)$ is the probability of observing action v_k at a time when the model is in state q_j.
- The action observed at time t is denoted as O_t.

We use the ON-line Implicit State Identification (ONISI) algorithm (Gorniak and Poole, 2000) in order to identify the future user action through a Markov model. In our context, this algorithm observes the developer interaction with AspectRT and, given a state, it predicts future developer actions. ONISI assigns probabilities to all possible actions in the currently observed state. These probabilities are calculated estimating how much observed history supports an action in the current context. This estimation is accomplished using a k-nearest neighbors scheme that ranks the actions in the state taking into account the length of the sequences of actions in observed history that match the actions the developer just performed. In this way, ONISI approximates the check of the Markov assumption for the current context (Gorniak and Poole, 2000). When the current state is observed by ONISI, probabilities to all possible actions from that state are assigned and ranked. The algorithm is configurable by means of two parameters: k to indicate number of maximum length pairs of ⟨state, action⟩ to be consider; and $0 \leq \alpha \leq 1$ to indicate the weight between the histories length and the frequency at which they occur.

In our context, we use the ONISI algorithm to help in the identification of which candidate aspect must be refactored (Step 1 of the refactoring process). We also use the ONISI algorithm in order to identify the additional restructurings that must be performed (at the end of Step 3 of the refactoring process) in case the refactoring is not sufficient. In order to fulfill these tasks, we create a model that represents the interaction between the developer with AspectRT. It has 3 simple states (Fig. 6):

1. Candidate aspect view This state represents the situation when a candidate aspect has been selected. The state is so named because AspectRT provides a view where the candidate aspects are listed.
2. Refactoring a language element of a candidate aspect This state occurs when an aspect refactoring has been selected and it is going to be applied or it has been applied.
3. Additional refactorings This state represents the situation when additional restructurings have to be done after applying an aspect refactoring.

As shown in Fig. 6, to transition to State 1 a language element of a candidate aspect must be selected. When the model is in State 1, it can only transition to State 2 with the selection of an aspect refactoring. In State 2, the model remains in the same state when an aspect refactoring is applied. If after applying an aspect refactoring, manual changes are made over a code structure, instead of selecting a new candidate aspect, the model transitions to State 3. Some structures that can be added, modified or deleted are language elements from the classes, or aspects.

The states and the actions that allow the transition between them are updated in each step of the refactoring process and saved into an observed history database (As was shown at the top of Fig. 3). Specifically, the database saves pairs of states and actions $\langle S_i, v_j \rangle$ where v_j is the action occurred in the model and S_i is the state to which the model is going to transition. For example, if the model is in State 3 and a method of a candidate aspect is selected in the Candidate aspect view of AspectRT, the pair ⟨state, action⟩ saved will be:

⟨Candidate aspect view, candidate aspect Method selected⟩

When a candidate aspect must be refactored, there usually are a set of possible language elements, of which it is composed, to be chosen. The order in which these are selected can produce problems of compilation or encapsulation in the resultant system. This order depends on the structure of the candidate aspects. This is why the order must be identified during the refactoring process. For example, consider the following code.

```
public class Foo {
  private int x;
  ...
  private doSomething(){
    ...
    x=3;
  }
  ...
}
```

If the field *x* and the statement in the method *doSomething* belong to a candidate aspect, the order of refactoring should be:

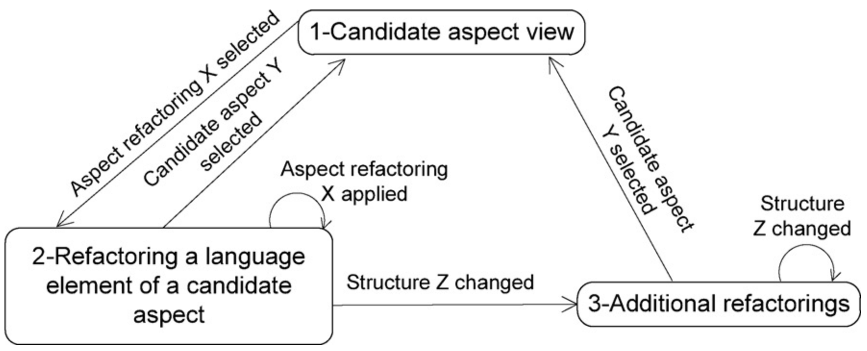

1-Candidate aspect view
2-Refactoring a language element of a candidate aspect
3-Additional refactorings
Aspect refactoring X selected
Candidate aspect Y selected
Aspect refactoring X applied
Candidate aspect Y selected
Structure Z changed
Structure Z changed

Fig. 6. AspectRT model.

1. Encapsulate *x* into an aspect using *Move Field from Class to Intertype* declaring the aspect as privileged temporarily.
2. Encapsulate the statement into the aspect using *Extract Fragment into Advice* declaring the aspect as public.

Encapsulating the statement first might lead to compilation errors.

In order to identify the refactoring order, the ONISI algorithm is run in Step 1 of the refactoring process and the first ranked result is proposed to the developer. There are two situations to analyze in this task: (1) the selection of the first language element of a candidate aspect to be encapsulated and (2) the selection of a language element of a candidate aspect after the encapsulation of other language element of the same candidate aspect. In this case, task 1 will be the selection of the field *x* and then, task 2 will be the selection of the statement. In order to be able to distinguish between these situations, the actions of selecting a candidate aspect must be different for each situation. This is achieved by adding a distinctive token to the action (of a ⟨state, action⟩ pair) when it is the first refactoring that is applied on a candidate aspect.

Another case that needs to be automated occurs when after applying a refactoring, some manual changes are needed. This situation is due to the presence of some complex concerns in the system to which a general refactoring must be applied followed by some manual changes. These additional manual changes must be performed in order to preserve the behavior of the system or improve the resultant code. We use a Markov algorithm to identify these situations and automatically propose changes when they are necessary, this is why the ONISI algorithm is run after Step 3 of the refactoring process. As it was shown in the model (Fig. 6), all the changes that are made by the developer when he/she applies a refactoring are made over a code structure. These changes can be the addition, deletion or modification of such structures. Examples of possible structures are access modifiers, pointcut declarations, try/catch blocks, etc.

An important point to be discussed is why these automations are not fulfilled with association rules. While in practice this would be possible, the reason why we can not use association rules to identify the fragments of code to be migrated and to recommend additional restructurings is that the association rules technique is not flexible enough to identify new problems without regenerating the association rules through the association rules algorithm. Since most of the aspect refactorings are applied over a well known language element (e.g. a method, a field, etc.), association rules excel in the suggestion of aspect refactorings if all of the supported aspect refactorings have been considered in the training. Regarding the identification of fragments of code to be migrated and the recommendation of additional restructurings, while the Markov approach could not correctly identify, for example, an additional restructuring from the first time that it is analyzed, this approach learns quickly without the need of re-running a training phase.

5. Algorithms configuration

An important issue to take into account is how to properly configure the algorithm parameters. In the case of the Apriori algorithm, the value of minimum support determines which association rules are generated and which are not. A high value will probably make us miss some important association rules. Thus, we have to determine a value for minimum support that enables us to discover an aspect refactoring to be applied to a specific situation. In our case, the refactoring database contains the context in which a refactoring was applied. Supposing that there are N different instances of a refactoring situation stored in the database, the main problem is that they are not equally probable. So, a small minimum support value will be needed to capture a significant number of aspect refactorings. On the other hand, the minimum confidence value should be high since it indicates the probability that a refactoring was used under a context situation. Otherwise, higher minimum confidence values can cause the loss of important rules. These hypothesis were confirmed after running an experiment in which the database of aspect refactorings was generated. The generation was based on small examples, taken from our own experience, in which all the aspect refactorings supported by AspectRT were applied. These examples belong to candidate aspects of the concerns Command, Undo, Observer, Persistence, and Composite (Marin et al., 2007) of the application JHotDraw.[7] Specifically, nearly 50 aspect refactorings were applied. While the number of times in which each refactoring was applied was not the same, the examples were selected in order to ensure that each aspect refactoring supported by AspectRT was applied on at least one occasion. Fig. 7 presents the variation of rules that were generated with different values of support and confidence. As can be seen, rules are only generated for low values of support. Also, we found, by comparing the results of running Apriori with different configuration values, that the number of rules generated for high numbers of confidence is not significantly lower than the number of rules generated for low values. After the analysis of the variations shown in Fig. 7, and taking into account our previous considerations, we found that an appropriate value for minsup is 0.1 and for minconf is 0.9. As was said, the main consideration to select these values is the fact that we need to generate rules that take into account all the aspect refactorings implemented in the tool (although some are applied more frequently than others) at the same time that we ensure that the context situation contemplated in the rule is one of the usual context in which the refactoring is applied.

With regard to the values of the k and α parameters that are used in the ONISI algorithm, we determine them by adopting the following considerations. We use a $0.7 \leq \alpha \leq 0.9$ in order to give more

[7] http://www.jhotdraw.org, version 5.4b1.

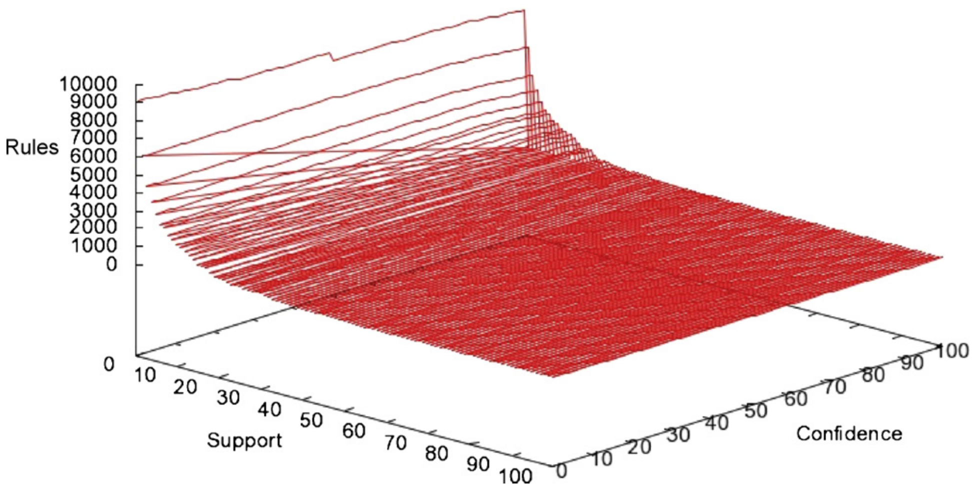


Fig. 7. Rules generated with support and confidence variation.

importance to the match length measure normalized rather than the frequency measure normalized. The next scenario will help to explain this decision. When the model is in the state "Candidate aspect view" the selection of an aspect refactoring is expected. So, in a situation like this, it is not important how many times a determined refactoring in this state was selected (which is measured by the frequency measure); however, it is important the frequency that an aspect refactoring was selected after the selection of a language element (which is measured by the match length measure). With respect to the k parameter, Gorniak and Poole (2000) claim that low values show the same performance as high values. Therefore we propose a $3 \leq k \leq 5$ value in order to make faster calculations.

6. Case study

In this section we conduct a case study in which the system Java Pet Store Demo[8] is refactored. This system is a J2EE demo application built with the goal of demonstrating the J2EE platform capabilities (especially, EJB technology). Also, it illustrates the use of good design practices such as the use of design patterns (Gamma et al., 1995).

In this case study, 4 crosscutting concerns found in Java Pet Store Demo are refactored: *Exception Wrapping and Business Delegate*, *Service Locator*, *Serializable Interfaces*, and *Precondition Checking*. These are the same crosscutting concerns that have been identified by other works in the field of aspect mining (Marin et al., 2007; Mesbah and van Deursen, 2005). For this reason, we use these candidate aspects' information as an entry to the process.

In order to generate the association rules that capture a significant number of aspect refactoring, the Apriori algorithm was run with $minsup = 0.1$ and $minconf = 0.9$. A small minimum support value is needed to contemplate all the possible aspect refactorings while the minimum confidence value should be high since it indicates the probability that a refactoring was used under a context situation. In addition, the ONISI algorithm was configured with $k = 4$ to consider four pattern matches (i.e. the algorithm will look for the 4 longest sequences in the recorded history that match the immediate history) and $\alpha = 0.8$ to give more importance to the match length measure.

[8] http://java.sun.com/blueprints/code/jps132/docs/index.html, version 1.3.2.

For the purpose of training the tool, smalls examples, taken from our own experiences and other's experiences (Laddad, 2003b; Gradecki and Lesiecki, 2003; Colyer et al., 2004),were refactored before starting the refactoring of the system Pet Store. This training allows the tool to generate a database of aspect refactorings to be used in the Apriori algorithm. Specifically, during the training 16 aspect refactorings were applied in different situations (i.e. each refactoring was applied in more than one occasion). These applications generated more than 700 database transactions and more than 1500 rules when the Apriori algorithm was run. In contrast with the association rules approach, no training of the ONISI algorithm was conducted. Moreover, the observed history database was reset before starting the case study. This was accomplished in order to show how the ONISI algorithm learns the developer behavior during the refactoring.

To start the refactoring process, the XML that contains the candidate aspects is loaded in the tool showing all the aspectizable language elements in the candidate aspects view (Fig. 8). Then, the tool selects the first language element from the list and the process of refactoring begins.

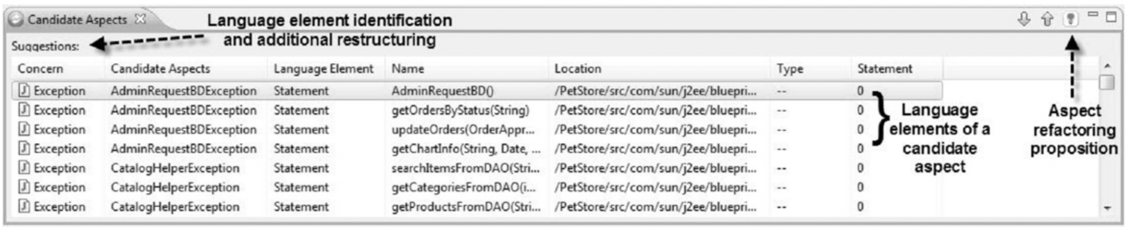
In the following sections the refactoring process of the 4 CCCs is described.

6.1. Exception Wrapping and Business Delegate

This crosscutting concern affects the exception handling of 41 classes in Pet Store. Specifically, when an exception is thrown it is caught and a new exception of a different type is rethrown. For this reason, the language elements of this CCC are try/catch blocks. There are almost 80 language elements pertaining to 31 candidate aspects.

The first language element of this concern links to the constructor method of the *AdminRequestBD* class. The language element is a try statement whose catch clauses throw an exception of type *AdminRequestBD* (Fig. 9). This exception handling should be encapsulated into an aspect, improving the separation from the primary base code (i.e. the code inside the try clause).

When the language element is analyzed, the aspect refactoring proposition button (Fig. 8) should be clicked. In this case, the aspect refactoring *Extract Fragment into Advice* (Monteiro et al., 2004) is proposed and applied by the tool creating a new aspect called *AdminRequestBDException* with a suitable pointcut and advice. *Extract Fragment into Advice* is proposed because it has as a goal


Candidate Aspects
Language element identification and additional restructuring
Suggestions:
Language elements of a candidate aspect
Aspect refactoring proposition

Concern	Candidate Aspects	Language Element	Name	Location	Type	Statement
Exception	AdminRequestBDException	Statement	AdminRequestBD()	/PetStore/src/com/sun/j2ee/bluepri...	--	0
Exception	AdminRequestBDException	Statement	getOrdersByStatus(String)	/PetStore/src/com/sun/j2ee/bluepri...	--	0
Exception	AdminRequestBDException	Statement	updateOrders(OrderAppr...	/PetStore/src/com/sun/j2ee/bluepri...	--	0
Exception	AdminRequestBDException	Statement	getChartInfo(String, Date, ...	/PetStore/src/com/sun/j2ee/bluepri...	--	0
Exception	CatalogHelperException	Statement	searchItemsFromDAO(Stri...	/PetStore/src/com/sun/j2ee/bluepri...	--	0
Exception	CatalogHelperException	Statement	getCategoriesFromDAO(i...	/PetStore/src/com/sun/j2ee/bluepri...	--	0
Exception	CatalogHelperException	Statement	getProductsFromDAO(Stri...	/PetStore/src/com/sun/j2ee/bluepri...	--	0

Fig. 8. Candidate aspect view.

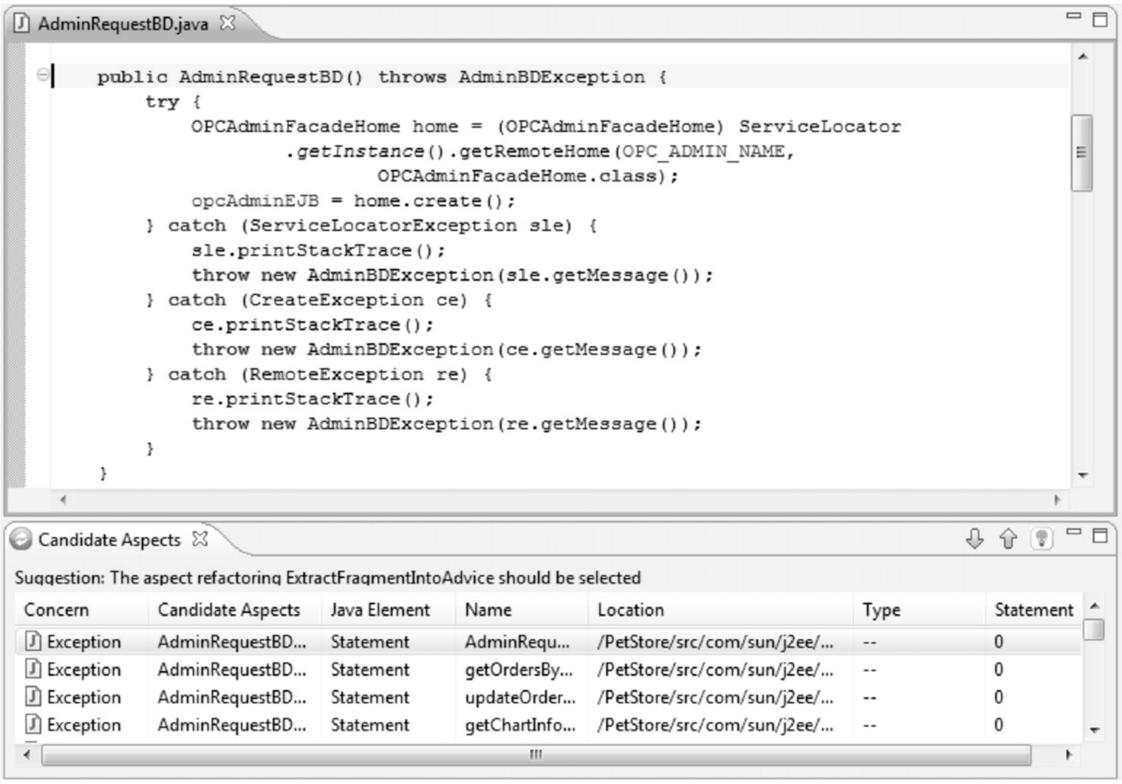
the encapsulation of a fragment of aspectizable code into an aspect creating a new advice and a pointcut.

However, this restructuration is not sufficient to properly encapsulate the CCC. Therefore, additional restructurings must be applied. These are proposed by the process by means of the suggestion label shown at the top of Fig. 8. As expected, the first time that the additional restructurings should be proposed, they are not identified because the algorithm has not learned similar cases yet. This is because the training, previous to the refactoring of the system, did not contain cases similar to this. The additional restructurings to encapsulate this language element are:

1. add a *SoftException* to the advice in order to wrap a caught exception.
2. create a suitable *declare Soft* for each catch clause.
3. delete the try/catch block from the class.

The second of these additional restructurings varies depending on the number of catch clauses related to the try statement. In the first case 3 *declare Soft* statements are created. Also, in the first case, the refactoring is accomplished manually by the developer and all the information related to the changes is saved by the tool in order to be used in similar cases in the future. The aspect that results from applying these restructuration to the first language element is shown in Fig. 10. As in the case of the refactored class, the try catch block is removed.

Having finished the encapsulation, the process proposes this candidate aspect's next statement to be refactored. Once all the candidate aspects are refactored, a language element of the next candidate aspect is proposed. The remainder of the language elements are encapsulated in a similar fashion (i.e. applying the *Extract Fragment into Advice* refactoring and similar additional restructurings). With regard to the proposition of additional restructurings to a refactoring, these are proposed by the process from the refactoring of the second language element of the first refactored candidate aspect. When the process recognizes these restructurings in subsequent refactorings, they are proposed in the suggestion label as "Add a SoftException to the aspect", "Add a declare soft to the aspect" and, "Delete the try/catch block in the class". The percentage of activities successfully proposed by the process increases as the algorithm learns the restructurings implemented by the developer. For example, in the first cases some activities were not properly proposed because they did not appear in the first place on the ONISI ranking. That is because the



```
public AdminRequestBD() throws AdminBDException {
    try {
        OPCAdminFacadeHome home = (OPCAdminFacadeHome) ServiceLocator
                .getInstance().getRemoteHome(OPC_ADMIN_NAME,
                        OPCAdminFacadeHome.class);
        opcAdminEJB = home.create();
    } catch (ServiceLocatorException sle) {
        sle.printStackTrace();
        throw new AdminBDException(sle.getMessage());
    } catch (CreateException ce) {
        ce.printStackTrace();
        throw new AdminBDException(ce.getMessage());
    } catch (RemoteException re) {
        re.printStackTrace();
        throw new AdminBDException(re.getMessage());
    }
}
```

Suggestion: The aspect refactoring ExtractFragmentIntoAdvice should be selected

Concern	Candidate Aspects	Java Element	Name	Location	Type	Statement
Exception	AdminRequestBD...	Statement	AdminRequ...	/PetStore/src/com/sun/j2ee/...	--	0
Exception	AdminRequestBD...	Statement	getOrdersBy...	/PetStore/src/com/sun/j2ee/...	--	0
Exception	AdminRequestBD...	Statement	updateOrder...	/PetStore/src/com/sun/j2ee/...	--	0
Exception	AdminRequestBD...	Statement	getChartInfo...	/PetStore/src/com/sun/j2ee/...	--	0

Fig. 9. Exception Wrapping and Business Delegate concern.

```
public aspect AdminRequestBDException {
  declare soft: ServiceLocatorException :
   call(public void AdminRequestBD.*(..)
   throws ServiceLocatorException);

  declare soft: CreateException : call(public void AdminRequestBD.*(..)
    throws CreateException);

  declare soft: RemoteException : call(public void AdminRequestBD.*(..)
    throws RemoteException);

  pointcut adminRequest(AdminRequestBD _this):
   execution(public AdminRequestBD.new()
   throws AdminBDException ) && this(_this);

  after(AdminRequestBD _this) throwing(SoftException sle)
    throws AdminBDException: adminRequest(_this){
      sle.printStackTrace();
     throw new AdminBDException(sle.getMessage());
  }
}
```

Fig. 10. Exception Wrapping and Business Delegate aspect.

observed history did not have enough support of similar cases and because of the small variations between the language elements.

6.2. *Service Locator*

This crosscutting concern is a J2EE pattern that is used to centralize the getting of services (Gamma et al., 1995). This is a special case of a CCC in Java Pet Store Demo because it is unique. That is, there is only one occurrence of the CCC that was indicated as aspectizable by aspect mining.

The candidate aspect is spread on a field and a method of the *ServiceLocator* class. As is mentioned by Marin et al. (2007), the main problem of this concern is the number of external methods that call to the service locator (namely, 30 calls). Moreover, the inversion of control pattern is usually used in J2EE applications to avoid the direct calls (Fowler, 2004). For this reason, its encapsulation into an aspect is a possible solution. To begin the refactoring, the field and the method language elements were encapsulated using the aspect refactorings *Move Field from Class to Inter-type* and *Move Method from Class to Inter-type* (both were properly proposed by the process). Then, some additional restructurings were necessary. These restructurings are performed manually because the tool did not have a registered history related to this kind of additional restructuration.

6.3. *Serializable Interfaces*

This crosscutting concern represents the serializable role that a class plays when it implements the *Serializable* interface. The CCC is spread across 31 classes. It was represented as a single candidate aspect that contains all the references to the implements declarations. In order to obtain clear classes, the implements declarations should be encapsulated into an aspect as a declare parents statement.

The refactoring begins when the process proposes the first implement declaration of the list to be encapsulated. Afterwards, the application of the aspect refactoring *Encapsulate Implements with Declare Parents* (Monteiro et al., 2004, 2005) is proposed. Once the language element is encapsulated, there is no need for additional restructurings. So, the process recommends the encapsulation of the next implements declaration of the candidate aspect. The process of refactoring this CCC continues in a similar fashion until all the language elements of the candidate aspect are encapsulated. Then, the encapsulation of a language element of the next concern is proposed.

6.4. *Precondition Checking*

The Precondition Checking concern is spread in 9 methods called *fromDOM(Node)*. These methods are implemented in different classes. As is shown in Fig. 11, in order to return an instance of the class that contains the method, the structure of the *Node* is checked. If the structure is not as expected, an exception is thrown. Therefore, this mechanism of checking a precondition and throwing an exception should be encapsulated into an aspect.

Each candidate aspect is composed of the statement that contains the throw clause. In order to start the refactoring of this CCC the process suggests the selection of one of the throw statements to be encapsulated. In the case presented in Fig. 11, after the selection of the throw statement, the aspect refactoring *Extract Fragment into Advice* is proposed and applied by encapsulating the throw clause into a new aspect called *AddressPrecondition*. Then, some additional restructurings are necessary to properly verify the checking condition. Six additional restructurings to achieve this refactoring are applied:

1. restructure the *Element* variable declaration in *fromDOM(Node)* method;
2. apply the OO refactoring *Extract Method* to the if clause creating a new method called *isPreChecked(Node)*;
3. move this new method to the aspect;
4. change the advice in order to throw the exception only when the check is false;
5. delete all the calls to *isPreChecked* in the *fromDOM* method;
6. delete the *isPreChecked* method.

The aspect that results after applying this restructuring is shown in Fig. 12 (developer interaction was needed to simplify the final structure of the aspect). After encapsulating the concern, the statement of the next candidate aspect is proposed by the process to be encapsulated.

```
public class Address {
  public static final String XML_ADDRESS = "Address";
  ...
  public static Address fromDOM(Node node)
   throws XMLDocumentException {
    Element element;
    if (node.getNodeType() == Node.ELEMENT_NODE &&
    (element = ((Element) node)).getTagName()
      .equals(XML_ADDRESS)) {
        Address address = new Address();
        ...
        return address;
    }
    throw new XMLDocumentException(XML_ADDRESS
     + "element expected.");
  ...
}
```

Fig. 11. Precondition Checking concern.

Table 2
Efficacy in process automation.

Assistance efficacy	% of Language elements
Automatically refactored	70
Fail in one or more recommendations	30 (Unsuccessful Java element identification 1%, Unsuccessful additional restructuring recommendation 24%, Unsuccessful identification and recommendation 5%)

Table 3
Distribution of aspect refactorings applied.

Aspect refactoring	% of application
Extract fragment into advice	76
Encapsulated implements with declared parents	22
Move field from class to inter-type	1
Move method from class to inter-type	1

7. Discussion

In this case study, 140 language elements pertaining to 51 candidate aspects of 4 CCC were refactored and encapsulated in aspects. As shown in Table 2, of these language elements, 70% were automatically refactored, by this we mean that the language element to be encapsulated, the aspect refactoring, and additional restructurings were identified and proposed correctly for each language element. The majority of the remaining 30% failed in the recommendation of one or more additional restructuring for a language element (24%). Others failed during the identification of the aspectizable code to be encapsulated (1%), or during both identification and recommendation (5%). As it is shown, failures were not observed during the recommendation of an aspect refactoring for a language element. These results are discussed in the following sections.

7.1. Evaluation of the use of association rules

During the Java Pet Store Demo refactoring 140 aspect refactorings were proposed and applied (one for each language element). All the aspect refactorings were properly identified by the process allowing the automatic application of them. However, in 77% of the cases, after applying an aspect refactoring, the application of some additional restructurings was necessary.

The good performance of the algorithm based on rules is due to the fact that the algorithm relied on a comprehensive database of aspect refactoring and also because, throughout the refactoring of the language elements, no atypical cases were found. With the objective of refactoring the CCCs 4 aspect refactorings were used (Table 3), which resulted in no major difficulties during the

```
public aspect AddressPrecondition {
  public boolean isPreChecked(Node node) {
    return (node.getNodeType() == Node.ELEMENT_NODE
   && ((Element) node).getTagName()
    .equals(Address.XML_ADDRESS));
  }
  pointcut fromDOM(Node node): call(public static
   Address Address.fromDOM(Node)
   throws XMLDocumentException ) && args(node);
  before(Node node) throws XMLDocumentException:
   fromDOM(node){
    if (!isPreChecked(node))
      throw new XMLDocumentException(Address
      .XML_ADDRESS + " element expected.");
  }
}
```

Fig. 12. Precondition Checking aspect.

proposition of the aspect refactorings. For example, in most of the cases where a method must be encapsulated, a simple aspect refactoring, such as *Move Method from Class to Inter-type*, should be applied. In contrast, if the method to be encapsulated is a constructor method, a more specific aspect refactoring such as *Partition Constructor Signature* (Monteiro et al., 2004, 2005) should be applied. That is to say, the aspect refactoring can not be properly identified when there is no minimum support or confidence to generate a rule where a specific aspect refactoring is applied in a determined context. However, these cases are those which appear to a lesser extent.

7.2. Evaluation of the use of Markov models

The process properly identified 93% of the language elements to be encapsulated during the refactoring of the system. It was observed that most errors related to identification occurred when the encapsulation of a crosscutting concern was completed and then the process continued into the next CCC without recognizing the different structure of that concern. This is because ONISI bases its predictions of the next action on the latest actions for a specific state (by this we mean that the predictive accuracy of the algorithm improves after refactoring the first language element of a candidate aspect). That is why when the first candidate aspect of a CCC is refactored there is a period during which the algorithm learns the order in which the language elements must be encapsulated. For the same reason, the effectiveness of the identification could be reduced if the language elements that compose the candidate aspects of a CCC are very dissimilar or the developer switches between the refactoring of the elements of different CCCs. However, the use of a Markovian algorithm with these characteristics is also an advantage because it facilitates the fast adaptation to new situations. For this reason, we can think of the Markovian algorithm as an algorithm that it is always under training. While only one example of the structure of a candidate aspect has to be refactorized to be taken into account by the algorithm, the whole history of refactorings needs to be registered to run the algorithm. In this way, the algorithm recognizes new cases rapidly. However, the number of candidate aspects that a developer needs to refactorize before the algorithm successfully suggests solutions depends on each CCC.

Additionally, during the refactoring of the system, 320 additional restructurings were applied. In total, 81% of these were properly recommended by the process. In this case, problems in the recommendation were found when there were too many variations in the encapsulation of the language elements of candidate aspect. For example, during the refactoring of the *Exception Wrapping and Business Delegate* concern, the major variations were due to the number of catch clauses for a try statement. However, as shown in Fig. 13, variations tend to decrease as the process has more recorded history with reference to the concern encapsulation. In this figure the effectiveness in identifying additional restructurings for each candidate aspect of the *Exception Wrapping and Business Delegate concern* is shown. The effectiveness for a candidate aspect is calculated as

$$\frac{\sum \textit{additional activities successfully proposed}}{\sum \textit{additional activities required}}$$

where the sums run over all the language elements of the candidate aspect. As it is shown, the percentage of effectiveness in the identification of candidate aspects that contain too many variations was increasing during the course of refactoring. The drops in the curve are additional restructurings that were not properly recommended by the process because of variations in the structure of the language elements under refactoring.

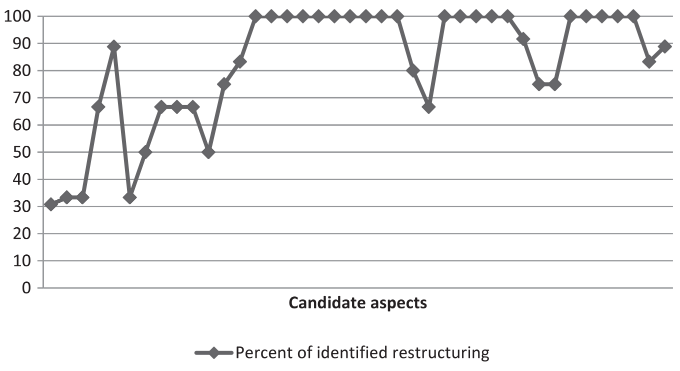


Fig. 13. Recommendation effectiveness of additional restructuring for the Exception concern.

Finally, an interesting remark to be taken into account in future works is the analysis of how the suggestions made through the Markovian algorithm depends on an specific application. That is, it is necessary to analyze how frequently the structures of candidate aspects and additional restructurings are detected, in different applications. This kind of analysis would allow us to identify if the *database of observed history* should be preserved after refactoring an application or, conversely, it should be reset for each application. Our work in the field indicates that the patterns detected are valid for different applications, especially if they present similar concerns. However, we think that a deeper analysis is needed.

7.3. Resulting code analysis

In order to show that the refactoring applied by means of the automatic algorithm increases the modularity of the source code, some metrics were collected. Significant improvements were identified in inheritance, coupling, and size in regard to the classes related to the refactored CCCs. Table 4 shows the measures obtained for the CCCs:

- Coupling Between Objects (CBO) The CBO metric quantitatively measures the coupling between classes (Chidamber and Kemerer, 1994). The more independent a class is, the easier reuse it becomes. So, a low value of CBO improves the modularity and the encapsulation of the system. To take into account the aspects we use an extension of this metric called Coupling between Components (CBC) (Garcia et al., 2005). In this case, as it is shown in Table 4, this value in the aspect-oriented system was reduced on average by 5.29% in regard to the object-oriented one (the values shown in the table are the percentage of change between the OO system and the AO one after summing of the values of the metric for each of the classes involved). That means that some couplings between components have been removed increasing the independence of classes and improving the reuse.
- Data Abstraction Coupling (DAC) The DAC measures the coupling caused by the abstract data types defined in a class (Li and Henry, 1993). Low values of DAC indicate better reuse. In the case of this CCC the AO refactored system reduced the values of DAC by

Table 4
Algo.

Concern	CBO	DAC	DIT	MPC	LOC
Exception Wrapping	−7.55%	−7.59%	0%	−25.34%	−2.37%
Service Locator	−2.17	−2.22	0%	−8.16%	−1.84%
Serializable Interfaces	−3.20%	−3.17%	−15.16%	−0.15%	1.15%
Precondition Checking	−8.26%	−8.23%	0%	−26.55%	−2.85%
Average	−5.29%	−5.3%	−3.79%	−15.05%	−1.48%

5.3% on average. This reduction is greater in *Exception Wrapping* and *Precondition Checking* due to the encapsulation of the exceptions (whose declarations are abstract data type) in aspects. Like the previous metric this result indicates an improving in the reuse of classes.

- Depth of Inheritance Tree (DIT) The DIT measures the number of super classes that can affect a class (Chidamber and Kemerer, 1994). The higher the value of DIT is, the greater the reuse of inherited methods it is. In this case, the value of DIT was unchanged in three of the CCCs because the encapsulation of these concerns is not directly related to inheritance. However, significant improvements were found in *Serializable Interfaces* with an improvement of 15.16%.
- Message Passing Coupling (MPC) The MPC measures the dependency of the method of a given class to a method implemented in other classes by means of its calls (Li and Henry, 1993). Low values of MPC imply more modularity. The aspect-oriented implementation of the CCCs *Exception Wrapping* and *Precondition Checking* reduced by more than 25% the value of MPC regarding the object-oriented implementation increasing the modularization of the system and as consequence improving the reuse and evolution. The change of the value of *Serializable Interfaces* is slight because its refactoring is focused on interface declaration rather than methods.
- Lines of Code (LOC) This metric counts the lines of code of a software entity (The aspects were taken into account during the calculation). In this case, the aspect-oriented system needed fewer operations than the object-oriented one (including the size of pointcut and advice declarations). In this way the LOC value was reduced on average by 1.48% as a result of the refactoring. Contrary to what might be expected, we have not found an important reduction in the number of lines of code. This is because, for example, in the *Serializables Interfaces* concern the LOC needed to create an aspect with the declare parents statements is larger than the LOC removed from the classes.

8. Experiment with users

Scoping. With the goal of evaluating if our approach helps developers to achieve the task of refactoring an OO system into an AO one, we conducted an experiment with undergraduate students. Specifically, we wanted to analyze if the refactoring time is reduced when AspectRT is used, compared with the refactoring without assistance. *Experiment planning*. We performed the experiment in the context of a university course of aspect-oriented software development. For this reason, the experiment is run offline. The undergraduate students are in their fourth and fifth year at the university. All of the students had previous experience with Java and OOP. Also, all of them had attended course classes, a tutorial on AspectJ and two laboratory classes where they practiced how to implement aspects. Additionally, they attended a tutorial of AspectRT in which how to use it was explained. For these reasons, it is possible to assume that their experience in AOP is, in general, the same. The research question that we have to answer is: Will a developer accelerate the refactoring of CCCs of a system by using AspectRT? In consequence, the null hypothesis of the experiment is:

- H_0: students that use AspectRT will spend on average the same time in refactoring CCCs as those students without assistance from AspectRT.

while the alternative hypothesis is:

Table 5
Time expended in refactoring.

	Using AspectRT	Without assistance
	102 m	117 m
	95 m	107 m
	87 m	112 m
Average	94.6 m	112 m
Std. Dev.	6.13 m	4.08 m
Variance	37.5 m	16.6 m

Table 6
Results from the Wilcoxon-test.

Factor	n	T^+	T^-
Time using AspectRT vs. without assistance	3	0	6

- H_1: students that use AspectRT will refactor CCCs faster than those students without assistance from AspectRT.

Experiment operation. The students were divided randomly into six groups of three people (this division was motivated by the number of available PCs in the laboratory at the time the experiment was run). Then the task of refactoring the *Command Concern* (Marin et al., 2007) of JHotDraw[9] was assigned to each group. We randomly defined the 3 groups that would fulfill this task using AspectRT. The remaining 3 groups accomplished the refactoring without tool.

The *Command Concern* refers to a common operation, in this case called *execute()*, that is implemented in the class *AbstractCommand* which is invoked by several clients. All the clients are the subclasses of the class *AbstractCommand* that make the call *super.execute()* at the beginning of the overwritten method *execute()*. In total, 17 calls exist to *super.execute()* that should be refactored. As proposed by Marin et al. (2007), in order to encapsulate the CCC, the calls to *super.execute()* should be encapsulated into an aspect to avoid the scattered code. Additionally, since AspectJ does not support the 'super' calls, the code of the method *AbstractCommand.execute()* should be moved to the aspect.

The experiment lasted 3 h: During the first 30 min we explained the CCC to be refactored. Then, the students were given up to 2 h to complete the refactoring. The groups that used AspectRT received an XML file with the crosscutting concerns to be refactored. Those groups that accomplished the refactoring without assistance received the same information in a TXT file.

Analysis and interpretation. During the experiment, we measured the time each group spent to accomplish the refactoring (Table 5). Those groups that used the tool spent, on average, 17 min less than those groups that did not use it.

In order to prove the existence of any statistically significant difference in the total time spent on refactoring we used the Wilcoxon-test (Wohlin et al., 2000). From Table 6, the null hypothesis (H_0) can be rejected with a one tailed test with a probability of error (or significance level) $\alpha = 0.05$ (i.e. there is a 5% chance of wrongly rejecting H_0) and a p-value of 0.05. Since $W = \min(T^+, T^-) = 0$ the effect size (Arcuri and Briand, 2011) is 0.

After a manual inspection of the refactored code, we found that two of the groups that refactored the concern without assistance partially encapsulated the concern. Specifically, they omitted to refactor 2 and 3 of the calls to the execute method respectively.

At the end of the experiment all participants who made the refactoring using AspectRT filled in a survey about the tool and the refactoring experience. The survey results are shown in Table 7. While this experiment is not completely comprehensive, it shows that the participants found the tool easy to use, and that they found

[9] http://www.jhotdraw.org, version 5.4b1.

Table 7
Results of the survey.

Statement	Strongly disagree	Disagree	Neither	Agree	Strongly agree
AspectRT is easy to use			1	2	6
The suggestions of fragments of code are useful			2	3	4
The suggestions of aspect refactorings are useful				2	7
The suggestions of additional restructuring are useful			2	5	2

the tool helpful to refactor a crosscutting concern. Specifically, we found that 66.6% of the participants strongly agree that AspectRT is easy to use. Also, 77.7% of them agree or strongly agree that the suggestion of fragments of code are useful. The same percentage strongly agree that the suggestions of aspect refactorings are useful. Finally, while a 55.5% of the participants agree that the suggestions of additional restructurings are useful only a 22.2% strongly agree. We think that this is because very few opportunities to make this kind of suggestions occurred during the experiment.

9. Threats to validity

Conclusion validity. While we have used a well known statistical technique a threat to conclusion validity is the low number of subjects in the experiment with users. This may reduce statistical power of the technique used to reveal patterns in the data. The same is true for the case study presented in Section 6 since only one system was refactored.

Internal validity. The main threat to internal validity in the experiment with users is the selection of subjects. That is, the selection of the subjects was not random because they were selected from the students that were attending a course. Additionally, other threat is that the refactoring was accomplished by groups instead of independently. However, since all the groups had the same number of members and their background knowledge is, in general, the same, we think that the results are still valid. A similar problem occurs to the threats to internal validity of the case study. For instance, the structural variability between the candidate aspects of a CCC could not be representative enough for the generalization of the results to any OO system. While the refactored concerns present similar structures for the candidate aspects, further experimentation is required.

Construct validity. A threat to construct validity of the experiment with users is hypothesis guessing. That is, while the students did not know which hypotheses were stated, they may intend to guess what was the result of the experiment. Regarding the case study, the main threat to construct is that the study was conducted with a single system which may under-represent the construct.

External validity. Since the case studies have been performed employing one project and a CCC, the external validity of the analysis is compromised. That is, the generalization of the results is limited and further empirical results are needed to strengthen the observations. Additionally, while the subjects of the experiment were advanced students, the applicability of the results to the software industry could be reduced since they were not developers.

10. Related work

The refactoring of OO systems by means of aspect refactorings has been discussed in several papers. Some of these works have focused their effort on the description of specific refactorings for aspect orientation or the adaptation of existing OO refactorings to AOP (Hanenberg et al., 2003; Iwamoto and Zhao, 2003; Laddad, 2002; Monteiro et al., 2004; Malta and de Oliveira Valente, 2009). This group of refactorings fulfills the activities needed to migrate an object-oriented system to an aspect-oriented one: the encapsulation of a CCC into an aspect (Refactorings of CCCs), the improvement of the internal structure of aspects (Refactorings to AOP constructs), and the use of traditional OO refactorings which ensures the correct update of the references to the AOP constructions (Aspect-Aware OO refactorings) (Hannemann, 2006). Taking into account the definition of the aspect refactorings, approaches of different granularity were presented in order to refactor and migrate OO systems to AO ones.

Low granularity This kind of approaches are based on programing languages elements. That is to say, the refactoring is oriented to encapsulate a CCC into an aspect focusing on the language elements where the CCC is spread. For this reason, low granularity approaches are usually independent of the aspect mining technique used. Our approach is of this type.

At this granularity level, Ceccato and Tonella (Ceccato, 2008; Tonella and Ceccato, 2005) present an approach restricted to refactoring scattered methods declared by interfaces (called aspectizable interfaces) and to encapsulating portions of code by means of pointcuts. Similarly to our work, this work uses a small set of aspect refactoring to perform the restructuring. While these refactorings can be applied automatically, this work does not present an automatic identification method for them. Binkley et al. (2005) present a semiautomatic iterative process which has 4 steps. First, given a fragment of aspectizable code, an OO refactoring is selected to be applied to it in order to enable the application of an aspect refactorings. Second, the OO refactoring is applied. Third, a suitable aspect refactoring is selected. In contrast to our approach, this task is carried out with a prioritization scheme that helps the developer in the refactoring selection. Finally, the aspect refactoring is applied.

van Deursen et al. (2005) propose a manual refactoring and testing strategy which aims to guide the developer during the migration. Hannemann et al. (2003) present two refactoring approaches that are based on a dialog with the developer. Similar to our work, one approach is based on the description of a crosscutting concern in the code. The other approach tries to refactor GoF design patterns (Gamma et al., 1995) focusing on the components of the pattern. The main difference between these approaches and ours is that our approach helps the developer by making suggestions during the refactoring.

High granularity This kind of approach tries to encapsulate into an aspect an architectural pattern that represents a CCC. That is, these approaches are focused on the refactoring of a specific type of concern. Generally, the high granularity processes comprise an aspect mining process which identifies the whole pattern and an aspect refactoring process which encapsulates the pattern into an aspect applying a set of refactorings. Consequently, this kind of refactoring is closely tied to the way that patterns are identified.

At this granularity level, Hannemann et al. (2005) present a role-based refactoring approach. Toward this goal, the CCCs are described using abstract roles. In the refactoring process, the developer chooses an appropriate refactoring for a role and then a mapping is done between the abstract CCC description and the program elements. Later, the refactoring is planned and executed. Marin et al. (2005) describe a process whereby the CCCs are described as types. Later, the described types are manually refactored using different kinds of aspect refactorings. da Silva et al. (2009) present an approach of metaphor-driven heuristics and associated refactorings. The refactoring of the code proposed is applicable on two concerns metaphors. These metaphors are recurrent code patterns that allow the identification of crosscutting concerns. During the identification and refactoring of the concern a developer interaction is needed. van der Rijst et al. (van der Rijst et al., 2008; Marin et al., 2009) propose a migration strategy based on crosscutting concern sorts. Once the CCC are described by means of concern sorts, they are refactored through interaction with the developer. In order to refactor the code, specific sorts that indicate what refactorings should be applied are used. Unlike our approach, these approaches propose refactoring strategies to specific cases of CCCs, while our approach can refactor any kind of concern.

We think that the aspect mining outputs of this kind of granularity can be easily adapted to be used in low granularity approaches.

Other works have also explored the automation of the refactoring process through machine learning. Cordy et al. (2002) propose the use of the TXL language to accomplish different code restructurings by means of a rule-based approach. Baxter et al. (2004) present an approach based on the theory of change to perform large-scale code transformations. This work was implemented as a commercial tool to assist developers in system maintenance. Tokuda and Batory (2001) analyze the automation of refactorings in three kinds of design evolution. The automation is based on checking conditions to apply a refactoring and the preservation of the behavior of the code. Weimer et al. (2009) present a technique to find and repair bugs in legacy applications. This technique uses a genetic programing algorithm to automate the analysis of possible solutions for a bug. Zibran and Roy (2011) propose the automation of code clone refactoring by analyzing the dependencies and conflicts among the refactorings that should be applied. For this analysis, this work uses a constraint programing approach.

11. Conclusions

The migration of OO systems into AO ones improves the separation of concerns thereby increasing the evolution, adaptation, and maintenance capabilities of systems. However, this is a difficult and time-consuming task that requires a lot of knowledge on the part of the developer. In this paper the assistance for a refactoring process that encapsulates CCCs into aspects is proposed. The assistance is based on the use of artificial intelligence techniques which are used to capture the user's behavior when he/she is refactoring a system so that this knowledge can be used later in similar situations. Specifically, association rules were used to determine a suitable aspect refactoring for a fragment of aspectizable code. In addition, Markov models were employed to identify the fragments of code to be migrated and to recommend additional restructurings when the application of an aspect refactoring is not sufficient.

In order to prove the benefits of the approach, the Java Pet Store Demo system was refactored. It was observed that much of the refactoring process was properly assisted. For this reason, developer interventions during the refactoring were significantly reduced. The algorithms based on association rules and Markov performed well. However, in regard to the latter, some errors were identified in the recommendation of additional restructurings when the language elements that compose the candidate aspects of a CCC differed greatly.

While the approach helps the developer during the selection of the aspectizable code to be encapsulated and the selection of the aspects refactorings to be used, the interaction with the developer is necessary during the refactoring process. For example, even though the approach can propose additional restructurings after applying an aspect refactoring, the developer must determine if the application of the aspect refactoring was sufficient to encapsulate the CCC under analysis.

In future work, we will conduct other case studies with the goal of generalizing our results of the refactoring of systems. We also plan to focus on other steps of the entire refactoring process (Vidal et al., 2009; Vidal and Marcos, 2009b) which involves the application of Aspect-Aware OO refactorings and refactorings to AOP constructs. Finally, we hope to analyze strategies for mapping between the output of high granularity aspect mining techniques and the input of our approach.

Acknowledgment

We would like to thank Alexandre Bergel for his helpful comments and suggestions.

References

Abait, E.S., Vidal, S.A., Marcos, C.A., Casas, S.I., Osiris Sofia, A.A., 2010]. Quality and communicability for interactive hypermedia systems: concepts and practices for design. In: IGI Global, Ch. An Integrated Process for Aspect Mining and Refactoring, pp. 175–193.

Agrawal, R., Srikant, R., 1994]. Fast algorithms for mining association rules. In: Proc. 20th Int. Conf. Very Large Data Bases, VLDB, Morgan Kaufmann, pp. 487–499.

Arcuri, A., Briand, L.C., 2011]. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Taylor, R.N., Gall, H., Medvidovic, N. (Eds.), ICSE. ACM, pp. 1–10.

Baxter, I.D., Pidgeon, C., Mehlich, M.,2004]. Dms&#174;: program transformations for practical scalable software evolution. In: Proceedings of the 26th International Conference on Software Engineering. ICSE '04. IEEE Computer Society, Washington, DC, USA, pp. 625–634 http://portal.acm.org/citation.cfm%253Fid=998675.999466

Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P.,2005]. Automated refactoring of object oriented code into aspects. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 27–36.

Ceccato, M.,2008]. Automatic support for the migration towards aspects. In: CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, Washington, DC, USA, pp. 298–301.

Chidamber, S.R., Kemerer, C.F., 1994]. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (June (6)), 476–493.

Colyer, A., Clement, A., Harley, G., Webster, M., 2004]. Eclipse AspectJ: Aspect-oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley Professional.

Cordy, J.R., Dean, T.R., Malton, A.J., Schneider, K.A., 2002]. Source transformation in software engineering using the TXL transformation system. Information and Software Technology 44 (13), 827–837.

da Silva, B.C., Figueiredo, E., Garcia, A., Nunes, D., 2009]. Refactoring of crosscutting concerns with metaphor-based heuristics. Electronic Notes in Theoretical Computer Science 233, 105–125.

Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., Aho, A., 2008]. Do crosscutting concerns cause defects? IEEE Transactions on Software Engineering 34 (July–August (4)), 497–515.

Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Figueiredo, E., Cacho, N., Lopes, F., Temudo, N., Silva, L., Soares, S., Rashid, A., Masiero, P., Batista, T., Maldonado, J.,2010]. An exploratory study of fault-proneness in evolving aspect-oriented programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1. ICSE '10. ACM, New York, NY, USA, pp. 65–74, http://dx.doi.org/10.1145/1806799.1806813.

Fowler, M., 1999]. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M., 2001. Inversion of Control Containers and the Dependency Injection Pattern. http://www.martinfowler.com/articles/injection.html

Gamma, E., Helm, R., Johnson, R.E., 1995]. Design Patterns. Elements of Reusable Object-Oriented Software, 1st ed. Addison-Wesley Longman, Amsterdam.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.,2005]. Modularizing design patterns with aspects: a quantitative study. In: AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development. ACM, New York, NY, USA, pp. 3–14.

Gorniak, P., Poole, D.,2000]. Predicting future user actions by observing unmodified applications. In: AAAI/IAAI. AAAI Press/The MIT Press, pp. 217–222.

Gradecki, J.D., Lesiecki, N., 2003]. Mastering AspectJ: Aspect-Oriented Programming in Java. John Wiley & Sons, Inc., New York, NY, USA.

Han, J., Kamber, M., 2000, September. Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems), 1st ed. Morgan Kaufmann.

Hanenberg, S., Kleinschmager, S., Josupeit-Walter, M.,2009]. Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. ESEM '09. IEEE Computer Society, Washington, DC, USA, pp. 156–167, http://dx.doi.org/10.1109/ESEM.2009.5316028.

Hanenberg, S., Oberschulte, C., Unland, R., 2003]. Refactoring of aspect-oriented software. In: Proc. Int'l Conf. Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), pp. 19–35.

Hannemann, J., 2006]. Aspect-oriented refactoring: classification and challenges. In: LATE '06.

Hannemann, J., Fritz, T., Murphy, G.C.,2003]. Refactoring to aspects: an interactive approach. In: Eclipse '03: Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange. ACM, New York, NY, USA, pp. 74–78.

Hannemann, J., Murphy, G.C., Kiczales, G.,2005]. Role-based refactoring of crosscutting concerns. In: AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development. ACM, New York, NY, USA, pp. 135–146.

Iwamoto, M., Zhao, J.,2003]. Refactoring aspect-oriented programs. In: The 4th AOSD Modeling With UML Workshop, UML'2003. ACM, New York, NY, USA.

Kellens, A., Mens, K., Tonella, P., 2007]. A survey of automated code-level aspect mining techniques. In: Transactions on Aspect-Oriented Software Development (TAOSD) IV (Special Issue on Software Evolution), pp. 143–162.

Kiczales, G., Lamping, J., Mendheka, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J., 1997, June. Aspect-Oriented Programming. In: Gjessing, S., Nygaard, K. (Eds.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 1241. Springer, Finland.

Laddad, R., 2002]. I Want My AOP: Separate Software Concerns with Aspect-oriented Programming. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html

Laddad, R., 2003a]. Aspect-oriented Refactoring. http://www.theserverside.com/news/1365184/Part-2-The-Techniques-of-the-Trade

Laddad, R., 2003b]. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co., Greenwich, CT, USA.

Li, W., Henry, S., 1993 May. Maintenance metrics for the object-oriented paradigm. In: Proc. IEEE Symp. Software Metrics, pp. 52–60.

Malta, M.N., de Oliveira Valente, M.T., 2009]. Object-oriented transformations for extracting aspects. Information and Software Technology 51 (1), 138–149.

Marin, M., Moonen, L., van Deursen, A.,2005]. An approach to aspect refactoring based on crosscutting concern types. In: MACS '05: Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software. ACM, New York, NY, USA, pp. 1–5.

Marin, M., van Deursen, A., Moonen, L., 2007]. Identifying crosscutting concerns using fan-in analysis. ACM Transactions on Software Engineering and Methodology 17 (1), 1–37.

Marin, M., van Deursen, A., Moonen, L., Rijst, R., 2009]. An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw. Automated Software Engineering 16 (2), 323–356.

Mens, T., Tourwe, T., 2008]. Evolution issues in aspect-oriented programming. In: Mens, T., Demeyer, S. (Eds.), Software Evolution. Springer, pp. 203–232.

Mesbah, A., van Deursen, A.,2005]. Crosscutting concerns in J2EE applications. In: WSE '05: Proceedings of the Seventh IEEE International Symposium on Web Site Evolution. IEEE Computer Society, Washington, DC, USA, pp. 14–21.

Monteiro, M.P., Fernandes, Jo, a.M.,2004]. Object-to-aspect refactorings for feature extraction. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'2004). ACM Press, p. 2004.

Monteiro, M.P., Fernandes, Jo, A.M.,2005]. Towards a catalog of aspect-oriented refactorings. In: AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented Software Development. ACM, New York, NY, USA, pp. 111–122.

Parnas, D.L., 1972]. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15 (12), 1053–1058.

Rabiner, L.R., 1989 Feb. A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77 (2), 257–286.

Srikant, R., Vu, Q., Agrawal, R.,1997]. Mining association rules with item constraints. In: Proceedings of the Third International Conference of Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA, pp. 67–73.

Tokuda, L., Batory, D., 2001]. Evolving object-oriented designs with refactorings. Automated Software Engineering 8 (January), 89–120 http://portal.acm.org/citation.cfm%253Fid=591992.592064

Tonella, P., Ceccato, M., 2005]. Refactoring the aspectizable interfaces: an empirical assessment. IEEE Transactions on Software Engineering 31 (10), 819–832.

van der Rijst, R., Marin, M., van Deursen, A.,2008]. Sort-based refactoring of crosscutting concerns to aspects. In: LATE '08: Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution. ACM, New York, NY, USA, pp. 1–5.

van Deursen, A., Marin, M., Moonen, L., 2005]. A Systematic Aspect-oriented Refactoring and Testing Strategy, and its Application to JHotDraw. CoRR abs/cs/0503015.

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

Vidal, S., Marcos, C., 2009a]. Identificacin automtica de refactorings. In: Tenth Argentine Symposium on Software Engineering (ASSE 2009), 38 JAIIO (Jornadas Argentinas de Informtica).

Vidal, S., Marcos, C., 2009b]. Un proceso iterativo para la refactorizacin de aspectos. Revista Avances en Sistemas e Informtica 6 (1), 93–103.

Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.,2009]. Automatically finding patches using genetic programming. In: ICSE. IEEE, pp. 364–374.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000]. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA.

Zibran, M.F., Roy, C.K., 2011]. Conflict-aware optimal scheduling of code clone refactoring: a constraint programming approach. In: ICPC, pp. 266–269.

Santiago Vidal graduated in system engineer from UNICEN University, Argentina in 2008. He is a Ph.D. candidate in Computer Science at ISISTAN Research Institute. Currently is a scholarship holder of the National Council for Scientific and Technological Research of Argentina (CONICET). His main research interests include aspect-oriented software development, software evolution and system maintenance.

Claudia Marcos has been a Professor in the School of Computer Science at UNICEN since 1991. From 2000 to 2005 she was co-director of the ISISTAN Research Institute. She is a CIC (Committee for Scientific Research of the Buenos Aires province) researcher. Her main research area is in software evolution, aspect-oriented development, and agile development. She teaches several undergraduate and postgraduate courses at the UNICEN and has also national and international publications in the area. She leads several university research projects in Argentina as well as abroad. At present, Dr. Marcos is advising postgraduate and undergraduate students. Dr. Marcos received her B.S. degree in 1993 from the UNCPBA State University in 1993. She obtained her Ph.D. degree in Computer Science in 2001.