Expert Systems with Applications 39 (2012) 3810–3816

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

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

Building an expert system to assist system refactorization

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

ARTICLE INFO

Keywords:
Interface agents
Expert systems
Aspect-oriented software development
Aspect refactoring

ABSTRACT

The separation of concerns is an important issue in the building of maintenable systems. Aspect oriented programming (AOP) is a software paradigm that allows the encapsulation of those concerns that crosscut a system and can not be modularized using current paradigms such as object-oriented programming. In this way, AOP increases the software modularization and reduces the impact when changes are made in the system. In order to take advantage of the benefits of AOP, the legacy OO systems should be migrated. To migrate object-oriented systems to aspect-oriented ones, specific refactorings for aspects should be used. This is a complex and tedious task for the developer because he/she needs to know how the refactorings should be applied and under what context. Therefore, it is desirable to have tools that help him/her through the process. In this article, we present an expert software agent, named *RefactoringRecommender*, that assists the developer during a refactorization of a system. The agent uses a Markovian algorithm with the goal of predicting the needed restructurings.

1. Introduction

In order to develop highly extensible, reusable, adaptable, and modifiable systems it is important to have proper separation of concerns (Clements & Kazman, 2003). Building systems with these quality attributes improves system modularization and maintenance. Proper separation of concerns can be achieved by means of object-oriented programming (OOP); however, there are some concerns that orthogonally crosscut the components of a system whose encapsulation is almost unviable. These kinds of concerns are called crosscutting concerns (CCCs). Aspect-oriented programming (AOP) is a software paradigm that complements OOP to address the problem of separation of concerns (Kiczales et al., 1997). AOP allows the encapsulation of CCCs into new components called aspects (Elrad, Filman, & Bader, 2001). Typical examples of CCCs are exception handling, logging, and concurrency control.

So as to take advantage of the benefits of aspect-oriented development (Binkley, Ceccato, Harman, Ricca, & Tonella, 2005; Ceccato, 2008; da Silva, Figueiredo, Garcia, & Nunes, 2009; Hannemann & Kiczales, 2002; Malta & de Oliveira Valente, 2009; Marin, Deursen, Moonen, & Rijst, 2009; Monteiro & Fernandes, 2008; van Deursen, Marin, & Moonen, 2005) and to achieve a system with the above listed quality attributes, legacy object-oriented systems should be migrated to aspect-oriented systems. To attain this goal, two activities are used: aspect mining and aspect refactoring (Kellens, Mens, & Tonella, 2007). During the aspect mining activity, the crosscutting concerns that may potentially become aspects must be discovered in the source code. Usually, there are different instances of each CCC in a system, they are called candidate aspects. Once the candidate aspects have been identified, they must be converted, during the aspect refactoring activity, to final aspects using a specific aspect language like AspectJ[1] or a framework that supports aspects like Spring/AOP.[2] Aspect refactorings are mechanisms or strategies of code restructuring similar to the OO refactorings presented by Fowler (1999) but with the difference that aspect refactorings take into account the presence of aspects in the source code. This kind of restructuration is used to encapsulate into aspects the CCCs found in the OO source code, to improve the internal structure of the created aspects and to apply OO refactorings updating the references to the AOP constructions (Hannemann, 2006). In this paper, we focus on the aspect refactoring activity.

The aspect refactoring activity is often tedious and repetitive but, at the same time, it is also a complex process. On the one hand, the refactorization of the candidate aspects of the same CCC are based on the application of a set of aspect refactorings because, in general, the candidate aspects are patterns in the source code (da Silva et al., 2009; Hannemann & Kiczales, 2002). That is, all

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

[1] http://www.eclipse.org/aspectj/.
[2] http://www.springsource.org/.

0957-4174/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2011.09.084

the candidate aspects belonging to the same concern have a similar code structure. On the other hand, there are variations in the implementation of candidate aspects that cause differences in the code transformations to be performed. This happens when, after applying an aspect refactoring, additionals restructurings should be applied with the goal of ensuring that the external behavior of the code is not altered.For these reasons, it is important to use refactoring tools that have user-level assistance in order to help the developer during the refactorization of a system.

In order to build a tool with these characteristics we argue that interface agents (Maes, 1994) are an appropriate approach. We base our decision on the fact that interface agents learn users' habits, preferences and interests to provide them personalized assistance with the tasks they perform with software applications (Hsu & Ho, 1999; Lieberman, 1997). Applying this to the refactorization of OO systems to AO ones, we envision the following scenario: (i) a developer begins the refactorization of a system being based on the candidate aspects identified previously by aspect mining; (ii) iteratively each candidate aspect is refactorized; (iii) an interface agent designed to help the developer during the refactorization proposes the elements to be encapsulated, the aspect refactorings to be applied, and the additional restructurings when necessary.

Along this line, we have developed an expert software agent called *RefactoringRecommender* which observes an aspect refactoring tool called AspectRT (Vidal, Abait, Marcos, Casas, & Díaz Pace, 2009; Vidal & Marcos, 2009b) and proposes the aforementioned refactoring activities to the developer. *RefactoringRecommender* implements a Markovian algorithm (Rabiner, 1989) with the goal of predicting the actions of the developer based on a recorded history of previous refactorizations and using a model that represents the refactoring process impose by AspectRT. By mean of this technique it is possible to identify the variations between the implementations of the candidate aspects and propose restructurations accordingly. That is to say, the use of this technique allows the incorporation of new restructuration cases when they emerge and they are saved in the recorded history of previous refactorizations. Therefore, the main contribution of this paper is assistance to a developer during the refactorization of an OO system into an AO one. The assistance is offered in the form of a proposition of a fragment of aspectizable code to be migrated, the proposition of an aspect refactoring to be applied, and the recommendation of additional restructurings when necessary.

The rest of this paper is structured as follows: Section 2 presents an overview of our proposed approach; Section 3 describes the main characteristics of our approach; Section 4 presents the results obtained when evaluating our approach in a number of experiments; Section 5 analyzes some related works; and Section 6 outlines the conclusions and future work.

2. AspectRT agent's overview

The expert agent *RefactoringRecommender* helps developers during the refactorization of a system. Fig. 1 shows an overview of the *RefactoringRecommender* functionality. In this scenario a developer interacts with a tool called AspectRT.[3] This tool is implemented as a plug-in for the Eclipse IDE and it is integrated with AspectJ. AspectRT helps developers to refactor object-oriented systems to aspect-oriented ones, providing a set of aspect refactorings (Monteiro, 2004).

When a developer[4] is refactoring a system using AspectRT the *RefactoringRecommender* agent monitors the user's behavior registering the context in which the changes are applied and by this means it builds the evidence for future recommendations. In order to refactorize the system, the developer chooses iteratively a candidate aspect to be encapsulate into an aspect. Each candidate aspect is spread over several Java elements, e.g. methods, classes, interface declarations, statements, etc. To encapsulate a Java element into an aspect, an aspect refactoring is applied and additional restructurings are done when necessary. In particular, the agent registers the aspect refactoring applied and also to which kind of Java element it is applied. Also, the agent registers the order in which the kind of Java elements are selected to be refactorized and registers the additional restructurings done. This information collected by the agent matches with a Hidden Markov Model (HMM) (Rabiner, 1989) of the process of refactoring implemented by AspectRT.

When the agent accumulates enough history it starts to advice the user using a Markovian algorithm to calculate the probabilities of the application of a particular action.In order to do that, the agent observes the developer when he/she is refactoring a system to determine the states of the process of refactoring. Once the state has been determined, the agent recommends a list of possible actions. The developer can accept one of them or ignore the suggestions. The developer's decision to accept or reject a recommendation is used by the agent as implicit feedback. The possible kind of recommendations of *RefactoringRecommender* are:

- A Java element of a candidate aspect to be refactorized.
- An aspect refactoring to be applied on a Java element previously selected.
- Additional restructurings to an aspect refactoring in order to complete the encapsulation of a Java element.

In the next section how evidence is collected and how the recommendations are proposed are explained in detail.

3. Capture of the user's knowledge

RefactoringRecommender uses a Markovian algorithm to obtain knowledge about how a developer refactorizes a system encapsulating the CCCs into aspects. A Markovian algorithm enables us to learn from the developer's actions during the refactorization of a system and to identify the restructurations necessary for different contexts. In this way, this kind of technique allows a flexible adaptation of the agent under different situations when it offers a piece of advice to the developer. This flexibility and learning can not be achieved with others simple techniques such as "if-condition-then-action" rules (Vidal & Marcos, 2009a). In our context, a Markovian algorithm could be used, for example, to identify the restructuring needed to complete the encapsulation of a Java element into an aspect after the application of an aspect refactoring.

A 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 Markovian algorithm implemented by *RefactoringRecommender* approximates this assumption.

3.1. Building the evidence

The Markovian algorithm that supports the assistance process is the ON-line Implicit State Identification (ONISI) algorithm (Gorniak & Poole, 2000; Gorniak, 2000). ONISI allows the prediction of the future user actions by means of the use of a Markov model.

[3] Available from http://sites.google.com/site/legacyandaop/Home/ar.
[4] The words developer and user are used indistinctly in this article.

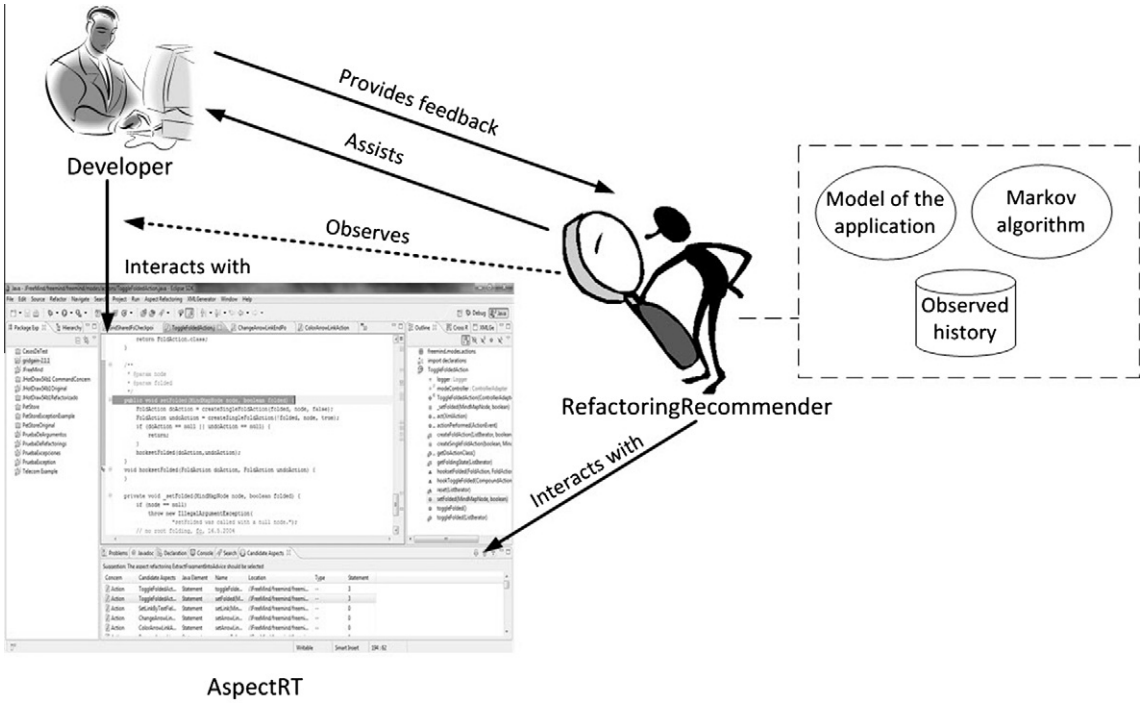

Fig. 1. Overview of our approach.

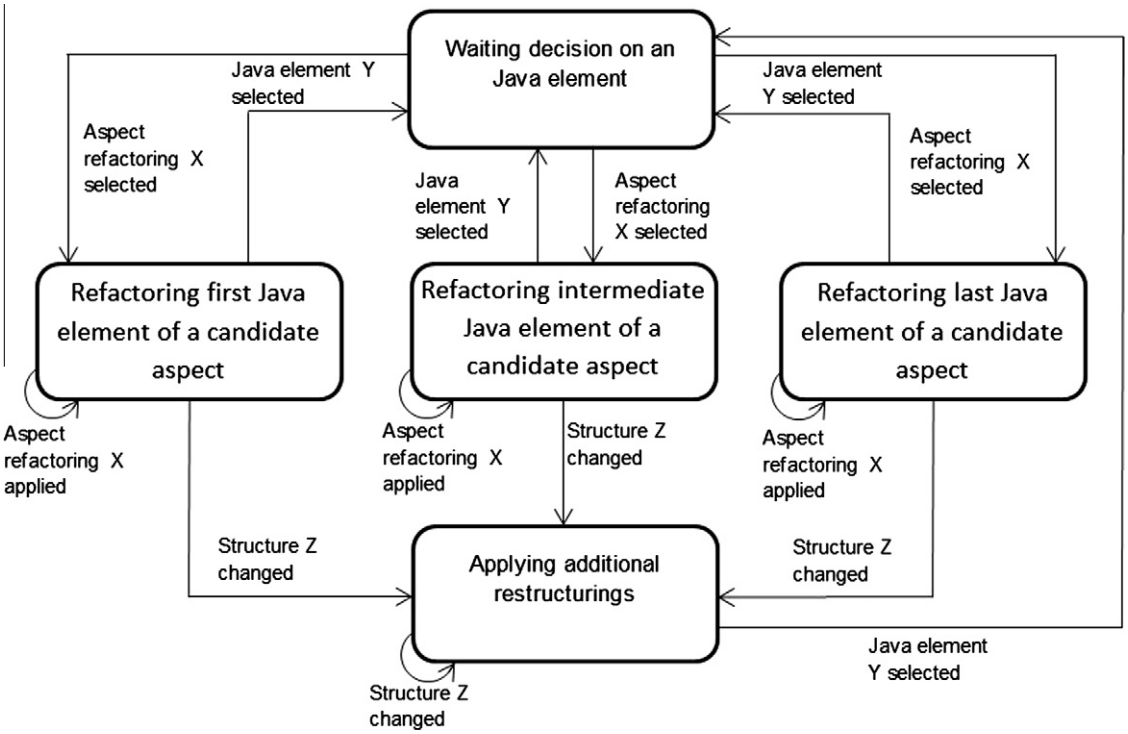

Fig. 2. Model of the refactoring process.

Specifically, by the observation of the interaction of the developer with AspectRT, ONISI infers a HMM composed of states and actions. This model is represented as a state machine in which the transitions from one state to another occur when certain actions are performed. As is shown in Fig. 2, the AspectRT model has five states. The first state, called "*Waiting decision on a Java element*", represents the situation in which a Java element has been selected and an aspect refactoring will be chosen to encapsulate the Java element. Once an aspect refactoring is selected, the model transitions to another state. This situation is represented by 3 different states: *"Refactoring first Java element of a candidate aspect"*, *"Refactoring intermediate Java element of a candidate aspect"*, and *"Refactoring last Java element of a candidate aspect"*. This differentiation occurs because additional activities are usually performed during the encapsulation of the first and last element (e.g. some structures are created to contain the changes or some actions are performed

in order to improve the resultant code). Finally, when an aspect refactoring has been applied, (i) an element could be selected transitioning to the first state; or (ii) additional activities to the refactoring could be applied transitioning to the state *"Applying additional restructurings"*.

In order to register the history of interactions between the user and the tool, the agent saves into a database each activity as a pair composed of a state and an action $\langle S,A\rangle$ where S is the state in which the model transitioned and A is the action which occurred in the model. Both the state and the action belong to the AspectRT model. For example, if the following scenario is considered:

1. A method is selected to be encapsulated into an aspect.
2. The aspect refactoring *Move Method from Class to Inter-type* (Monteiro, 2004) is selected.
3. The aspect refactoring is applied.
4. Another Java element that is a statement is selected.

The pairs of states and actions saved by *RefactoringRecommender* will be:

⟨Java element Method selected, Waiting decision on a Java element⟩.
⟨Aspect refactoring Move Method from Class to inter-type selected, refactoring first Java element of a candidate aspect⟩.
⟨Aspect refactoring Move Method from Class to Inter-type applied, Refactoring first Java element of a candidate aspect⟩.
⟨Java element Statement selected, Waiting decision on a Java element⟩.

Another source of knowledge of the agent, besides those that can be inferred from the context, is the feedback given by the developer. The developer expresses implicit feedback when he/she accepts (or not) a recommendation proposed by *RefactoringRecommender*. When the developer does not accept a recommendation, he/she can apply an alternative solution and the activities involved in this solution are saved as state-action pairs. In this way the algorithm can use this information to make successful future recommendations.

3.2. Making recommendations

The goal of our agent is to assist the developer during the refactorization of an OO system into an AO one by means of recommendations. These recommendations are oriented to help in the selection of a Java element to be encapsulated, the selection of the aspect refactoring to be used, and the application of additional restructurings to the refactorings when necessary.

With this goal in mind, the ONISI algorithm is used to analyze the interaction history and predict the most probable action to be taken given the current state. 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 found in the history. When the current state is observed by ONISI, probabilities to all possible actions from that state are assigned and ranked.

At its core, the algorithm uses two measures: the match length measure and the frequency measure (Gorniak & Poole, 2000; Gorniak, 2000). The former calculates the average of the lengths of the k longest sequences that end with a determined action in a specific state where k is a small integer. The latter counts the occurrence of an action in a state. The ranking of a possible action for the current state is calculated using a parameter $0 \leqslant \alpha \leqslant 1$ to indicate the weight between the match length measure normalized and the frequency measure, also normalized. So, every time a transition is detected in the model, ONISI is executed with the goal of proposing to the user possible actions to be made using the rankings obtained for the potential actions of the current state.

When a recommendation is made, the first three results of the ranking generated by ONISI are proposed to the developer in order (where the first is the most possible). As is shown in Fig. 3, the recommendations are made in a noninvasive way, in order not to interrupt the developer, using a recommendation bar that is integrated with AspectRT. Some examples of possible advice are: "The aspect refactoring *Extract Fragment into Advice* should be selected," "The Java element *Statement* should be selected," or an additional restructuring related to aspect languages as "A privileged statement should be added into the aspect." The recommendations are made in as much detail as possible in order to allow the automatic application of the changes by AspectRT when the user accepts a recommendation. This is especially useful in the recommendation of additional restructurings where there are several possible activities. In this case the level of detail is the change of a Java (or aspect language) structure. This kind of recommendation is proposed incrementally by the agent. That is, a restructuring recommendation is proposed and if accepted, then the next recommendation is suggested. If the user does not accept any of the three recommendations, he/she can proceed with a restructuration on his/her own. This solution is stored in the database. When a recommendation is accepted (or not) by the developer, the agent uses the developer's decision as an implicit feedback in order to use this knowledge in making future recommendations.

In regard to the values of the k and α parameters that are used in the algorithm, they were determined experimentally for this domain.We uses a $0.7 \leqslant \alpha \leqslant 0.9$ in order to give more 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 "Waiting decision on a Java element" 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 a priority to know the frequency that an aspect refactoring was selected after the selection of a Java element (which is measured by the match length measure). With respect to the k parameter (Gorniak & Poole, 2000) claim that low values show the same performance as high values.

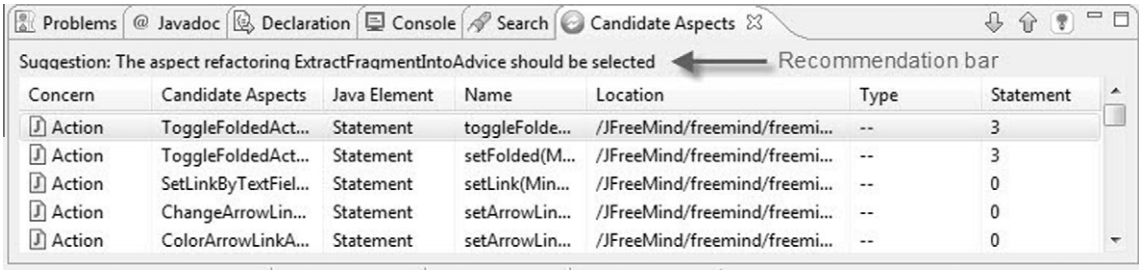

Fig. 3. Recommendation bar.

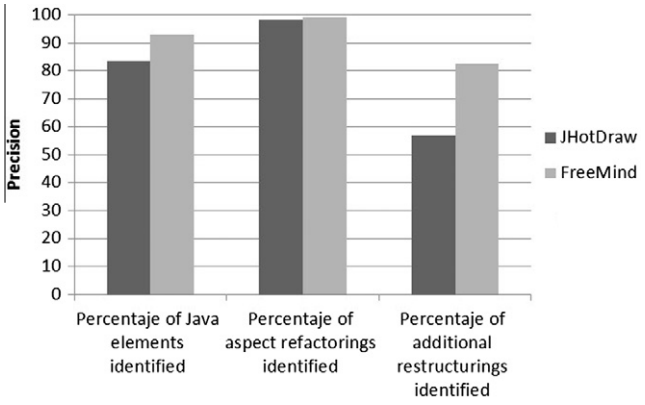

Fig. 4. Identification precision.

We experimentally obtained the same results, therefore we propose a $3 \leqslant k \leqslant 5$ value in order to make faster calculations.

4. Experimental results

In order to assess the recommendation precision of *Refactoring-Recommender* we conducted the refactorization of two OO systems: JHotDraw[5] and FreeMind.[6] Before the refactoring, the training of the tool was carried out using small examples of CCCs encapsulation, like the presented by Monteiro (2004) to illustrate his catalog. This training allowed the agent to identify the context in which an aspect refactoring is used and also some policies about the order in which the Java elements should be encapsulated. In contrast, the training did not contain situations of additional restructurings so as to show how the ONISI algorithm learns the developer's behavior during refactorization. The ONISI algorithm was configured using $k = 5$ and $\alpha = 0.9$.

The refactored CCCs from JHotDraw were *Command*, *Undo*, *Persistence*, *Observer* and *Composite* (Marin, Deursen, & Moonen, 2007) and in FreeMind they were *Action* and *XML Attribute Serialization* (Yuen & Robillard, 2007).

The average precision of the agent during the refactorization of JHotDraw was 84.27% as compared to FreeMind's 91.51%. The precision was determined by how many of the recommendations made by the agent were correct (where a recommendation is correct when it occupies the first position in the ONISI ranking). The recommendations taken into account were of the three possible kinds (the selection of a Java element to be encapsulated, the selection of an aspect refactoring to be applied, and the proposition of additional restructurings), however, differences in the recommendation precision among them were found. As can be seen in Fig. 4, the percentage of Java elements and aspect refactorings identified were above 80% for both systems as was the precision for the recommendation of additional restructurings for FreeMind. Instead, the precision for the recommendation of additional restructurings for JHotDraw was 57%. The differences between the recommendation precision of both system is due to the variation among the pattern structuresof the candidate aspects of a concern. We found that when the Java elements that compose the candidate aspects of a same concern varies greatly, the agent experiences a delay in adapting to the different candidate aspects. As consequence, these fails in the identification of restructurings impact the identification of a Java element. This is because in some of these cases, an additional step is wrongly proposed instead of the selection of a Java element. Also, we found that some of the additional restructurings proposed by the agent are not specific enough in some occasions. For example, the agent proposed making changes in the structure of a pointcut; however, the specific changes to be made in the pointcut were not proposed.

Since the lowest precision rates were in the recommendation of additional restructurings it is useful to discuss these recommendations. Fig. 5 shows the location of the ranking in which the correct recommendation for the proposition of additional restructurings appeared. As can be seen, taking into consideration the three recommendations that the agent proposes (i.e. the top three ONISI ranked positions and not simply the first), the precision increases to 68% (57% were ranked at first place and 11% were ranked between the second and third place) for JHotDraw and 93% (82% and 11%) for FreeMind. Additional significant percentages in both cases are those recommendations that did not appear in the ranking. This mainly happens when there is not enough information to predict an action for the current state or when there are variations on the Java elements for a candidate aspect.

The above situation in which there are variations on the Java elements, can be seen in the charts presented in Fig. 6. Chart (a) shows the refactorization of the Java elements for a candidate aspect of the *Command* concern comparing one scenario with previous training and another without training (only the recommendations of additional restructurings are shown). The letter R_{place} indicates the correct recommendation place of the ONISI ranking when it was not in the first position ($R\_$ indicates that the correct recommendation was not taken into account to generate the ranking). In the untrained scenario it is shown the situation where there is not enough information to predict an action and how the correct recommendation ascends in the ranking. For example, during the recommendation of the fourth Java element the ranking position for the correct action was 5, so it was not proposed. Later, during the recommendation of the fifth Java element (which was a case similar to the fourth Java element) the ranking position for the correct action was 2, so it was proposed in second place. From there the correct action was recognized in the first position. In the chart (a), is also shown the improvement when the agent has a previous training.

Fig. 6(b) shows the precision of the refactorization of the candidate aspects of the *Undo* concern in the recommendation of additional restructurings without training. As can be seen in the chart, during the refactorization of the first two candidate aspects, the recommendations are not successful. Then, the precision of the recommendation improves incrementally to achieve 100% accuracy. When a variation in the code structure of the candidate aspect occurs, there is a descent in the percentage of precision (i.e. the refactorization of the seventh candidate aspect). It was observed that the descent of precision for these changes gradually decreases over time because the agent learns the possible additional restructurings for a concern.

5. Related work

Since the emergence of AOP, the refactorization of systems has been widely investigated, and a variety of aspect refactorings and refactorization process have been proposed (Binkley et al., 2005; Ceccato, 2008; Hannemann, Fritz, & Murphy, 2003; Hanenberg, Oberschulte, & Unland, 2003; Iwamoto & Zhao, 2003; Laddad, 2002; Marin, Moonen, & van Deursen, 2005; Monteiro, 2004; van Deursen et al., 2005; Vidal et al., 2009). The majority of these processes try to refactorize an OO system to an AO one using different kinds of refactorings (Hannemann, 2006). However, few of them incorporate AI techniques to automatize the refactorization.

[5] http://www.jhotdraw.org, version 5.4b1.
[6] http://freemind.sourceforge.net, version 0.8.0.

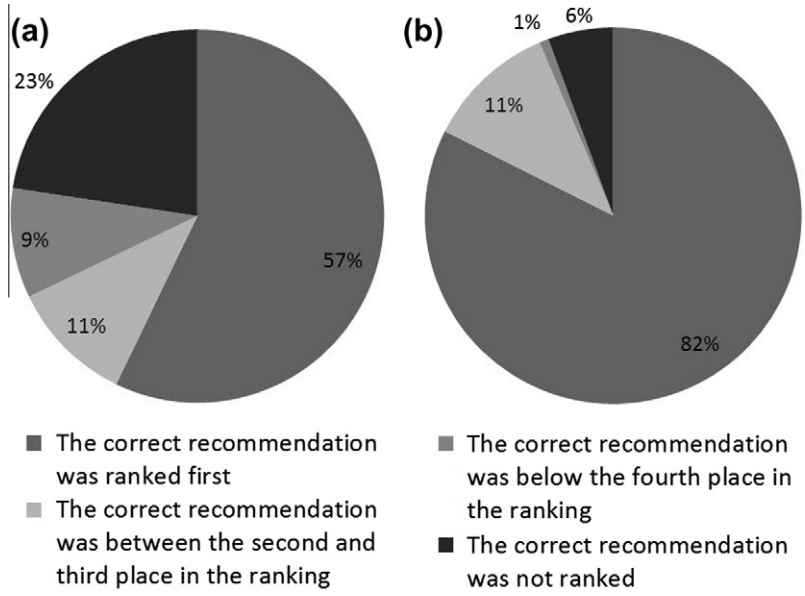

Fig. 5. Distribution of the recommendations in the ONISI ranking.

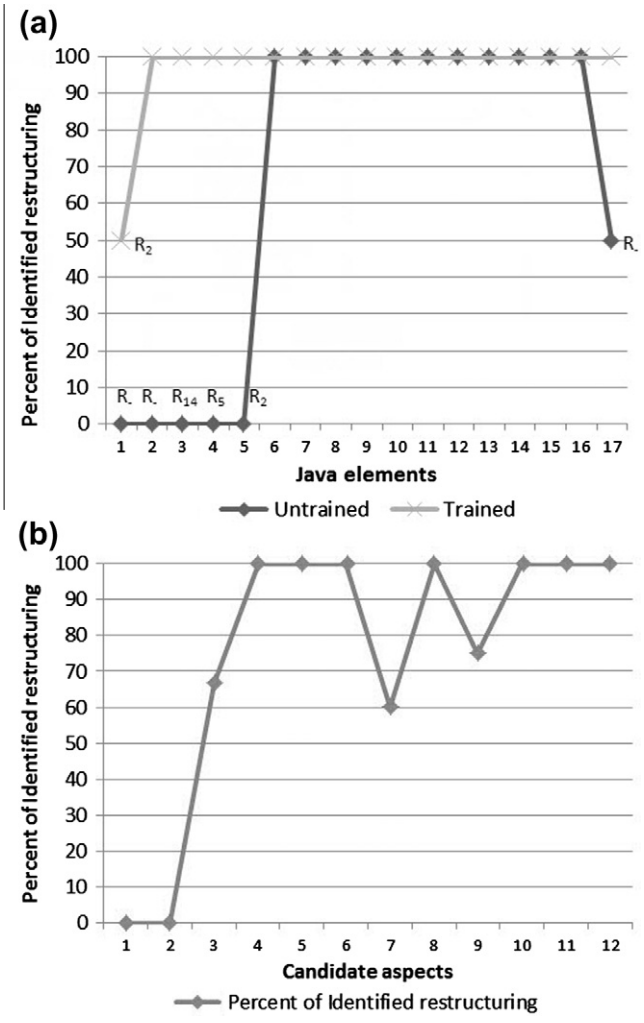

Fig. 6. Identification of additional restructurings.

The approaches presented by Binkley et al. (2005), Ceccato (2008) base their automation on iterative processes that use a small subset of aspect refactorings. The selection of these refactorings is accomplished by means of rules such as "if the code has a certain characteristic, then a particular refactoring is applied". In contradistinction to *RefactoringRecommen der*, these approaches do not fully cover the situation when additional restructurings must be done after the application of an aspect refactoring. Additionally, *RefactoringRecommender*, takes into account the preferences of the developer during the refactorization.

The processes presented in Marin et al. (2009), van der Rijst, Marin, and van Deursen (2008) use a type of CCC documentation called *crosscutting concerns sorts*. The refactorization of each CCC is done through a series of transformations associated with a *crosscutting concerns sorts*. When a compilation problem is found after the refactorization, these approaches propose possible solutions using the sort documentation. Unlike our approach, these processes do not fully cover those cases in which variations in the implementation of a CCC result in the incomplete encapsulation of the CCC.

da Silva et al. (2009) propose a high level technique of aspect refactoring based on metaphor-based heuristics. Similar to the techniques enunciate above, the refactorings are used according to the way in which the CCCs are documented. However, the approach only proposes a subset of possible refactorings, so the selection of a specific aspect refactoring and additional restructurings to be applied are done by the developer.

Finally, some OO refactoring approaches that incorporate AI techniques have been proposed (Hoffmann, Janssens, & Eetvelde, 2006; Kösker, Turhan, & Bener, 2009), but although useful, these techniques can not be transferred to AOP.

6. Conclusions and future work

In this paper, we presented an approach that assists the developer in the refactorization of an object-oriented system into an aspect-oriented one. Toward this goal, we have developed an agent called *RefactoringRecommender* which interacts with an aspect refactoring tool. The agent observes the developer when he/she is refactoring a system in order to recommend a fragment of a candidate aspect to be encapsulated, aspect refactorings to be applied, and additional restructuring to them. *RefactoringRecommender*uses a Markovian algorithm to made the recommendation analyzing the

interaction history of the user with the aspect refactoring tool. The acceptance (or not) of a recommendation by the developer is saved into the interaction history in order to improve the agent's knowledge.

The results obtained when evaluating the agent demonstrated the advantages of using an agent-based approach during the refactorization of systems. Along this line, despite some limitations, *RefactoringRecommender* has corroborated our feelings about the advantages of agent-based assistance, reducing the developer's intervention during the refactoring process.

As future work, we are planning to refine the recommendations of additional restructurings in order to enable their automatic application when they are accepted by the developer.

References

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* (pp. 27–36). Washington, DC, USA: IEEE Computer Society.

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* (pp. 298–301). Washington, DC, USA: IEEE Computer Society.

Clements, P., & Kazman, R. (2003). *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

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.

Elrad, T., Filman, R. E., & Bader, A. (2001). Aspect-oriented programming: Introduction. *Communications of the ACM, 44*(10), 29–32.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Gorniak, P. (2000). Keyhole state space construction with applications to user modeling. Ph.D. Thesis, University of British Columbia.

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

Hanenberg, S., Oberschulte, C., & Unland, R. (2003). Refactoring of aspect-oriented software. In *Proceedings of international conference on 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* (pp. 74–78). New York, NY, USA: ACM.

Hannemann, J., & Kiczales, G. (2002). Design pattern implementation in java and aspectj. *SIGPLAN Not, 37*(11), 161–173.

Hoffmann, B., Janssens, D., & Eetvelde, N. V. (2006). Cloning and expanding graph transformation rules for refactoring. *Electronic Notes in Theoretical Computer Science, 152*, 53–67.

Hsu, C.-C., & Ho, C.-S. (1999). Acquiring patient data by an intelligent interface agent with medicine-related common sense reasoning. *Expert Systems with Applications, 17*(4), 257–274<http://www.sciencedirect.com/science/article/B6V03-3XSJS5W-3/2/1774b19527b11600db2a92724b53da05>.

Iwamoto, M., & Zhao, J. (2003). Refactoring aspect-oriented programs. In *The 4th AOSD modeling with UML workshop. UML'2003*. ACM.

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

Kiczales, G., Lamping, J., Mendheka, A., Maeda, C., Lopes, C. V., & Loingtier, J.-M. (1997). Aspect-oriented programming. In *Proceedings of the european conference on object-oriented programming (ECOOP)*. In S. Gjessing & K. Nygaard (Eds.). *Lecture notes in computer science* (Vol. 1241). Finland: Springer.

Kösker, Y., Turhan, B., & Bener, A. B. (2009). An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications, 36*(6), 10000–10003.

Laddad, R. (2002). I want my AOP: Separate software concerns with aspect-oriented programming. URL <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>.

Lieberman, H. (1997). Autonomous interface agents. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 67–74). NY, USA, New York: ACM.

Maes, P. (1994). Agents that reduce work and information overload. *Communications of the ACM, 37*(7), 30–40.

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., Deursen, A. V., & Moonen, L. (2007). Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering Methodology, 17*(1), 1–37.

Marin, M., 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.

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* (pp. 1–5). New York, NY, USA: ACM.

Monteiro, M.P. (2004). Catalogue of refactorings for aspectj. Tech. Rep. UM-DI-GECSD-200402, Universidade do Minho.

Monteiro, M. P., & Fernandes, Jo a. M. (2008). An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms. *Software – Practice and Experience, 38*(4), 361–396.

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

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* (pp. 1–5). New York, NY, USA: ACM.

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., & Marcos, C. (2009a). Identificación automática de refactorings. In *Tenth Argentine symposium on software engineering (ASSE 2009)*, 38 JAIIO (Jornadas Argentinas de Informática).

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* (pp. 23–27) New York, NY, USA: ACM.

Vidal, S., & Marcos, C. (2009b). Un proceso iterativo para la refactorización de aspectos. *Revista Avances en Sistemas e Informática, 6*(1), 93–103.

Yuen, I., Robillard, M. P. (2007). Bridging the gap between aspect mining and refactoring. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution* (p. 1). New York, NY, USA: ACM.