http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (3): 573-576, 2006
ISSN 1812-5638

A Model-driven Approach to Aspect Mining

[1]Bounour Nora and [2]Ghoul Said
[1]Department of Computer Science, Laboratory of Computer Science,
University Badji Mokhtar of Annaba, B.P. 12 Annaba (23000), Algeria
[2]Institute of Computer Science, University of Philadelphia, Jordan

Abstract: Aspect mining aims to identify crosscutting concerns in legacy systems. The code implementing a crosscutting concern is often spread over many different parts of an application. Identifying such code automatically greatly improves the maintainability of the application and enables migration of exsisting (object-oriented) programs to aspect oriented ones. In this study, we present a model driven approach for aspect mining, which automatically identifies desirable candidates concerns, without requiring input from the user. The aspect miner acts as a model transformer converting the program model to a concern-oriented model. This model is more abstract and it is constructed by using a concern library. The concern library give concern descriptions in terms of plans.

Key words: Aspect oriented programming, aspect mining, crosscutting concern, program analysis, programming plan

INTRODUCTION

Programs are typically designed with a principal decomposition in mind, but some functionalities will arise which cannot be assigned to a single unit in such decomposition. This kind of functionalities are called crosscutting concerns because they involve more than one decomposition unit. Examples of crosscutting concerns are persistence, synchronization, exception handling, error management and logging. In fact, code treating this kind of functionalities is scattered through all the units participating in the concern. Crosscutting concern's code is also mixed and confused with the rest of the code in each unit. This problem is known as tangling.

A classic example of a situation with concerns that are tangled in a class is the programming of shoppingcart (Fig. 1). This code mixes business logic with logging concern. The Logging concern is a particular set of program points which should log some particular data.

Aspect oriented programming (AOP) captures crosscutting concerns by defining special classes called aspects (Kiczales, 1997), just as the crosscutting concerns that they capture. We illustrate in Fig. 2 the logging aspect corresponding to the code of shoppingCart. This aspect keeps the logging out of the business logic. It places a pointcut (Aspect *et al*., 2001) on every public call to any method contained in shoppingCart class. This allows the aspect to do all logging necessary when a call

```
import java.lang.reflect.*;
public class ShoppingCart { private List items = new
Vector();
public void addItem(Item item) {
System.out.println("Log:"+this.getClass().getName());
items.add(item);
}public void removeItem(Item item) {
System.out.println("Log:"+this.getClass().getName());
items.remove(item);
}public void empty() {
System.out.println("Log:"+this.getClass().getName());
items.clear();
}}
```

Fig. 1: Logging concern tangled in shoppingcart class

```
public aspect LoggingAspect {
pointcut loggedMethods(ShoppingCart
 shoppingcart):this(shoppingcart)
&& (execution(void ShoppingCart.*(..)));
before(ShoppingCart shoppingcart):
loggedMethods(shoppingcart) {
System.out.println("Log:"+shoppingcart.getClass().get
Name());
}}
```

Fig. 2: Logging aspect

Corresponding Author: Bounour Nora, Department of Computer Science, Laboratory of Computer Science, University Badji Mokhtar of Annaba, B.P. 12 Annaba (23000), Algeria

is made to the methods of this class. The logging is now done in the before advice.

Code scattering and code tangling are problems that affect applications in a systematic way. Aspect mining aims to identify crosscutting concerns in legacy systems. Automatic aspect mining would not only prevent legacy systems from accumulating scattered and duplicated code, but also enable migration of legacy systems to Aspect oriented programming.

In this study, we introduce an alternative approach to aspect mining, in which the code is treated as multiple inter-related concerns. The aspect mining process acts as a model transformer converting the program model to a more abstract one by using a concern library. The concern library give concern descriptions in terms of plans. Several works have shown the value of programming plan in program understanding (Soloway and Ehrlich, 1984). The term plan has been used in program understanding literature to denote two different things. In some literature (Soloway, Ehrlich, 1984), plans are viewed as program fragments which represent an abstract programming concept or a stereotypical action sequences. Other literature (Kozaczynski and Ning, 1994; Kozaczynski *et al.*, 1992) views them as units of knowledge necessary for identifying abstract programming concept. In this study, we have used plan to refer to a unit of knowledge required to identify a crosscutting concern.

BACKGROUND: ASPECT MINING APPROACHES

Aspect mining is described as a specialized reverse engineering process (Loughran and Rashid, 2002; Deursen *et al.*, 2003), which is to say that legacy systems (source code) are investigated (mined) in order to discover which parts of the system can be represented using aspects. This knowledge can be used for several goals, including reengineering, refactoring and program understanding.

The research in aspect mining has known different approaches (Bounour and Ghoul, 2005). It has start with tools that can be categorized as being exploratory. Such tools aid a developer in manually browsing the code while looking for crosscutting concerns. Examples of such approaches are Aspect Browser (Griswold *et al.*, 1999), Aspect mining tool (Hannemann and Kiczales, 2001), Feat (Robillard and Murphy, 2002) and Jquery (De Volder, 2002). Both require an input from the user and depend on the user's understanding of the software to be mined.

Another group of approaches aim to automatically detect aspects. A number of these are based on the idea of detecting code duplication: Tom Tourwé and Paolo Tonella identify aspectual views using Formal concept analysis (Tonella and Ceccato, 2004; Tourw´e and Mens, 2004; Mens and Tourw, 2005). Bruntink (2004) proposes the use of clone class metrics. Shepherd *et al.* (2004) perform PDG-based clone detection to identify candidate concerns.

Other approaches that aim for automation are based on the idea of detecting scattering: Marin *et al.* (2004) perform Fan-in analysis in mining candidates aspects. Other techniques uses dynamic information to identify candidate aspects (Breu and Krinke, 2003; Krinke and Breu, 2004).

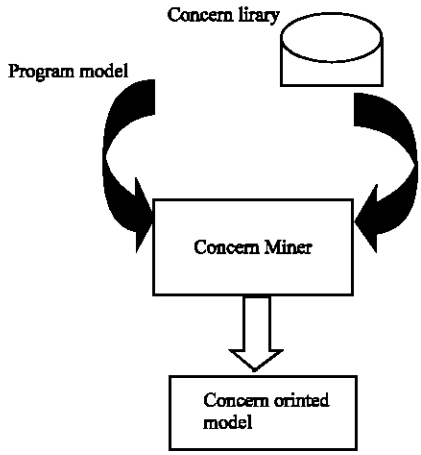

Fig. 3: Concern identification

A MODEL-DRIVEN APPROACH

Our approach perceives the program as a set of concerns interrelated by different kind of relationship. The purpose is to reconstruct the hierarchy of concerns implemented by the actual program. The aspect mining process aims to identify and locate the relevant crosscutting concern in the source code. We consider this process as driven by abstract model of concerns described in the concern library. The concern miner will analyse the source code (program model) and verify if the code implements models of concerns. The concern miner acts as a model transformer converting program model to more abstract one by using concern library. The mapping from the concrete occurrences of concerns in the program to the concern-oriented model can be perceived as model transformation (Fig. 3).

Program model: We use a program model which abstracts the code by eliding implementation details. And it augments code by making the dependencies between different elements explicit.

```
Plan Logging;
Precondition: ∃ log.statement.
Postcondition:
Data_flow (log.statement,*)
Control_flow (log.statement, *)
End.
```

Fig. 4: Programming plan of the logging concern

Formally, a program is represented as $G_p = \{V_p, E_p\}$ where V_p is a set of vertices and E_p are directed edges which represent interactions between classes. A vertex in V_p can be one of three types.

Class vertex (C): Represents a global class, considered without its members.
Field vertex (F): Represents a field member of a class.
Method vertex (M): Represents a method member of a class.

An edge in E_p can be one of six types, depending on the type of vertices it connects: (M,M), (M,F), (M,C), (C,C), (C,M) and (C,F). Edges are labeled with the semantic relationships they represent.

At each call site in the method, a call vertex is created for connecting the called method and there is an actual-in vertex for each actual parameter and an actual-out vertex for each actual parameter that may be modified by the called method. Each formal parameter vertex is control-dependent on the method entry vertex and each actual parameter vertex is control-dependent on the call vertex. Some special kinds of dependence arcs are created.

A call dependence arc represents call relationships between a call method and the called method and is created from the call site of a method to the entry vertex of the called method.

Concern library: Each crosscutting concern is described in the library via a plan. The plans stored in the library are used in identifying crosscutting concern. They are used as an inference rule. Their structure is divided into two parts: precondition and postcondition.

In general the precondition represents the key statements of the concern. For some concerns such as synchronization they are often dealt by using synchronized methods or synchronization statement (wait, join..). For other concerns, such as logging and state change updating they are implemented as a single method that is called from a wide spread number of places in the code. So, the Logging concern will exist if there exist a statement matching the part precondition of the plan (Fig. 4). The postcondition represents all statements participating to the implementation of the concern by control and data flow dependencies.

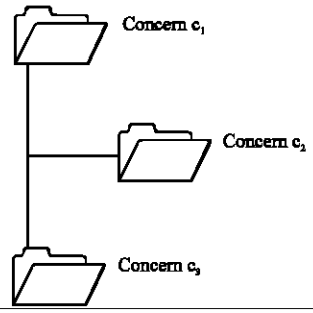

Fig. 5: Concern-oriented model

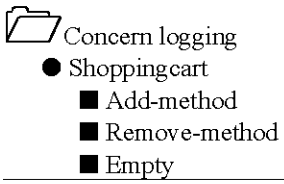

Fig. 6: Concern-oriented view

Concern-oriented model: When concerns are identified the concern miner gives an abstract model of the program. The program is described as an hierarchy of concerns identified, which we called concern-oriented model.

Formally the concern-oriented model is defined as a conceptual graph $C_p=\{C_i, E_j\}$ where Ci are concerns identified and Ej are relations between the concerns. We define two relations between concerns, which are *inside* and *after*. We say that a concern c_2 is inside a concern c_1 if c_2 is a part of the concern c_1. The concern c_3 is after c_1 if it is outside c_1 and after it (Fig. 5).

The model refinement is a direct manipulation technique designed to aid programmers in studying the details of a concern's implementation in the source code. Each concern has its own refinement view (Fig. 6). It is described via classes, methods and fields participating to its implementation. In this way, the programmer can control the gradual exposition of implementation details about concerns.

CONCLUSIONS

Many of aspect mining tools are semi automatic, which means they require some form of input by the developer. More advanced tools, which are able to identify aspects without human intervention, are based on static analysis (clone detection techniques, formal concept analysis, Fan in), or dynamic analysis. Although, these tools are limited. Full automation of aspect mining process remains a lofty goal.

In this paper, we have presented a model driven approach for aspect mining which detect aspect automatically. The approach is based on knowledge of crosscutting concerns, described by the concern library. Each concern is described via a plan. The mining process transform the program dependence model to a concern-oriented model.

We have developed a prototype of the model-driven mining tool for Java programs. The tool have been tested for short program Java which contains logging, trace and synchronization concerns. The first tests of the tool have encouraging results.

REFERENCES

Aspect, J. *et al.*, 2001. The Aspect Programming Guide.

Bounour, N. and S. Ghoul, 2005. A survey of aspect mining techniques. Congrès International en informatique appliquée-CIIA05-Bordj Bou Arreridj, Algérie, pp: 241-246.

Breu, S. and J. Krinke, 2003. Aspect mining using dynamic analysis. In GI-Software technik-Trends, Mitteilungen der Gesellschaft f'ur Informatik. Bad Honnef, Germany, 23: 21-22.

Bruntink, M., 2004. Aspect mining using clone class metrics. In 1st Workshop on Aspect Reverse Engineering.

Deursen, A.V., M. Marin and L. Moonen, 2003. Aspect mining and refactoring. In First Intl. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE).

De Volder, K., 2002. The jquery tool: A generic query-based code browser for eclipse. Presentation at Eclipse. BoF at OOPSLA.

Griswold, W.G., Y. Kato and J.J. Yuan, 1999. Aspect browser: Tool support for Managing Dispersed Aspects. Technical Report CS99-0640, Department of computer Science and Engineering, University of California, San Diego.

Hannemann, J. and G. Kiczales, 2001. Overcoming the prevalent decomposition in legacy code. In Proc. ICSE Workshop on Advanced Separation of Concerns, Toronto, Canada.

Kiczales, G., 1997. Aspect-oriented Programming. Proc. European Conf. on object-oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241.

Krinke, J. and S. Breu, 2004. Control-flow-graph-based aspect mining. In Workshop on Aspect Reverse Engineering.

Kozaczynski, W., J. Ning and Andre Engberts, 1992. Program concept recognition and transformation. IEEE Trans. Software Eng., 18: 1065-1074.

Kozaczynski, W. and J. Ning, 1994. Automated program understanding by concept recognition. Automated Software Engineering, No 1, pp: 61-78.

Loughran, N. and A. Rashid, 2002. Mining aspects. Workshop on early aspects: Aspect-oriented requirements engineering and Architecture Design. Enschede, The Netherlands.

Marin, M., A.V. Deursen and L. Moonen, 2004. Identifying aspects using fan-in analysis. In Proc. 11th Working Conf. Reverse Engineering, IEEE Computer Society.

Mens, K. and T. Tourwé, 2005. Delving source-code with formal concept analysis. Elsevier J. Computer Languages, Systems and Structures (In Press).

Robillard, M.P. and G.C. Murphy, 2002. Concern graphs: Finding and describing concerns using structural program dependencies. In ICSE.

Shepherd, D., E. Gibson and L. Pollock, 2004. Automated mining of desirable aspects. Technical Report 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716.

Soloway and K. Ehrlich, 1984. Empirical studies of programming knowledge. IEEE Trans. Software Eng. SE-10.5: 595-609.

Tonella, P. and M. Ceccato, 2004. Aspect mining through the formal concept analysis of execution traces. In Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society, pp: 112-121.

Tourw´e, T. and K. Mens, 2004. Mining aspectual views using formal concept analysis. In 4th IEEE Intl. Workshop on Source Code Analysis and Manipulation, pp: 97-106.