A NOTATIONAL DESIGN OF JOIN POINTS

Saqib Iqbal, Gary Allen
University of Huddersfield, Queensgate, Huddersfield HD1 3DH, UK

ABSTRACT

Join points are the composition points where aspects are composed with the base system. In other words they are target hooks in the base system where aspect's implementation is weaved in. Join points are not defined separately in aspect-oriented design rather a pointcut model is designed which consists of related join points. Grouping of related join points in a pointcut depends on designer's intuition and corresponding aspect's nature which makes some of the join points overlooked or not properly grouped with the related join points. This paper proposes a solution to this problem by providing design notations for representing join points in design. This kind of design of join points help designers design join points properly and group the related join points in one pointcut.

Keywords aspect-oriented programming, join point, pointcut, advice, aspect

1 INTRODUCTION

Aspect-oriented programming [1] was introduced to provide separation of concerns [3] by modularizing crosscutting concerns. Crosscutting concerns are the concerns which have their implementation scattered and tangled in the system code. These crosscutting concerns are implemented as separate modules and are called aspects. Aspects are implemented as specific points in the base program which are called join points. A join point could be the execution or calling of a method or exception, access of a local variable, initialization of objects or a specified piece of code in the system.

A lot of research is being carried out on the earlier stages of aspect-oriented software development (AOSD) like on the requirement engineering [2] [5] [6] of aspects and design [4] [7] of aspects. The purpose of research on the earlier phases of AOSD is to bridge the gap between implementation and requirement engineering and design of aspects so that the aspect-oriented system could become more traceable, extensible, maintainable and reusable. Many approaches have been suggested for join point modelling [5] [8] [10] which propose different ways of representing join points. The problem with all of them is that they don't consider join points as a distinct designing element rather they view it as a constituent of a pointcut.

This paper provides a new way of designing join points. This approach views join points as interface-like entities which provide a medium for aspects to interact with the classes of the system. Two join point models have been proposed, static join point model and behavioural join point model. Static join point model helps in representing the definition of join points along with the relationship between interacting aspect and system's class. Behavioural join point model helps in showing the exact location of join points during the execution of the system. In the rest of the paper, section 2 provides an introduction of aspect-oriented programming, section 3 describes proposed join point models and section 4 concludes the paper.
.

2 ASPECT-ORIENTED PROGRAMMING

Complex, large and distributed systems are not easy to develop. Complexity of such systems lies on the intermingled and interdependent nature of primary modules of the system. Dependency of these modules on other modules must be minimal. Such separate modules are represented differently in different languages, for example, as modules in Standard ML [9] and as classes in object-oriented languages, such as Java [11]. Implementation of these modules is strived to be done individually in such languages but total-separation has never been achieved. For instance, in Java, classes are individual distinct entities of the program but there are almost always functional requirements which cannot be implemented within one individual class's scope. Such business logic is dependent on the functional implementation of multiple classes. Such nature of implementation of system requirements

can result in scattered code through the system. To solve this problem, rather than decomposing the system during implementation, a designer should separate all the concerns as individual modules.

The term “separation of concerns” [3] is not a new principle in software development. It suggests that complex systems must be divided into sub-systems, usually called modules, to make them better to implement and understand. Modules can be developed separately and then be joined together to form the whole system. This property enhances the system’s maintainability, extensibility and traceability.

The object-oriented software development paradigm may be considered to be based upon this principle. Each concern or functionality of a system is encapsulated in objects and they only perform their own specified functions without accessing other object’s private elements. This is an example of “separation of concerns”. Though this property is achieved in object-oriented programming (OOP), over time developers started to feel that OOP does not separate all the concerns in from of classes [1]. There are some concerns such as tracing, logging, fault-tolerance, synchronization, to name a few, which cannot be implemented in distinct classes, rather their code is present in more than one class, usually referred to as the scattered code problem, and such concerns are called crosscutting concerns. To separate such crosscutting concerns from being scattered, new separation or modularization techniques are required. Two different methods have been used by software developers to handle crosscutting concerns: Interception based approaches, in which event or subroutines to implement crosscutting concerns are intercepted in the execution, and Code Weaving approaches, in which code is woven into the base code with the help of a tool. Based on the later solution, many programming paradigms have been suggested, such as aspect-oriented programming [1], adaptive programming [12], role-modeling [13], composition filters [14] and subject-oriented programming [15]. These approaches decompose each concern in an individual way.

This paper will concentrate on aspect-oriented programming. In aspect-oriented programming, each concern, which has scattered code, is implemented in an individual module, called an aspect. Aspect-oriented technology provides efficient modularity and serves its purpose as an approach to implement separation of concerns. A popular implementation of aspect-oriented techniques is an extension to the Java language, called AspectJ [9], which provides support to represent aspects and their constructs in the program and also provides interfaces to join aspects with the base program. Aspects comprise of an advice, a piece of code to implement the aspect’s logic, and pointcuts, which are packages of related join point. Join points are the points in the base program where an advice has to be woven into the program.

3 JOIN POINT MODEL

Join points are well-defined points in the code of the system where aspect’s code can be executed. Some of the join points in AspectJ are calling and execution of methods or constructor, object initialization, field’s setters and getters and exception handlers. At these defined points, aspect’s advices are called to be executed. Join points are represented in groups in aspect’s body of code. These groups are called pointcuts. Join points are required to be identified carefully during the design process because aspect’s implementation and its interaction with system entirely depend on their location.

This paper proposes a design technique to represent and design joins points. This technique provides a notational and diagrammatical approach for representing join points. A static join point model has been proposed to represent joining of classes with aspects and a behavioural join point model has been proposed to show the exact location of weaving of aspects in the system during its execution.

3.1 Static Join point model

Join points are interfaces that join aspects with classes so their design is supposed to show this relationship. Join points have a definition as well which includes the reference of the exact location of join points.

Static Join Point Model provides a notational way of representing join point definition and the interacting classes and aspects. Figure 1 shows an example of a join point in an ATM system. In this example an aspect *Logging* logs data after the execution of a withdraw cash transaction. The

definition of join points**, after (execution (*.withdraw (*))),** shows that aspect Logging's implementation executes after the execution of a withdraw function. Aspect *Logging* and interacting class Account are also represented to show the relationship between them.

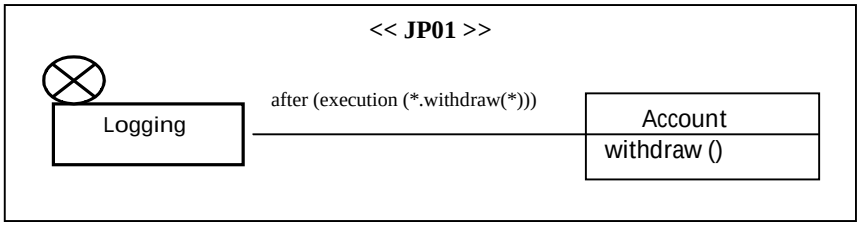

Figure 1: Static Join Point Model

3.2 Behavioural Join Point Model

As Join points are not physically represented in the code of the system so we have to show their actual location while we are designing them. Behavioural Join Point Model provides a diagram which shows the activity of a process in which an aspect's implementation is weaved in at a specified join point.

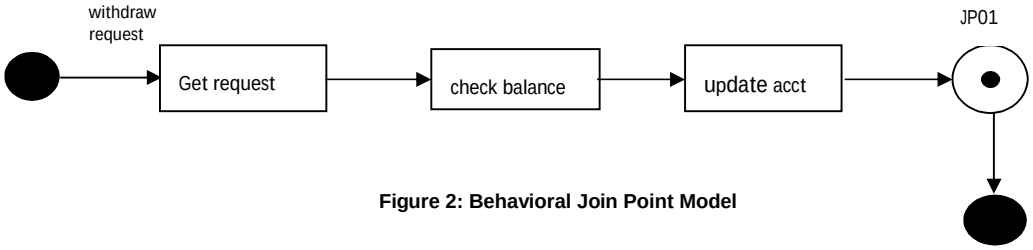

Figure 2: Behavioral Join Point Model

Example of a behavioural join point model of a withdrawal process of an ATM system is shown in figure 2. It shows that after the *update acct* task, a join point JP01 is reached where its corresponding aspect, *Logging* in this case, will execute its implementation. This kind of design representation of join points allows designers to identify exact and accurate join points and it also helps in understanding the interaction of aspect during the execution of the system.

4 CONCLUSIONS AND FUTURE WORK

Join points emulate nature of interfaces. Just like interfaces, they provide a communication bridge between aspects and base classes. Data is passed from classes to aspects in the form of parameters and aspects process their advices to perform certain tasks in return. Join point's design, that's why, is as important as design of aspects and its elements. Join point models described above meet this need by providing a static model for representing major entities involved in an aspect-class relationship and the definition of join points. On the other hand, behavioural model is responsible for identifying the exact location during the execution of system where aspects' implementation is weaved-in.

Further research work is required for refining static join point models so that compound join points could also be represented using the same model. Compound join points are group of join points with logical conditions like AND, OR, NOT, etc.

REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, Ch.V., Loingtier, J.-M., Irwin, J., Aspect-Oriented Programming, in: Proceedings of ECOOP 1997, Jyväskylä, Finland, June 9-13, 1997, pp. 220-242, in: Lecture Notes in Computer Science, vol. 1241, Springer, 1997

[2] Baniassad, E., Clarke, S.: Theme: An Approach for Aspect-Oriented Analysis and Design. In: Proceedings of the 26th International Conference on Software Engineering (ICSE). Edinburgh, Scotland (2004) 158-167

[3] Dijkstra, E.: A discipline of programming, Prentice Hall (1976)

[4] Baniassad, E., Clarke, S.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (2005)

[5] Rashid, A., Moreira, A., Araújo, J.: Modularisation and composition of aspectual requirements. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Design (AOSD). ACM Press (2003) 11-20

[6] Rashid, A., Sawyer, P., A., Moreira, A., Araújo, J.: Early aspects: A model for aspect-oriented requirements engineering. In RE 2002 (2002) 199-202

[7] Clarke, S., Walker, R. J.: Generic Aspect-Oriented Design with Theme/UML. In: Filman, R. E., Elrad, T., Clarke, S., Aksit, M. (eds): Aspect-Oriented Software Development. Addison-0Wesley (2004)

[8] AspectWerkz Home Page: http://aspctwerkz.codehaus.org

[9] M. Blume and A. W. Appel. Hierarchical modularity. ACM Transactions on Programming Languages and Systems, 21(4):813–847, 1999.

[10] Noorazean Mohd Ali and Awais Rashid. A State-based Join Point Model for AOP. In Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role (VAR'05), in 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, Scotland, July 2005.

[11] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java. In ACM conference on Object-Oriented Programming, Systems, Languages and Applications, volume 34 of ACM Sigplan Notices, pages 132–146, Denver, CO, Aug. 1999. ACM Press.

[12] Lieberherr, K.J., Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, 1996

[13] Reenskaug, T., Wold, P., Lehne, O.A., Working with Objects: The OORam Software Engineering Method, Manning/Prentice Hall, Upper Saddle River, New Jersey, 1995

[14] Aksit, M., Bergmans, L., Vural, S.: An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach, in Proceedings of the ECOOP 1992, Utrecht, The Netherlands, June 29 - July 3, 1992, pp. 372-395, in: Lecture Notes in Computer Science,vol. 615, Springer, 1992

[15]] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code," presented at Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999), Denver, Colorado, USA, 1999.