Singaporean Journal Scientific Research (SJSR)
ISSN: 2231 - 0061Vol.3, No.2 pp.155 - 157

available at: : http://www.iaaet.org/sjsr

AOP based Refactoring of Java Legacy System

E. Kodhai[1], R. Ragha Sudha[2], K. Reka[3]and A. Amudha[4]
[1] *Sri Manakula Vinayagar Engineering College/Information Technology, Puducherry, India*
Email: kodhaiej@gmail.com
[2,3,4]*Sri ManakulaVinayagar Engineering College/Information Technology, Puducherry, India*
Email: raghasudha_r@yahoo.in, nila_selvi@yahoo.co.in, amudha3190@yahoo.com

***Abstract*—Maintenance and refactoring of legacy System is difficult due to lack of necessary documents and source codes. It is a great deal to generate valuable information through refactoring. Our Objective is to develop a class diagram and sequence diagram from the binary byte code of a java legacy system and then to obtain coding from the developed diagrams. To trace the system behaviors, the pattern of Aspectj concept namely weaving is applied to resolve the binary byte codes during runtime of the system.**

Index Terms—refactoring, binary byte code, class diagram, sequence diagram, Aspect

1. INTRODUCTION

The software development cyclealways passes through requirement analysis, design, coding, testing and runtime maintenance. In fact, however, it needs many efforts so that under the pressure of capital and time, the developers only focus on coding (no proper comments) and neglect the indispensable documents. When time passed and developer alternated, many valuable documents and source codes are lost while only the binary executable files left. Although these binary files can implement the predefined function, they cannot be updated with any modification or extension for no documents and source codes support. Then this software evolves into a "legacy system" and becomes more and more unmaintainable and it will be left unused.

For a legacy system of Java with source codes, the current mature IDEs (Integrated Development Environment), such as Eclipse Modeling Framework [4] and NetBeans UML Modeling [5], only provide the capability of reverse engineering from source codes to documents or UML diagrams. But for a legacy system with only binary executable files, they can not provide program design information without source code help. There are some Java decompilers which can find the original source code to a degree. But they also have some constraints and cannot get the precise program structure. We utilize the techniques of reflection and decompilation to generate the class diagrams from binary bytecodes. Moreover, through weaving the stub codes into original binary files to track the runtime behaviors, we gracefully get the precise sequence diagrams and class diagrams based on Aspectj concept. The class diagram and sequence diagram of legacy system can be generated automatically and very crucial for system refactoring.

2. ASPECT-ORIENTED PROGRAMMING

Aspect-oriented programming (AOP) is an approach to programming that allows global properties of a program to determine how it is compiled into an executable program. AOP can be used with object-oriented programming. Object oriented programming has become mainstream over the last years, having almost completely replaced the procedural approach. One of the biggest advantages of object orientation is that a software system can be seen as being built of a collection of discrete classes. Each of these classes has a well defined task; its responsibilities are clearly defined. In an OO application, those classes collaborate to achieve the application's overall goal. However, there are parts of a system that cannot be viewed as being the responsibility of only one class, they cross-cut the complete system and affect parts of many classes. Examples might be locking in a distributed application, exception handling, or logging method calls. Of course, the code that handles these parts can be added to each class separately, but that would violate the principle that each class has well-defined responsibilities. This is where AOP comes into play: AOP defines a new program construct, called an *aspect*, which is used to capture cross-cutting aspects of a software system in separate program entities. The application classes keep their well-defined responsibilities. Additionally, each aspect captures cross-cutting behavior.

The components of AOP are advices/interceptors, introductions, metadata, and pointcuts. An advice / interceptor is an object that intercept the invocation of a method before its execution. Interceptor embodies the behavior to add or remove or replace the functionality of infrastructure. Perfect pluggability without changes need for business logic is provided by Interceptor. An introduction adds states and functionality to the existing objects. Pointcuts allows defining the location of interceptors and introductions are to applied. Metadata provide information about class and runtime system hints for treating the classes. For example, AOP logging, monitoring of the code without instrumenting with the code. For example, a method named Deposit receives the arguments of Account NO and Amount, then does the withdraw service. Before calling Deposit, a necessary identity checking must be executed automatically. Additionally, to audit this transaction, a whole logging operations must be executed around (before and after) Deposit service. Thus, any call of Deposit method is worthy concerned, and this evolves into a joinpoint. A pointcut is a joinpoint container whose joinpoints have the same or similar features. A pointcut is defined as follows:

PointcutDepositPoint(String AccountNO, double Amount):Call(public void Deposit(..));
The locations of each call to the method Deposit are named as DepositPoint.

An advice is weaving the separated code with the predefined pointcut to generate a composite service. The weave type has three kinds: before, after and around. The following is a before advice.

Before(String AccountNO, double Amount):
DepositPoint(AccountNO, Amount){
……..//Write any code you need
. ……..// These codes will be weaved into the entrance of Deposit and be executed before the body of method. These can also access current program context during runtime execution.
}

For the after weave type, the separated codes is executed after the pointcut method finished. As to another around weave type, its function is equal to before type plus after type.

The whole trigger procedure of AOP is that: When program execution reaches a predefined method or variable, if it gets into a joinpoint, the main thread will be suspended and sequently invoke the ownership pointcut. According to the definition of pointcut, the advice code will be executed with current program context. After the advice code finished, the main thread will resume and continue execution.

4. RELATED WORK

Object-oriented legacy system behavior is distributed over many interacting objects, making it necessary to test for complex collaboration scenarios. The existing system [2] shows how to use the execution traces as a basis for expressing tests. E.g.: Query library using SOUL, a logic engine implemented in Smalltalk. The architecture of TESTLOG consists of 5 top-down layers for SOUL and one layer below Soul is for Smalltalk. The bottom layer comprises an object-oriented model that represents the execution trace. Each trace is stored as an object in the Smalltalk image. At the next abstraction level TESTLOG provides queries to access single events and states.

Data flow analysis was originally used as a technique for code optimization in compilers. It [3, 8] has also been shown to be a useful technique in other areas, such as performance tuning, testing, and debugging. This study describes the fundamentals of data flow analysis, and specifically dynamic data flow analysis. The study concludes with a number of requirements for new testing approaches using dynamic data flow analysis. Dynamic data flow analysis is a method for analyzing the sequence of actions on data in a program as it is being run. Huang introduced tracing the data flow anomalies through state transitions instead of sequences of actions. When an action is applied on a variable, its state follows transitions according to the state transition diagram.

AOP is used for instrumenting the system and for gathering the data. This approach [6] works and is conceptually very clean, but comes with a major quid pro quo: integration of AOP tools with the build system proves an important issue. This leads to the question of how to reconcile the notion of modular reasoning within traditional build systems with a programming paradigm which breaks this notion.

Another approach [7] that relates on a first attempt to see if aspect-oriented programming (AOP) and logic meta-programming (LMP) can help with the revitalization of legacy business software. By means of four realistic case studies covering reverse engineering, restructuring and integration, it discuss the applicability of the aspect-oriented paradigm in the context of two major programming languages for legacy environments: Cobol and C.

5. PROPOSED SYSTEM

We proposed a general approach to get class diagram and runtime calling sequence diagram for legacy system of Java without any support of source codes. Through Java Reflection, we can easily get some basic information of classes, including class name, member variables, member functions/methods with parameter signature, super class/interface name. This information can be used to rebuild the class diagram, but it is not enough to get out the method calling sequence diagram, because Java Reflection cannot find out the detailed information hided in the internal body of method. Through decompiler tools, we can get the readable bytecodeinstructions.From the readable bytecode, one can find the four bytecode instructions: invokevirtual, invokespecial, invokestatic and invokeinterface. With the help these four bytecode we can analyze the system easily. The above techniques are grouped together and theapproach is listed as follows.

A . Java Reflection:

By using Java Reflection, one can load the binary bytecode and reflect the peripheral information of Java class, including interface, super class, class modifiers, constructors, methods, method signatures.

B . Java Decompiler:

Using Java decompiler tools to resolve the binary bytecode, the preliminary calling sequence diagram can be derived, whose most methods are affiliated to interface or abstract classes, not concrete classes.

C . Joinpoint:

If a method of derived class overwrites the same method of super class, set a joinpoint on any call to the method overwritten. Then the joinpoint can be encapsulated as a pointcut.

D . Pointcut:

For each pointcut, setup a new “before” advice to trace the real runtime information of Java objects. The code of joinpoint, pointcut and advice can be written into a single aspect file.

E . Aspect Weaver:

Weave the aspect file into the binary class bytecode during legacy system runtime and trace the real calling sequence.Finally analyze to the trace log, adapt the preliminary result of Java Decompiler and generate the calling sequence diagram automatically. Through bytecode

analysis, one can get the class diagram and sequence diagram of the source code.

The system architecture for our proposed system is given below which shows how we are able to generate class diagrams and sequence diagrams from the binary byte code.

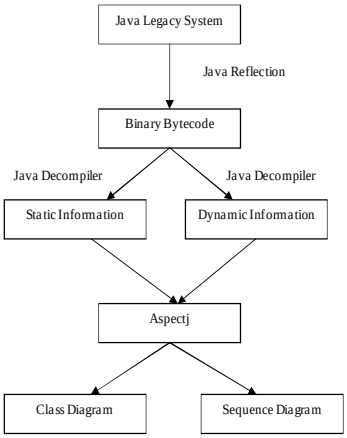

Figure 1. System Architecture

This will ensure the software maintenance in a better manner. If we arrived at class diagrams and sequence diagrams then with that help we can go for design, analysis, source code, implementation and finally maintenance.

5. CONCLUSION

Obtaining valuable information from the legacy system by refactoring it and without any source code support seems very difficult in reverse engineering. In this paper, based on the techniques of bytecode analysis and aspect-oriented programming, the class diagram and sequence diagram are automatically generated from Java legacy system. And with the help of those diagrams, the source code of a java legacy system is generated. This approach is useful in software maintenance, system reengineering and refactoring.

REFERENCES

[1] Liangyu Chen, Jianlin Wang, Ming Xu, ZhenbingZeng,"Reengineering of java legacy based on aspect – oriented programming", in Second International Workshop onEducation Technology and Computer Science, 2010.Besselfunctions,Phil. Trans. Roy. Soc. London, vol. A247, pp.529–551, April 1955.

[2] S.Ducasse, T.Girba, R.Wuyts, "Object-oriented legacy system trace based logic testing", in Proceedings of the Conference on Software Maintenance and Reengineering (CSMR2006), 2006.

[3] A.Cain, J. Schneider, D.Grant and T.Chen, "Runtime Data Analysis for Java Programs", Proceedings of 1st workshop on advancing the state of-the-art in runtime-inspection (ECOOP2003), July, 2003.

[4] Eclipse Project, http://www.eclipse.org/.

[5] NetBeans Project, http://www.netbeans.org/.

[6] B.Adams, K.Schutter, A.Zaidman, S.Demeyer, H.Tromp and W.Meuter, "Using aspect orientation in legacy environments for reverse engineering using dynamic analysis"—An industrial experience report, The Journal of Systems and Software", 82:668-684, 2009.

[7] K.Schutter, B.Adams, "Aspect-orientation for revitalising legacy business software". Electronic Notes in Theoretical Computer Science 166 (1),63-80,2007.

[8] T. Systa, "Static and dynamic reverse engineering techniques for Java software systems", Ph.D. Thesis, University of Tampere, Finland, 2000.