2010 Second International Workshop on Education Technology and Computer Science

Reengineering of Java Legacy System Based on Aspect-Oriented Programming

Liangyu Chen, Jianlin Wang, Ming Xu, Zhenbing Zeng

Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute
East China Normal University, Shanghai, China
zbzeng@sei.ecnu.edu.cn

***Abstract*—Legacy System is difficult to be maintained and refactored for lack of necessary documents and source codes. How to generate some valuable information from system runtime behaviors is a big challenge to systems reengineering. In this paper, we attempt to reconstruct class diagram and sequence diagram from the binary bytecode of Java program by reflection and bytecode decompilation. The pattern of Aspect-Oriented Programming is applied to resolve the intricate codes of Interface and Dependency Injection pattern through weaving aspect codes into binary bytecodes during runtime to trace the system behaviors. The experiments show our approach can exactly generate the class diagram and sequence diagram from legacy Java System.**

Keywords-component; Java Legacy System; Aspect-Oriented Programing; Runtime Behavior; Reengineering

I. INTRODUCTION

In the traditional normal process of software development, it always passes through requirement analysis, design, coding, testing and runtime maintenance in order. According to CMM standards, each step must have enough documents to completely explain how to develop the software. In fact, however, it needs many efforts so that under the pressure of capital and time, the developers only focus on coding (with poor comments) and neglect the indispensable documents. Along with 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 [1-3]. Finally this software is put to "death" while another new software is to be proposed and may repeat the same cycle.

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 some helpful 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 [6-8].

In this paper, we consider a kind of Java legacy systems who only have binary executable file, no any source code and document and use Java design patterns, such as Interface and Dependency Injection pattern [9]. 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 based on Aspect-Oriented Programming (AOP) [10]. The class diagram and sequence diagram of legacy system can be generated automatically and very crucial for system reengineering.

The paper is organized as follows. Section 2 reviews the interface and Dependency Injection pattern of Java and analyses their influence; Section 3 introduces the principles of Aspect-Oriented Programming. Section 4 demonstrates the generation of class diagram and sequence diagram by tracing the runtime behaviors of system based on AOP. Section 5 is the conclusion.

II. INTERFACE AND DEPENDENCY INJECTION PATTERN

A common class aggregation is showed in Fig.1. The Button class has a member variable Lamp, thus Button is high coupling with Lamp, which is deemed as a bad smell in program design paradigm.

```
public class Button {
        /**
         * Button is high coupling with Lamp.
         * This is a bad design.
         */
        /* We skip the default getter and setter method */
   private Lamp lamp;
   public void push(){
      lamp = new Lamp(); //Initialization
      lamp.turnOn();
   }
}
```

Fig.1. Button is high coupling with Lamp

One can adopt the interface pattern that Button only relies on the interface SwitchableDevice instead of any concrete class [9], showed in Fig.2.

```
public interface SwitchableDevice{
        Public void turnOn();
}
public class Button {
        /**
         * Button is only coupling with the interface SwitchableDevice.
```

This work is supported by National Natural Science Foundation of China (No. 90718041), Shanghai Leading Academic Discipline Project (No. B412)

978-0-7695-3987-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ETCS.2010.298

```
        * The lamp can be instantiated by any class who implements SwitchableDevice.
        * The instantiation is done in the outside container, such as Spring.
        */
        /* For brevity, we skip the getter and setter method */
        private SwitchableDevice lamp;
        public void push(){
      lamp.turnOn();
    }
}
```

Fig.2. The coupling is removed by interface and DI.

A remarkable feature of code in Fig.2 is the member variable lamp doesn't be initialized before use, which seems to violate the base rules of Java. Actually, the latent initialization before use, is be executed by the outside container, such as Spring. This external initialization is also called Dependency Injection (DI), another crucial design pattern [9]. The following configuration of Spring container defines the concrete class of lamp [11].

```
<bean id=" lampAlias" class="cn.com.test.Lamp" />
<bean id=" Button " class="cn.com.test.Button ">
<property name=" lamp"><ref bean=" lampAlias" />
  </property></bean>
```

From the configuration file, it is easily known that the real class of lamp is Lamp. Importantly, the style of configuration file is changeful and dependable on the outside container. Thus, it seems infeasible to get the dynamic information through parsing the configuration files.

The patterns of Interface and Dependency Injection alleviate the high coupling of program components and provide more flexibility for system design and coding. However, they also impose many obstacles in reverse engineering analysis and verification. It seems very difficult to get precise information by the techniques of program static analysis. This makes more difficult for legacy system, which may lose source codes.

III. ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Programming (AOP) is a new emerging design pattern in software programming, whose concern is the separation of business modules [10,12,1]. Traditional Object-Oriented programming (OOP) emphasizes on the design of individual object and the inheritance relation between objects. This OOP is dominating in current software development process; however, it always neglects crosscutting business logic requirements. For example, in OOP, if the security is obligatory before operation, then it must add some security code about identity checking in entrance of every critical method of business logic. This has three defects. Firstly, the repeated security codes are distributed in many files without uniformly management. Secondly, this makes the method of business logic "not pure" and "swelling", whose violates the principle of "high cohesion" in software engineering. Thirdly, most importantly, the security code is badly coupling with business logic. If it need remove or renew the confidential mechanism, one must modify many source files and compile them, which is seriously ungraceful in software maintenance. AOP provides a graceful way to eliminate this "bad swell". The coupling operation can be separated into several basic, independent and unattached modules, which can be composed by common configuration files to satisfy the business requirements. If the requirements change, the only thing is to adapt the configuration file to recompose the basic modules into a new service for latest requirements, which doesn't change any source code and cause a new compilation. AOP doesn't take the place of OOP, but a powerful supplement. In software development, OOP focuses on object and object relation, while AOP emphasizes service dynamic composition and separation.

AOP has three basic components: joinpoint, pointcut, advice. A joinpoint can be a separated point or concerned point of business logic. During program runtime, a joinpoint is a reachable location of program execution sequences, such as accessing a special method or variable. For example, a method named WithdrawMoney receives the arguments of card NO and money number, then does the withdraw service. Before calling WithdrawMoney, a necessary identity checking must be executed automatically. Additionally, to audit this transaction, a whole logging operations must be executed around (before and after) WithdrawMoney service. Thus, any call of WithdrawMoney 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:

```
Pointcut withdrawPoint(String cardNO, double money) :
  Call(public void WithdrawMoney(..));
```

The locations of each call to the method WithdrawMoney are named as withdrawPoint.

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 cardNO, double money): withdrawPoint(cardNO, money){
  ……..//Write any code you need.
  ……..//These codes will be weaved into the entrance of WithdrawMoney 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.

AOP has several implemented libraries for different programming languages. The AspectJ is provided by Eclipse.org [4] and developed for Java web project. In AspectJ, an aspect is a big container of joinpoints, pointcuts and advices. An aspect can be written into an individual file or embedded into Java source codes. The aspect can be compiled statically into the related class bytecodes or weaved dynamically during the runtime execution. The weave in compilation time would generate coupling in class bytecode level despite the business logic has been separated in source level. Aspect loaded during

runtime execution seems be better than the approaches in [7, 8] since it can eliminate the coupling smell in both source and bytecode level completely.

IV. Dynamic Object Information and Calling Sequence Diagram

In this paper, we only consider the J2EE legacy system, which has no any source code and document and attempt to reconstruct the class diagram and sequence diagram of method calling, which are most important to system analysis and refactoring. More complexly, the system may use the design pattern of Interface and Dependency Injection [9]. Therefore it is infeasible to do reverse engineering by popular IDEs who rely on static analysis of source codes. The only way is to analyze the Java Class bytecodes and track the dynamic behaviors of program runtime execution.

For a J2EE system, its basic components are general Java classes, including interfaces, abstract classes and common classes. Each class is compiled into binary bytecode, which is an immediate language with extension name .class and can be loaded, interpreted and executed by Java virtual machine. 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. The information can be used to rebuild the class diagram, but 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, such as Javap.exe or ASM [6], we can get the readable bytecode from original binary class file. From the readable bytecode, one can find the four bytecode instructions: invokevirtual, invokespecial, invokestatic and invokeinterface. For example, there is a method callFoo in class B as follows:

```
Public void callFoo(A a){
        a.foo();
}
```

Decompile the binary bytecode, the readable bytecode is:

```
Public void callFoo(A)
  Code:
    0: aload_1
    1: invokevirtual #17; //Method A.foo:()V
    4: return
```

From the above readable bytecode, it can easily get the structure sequence: B.callFoo() calling A.foo(). But for the example showed in Fig.2, the real class type of lamp is determined at program runtime execution. If we use the bytecode analysis, we only get the sequence: Button.push() calling SwitchableDevice.turnOn(), however, we need the real runtime execution sequence: Button.push() calling Lamp.turnOn().

Fortunately, it is remarkable that any concrete class of lamp must implement the common interface SwitchableDevice and rewrite the method turnOn(). Therefore, if it defines a joinpoint as any call of SwitchableDevice.turnOn(), the real class can be tracked within the code of pointcut and advice as follows.

```
pointcut callTrack(Object caller, Object callee):
    call(*SwitchableDevice.turnOn() && this(caller) && target(callee);
before(Object caller, Object callee):callTrack(caller, callee){
    System.out.println("caller class:" + caller.toString());
    System.out.println("callee class:" + callee.toString());
    System.out.println("signature:" + thisJoinPoint.getSignature());}
```

We sum up the above analyzing techniques, and present a general approach to get class diagram and runtime calling sequence diagram for legacy system of J2EE without any support of source codes. The approach is listed as follows.

Step 1. 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.

Step 2. By 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.

Step 3. 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.

Step 4. 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.

Step 5. Weave the aspect file into the binary class bytecode during legacy system runtime and trace the real calling sequence.

Step 6. Analyse to the trace log, adapt the preliminary result of Step 2 and generate the calling sequence diagram automatically.

We have implemented the above approach into a prototype system and tested all source code of the book "Spring in Action" 2e written by Craig Walls and Ryan Breidenbach, published by Manning Press 2007. A typical example Knight and HolyGrailQuest in the first chapter can show our approach right and effective. Through bytecode analysis, one can get the class diagram showed in Fig.3 and sequence diagram showed in Fig.4.

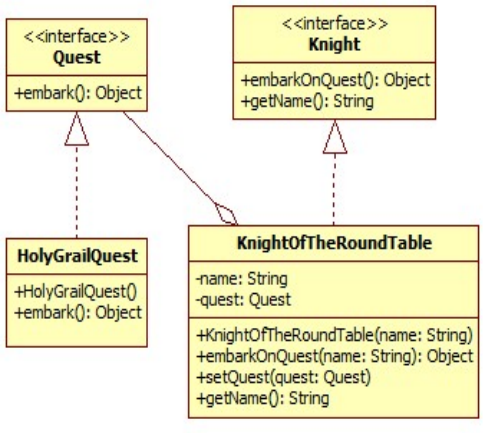

Fig.3. The class diagram

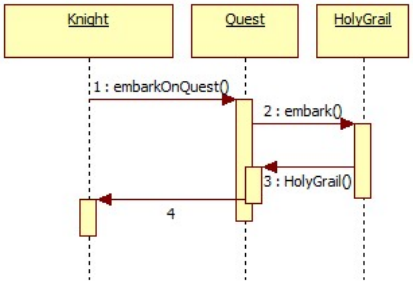

Fig.4. Sequence diagram from bytecode analysis

This calling sequence in Fig.4 doesn't give out enough information of calling sequence in runtime because Knight and Quest are interfaces and their methods are all empty. If we set two joinpoints at any call to the interface method Knight.embarkOnQuest() and Quest.embark(), we can use AOP techniques to get the runtime calling sequence diagram, showed in Fig.5.

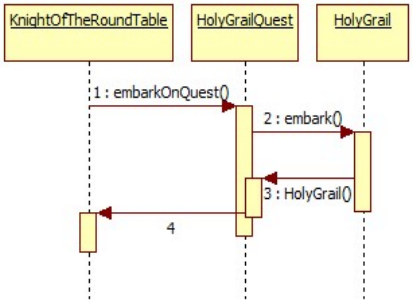

Fig.5. Runtime sequence diagram

V. CONCLUSION

Refactoring from legacy system to get valuable information without any support of source codes and documents 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 legacy Java system. We believe our approach is useful in software maintenance and system reengineering.

REFERENCES

[1] 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.

[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] K. Bennett, "Legacy systems: coping with success", IEEE Software 12(1),19-23,1995.

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

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

[6] ASM, http://asm.ow2.org/.

[7] 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.

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

[9] G.Erich, R.Helm, R.Johnson and J.Vlissides, "Design patterns: elements of reusable object-oriented software", Addison-Wesley, 1995.

[10] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. In Proc. of ECOOP 1997, vol.1241 of LNCS, pp.220-242, June, 1997.

[11] Spring, http://www.springsource.org/.

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