ISSN No. 0976-5697

Volume 2, No. 5, Sept-Oct 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Generalized Representation of Advice Using Sequence Trace Diagram

Shruti Dubey*
School of Computer Science and IT
DAVV, Indore, India
shruti.rose09@gmail.com

Neha Rahatekar
School of Computer Science and IT
DAVV, Indore, India
neha85944@gmail.com

Ugrasen Suman
School of Computer Science and IT
DAVV, Indore, India
ugrasen123@yahoo.com

Abstract: Aspect-oriented programming is built on the concept of separating concerns. While separation of concerns reduces textual scattering and tangling by encapsulating concerns within a localized module, the behavior of an aspect-oriented program becomes scattered. Capturing the sequential behavior of an aspect-oriented program is essential for the validation of the program's run-time semantics. With AspectJ, a suitable aspect-oriented programming language is at hand, no feasible systematic pictorial notation is available that supports the design of AspectJ programs. In this paper, a generalized design representation for AspectJ programs is presented in the form of sequence diagram. It provides representations for language construct called advice, of different types in AspectJ and specifies its runtime execution.

Keywords: AOP; Aspect; advice; circular relationship; sequence trace diagram.

I. INTRODUCTION

Aspect-oriented programming (AOP) is a new software development paradigm that aims to increase comprehensibility, adaptability, and reusability by introducing a modular unit, called "aspect", for the specification of crosscutting concerns [1] [2]. AspectJ is a programming language that supports the aspect oriented programming paradigm by providing new language constructs to implement crosscutting code [1]. Cross-cutting concerns are the functionality of a program which affects other concerns. These concerns often cannot be clearly distinguished from the rest of the system in both the design and implementation, and can result in either scattering (code duplication), tangling (significant dependencies between systems), or both [6] [7].

Aspect-oriented programming is accomplished by implementing a series of primary concerns in a specified language. These crosscutting concerns are added to the system through an aspect-oriented language. The primary concern and crosscutting concern are weaved using weaver. AspectJ is an implementation of aspect-oriented programming for Java [10]. It adds several program elements to Java that defines modular units of crosscutting code. AspectJ provides the concept of join points and pointcuts to enable dynamic crosscutting of program behavior [1]. A *join point* is a well-defined location within the primary code, where a concern will crosscut the application. Join points can be method calls, constructor invocations, exception handlers, or other points in the execution of a program. *Pointcuts* are sets of join points.

Pointcuts are used to specify at which join points crosscutting behavior is to be executed. Pointcuts are defined in terms of pointcut designators. Some of those pointcut designators (such as call, execution, args etc) select join points based on the dynamic context they come to pass in.

Advice defines code to be executed, whenever a join point of a particular set of join points is reached. It is part of the advice declaration to specify the set of join points, where it is to be executed. *Introductions* are used to crosscut the static type structure of the classes. With introductions, additional class members like constructors, methods, and fields may be inserted into classes as if they were declared in the classes themselves. *Aspects* are used to implement the crosscutting concerns. Aspects are additional unit of modularity and can be reused. It serves as containers for pointcuts, pieces of advice, introductions, and ordinary Java members. Aspects in AspectJ are instantiated by an extraordinary instantiation mechanism.

A. Advice:

Advice is an action that is being executed when an application reaches the join point [1] [3]. It is always defined relative to a pointcut. The pointcut selects the join points at which the advice executes. As the control passes through each join point twice (once, when join point is invoked and once, when join point completes its execution) the designer needs to specify at what point in time relative to the execution of join point the advice is to be executed. The body of the advice seems to be a body of method in a function call.

The types of advice are distinguished by when they run relative to the join points they affect:

a. *Before advice* runs before each affected join point.
b. *After advice* runs after the join point and comes in three flavors, i.e., unqualified after advice, after returning advice, and after throwing advice. Unqualified after

advice runs no matter what the outcome of the join point. After returning advice runs only if the join point returned normally (and it can perform additional matching based on the type of the value returned). After throwing advice executes if the join point ended by throwing an exception (and it can perform additional matching based on type of the exception thrown).

c. *Around advice* is the most intrusive. It runs instead of the join point and has the ability to invoke the join point (if it chooses) using the special proceed () syntax [3].

B. Advice Precedence:

Precedence affects both *when* advice executes and *whether* it executes at all. AspectJ allows programmers to control the precedence of advice because there's no way for the weaver to automatically know which advice should take precedence [2].

If two pieces of advice are defined in the same aspect, their precedence is determined by their order and type. There are two main situations. One of the pieces of advice is after advice. In this case, the advice defined later in the file takes the higher precedence. Neither advice is of the after type. In this case, the advice defined earlier in the file takes precedence.

The weaver reports some of the precedence relations as errors. These precedence relations are called as circular relationships. Circular precedence arises when the following precedence declarations appear in the same weaving:

declare precedence: A, B;
declare precedence: B, C;
declare precedence: C, A;

C. Runtime Execution:

The order of advice execution is determined by precedence. The advice with the highest precedence does not always execute first [1].

a. After advice defers execution. Instead of running immediately, after advice forwards control to the advice with the next highest precedence. When that advice finishes executing, the after advice runs its body if its subtype matches the outcome of the join point. (In other words, after throwing executes only if the next advice-or the join point-throws an exception.)

b. Before advice executes its body. If the advice terminates normally, it forwards control to advice with the next highest precedence. If the before advice throws an exception, it will prevent any advice of lower precedence from running.

c. Around advice also executes its body. It has the option of running the next advice by calling proceed(..). If it throws an exception or otherwise terminates before calling proceed (..), advice of lower precedence (and the join point) will not run.

The sequence trace representation for the runtime execution of AspectJ programs can be designed to ease the development of AspectJ programs [5]. A diagrammatic representation can help the developers to design and comprehend AspectJ programs. Its application carries over the advantages of aspect orientation to the design level and facilitates adaptation and reuse of existing design constructs. A sequence trace representation shows the flow of runtime execution in two-dimensional chart. The vertical dimension is the time axis, where time proceeds down the page. The horizontal dimension shows the roles of interacting objects [4] [9].

The order of introducing advice in aspect and their respective run time precedence with their sequence of execution is discussed in Section II. The Section III describes the concluding remarks and the related future research work.

II. SEQUENCE TRACE OF ADVICE

In the following sequence trace representations, the general roles are: main thread, object creation, aspect, before advice, after advice, around advice, proceed(..), and join point implementation. The control flow has been represented by numbered horizontal arrows. In the sequence trace diagram, program execution starts from main thread which creates the class object and then the control passes to the join point [8]. At the moment, aspect comes into picture and different types of advices starts their execution on the basis of the precedence order identified. Finally, control returns back to the main thread. In any AspectJ program that includes advice, there can be six different ways to introduce three different types of advice in aspect. In all cases, the advice with the next precedence includes the original join point if no further advice affects the join point.

The six different cases and their advice precedence are discussed in the following subsections specified by the figures and followed by programming examples with their respective outputs. All the cases will be illustrated with an example and related aspect that provides the idea of runtime execution of before, around and after advice. The Employee class has single attribute and a method. A main() method is used to instantiate an object of the class and calls to the setSalary() method. The aspect called MoneyAspect provides code to specify pointcut and the different advice. The code written for advice is in the order as the case specifies.

A. Case 1: Before, Around, After
Precedence Order: After, Before, Around

In this generalized case, after advice defers the execution and forwards the control to before advice, which acquires the next highest precedence. Before advice executes its body and forwards control to around advice. Around advice also executes its body. When around advice calls proceed(..) then the control passes to the join point implementation and it gets executed and returns the control back to around advice to execute the remaining body. Finally, after advice executes its body. The above case is represented in the form of sequence trace diagram, which is shown in Fig.1.

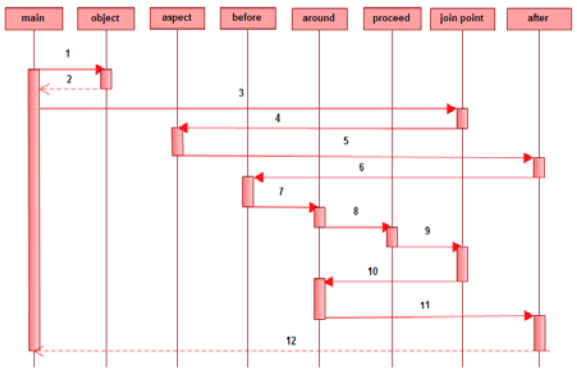

Figure 1. Before, Around and After Case.

We have illustrated the case below with the help of example: Employee.java and MoneyAspect.java programs. The output for the example is shown in Fig. 2.

a. Employee.java:

```
public class Employee {
int salary;
public void setSalary(int salary) {
this.salary = salary;
}
public static void main(String[] args) {
Employee emp = new Employee();
System.out.println("Salary before ");
emp.setSalary(50000);
System.out.println("Salary: " + emp.salary);
}
}
```

b. MoneyAspect.java:

```
public aspect MoneyAspect {
pointcut employeePC(int salary) : call(*
Employee.setSalary(..)) && args(salary) ;
before(int salary) : employeePC(salary){
System.out.println("Before");
}
void around(int salary) : employeePC(salary) {
System.out.println("Around 1 before proceed()");
salary *= 2;
System.out.println("Around 2 before proceed()");
proceed(salary);
System.out.println("Around 3 after proceed()");
}
after(int salary) : employeePC(salary){
System.out.println("After");
}
}
```

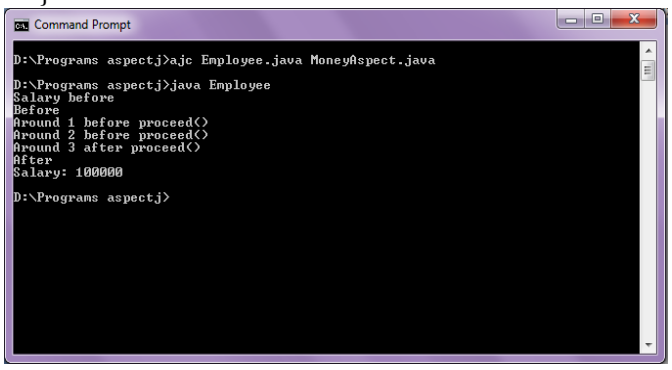

Figure 2. Output for Case 1.

B. Case 2: Around, Before, After
Precedence Order: After, Around, Before

In this case, after advice defers the execution and forwards the control to around advice which acquires the next highest precedence. Around advice also executes its body. When around advice calls proceed (..) then the control passes to the join point implementation and it gets executed and control passes to before advice. Before advice executes its body and passes the control back to around advice to executes its remaining body. Finally, after advice executes its body. The above case is represented in Fig.3 as sequence trace diagram.

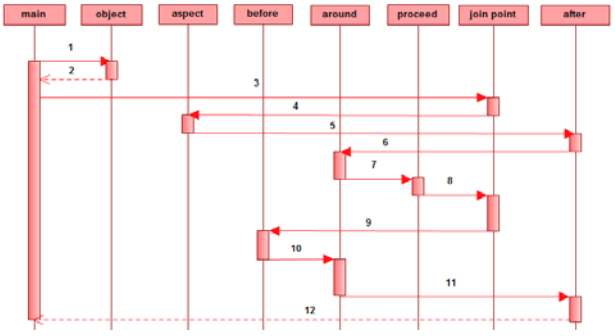

Figure 3. Around, Before and After Case

We have illustrated the case below with the help of example: Employee.java and MoneyAspect.java programs. The output for the example is shown in Fig. 4.

a. Employee.java:

```
public class Employee {
int salary;
public void setSalary(int salary) {
this.salary = salary;
}
public static void main(String[] args) {
Employee emp = new Employee();
System.out.println("Salary before ");
emp.setSalary(50000);
System.out.println("Salary: " + emp.salary);
}
}
```

b. MoneyAspect.java:

```
public aspect MoneyAspect {
pointcut employeePC(int salary) : call(*
Employee.setSalary(..)) && args(salary) ;
void around(int salary) : employeePC(salary) {
System.out.println("Around 1 before proceed()");
salary *= 2;
System.out.println("Around 2 before proceed()");
proceed(salary);
System.out.println("Around 3 after proceed()");
}
before(int salary) : employeePC(salary){
System.out.println("Before");
}
after(int salary) : employeePC(salary){
System.out.println("After");
}
}
```

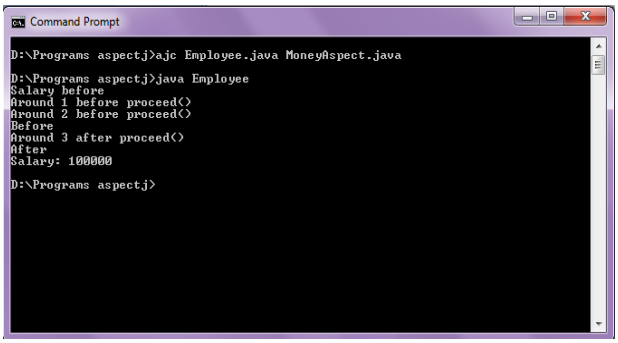

Figure 4. Output for Case 2.

C. Case 3: After, Before, Around
Precedence Order: Before, Around, After

In this case, before advice with the highest precedence, executes its body and forwards control to around advice with the next highest precedence. Around advice also executes its body. When it calls the proceed(..), the control passes to the join point implementation and passes the control to after advice having the lowest precedence and gets executed as there is no advice with next highest precedence. Finally, control passes to around advice to execute its remaining body. The above case is represented in Fig.5 as sequence trace diagram.

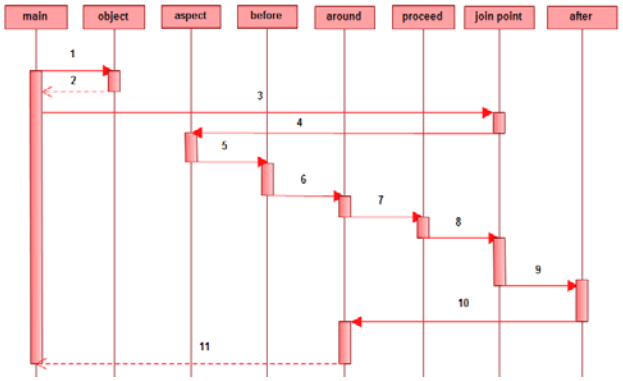

Figure 5. After, Before and Around Case.

The same case is illustrated below with the help of example: Employee.java and MoneyAspect.java programs. The output for the example is shown in Fig. 6.

a. Employee.java:

```
public class Employee {
int salary;
public void setSalary(int salary) {
this.salary = salary;
}
public static void main(String[] args) {
Employee emp = new Employee();
System.out.println("Salary before ");
emp.setSalary(50000);
System.out.println("Salary: " + emp.salary);
}
}
```

b. Money Aspect.java:

```
public aspect MoneyAspect {
pointcut employeePC(int salary) : call(*
Employee.setSalary(..)) && args(salary) ;
after(int salary) : employeePC(salary){
System.out.println("After");
}
before(int salary) : employeePC(salary){
System.out.println("Before");
}
void around(int salary) : employeePC(salary) {
System.out.println("Around 1 before proceed()");
salary *= 2;
System.out.println("Around 2 before proceed()");
proceed(salary);
System.out.println("Around 3 after proceed()");
}
```

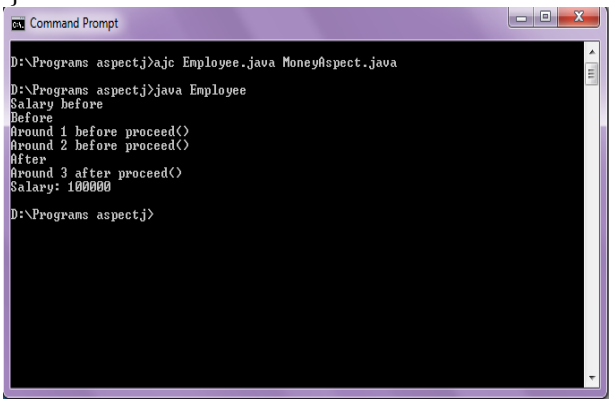

Figure 6. Output for Case 3.

D. Case 4: After, Around Before
Precedence Order: Around, Before, After

In this case, around advice with the highest precedence, executes its body and calls the proceed (..), the control passes to the join point implementation and goes to before advice with the next highest precedence. Before advice executes its body and passes the control to after advice with the next highest precedence. After advice executes its body and passes the control back to around advice to execute its remaining body. The above case is represented in Fig.7 as sequence trace diagram.

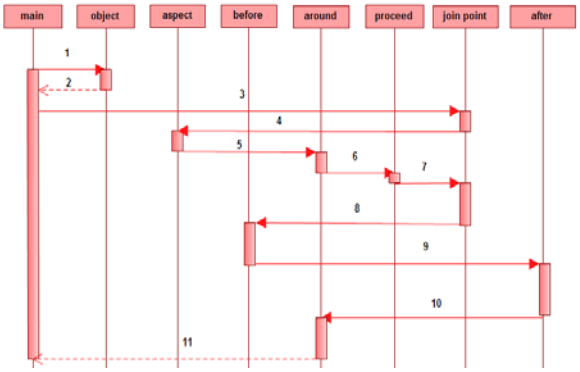

Figure 7. After, Around and Before Case.

We have illustrated the case below with the help of example: Employee.java and MoneyAspect.java programs. The output for the example is shown in Fig. 8.

a. Employee.java:

```
public class Employee {
int salary;
public void setSalary(int salary) {
this.salary = salary;
}
public static void main(String[] args) {
Employee emp = new Employee();
System.out.println("Salary before ");
emp.setSalary(50000);
System.out.println("Salary: " + emp.salary);
}
}
```

b. Money Aspect.java:

```
public aspect MoneyAspect {
```

```
pointcut employeePC(int salary) : call(*
Employee.setSalary(..)) && args(salary) ;
after(int salary) : employeePC(salary){
System.out.println("After");
}
void around(int salary) : employeePC(salary) {
System.out.println("Around 1 before proceed()");
salary *= 2;
System.out.println("Around 2 before proceed()");
proceed(salary);
System.out.println("Around 3 after proceed()");
}
before(int salary) : employeePC(salary){
System.out.println("Before");
}
}
```

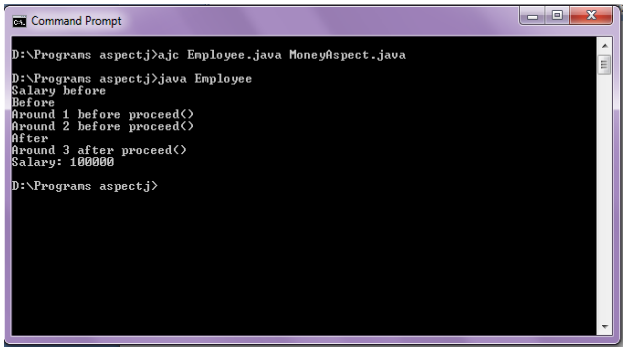

Figure 8. Output for Case 4.

E. Other Two Cases:

In the next two cases where the order of introducing different advices in a program code, is either "before, after, around" or "around, after, before", the circular precedence error arises in the runtime execution. Hence, the program cannot be executed. Its output is shown in Fig. 9.

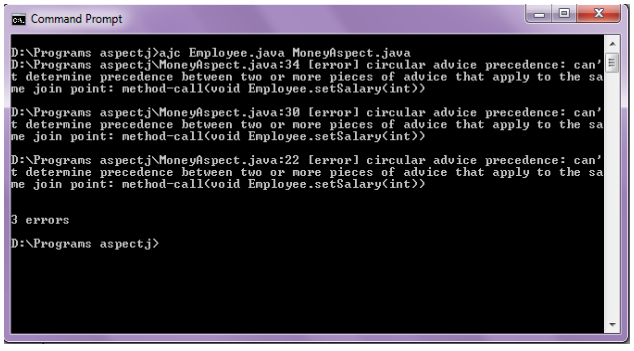

Figure 9. Output for other two cases.

III. CONCLUSION

To understand the behavior of a program it is essential to validate its run-time semantics. With the introduction of new abstraction techniques, such as AOP, the gap between the textual representation of a program and its behavioral semantics begin to widen. In AOP, aspects enable programmers to encapsulate crosscutting concerns into a module. While, encapsulation localizes a concern's source code in one location, the code can still affect multiple points of execution.

In this research work, the visual representation of different types of advice execution in program is presented via sequence trace diagram. The sequence trace diagram has been validated with the practical implementation of AspectJ programs. A sequence diagram helps the designers to examine the behavior of the system. It also benefits the developers to understand the runtime environment and execution preferences more efficiently. The goal of these diagrams is to reduce the gap between code and documentation. The approach can be extended to represent after throwing and after returning advice in sequence trace diagram.

IV. REFERENCES

[1] Joseph D. Gradecki, Nicholas Lesiecki, "Mastering AspectJ Aspect- Oriented Programming in Java," Published by Wiley Publishing, Inc., 2003.

[2] Ramnivas Laddad, "AspectJ in Action Practical Aspect-Oriented Programming," Manning Publication Co., 2003.

[3] Russell Miles, "AspectJ Cookbook," O'Reilly Publications, 2005.

[4] James Rumbaugh, Ivar Jacobson, Grady Booch, "The Unified Modeling Language Reference Manual," Addison-Wesley Publication,2nd edition.

[5] Steven She, "Retrieving Sequence Diagrams from Aspect-Oriented Systems," unpublished.

[6] Dominik Stein, "An Aspect-Oriented Design Model Based on AspectJ and UML," Master thesis submitted to Department of Business Arts,Economics and Management Information Systems, University of Essen, Germany.

[7] Dominik Stein, Stefan Hanenberg, Rainer Unland, "An UML-based Aspect-Oriented Design Notation For AspectJ," Institute for Computer Science University of Essen, Germany.

[8] YAN Han, Gunter Kniesel, Armin B. Cremers, "Towards Visual AspectJ by a Meta Model and Modeling Notation,"

[9] G. Booch, J. Rumbaugh, and I. Jacobson "The Unified Modeling Language User Guide," Addison-Wesley, Reading, Massachusetts, USA,1st edition, 1999.

[10] www.eclipse.org/aspectj