Aspect-Oriented Debugging

John W. Stamey, Jr., **Bryan T. Saunders, Ryan Watts**
Department of Computer Science
Coastal Carolina University
Conway, SC 29528
jwstamey@coastal.edu, btsaunde@coastal.edu, mrwatts@coastal.edu

ABSTRACT
Aspect-Oriented Programming (AOP) may be used to implement non-invasive debugging for Java Programs whereby program code (code to be debugged) is not altered. This paper illustrates techniques for tracing the execution of loops, methods and constructors using AOP. We also introduce a Java packagenamed AOBugz which has methods allowing programmers to insert break points and put comments around code. The AOBugz package is written using aspects.

Categories and Subject Descriptors
D.2.10 [**Software Engineering**]: Design
D.3.2 [**Programming Languages**]: Languages

General Terms
Software Engineering, Testing

Keywords
Aspect-Oriented Programming, Debugging

1. INTRODUCTION
While a number of excellent testing and debugging tools for Java are available for free download and use, the installation and learning curve can be significant. [1]. AOP provides a new and valuable alternative to tools such as JUnit (http://www.junit.org), Jtest (http://www.parasoft.com), and the JDB Java Debugger from Sun Microsystems (http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html).

The advantage of using aspects in debugging Java code is that few if any changes need to be made to the code being tested. All of the statements performing tracing functions are written in the aspects (files with the .aj extension), and are woven into the code by the AspectJ extension to Java. The Eclipse Java compiler (http://www.eclipse.org/) was used with the AspectJ extension to Java (http://www.eclipse.org/aspectj) to implement each of the examples presented in this paper.

Debugging is a skill critical to success in programming. Client-side Java development, in both command-line and sophisticated Integrated Development Environments (IDEs), requires that we perform debugging tasks such as setting breakpoints and tracing [2]. Tracing is best when it can be done at the variable level (when a variable changes), upon statement execution, and upon execution of a method or constructor. Aspects will be found to be helpful in development of these techniques.

2. DEBUGGING TECHNIQUES
The examples presented have the following implementation characteristics:

Debugging Action	Method Used
Tracing – variable level	Aspects only
Tracing – statement level	Aspects & Method Calls
Tracing – method/constructor level	Aspects only
Setting Breakpoints	Aspects & Method Calls

Figure 1

We first present tracing techniques that use only aspects (Sections 3 and 4). Then, we present techniques that use an additional class called AOBugz (Sections 5). We have included a representative sample of code and output in this article. All code and working examples are available at the website http://www.AspectOrientedProgramming.org/ .

3. TRACING AT THE VARIABLE LEVEL
Our first example, uses an aspect to trace of variables throughout the execution of a FOR loop. Figure 1A show sample client (target) code to be debugged and Figure 1B output resulting from the use of aspects for debugging Figure 2 shows the aspect class Debugger that created the trace found in Figure 1. The SET joinpoint, used when variables are "written to," is executed every time the value of x and y change. The GET joinpoint is executed when y is passed to a function (y is read). This is reflected in the trace through "`=== Setting >`" and "`=== Getting >`", respectively. The construct `thisJoinPoint.getSignature()` will display the name of the variable being used. thisJoinPoint is the equivalent to this at the level of joinpoints. The before advice code shows the signature of the method or constructor invoked, any parameters passed, and the line number from which the call was made. In this case, the before advice is called prior to execution of any method in the class Client that returns an integer. The after advice simply prints out the method signature and states that the call has completed.

4. TRACING AT THE METHOD/CONSTRUCTOR LEVEL

Our second example uses aspects to track the call and execution of both methods and constructors. The client code may be seen in Figure 2A. Aspects allow us to view parameters passed into the methods and constructor, and provide tracing through their execution. The before advice of the CALL joinpoint executes prior to the `add()` and `mult()` methods in the client code (Figure 2B). The after advice of the CALL joinpoint executes after the `add()` and `mult()` methods in Figure 2A have completed. The EXECUTION joinpoint happens before code of the method or constructor starts or ends execution. We see that once aspects are in place to provide tracing capabilities, they are quite straightforward to modify. It is a real benefit that the target code (written for the assignment or system) is never changed; this eliminates the possibility of errors that can inadvertently creep into code where debugging is invasive (hard-coded into the program as opposed to using aspects).

```
=== Getting >: int Client.i
Location: Client.java:15
=== Setting >: int Client.i
New Value: 1
Location: Client.java:15
=== Getting >: int Client.i
Location: Client.java:15
=== Getting >: int Client.y
Location: Client.java:16
=== Setting >: int Client.x
New Value: 9
Location: Client.java:16
=== Getting >: int Client.y
Location: Client.java:17
=== Setting >: int Client.y
New Value: 12
Location: Client.java:17
```

Figure 1A

```
//Client Code

 public class Client {
 public static int x = 4;
 public static int y = 6;
 public static int i = 0;
 public static void main(String[] args) {
       for(i = 0;i<10;i++)  {
              x = y;
              y = add3(y); }
       }
}
public static int add3(int a)
       { return a += 3; }

//Aspect Code

public aspect Debugger {
   before(int n): set(int Client.*)
      && args(n){
      System.out.println("=== Setting >:
         +thisJoinPoint.getSignature());
      System.out.println("New Value: "+n);
      System.out.println("Location:
        "+thisJoinPoint.getSourceLocation());
      }
   after(): get(int Client.*) {
      System.out.println("=== Getting >:
        "+thisJoinPoint.getSignature());
      System.out.println("Location:
      "+thisJoinPoint.getSourceLocation());
   }
}
```

Figure 1B

```
// Client and Class Code B
public class Client1 {
       public static int add(int x, int
y){ return x+y; }
       public static int mul(int x, int
y){ return x*y; }


public static void main(String[] args) {
       int x = 4;
       int y = 3;
       Num z = new Num();
       Num q = new Num(4);

       int a = add(x,y);
       int b = mul(x,y);
       }
}


public class Num {
       private int num;
       public Num(){num = 0;}
       public Num(int x){num = x;}
       public int getNum() {return num;}
       public void setNum(int num) {this.num =
num;}
}
```

Figure 2A

5. USING AOBugz

A Java package named AOBugz, was created to help with tracing at the statement level and to set breakpoints. Currently, AspectJ supports advice on methods, constructors and fields, but not on particular statements. The code in Figure 3 illustrates the use of this class to create statement level traces with the method `AOBugz.tracez()` and to create breakpoints with the method `AOBugz.breakz()`. We have worked to require minimum change in the Java code to be tested with AOBugz.

`AOBugz.tracez()` allows text comments to be inserted, as well as any number of variables to be displayed. The aspect prints out a sequence of parameters specified in the program being tested.

`AOBugz.breakz()` allows for specific variables to be sent as parameters and displayed, as well as halting the code by waiting for a prompt from the keyboard to continue. This aspect also has an option whereby all variables to be displayed by taking advantage of the privileged declaration of the aspect. Sample output from both of these methods is found in Figure 4.

```
// Debug Code
public aspect Debugger {
   // Before a Call to Any Method
   before() :call(* Client1.*(..)){
      System.out.println("=== Calling >:
         "+thisJoinPoint.getSignature());
      Object[] jpargs = thisJoinPoint.getArgs();
      if(jpargs.length >= 1){
         System.out.print("Using Args: ");
         for(int i = 0;i<jpargs.length;i++){
            System.out.print("["+i+"]=
               "+jpargs[i]+"");
          }
      System.out.println();
      }
      System.out.println("Location:
        "+thisJoinPoint.getSourceLocation());
   }
// After a Call to Any Method of Client B
   after() :call(* Client1.*(..)){
      System.out.println("=== End Call >:
         "+thisJoinPoint.getSignature());
      System.out.println("Location:
         "+thisJoinPoint.getSourceLocation());
      System.out.println();
   }
// Before Execution of Any Method
// Code available on
AspectOrientedProgramming.org

// After the Execution of Any Method
// Code available on
AspectOrientedProgramming.org

// Before the Execution of Any Constructor of Num
   before() :call(public Num.new(..)){
      System.out.println("=== Constructing >:
         "+thisJoinPoint.getSignature());
      System.out.println("Target:
         "+thisJoinPoint.getThis());
      Object[] jpargs = thisJoinPoint.getArgs();
      if(jpargs.length >= 1){
         System.out.print("Using Args: ");
         for(int i = 0;i<jpargs.length;i++){
            System.out.print("["+i+"]=
               "+jpargs[i]+" ");
         }
      System.out.println();
      }
      System.out.println("Location:
         "+thisJoinPoint.getSourceLocation());
      }
```

Figure 2B

6. CONCLUSIONS

We have shown two techniques in which aspects can be used to provide relatively non-invasive techniques to help in the task of debugging. These solutions do not require a great deal of time for installation and learning, and are easily reusable. Further research by the authors is underway to port these methods to Aspect-Oriented PHP (http://www.aophp.net/).

7. ACKNOWLEDGMENTS

The authors would like to thank Jean-Louis Lassez for his enthusiasm and encouragement in this work.

8. REFERENCES

[1] Miller, K.W. Test Driven Development on the Cheap: Text Files and Explicit Scaffolding. Journal of Computing Sciences in Colleges, Vol. 20, No. 2 (December 2004), 181-189.

[2] Bennett, L. Java Debugging. Archived at http://www.ibm.com/developerWorks.

```
public class Test {
   public static void main(String[] args) {
       int x = 0;
       int y = 3;
       for(int i = 0;i<10;i++){
           x = i;
           y = i-4*4/3;
           if(i == 5){
               AOBugz.breakz(new
Integer(y),"Y",new Integer(x),"X");
           }
           AOBugz.tracez(new
Integer(i),"Iteration",new Integer(x),"X");
       }
   }
}
```

Figure 3

```
Iteration: 0  X: 0
Iteration: 1  X: 1
Iteration: 2  X: 2
Iteration: 3  X: 3
Iteration: 4  X: 4
View All(ALL),Specific Var(Y,X), Continue(CONT)Y
Y: 0
View All(ALL),Specific Var(Y,X), Continue(CONT)ALL
Y: 0  X: 5
View All(ALL),Specific Var(Y,X),
Continue(CONT)CONT
Iteration: 5  X: 5
Iteration: 6  X: 6
Iteration: 7  X: 7
Iteration: 8  X: 8
Iteration: 9  X: 9
```

Figure 4