UNIT TESTING AND DEBUGGING WITH ASPECTS

John Stamey, Jr.
Bryan Saunders
Department of Computer Science
Coastal Carolina University
Conway, SC 29528
(864) 359-2552
jwstamey@coastal.edu, btsaunde@coastal.edu

ABSTRACT

Aspect-Oriented Programming (AOP) may be used to implement non-invasive debugging for Java Programs whereby target code (code to be debugged) is not altered. This paper illustrates techniques for tracing the execution of loops, methods and constructors using AOP. The use of aspects to implement debugging can also eliminate the need to install and learn new debugging packages for code tracing.

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. [4]. 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 (http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html).

The advantage of using aspects in testing and 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. Examples are presented that are straightforward to implement for both students and professional programmers. 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.

ABOUT ASPECTS

An *aspect* is a fundamental unit in an object-oriented programming language, much the same as a class. Aspects contain data, methods, and objects of other classes, just like classes do. They are instantiated at runtime, based on program execution, exhibiting polymorphic behavior [3].

One way of thinking about aspects is to first imagine some Java code to do something nontrivial, such as take input from a browser, validate the input, and update a database. It is reasonable to think of the input and output (database update) as the important and critical steps in this program. The validation, necessary for correct

operation of the system, is both a vital part of reliable system operation yet at the same time not particularly related to the pure input-update algorithm.

In a perfect world, we would like to somehow remove all code that is not part of our computation/system goal, thus making out code simpler and easier to understand (and maintain). However, we still need for non-related concerns such as validation to be performed at the appropriate time. While the example presented is quite contrived and trivial, it should be apparent that real systems can have thousands of joinpoints. From that standpoint, AOP provides a powerful way to keep code clean and free of crosscutting concerns (system requirements that occur more than once, and in more than once location) that are not actually a part of the computation.

Aspects allow us to have the best of both words. Figure 1 shows some Java code to read in a file and write to a different with the validation code as part of the same program. Figure 2 shows how much cleaner the code can be if the validation concern is separated out; now there are two files – a .java file with input and output, and a .aj file with the validation code. The validation code, more properly called around advice, writes a record to the file if the data is valid; otherwise, nothing is written.

```
public class Main {

	public static void main(String[] args) throws IOException, FileNotFoundException
{
		Scanner fileIn = new Scanner(new File("in.txt"));
		String input = "";
		PrintWriter fileOut = new PrintWriter(new FileWriter("out.txt"));
		while(fileIn.hasNext()){
			input = fileIn.nextLine();
			if(isValid(input)){
				fileOut.println(input);
			}
		}
		fileOut.close();
		fileIn.close();
	}

	private static boolean isValid(String in){
		boolean valid = false;
		int num;
		try{
			num = -1;
			num = Integer.parseInt(in);
			valid = (num < 0) ? false : true;
		}catch(NumberFormatException e){
			valid = false;
		}
		return valid;
	}
}
```

Figure 1

Advice is then integrated into the code through a process called weaving, to form the final system. Figure 3 gives a graphical depiction of how this works with AspectJ and the Eclipse Java IDE.

```
public class Main {

	public static void main(String[] args) throws IOException, FileNotFoundException
{
		Scanner fileIn = new Scanner(new File("in.txt"));
		String input = "";
		PrintWriter fileOut = new PrintWriter(new FileWriter("out.txt"));
		while(fileIn.hasNext()){
			input = fileIn.nextLine();
			fileOut.println(input);
		}
		fileOut.close();
		fileIn.close();
	}
}

import java.io.PrintWriter;
public aspect Aspect {
	void around(String in): call(* PrintWriter.println(String)) && args(in){
		boolean valid = false;
		int num;
		try{
			num = -1;
			num = Integer.parseInt(in);
			valid = (num < 0) ? false : true;
		}catch(NumberFormatException e){
			valid = false;
		}
		if(valid){
			proceed(in);
		}
	}
}
```

Figure 2

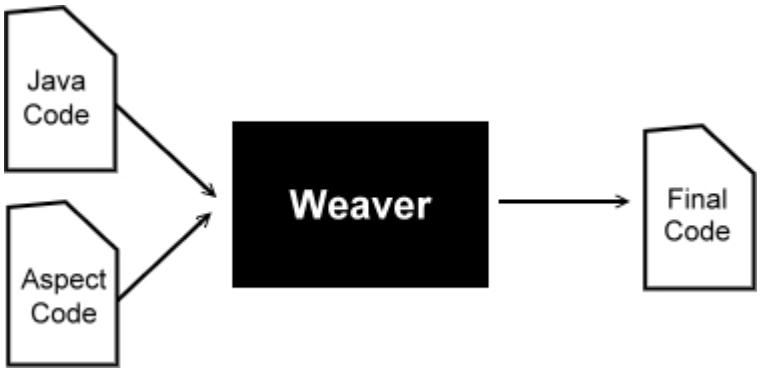

Figure 3

Aspects can execute at any of three different points in program execution, through action (program statements) that we call *advice*. :

- Before advice is a set of statements that execute prior to a join point
- After advice is a set of statements that execute after a join point
- Around advice is a set of statements that can alter the context (and therefore execution) of statements at a join point.

A *concern* is a system requirement. Concerns include security, logging, authentication and authorization, business rules, and debugging/testing. Each concern executes from code location called a *joinpoint*. A *pointcut* is a collection of joinpoints for one specific concern. *Crosscutting concerns* occur when a particular system

requirement/concern is implemented in more than one location [5]. Code to implement aspects, known as *advice*, can execute before, after or around (usually altering execution through the use of conditional statements) a joinpoint. Such code segments are called *before advice*, *after advice*, and *around advice*, respectively.

As system development, system modifications and new releases occur, the idea of streamlining that code (basically, refactoring) can be overlooked or suffer from a low position on the priority list. Resulting problems of *code scattering* (duplicated code) and *code tangling* (multiple concerns are implemented within the same programming logic) can plague the system. [2]

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 [1]:

- Setting breakpoints
- Inspecting variables
- Using comments to block out code
- Tracing

AOP will be found to provide a straightforward solution to inspecting variables and tracing.

```
//Client code called client.java
public class Client {
        public static void div(double n1, double n2){
                System.out.println(n1+"/"+n2+"="+(n1 / n2));
        }
        public static void main(String[] args) {
                double x = 4.0;
                double y = 12.0;
                double z = 0.0;
                div(y,x);
                System.out.println("==============");
                div(x,z);
        }
}

// Aspect Code, saved in a file called aspect.aj (Eclipse 3.0)
public aspect Aspect {
        // Before Advice
        before(double s1, double s2):
                call(void *.div(double,double)) && args(s1,s2){
                System.out.println("About to Divide "+s1+" by "+s2);
        }
        // Around Advice
        void around(double s1, double s2):
                call(void *.div(double,double)) && args(s1,s2){
                if(s2 == 0.0){
                        System.out.println("Cant Divide by Zero");
                } else {
                        proceed(s1,s2);
                }
        }
        // After Advice
        after(): call(void *.div(double,double)){
                System.out.println("Division Complete");
        }
}
```

Figure 4: A Simple Example

A FIRST EXAMPLE

We start with a look at a simple example of AOP. Figure 4 shows some client code that could have the problem of division by zero. The before advice, executing before the `div` method, states that division is about to commence. The around advice checks to see if division by zero will occur. If the denominator (s2) is zero, then an error message appears; otherwise, the division proceeds with variables s1 and s2. The after advice, executing upon completion of the `div` method, states that division has completed. We note that the same functionality performed by the around advice could just as well have been implemented with a try-catch block. However, should this functionality need to have been inserted around dozens of methods, aspects provide a superior solution.

ASPECT-ORIENTED DEBUGGING TECHNIQUES

Our first example, Code A, uses an aspect to trace variables throughout the execution of a FOR loop. Figure 5 show sample client (target) code to be debugged and output resulting from the use of aspects for debugging.:

```
=== 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
```

```
 //Client Code A
 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; }
```

Figure 5: Output (left) and Client Code (right) from A

Figure 6 shows the aspect class Debugger that created the trace found in Figure 5. 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.

```
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 6: Aspect Code A

Our second example, Code B, uses aspects to track the call and execution of both methods and constructors. The client code may be seen in Figure 7. 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 Client Code B (Figure 8). The after advice of the CALL joinpoint executes after the `add()` and `mult`() methods in Client Code B have completed. completed. The EXECUTION joinpoint happens before code of the method or constructor starts or ends execution. A general graphic depiction of the difference between CALL and EXECUTION joinpoints may be seen in Figures 9. Careful examination of the coding examples contained herein show 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).

```
=== Executing >: void
Client1.main(String[])
Using Args:
[0]=[Ljava.lang.String;@3ee284
Location: Client1.java:11
=== Constructing >: Num()
Location: Client1.java:14
=== Constructing >: Num(int)
Using Args: [0]=4
Location: Client1.java:15
=== Calling >: int
Client1.add(int, int)
Using Args: [0]=4 [1]=3
Location: Client1.java:17
=== Executing >: int
Client1.add(int, int)
Using Args: [0]=4 [1]=3
Location: Client1.java:8
=== End Execution >: int
Client1.add(int, int)
Location: Client1.java:8
=== End Call >: int
Client1.add(int, int)
Location: Client1.java:17
=== Calling >: int
Client1.mul(int, int)
Using Args: [0]=4 [1]=3
Location: Client1.java:18
```

```
// 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 7: Output (left) and Client Code (right) from B

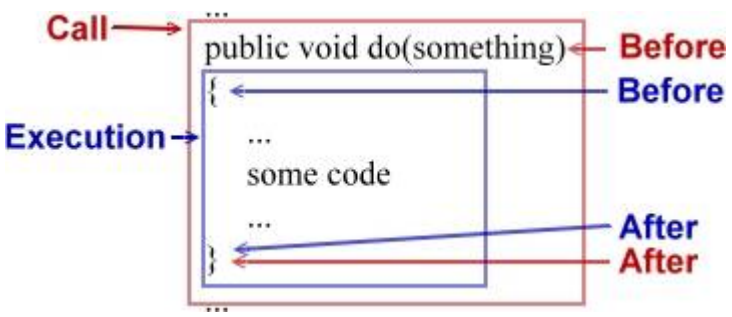

Figure 8: CALL and EXECUTION Joinpoint Execution Diagram

```
// Debug Code B
public aspect Debugger {
	// Before a Call to Any Method of Client B
	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 the Execution of Any Method of Client B
	before() :execution(* Client1.*(..)){
		System.out.println("=== Executing >:
			"+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 the Execution of Any Method of Client B
	after() :execution(* Client1.*(..)){
		System.out.println("=== End Execution >:
			"+thisJoinPoint.getSignature());
		System.out.println("Location:
			"+thisJoinPoint.getSourceLocation());
	}

	// 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 9: Aspect Code B

CONCLUSIONS

In this paper, we have shown a straightforward and non-invasive way implement two important elements debugging in Java, variable tracing and inspection. The solutions do not require time for installing and learning an entirely new debugging package. All that is required is the code presented herein, as well as the AspectJ add-in to the Java compiler. These examples are simple enough to be used by students in CS2 programming projects, as well as professional programmers looking for native support for debugging. Further research by the authors is underway to develop an API that would allow setting of breakpoints and code blocking.

REFERENCES

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

[2] Elrad, T., Filman, T.E. & Bader, A., Editors, *Communications of the ACM*, Vol. 44., No. 10, October 2001, pp. 29-97 (Collected Articles)

[3] Kiczales, G. Aspect-Oriented Programming. ACM Computing Surveys, December 1996.

[4] 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.

[5] Stamey, J.W. & Saunders, B.T. Implementing Database Solutions on the Web with Aspect-Oriented Programming. *Journal of Computing Sciences in Colleges (to appear)*, Vol. 21, No. 1.