Bugdel: An Aspect-Oriented Debugging System

Yoshiyuki Usui and Shigeru Chiba
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku Tokyo 152-8552, JAPAN
Phone: +81–3–5734–3041 Fax: +81–3–5734–2754

Abstract

This paper presents our aspect-oriented system specialized for debugging named Bugdel. Bugdel is a plug-in module for Eclipse, which is an integrated development environment. Unlike other general-purpose aspect-oriented systems, Bugdel allows user to specify aspects using a graphical user interface. Furthermore, Bugdel provides new functions that AspectJ or others do not provide, since they break the modularity of classes. This paper discusses limitations of related general-purpose aspect-oriented systems, such as AspectJ, and shows the solutions that we have implemented in Bugdel.

1 Introduction

Debugging is a well-known application of aspect-oriented programming (AOP) [13]. Execution traces (or logs) are often significantly useful for fixing complex bugs, and AOP helps producing such traces: in that case, advices are used to print trace messages. Unfortunately, current AOP systems such as AspectJ [12, 14] are general-purpose systems, thus they have limitations for debugging. First, the developers have to learn the syntax of AspectJ only for describing an aspect for debugging. Second, the available join points are limited: some kinds of execution points cannot be selected by pointcuts. When using advices for trace printing, this restriction is sometime a serious problem: limits the events that can be traced. Third, an advice body cannot access local variables at join points.

This paper presents our AOP system, named Bugdel, which is specialized for debugging. It is part of an integrated development environment (IDE), and users can describe aspects using GUI. In addition, to solve the limitations of general-purpose AOP systems, Bugdel provides two new pointcut designators. One is for selecting lines specified by line numbers as join points, and the other is for selecting all the lines in the body of a specified method. An additional original feature is that, an advice body can access local variables at join points. These features of Bugdel might not be appropriate for general-purpose AOP systems but they are necessary in practice for debugging. We have adopted them for Bugdel since we have designed Bugdel as a domain-specific AOP system [4, 6, 11, 15] to debugging.

The rest of this paper is organized as follows. Section 2 describes benefits and limitations of using general-purpose AOP systems for debugging. Section 3 presents our AO debugging system Bugdel, and section 4 focuses on breakpoint emulation in Bugdel. Section 5 compares Bugdel and other systems. We conclude this paper in section 6.

2 Debugging by using an AOP system

Using aspect-oriented programming for debugging is a well-known application of AOP. This section presents potential benefits of using AOP for debugging and limitations of existing general-purpose AOP systems such as AspectJ with respect to debugging.

2.1 Benefits

The code for debugging often includes crosscutting concerns, such as the producing of trace messages. When developers do not have any ideas of where a bug exists in source files, they will want to see a log of trace messages printed during program execution. The debug code for producing such messages cuts across several program modules. Interactive debugging by using breakpoints and step execution is not appropriate for such an early phase of debugging. For instance, if developers want to investigate how the value of a variable changes during loop iteration, they will not want to use a breakpoint to stop the execution at every iteration and examine the value of the variable. Modifying the loop statement to print the value of the variable at every iteration is a better and common approach. If a program were multi-threaded, interactive debugging with breakpoints would be more difficult.

There are a few benefits of describing a crosscutting concern for debugging as an aspect. First, AOP simplifies the description for specifying a number of execution points where a trace message is printed. For example, the simple pointcut description set(* Point.x) in AspectJ identifies all occurrences of assignments to the field x in Point. Developers can use this pointcut description to print trace messages at those field assignments. They need not enumerate a large number of occurrences of the field assignments.

Second, debug code can be clearly separated from the rest of the source code. A debug concern is essentially irrelevant to the other concerns described in the rest of the source code. Thus, if the debug code is tangled with the rest of the source code, the resulting code is difficult to read and understand. Furthermore, when developers insert or remove the debug code, they might wrongly change the source code around the debug code and then that change might cause another bug. Such problems are avoided when using AOP for separating debug code as an aspect.

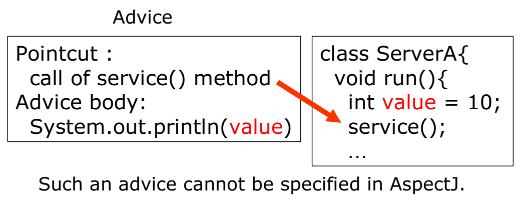
2.2 Limitations of general-purpose AOP systems

Although AOP has potential to be a useful tool for debugging, existing AOP systems like AspectJ are not satisfactory. Those general-purpose AOP systems have limitations to support debugging fully. Even if AspectJ is used on top of the Eclipse [7, 8, 9] software development environment with AJDT [1], which is a plug-in to support AspectJ, it is not sufficient.

First, developers must learn the grammar of AspectJ for specifying an aspect for debugging. They must also learn the primitive pointcuts provided by AspectJ. Learning them only for debugging does not pay off.

Second, some execution points cannot be specified by combinations of pointcuts in AspectJ. For example, while debugging, developers may want to run debug code when the thread of control reaches the line 10 in some source file. However, the current version of AspectJ does not allow such a specific line to be identified as a join point. The join points available in AspectJ are only field accesses, method calls, and other kinds of execution for object-oriented programming. However, it is not a design flaw but a right decision that general-purpose AOP systems like AspectJ do not provide a pointcut for selecting a particular line number as a join point. If such a pointcut is available, developers may specify an aspect heavily depending on the representation of the target source code. Such an aspect breaks modularity and encapsulation since the aspect must be modified as well if the target source code is modified and then the line numbers change.

Third, local variables around a join point are not accessible from advice bodies, that is, debug code as illustrate in Figure 1. This limitation is often inconvenient for debugging.



```
Pointcut :
  call of service() method
Advice body:
  System.out.println(value)
```

```
class ServerA{
  void run(){
    int value = 10;
    service();
    ...
```

Figure 1. Accessing local variables

Again, this is not a design flaw of AspectJ, which is a general-purpose AOP language. If an advice body could access the local variables, the modularity and encapsulation of classes would be broken since local variable names are exposed to other modules. Moreover, AspectJ does not allow advice bodies to access private fields or methods of the target objects unless the aspect including those advice bodies are privileged. This restriction for keeping modularity and encapsulation is often inconvenient for debugging.

Finally, existing AOP systems like AspectJ are not integrated with software development environments. For example, AspectJ does not provide direct supports for step execution. It would be useful if an aspect can directly interact with a debugger to specify a break point and perform step execution. These limitations are also inconvenient for debugging.

3 Bugdel

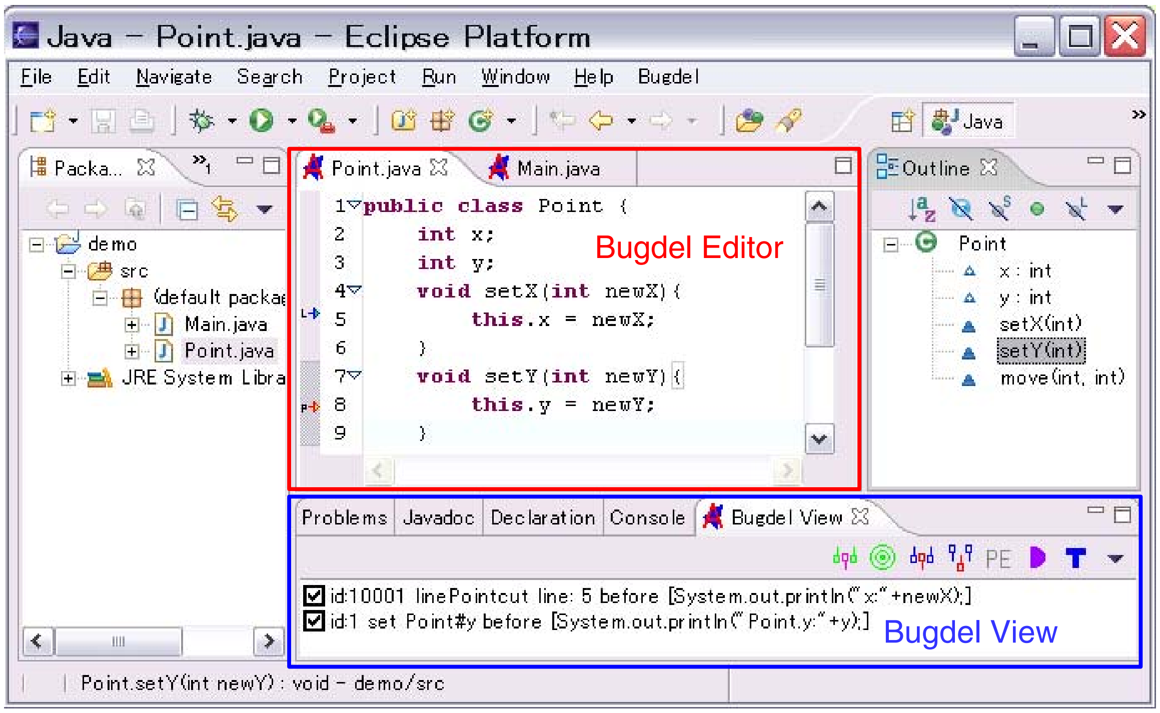
We have developed Bugdel [3], which is an aspect-oriented debugging system for Java. Bugdel is a specialized system for debugging and thus overcome the limitation of general-purpose systems involve. It has three original features:

1. Pointcuts and advices are specified through a GUI.
2. Custom pointcuts for debugging (line pointcut and allLines pointcut) are provided.
3. Local variables around a join point are accessible from an advice body.

Bugdel is a plug-in module of Eclipse, which is a widely used software development environment. The GUI of Bugdel consists of Bugdel editor and Bugdel view (Figure 2). The developers use them for specifying pointcuts and advices. Debug code described as an aspect is woven into class files by the weaver of Bugdel.

3.1 GUI based programming

Pointcuts and advices are specified through mouse operations and dialog boxes. Developers do not have to write



```
public class Point {
    int x;
    int y;
    void setX(int newX){
        this.x = newX;
    }
    void setY(int newY){
        this.y = newY;
    }
```

```
☑ id:10001 linePointcut line: 5 before [System.out.println("x:"+newX);]
☑ id:1 set Point#y before [System.out.println("Point.y:"+y);]
```

```
Point.setY(int newY) : void - demo/src
```

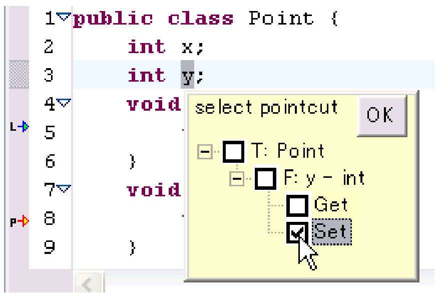
Figure 2. Bugdel Editor and Bugdel View

a program directly in an AOP language like AspectJ. This GUI-based programming interface reduces the amount of knowledge that Bugdel users have to learn for using Bugdel, compared to AspectJ with AJDT for Eclipse.

If developers want to specify a pointcut for identifying field accesses or method calls, they can first select the target field or method by mouse clicking on a field or method name in a source file. Then they can select the pointcut menu in a pop-up menu and see the list of the candidates of the pointcut. They can choose an appropriate pointcut from those candidates.

In Figure 3, the field y in the Point class is being selected. Thus, Bugdel propose two candidate pointcuts: set (field assignment) and get (field reference).

Developers can also specify a pointcut through dialog boxes, by selecting pointcut from the Bugdel menu. Then they can input a kind of pointcut, a class name, a field name, and so forth, in the dialog box window. Wildcards * are available for class names, field names, and so on. Figure 4 illustrates the dialog box for specifying a pointcut for selecting join points of the assignment to a field y in Point class through a dialog box. Information about a specified pointcut is displayed in the Bugdel view. Moreover, the positions of the shadow of the selected join points are visually indicated by markers on the source file displayed in the Bugdel editor. This helps users recognize when an advice will be executed at runtime.

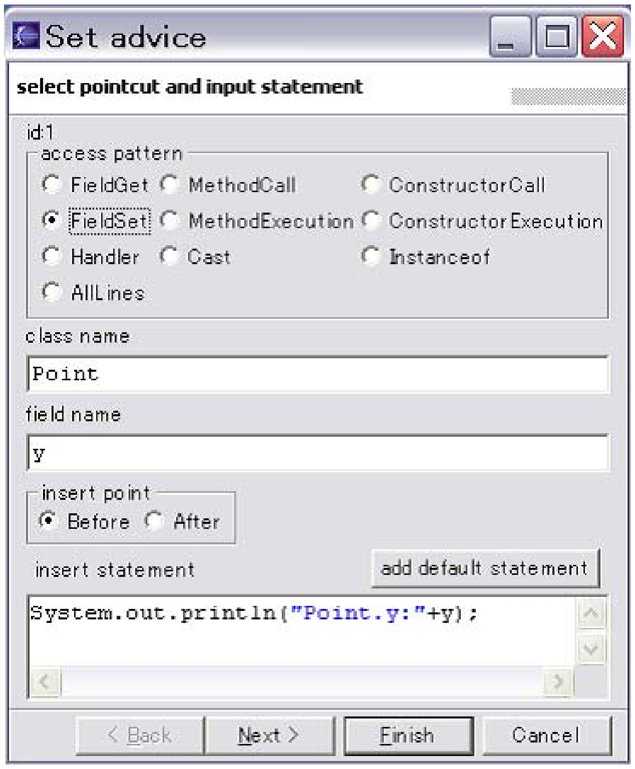


```
public class Point {
    int x;
    int y;
    void
    
    }
    void
    
    }
```

Figure 3. Candidates of the pointcut

3.2 Pointcut

Bugdel provides not only the pointcut designators of AspectJ but also the line pointcut and the allLines pointcut (Table 1). The line pointcut selects a join point that is the beginning of the specified line in a source file. To declare a line pointcut, users select "line pointcut" in a pop-up menu shown when the mouse cursor is on the ruler (on the left border of the editor). The line under the mouse cursor is selected as a join point. Alternatively, the line pointcut can be specified by double-clicking on the line that the users want to select. As a breakpoint, the line pointcut cannot select a blank line, an empty statement, or only a declaration


Set advice

select pointcut and input statement

id:1

access pattern

FieldGet MethodCall ConstructorCall [✓ FieldSet] MethodExecution ConstructorExecution Handler Cast Instanceof AllLines

class name: Point

field name: y

insert point: [✓ Before] After

insert statement

```
System.out.println("Point.y:"+y);
```

Figure 4. Advice dialog

statement. The line must include executable statements.

The use of line pointcuts may break the modularity and the encapsulation of classes since the line pointcut heavily depends on the implementation of a target class. The line selected by the line pointcut may move to another position or disappear if the method body including that line is modified. However, Bugdel's line pointcut designator is useful for debugging in practice and enforcing encapsulation is not our primary concern. In order to reduce the dependency of a line pointcut to the target class, Bugdel automatically tracks the move of the lines selected by every line pointcut, by using the resource markers of Eclipse. For example, if a new line is inserted before the line selected by a line pointcut, the line number of that selected line is increased by one. Bugdel tracks this change of the line number so that Bugdel can keep selecting the same line.

Bugdel also provides the allLines pointcut designator. It selects all the lines within the body of a specified method. Each of the selected lines becomes a join point, and an advice body associated with the join points selected by an allLines pointcut is executed every time the Java virtual machine executes one of the selected lines, step by step. For example, suppose that the move method in the Point class is specified by an allLines pointcut. If an advice body associated with this pointcut prints the values of the fields of a Point object, the users can see the changes of the values of those fields while the move method is executed: the field values are printed for every executed line.

3.3 Advice

When a user selects a pointcut in the Bugdel view, a dialog box as shown in Figure 4 is popped up. The users can describe an advice body in the dialog box.

3.3.1 Accesses to local variables and private fields

In Bugdel, an advice body can access local variables visible at the join point. Furthermore, an advice body in Bugdel can also access private fields visible at the join point. These capabilities break the encapsulation and modularity of classes and thus AspectJ designers might have decided not to include this capability in AspectJ. However, Bugdel provides these original capabilities since they are useful for debugging and enforcing encapsulation and the modularity are not a primary concern of Bugdel. If an advice body wrongly accesses a variable that does not exist, Bugdel shows an error message, and warning maker is displayed in the Bugdel editor.

3.3.2 Reflection

An advice body can access a special variable named thisJoionPoint. This variable is for reflection. An advice body can access the execution contexts of a join point through this variable. We below show some of the fields of the object that thisJoionPoint refers to.

- thisJoinPoint.target

  This represents the target object if the pointcut includes fieldSet, fieldGet, or methodCall.

- thisJoinPoint.line

  The line number of the source code corresponding to the join point.

- thisJoinPoint.filePath

  The path name of the source file including the join point.

- thisJoinPoint.variables

  The list of the local variables and the fields visible at the join point. The fields are those of the object represented by this at the join point. The type of thisJoinPoint.variables is Object[][]. The value is an array of pairs of the name and the value of each variable and field. If the type of a variable (or a field) is a primitive type, the value included in the array is a wrapper object representing the primitive-type value. For example, the following advice code shows the list of variables:

Pointcut	Join points selected
FieldGet (field name)	Field reference
FieldSet (field name)	Field assignment
MethodCall (method name)	Method call
MethodExecution (method name)	Method execution
ConstructorCall (constructor name)	Constructor call
ConstructorExecution (constructor name)	Constructor execution
Handler (type name)	Exception handler execution
Cast (type name)	Cast execution
Instanceof (type name)	Instanceof execution
Line (line number)	Beginning of the line in a source file
AllLines (method name)	All lines in a method

Table 1. The pointcuts of Bugdel

```
Object[][] vs = thisJoinPoint.variables;
for(int i=0; i<vs.length; i++){
    String name = (String) vs[i][0];
    Object value = vs[i][1];
    System.out.println(name+" = "+value);
}
```

3.3.3 Built-in methods

The following methods can be called in an advice body.

- bugdel.Bugdel.openEditor(String host, int port, String filePath, int line)

  This method opens a source file with the Bugdel editor. It remotely connects to Eclipse through a socket and runs the Bugdel editor in Eclipse. host specifies the name of the host where Eclipse and Bugdel is running. port specifies the port number that Bugdel is listening on. filePath specifies the path name of the source file shown with the Bugdel editor. line specifies the number of the line that must be highlighted. For example, the following advice shows the source file including the join point.

```
String file = thisJoinPoint.filePath;
int line = thisJoinPoint.line;
bugdel.Bugdel.openEditor("localhost", 5555,
                                  file, line);
```

- bugdel.Bugdel.jump(int line)

  This method performs a transfer of control to the specified line number. The line number given by line must be statically determined at weaving time. For example, suppose that an advice body is executed at the join point of line number 10. If the advice body executes:

```
bugdel.Bugdel.jump(15);
```

  Then the lines from 10 to 14 are skipped and the program execution continues at line 15. The Bugdel weaver loads the resolving class files for verification. Thus if the line number given to the jump method is wrong, the bytecode verifier of the Java virtual machine throws an exception and Bugdel shows an error message at weaving time.

3.4 Weaving

The Bugdel weaver is implemented using a bytecode engineering toolkit Javassist [5], and is executed when the user selects “weave this file” or “weave all” in the Bugdel menu. “weave this file” performs weaving aspects with the class file corresponding to the source file currently opened. “weave all” performs weaving aspects with all the class files included in the current project.

Since the weaver of Bugdel produces class files in which aspects are woven, Eclipse or Bugdel are not necessary to run the debugged program with aspects. The debugged program can be run even on the Java virtual machine without the debug mode.

4 Breakpoint emulation

Bugdel can be used to emulate breakpoints on JVM that does not support a debug mode. To emulate, developers specify the following code in an advice body:

```
Object[][] vs = thisJoinPoint.variables;
printVariables(vs);
bugdel.Bugdel.openEditor(host, port, file, line);
showOKbutton();//block until OK button is pressed
```

This advice prints the values of the local variables visible at the join point. It also opens the Bugdel editor for displaying the source file including the join point. Then it calls the showOKbutton method to block until the developer presses the OK button. The locations of breakpoints are specified by pointcuts. If developers want to suspend the execution at several method entry and exit points, they can use the methodExecution pointcut. If they want to suspend at a specific line in a source file, they can use the line pointcut. If they want to emulate single step execution, they run the advice shown above at the join points selected by the allLines pointcut.

This feature of Bugdel is being used by a Java debugger of Hitachi Software Engineering Corp. This debugger is part of the commercial J2ME^{TM} execution environment named SuperJEngine® [10]. It integrates Bugdel for breakpoint emulation for their Java virtual machine although Bugdel itself is open source software freely available. Since their debugger is for controlling a program running on a remote machine, a specific implementation of the shwoOKbutton method in the advice shown above is provided by SuperJEngine® [10]. It blocks the program execution on the remote machine until the developers press a button on the host machine.

5 Related work

AspectJ [2] is a popular aspect-oriented language based on Java. Users of AspectJ describe aspects in the text form whereas the Bugdel users describe aspects through a GUI. Furthermore, since AspectJ is a general-purpose language, it does not provide specific pointcut designators, such as line and allLines that Bugdel provides. It does not allow an advice body to access local variables visible at the join point, either.

Typical debuggers [16] such as JDB included in Sun's JDK (Java Developers Kit) allows the users to set breakpoints so that they can interactively obtain the values of variables at runtime. However, to use such a debugger, the debugged program must run on a Java virtual machine under the debug mode. For example, to debug a servlet, which is a server-side program written in Java, the whole servlet engine must be launched in a debugger, which is often a complicated task. On the other hand, Bugdel does not need to run the Java virtual machine in the debug mode since it enables not interactive but trace-based debugging, and all necessary debug code is embedded in class files at weaving time. The trace-based debugging is useful in particular for debugging a multi-threaded program. It can also be used to emulate breakpoints and step execution on Java virtual machines that do not support a debug mode.

6 Conclusion

This paper presents our aspect-oriented programming (AOP) system named Bugdel, which is an Eclipse plug-in specialized for debugging. Users of Bugdel can easily specify aspects through a GUI. Bugdel also provides a few original AOP mechanisms that are not appropriate for general-purpose AOP systems but are necessary for debugging. This is because Bugdel is a domain-specific AOP system and hence the modularity or the reusability of aspects is not our primary concerns.

Bugdel is now a production-quality software and is distributed as open-source software on the Bugdel Home Page [3].

References

[1] AJDT: AspectJ Development Tools eclipse subproject, http://www.eclipse.org/ajdt/.

[2] AspectJ Project, http://www.eclipse.org/aspectj/. *The AspectJ website*.

[3] Bugdel Project, http://www.csg.is.titech.ac.jp/~usui/bugdel/. *Bugdel Home Page*.

[4] A. Charfi and M. Mezini. Using aspects for security engineering of web service compositions. In *Proceedings of the 2005 IEEE International Conference on Web Services, Volume I*, pages 59–66, July 2005.

[5] S. Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming (ECOOP 2000)*, LNCS 1850, pages 313–336. Springer Verlag, 2000.

[6] G. Duzan, J. P. Loyall, R. E. Schantz, R. Shapiro, and J. A. Zinky. Building adaptive distributed applications with middleware and aspects. In *Aspect-Oriented Software Development (AOSD 2004)*, pages 66–73. ACM Press, 2004.

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

[8] S. Fairbrother, D. Kehn, J. Kellerman, P. McCarthy, and J. D'Anjou. *The Java Developer's Guide To Eclipse*. Addison-Wesley, October 2004.

[9] E. Gamma and K. Beck. *Contributing To Eclipse: Principles, Patterns, And Plug-Ins*. Addison-Wesley, November 2003.

[10] Hitachi Software Engineering Corp, http://www.hitachi-sk.co.jp/Products/SuperJ/. *SuperJEngine®*.

[11] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE 1997)*, LNCS 1343, pages 249–256. Springer, 1997.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *European Conference on Object-Oriented Programming (ECOOP 2001)*, LNCS 2072, pages 327–353. Springer, 2001.

[13] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP 1997)*, LNCS 1241, pages 220–242. Springer, 1997.

[14] C. V. Lopes and G. Kiczales. Recent developments in aspectj. In *Aspect-Oriented Programming workshop at ECOOP'98*, LNCS 1543. Springer, 1998.

[15] A. Mendhekar, G. Kiczales, and J. Lamping. *RG: A Case-Study for Aspect-Oriented Programming*. Xerox Palo Alto Research Center, 1997. Technical report SPL97-009 P9710044.

[16] Sun Microsystems, Inc, http://java.sun.com/products/jpda/. *Java Platform Debugger Architecture (JPDA)*.