Aspectual Comprehension: Program Understanding Using Aspects*

David R. Kaeli Sergei Kojarski David H. Lorenz Darren Ng
Institute for Complex Scientific Software
Northeastern University
Boston, Massachusetts 02115 USA
{kaeli@ece,kojarski@ccs,lorenz@ccs,dng@ece}.neu.edu

Abstract

Aspect Oriented Programming (AOP) is a powerful reflective programming tool. In this paper we discuss how AOP can be used to facilitate the process of program understanding. We call this process "aspectual comprehension." We analyze the characteristics of aspectual comprehension on three bodies of code. The first is a sizable third party Java legacy system for manipulating and displaying protein sequences entitled Friend. The second is Eclipse, an open source Java IDE. The third is Compress, a SPEC JVM98 Java benchmark. We study uses of the AspectJ AOP language to expose both dynamic and static software characteristics. Examples provided are actual code and data from re-engineering the first body of code—the Friend system.

1 Introduction

Maintenance of software systems inevitably rely on a understanding of the program structure. However, it is not uncommon for programmers to inherit and work with unfamiliar code. Therefore, a majority of software maintenance time is spent on program comprehension [26].

Program comprehension is the process of understanding a program through feature and documentation analysis. Studies and experiments [14] reveal that the success of decomposing a program into effective mental models depend on one's general and program-specific domain knowledge. While a number of different models for the cognition process have been proposed most models fall into one of three categories: top-down comprehension [1], bottom-up comprehension [2], and a hybrid model.

The top-down model is traditionally employed by programmers with code domain familiarity. By drawing on their existing domain knowledge, programmers are able to efficiently reconcile application source code with system goals. The bottom-up model is often applied by programmers working on unfamiliar code. To comprehend the application, they build mental models by evaluating program code against their general programming knowledge. Finally, the integrated or hybrid model reflects a combination of the previous two modeling techniques. It is commonly utilized for analysis of large applications.

Program comprehension, especially for large systems, requires the inspection of a plethora of application attributes such as dynamic call graph, source code, and documentation. For programs with an abundance of classes, the organization of these program characteristics presents a complex and problematic task.

Many tools strive to address the organization problem by analyzing and categorizing the data into meta-information [22, 3, 4, 5, 6]. Tools (e.g., [10]) also provide visualization of different perspectives of the program and its execution. However, today's tools are limited by:

- *User experience* - Studies in program comprehension revealed that expert programmers spent a majority of their time learning an unfamiliar programming environment than deciphering the target program itself [14].
- *Flexibility* - Tools provide only a fixed set of functionality that may not be sufficient for a task.
- *Expressiveness* - A tool's ability to interpret the user's command. Often, a tool has to balance between ease of use and expressiveness [21]. That's why some experts prefer powerful text based tools rather than simple to use visual tools.

In this paper we propose the use of Aspect-oriented programming (AOP) [13] as a program comprehension tool. AOP is a new programming paradigm that allows crosscutting concerns to be modularized [20]. We show how aspects provide programmers with a methodology to rapidly

*Supported in part by the National Science Foundation (NSF) under Grant CCR-0204432, and by the Institute for Complex Scientific Software at Northeastern University.

and easily reverse engineer software into understood models. Program compile- and run-time reflection using AOP benefit from an extremely flexible yet simple language construct absent from many of today's profiling tools.

We discuss AspectJ [18, 12], the mainstream Java implementation of AOP, as a program understanding tool. We show that fundamental terms of the language (e.g., join point, advice, pointcut) allow programmers to easily express requests for dynamic program meta data. We illustrate simple yet powerful aspects that expose, filter, and detail a program call graph extensively. Furthermore, we explore compile-time aspects (declare warning) for static program browsing. We detail how aspects go beyond Java Core Reflection in exposing meta-information located within method bodies. We assert that compile-time aspects are practical for solving a number of tasks, such as orphaned or "dead" code identification, class hierarchy analysis, and style-rule checking. We believe that AOP methods can be easily adapted by experienced programmers who desire more insight into a program's execution.

Our choice of AOP as a program comprehension tool was motivated by the following reasons (Table 1):

- *User language familiarity* - The AOP language is normally implemented as an extension to a base object-oriented language. As a result, the programmer/maintainer of an application is already familiar with the AOP language syntax and context.
- *Flexibility* - AOP provides the user with total control over an aspect profiler. Unlike other programming tools, programmers can tailor the profiling aspect as they see fit.
- *Expressiveness* - In addition to being highly flexible, AspectJ allows users to easily target data in both a static and dynamic program environment.

1.1 Outline

In Section 2, we elaborate on our three test cases. In Section 3, we consider how best to apply compile-time aspects to perform maintenance and reverse engineering tasks. In Section 4, we show how AspectJ can be used to expose the dynamic call graph of a program. We provide examples of simple aspects that allow programmers to reflect, filter and select run-time information. Section 6 concludes the paper and discusses other potential maintenance tasks using AspectJ.

2 Case Studies

This paper reports on using aspectual comprehension to understand three bodies of code. The first is a legacy system called Friend. The second is Eclipse, an open source Java IDE. The third is Compress, a SPEC JVM98 Java benchmark.

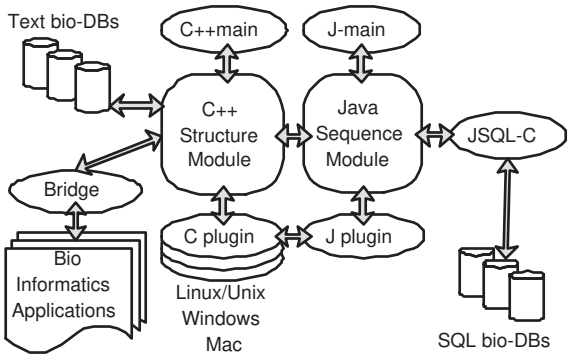

Figure 1. The Friend Software

2.1 The Friend System

Friend is short for an "**I**ntegrated Analytical **Fr**ont-**End** Application for Bioinformatics" and was developed by the Northeastern University Biology Department. The top level software diagram of Friend is shown in Figure 1. Friend was designed to aid scientists interactively visualize proteins and their interactions along multiple alignments, domains, fragments, and binding sites in a 3-D environment [7].

The Friend system was written in Java in the early 1990s and has been maintained by various programming groups during its lifetime. With little to no documentation, the Friend program is a classical example of a legacy product that is difficult to maintain and almost impossible to evolve. Our initial knowledge of the Friend system is summarized in Table 2.

To help us understand Friend we used AspectJ.

Package/Dir	files	classes	code lines
jalview	138	140	26742
jalview.parsers	20	40	3550
aliface	11	16	8341
friendMain	3	5	380
friendmenu	35	79	3937
friendcommon	4	5	290
friendblast	27	39	3389
libJMF	14	14	1757
TOTAL	255	338	48386

Table 2. The Friend system

Perspective	Aspectual Comprehension	Program Visualization
client	programmer	user
activity	program	view
domain	same language	visual presentation
proficiency	expert	intermediate
expertise	Java/AspectJ	interpreting views

Table 1. Aspectual Comprehension versus Program Visualization

2.2 Eclipse

Eclipse is an extendible software integrated development environment (IDE). Third party visual (i.e., views, menus, property pages etc.) and non-visual (i.e., builders, compilers etc.) components interact with the Eclipse application programmer's interface (API) to augment the IDE functionality.

Although the Eclipse architecture is well-designed and contains a feature rich API, the system and its interface are not sufficiently documented. The creator of Eclipse extensions or plug-ins is often overwhelmed programming for the complex API. As a result, developers routinely study existing plug-in source code to augment their Eclipse IDE programming proficiency.

We propose the use of aspectual comprehension techniques in the development of plug-ins for Eclipse. In our test case, the DAJ plug-in introduces the DemeterJ traversal language to the IDE. The central component of the plug-in is the DAJ project builder. To understand how Eclipse builders are implemented, we studied the existing Eclipse AspectJ plug-in. Due to the plug-in's entangled code however, source code analysis did not yield compelling results. To further facilitate the understanding process, we employed an AOP-based comprehension strategy. Using run-time and compile-time aspects we managed to reverse engineer the builder architecture and successfully completed our DAJ plug-in.

2.3 Compress - A Java Benchmark

Compress is a Java application that is part of the SPEC JVM98 benchmark suite. It is based on a modified Lempel-Ziv compression method (LZW) that replace common data substrings with variable size code [25]. In JVM98, *Compress* is executed on a variety of test data files and the total benchmark run-time is recorded. The *Compress* results factor into the overall SPEC JVM98 performance measurement.

The execution characteristics of the SPEC JVM98 benchmarks have been well studied [9]. However, program understanding is more easily deduced from the program's structure and method interactions than from its low-level instructions. We used dynamic and static aspectual comprehension techniques to decipher the inner framework of *Compress*.

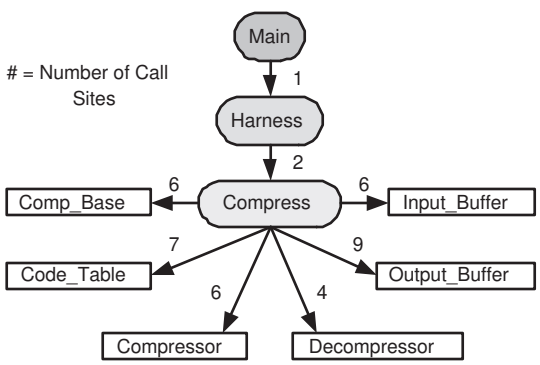

Figure 2. Compress Call Hierarchy

Figure 2 details the class interactions in the *Compress* application. Call sites, discussed in more detail in Section 3 track the inter-dependencies between classes. In Figure 2 for example, the `Compress` class at compile time calls the `Code_Table` seven times.

Aspects also allow the programmer to capture the dynamic call graph of the target program. A segment of the *Compress* call graph behavior captured by an aspect is shown in Listing 1.

From the call graph, it is evident by the sequence of methods calls that the Compress algorithm is updating its code and hash tables as it compresses data from the `Input_Buffer`. Afterwards, the compressed data is stored in the `Output_Buffer`. Once the `Output_Buffer` is full, the `Output_Buffer` putbyte method is executed to drain the buffer data.

3 Static Analysis

AspectJ allows compile-time aspects which can raise textual warnings or a compile error if targeted program characteristics are found in the code. This powerful mechanism can extract important information from the program source and enforce proper software engineering principles.

Listing 1. Segment of Compress call graph

```
...
execution(void _201_compress.Code_Table.set
    (int,int)
execution(void _201_compress.Compressor.
    Hash_Table.set(int,int)
execution(int _201_compress.Input_Buffer.
    getbyte()
execution(int _201_compress.Compressor.
    Hash_table.of(int)
execution(void _201_compress.Output_Buffer.
    Output(int)
 execution (void _201_compress.
     Output_Buffer.putbyte(int)
 execution (void _201_compress.
     Output_Buffer.putbyte(int)
...
```

Listing 2. aspect SequenceClients

```
package aspects;
aspect SequenceClients {
 pointcut Scope(): !within(aspects..*) &&
      !within(jalview.Sequence+);
 pointcut profile(): Scope() &&
   call(* jalview.Sequence.*(..));
 declare warning: profile(): "Sequence
     method call";
 }
```

3.1 Identifying Orphaned or "Dead" Code

Legacy systems that are maintained and extended by various programmers, usually contain a significant percentage of orphaned or "dead" code. These "dead" pieces contaminate class interfaces with multiple unused methods. Identification of obsolete methods is no easy task and often requires extensive analysis of the system code.

One of the Friend interfaces, `jalview.Sequence` defines a basic residue sequence abstraction that is the foundation of all data types such as RNA, Amino Acids, and Micro-arrays in the Friend system. The interface initially defined 120 methods. To locate the methods used by client classes and those unemployed, we created the compile-time aspect shown in Listing 2. The aspect detects all call sites to the `jalview.Sequence` interface used by its clients (not implementors). Each encountered call site event prints a message similar to the one shown below:

```
[ajc] ../jalview/AlignFrame.java:555:68:
      Sequence method call (warning)
[ajc] String newstr = AlignSeq.extractGaps(
     " ",ap.align.ds[i].getSequence());
```

Using the results gathered from the compile-time aspect, we identified and safely removed 50 unused `Sequence` methods out of 120.

In addition, tracking the method usage output allows the detection of program hot spots that can be later singled out for optimization.

3.2 Identifying Class Scope

Besides interface cleaning, aspects similar to the `SequenceClients` aspect can be used to reveal the class scope within the system, i.e., list of class clients. Moreover, compile-time aspects also illustrate the degree of coupling between an interface and each of its clients. For example, the `SequenceClients` aspect revealed that the interface is used by 54 classes in 5 system packages in 607 call sites. However, only 8 packages are tightly coupled with the interface. The other packages contain less than 20 (most less than 9) call sites targeting `Sequence` (Table 3).

The `SequenceClients` aspect exposes the Sequence class clients and allows the programmer to evaluate the cost of its maintenance. The more clients a class has, the more expensive it is to maintain. On the contrary, if a class has few clients, the decision to change, remove, or augment the class can be more easily evaluated.

3.3 Subtyping Relations

Compile-time aspects can also be utilized in categorizing subtype relations. Consider the aspect in Listing 3.

Client class	Call sites
aliface.SkyInterface	81
jalview.Alignment	70
jalview.SeqPanel	49
jalview.parsers.MSPFile	38
jalview.AlignFrame	36
jalview.DrawableAlignment	33
jalview.JnetCGI	29
jalview.AlignmentPanel	22
OTHERS	121
TOTAL	607
CLIENT CLASSES	54

Table 3. The `Sequence` clients

Listing 3. aspect SequenceClasses

```
package aspects;
aspect SequenceClasses {
  pointcut jps(): staticinitialization(
      jalview.Sequence+);
   declare warning: jps(): "Warning";
}
```

The `SequenceClasses` aspect provides a list of implementors and subinterfaces of the `jalview.Sequence` interface. This information is crucial to determine how many classes will be affected by an "interface cleaning" operation. The output detailed below illustrates how the Friend application contained very few `Sequence` implementors. Therefore, modifications to the `Sequence` code can be easily navigated.

```
[ajc] jalview/BinarySequence.java:24:1:
[ajc] public class BinarySequence
[ajc] jalview/DrawableSequence.java:9:1:
[ajc] public class DrawableSequence
[ajc] jalview/MSPSequence.java:24:1:
[ajc] public class MSPSequence extends
    DrawableSequence {
[ajc] jalview/ScoreSequence.java:7:1:
[ajc] public class ScoreSequence extends
    DrawableSequence {
[ajc] jalview/Sequence_impl.java:8:1:
[ajc] public class Sequence_impl
```

3.4 Style Rules

So far we have used the **`declare warning`** AspectJ construct to output our results. The language also provides an alternative **`declare error`** construct that halts the compilation process if specified join points are found in the source code. **`declare error`** can enforce adherence to design rules during program maintenance. The most common rule in OOP requires client classes (with the exception of subclasses) to access class state via methods only. Consider an aspect that produces compiler errors if clients of the `jalview.Sequence_impl` class try to access its state directly without the use of helper methods. The aspect in Listing 4 may be included by the maintainer to prevent rule violation during code modification.

4 Dynamic Analysis

To understand the structure hierarchy of a program, the control flow model of the system can be extracted from its dynamic call graph [2]. In general, program execution

Listing 4. aspect RuleChecker

```
package aspects;
aspect RuleChecker {
  pointcut Scope(): !within(aspects..*);
  pointcut field_access(): get(*
      Sequence_impl.*) ||
                        set(* Sequence_impl.*);
  pointcut prohibit(): Scope() &&
      field_access();
  declare error: prohibit():
    "!!!Sequence_impl state access violation
        !!!";
}
```

is normally orthogonal to the program structure: a single task usually cross-cuts or traverses multiple program modules. By invoking user configured join points and point cuts, AOP reflection is effective in exposing the program's cross-cutting behavior.

Source code analysis is often employed by programmers to construct a program call graph. By mentally "executing" the software code, the programmer can generate a complete runtime picture of small program instances. However, the amount of information a programmer can simultaneously process severely restricts the target code size. For medium and large-sized systems which includes the Friend system, dynamic call graph generation is tasked to automated *profiling* methods. Profiling mechanisms provide useful abstraction of the source code by reflecting the methods that are actually executed.

4.1 Profiling

Profiling and logging are two well-known examples of AOP uses [18, 12]. AspectJ, the mainstream AOP extension for Java, provides an easy way to expose program runtime attributes. Consider the `DynProf` aspect we used to monitor and create an application's dynamic call graph (Listing 5).

The `DynProf` aspect profiles join points selected by the `profile` pointcut, which encapsulates all method and constructor executions outside the `aspects` package and subpackages. By eliminating profiling in the `aspects` package, we limit the profile trace data to only include events in the target program environment. As a result, our trace data is more concise and program execution is unimpeded by superfluous data monitoring. The **`around`** advice specifies profiling logic to execute "around" the `profile` point cut. In our case, the `level` instance variable is used to keep track of the current method level in the call graph. Logging is provided by the `log` method which converts **`thisJoinPoint`** and `level` arguments into a string representation before commitment into the log.

Although simple, the `DynProf` aspect practically constructs the control flow model of the system. Without any extra tools we were able to extract the complete call graph of the analyzed application.

The output of the aspect, however, is problematic to read. Loops in the program execution produce a large number of log messages. While contributing little to the program understanding, these repetitious messages severely hinder output readability. For example, a single open file operation in Friend produced a 18 MB log file containing 17.7 MB of loop-generated messages.

Listing 5. aspect DynProf

```
package aspects;
public aspect DynProf {
  pointcut Scope(): !within(aspects..*);
  pointcut profile(): Scope() &&
    (execution(*.new(..)) || execution
        (* *(..)));
  private int level=0;

  Object around(): profile() {
    level++;
    log(thisJoinPoint,level);
    Object result = proceed();
    level--;
    return result;
  }

  void log(JoinPoint jp, int level) {
   String message = "";
   while(level>0) {
    message=message+" ";
    level--;
   }
   message = message+jp.toString();
   System.out.println(message);
  }
}
```

4.2 Filtering

The readability of the call graph can be improved by filtering the loop output. Loops can be easily identified in the log file by their repeating output messages pattern. For example, the file open operation output discussed earlier produced two distinct patterns:

```
...
public String Sequence_impl.getSequence()
public void DrawableSequence.
    setResidueBoxColour()
...
```

and

```
...
public int Alignment.maxLength()
 public int DrawableSequence.length()
  public int Sequence_impl.length()
 public String DrawableSequence.getSequence
     ()
  public String Sequence_impl.getSequence()
 public String DrawableSequence.getSequence
     ()
  public String Sequence_impl.getSequence()
...
```

The first pattern was created by the control flow loop of the `public DrawableSequence(Sequence)` constructor. The second pattern is contained within the `jalview.Alignment.findQuality(int, int)` method.

Control flow loops are an inevitable consequence of any structured program. As such, we propose an elegant and efficient solution to the logging problem using AspectJ filtering.

The `cflow` and `cflowbelow` pointcut designators allow the programmer to specify subtrees of the call graph (i.e. loops) to be avoided. Additional filtering can be enforced by specifying extra conditions in the `profile()` pointcut definition.

Listing 6. aspect DynProf (revised)

```
public aspect DynProf {
  pointcut profile(): Scope() &&
    (execution(*.new(..)) ||
     execution(* *(..))) &&
    !cflowbelow(
      execution(jalview.DrawableSequence.new
          (..)) ||
      execution(void jalview.Alignment.
          findQuality(int, int)))
   ...
}
```

4.3 Selective Profiling

The `DynProf` aspect can be improved by avoiding profiling loops. In our case, by bypassing loops, our log file size decreased from 18 MB to 300 KB. Similar techniques can also be used to achieve selective profiling. AspectJ allows programmers to focus on points of interest in the call graph by specifying additional conditions in the `profile()` pointcut definition. For example, to profile the control flow of the `aliface.SkyInterface.executeOneCommand(String)` method we write:

```
 pointcut profile(): Scope() &&
  cflow(* aliface.SkyInterface.
      executeOneCommand(String));
```

In general, the pointcut designators supported by AspectJ allow for a very precise profile targeting mechanism.

5 Related Work

5.1 Current Profiling Techniques

A majority of existing Java profiling applications invoke a customized instrumented Java Virtual Machine (JVM) or the experimental Java Virtual Machine Profiler Interface (JVMPI) to gather program runtime attributes [24]. Each of these profiling techniques are effective but not without their limitations.

Instrumented JVMs, as shown in Figure 3, are user modified virtual machines that generate special events at targeted program points [11]. For instance, a byte-code instrumented JVM can create method entry and exit events to track occurrences in a program execution.

By default, instrumented JVMs are cumbersome due to their required inclusion in each profiling tool instance [24]. They are also limited in configuration due to their highly customized nature. An unsupported profile feature requires the JVM to be painstakingly altered and reconfigured for the addition.

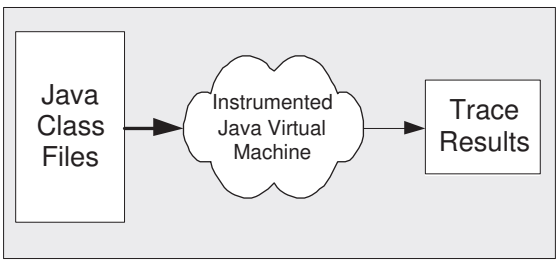

Figure 3. Instrumented Java Virtual Machine

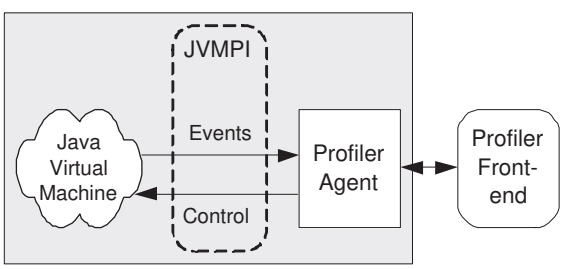

Figure 4. Java Virtual Machine Profiler Interface

Sun Microsystem's upcoming JVMPI mechanism which is integrated into all Sun JDKs version 1.2 and above, provides a standard Java profiling structure [23]. The JVMPI, shown in Figure 4, consists of two parts - 1) a profiler agent and 2) a profiler front-end. The profiler agent is a user-created Java application that interfaces with the built-in profiling "hooks" in the Java Virtual Machine. The profiler agent instructs the JVM on profile event types and times according to user specifications delegated through the profiler front-end. The profiler front-end also handles and manipulates the data captured by the profiler agent.

JVMPI profiling takes an all or nothing approach to data traces triggered by user specified events [8]. For example, the JVMPI will capture all class method entries and exits during a METHOD_ENTER or METHOD_EXIT event, even though the user is only interested in a particular class method behavior. It is left to the profiler agent's front end to remove irrelevant data among the mass of results. Also, since the JVMPI interface is a integral part of the Sun Microsystems' JVM, user alterations to the JVMPI to support new functionality such as new event types are not allowed.

AOP, shown in Figure 5, bypasses the limitations of an instrumented JVM and Sun's JVMPI by integrating a highly-configurable language profiling mechanism with direct access to a program's environment.

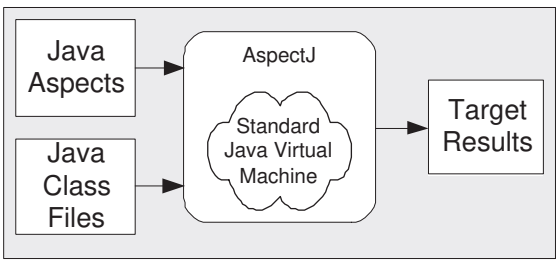

Figure 5. Aspect Profiling Framework

5.2 Aspectual Reflection and Unplugging Component using Aspect

In the process of understanding and re-engineering the Friend software, we are also applying AOP techniques to dynamically identify system design issues. It is our goal to re-engineer Friend to be more structured and maintainable.

By utilizing aspectual reflection [15, 16, 19] to support profiling methods and by selectively unplugging components using aspects [17], we have the tools to profile beyond traditional Java core reflection and the capability to perform controlled refactoring of legacy code. In addition to the immediate benefits obtained by improving a particular body of code, the application of aspects to program comprehension advances the understanding of AOP in which both success and failure yield important lessons.

6 Conclusion

In this paper, we showed how AOP can be effectively used for both dynamic and static program analysis. We also illustrated the usefulness of aspects to enforce program style rules from a program maintenance perspective.

This paper represents aspectual comprehension, validating the use of AOP as a feasible profiler. Aspect Oriented Programming provides the following advantages:

- *Expressiveness.* AspectJ is essentially a behavioral reflection tool that can be easily adapted to expose a program's dynamic execution profile. AOP's fundamental language component, a join point, can be viewed as a program's instruction evaluation abstraction. Evaluations such as method calls, field accesses, method body evaluations (execution), and object initializations, can be monitored and analyzed with almost unrestricted visibility via the join point model.

- *Crosscutting.* AOP's flexibility is due to the programmable aspect pointcut designators. Pointcut designators empower programmers to selectively target classes and methods of interest in a program's call graph. As a result, the readability of the profiler output increases due to the absence of irrelevant trace data which leads to a deeper and quicker understanding of the program.

- *Programmer-oriented.* Most of all, the AOP language is within the same language domain as the program being studied. For instance, aspects woven into a Java application are created in the Java language with AspectJ. A profiler programmed in the same language construct as a target application allows the maintainer/programmer to utilize their coding expertise in the profiler learning process. On the contrary, visualization tools that utilize non-standard protocols and different language domains often require experience and special skills to be used efficiently.

Experienced programmers should also appreciate the intimate access to a program's underlying code via AOP reflection that other profiling methodologies lack.

References

[1] Elliot Soloway and Beth Adelson and Kate Ehrlich. Knowledge and Processes in the Comprehension of Computer Programs. In *The Nature of Expertise*, Eds. M. Chi, R. Glaser, and M. Farr, pages 129-152, 1988.

[2] Nancy Pennigton. Comprehension Strategies in Programming In *Empirical Studies of Programming: Second Workshop*, pages 100-112, 1987.

[3] Kenny Wong. Rigi User's Manual. `http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf`.

[4] Michael W. Godfrey. Practical Data Exchange for Reverse Engineering Frameworks: Some Requirements, Some Experience, Some Headaches. In *ICSE 2000 Workshop on Standard Exchange Format*, 2000. `http://plg.uwaterloo.ca/~migod/papers/wosef00.pdf`.

[5] B. Kullbach and A. Winter and P. Dahm and J. Ebert. Program Comprehension in Multi-Language Systems In *Proceedings of the 5th Working Conference on Reverse Engineering 1998 (WCRE '98)*, pages 135-143, Los Alamos, 1998.

[6] Jrg Czeranski and Thomas Eisenbarth and Holger Kienle and Rainer Koschke and Daniel Simon. Analyzing xfig Using the Bauhaus Tool. *Working Conference on Reverse Engineering*, November 23-25, Brisbane, Australia, pages 197-199, IEEE Computer Society Press, 2000.

[7] A. Abyzov, C. Leslin, and V. Ilyin. Friend - an integrated analytical front-end for bioinformatics. In *Computational and Systems Biology at MIT (CBSi)*, 2003. `http://mozart.bio.neu.edu/friend/poster_jan2003.html`.

[8] P. Bellavista, A. Corradi, and C. Stefanelli. Java based online monitoring of heterogeneous resources and systems. Technical report, Universita di Bologna, 2000.

[9] K. Bowers and D. Kaeli. Characterizing the spec jvm98 benchmarks on the java virtual machine. Technical report, Northeastern University, Dept. of ECE, Computer Architecture Group, 1998.

[10] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 219–234, Santa Fe, New Mexico, Apr. 27-30 1998. USENIX Association, 2560 Ninth Street, Suite 215 Berkeley, CA 94710 USA.

[11] M. Dmitriev. Application of the hotswap technology to advanced profiling. Technical report, Sun Microsystems Laboratories, 2002.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[14] A. J. Ko and B. Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. In *11th IEEE International Workshop on Program Comprehension (IWPC'03)*, pages 175–184, Portland, Oregon, USA, May 10-11 2003.

[15] S. Kojarski, K. Lieberherr, D. H. Lorenz, and R. Hirschfeld. Aspectual reflection. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Boston, Massachusetts, Mar.18 2003. AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, ACM Press.

[16] S. Kojarski and D. H. Lorenz. Reflective mechanisms in AOP languages. Technical Report NU-CCIS-03-07, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Mar. 2003.

[17] S. Kojarski and D. H. Lorenz. Unplugging components using aspects. In J. Bosch, C. Szyperski, and W. Weck, editors, *ECOOP 2003 Eighth International Workshop on Component-Oriented Programming*, 2003.

[18] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 398–401. Workshop Proceedings, Brussels, Belgium, Springer Verlag, July 20-24 1998.

[19] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Portland, Oregon, May 1-10 2003. ICSE 2003, IEEE Computer Society.

[20] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming*, number 2743 in Lecture Notes in Computer Science, pages 2–28, Darmstadt, Germany, July21-25 2003. ECOOP 2003, Springer Verlag.

[21] A. Repenning and J. Ambach. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. pages 102–109.

[22] S. C. Richard Wheeldon and K. Keenoy. Using memex-like trails to improve program comprehension. In *2nd Annual Designfest on Visualizing Software for Understanding and Analysis of Software (VISSOFT)*, Amsterdam, Sept. 22 2003.

[23] Sun Microsystems. The java virtual machine profiler interface, 2003. `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`.

[24] D. Viswanathan and S. Liang. Java virtual machine profiler interface. In *IBM Systems Journal*, volume 39, pages 82–95, 2000.

[25] T. A. Welch. A technique for high performance data compression. In *IEEE Computer*, volume 17-6, pages 8–19, June 1984.

[26] S. S. Yau and J. Collofello. Some stability measures for software maintenance. In *IEEE Transactions on Software Engineering*, volume SE-6, pages 545–552, 1980.