A Comparative Analysis of Meta-programming and Aspect-Orientation

Suman Roychoudhury, Jeff Gray, Hui Wu, Jing Zhang, Yuehua Lin
{roychous, gray, wuh, zhangj, liny}@cis.uab.edu

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama
http://www.gray-area.org/Research

ABSTRACT

This paper presents an investigation into language constructs for supporting improved separation of crosscutting concerns. Traditionally, this separation has been performed using meta-programming and other related techniques. A growing area of research, called aspect-oriented software development, offers a new approach. We describe several distinctive characteristics of the two approaches with respect to their ability to modularize crosscutting concerns. The paper also reports on a survey that was conducted to assess software developers' general intuition relating to the comprehensibility of these techniques. Our initial research suggests that aspect-orientation offers several improved capabilities for realizing important software engineering principles.

1. INTRODUCTION

Even though the general notion of separation of concerns is an old idea, one can witness the nascence of a research area devoted to the investigation of new techniques to support *advanced* separation of concerns. It has been recognized by numerous researchers that the software modularization constructs developed over the past quarter-century are sometimes inadequate for capturing certain types of concerns [2, 4, 9]. This has serious consequences with respect to modular composition.

As noted in [9], previously defined modularization constructs are most beneficial at separating concerns that are orthogonal. However, these constructs often fail to capture the isolation of concerns that are non-orthogonal. Such concerns are said to be crosscutting and their representation is scattered, and tangled among the descriptions of numerous other concerns. Two concerns crosscut if the representation of one concern intersects the representation of another. Crosscutting concerns are denigrated to second-class citizens in most languages (i.e., there is no explicit representation for modularization of crosscutting concerns). As a result, crosscuts are difficult to compose and change without invasively modifying the description of other concerns (i.e., crosscuts are highly coupled with other concerns).

For example, in a very large system, logging method entrance/exit calls may lead to scattering of the logging concern across the object or functional boundaries. This may introduce unnecessary noise into the system resulting in poor modularity. Most operating system code provides a good repository for observing the effects of crosscuts, where examples such as prefetching and disk quota operations have been shown to be difficult to modularize using traditional object-oriented languages [1].

During the early part of the last decade, separation of crosscutting concerns was attempted using meta-programming, reflection, and other related techniques [2, 4, 8]. Such techniques provided a separation of the traditional *base* part (which describes the application) and the *meta* part (which describes reflective information and adaptations to the underlying base semantics). Although meta-programming provides a powerful mechanism for adaptation and concern separation, it often does so by sacrificing comprehensibility. That is, the conceptual intention of a concern can be difficult to discern in the presence of meta-level adaptation.

Recent research efforts, under the name of Aspect-Oriented Software Development (AOSD), are exploring fundamentally new ways to carve a system into a set of elemental parts in order to support crosscutting concerns. The goal is to capture crosscuts in a modular way with new language constructs called *aspect*s [4]. In AOSD, a translator called a *weaver* is responsible for taking code specified in a traditional programming language, and additional code in an aspect language, and merging them together.

In this paper, we describe distinctive characteristics between reflective meta-programming techniques and the new capabilities offered by aspect-oriented programming (AOP). We compare and contrast their capabilities with respect to their ability to modularize crosscutting concerns to satisfy sound principles of software development (e.g., the "Parnasian" benefits of changeability, independent development, and comprehensibility [5]). We compare the capabilities of OpenJava [8] (a compile-time meta-object protocol for Java) and AspectJ [4] (a general aspect-oriented language for Java). The paper also reports on a survey that was conducted that assessed software developers' general intuition relating to the comprehensibility of these techniques. Our initial research suggests that AOP offers several improved capabilities for realizing important software engineering principles.

2. BACKGROUND: OpenJava and AspectJ

"Meta" means that you step back from your own place. What you used to do is now what you see. What you were is now what you act on. Verbs turn to nouns. What you used to think of as a pattern is now treated as a thing to put in the slot of another pattern. A metafoo is a foo into whose slots you can put parts of foo." Guy Steele [7]

Smith defined *procedural reflection* as the concept of a program knowing about its implementation and the context in which it is executed [6]. A reflective system is capable of reasoning about itself in the same way that it can reason about the state of some part of the external world. Reflective systems must be causally connected; that is, manipulation of the internal representation structures directly affects the observable external behavior.

Reflective techniques play an important role for separation of concerns, permitting a program to be written with a higher-level of abstraction to support better modularization. Moreover, reflective systems may provide meta-objects for intercepting object behavior such as method execution, field access, method call [2, 6, 8]. Java provides its own set of reflective APIs to access an object's structure, invoke an object's method, and to load a class at runtime. However, Java's reflective API is a weaker form of reflection known as *introspection*. A form of reflection known as *intercession* provides additional reflective power to intercept as well as alter the object behavior.

2.1 OpenJava

OpenJava is a compile-time extension to Java which has static access to the data structures representing a program [8]. It produces an object representing a logical structure of a class definition for each class in the source code. This object is called a class meta-object. Programmers may customize the definition of the class meta-objects for describing adaptations to the structure of the base program. Figure 1 shows a simple example of OpenJava. In this example, the class `Foo` is a meta-class, which is inherited from OJClass. In OpenJava, every meta-class must inherit from OJClass, which is predefined in OpenJava. The method `translateDefinition`() is also inherited from OJClass, and is used to perform adaptation of the base class. In this example, `MyClass` is a regular Java base class. The *instantiates* clause declaration binds the meta-class `Foo` with the declared class object by creating an instance of this meta-class.

```
class MyClass instantiates Foo
 extends MyObject implements MyInterface
{...}

class Foo extends OJClass
{
  void translateDefinition() {....}
}
```

Figure 1. OpenJava Class Structure

Another representation of OpenJava adaptation is shown in Figure 2. This figure shows a clear separation of the *meta* part Foo from the *base* part *MyClass*. There is also a causal connection between the meta and the base class, i.e., any action performed by the base class is interpreted by its meta-object class. For example, a method call on any method of the base class can be trapped and corresponding changes in the behavior of the base program may be translated using the `translateDefinition`() method of the meta-object class. A typical translation mechanism, as performed by OpenJava, may be summarized as follows: -

1. Analyze the source program to generate a class meta-object for each class.
2. Invoke the member methods of class meta-objects to perform base-class adaptation.
3. Generate the regular Java source program reflecting the modifications made by the class meta-objects.
4. Execute the regular Java compiler (javac) to generate the corresponding byte code representation.

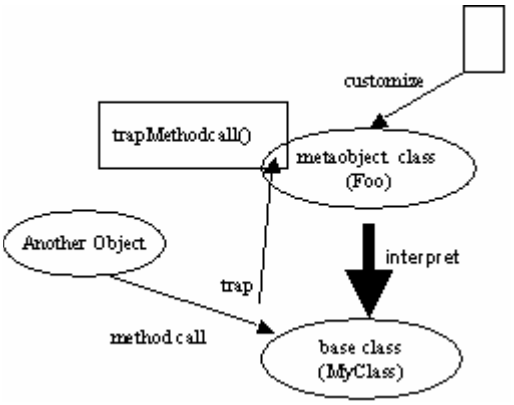

Figure 2. Base and Meta separation

2.2 AspectJ

A general aspect-oriented language for supporting separation of crosscutting concerns is AspectJ [4]. Similar to OpenJava, AspectJ is also an extension of regular Java, but unlike OpenJava it uses an AOP model to assist in separating concerns. A typical AOP model has 3 important elements [4]:

- the *join point model* (JPM): these are points in the runtime execution of a program (e.g., method call/execution, field get/set)
- *pointcuts*: the means for identifying join points
- *advice*: the means for specifying semantics at join points (e.g., before/after/around advice)The JPM terminology has several kinds of join points, for example, method call join points, method execution join points, field get/set joint point, exception handler execution join point.

A *pointcut* is a predicate that can match any given join point. For instance, an example of a primitive pointcut in AspectJ is:

```
call (void Line.setP1(Point))
```

This pointcut specifies a '*method call join point*' whose structure matches the given method signature (in this case, the `setP1` method that is defined in class `Line` that takes a `Point` argument and returns void).

An *advice* is an additional action to take at join points. In general, there are 3 kinds of advice - before, after and around. A before advice is invoked preceding the join point. An after advice is called after processing any computation under a join point, whereas an around advice wraps behavior at a join point.

A pictorial view of method call and method execution join points is shown in Figure 3. In this figure, two method call join points are pointed out (e.g., the call to the `add` method of `Figure`, and the call to `movedBy` on `myPoint`). The method execution join point for the `show` method of class `Display` is also highlighted. All of these join points represent points in the execution of the program that can be extended with advice in order to capture a crosscutting concern.

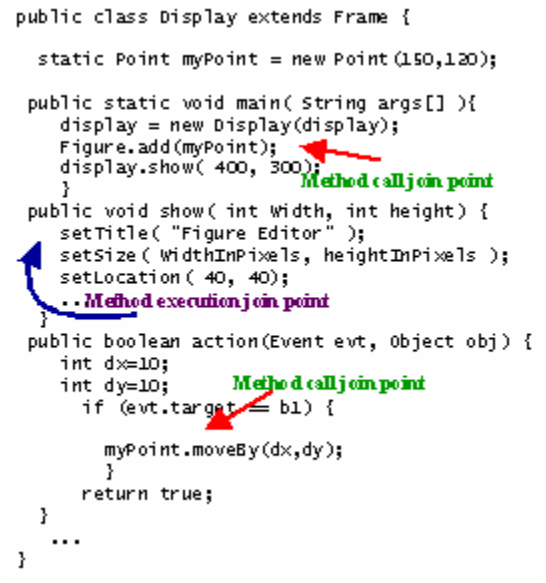

Figure 3. Join Points

In AspectJ, pointcut and advice specifications combine to form aspects. They play the critical role in encapsulating crosscutting concerns into single separated source and subsequent composition of the separated concerns into various structural elements of the base object. A typical example of an aspect to log method entrance and exit calls is shown in Figure 4. This code may look simple but primarily it isolates the logging concern into a single separated aspect, which can be eventually weaved into any system that requires logging.

```
aspect logEntXitCalls {

  before(): execution(* *.*(..)) &&
            !within (logEntXitCalls) {
    LogToOutputstream();
  }

  after (): execution(* *.*(..)) &&
            !within (logEntXitCalls) {
    LogToOutputstream();
  }

  void LogToOutputstream() {…}

}
```

Figure 4. An AspectJ Example

3. A Qualitative Analysis of Capabilities

This section introduces the constructs within OpenJava and AspectJ that support separation of crosscutting concerns. These features are at the core of both languages, but definitely they are not a complete overview of AspectJ and OpenJava. By comparing these key features we would like to illustrate the enhanced capability of one extension over the other. We start with comparing their semantics using a simple figure editor system shown in Figure 5 (this example is adapted from [4]).

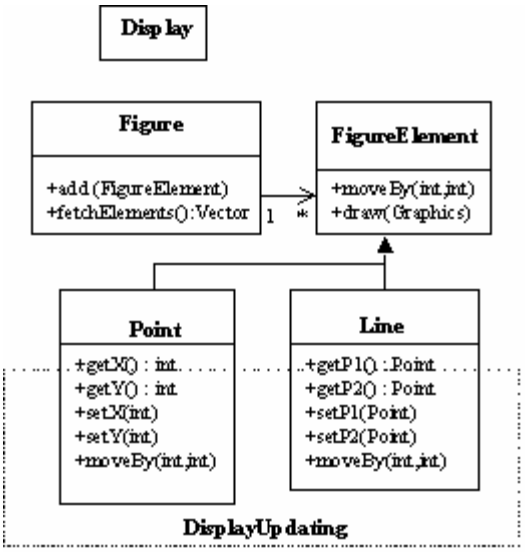

Figure 5. UML for Figure Editor (adapted from [4])

In this system, the `Figure` class is a container class, which consists of `FigureElements`, which can either be `Points` or `Lines`. A single `Display` class is used to view elements of the container class.

The system has a two level hierarchy with base objects like `Points` and `Lines` residing in the lowest level. The interface `FigureElement` defines a `moveBy` method that is implemented by each of these base objects. The `moveBy` method is used to shift the base object by a scalar depending on the input parameters. Moreover, the `Display` must also be updated or refreshed whenever a figure element moves.

A regular Java implementation of the `Point` class to meet this requirement needs an explicit call to the display update method (i.e., `Display.getContext().update`) in each of the methods that move a figure element. The Java implementation is illustrated in Figure 6. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case. The `Point` class, which previously seemed to exhibit good modularity, now finds itself to be coupled with the concern of the `Display` class. The `Line` object would also require similar crosscutting implementation (the crosscutting `DisplayUpdating` box in Figure 5 even shows this). This example problem might appear to be simple, but in a complex scenario with several hundreds of such classes being involved, this could severely damage the modularity of the system and increase the cost of maintenance.

```
public class Point implements FigureElement
  private int x=0, y=0;

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) {
    this.x = x;
    // cross-cutting concern
    Display.getContext().update();
  }

  public void setY(int y) {
    this.y = y;
    // cross-cutting concern
    Display.getContext().update();
  }

  public void moveBy(int dx, int dy) {
    setX(getX() + dx);
    setY(getY() + dy);
  }
…
}
```

Figure 6. Java Implementation Showing the Crosscutting Nature of DisplayUpdating

OpenJava, however, can address this problem by encapsulating the crosscutting part of the code into a single separated module. The following figure shows the OpenJava code, which does this separation. In this figure, the `translateDefinition()` function searches for setter methods in both `Point` and `Line` classes and adds the `Display.getContext().update()` statement after the set assignment in each matching. The OpenJava compiler inserts this separated concern back into the base class during compile time.

```
public class DisplayUpdating
       instantiates MetaClass extends OJClass
{

  public void translateDefinition() {
   statement stmt;
   OJMethod[] methods = getDeclaredMethods();
   for (int i=0; i<methods.length(); ++i) {
    if(methods[i].getName().equals("setX") ||
      methods[i].getName().equals("setY") ||
      methods[i].getName().equals("setP1") ||
      methods[i].getName().equals("setP2") ||
     {
      stmt=makeStatement("Display.getContext().update();");
      methods[i].getBody().insertElement(stmt, 1);
    }
  }
}
```

Figure 7. OpenJava DisplayUpdating Meta-object Class

Such a mechanism decouples the `Point` class from the concern of the `Display` class. Thus, the Java implementation of the base classes can be written with the `DisplayUpdating` concern factored out by instantiating the associated meta-object. This is shown in Figure 8.

```
public class Point instantiates DisplayUpdating
                  implements FigureElement
  private int x=0, y=0;

  public int getX() { return x; }
  public int getY() { return y; }

  public void setX(int x) {
    this.x = x;
  }

  public void setY(int y) {
    this.y = y;
  }

  public void moveBy(int dx, int dy) {
    setX(getX() + dx);
    setY(getY() + dy);
  }
…
```

Figure 8. Removal of DisplayUpdating Concern

However, such an approach introduces new drawbacks. As shown in Figure 8, OpenJava enforces the *instantiates* clause to be declared along with the base class declaration. Thus, the `Point` class and the `Line` class now need to add the *instantiates* clause along with their declaration. This may appear to be acceptable when the number of such base classes is few, but as the number increases, it induces an additional burden and cost on maintenance and changeability. An additional disadvantage is that the structure of the separated module does not explicitly tell us about the names of the base classes that are being affected by the crosscutting concern.

An alternative approach to the same problem using AspectJ produces an improved solution, which not only separates out the crosscutting element but also preserves the integrity of the base class (i.e., no structural changes are needed to the base). Implementing the "*display updating*" functionality using AspectJ is rather straightforward. Figure 9 depicts an AspectJ solution (adapted from [4]) of the same problem. The solution may appear to be similar but the structure of the crosscutting concern is captured more explicitly in AspectJ. The `move` pointcut clearly states the names of each method in each class that are involved in the crosscutting process, so the programmer can easily identify the part of the code that needs to be updated after each move (additional tool support for most major Java environments is also available to assist in understanding more complex aspects).

Moreover, this technique does not enforce the programmer to manually change each of the base classes to support separation of the crosscutting concerns. The AspectJ compiler can dynamically weave in all the separated concerns into the base classes at compile time.

In addition, this functionality is pluggable. For instance, if the *"display updating"* feature needs to be removed from the system (either because it is no longer needed, or because it is not needed in certain configurations), one needs to simply recompile the base classes *without* binding the aspect. However, in OpenJava this would require a programmer to manually visit each affected class and remove the *instantiates* clause (see the first line of Figure 8).

```
public aspect DisplayUpdating {
 pointcut move() :
  call(void FigureElement.moveBy(int, int) ||
  call(void Line.setP1(Point) ||
  call(void Line.setP2(Point) ||
  call(void Point.setX(int) ||
  call(void Point.setY(int));

  after() returning : move()
    Display.getContext().update();
  }
}
```

Figure 9. An Aspect to Control DisplayUpdating

Another distinctive advantage of AspectJ is the use of its wildcards. For instance, the `call` pointcuts (Figure 9) used in display updating of the `Point` and `Line` objects can be further reduced to a single primitive `call` statement; for example,

```
call(* *.set*(*))
```

The use of wildcards allows new figure elements like circle or rectangle to simulate movement *without* any change of the pointcut `move`.

Both OpenJava and AspectJ support an *implicit* form of invocation that leads to better separation of crosscutting concerns. This offers improved capabilities for independent development. As shown here, AspectJ has an advantage over OpenJava with respect to the ability to make changes to new concerns. Regarding the more subjective characteristic of comprehensibility, the reader is asked to compare Figure 7 and Figure 9 to arrive at their own opinion as to which technique is more comprehensible. We also conducted a survey to gain some additional understanding of the comprehensibility offered by each technique. This survey is described in the next section.

4. COMPREHENSIBILITY ASSESSMENT

In this section we will primarily analyze the comprehensibility of these two techniques in the form of a survey that was conducted to evaluate software developers' general intuition in understanding the language constructs and features of both OpenJava and AspectJ. The survey was handed out to 19 industry professionals and students in the Birmingham and Nashville areas. Each of these individuals had previous development experience using OOP techniques and Java.

The survey consisted of a set of four simple crosscutting problems with their corresponding analogous solutions in OpenJava and AspectJ. In addition to answering the questions, the participants were requested to indicate the time taken to comprehend each question in each of the sections. The actual survey can be found at http://www.gray-area.org/Research.

The first question showed how the behavior of execution of a method in a specific class could be altered using OpenJava and AspectJ. The survey participants were asked to predict the changes in the class behavior. The second question was an extension to the first one. The participants were asked to describe how the first question could be extended to the whole system. Question three tested the ability to comprehend meta-objects and aspects that were used for pre and post condition checks on method parameters. It also showed how the parametric values could be overwritten by the values supplied in the separated aspects and meta-objects. The last question was based on tracing name-based method patterns in a typical class object. Participants were asked to interpret the results surfacing from such behavioral patterns.

Although there are many other complex features that are individually supported by OpenJava and AspectJ, the goal of this survey was to evaluate the reactions of software developers to the new language constructs as provided by these two languages.

The following charts summarize the results obtained from the survey. Figure 10 shows the *accuracy level* of the group who participated in the survey. It indicates that almost *75%* of the participants were able to correctly predict AspectJ related problems whereas the accuracy level for OpenJava was around *60%.*

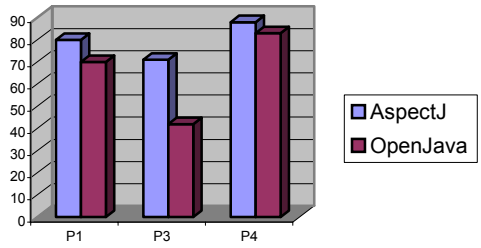

Figure 10. Accuracy Level of Participants

The next chart shows the average *response time* of the participants in comprehending each problem. This indicates that OpenJava based questions took nearly four minutes (on an average) to answer. The equivalent AspectJ based questions took half of the time to answer compared to OpenJava.

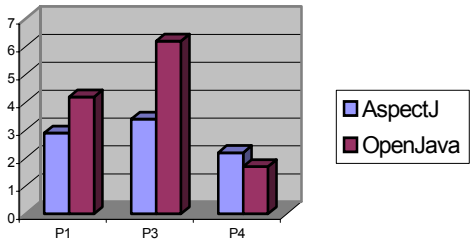

Figure 11. Response Time of Participants

Finally, we present the general view of the participants in understanding the constructs of these two languages. This is shown in Figure 12.

Almost *75%* of the participants' initial reaction was that AspectJ was easier to understand whereas only *13 %* were in favor of OpenJava.

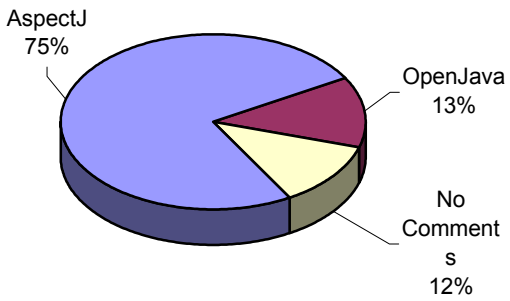

Figure 12. General Preference of Participants

5. CONCLUSION

"Program structure should be such as to anticipate its adaptations and modifications. Our program should not only reflect (by structure) our understanding of it, but it should also be clear from its structure what sort of adaptations can be catered for smoothly. Thank goodness the two requirements go hand in hand." E. Dijkstra

AspectJ and OpenJava are both compile-time extensions to Java that uses intercessional reflective techniques to alter the semantics or behavior of objects by separating crosscutting concerns in a modular implementation.

Aspects written in AspectJ are explicit and easy to comprehend. Moreover, the inherent modular characteristics of aspects enable easy plug-and-play functionality. In addition, the power of AspectJ is captured in the core language constructs making it possible for safer use. Furthermore, adoption of AspectJ into existing systems is relatively straightforward with negligible impact on the changed system. Maintainability of such systems is easier and the capability to adapt to future changes is significantly improved.

Although OpenJava also provides a powerful mechanism for adaptation and concern separation, it often does so by sacrificing comprehensibility. That is, the conceptual intention of a concern can be difficult to discern in the presence of meta-level adaptation. In addition, the presence of *instantiates* clause in every base object makes it difficult to perform an easy plug-and-play functionality. The language constructs provide greater power to the programmer, which may, however, result in unsafe use. Also, adoption of OpenJava into existing projects is relatively more complex and may require more time, suggesting a more cautious approach to adoption.

6. ACKNOWLEDGEMENT

This work is funded by the DARPA Information Exploitation Office (DARPA/IXO), under the Program Composition for Embedded Systems (PCES) program.

7. REFERENCES

[1] Yvonne Coady and Gregor Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code," *Second International Conference on Aspect-Oriented Software Development*, Boston, MA, March 2003.

[2] Krzysztof Czarnecki and Ulrich Eiseneker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[3] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[4] Gregor Kiczales, Eric Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold, "Getting Started with AspectJ," *Communications of the ACM*, October 2001, pp. 59-65.

[5] David Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972, pp. 1053-1058.

[6] Brian Smith, "Reflection and Semantics in Procedural Languages," Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[7] Guy Steele, "Growing a Language," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Keynote Address, Vancouver, British Columbia, Canada, October 22, 1998.

[8] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A Class-based Macro System for Java," in Reflection and Software Engineering (W. Cazzola, R. J. Stroud, and F. Tisato, eds.), LNCS 1826, Springer Verlag, 2000.

[9] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *International Conference on Software Engineering (ICSE)*, Los Angeles, California, May 1999, pp. 107-119.