DAJ: A Case Study of Extending AspectJ

John J. Sung, Karl Lieberherr
College of Computer Science
Northeastern University
Boston, MA 02115
{jser, lieber}@ccs.neu.edu

ABSTRACT
AspectJ is a tool that allows programmers to practice aspect-oriented programming (AOP) by using an extension of the Java programming language. However, it does not allow programmers to conveniently experiment with other paradigms such as Demeter or Composition Filters. We propose to extend AspectJ with minimal language extensions that allow programmers to experiment with other AOP paradigms besides AspectJ.

We explore the design space of extending the AspectJ programming language with the Demeter AOP paradigm. Specifically, we explore the conceptual integration between AspectJ and Demeter, and the different implementation possibilities for the new language extension. These explorations will give an overview of the issues involved in extending the AspectJ language.

We introduce DAJ, a small extension of AspectJ using the AspectJ declare mechanism. DAJ allows programmers to practice Demeter concepts by extending the AspectJ programming language with Demeter concepts such as class graphs, traversals, and visitors. DAJ promises to become the best implementation of the Demeter AOP ideas and it has a very small learning curve for AspectJ programmers.

KEYWORDS
Aspect Oriented Programming, AspectJ, DJ, DemeterJ, DAJ

1 INTRODUCTION
The aspect-oriented programming tool, AspectJ [1, 2], has become very popular with programmers to experiment with aspect-oriented programming concepts. However, AspectJ does not implement other AOP concepts such as Demeter [3, 4] or Composition Filters [6]. Addressing this shortcoming would greatly increase the rate of adaptation of AOP to the programming community at large. Having one software development package to download and learn would decrease the learning curve and simplify the software distribution process. Furthermore, having one package would facilitate conceptual discourse between AOP researchers.

In order to address the fore mentioned issues, we have explored different designs for extending the AspectJ Programming Language and its implementation, an approach similar to [7]. In this paper, we will present several possible ways to integrate an AOP concept to the AspectJ language. This is not a complete study, but only a case study of the methodology in which we integrated an AOP concept into AspectJ.

We will also explore the design space of implementing the language extensions that were discussed into AspectJ. Again, these explorations will yield possible implementations for language extensions to an existing compiler. In our case it will be the AspectJ compiler.

Next, we will describe DAJ [9] that implements a proposed methodology for extending the AspectJ language. DAJ integrates the Demeter concepts with AspectJ by using AspectJ, DemeterJ, and DJ. It is an implementation that provides AOP researchers a method of integrating their ideas into AspectJ via addition instead of modification.

DAJ is a sufficiently stable open-source research tool available on SourceForge and from the Demeter website and it is in the process of being used in a graduate course.

Finally, we conclude with an overview of the methodology that we have discussed in extending AspectJ. We also propose other possible research directions that other researchers may investigate further in extending AspectJ and developing DAJ.

2 EXTENDING ASPECTJ LANGUAGE
In extending the AspectJ language, we need to analyze the relationship between the extension and the AspectJ base language. How would the proposed extension interact with the existing AspectJ language? How should the programmer using AspectJ and its extensions think about solving the problem at hand? How should an AOP researcher experiment with the new extension to AspectJ? These are some of the questions that will need to be answered in order to understand how to extend AspectJ effectively.

In this paper, we analyze the metaphors in Demeter and AspectJ, and perform a translation experiment to understand those questions. The metaphorical analysis allows us to understand the conceptual interaction and how the programmer should think about the AspectJ language extensions when solving their problems. In contrast the translation experiment will allow us to understand the relationship between AspectJ and the new language extension.

In the translation experiment, a small example from an AOP tool will be translated to AspectJ. This translation allows us to find semantic overlap and compatibility between the extension language and the AspectJ base language. In this paper, we only translate to AspectJ and not vice versa, because we are interested in extending the AspectJ language.

All of these investigations will allow us to extend AspectJ with features that will make AspectJ more expressive. The new expressiveness of AspectJ should allow programmers to express programs in ways that they could not before with AspectJ. It should
also facilitate the integration of AOP concepts into the general public and the conceptual discourse between researchers.

In this section, we will first introduce the major concepts in Demeter, specifically DemeterJ. It will give a basic conceptual view of Demeter. This will allow us to analyze the metaphors used in Demeter and AspectJ. Following the metaphorical analysis, we present a translation experiment. In this experiment, we translate a DemeterJ example to AspectJ. We should gain some understanding of the relationship between AspectJ and Demeter in the translation process. The translation experiment with the conceptual analysis will tell us the compatibility between the concepts in AspectJ and Demeter.

2.1 Demeter Introduction

Demeter is a conceptual framework that has been developed at Northeastern University. Demeter abstracts the program as graph definitions and graph traversals. The main concepts of Demeter are class graph, strategy, visitor, and advice. The class graph is the schema of the data structures used within a program. It defines all the possible manifestations of object graphs created during a program execution. A strategy is a direction on how to traverse the object graph. Strategies such as "from Company to Employee" and "from Top via Middle to Bottom" combined with a class graph yields a traversal graph. This traversal graph is a sub-graph of the class graph that includes all possible paths defined by the given strategy. The visitor in Demeter is the agent that traverses the traversal graph. It has advice that is invoked for particular types of nodes in the traversal graph. Thus, a task within Demeter requires the programmer to specify at least the class graph, strategy and visitor. The traversal graph is not central to Demeter. However, a traversal graph may be substituted for the class graph and the strategy. This Demeter process is shown in Figure 1 below.

Figure 1: Demeter Process Overview

The mechanism in which the programmer specifies the three components: class graph, strategy and visitor are different for different implementation of Demeter paradigm. In one such implementation, DemeterJ, the class graph is described within the class dictionary. The syntax of the class dictionary is a modified Backus Naur Form (BNF) that allows programmers to describe the class graph efficiently. The visitor is declared within the class graph as well, since it is implemented as a Java class. The strategies and the definitions of the visitors are specified in behavior files.

We will go through the Basket Example to illustrate how DemeterJ is used to implement programs. The class graph of the Basket Example is shown in Figure 2. The class Basket may contain three objects, one of type Pencil and two of Fruit. A Fruit may also be an Orange. Every Fruit has a reference to the class Weight, which has an integer. An Orange has a reference to a Color object, which is includes a String class. In this example, we will sum the integers in the Weight objects that are contained within a Basket.

Figure 2: Class Graph of the Basket Example

In DemeterJ, we first need to specify the class graph in a class dictionary. We specify all of the relationships within the class graph using the modified BNF as shown in Figure 3. The identifier on the left is the class being defined. The "$=$" can be interpreted as "has-a" relationship in object oriented programming terminology. The identifiers within "$<$" and "$>$" are labels for these "has-a" edges within the class graph. The "is-a" relationship is defined by the ":" symbol. The "common" keyword indicates that the class Fruit has a "has-a" relation ship with Weight with label "$w$". In the classic case of inheritance, all of the classes that have "is-a" relationship with Fruit will inherit this "has-a" relationship. Lastly, the period ends the sentence within the class dictionary.

Figure 3: Class Dictionary of the Basket Example

Because of the way DemeterJ implements alternation classes as abstract classes, we need to change the way inheritance is expressed between Fruit and Orange to complete our example. We want the class Fruit to be a concrete class instead of an abstract class and we accomplish this by using "extends" instead of the ":" to express the inheritance. The modified class dictionary is shown in Figure 4. The "$\rightarrow$" for Fruit has changed to "$\Rightarrow$" and the string "extends Fruit" was added to the definition of Orange. This signifies that Orange inherits from Fruit, i.e. Orange is a Fruit.

Figure 4: Class Dictionary of Basket Example with Fruit as Concrete Class Instead of Abstract Class

Next, we need to specify the strategy for the traversal. Because we want to sum all of the integers in Weight objects within a Basket, we need to traverse from the class Basket to Weight. Therefore, the strategy becomes "from Basket to Weight". The resultant traversal graph from applying this strategy to the Basket Example class graph is shown in Figure 5.
Now that we have a traversal graph that we want to traverse, we need to define a traversal and a visitor that will sum the integers stored within each Weight object. We accomplish this by defining the traversal and the visitor in a DemeterJ behavior file. However, we have not actually declared the visitor class in the class dictionary. We accomplish this by adding the line shown in Figure 6 to the class dictionary. It defines a class CountWeightVisitor with a data member variable total of type int.

```
CountWeightVisitor = <total> int.
```

The definition of the CountWeightVisitor is similar to the traversal. We start with the scope of the behavior by having the name of the class CountWeightVisitor and "{". Then, we specify two methods and an advice. The two methods are start() and getReturnValue(). The start() method initializes the running total of the weights, while getReturnValue() returns the total weight counted. The definition of these methods is similar to Java, except for the double curly-braces. These double curly-braces are used to signify pure Java code to the DemeterJ weaver.

The advice specified within CountWeightVisitor is a before advice. It is executed before visiting all the "children" of the node Weight. Within the advice, there is a keyword "host", which is similar to "this" in Java. Instead of the current object, it refers to the object of type Weight we are currently visiting. The accessor method get_i() that is used in the advice is generated automatically for us by DemeterJ. This advice is executed every time we encounter an object of type Weight and sums the integer within Weight. The current running total is stored in the integer variable total.

This does seem like a lot of work, so the designers of DemeterJ created a way of specifying the strategy and the visitor at the same time using inlined visitors. This is useful for small traversals, such as accessor traversals that retrieve one object from a complicated object graph. In this method, we do not need to declare the visitor as a class in the class dictionary. This usage of the inlined visitor is illustrated in Figure 8 below.

```
Basket {
   public int getTotalWeight() to Weight {
      \{ int total = 0; \}
      \{ \}
      before Weight \{
         \{ total += host.get_i(); \}
      \}
   return int \{ total \}
}
```

The body of the method getTotalWeight is very similar to the CountTotalWeight visitor. The difference is that the method has the string "to Weight" to specify the strategy "from Basket to Weight". Next, we have the statement "\{ \} \{\} " , which defines and sets the running total of the weights for the inlined visitor. We specify the advice similar to the one in Figure 7. Then, we specify the return value total and the return type to be int. This inlined visitor is very useful if you do not have complicated visitors that are being reused.

The way in which the programmers invoke the traversal is the same in both traversal definitions. The traversals are invoked via a method call to Basket. In the first traversal definition we would call the method countTraversal with a CountWeightVisitor object as an argument. For the inline visitor traversal, the programmer would simply call the method getTotalWeight for a Basket object. This way of invoking traversals is consistent with how one would invoke methods in Java.
Another significant feature of DemeterJ is its ability to generate a parser for the class dictionary. As an example, we will convert the class dictionary in Figure 3 to read in a XML document and it can be used to create an object graph. In order accomplish this task, we add the appropriate XML tags around the statements in the class dictionary as shown in Figure 9 above. An example of a valid sentence for the grammar specified in Figure 9 is shown in Figure 10 below. The specified sentence in Figure 9 defines a Basket object with a Pencil and two Oranges. Each of the Orange has the string “orange” as the color and has weights 5 and 9 respectively.

Figure 10: Valid Sentence for the Grammar Specified in Figure 9

We have just introduced basic Demeter concepts and DemeterJ, an implementation of Demeter concepts. We have presented the DemeterJ Basket Example to illustrate the uses of the Demeter concepts using DemeterJ to code programs. This introduction should give a basic understanding of Demeter in order to analyze the metaphors used in AspectJ and Demeter.

2.2 Metaphorical Analysis

We will analyze the two metaphors being used in Demeter and AspectJ. The metaphors being used in the two are “Program as a Journey” and Join Point Model respectively. The Join Point Model (JPM) abstracts a program into join points of programming artifacts, such as classes, methods, member variables, etc. The JPM is further categorized into static and dynamic JPM. The static JPM deals with the data structure while the dynamic JPM deals with method calls. Because the static JPM is very similar in function to the class dictionary in DemeterJ, we will analyze the dynamic JPM.

The dynamic JPM is method centric, i.e. the join points tend to be artifacts related to method calls. Thus, the main concern of a JPM is the call graph and the class graph within a program. We visualize the method centricity of dynamic JPM in Figure 11. These join points are described by pointcuts and these in turn are used to specify the points in which the advice within an aspect should be invoked.

Figure 11: Visualization of JPM

In contrast to the dynamic JPM, ”Program as a Journey” metaphor used in Demeter is concerned with the data structure, i.e. class graph of a program. In DemeterJ, the programmer constructs a class graph through the class dictionary. Then, the programmer specifies the strategy in which a visitor should follow when creating the traversal graph. The advice, the descriptions of what to do at each node, is invoked during the traversal of the object graph. Thus, the class graph becomes the "world" in which the journey takes place, strategy becomes the instructions for the path in the journey and the visitor becomes the person who is on the journey that does some work along the way. This concept is visualized in Figure 12 below.

Figure 12: Visualization of Program as a Journey Metaphor Used in Demeter

Despite this contrast of the two metaphors, AspectJ and Demeter both have advice. JPM's advice is a description of a process to be executed when a specified join point is encountered during an execution of a program. However, Demeter advice is executed when the visitor visits certain nodes and edges within an object graph. These two ways of executing advice seem to be incompatible at first, until you consider how traversals are implemented in DemeterJ. DemeterJ combines the strategy and the class graph and creates method calls to traverse the object graph. Because this implementation is possible, we may use the same process to translate Demeter traversals to AspectJ as shown later.

2.3 Translation Experiment

As an example, we will implement the traversal as a result of applying the strategy "from Basket to Weight" to the Basket Example class graph from Figure 2. The traversal graph that we will translate is shown in Figure 5. In this translation process, we will use introductions to introduce method calls that will traverse the traversal
In this manner, we may translate Demeter style visitor to the class Weight. The method totalWeight() that we introduce to the class Basket is defined to initialize the running total, start the traversal, and return the running total. The pointcut designator t1Weight uses the pointcut for the Weight.t1() traversal method and the target() pointcut is used to pass the reference to the Weight object. Lastly, we add the integer of the Weight object to the running total in the before advice.

```java
aspect CountWeightVisitor {
  static int returnVal;

  int Basket.totalWeight1() {
    returnVal = 0;
    t1();
    return returnVal;
  }

  pointcut t1Weight(Weight weight) : (call(void Weight.t1()) && target(weight);
    before(Weight weight) : t1Weight(weight) {
      returnVal += weight.get_i();
    }
}
```

Figure 14: Demeter Style Visitor for Basket Example Implemented in AspectJ

In this manner, we may translate Demeter traversals and visitors using AspectJ features. This translation process was a fairly straightforward process. Because of this straightforwardness of the translation process, we should be able to create a program that will accomplish this automatically for us. Thus, we may create a preprocessor that will preprocess an AspectJ language extension to AspectJ language. Using the preprocessor we can create a new development environment that will use the preprocessor as the frontend and the AspectJ as the back-end.

In this section, we have introduced Demeter concepts and DemeterJ, analyzed the metaphors used in Demeter and AspectJ, and conducted translation experiments. From these experiments, Demeter concepts are describable in terms of AspectJ and DemeterJ traversals are translatable to AspectJ. Thus, Demeter and AspectJ are conceptually compatible and the concepts are implementable using AspectJ.

3 IMPLEMENTING ASPECTJ LANGUAGE EXTENSION

There have been several implementation options that we have explored for extending the AspectJ language. The first choice that we had to make was whether to modify the AspectJ compiler and to use the AspectJ compiler as the backend to our own development environment. The second choice was between using traditional OOP method of programming or attempt to use existing AOP tools. These two choices needed to be analyzed in order to make the most sensible decisions.

The first option of modifying the AspectJ compiler was not considered for very long. The major problem with this option is that the coordination between the releases of AspectJ and our language extension modifications become unmanageable. The AspectJ team is a fully funded team of highly trained experts that work on AspectJ while our tool would only be developed for research purposes and maintained by a graduate student. Thus, the amount of effort, quality, and rate of releases would be highly in favor of AspectJ and our tool would be hopelessly lag behind and outdated in respect to AspectJ very quickly. This synchronization problem alone would be enough to not consider this option further.

However, we did consider this option and did some preliminary work in how one would go about modifying the AspectJ compiler directly. The first problem that one encounters is familiarizing oneself with the compiler code. This task was quite large and does take a long time. In order to minimize this, we decided to use AspectJ to insert code at certain points during the compilation process. This allowed us to insert code to analyze/modify the parse tree and other internal data structures at certain points within the AspectJ compiler. Adding functionality in this manner would minimize the dependency between our modification and the base AspectJ compiler source code. Then, the issue of synchronizing our addition with the AspectJ compiler would be minimized.

Utilizing DJ could further minimize the dependency with AspectJ compiler. Using DJ would allow us to process the parse tree that AspectJ compiler has built and translate all of the extension AspectJ language to the base AspectJ language. The dynamic nature of DJ allows us to add code that would traverse the parse tree relatively easily with minimal dependence on the existing AspectJ compiler source code.

Another considered option of modifying the backend of the AspectJ compiler to process the new extensions, ended up being very impractical. We would have to modify and add code to the AspectJ compiler source code to process the extensions. This in effect makes our implementation to be highly dependent on the AspectJ compiler source code and exacerbates the code synchronization problem.

In all of these options, there needs to be much code that needs to be modified/added to parse the new language extension. We cannot avoid from this code modification, unless we created our own development environment and used the AspectJ compiler as the
backend. In effect making our language extension implementation dependent on the AspectJ language and not the implementation. Thus, these options were not used in our final implementation.

The second option of creating our own development environment was determined to be the best choice for our purposes. It allowed maximum independence from the AspectJ compiler source code by only using the AspectJ compiler as a backend compiler. We only depend on the AspectJ language since we would be translating the new language extensions to the base AspectJ language, which was explored in the previous section.

Also, this option gives us maximum flexibility. We may add any number of steps or complexity into our development environment and specify our own parser and translator using any methods that we choose. However, the users of this new development environment have to learn a new method of developing programs no matter how close the new development environment was to the AspectJ compiler.

For the implementation of the new development environment, we decided to use DemeterJ. DemeterJ allows us to specify a parser very quickly using the class dictionary feature as shown earlier. The method of traversing object graphs allows us to develop the code for processing the new language extension very quickly. These two features of DemeterJ give us all the benefits of AOP and increases productivity in developing the new development environment.

4 DAJ DESIGN

DAJ [9], pronounced as "dodge", is an application that allows programmers to add Demeter traversals to their Java programs. This is the result of the analysis described in the previous section. DAJ integrates DJ [8], DemeterJ and AspectJ to implement a system that allows programmers to specify traversals for their AspectJ programs. Thus, DAJ extends AspectJ and does this incrementally without modifying the AspectJ compiler.

The DAJ programmer uses *.java files to write AspectJ code and in addition *.trv files to express the concepts of Demeter such as class graphs, traversal strategies, visitors and adaptive methods. All those concepts are expressed using the AspectJ declare statement which is intended for extending AspectJ.

The system architecture was designed such that DAJ will minimize coupling with the AspectJ compiler as discussed earlier. We also wanted the new language to have similar syntax as AspectJ and Java to lower the learning curve for programmers. Therefore, we decided to create a tool that would use AspectJ, DemeterJ and DJ to generate the traversals in AspectJ and use AspectJ's compiler as the backend weaver. Fundamentally, DAJ is a reimplementation of DemeterJ traversals, but using AspectJ as the weaving language instead of the DemeterJ weaving language. Our goal is to focus our future development of Demeter tools on DAJ because it has a much easier learning curve than DemeterJ (assuming AspectJ is already known) and DAJ runs as efficiently.

The grammar aspect from DemeterJ is partially added to DAJ but is not described further in this paper. It allows AspectJ programmers to develop in the Java data binding style (JSR 31) by using a schema to both define a class graph as well as a language for describing the objects of the class graph. While the Java data binding style focuses on XML languages, DAJ programmers have the more powerful LL(k) languages available which allow for more structure-shy representations of objects. We have also added the capability to use traversals to define complex AspectJ type patterns in a robust manner.

The DAJ notation that we use in this paper is the notation in the current implementation. It is not yet optimally aligned with the AspectJ syntax but this will be easy to change. We hope that DAJ is only a temporary tool and that most of its features will be added eventually to AspectJ because they fit in very well and give significant power to the AspectJ programmer.

**Figure 15: Four Compilation Phases of DAJ**

There are four phases in DAJ as shown in Figure 15: stub generation, traversal generation compilation, traversal generation, and traversal compilation. In the stub generation phase, the traversal files are parsed and stub traversal methods are generated for compilation in the next phase. Without these stubs, the method calls to the expected traversal methods that will be generated in the traversal generation phase, will cause a compilation error. Thus, for every traversal, DAJ will generate a stub method for the source of the strategy for a particular traversal.

In the following phase, the AspectJ compiler compiles the generated stub methods, CreateClassGraph.java and the user code. The CreateClassGraph.java, distributed in the installation of DAJ, is needed in the traversal generation phase to intercept the call to the main method, create an instance of ClassGraph using DJ, and generate the traversals. Finally, the AspectJ compiler is used to generate the .class files that will be executed in the next phase.

In the traversal generation phase, DAJ executes the code that has been compiled in the previous phase. During the execution the call to the main method is intercepted. DAJ then generates a ClassGraph object using DJ and uses it and DJ to generate the traversals specified in the .trv files. These traversals are then translated to AspectJ code as outlined in the translation experiment.

In the last phase, DAJ compiles the generated traversals and the user's code using the AspectJ compiler. The traversal compilation phase generates the .class files that the user is expecting from DAJ. These four phases that have been outlined are necessary to decouple DAJ from DJ and AspectJ.

These four processes are managed by the DAJ’s main method. DAJ’s main method processes the command line arguments. Then, it uses the appropriate methods to invoke the four processes for generating the AspectJ traversal code. After the command line argument processing, it calls the appropriate methods to generate the stubs in the stub generation process. In the next three processes, it creates a
shell command line and uses Java’s API to execute that command. DAJ translates the command line given by the user to the appropriate command line arguments in each of the phases that is executed using the shell. This relationship between the DAJ main method and the four processes of the traversal generation is shown in Figure 16 below.

The traversal file, with .trv extension, has three major components, the class graph declarations, traversal declarations and aspect definition that enclose the class graph and traversal declarations. The class graph declaration may have two different types of class graphs: a default or a class graph slice. The class graph defined by the default class graph declaration is the class graph object that is obtained from CreateClassGraph.java using DJ. Thus, it contains all of the classes and relationships contained within a program.

The traversal declaration has two forms as well: default traversal and traversal with class graph as an argument. The default traversal declarations allow DAJ to generate the AspectJ traversal implementation using the default class graph as shown in Figure 19. Its syntax, with the "declare" and "traversal" keywords, is consistent with the AspectJ declare statements. These keywords are followed by a variable identifier, colon, strategy in quotes, and a semi-colon. When processing the default traversal, DAJ uses the default class graph from DJ to generate the traversal implementation.

As of version 1.0, DAJ allows programmers to specify visitors for specific traversals. DAJ uses Java reflection to obtain the method signatures of the visitor specified by the user. It then generates the appropriate AspectJ code to call the visitor advice at the right point during the traversal. In order to use this feature, the programmer must define a Java class with advice methods, declare the visitors in the .trv file, and specify the declared visitors in a traversal declaration. The methods that are processed by DAJ are specified in Figure 21.
• void start() – method called before the traversal starts
• void finish() – method called after the traversal finishes
• void before(C c) – method called before visiting a node of type C. The c parameter is the object of type C it is currently visiting.
• void after(C c) – method called after visiting a node of type C. The c parameter is the object of type C it is currently visiting.
• type around(C c) – method called instead of the traversal code
• type returnValue() – method that returns the value it stores. Will be used in later versions.

Figure 21: Methods Processes by DAJ for a Declared Visitor
The method start() is called before the traversal starts. The initialization code for the visitor should be defined in this method. The method finish() should contain code that should be executed when the traversal is finished. The typical advice: before, after and around are defined as methods with one argument. The argument is the type of node that advice should be executed when the visitor encounters that object type. The around method also includes a return type. However, this functionality is not very useful, because the returned value would be lost. This return type is allowed for future addition to DAJ. The last method allowed is the method returnValue(). This method would be invoked when the traversal finishes. However, this functionality is not implemented currently. In the future, the user would be able to declare a behavior that would generate wrapper method for the traversal such that it would create a new visitor, execute the traversal and then return the value from the method returnValue(). This is scheduled to be implemented in DAJ 1.1.

Next, the user may declare visitors as shown in Figure 22. The keyword Visitor is used to signify that this statement is declaring a visitor, followed by a variable. This variable should be the class name of the visitor defined in plane Java. Finally, the statement ends with a semicolon. This syntax is consistent with Java’s variable declaration syntax. The visitor declaration allows DAJ to use these visitors for traversals.

Visitor variable;
Visitor BasketVisitor;
Visitor MyVisitor;
Visitor UniversalVisitor;

Figure 22: Visitor Declaration Syntax with Examples
Next, we add another traversal declaration type to allow visitors as specified in Figure 23. This is the same syntax as the traversals with class graph, except that we have added a second “argument” as the visitor variable that was declared earlier. DAJ will generate the traversal method for the strategy, similarly to the other traversal declarations. However, DAJ will generate another method that has the visitor type as an argument. This allows the user to use the traversal with or without a visitor, depending on what the user requires. For example, the first traversal declaration in Figure 23 would produce methods void t1() and void t1(BasketVisitor).

declare traversal t2(cg2, BasketVisitor): "from Here to There via Points";
declare traversal myTrav(default, MyVisitor): "from A via X to B";

Figure 23: Traversal Declaration with Visitor Syntax and Examples
The last language feature in DAJ is the aspect declaration. It contains class graph declarations, visitor declarations and traversal declarations as shown in Figure 24. The syntax is designed to be similar to AspectJ. However, programmers may not add AspectJ code within the traversal files. This capability requires that DAJ be able to parse AspectJ syntax, however it was not implemented because of the time constraint.

aspect MyTraversal {
    ClassGraph defaultCG;
    ClassGraph cg1 = new ClassGraph(defaultCG,
        "from * bypassing { BadNode } to *");
    declare traversal t1: "from CompoundFile to SimpleFile";
    declare traversal t2(cg1): "from CompoundFile to *";
    Visitor FindVisitor;
    declare traversal t3(cg1, FindVisitor): "from CompoundFile bypassing { ->*,parent,*} to File";
}

Figure 24: Aspect Declaration Example Containing Different Types of Declarations
Lastly, we use DAJ to implement the traversal for the Basket Example. The DAJ traversal file in Figure 25 will generate the traversal code similar to the one shown in Figure 13. Thus, we may replace the traversal code in Figure 13 with the DAJ traversal file in Figure 25 below. The other user code may stay the same.

aspect BasketTraversal {
    declare traversal t1: "from Basket to Weight";
}

Figure 25: Traversal File for the Basket Example
DAJ is a tool that allows AspectJ programmers to use Demeter concepts in their programs. We have presented the compilation process, syntax for the traversal specification files, and an example that implements the traversal from the Basket Example in DAJ. It is implemented using DJ, AspectJ and DemeterJ and combines concepts from two aspect-oriented programming tools AspectJ and DemeterJ. Because of the way DAJ was implemented, it extends the AspectJ language by incremental addition rather than by modification. It adds a preprocessing on top of the AspectJ compiler. This implementation is possible because of the powerful features of aspect-oriented programming.
5 CONCLUSION AND FUTURE WORK

The AOP tool AspectJ has become very popular with programmers and researchers. By extending AspectJ with other AOP concepts could increase the rate of general adoption of AOP and facilitate conceptual discourse between AOP researchers. Demeter and AspectJ are conceptually compatible: Both use a dynamic join point model. While in Demeter we tend to focus on advising traversal methods, AspectJ advises a richer set of join points. This affinity between the two approaches makes an integration attractive which lead us to the development of DAJ.

DAJ is basically a re-implementation of DemeterJ using AspectJ as the weaver instead of using our own weaver. We plan to focus our further development of Demeter tools on DAJ because DAJ gives programmers the benefits of AspectJ and Demeter combined. DAJ has a smaller learning curve than DemeterJ assuming programmers know AspectJ.

We explored several implementation options for extending the AspectJ language and we found that using an approach that relies only on the published language interface of AspectJ is the best for us. We invented a simple little aspect trick with which we extend the AspectJ language with new features without ever touching the compiler.

DAJ is implemented using DemeterJ, DJ, and AspectJ. DAJ would not have been possible if we did not have all three AOP tools available to us. Using the three in combination lowered the software development time. And DAJ does allow programmers and researchers to download one software development kit and obtain access to Demeter concepts and AspectJ.

For future development of DAJ, we may integrate the language extension and the AspectJ language such that we do not need to have a separate file with its own language requirements. Also, adding the class dictionary feature of DemeterJ (already partially done) would eliminate the need to use DJ and AspectJ to obtain the class graph. The class graph could be obtained from the class dictionary instead of DJ at runtime. This would simplify the implementation and increase the utility of DAJ.

Finally, for future extensions to AspectJ, it would be most efficient if AspectJ included an API to add plug-ins for this purpose. Having the plug-in API for language extension would eliminate the need for creating our own wrapper to process the language extension.

REFERENCES