ASPECTUAL CONCEPTS

by

John J. Sung

College of Computer Science

Northeastern University

2002

© Copyright

John J. Sung

jser@ccs.neu.edu

ASPECTUAL CONCEPTS

by

John J. Sung

A thesis submitted as part of requirements

for the degree of

Master of Science in Computer Science

School of Computer Science

Northeastern University

2002

Supervisor: Karl Lieberherr ______________________ Date ______________________

Reader: Mitch Wand ______________________ Date ______________________

TABLE OF CONTENTS

Table of Contents....i
Table of Figures....iii
Acknowledgements....vi
Thesis Proposal....1
Introduction to AOP Tools....5
    Fred....5
    Demeter Concepts....7
    DemeterJ....7
    DJ....13
    AspectJ....16
Translating Fred....23
    Translating Fred to a Graphical Notation....23
    Translating the Factorial Example....24
    Translating the String Example....26
    Translating Fred to AspectJ....29
    Conclusion....31
Translating Between Aspectj and Demeter....33
Join Point Model VS. Traversal Model of Programming....40
DAJ: Demeter integrated with AspectJ....46
    Compilation Process....46
    Traversal Specification Following DJ and AspectJ Syntax....48
    Conclusion....53
Performance Analysis of DAJ....54
    Performance Measurement Methodology....54
    Experimental Results and Analysis....59
    Conclusion....60
The Four Graph Model of Programs (4GMP)....61
    Four Graph Model of Programs....61
    Factorizational Concerns....64
    Organizational Concerns....65

Concern Relationship Diagram........67
AspectJ Analysis........69
DemeterJ Analysis........73
Conclusion........77
Summary and Future Direction........78
Bibliography........80

TABLE OF FIGURES

Figure 1: Factorial Example in Fred ... 1
Figure 2: Factorial Example in AspectJ ... 2
Figure 3: Pointcut Exposing the Caller ... 3
Figure 4: NodesOf Operator to Specify a Set of Classes ... 3
Figure 5: NodesOf Operator Intersected with Worker ... 3
Figure 6: Factorial Example in Fred ... 6
Figure 7: Demeter Process Overview ... 7
Figure 8: Class Graph of the Basket Example ... 8
Figure 9: Class Dictionary of the Basket Example ... 8
Figure 10: Class Dictionary of Basket Example with Fruit as Concrete Class Instead of Abstract Class ... 9
Figure 11: Traversal Graph as a result of applying the Strategy "from Basket to Weight" to the Basket Example Class Graph ... 9
Figure 12: Declaration of the CountWeightVisitor in the Class Dictionary ... 10
Figure 13: Definition of the Traversal with Strategy "from Basket to Weight" with the CountWeightVisitor and the definition of CountWeightVisitor. ... 11
Figure 14: Definition of Strategy and an Inline Visitor for Counting Total Weight within a Basket ... 12
Figure 15: Class Dictionary of Basket Example Converted to Generate a XML Parser for the Schema Specified by the Class Dictionary ... 12
Figure 16: Valid Sentence for the Grammar Specified in Figure 15 ... 13
Figure 17: Definition of the Basket Example Class Graph in Figure 8 ... 14
Figure 18: Obtaining a Reference to the Class Graph via Constructor Call for Class ClassGraph in DJ ... 14
Figure 19: Definition of the CountWeightVisitor for DJ ... 15
Figure 20: Java Code Showing How Demeter Concepts Work Together in DJ ... 16
Figure 21: Defining Classes for the Basket Example ... 17
Figure 22: Defining Relationships for the Basket Example Class Graph ... 17
Figure 23: Definition of Constructors and Methods for the Basket Example ... 18
Figure 24: CountWeight Aspect of Basket Example ... 20
Figure 25: Code for Testing the AspectJ Basket Example ... 20
Figure 26: A Conceptual Model of AspectJ ... 21
Figure 27: Primitives for Graphical Expression of Fred ... 23
Figure 28: Graphical Translation of Factorial Example ... 24
Figure 29: Graphical Notation for concat ... 26
Figure 30: Graphical Notation for len ... 27
Figure 31: Graphical Notation for ref ... 28
Figure 32: Factorial Example Translated from Fred to AspectJ ... 30
Figure 33: Translation of concat from Fred to AspectJ ... 31
Figure 34: Visualization of JPM ... 33
Figure 35: Visualization of Program as a Journey Metaphor used in Demeter ... 34
Figure 36: Implementation of the Basket Example Traversal in AspectJ and the Pointcuts for the Relevant Join points within the Traversal ... 35
Figure 37: Demeter Style Visitor for Basket Example Implemented in AspectJ ... 36

Figure 38: AspectJ Basket Example Translated to Modified DemeterJ.. 38
Figure 39: Schematic Diagram of an XOR from [20 HowStuffWorks].. 40
Figure 40: Join points for the Molecular Model of Caffeine from [21 KoffeeKorner]............. 41
Figure 41: Join points for the Basket Example Class Graph.. 41
Figure 42: Four Compilation Phases of DAJ.. 46
Figure 43: Management of the DAJ Traversal Generation Process.. 48
Figure 44: Default Class Graph Declaration Syntax and Examples.. 48
Figure 45: Class Graph Slice Declaration Syntax and Examples.. 49
Figure 46: Default Traversal Declaration Syntax with Examples.. 49
Figure 47: Syntax and Examples of Traversal Declaration with a Class Graph as an Argument... 50
Figure 48: Methods Processes by DAJ for a Declared Visitor.. 50
Figure 49: Visitor Declaration Syntax with Examples... 51
Figure 50: Traversal Declaration with Visitor Syntax and Examples.. 52
Figure 51: Aspect Declaration Example Containing Different Types of Declarations............. 52
Figure 52: Traversal File for the Basket Example.. 53
Figure 53: Class Dictionary of the Class Graph Used in Performance Tests............................. 55
Figure 54: Sample Input File for the Class Dictionary in Figure 53... 55
Figure 55: Traversal Implemented with Inlined Visitor in DemeterJ... 56
Figure 56: DJ Visitor Implementation in DemeterJ Class Dictionary and Behavior file Statements.. 57
Figure 57: DJ Code to Execute the Traversal on the Object Graph Rooted at Container c..... 57
Figure 58: DAJ Visitor Implementation using DemeterJ Class Dictionary and Behaviroal File Statements... 58
Figure 59: DAJ .trv File to Implement the Traversal.. 58
Figure 60: Table of Results for the Performance Experiment... 59
Figure 61: Graph of Elapsed Time of DemeterJ, DJ, and DAJ Implementation of the Traversal Varied with Object Graph Size.. 59
Figure 62: Graph of Elapsed Time of DemeterJ and DAJ Implementation of the Traversal Varied with Object Graph Size.. 60
Figure 63: Graphic Decomposition of Programs.. 62
Figure 64: Graph Transformation Model (GTM) in 4GMP... 63
Figure 65: Application of Graph Transformation Model for Compiled Programs.................... 64
Figure 66: Factorizational Concern Process... 65
Figure 67: Organizational Concerns in 4GMP.. 66
Figure 68 Cohesion and Coupling from [8 Tucker 1997].. 66
Figure 69: Concern Relationship Diagram of 4GMP.. 67
Figure 70: Concern Relationship Diagram for OOP Features Class and Inheritance................ 68
Figure 71: Proposed Ideal CR Diagram.. 69
Figure 72: Refined Ideal CR Diagram... 69
Figure 73: AspectJ Introductions for Introducing Data Members and Methods for the Basket Example.. 70
Figure 74: AspectJ Pointcut Examples... 71
Figure 75: AspectJ Before Advice.. 72
Figure 76: Factorial Example Implemented in AspectJ.. 72
Figure 77: AspectJ Concern Relationship Diagram... 73
Figure 78: Class Dictionary of Basket Example that Generates a XML Parser for the Schema Specified by the Class Dictionary... 74

Figure 79: Visualization of the Relationship between Class Graph, Strategy Graph and Traversal Graph 75
Figure 80: CR Diagram of DemeterJ 76

ACKNOWLEDGEMENTS

I would like to thank my advisor, Karl Lieberherr for entertaining my interests in this research and giving me the opportunity to contribute to the aspect oriented programming community. He has always been supportive in giving me valuable advice and direction for this thesis. I would also like to thank my brothers-in-arms in the field of aspect oriented programming, Doug Orleans, Pengchang Wu, Johan Ovlinger, and Therapon Skotiniotis for their helpful comments, questions and suggestions. One of the unsung heroes of this thesis is Jon Kelly. The quality of DAJ would not be what it is without his testing efforts. This thesis would not have been possible if Mitch Wand were not willing to read and review my small contribution to the aspect oriented programming field. Lastly, this thesis would never have been conceived without the faculty, students and administrators of the School of Computer Science at Northeastern University.

Thank you all!

THESIS PROPOSAL

Aspect Oriented Programming (AOP) is becoming prominent in the field of Computer Science. The tools that allow programmers to use AOP methodologies to improve their productivity have been around for quite sometime. However, there has not been much research into the basic concepts used in these methodologies. We bring this to the attention of the research community by attempting to understand the fundamental concepts behind these aspect oriented programming tools, how those concepts are applied in different aspect oriented programming tools, and create a coherent model of AOP concepts that would allow us to make informed decisions about how these concepts could be applied in novel ways.

First, we will investigate the fundamental concepts that are evident in AspectJ [9 Kiczales2001], DemeterJ [14 DemeterJ], DJ [15 DJ], and Fred [1 Orleans02]. These tools use the concepts of predicated execution, advice and traversal in different ways. AspectJ uses predicated execution and advice concepts on a program's control flow graph. Fred attempts to define the program's control flow graph using the predicated execution and around concepts. DemeterJ and DJ use traversals and advices on an object graph. Some of these tools have been around for years, but we still do not understand how these tools and their concepts are related. We will investigate these relations and determine their usability for AOP tool analysis.

One way to explore these relations is to attempt to implement concepts from one language to another. The simple implementation of factorial function written in Fred and AspectJ in Figure 1, shown below, illustrates how one may apply the concepts from branch oriented programming from [1 Orleans02] using AspectJ.

```
(define-msg fact)

(define-branch (and (eq? (dp-msg dp) fact)
                    (= (car (dp-args dp)) 1))
  1)

(define-branch (eq? (dp-msg dp) fact)
    (let ((x (car (dp-args dp))))
        (* x (fact (- x 1)))))
```

Figure 1: Factorial Example in Fred

In this simple example that implements the factorial function, the first line defines a message named **fact**. The second Scheme statement defines a branch that handles the case where the first argument is equal to one. The last line defines the default branch that handles the case where the first argument is not equal to one. In this manor, a programmer would describe each case of what happens for this particular message.

```
class FactClass {}
aspect FactExample {
    static int FactClass.fact(int x) {
        return 1;
    }

    int around(int x) : args(x)
            && call(static int FactClass.fact(int))
            && if(x==1) {
        return 1;
    }

    int around(int x) : args(x)
            && call(static int FactClass.fact(int)) {
        return x * FactClass.fact(x-1);
    }
}
```

Figure 2: Factorial Example in AspectJ

We translate the Fred implementation of the Factorial Example in Figure 1 to AspectJ as shown in Figure 2. In this AspectJ implementation, the **FactClass** is empty and the factorial method is defined within the **FactExample** aspect. The two around advices and their inlined pointcuts are the two possible branches for the method **fact()**.

The close correspondence between the Fred implementation and the AspectJ implementation of the factorial function is very clear in the Factorial Example. There is a one to one correspondence between the two implementations. This relatively simple translation process opens the possibility for implementing Fred concept using AspectJ.

In order to find more of these types of conceptual applications, a consistent view that one can take to make all of the AOP approaches clearly would be useful. One way to accomplish this is to view these ideas as abstracting the different parts of a program into graphs. The instance of nodes and edges in each AOP tool are different, but the basic concepts are same. This allows

us to think of all of the different things that one can do with graphs, traversals, mapping, grouping of nodes, subgraphs, etc. If we apply these graph concepts to the different graphs evident within programs that is consistent with the different implementations of aspect oriented programming tools, we maybe able to find possibilities for future development of those aspect oriented programming tools.

```
pointcut perCallerWork(Caller c, Worker w):
    cflow(invocations(c)) && workPoints(w);
```

Figure 3: Pointcut Exposing the Caller

This idea of abstracting AOP concepts into generic graphs will be explored by attempting to implement an idea from a graph operation to different tools. In [10 Kiczales et. al], the authors present an example in which one wants to have access to some node or context that the entity doing the traversal has seen before. In AspectJ, you would use a pointcut to specify the specific data at the join point that you want to expose to the join point encountered later in the traversal of the dynamic call graph. In the example in [10 Kiczales et. al], the `Caller c` is exposed to all of the `Workers w` by the pointcut shown in Figure 3.

```
NodesOf(from Caller to *)
```

Figure 4: NodesOf Operator to Specify a Set of Classes

This in effect allows programmers to specify a subsection of the program flow by using a `cflow` pointcut. Thus, the `cflow` pointcut extends AspectJ by allowing programmers to specify a specific program flow. In Demeter, this would be a traversal. Therefore, we may extend AspectJ in a similar manner. We may extend the power of pointcuts by applying the traversal strategies specified in [19 Lieberherr et al.] to type patterns in pointcuts. The programmer would specify a strategy and a `NodesOf` operator to specify the nodes encountered during the traversal as shown in Figure 4.

```
NodesOf(from Caller to *) intersect Worker
```

Figure 5: NodesOf Operator Intersected with Worker

We would intersect this set of classes with the set of classes in set `Worker` to obtain the set of `Worker` nodes encountered during the traversal, as shown in Figure 5.

When the `NodesOf` operator is applied to a graph, it returns all the nodes in the class graph. Given a strategy, it would return all of the nodes that are in the traversal graph that is the result of applying the strategy to a class graph. It would be useful to extend AspectJ with strategies and the `NodesOf` operator to define AspectJ type patterns more flexibly.

In DJ, one would use the `ContextVisitor` to expose the stack of objects that the visitor has encountered thus far in the object graph during a traversal. In both of these cases, the program can expose parts of the graph that was traversed earlier in the traversal. This is a powerful concept that is evident in AspectJ, DemeterJ and DJ. This thesis will attempt to find and categorize these types of commonalities between AOP tools and integrate them into a consistent view of aspect oriented programming concepts.

The relevant type of experiments for this thesis include using AspectJ to implement the concepts from other AOP tools, integrating AspectJ and AP Library, and implementing a real-world application using the tool from the integration work. The purposes of the first experiments should expose the relationships between AspectJ and other AOP tools. Some examples of these experiments have already been presented. The next type of experiment, integrating AspectJ and AP Library, is designed to find out how well these tools interact with each other. The last experiment will measure the effectiveness of the integration experiment. All of these experiments have one purpose, to direct the shaping of the consistent view of aspect oriented programming that integrate concepts from AspectJ and Demeter.

The consistent view of aspect oriented programming that will be shaped by the proposed experiments should be based on abstracting aspect oriented programming concepts into graphs and the manipulation of these graphs as mentioned. This new view should be a starting point for a new way of analyzing aspect oriented tools, i.e. aspect oriented tool analysis. This new type of analysis should expand the current understanding of aspect oriented programming tools and concepts that will lead to future development of those tools and concepts.

INTRODUCTION TO AOP TOOLS

This master's thesis started with vague notions that there have to be better ways of expressing what a user would want the computer to accomplish on behalf of the user. Because of this notion, a quick survey of latest aspect oriented programming languages were taken from such source as [22 CACM]. The languages surveyed were DemeterJ [14 DemeterJ], DJ [15 DJ], Fred [1 Orleans02], and AspectJ [9 Kiczales2001]. DemeterJ and DJ were developed at Northeastern University and were a natural selection as a graduate student of Northeastern University. Doug Orleans, from Northeastern University, developed Fred as part of his Ph.D. dissertation and its close ties with AspectJ made it a prime candidate for this research. AspectJ was selected for its popularity among the AOP community. These four AOP tools were determined to be the tools that will be used in our experimentation and analysis.

However, we will not present the idea of cross cutting concerns and the reason why this is important in AOP. Much of this has been discussed is many papers and magazine articles. If you would like a quick overview of the issues in AOP, then one should read the October, 2001 issue of the *Communications of the ACM* [22 CACM].

Fred

Fred [1 Orleans02], developed by Doug Orleans, is a language that integrates aspect oriented programming and predicate dispatching and it is implemented in [13 MzScheme]. The main concepts in Fred are message, branch, and decision point. In order to simplify these terminologies, we will map these concepts to functional programming concepts. A message maybe thought of as a function. Defining a message is equivalent to declaring a function. Then, a branch is analogous to a function call. It is actually more complicated than that, but we will use this analogy for the purpose of this thesis. The decision point is where a decision is made about which branch should be invoked. This requires that a predicate is associated with each branch. Fred follows the branch that satisfies Fred's rules about predicate precedence in cases where multiple branch predicates are true.

In order to understand how one would use these concepts, we will present the Factorial Example that implements the factorial function. In Fred, a programmer defines a message by

calling the function **define-msg**. In Figure 6, the first statement defines a message called **fact**. Next, we define some branches for the message **fact**. The first case is the base case, where the first argument is equal to one. This is expressed in Fred in Figure 6 by the second MzScheme statement. We use the function **define-branch** to define a new branch that executes its body when the message is equal to **fact** and the first argument is equal to one. Within this body, we return one, which is what would happen if we called the function factorial with one as the argument. Next, we handle the recursive case by defining another branch as represented by the third MzScheme statement in Figure 6. It defines a branch that is invoked when the message is **fact**. In the body of the branch, we make a recursive call to **fact** with x - 1 as the argument and multiply that by x. Then, we return the result of that multiplication as shown in Figure 6.

```
(define-msg fact)

(define-branch (and (eq? (dp-msg dp) fact)
                    (= (car (dp-args dp)) 1))
  1)

(define-branch (eq? (dp-msg dp) fact)
    (let ((x (car (dp-args dp))))
        (* x (fact (- x 1)))))
```

Figure 6: Factorial Example in Fred

There is some precedence ordering for the two branch predicates. The predicates for both of these branches maybe true for the case **(fact 1)**. What does Fred do in this case? Fred determines the more specific predicate and invokes that branch. Thus, when **(fact 1)** is encountered, Fred invokes the first branch instead of the second branch. The rationalization for this decision is that the first predicate is more specific than the second predicate, i.e. the second predicate is always true when the first one is true, but not vice versa. Thus, Fred always selects the more specific predicated branch. Therefore, the second branch is invoked for all other cases for the call to the function **fact**. This gives us the behavior that we want from Fred for the implementation of the factorial function.

Demeter Concepts

Next, we will introduce Demeter concepts and how these concepts are implemented in DemeterJ and DJ. 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 one that traverses the traversal graph. It has advices that are 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 substitute for the class graph and the strategy. This Demeter process is shown in Figure 7 below.

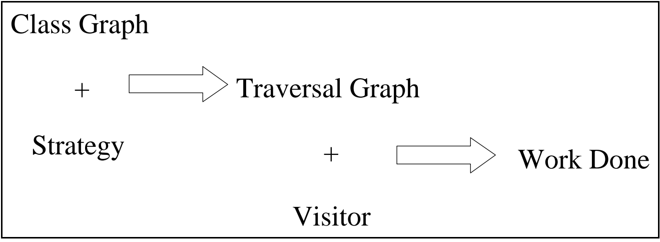

Figure 7: Demeter Process Overview

DemeterJ

The way in which the programmer specifies the three components class graph, strategy and visitor is different for DemeterJ and DJ. In 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 cictionary as well, since it is implemented as a Java class. The strategy and the definition of the visitor 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 example is shown in Figure 8. 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** class, which is represented by a **String** class. In this example, we will count the integers in the class **Weight** that are contained in a **Basket**.

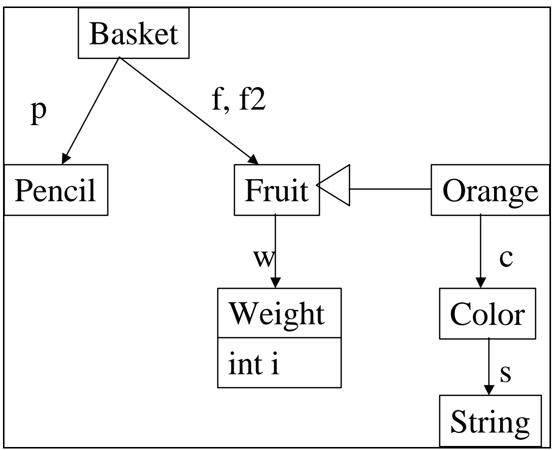

Figure 8: Class Graph of the Basket Example

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 9. 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.

```
Basket = <p> Pencil <f> Fruit <f2> Fruit.
Fruit : Orange common <w> Weight.
Pencil = .
Weight = <i> int.
Orange = <c> Color.
Color = <s> String.
```

Figure 9: 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 inheritance. The modified class dictionary is shown in Figure 10. The ":" for **`Fruit`** has changed to "=" 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`**

```
Basket = <p> Pencil <f> Fruit <f2> Fruit.
Fruit = <w> Weight.
Pencil = .
Weight = <i> int.
Orange = <c> Color extends Fruit.
Color = <s> String.
```

Figure 10: Class Dictionary of Basket Example with Fruit as Concrete Class Instead of Abstract Class

Next, we need to specify the strategy for the traversal. Since 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 11.

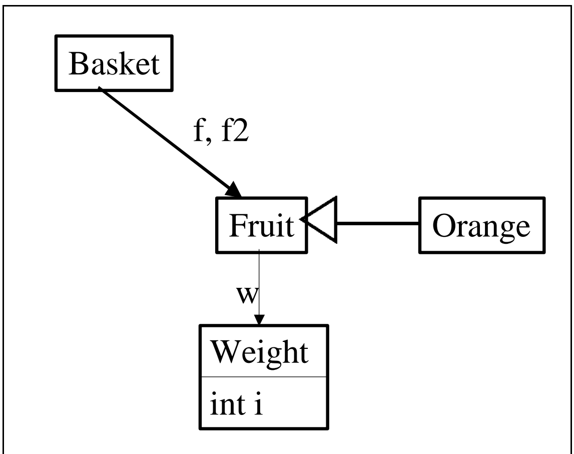

Figure 11: Traversal Graph as a result of applying the Strategy "from Basket to Weight" to the Basket Example Class Graph

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 12 to the class dictionary. It defines a class **`CountWeightVisitor`** with a data member variable **`total`** of type **`int`**.

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

Figure 12: Declaration of the CountWeightVisitor in the Class Dictionary

Now, we are ready to define the traversal and the visitor in the behavior file. When defining the traversal, you have to first specify the starting node of the traversal in our strategy, i.e. **`Basket`**. Then, we type in "{" to start the scoping of the behavior to be added to the class **`Basket`**. We start the definition with the key word "traversal" then the name of the traversal. Next, we have the visitor name and a variable name with parenthesis around them. Then, another "{" to start specifying the rest of the strategy. We add the keyword "to" and then the destination, **`Weight`**. We are finished with this definition of a traversal, so we signifying the end of the strategy with ";" and "}". Then, close the scope of the class Basket with another "}". This traversal definition is shown in Figure 13.

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 looks exactly like Java code, except for the double curly-braces. These double curly-braces are used to signify pure Java code to the DemeterJ weaver.

```
Basket {
   traversal countTraversal(CountWeightVisitor cwv) {to Weight;}
}

CountWeightVisitor {

   public void start() {{ total = 0; }}

   before Weight {{
        total += host.get_i();
   }}

   public int getReturnValue() {{
        return total;
   }}
}
```

Figure 13: Definition of the Traversal with Strategy "from Basket to Weight" with the CountWeightVisitor and the definition of CountWeightVisitor.

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 in lined 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 in lined visitor is illustrated in Figure 14 below.

```
Basket {
   public int getTotalWeight() to Weight {

         {{ int total = 0; }}

         before Weight {{
            total += host.get_i();
         }}

         return int {{ total }}
   }
}
```

Figure 14: Definition of Strategy and an Inline Visitor for Counting Total Weight within a Basket

The body of the method **getTotalWeight** looks almost exactly like 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 "{{ in total=0;}}", which defines and sets the running total of the weights for the in lined visitor. We specify the advice similar to the one in Figure 12. Then, we specify the return value total and the return type to be **int**. This in line visitor is very useful if you do not have complicated visitors that you want to reuse.

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 inlined 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.

```
Basket = “<basket>” <p> Pencil <f> Fruit <f2> Fruit “</basket>”.
Fruit : Orange common <w> Weight.
Pencil = “<pencil>” “</pencil>”.
Weight = “<weight>” <i> int “</weight>”.
Orange = “<orange>” <c> Color “</orange>”.
Color = “<color>” <s> String “</color>”.
```

Figure 15: Class Dictionary of Basket Example Converted to Generate a XML Parser for the Schema Specified by the Class Dictionary

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 9. We will convert it in a way that the grammar will specify a language that reads in XML as input and creates 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 15. An example of valid sentence for the grammar specified in Figure 15 is shown in Figure 16. It specified a `Basket` object with a `Pencil` and two `Oranges`. Each of the `Oranges` has the string “orange” as the color and has weights 5 and 9 respectively.

```
<basket>
   <pencil></pencil>
   <orange>
     <color>"orange"</color>
   </orange>
    <weight>5</weight>
  <orange>
     <color>"orange"</color>
   </orange>
     <weight>9</weight>
</basket>
```

Figure 16: Valid Sentence for the Grammar Specified in Figure 15

This parser generation feature of DemeterJ is significant, because it allows programmers to add minimal amount of strings to specify an input file language. DemeterJ also generates default visitors and traversals to print and display the object graph parsed from an input file. This maybe used for internal testing of the application being developed and it has been an invaluable feature for testing of applications and programs developed for this thesis.

DJ

Next application of the Demeter concepts is DJ. This is a Java library that allows programmers to specify the class graph, strategy, and visitor through a Java API. This is useful for programmers that do not want to learn a new language to reap the benefits of the Demeter concepts. Programmers may include the AP Library in their program and obtain the full capabilities of these concepts.

As in DemeterJ, DJ allows programmers to specify the class graph. While DemeterJ uses class dictionaries, DJ uses Java Reflection to construct a representation of the class graph. DJ provides an API to access this functionality. The class **ClassGraph** in the AP Library [16 APLib], allows programmers to obtain the class graph and use it to traverse object graphs.

```
class Basket {
   Basket(Fruit _f, Pencil _p) { f = _f; p = _p; }
   Basket(Fruit _f, Fruit _f2, Pencil _p) { f = _f; f2 = _f2; p = _p; }
   Fruit f, f2;
   Pencil p;
}
class Fruit {
   Fruit(Weight _w) { w = _w; }
   Weight w;
}
class Orange extends Fruit {
   Orange(Color _c) { super(null); c=_c;}
   Orange(Color _c, Weight _w) { super(_w); c = _c;}
   Color c;
}
class Pencil {}
class Color {
   Color(String _s) { s = _s;}
   String s;
}
class Weight{
   Weight(int _i) { i = _i;}
   int i;
   int get_i() { return i; }
}
```

Figure 17: Definition of the Basket Example Class Graph in Figure 8

The actual specification of the class graph is in plain Java as shown in Figure 17. This Java code specifies the Basket Example class graph in Figure 8. The usual method of defining classes and their data member variables for "has-a" relationships and the "extends" keyword for the "is-a" relationships are used to define the class graph. In DemeterJ, this Java code would have been generated from the class dictionary.

```
ClassGraph cg = new ClassGraph();
```

Figure 18: Obtaining a Reference to the Class Graph via Constructor Call for Class ClassGraph in DJ

Obtaining the reference to this class graph is simple as calling the constructor to the class **ClassGraph** as shown in Figure 18. DJ uses Java Reflection to construct the class graph representation from the current running program. This allows programmers to adapt their existing applications to use DJ without learning a new programming language.

```
class CountWeightVisitor extends Visitor {
   int total;

   public void start() {
        total = 0;
   }

   public void before(Weight w) {
        total += w.get_i();
   }

   public int getReturnValue() { return total; }

}
```

Figure 19: Definition of the CountWeightVisitor for DJ

Next, we define a visitor to count the weights within the Basket as shown in Figure 19. The Java code is very similar to the definition of the visitor in DemeterJ from Figure 13, except for minor syntactic differences. Therefore, DemeterJ's syntax for the behavior files is very close to their Java equivalents.

Lastly, we need to invoke the traversal with the object graph, strategy and visitor. The object graph is created by calling the constructors for the classes. We also need to instantiate an instance of the **CountWeightVisitor** to visit the nodes in the object graph. Then, we call the method **traverse()** for class **ClassGraph** with the "root" of the object graph, the strategy and the visitor. The actual Java code showing how all of these components come together is shown in Figure 20.

```
class DJBasket {

   static public void main(String args[]) throws Exception {

       Basket b = new Basket(new Orange(new Color("orange"),  new Weight(5)),
                             new Fruit( new Weight(10)),
                             new Pencil() );

        CountWeightVisitor cwv = new CountWeightVisitor();
        ClassGraph cg = new ClassGraph();
        cg.traverse(b, "from Basket to Weight", cwv);

        System.out.println("total weight: " + cwv.getTotal());
   }
}
```

Figure 20: Java Code Showing How Demeter Concepts Work Together in DJ

These examples of DemeterJ and DJ illustrate how Demeter concepts are implemented. We have only presented the major features of these tools. Even with these minimal set of features, they are powerful in their concept and application.

AspectJ

The last AOP tool that we will be discussing is AspectJ [10 Kiczales et. al]. It extends the Java Language to allow programmers to express crossing cutting concerns. The major feature of AspectJ that we are concerned with are introduction, join point, pointcut, and advice. Introductions allow programmers to define classes, data members, inheritance relationships and methods within another scope, besides the scope of the class. Thus, it empowers the programmers to organize the code in a more flexible manner. Join point allows programmers to specify a point within the execution of an application. Pointcut specifies a set of join points within an application. Finally, the advice is executed at the join points specified by the pointcut. This type of description of programs using join points is called Join Point Model (JPM).

The AspectJ features that were outlined are a party of the dynamic and static Join Point Models in AspectJ. A JPM consists of specification of what the join points are, a way to specify these join points, and a way to specify some semantics that will be added to the join point. The dynamic JPM is points in the execution of the program, pointcuts and advice. The static JPM

is classes and methods, introductions, and introduction bodies. These two JPMs, static and dynamic, are the two JPMs used in AspectJ.

We will show these concepts in action with the Basket Example. As with other examples, we need to present how the data structure, i.e. the class graph, is specified. We may accomplish this as with the DJ example using Java, however will use AspectJ's introduction feature to demonstrate the power of AspectJ.

```
class Basket { }
class Fruit { }
class Orange {}
class Pencil {}
class Color {}
class Weight{}
```

Figure 21: Defining Classes for the Basket Example

First, we define empty classes as shown in Figure 21. Then, we use the AspectJ introductions to introduce the data members and the inheritance relationship as shown in Figure 22. There are no requirements for the location of these introduction statements, except that they have to be within the scope of an aspect. These aspects allow the programmer to organize the different aspects of an application and all of the AspectJ extensions to Java are within the scope of these aspect statements.

```
aspect BasketRelations {

   declare parents: Orange extends Fruit;

   Fruit Basket.f, Basket.f2;
   Pencil Basket.p;

   Weight Fruit.w;

   Color Orange.c;

   String Color.s;

   int Weight.i;
}
```

Figure 22: Defining Relationships for the Basket Example Class Graph

In Figure 22, the declare parent statement allows programmers to introduce is-a relationships to their class graph. The data members declared in the aspect have to include the type, class name and the variable name. Thus, all of the data members are of the form "type classname.variablename;". This gives all of the information that the AspectJ compiler needs to introduce a data member to a class.

Next, we define the constructor and accessor methods for the classes within the Basket Example. In Figure 23, we accomplish this by introducing the constructors by defining the method `new()` for the classes and accessor method for `Weight`'s integer, `i`. Note that there is more typing involved than specifying these methods in the scope of the class, because we need to specify the scoping for each method introduced by specifying the name of the class.

```
aspect BasketConstructorsAndMethods {

   Basket.new(Fruit _f, Pencil _p) {
        f = _f;
        p = _p;
   }

   Basket.new(Fruit _f, Fruit _f2, Pencil _p) {
        f = _f;
        f2 = _f2;
        p = _p;
   }

   Fruit.new(Weight _w) { w = _w; }

   Orange.new(Color _c) { super(null); c=_c;}
   Orange.new(Color _c, Weight _w) {
        super(_w);
        c = _c;
   }

   Color.new(String _s) { s = _s;}

   Weight.new(int _i) { i = _i;}

   int Weight.get_i() { return i; }
}
```

Figure 23: Definition of Constructors and Methods for the Basket Example

Now, we specify the code to count the integer values within `Weight` objects in Figure 24. This specification is similar to the `CountWeightVisitor` for the Demeter implementations. We declare a static integer to be used within all of the methods to sum the integers within `Weight`

objects. We define the method `getTotal()` for `Basket`, `Fruit`, and `Weight`. These methods are used to traverse to the `Weight` objects.

In order to sum the weights, we have declared a pointcut `weightpc()`. It specifies the join points of all method calls to `getTotal()` and the target of the method call is to an object of type `Weight`. The after advice is executed every time the pointcut holds true. The "arguments" to the pointcut and the advice allows forwarding of the references accessible at the join point to the advice body. We did not have to use the pointcut and the advice, but we wanted to demonstrate a simple use of these AspectJ features. Lastly, we added a before advice to initialize the weight counter `total`.

The `CountWeight` aspect shown in Figure 24 has the same semantics as the implicit visitor in Figure 14. While the methods `getTotal()` defined for `Basket` and `Fruit` are written by hand for this example, it is generated by DemeterJ in Figure 14. While in DJ, the call to `traverse()` takes care of this function. The first three method introductions deal with the traversal to the `Weight` objects and the last three statements deal with the visitor in Figure 24

```
aspect CountWeight {

    static int total;

    int Basket.getTotal() {
        if (f!=null)
            f.getTotal();
        if (f2!=null)
            f2.getTotal();

        return total;
    }

    void Fruit.getTotal() {
        if (w!=null)
            w.getTotal();
    }

    void Weight.getTotal() {
    }

    pointcut weightpc(Weight w):
        call(* *.getTotal()) && target(w);

    before(Weight w) : weightpc(w) {
        total += w.get_i();
    }

    before() : call(* Basket.getTotal()) {
        total = 0;
    }
}
```

Figure 24: CountWeight Aspect of Basket Example

Finally, we define the main method to create an instance of **`Basket`** and then call **`getTotal()`** to obtain the total weight within the **`Basket`** in Figure 25. Then, we print out the result to the screen.

```
class AJBasket {

    static public void main(String args[]) throws Exception {

        Basket b = new Basket(new Orange(new Color("orange"),  new Weight(5)),
                    new Fruit( new Weight(10)),
                    new Pencil());

         int total = b.getTotal();
         System.out.println("total weight: " + total);
    }
}
```

Figure 25: Code for Testing the AspectJ Basket Example

From this example, we can see that in order to use the join points, we need to have some Java code to have meaningful pointcuts. From this we can create a conceptual model with two different "planes" of execution as shown in Figure 26. One plane is the original Java program and the other is where the bodies of advices in AspectJ are executed. That advice body, in turn, may contain method calls to the "Java Plane." In this manner, we can have "ping-ponging" back and forth between the two planes of execution as shown in Figure 26.

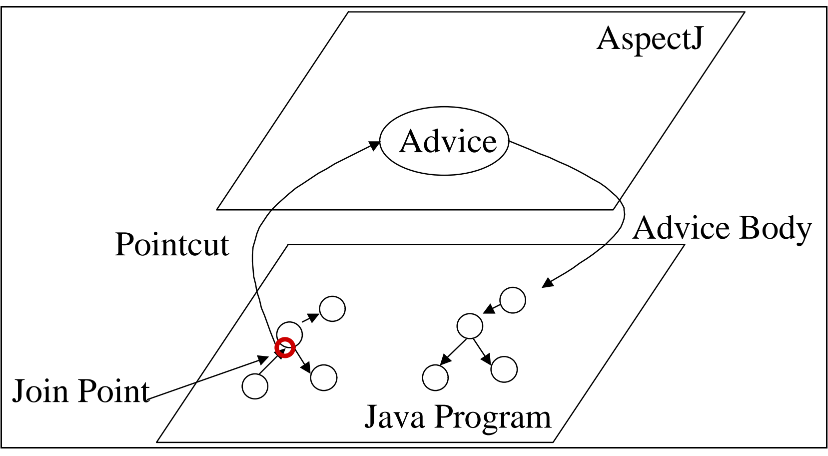

Figure 26: A Conceptual Model of AspectJ

We have introduced the AOP tools Fred, DemeterJ, DJ and AspectJ in this section. Examples of how these tools are used to specify a program have been presented. It is important to understand the fundamental concepts behind these tools for our exploration of the aspect oriented programming tools concepts. In this exploration we will attempt to understand the strengths of these concepts in the subsequent chapters.

In these subsequent chapters, we will translate some examples of Fred to a graphical notation and AspectJ to better understand the semantics of Fred. Then, we will attempt to translate Demeter traversals to AspectJ and explore the possibility of translating AspectJ programs to DemeterJ. This leads us to the analysis of Join Point Model and the "Program as a Journey" metaphors. Next, we introduce DAJ that integrates Demeter concepts with AspectJ. Because of the relatively poor performance of DJ compared to DemeterJ, we will analyze the performance of DAJ relative to DJ and DemeterJ. From the discussions and experiments presented, we apply a graph theoretical view to develop an analysis methodology for aspect

oriented programming tools with a Four Graph Model of Programs (4GMP). Finally, we conclude with conclusions and suggestions for further research.

TRANSLATING FRED

Fred [1 Orleans02] is an extension to Scheme that allows programmers to program incrementally using decision points developed by Doug Orleans. We will attempt to translate some example code to a graphical notation and AspectJ, as a starting point of this thesis. The translation process from Fred to a graphical notation will bring out implicit semantics within the language, while translating it to AspectJ will allow us to explore the power of AspectJ. Because this is an exercise to learn about Fred's semantics, it is not necessary to create a perfect graphical notation for Fred.

Translating Fred to a Graphical Notation

The idea behind the graphical notation for Fred arose out of the fact that many people like to visualize their programs. Languages such as UML arose out of this need for visualization of programs. The purpose of this exercise is to discover the semantics of Fred that is not immediately apparent.

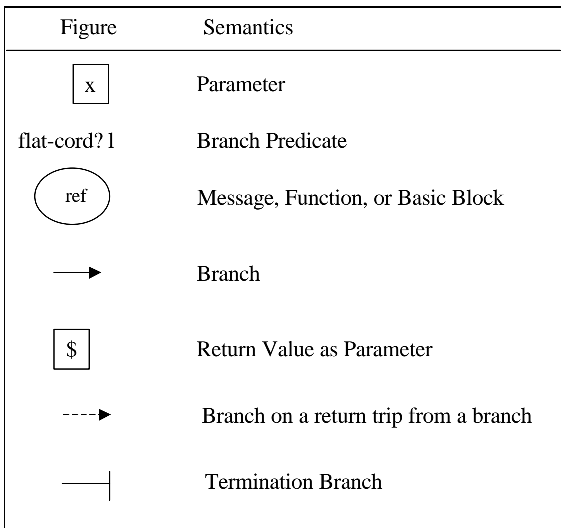

Figure 27: Primitives for Graphical Expression of Fred

In order to describe Fred, we have broken down what actually happens within each function. In Figure 27, we have defined some graphical primitives for this translation process. We have a circle with a function name to represent the function. This is a node within our graphical

representation. Next, we need a way to connect these nodes together and we use an arrow to signify a branch or a function call. Since each branch may have a predicate, a Scheme style predicate without the parenthesis is one of the labels for the branch. We also need to deal with parameters and return values. A variable with a box around it signifies parameters passed to the function. The dollar sign with a box around it signifies the value returned from a function call. This is used for the branch that is visited after a return from the original branch. You can think of this as a "side trip" to another node during the return trip. Lastly, we need something to signify that the trip has ended and it should start the return trip. Therefore, a termination branch was added.

Translating the Factorial Example

As an example, we have translated the Factorial Example from Figure 6 to the graphical notation. In Figure 28, the Scheme expressions are shown on the left and their graphical equivalent is shown on the right. The first Scheme expression, `(define-msg fact)` defines a node named fact. The second defines the termination branch on the left with the predicate `(= x 1)`. If this predicate is true, the "traveler" should stop and start its return trip. The last Scheme expression specifies the branch on the right. Since, this is a recursive function, this branch goes to the function fact back to itself with (- x 1) as the parameter. The return branch from this recursive function call goes to the multiplication function with the parameters x and the value returned from the original branch. This is how we translate code written in Fred to the graphical notation.

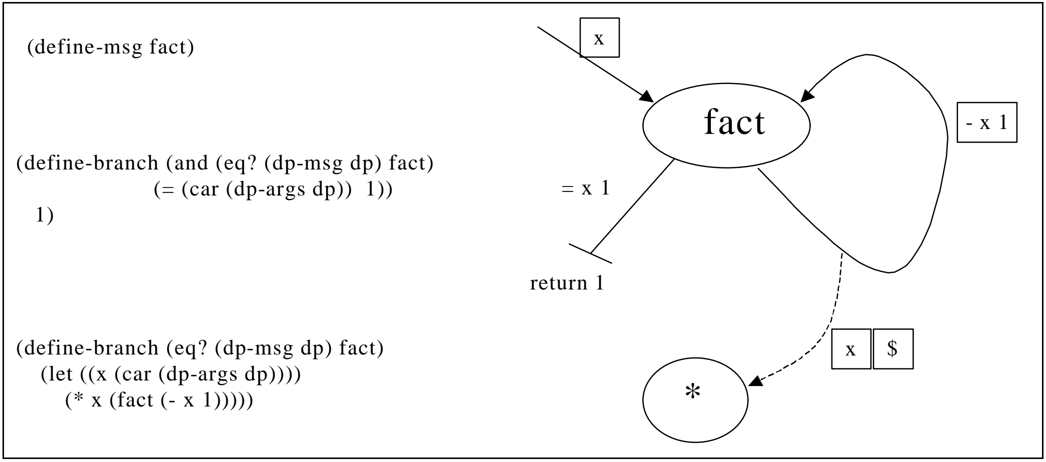

Figure 28: Graphical Translation of Factorial Example

From this translation process, we can make several observations. First, this simple version of Fred does not have any mechanism to group the branches that are associated with a message. Fred associates the branches with messages by using the predicates such as `(eq? (dp-msg dp) fact)`. Because of this feature, a programmer can reuse branches for different messages or nodes by or'ing the message name matching predicates.

Another ramification of this feature is that the programmer is able to place the branches anywhere. There is no requirement or scoping of the branches as you would in other programming languages such as C++. Thus, it is up to the programmer to organize these branches and messages in anyway that is optimal for the particular application.

The second observation to make is that the mechanism to describe the return value is not elegantly represented in the graphical notation. It is not very intuitive and hard to understand what is happening. The reason for this fact is that the ordering of the return branches is not clear from the graphical notation. In addition, the dollar sign for the return value is not intuitive. This awkwardness arises from the fact that features in Scheme are gear towards describing a process at very specific points, i.e. localized. There is no simple way to describe global semantics such as “traverse all the items in the list and retrieve a string” that the programmer wants to express in a program. The programmer has to break this down into steps and explicitly describe what happens at each step, even though this type of semantics is regular and can be automated.

The third observation to make is that the function factorial is a schema for all of the call frames that are generated during run-time. This is the same relationship as the class graph - object graph relationship. It implies that the static call graph is a schema for the dynamic call graph and the CPU is managing the dynamic call graph using the static call graph and the inputs. Since the behavior of any programs maybe described in this manner, it maybe possible to describe all programs using the Demeter concepts.

From these three observations from the simple factorial example, we can see the usefulness of this experiment. We have hit upon some issues in programming languages and we can explore the semantics and concepts behind AOP tools further.

Translating the String Example

We will take a look at a more complex example of the graphical notation for Fred to explore these observations. In the next example, we are translating a Fred implementation of string. We will explore the function `concat`, `len`, and `ref` that concatenate two strings, find the length of a string and retrieve the nth character of a string respectively. For the sake of simplicity, we will assume that other predicates and methods such as `flat-cord?`, `make-cord`, etc. are defined and concentrate on the three functions `concat`, `len` and `ref`.

First, we will present and analyze the function `concat` that concatenates two strings. In this example, the method `define-method` is used. This collapses the two methods `define-node` and `define-branch` into one method. Therefore, it has three parameters: message, predicate statement for the branch, and a list of Scheme statements as the body of the branch. As in the factorial example, the Fred code that we are translating is shown on the left and the translated graphical notation shown on the right in Figure 29.

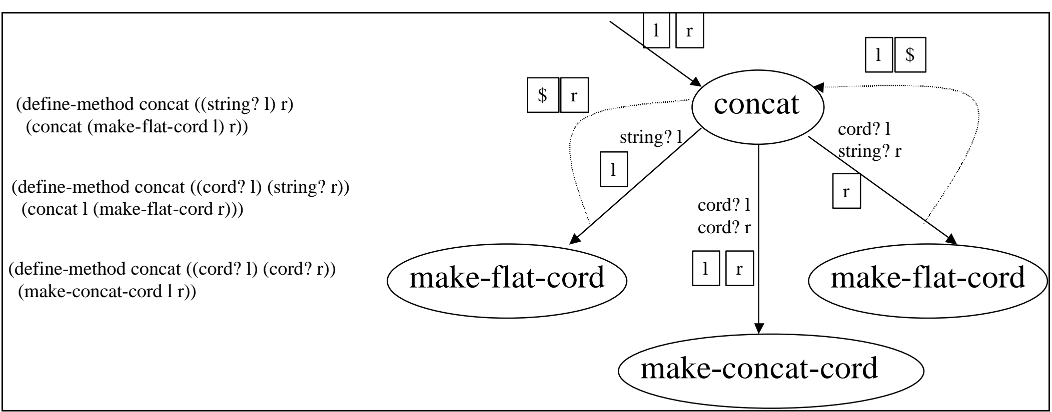

Figure 29: Graphical Notation for concat

In Figure 29, the first define-method statement handles the case where the first parameter is of type string. In this case, we make a `flat-cord`, an internal data structure, and calls itself recursively. The left branch from `concat` to `make-flat-cord` is the translation of this Scheme statement in the graphical notation. The "`string? l`" signifies the predicate that checks for the data type of the first argument "`l`" to be of type `string`. The return branch, with the `$` as the first argument and the "`r`" as the second argument, signifies the subsequent recursive call to `concat` after the "traveler" returns from `make-flat-cord`.

The second Scheme statement is expressed by the right branch of **concat**. This is a similar situation to the first statement, except that the second argument, "**r**", is a **string**. Thus, we call **make-flat-cord** and then a recursive call to **concat** with **r** and **$** as arguments. The third statement is the default case where both of the arguments are **cord**'s. It calls **make-concat-cord** that concatenates the two **cords** together. This function call is represented by the middle branch in Figure 29. The function, **concat** is described by these three Scheme statements, assuming that **make-flat-cord**, **make-concat-cord**, **string?**, and **cord?** are implemented else where.

The second method that we will be translating is the **len** function, which calculates the length of a string. In order to accomplish this task, this function has to visit every **flat-cord** and add the lengths of all the strings stored within the tree of **concat-cord**'s. The base case for this tree traversal method is the case where the first parameter is of type **flat-cord**. For this case, it returns the length of the string stored within it. This case is represented graphically by the branch with **flat-cord? x** predicate in Figure 30 below. Within the body of this Scheme statement, it calls **flat-cord-string**, which is represented by the node by that name. The value returned from this function call is passed to the function **string-length**, which is represented by the return branch with **x** as the argument.

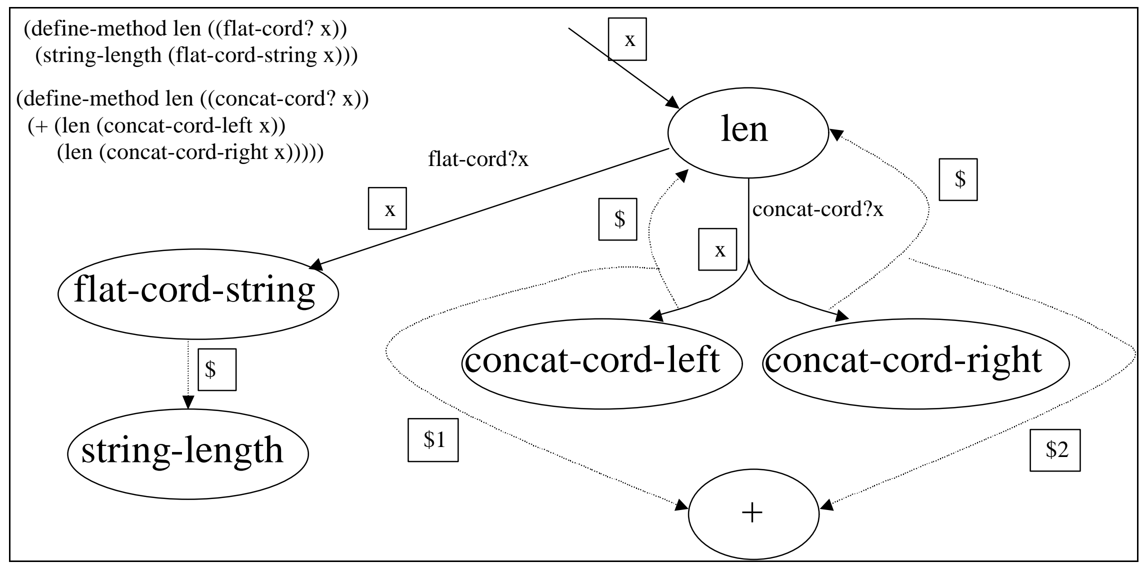

Figure 30: Graphical Notation for len

More complicated branch on the right is representing the second Scheme statement in Figure 30. This branch splits to **concat-cord-left** and **concat-cord-right**. This means that the calls to these two functions can be executed in parallel. The return branch is the subsequent recursive call to **len**. However, we need to call + after this return branch returns. The return branch for the first return branch represents the call to the function +. The two branches that split originally merge at this point. This is because the values from the recursive call to len are needed for the call to +. The resultant value from this + function call is returned as the result of the original function call to len.

The last function that we'll explore is the function **ref** that returns the ith character within a string. There are four cases: the base case where the argument is a **string**, the second case is where the argument is a **flat-cord**, the third case is where the character is within the right **concat-cord**, and the last case is where the character is within the left **concat-cord**. The base case for **ref** is trivial and is not shown.

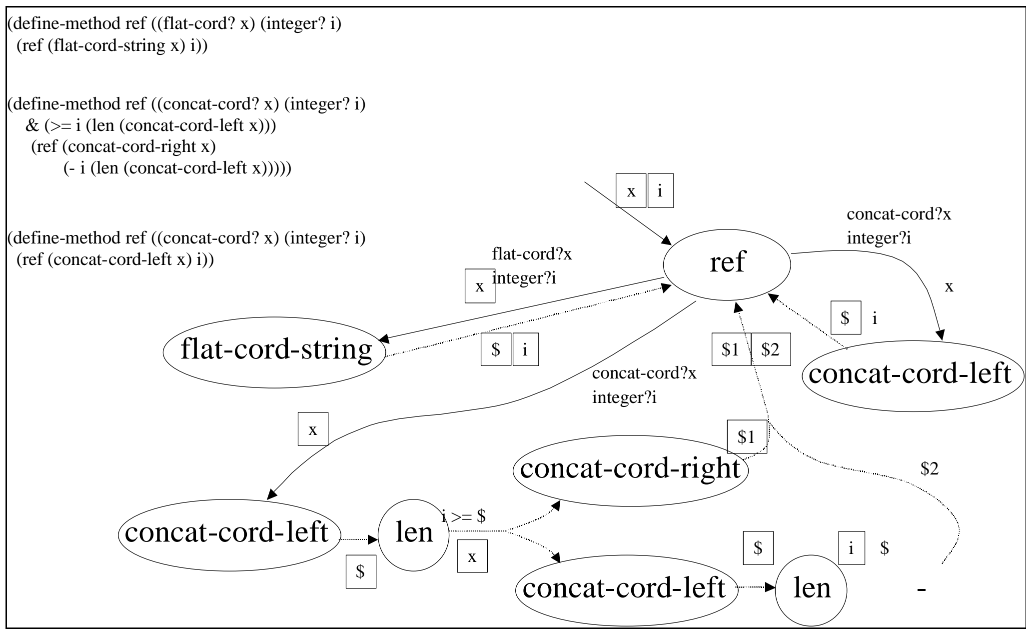

Figure 31: Graphical Notation for ref

The second case where the arguments are a **flat-cord** and an **integer**. We call the method **flat-cord-string** and pass the result to the recursive call to **ref**. This is shown in the graphical notation in Figure 31 by the left branch to **flat-cord-string**. The return branch

with **$** and **i** as arguments is the recursive call to **ref** after it obtains the **string** stored in the **flat-cord**. This recursive call is the base case, which is not shown.

The third case is where the character that we are searching is in the right **concat-cord**. Thus, we check for the type of the first argument to be of **concat-cord** and the second to be an **integer** that is greater than or equal to the length of the left **concat-cord** string. In this case, we traverse the node **concat-cord-right**. This corresponds to the middle branch with "**concat-cord? x**" and "**integer? x**" as the predicates. The first function call is to **concat-cord-left** to obtain the **concat-cord** on the left. The result from **concat-cord-left** is passed to len to obtain the length of the **concat-cord**. This is represented by the return branch from **concat-cord-left** to **len**. If the predicate "**i >=$**", i.e. the integer argument is grater than or equal to the result from **len**, then it calls **concat-cord-right** and **concat-cord-left** with **x** as the argument. The result from **concat-cord-left** is passed to function **len**. The results from **concat-cord-right and** len are passed to the recursive call to **ref**.

This process of translating the more complicated implementation of strings support the three observations we have made earlier. No grouping of branches, awkwardness of return branches and the static call graph as a schema for the dynamic call graph are more pronounced. These three observations have given us insights into some issues in programming languages. The grouping observation points to the scoping of different programming primitives and a need for organizing code. The awkwardness of return branches point to the fact that we tend to program thinking about making progress forward and not thinking about what happens when things return. This awkwardness could also be attributed to the implicit nature of how return values are treated. Lastly, the schema relationship creates opportunities for translation of all programs using the Demeter concepts. These points will be pivotal in creation of AOP Tools analysis methods.

Translating Fred to AspectJ

Now, we will translate some Fred examples to AspectJ. The purpose of this exercise is to understand how Fred and AspectJ are related. We want to investigate how the observations that we have made in translating Fred to a graphical notation are manifested in this process.

First, we will attempt to translate the factorial example. The code in Figure 32 shows how Fred code maps into AspectJ code. The definition of a message translates to a definition of a method. The branches are split into two parts, the predicates and the body. The Fred predicates are translated into AspectJ pointcuts and the bodies of the branches are translated to AspectJ advices.

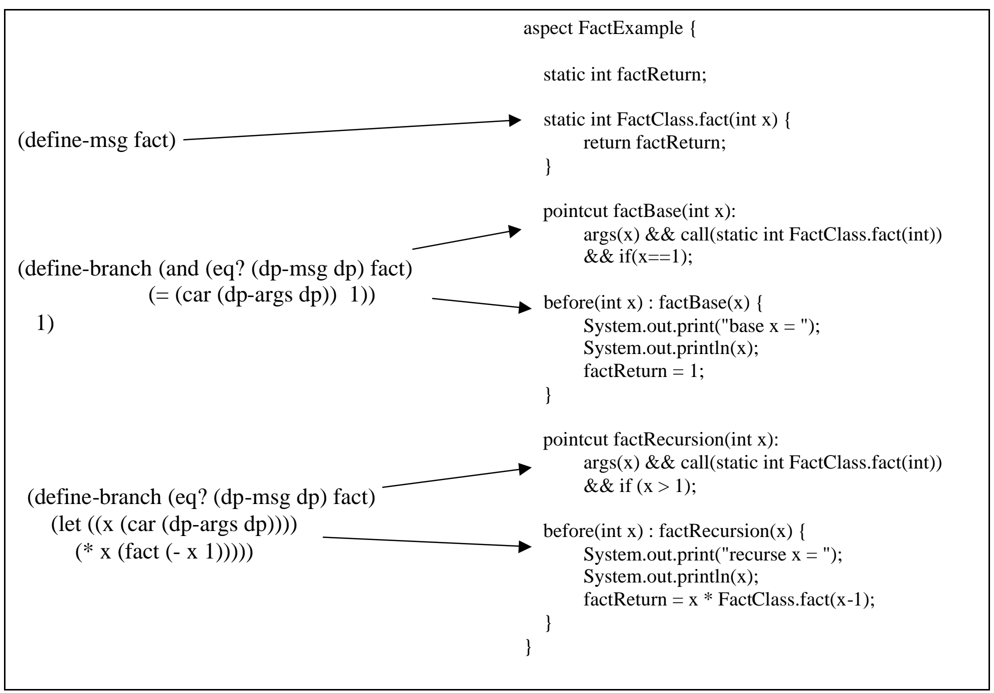

Figure 32: Factorial Example Translated from Fred to AspectJ

Notice the return value `factReturn` for the aspect `FactExample`. It is used as a means to propagate the return value calculated. Even though this data member is static, this program still works because the variable `factReturn` is set right before it returns from the method. This could be converted into non-static data member, but we wanted to make this example as simple as possible.

This explicit handling of the return value is similar to the way we had to use return branches in the graphical notation. However, if we used around methods, this explicit handling of the returns values would not have been needed. In addition, this code does not execute properly as

of AspectJ 1.0. The complier does not handle the case where an execution of an advice is the result of a pointcut describing a method call originating from the same advice.

Next, we will translate the function **concat** from Fred's implementation of string. Again, the mapping from Fred to AspectJ is shown in Figure 33. The process is the same as the Factorial Fxample where we map the branches to pointcuts and advices. Also, we use the same technique to handle the return values from **concat**. However, the **define-msg** call for **concat** has been folded into the first **define-method** call. Thus, the first **define-method** call also maps to the definition of the method **concat**.

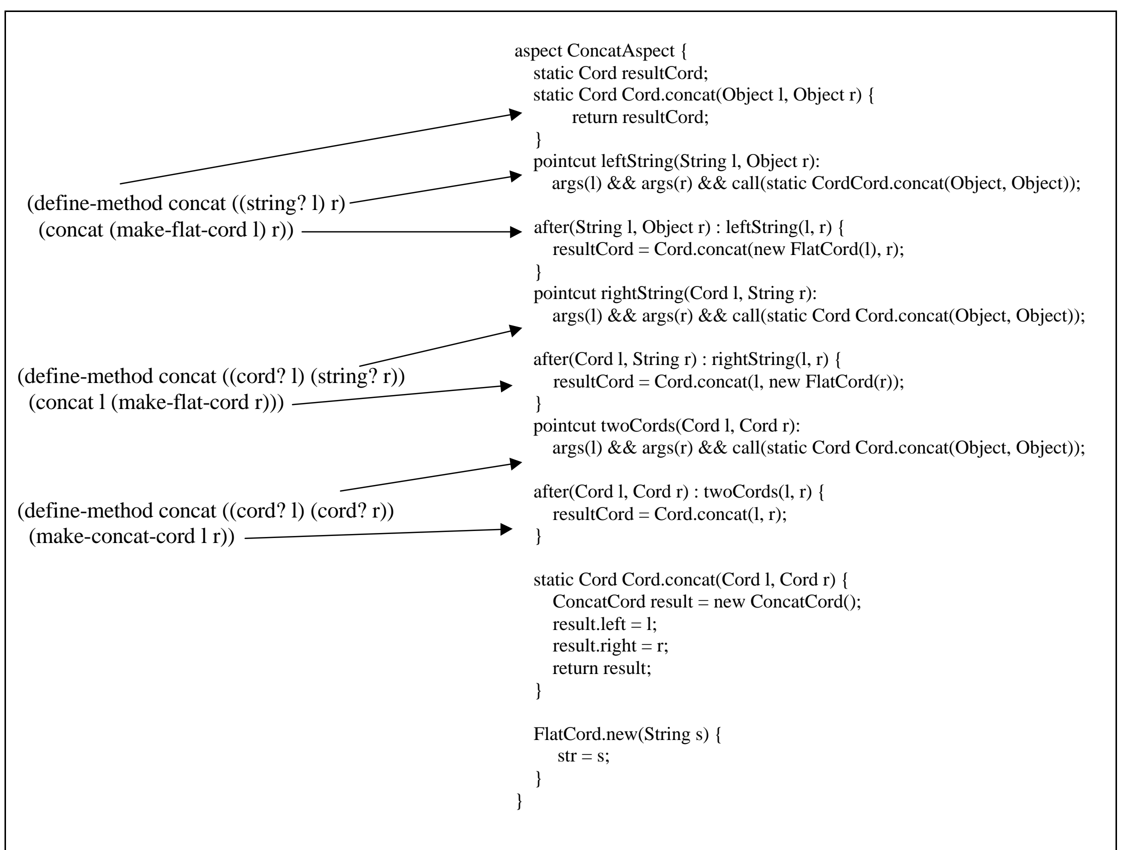

Figure 33: Translation of concat from Fred to AspectJ

Conclusion

In this chapter, we have translated the Factorial and String Examples in Fred to a graphical notation and AspectJ. These two translations of Fred examples to AspectJ have been surprisingly trivial. Explicit and implicit semantics that were evident in Fred and AspectJ were

discovered. Mainly, the values that are returned from a function call are handled implicitly and this caused difficulty in describing the semantics of examples using the graphical notation. Also, this relative simplicity in the translation process opens up the possibility of translating programs written in other AOP tools between each other.

TRANSLATING BETWEEN ASPECTJ AND DEMETER

In this chapter, we will present the process of translating between Demeter style traversals and AspectJ. First, we will present the translation process from Demeter traversals to AspectJ, then a strategy for translating AspectJ code to Demeter traversals. These translation processes are important in understanding the relationship between the two different programming metaphors, JPM in AspectJ and "Program as a Journey" in Demeter. The Join Point Model (JPM) abstracts a program into join points of programming artifacts, such as classes, methods, member variables, etc. However, 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 within a program. We visualize this method centricity of dynamic JPM in Figure 34.

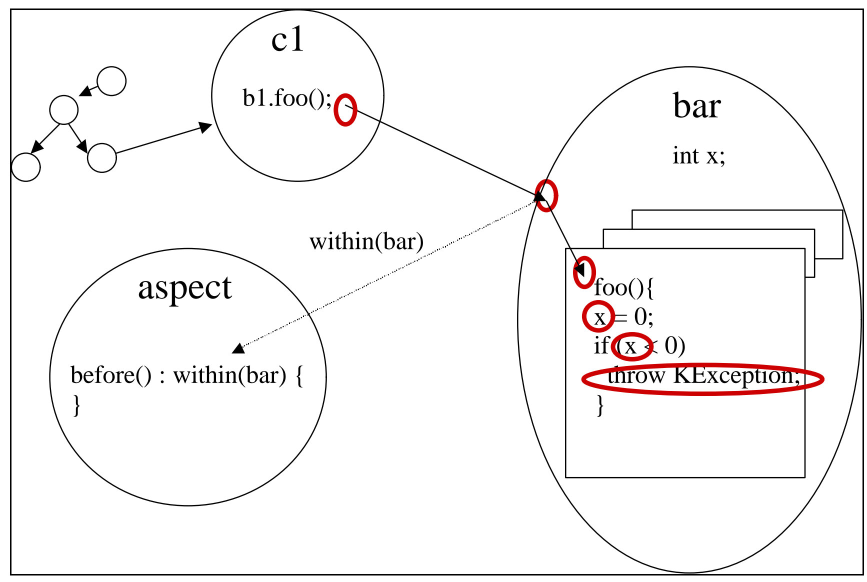

Figure 34: Visualization of JPM

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 during the traversal of the class graph. The advices, the descriptions of what to do at each node, are 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 35 below.

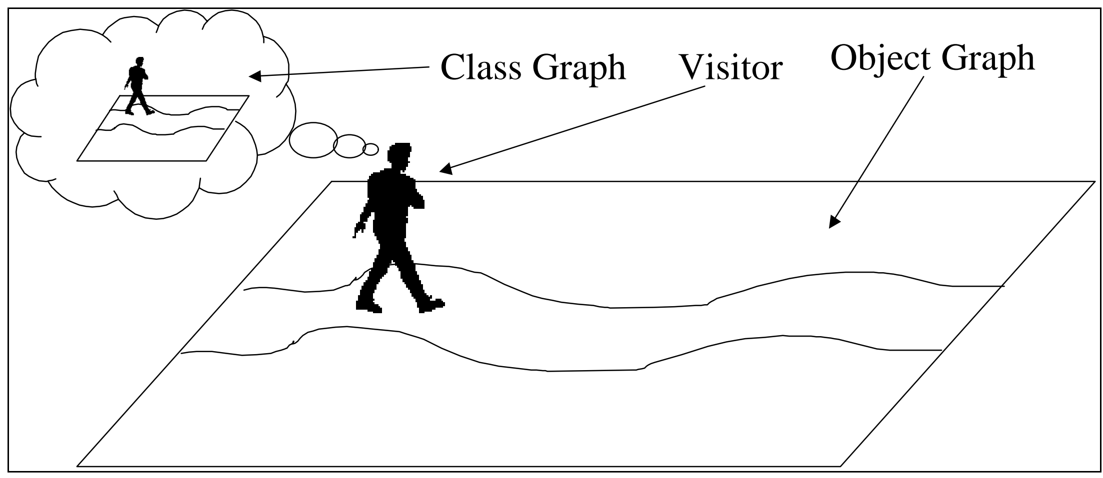

Figure 35: Visualization of Program as a Journey Metaphor used in Demeter

Despite this contrast of the metaphors, AspectJ and Demeter both have advices. JPM's advices are descriptions of a process to be executed when a specified join point is encountered during an execution of a program. However, Demeter advices are executed when the visitor visits certain types within the class graph. These two ways of executing advices seem to be incompatible at first, until you consider how traversals are implemented in DemeterJ. DemeterJ combine 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 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 8. The traversal graph that we will translate is shown in Figure 11. In this translation process, we will use introductions to introduce method calls that will traverse the traversal graph. First, we generate code to traversal all of the has-a edges for each node that will lead us to class `Weight`. Then, we add wrapper methods for each label, i.e. data member variables, so that we may expose the name of the label to be used within a pointcut. We have to use method wrappers because there is no

way to distinguish the label using join points. We also need to take of the inheritance relationship between **Orange** and **Fruit** by calling **super.t1()** in the traversal method **Orange.t1()**. This will handle the case where the **Fruit** reference in basket is referencing an **Orange** object. The implementation of this traversal and the pointcuts that describe the relevant join points are shown in Figure 36.

```
public void Basket.t1(){                   ──────────▶ call (void Basket.t1());
  if (f != null) t1_crossing_f();
  if (f2 != null) t1_crossing_f2();
}
public void Basket.t1_crossing_f() { f.t1();}  ──▶ call (void Basket.t1_crossing_f());
public void Basket.t1_crossing_f2() { f2.t1();} ─▶ call (void Basket.t1_crossing_f2());
public void Fruit.t1(){                    ──────────▶ call (void Fruit.t1());
  if (w != null) t1_crossing_w();
}
public void Fruit.t1_crossing_w() { w.t1();}   ──▶ call (void Fruit.t1_crossing_w());
public void Weight.t1(){                   ──────────▶ call (void Weight.t1());
}
public void Orange.t1(){                   ──────────▶ call(void Orange.t1());
  super.t1();
}
```

Figure 36: Implementation of the Basket Example Traversal in AspectJ and the Pointcuts for the Relevant Join points within the Traversal

With the relevant pointcuts, we may create Demeter style visitor to accomplish the task of counting the **Weight** integers within a **Basket** as shown in Figure 37 below. The method **totalWeight1()** that we introduce to the class **Basket** is there 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.

```
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 37: Demeter Style Visitor for Basket Example Implemented in AspectJ

In this manor we may translate Demeter traversals and visitors using AspectJ features and it 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. This new tool that integrates Demeter concepts with AspectJ became DAJ that is presented later in this thesis.

We have verified that we may implement Demeter concepts using AspectJ, but can we implement AspectJ concepts using DemeterJ? We believe that this can be done but not possible with the current implementation of DemeterJ. In order to implement AspectJ concepts in DemeterJ, we need to find the correct conceptual mapping of the dynamic JPM concepts to Demeter. First, we need some kind of graph that we need to traverse. In AspectJ's dynamic JPM, the join points are mostly based on the static and dynamic call graphs. Thus, it would be natural for us to treat the static call graph as the class graph and the dynamic call graph as the object graph. Each node would be a method and labels would be the parameters. This way of describing the call graph is similar to the graphical notation for Fred. Next, we would specify the strategy for traversing this call graph to be "from main to *". This would traverse all nodes reachable from `main()`. Then, the visitor includes advices to execute when it reaches a certain node, i.e. a method.

In this manner, we may translate AspectJ code in terms of DemeterJ. One might ask, "How can we rationalize why this is possible?" We may rationalize this by the fact that the CPU/OS is doing exactly what has been described, except for the advices. The OS reads the program and start to execute the program. During the execution, the CPU/OS manages the data structures and the call frames. These are object graphs and dynamic call graphs respectively. Therefore, the CPU/OS is managing and traversing the dynamic call graph at the same time.

From this view of how programs are executed, we can conclude that Demeter concepts can be implemented using AspectJ and AspectJ concepts maybe implemented using Demeter. However, the predicated method calls, which are not addressed in Demeter needs to be addressed. One approach is to merge the ideas from Fred into DemeterJ. We have shown that Fred maybe translated to AspectJ and we may gain clues for developing this translation methodology.

In Figure 38, we attempt to translate AspectJ Basket Example to a modified DemeterJ. First, we create the call graph by using a modified BNF in class dictionaries. The first three lines in Figure 38 define the static call graph. Each node being a class name, it is now a name of a method. These statements create the connections, i.e. method calls between the three methods. Following the definition of the static call graph, `MyVisitor` is defined similarly as the DemeterJ implementation of the Basket Example. Next, we define a new programming feature called a guide. A guide is defining the predicates that are used to determine whether we choose to traverse down an edge in the call graph. This is consistent with the "Program as a Journey" metaphor in which a local guide makes local decisions about a trip, while the visitor follows along.

The statements in the scope of a guide specify the method, colon, predicate and the body to be executed. The method is the node in which this guide statement should be applied. The colon separates the node from the rest of the guide statement. The predicate specifies the condition in which the body should be executed. The body is a statement that specifies what should be done when the predicate is true. Lastly, the guide statement is ended with a semicolon.

The last guide statement in Figure 38 is the case where any method of the name `getTotal()` should have predicated method invocation if its date member is not equal to null. This

statement is included to show how we may use the star operator and some keywords to replace the first two guide statements. These new guide statements have some semblance to the AspectJ pointcut syntax as we are describing nodes in the static call graph. Lastly, we define the traversal with the visitor, guide, and a strategy.

```
int Basket.getTotal() = f.getTotal() f2.getTotal().
void Fruit.getTotal() = w.getTotal().
void Weight.getTotal() = .

MyVisitor {

    {{int total = 0;}}
    before void Weight.getTotal() {{
    target.get_i();
    }}

    int returnValue() {{
    return total;
    }}

}

Guide MyGuide {
    int Basket.getTotal() :
    (f!=null) f.getTotal();

    int Basket.getTotal() :
    (f2!=null) f2.getTotal();

    void *.getTotal() :
    (datamember!=null) datamember.getTotal();
}

traversal a(MyGuide, MyVisitor) :
    from int Basket.getTotal() to void Weight.getTotal();
```

Figure 38: AspectJ Basket Example Translated to Modified DemeterJ

This modified DemeterJ has predicated execution by using the guide, which is consistent with the metaphor that Demeter is implementing. This incorporates the predicate execution from Fred and the pointcut syntax from AspectJ. Thus, merging these tools into the modified DemeterJ would be a new direction, which DemeterJ may explore.

From this experiment of translating between AspectJ and Demeter, we may conclude that both of these concepts are similar in their power. Then what is the differentiating factor that

an analyst should consider during AOP tools analysis? We suggest that we examine the usability of the metaphors and the economy of expression. These two would be paramount in the success of these tools because of the issues from Human-Computer Interaction and Software Engineering.

We have introduced the metaphors, JPM and "Program as a Journey", used in AspectJ and Demeter respectively. By examining the metaphors and understanding how DemeterJ implements traversals, we were able to translate the traversal and the visitor from the Basket Example to AspectJ. Since, this translation was possible, we presented a way to describe the Demeter concepts in terms of JPM concepts. However, the translation from AspectJ to the current version of DemeterJ was not possible without modifying DemeterJ. We proposed to add the concept of guide to DemeterJ that would allow traversal decision to be made at each node with predicates; similar to the way Fred uses predicates. Thus, there is no significant differentiating factor by their descriptive power. However, other differentiating factors maybe obtained from the field of Human-Computer Interaction and Software Engineering.

In this chapter, we will discuss and analyze the metaphors Join Point Model and "Program as a Journey" being used in AspectJ and Demeter tools respectively. The Join Point Model that is used in AspectJ is a model in which a program is broken down into connections or join points. Wherever things connect, we may describe those points within a program and execute some code before, after or around those join points.

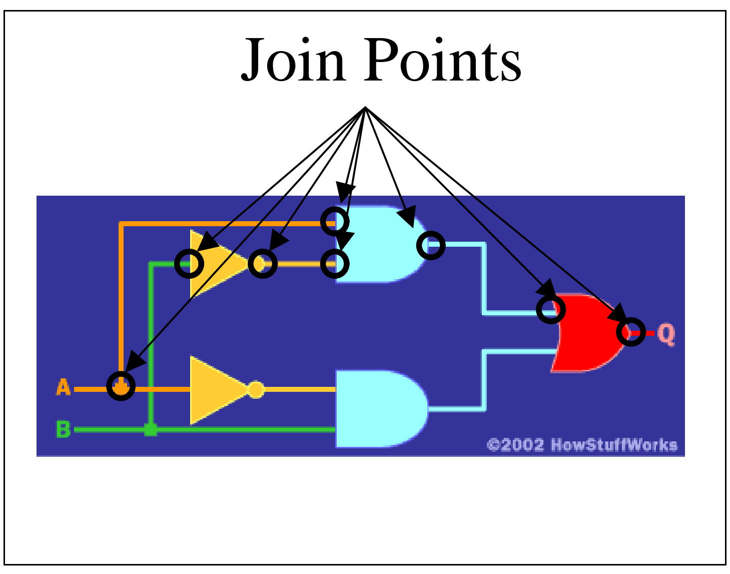

Figure 39: Schematic Diagram of an XOR from [20 HowStuffWorks]

We would call the Join Point Model a "construction" metaphor. Many things in science and engineering use this type of metaphor in which things are constructed with building blocks. Things such as furniture, molecules, and buildings have points in which building blocks that join together. This is a widely used metaphor for scientists and engineers. The "join points" for a schematic diagram of an XOR Gate implemented using AND, OR, and NOT gates is shown in Figure 39. The join points for a caffeine molecule are shown in Figure 40.

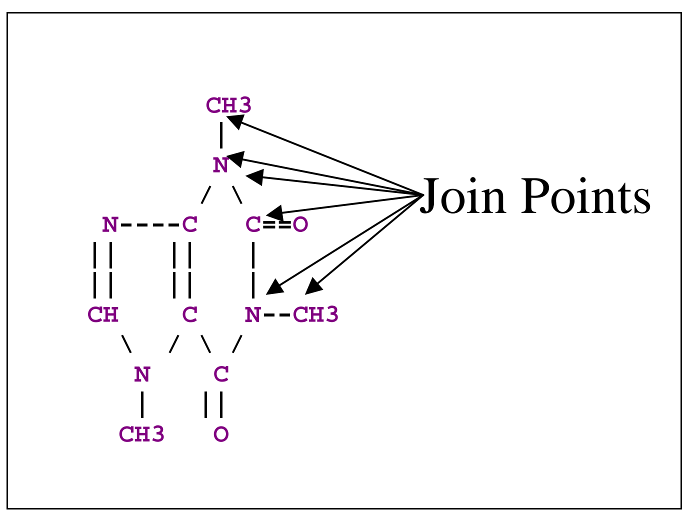

Figure 40: Join points for the Molecular Model of Caffeine from [21 KoffeeKorner]

As a metaphor that is natural for scientists to use, and we will attempt to describe Demeter in terms of a Join Point Model. First, we need things that are connected to create join points. We can use the class graph for this purpose. Thus, join points are the points in which edges and nodes are joined. Examples of join points within a class graph are shown in Figure 41.

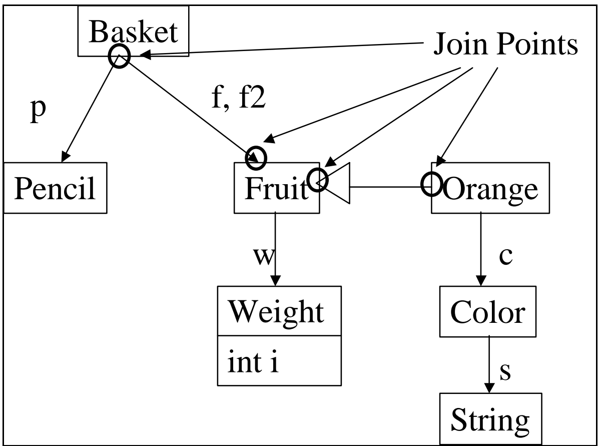

Figure 41: Join points for the Basket Example Class Graph

However, we need to add something to the metaphor in order to add advices. We need to add traversals for this purpose. In AspectJ, the join points are points within the execution of a program. In this model, something is already happening, so adding advices is simple. We just

attach advices at certain points during this process. However it is not so simple for the class graph. There is nothing happening in the graph itself implicitly, until we add the traversals for the object graph. Then, we may say that the traversal uses the class graph as a map to the object graph during the traversal as presented earlier in the ascription of Demeter. In this view, we may use any of the pointcut designators in AspectJ for describing certain points within a traversal. Thus, if we wanted to describe the join points between `Basket` and `Fruit`, the pointcut would be "`this(Basket) && target(Fruit)`".

As we can see, the Join Point Model can describe Demeter perfectly well consistently. In a similar manner, Join Point Model should be able to describe many of the other programming languages. The reason for this is that the computer related fields and other scientific fields use this type of "building" metaphor to describe structures and processes.

The "Program as a Journey" metaphor used in Demeter is an instance of the Journey metaphors as described in [18 Lakoff & Johnson]. The Source-Path-Goal Schema and the metaphors "States are Locations", "Change is Motion", "Purposes are Destinations", "Linear Scales are Paths", "Lovers are Travelers", and "Love Relationship is a Vehicle" are all listed and discussed as commonly used metaphors in everyday conversations. All of these metaphors are instances of the Journey metaphor, where we conceptualize everyday things as a journey.

The concepts used in Demeter are also an instance of the Journey metaphor. We may map each of the major components in Demeter to a person taking a journey. The class graph could be the map in which the person is using to understand how to conduct his journey. The object graph would be the world or place in which he is traveling. The strategy is the directions that the person is following in order to accomplish his objective. The visitor is the person on the journey. The advices for the visitor are the actions the visitor will perform at each point in the journey. The guides, as introduced in “Translating Between AspectJ and Demeter,” are the local guides that will help make localized decisions during the journey. Because of this perfect mapping between Demeter and a journey, Demeter is a perfect example of the Journey metaphor at work.

At first glance, it does not seem to be possible that the "Program as a Journey" metaphor could be used to describe the Join Point Model. However, the JPM as applied in AspectJ uses

the call graph to specify the join points. Therefore, we may apply the "Program as a Journey" metaphor to the call graph. We can rationalize this by the fact that when many things are connected together, the result of these connections can be abstracted as graphs. In Figure 39, the XOR gate is implemented using the primitive AND, OR, and NOT gates. The gates are joined together by connecting wires. The "join points" in this case are the points in which wires and gates are joined together. These schematic diagrams of logical devices are also thought of as a network of gates where each gate is a node and each wire an edge. In this network of gates, we may think of zeros and ones as the traveler that is traveling through the network. Thus, we may apply the JPM and the Journey metaphors to the XOR schematic diagram in Figure 39.

The mapping of roles from the Journey metaphor to the major concepts in the JPM is similar to the way Demeter was mapped. The static call graph is the map in the journey. The dynamic call graph is the place in which the journey takes place. The visitor is the CPU/OS that is on the journey. The join points describe certain points within in the path of the journey. The advices are the actions that visitors should perform at these points in the path. In this manner, we can describe AspectJ's dynamic JPM in terms of the "Program as a Journey" metaphor.

If the JPM and "Program as a Journey" metaphors can describe each other, is there any advantage of using one verses the other? In terms of applicability of the Metaphor there does not seem to be any difference. If we can apply one, then we may apply the other. However, if we examine other factors such as economy of expression, learning curve, usability, etc., there are differences.

Many of these factors seem to favor the "Program as a Journey" metaphor for the general programming language. It is easily understood by more people in general, because of its wide use in everyday language to describe everyday events and relationships. The ambiguous nature of the strategy allows programmers to specify only what they need to accomplish a certain task instead of having to specify more. These are the two major strengths of the "Program as a Journey" metaphor that points it as the metaphor to be used in a general programming language.

On the other hand, the Join Point Model seems to be a programming language for scientists and engineers. The "Building" metaphor is used through out the scientific community. Lakoff and Johnson [18 Lakoff & Johnson] call it "Organization as Physical Structure." Because of its precise nature of specifying the join points and its flexibility in organizing programs it is well suited for a scientist or an engineer. Thus, the decision to use one metaphor over another would have be made in a case by case basis.

Even with strengths outlined for the metaphors used in AspectJ and Demeter, the ability to gain feed back about the way a program is executing is lacking. The traditional way of debugging with debuggers and print statements are things that were added on as an after thought. This is similar to the forward progress thinking of writing programs. The programming languages were designed to allow programmers to specify processes that allow the program to make forward progress. There has not been any successes in adding programming language features that allow programmers to add visibility into their applications easily. Therefore, the visibility of programs during execution needs to be integrated in to the design of the programming language and the metaphor that is used. It would allow programmers to find bugs and fix them quicker, reduce the cost of development and support, and create more robust and cost effective applications.

We have analyzed the metaphors used in AspectJ and Demeter. These metaphors, Join Point Model and "Program as a Journey" can be used to describe each other. These metaphors are also used in other fields and places for other purposes. Join Point Model seems to be well suited for scientists and engineers while "Program as a Journey" would have a general appeal. However, these two metaphors and their applications in AOP do not address the issue of visibility in programs. Thus, this should be taken into account in the metaphor and the programming language feature design of the AOP tools in the future.

In this chapter, we have analyzed the JPM and “Program as a Journey” metaphors that are used in AspectJ and Demeter tools. The two metaphors are very similar and both are present in Demeter and AspectJ. However, the terminology and features used in AspectJ and Demeter tools emphasizes one metaphor over the other. We also found that JPM is maybe applied to many things that are in scientific fields while the “Program as a Journey” is similar to the different types of Journey metaphor that is used by everyone in everyday conversations as

presented in [18 Lakoff & Johnson]. Because of these points, scientist and engineers would easily adapt the JPM, while the "Program as a Journey" maybe easily understood by the general public.

DAJ, pronounced as "dodge", is an application that allows programmers to add Demeter traversals to their Java programs. This is the result of the work outlined in the previous chapter named "Implementing Traversals in AspectJ." DAJ integrates DJ, DemeterJ and AspectJ to implement a system that allows programmers to specify traversals for their AspectJ programs. DAJ extends AspectJ incrementally without modifying the AspectJ compiler. This was possible because of the powerful AOP tools that were available.

Compilation Process

The system architecture was designed such that DAJ will minimize coupling with the AspectJ compiler. Since, the AspectJ compiler is out of our control, this was the decided as the most sensible strategy. 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. Basically, DAJ is a reimplementation of DemeterJ traversals, but using AspectJ as the weaving language instead of the DemeterJ weaving language.

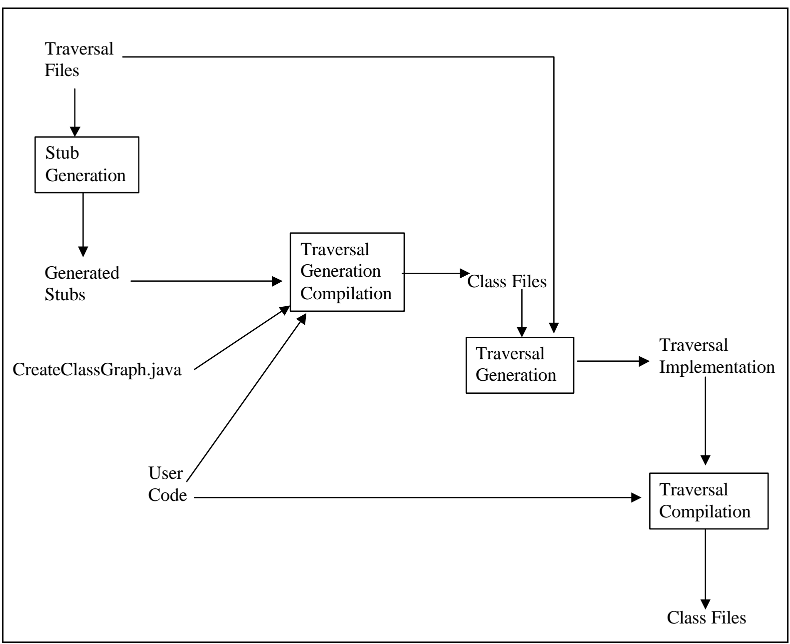

Figure 42: Four Compilation Phases of DAJ

There are four phases in DAJ as shown in Figure 42: 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 `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. 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 chapter "Translating between AspectJ and Demeter."

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 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. First, 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 43 below.

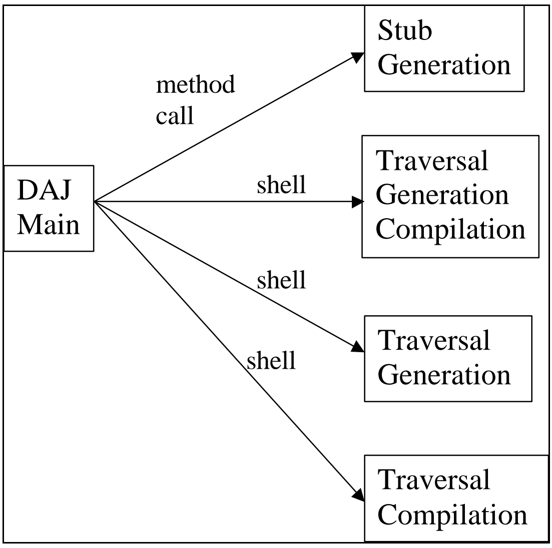

Figure 43: Management of the DAJ Traversal Generation Process

Traversal Specification Following DJ and AspectJ Syntax

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.

```
ClassGraph variable;
----------------------------------
ClassGraph cg1;
ClassGraph defaultCG;
ClassGraph oneMoreDefaultCG1;
```

Figure 44: Default Class Graph Declaration Syntax and Examples

As shown in Figure 44, the default class graph declarations have the keyword "**ClassGraph**" followed by a variable identifier and ends with a semicolon. This is consistent with the way in which one would declare a local variable in Java and similar to the way a programmer would use DJ. This similarity allows Java/DJ programmers to understand this syntax very easily.

The class graph slice form of the class graph declaration looks like a DJ class graph declaration with a class graph and a strategy as shown in Figure 45. Just like the default class graph declarations, class graph slice declarations start with the keyword "**ClassGraph**" and a variable identifier. Then, it has what looks like an assignment operator, equals symbol, and a call to the constructor for the class **ClassGraph**. The constructor has a **ClassGraph** variable and a strategy in string format as the two arguments as shown in Figure 46.

```
ClassGraph varible = new ClassGraph(cg_var, “strategy”);

ClassGraph cg2 = new ClassGraph(cg1, "from A to B via C");
ClassGraph cg3 = new ClassGraph(cg2,
              "from Me via Telephone to You");
```

Figure 45: Class Graph Slice Declaration Syntax and Examples

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 46. 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.

```
declare traversal variable: “strategy”;

declare traversal t1: "from A via C to B ";
declare traversal myTraversal1: "from Me via EMail to You
```

Figure 46: Default Traversal Declaration Syntax with Examples

The second form of the traversal declaration is the one where a class graph that was declared earlier is used. The difference from the default traversal declaration is the class graph identifier within parenthesis between the traversal variable identifier and the colon as shown in Figure 47. This has the same conceptual model as calling a function. Thus, the programmer may think of this type of traversal declaration as passing a class graph to the traversal. The class graph

that is passed would be used during the traversal generation instead of the default class graph obtained from DJ. This syntax is shown in Figure 47.

```
declare traversal variable(cg_var): "strategy";

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

Figure 47: Syntax and Examples of Traversal Declaration with a Class Graph as an Argument

As of version 1.1, 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 advices 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 process by DAJ are specified in Figure 48.

- **`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 48: 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 advices 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 1.2.

Next, the user may declare visitors as shown in Figure 49. 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 49: Visitor Declaration Syntax with Examples

Next, we add another traversal declaration type to allow visitors as specified in Figure 50. 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 50 would produce methods **void t1()** and **void t1(BasketVisitor)**.

```
declare traversal variable(cg_var, visitor): “strategy”;

declare traversal t2(cg2, BasketVisitor):
     "from Here to There via Points";

declare traversal myTrav(default, MyVisitor):
     "from A via X to B";
```

Figure 50: 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 51. 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 and it would be outside the scope of this thesis.

```
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 51: 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 52 will generate the traversal code similar to the one shown in Figure 36. Thus, we may replace the traversal code in Figure 36 with the DAJ traversal file in Figure 52 below. The other user code may stay the same.

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

Figure 52: Traversal File for the Basket Example

Conclusion

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 is possible because of the powerful features of aspect oriented programming.

PERFORMANCE ANALYSIS OF DAJ

In [24 Orleans], the performance of traversals using DJ was determined to be 20 to 30 times slower than DemeterJ. Therefore, the DAJ implementation of the traversal was compared with DJ and DemeterJ implementations. This performance analysis is important, since the efficiency of traversals is central to the performance of the applications using these tools. We will describe the methodology used to measure the relative performance, describe the results that have been obtained, and describe the analysis done on those results.

Performance Measurement Methodology

In order to measure the performance of traversals, we measure the amount of time it takes to perform a certain task with the traversal. First we define a class graph that allows us to create large object graphs. We want to vary the size of the object graph and measure the elapsed time. This will allow us to plot the results in a graph and find the relative performance between the different tools. Next, we define a traversal to accomplish some task, such as counting the number of leaf nodes within the object graph. This object graph happens to be a parse tree, because the class graph is implemented using a class dictionary in DemeterJ. Then, we implement the traversal using the three tools, DJ, DemeterJ and DAJ. We also implement the visitors to go along those traversal implementations. After that, we create an object graph and execute the three implementations of the traversal. Finally, we measure the elapsed time between start and the end of the traversal.

The class dictionary used to generate the object graph is shown in Figure 53. The top level class is the class `Container`. It contains a list of `ContainerIterms` that is surrounded by the strings "container" and "end". Each `ContainerItem` can be either a `SubContainer` or an `Item`. A `SubContainer` sentence starts with the string "sub" followed by a `Container`. This is the recursive part of the class graph that allows the object graph to grow to any size. Finally, an `Item` can be `I1`, `I2`, or `I3`. Each of these `Item` classes does not have any data members. However, they do define strings that allow the parser to distinguish between each of the `Items`.

```
Container = "container" List(ContainerItem) "end" .
ContainerItem : SubContainer | Item.
Item : I1 | I2 | I3.
I1 = "i1".
I2 = "i2".
I3 = "i3".
SubContainer = "sub" Container.

List(S) ~ { S }.
```

Figure 53: Class Dictionary of the Class Graph Used in Performance Tests

An example input file for the class dictionary in Figure 53 is shown in Figure 54. It defines a top level container with I2, I3, SubContainer, I1 and I2 objects. As demonstrated in the example, it is very easy to create large object graphs with this class dictionary.

```
container i2 i3
  sub container
    sub container
      sub container
        sub container i1 end
      end
    end
  end
  i1 i2
end
```

Figure 54: Sample Input File for the Class Dictionary in Figure 53

The number of nodes in the object graph is determined to be the number of objects of type `Container`, `ContainerItem`, `SubContainer`, `Item`, `I1`, `I2` and `I3`. These class types were chosen because these are the node types that a programmer would create advices for. Even though an object of type `I1`, `I2` and `I3` are also of type `Item` as well. Thus, adding a node of type `I1` would add two nodes in the traversal graph. Other classes that were generated by DemeterJ to implement the list used in the class dictionary were not counted. Since, we only care about relative performance as we increase the object graph, they should not impact the final analysis.

We selected the traversal with the strategy "from Container to *". It forces the traversal to touch all of the nodes within the object graph. The performance of the traversal should be apparent because of this property. First, we implemented the traversal using inlined visitor in DemeterJ as shown in Figure 55. The inlined visitor initializes the `total` to be zero, then increments the `total` before `I1`, `I2` and `I3` type objects. Then, it returns the total sum after the traversal has completed.

```
Container {
  int demeterjTotalLeafNodes() to * {
        {{ int total = 0; }}
        before { I1, I2, I3 } {{

           total++;

        }}
        return int {{ total}}
  }
}
```

Figure 55: Traversal Implemented with Inlined Visitor in DemeterJ

The elapsed time was calculated by obtaining the current time in milliseconds before and after the traversal. Then, we calculated the difference between the start time and the finish time. The current time was obtained by calling the static method `System.currentTimeMillis()` in the Java API. This method gives us the resolution of our measurement to be milliseconds. Thus, the traversals being tested should take at least several milliseconds to obtain meaningful results.

The DJ implementation of the traversal is in two parts. First, the visitor was defined using DemeterJ and the DJ API was used to calculate and execute the traversal. The DemeterJ statements for declaration of the `DJVisitor` and the definition of the advice methods are shown in Figure 56. The first statement is a DemeterJ class dictionary statement that defines the class `DJVisitor`. It inherits from the visitor interface defined by DJ. The following statements are from the DemeterJ behavior file that defines the advices for the visitor to count the `I1`, `I2` and `I3` objects. Finally, an accessor method to retrieve the total from the visitor is defined.

```
DJVisitor = <total> int extends edu.neu.ccs.demeter.dj.Visitor .

DJVisitor {
   public void before(I1 i) {{
        total++;
   }}
   public void before(I2 i) {{
        total++;
   }}
   public void before(I3 i) {{
        total++;
   }}
   public int getTotal() {{
        return total;
   }}
}
```

Figure 56: DJ Visitor Implementation in DemeterJ Class Dictionary and Behavior file Statements

Next, we use regular Java code using the DJ API to execute the traversal given a **Container** object **c** as shown in Figure 57. This **Container** object **c** is the root of our object graph. First we create a new **ClassGraph** object and a **DJVisitor** object. Then, we obtain the current time, calculate the traversal graph, traverse the object graph, and finally obtain the finish time. The calculation time for the traversal graph is included in the elapsed time, because this traversal time has to be calculated by the user at least once. In DemeterJ or AspectJ, the traversal graph is calculated at compile-time.

```
ClassGraph cg = new ClassGraph(true, false);
DJVisitor v = new DJVisitor();
long start = System.currentTimeMillis();
TraversalGraph tg = new TraversalGraph("from Container to *", cg);
tg.traverse(c, v);
long end = System.currentTimeMillis();
```

Figure 57: DJ Code to Execute the Traversal on the Object Graph Rooted at Container c

Lastly, we implement the traversal in DAJ. The same traversal is implemented similarly to DJ. A visitor is defined, and then the traversal is implemented by specifying it in a.trv file. The **DAJVisitor** is implemented in the same manner as **DJVisitor**. First, the class is defined in

the class dictionary and then the advices and the accessor method for the total is defined in the DemeterJ behavior file as shown in Figure 58 below.

```
DAJVisitor = <total> int.

DAJVisitor {
   public void before(I1 i) {{
        total++;
   }}
   public void before(I2 i) {{
        total++;
   }}
   public void before(I3 i) {{
        total++;
   }}
   public int getTotal() {{
        return total;
   }}
}
```

Figure 58: DAJ Visitor Implementation using DemeterJ Class Dictionary and Behaviroal File Statements

The traversal was modified slightly to bypass the interfaces that the DemeterJ generated list classes implement. Thus, the strategy changed to "from Container bypassing {java.lang.Object, java.util.Enumeration} to {I1, I2, I3, Container}". This should traverse to all relevant nodes in the object graph. The .trv file to generate the DAJ implementation of the traversal is shown in Figure 59.

```
aspect performance {
   ClassGraph default;
   Visitor DAJVisitor;
   declare traversal dajTrv(default, DAJVisitor) :
        "from Container bypassing { java.lang.Object, java.util.Enumeration } to { I1, I2, I3, Container}";

}
```

Figure 59: DAJ .trv File to Implement the Traversal

Experimental Results and Analysis

In our experiment, the object graph was varied from 10,000 nodes to 20,000 nodes by 5,000 increments. Each traversal implementation's elapsed time was measure on a SunBlade 100 running SunOS 5.8. The result of this experiment is shown in Figure 60 below.

OG Size	DemeterJ	DJ	DAJ
5000	23	3577	32
10000	40	7908	48
15000	55	11863	62
20000	60	14905	70

Figure 60: Table of Results for the Performance Experiment

The values in Figure 60 is graphed in Figure 61. In this graph, we can see that the running time for the DJ implementation increases linearly as we increase the number of nodes in the object graph. However, the rate of change of the increase is much larger than for DemeterJ or DAJ. The magnitudes of the elapsed times for each object graph are much lower as well.

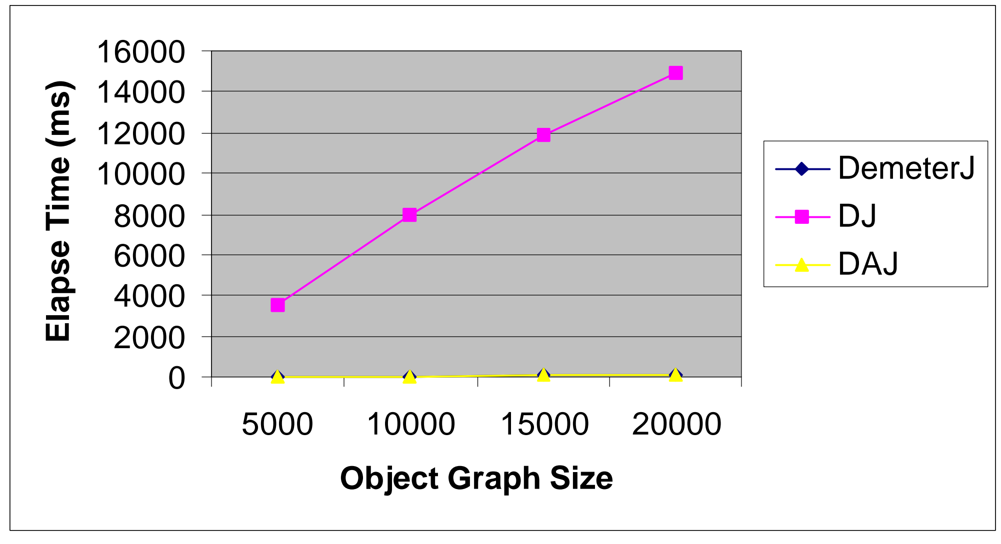

Figure 61: Graph of Elapsed Time of DemeterJ, DJ, and DAJ Implementation of the Traversal Varied with Object Graph Size

The results for DemeterJ and DAJ have been graphed in Figure 61 to compare the results. The elapsed times for both implementations are very similar. However, there seems to be some constant overhead for the DAJ implementation verses the DemeterJ implementation. This may be due to the way AspectJ implements the introductions and advices. Thus, the DAJ implementation of the traversal has significant performance advantage over DJ for relatively large object graphs, while it performs comparatively similar to DemeterJ.

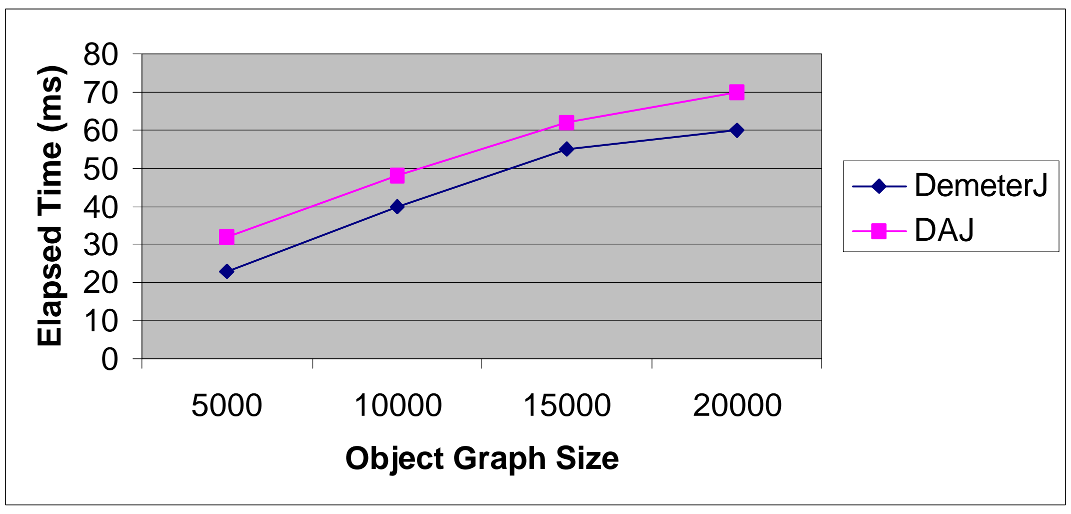

Figure 62: Graph of Elapsed Time of DemeterJ and DAJ Implementation of the Traversal Varied with Object Graph Size

Conclusion

In this chapter, we have described the methodology for measuring performance for traversal implementations. We implemented the traversals and measure the performance using the described methodology. We varied the object graph in which the traversal was executed and obtained the results. From these results, we have found that DAJ performs significantly better than the DJ implementation of the traversal and performs comparatively to DemeterJ implementation. Therefore, the performance of traversals implemented by DAJ should be high enough for most programmers.

THE FOUR GRAPH MODEL OF PROGRAMS (4GMP)

An objective method of analyzing any type of tool requires a model and fundamental rules about how different parts of the model interact. Currently, there are no such models for analyzing aspect oriented programming (AOP) tools. However, there has been some mention of this issue in different publications. In [3 Aldrich], Alderich attempts to provide different types of concerns and illustrate the types of problems that AOP tools must solve. While in [4 Chu-Carroll], Chu-Carroll attempts to organize the program in a novel program organization. A very abstract view of program organization is shown in [5 Black et. al.] by Black and Jones. Sutton and Rouvellou take a novel approach by attempting to apply Categorizational Theory in [6 Sutton et. al.]. All of these approaches attempt to come up with a method for analyzing and thinking about concerns and how we may deal with them.

In this chapter, we propose the Four Graph Model of Programs (4GMP) by applying a graph-theoretical view of programs and merging the graphs in Demeter and AspectJ. These four graphs in 4GMP can be analyzed to formulate fundamental relationships about the graphs within 4GMP and the different features within an AOP tool. This type of analysis will allow us to create a map of the features within AOP tools, which will allow us to identify the strengths and weakness. Thus, the proposed 4GMP and its methodology for feature analysis would be useful for feature inclusion decisions for AOP tools and languages.

First, we will present the 4GMP and how we formulated the model. We will analyze some of the properties of the graphs to understand why this model might be useful. Then, the model would be applied on DemeterJ [11 Lieberherr et. al] [14 DemeterJ] and AspectJ [9 Kiczales2001] [10 Kiczales et. al] [12 Kiczales] to illustrate the usefulness of 4GMP for AOP tool analysis. Finally, we complete the proposal with some conclusions and possible future work.

Four Graph Model of Programs

By using a graph-theoretical view (GTV), we try to abstract everything within in a program into graphs. This allows us to simplify the program and allow us to find relationships between the different graphs. The relationships between these graphs will give us more insight into the

different problems evident in writing programs. We believe that we can use this insight to develop methodologies for analyzing AOP Tools.

First, we try to break up a program and its execution into graphs. The most common boundaries are the compile-time/runtime boundary and the data structure/algorithm boundary. These distinctions come from AspectJ and Demeter concepts of static JPM, dynamic JPM, class graph and object graph. If you break up a program with these boundaries, you get four graphs: class graph, object graph, static call graph and dynamic call graph. These are the four graphs in Four Graph Model of Programming (4GMP). These four Graphs are mentioned in Demeter and AspectJ.

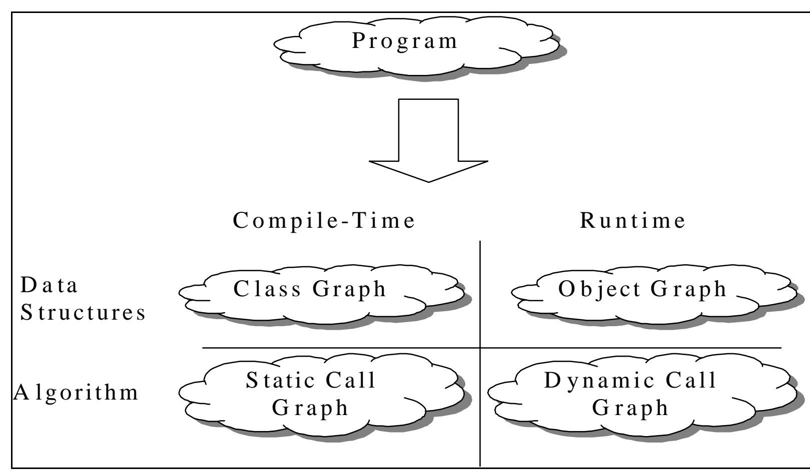

Figure 63: Graphic Decomposition of Programs

To understand how the different graphs are related, one needs to analyze how programs are executed and how programmers make decisions about data structures and algorithms. In DemeterJ, the class graph is used to generate the object graph given some input and the class graph. Another way of looking at this system to look at the class graph as a specification of the pattern of some data structure and the object graph is a valid instance of that pattern.

An analogous relationship exists between static call graph and the dynamic call graph as well. The static call graph is made up of methods and method calls. During the execution of a program, the CPU/OS generates the dynamic call graph using the static call graph and some inputs. Therefore, the class graph specifies all valid instances of the dynamic call graph. These relations are analogous to how grammars specify all possible set of parse-trees that can be

generated with the set of all possible sentences in a particular language. This is how the class dictionaries and the object graphs are viewed in Demeter.

From this transformation of a compile-time graph into a runtime graph, we can create a generic model of graph transformation. The basic model of graph transformation in 4GMP is shown in Figure 64. The basic components in this basic model are input, specification, and output and each of these components can be abstracted as graphs. The processor is the processing logic that uses the specification and transforms the input to generate the output. This processor may be composed of input, specification, processor and output. Thus, it is recursively defined.

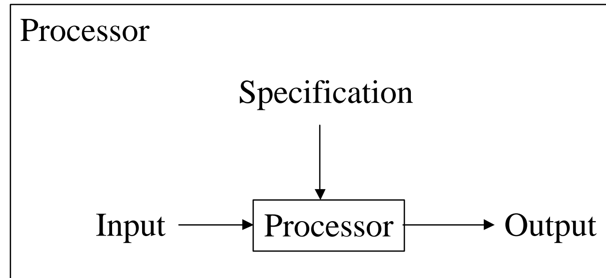

Figure 64: Graph Transformation Model (GTM) in 4GMP

This model can be applied to the current compiler and execution process as shown in Figure 65. It illustrates how specification and input combine to generate output for compiled language such as C. We first start with some input file, which can be in C, C++ or any other compiled program. Then, we use the token specification to transform the linked list of input characters into a linked list or a stream of tokens. The grammar specification then transforms the tokens into a parse tree. Then, the back end of the compiler converts the parse tree into an executable program. In each step, the resultant output data structure tends to be much larger than the specification and the input combined.

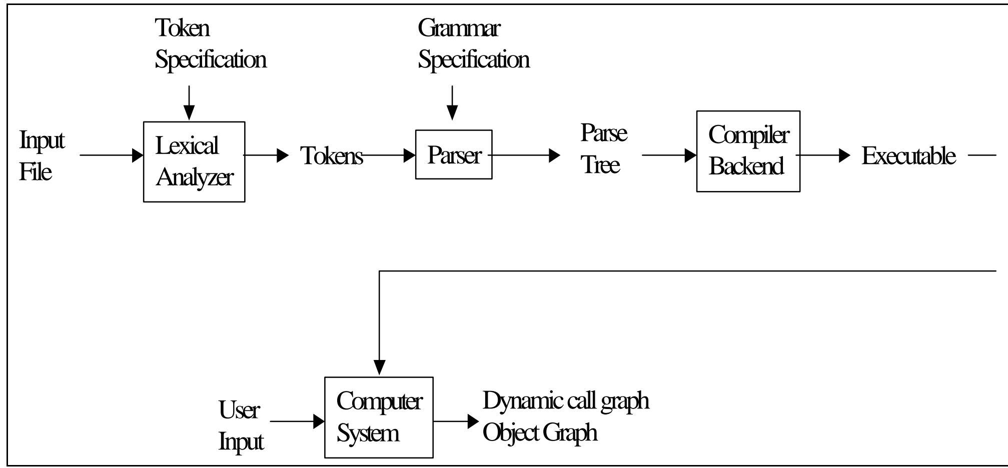

Figure 65: Application of Graph Transformation Model for Compiled Programs

The other boundary in the 4GMP, Data Structures/Algorithm boundary, is a familiar boundary from the undergraduate data structures and algorithms class. In many cases, you need to make decisions on what types of structures you would use to store the information and how you can apply different algorithms to solve the problem at hand. These decisions are of the type where the size of the data structures can be increases to lower the execution time of the program or vice versa. This implies that you can some how reduce one graph by increasing the other. There is still very little understanding of this type of relationship between these two graphs within the 4GMP and further investigation should be pursued.

Factorizational Concerns

When we see a repeated pattern, we factor the common parts out to reduce the amount of code and increase the modularity of the system. This has the effect of creating a more concise description of the program and thus a concise description of the graph. The concerns of this factorization process are called the factorizational concerns. These concerns are major factors that programmers use when making factorizational decisions. These factorizational concerns occur at many different levels within a computer system.

Functions that are called multiple times, inherited methods and data members, parser generator in DemeterJ, wildcards in pointcut designators, and traversal generation in DAJ are all examples of features that are used to address the factorizational concern. When a function in C or Scheme are called multiple times, the programmer has effectively factored out the

commonalities of the code and created a function that may be invoked many times. The inherited methods and data members are also factored from the classes that inherit them. The parser generator in DemeterJ is a specialized factorization of parsing code. It allows programmers to specify a parser with minimal amount of code. Thus, the programmer may choose to use DemeterJ's parser generator instead of writing one him or her self.

The wildcard feature for pointcut designators in AspectJ allows programmers to insert advice bodies to many different points during a program's execution. Thus, all of the method calls have been factored. The traversal generator in DAJ has factored out the regularities of the traversal code. In all of these cases, the user has the option of using some feature to replace regularities in the program.

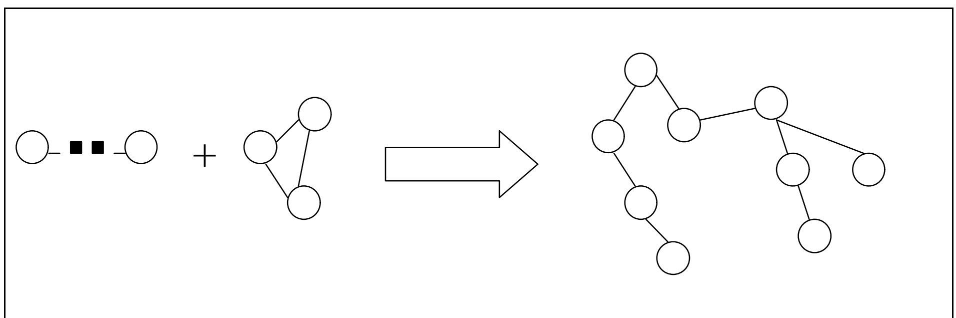

Figure 66: Factorizational Concern Process

The factorizational concern process pictured in Figure 66 should look very familiar. It is the graph transformation model of 4GMP presented earlier. The input is the first graph, the specification the second graph, the output is the graph on the right and the processor is the plus operator. This relationship between the specification and the output is called factorizational concern relationship within 4GMP. From this, we can conclude that there is a factorizational concern (FC) relationship between the class graph and the object graph and between the static call graph and the dynamic call graph.

Organizational Concerns

When a graph gets to be very large and unmanageable, it gets to be very hard to modify or understand it. Sorting and/or organizing are ways to solve this problem. Thus, we can organize the graphs into different subgraphs and we can even create some hierarchy of subgraphs. The

factors in which a programmer needs to take into account when make the decisions for this grouping process is called organziational concerns (OC) in 4GMP.

The organizational concerns (OC) are important, because it allows us to create some type of order. This is necessary for us to divide the program into manageable chunks that we can create and manipulate. The organizational concern is visualized in Figure 67. However, this creates another problem: how do we determine the optimal subdivision of the graphs? More importantly, how do we know when it is optimal?

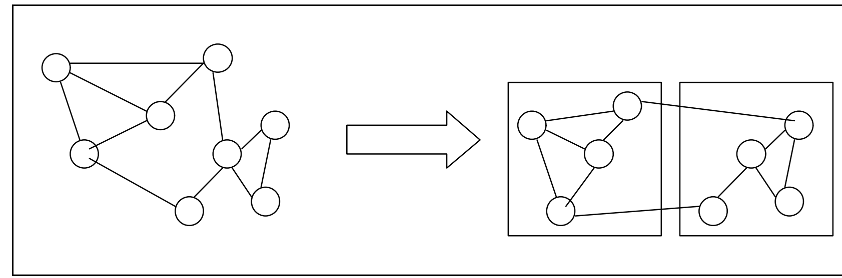

Figure 67: Organizational Concerns in 4GMP

For this problem, we can borrow the concept of cohesion and coupling from software engineering as shown in Figure 68. Under the 4GMP, OC with low cohesion and high coupling is desirable to minimize the propagation of changes and high degree of interaction between collaborators in collaborations, whether the collaboration is a data centric view or an algorithm centric view of the collaboration.

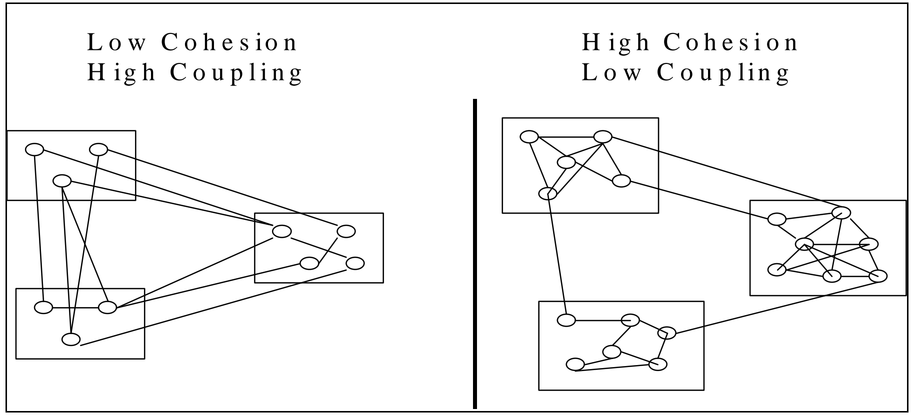

Figure 68 Cohesion and Coupling from [8 Tucker 1997]

Concern Relationship Diagram

From the FC and OC relationships from the four graphs in 4GMP we can create a concern relationship (CR) diagram. The CR diagram of the four graphs in 4GMP is shown in Figure 69. In the CR diagram, the four graphs are connected by either factorizational concern (FC) or organizational concern (OC) relationships.

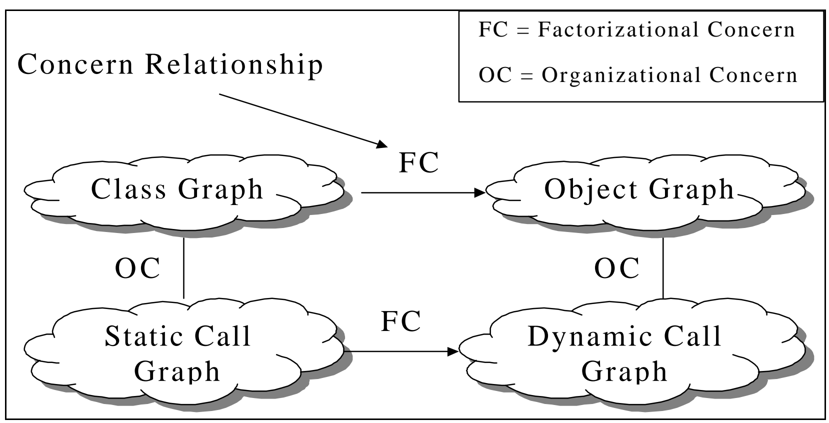

Figure 69: Concern Relationship Diagram of 4GMP

From the CR diagram in Figure 69, the factorizational concerns and organizational concerns seems to be orthogonal, i.e. they do not interfere with each other. As an example, let us analyze the concepts class and inheritance in object oriented programming. Classes are grouping of data and their associated operations. If an object oriented program was re-implemented using functional programming then one of the major changes would be the way in which the program is organized. The functions would not be tied to the data structure except by their arguments and maybe declared or defined anywhere in the source tree. This major difference brought on by the class concept in OOP enforces some organization of the functions and the related data structures. Therefore, the class concept in OOP is an organizational feature, a programming feature that assists programmers in organizing the program.

The inheritance feature on the other hand, allows programmers to factor out common data and operations from the base class. By moving the commonalities from the inherited classes to the base class, the programmer has effectively factored the data and/or their associated functionality from the inherited classes to the base class. Thus, this is a factorizational feature, a programming feature that allows programmers to factor out commonalities within class

hierarchies. Therefore, in object oriented programming, the class feature have an OC relationship, and the inheritance feature has a FC relationship.

The OC relationship that the class feature has is with the class graph and the static call graph. The classes in OOP allow programmers to organize data and methods in a program. The inheritance feature has a FC relationship with the class feature, since OOP allows inheritance on classes. These relationships are illustrated in Figure 70.

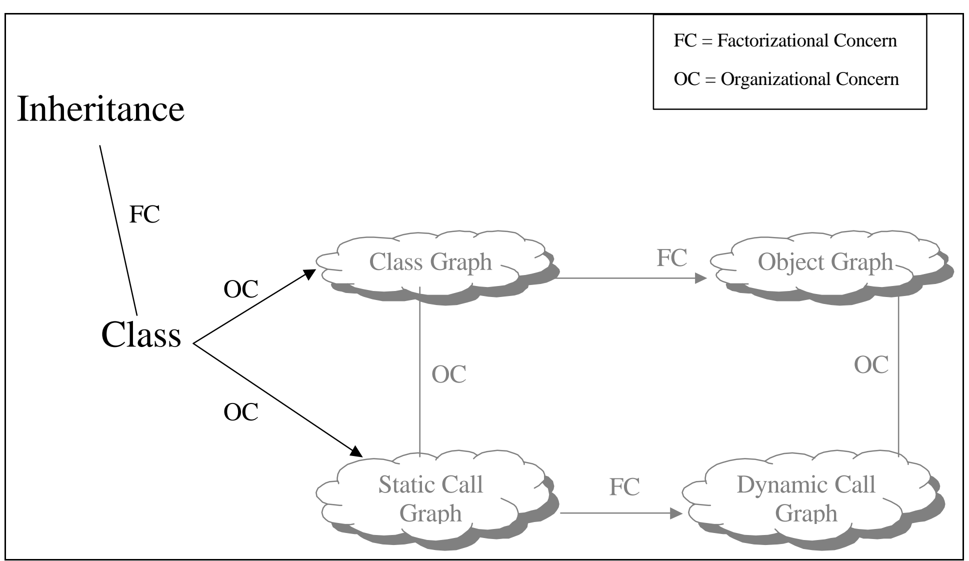

Figure 70: Concern Relationship Diagram for OOP Features Class and Inheritance

One interesting observation to make in the CR diagram in Figure 70 is that the class feature without inheritance only allows the programmer to organize the program in a different way. The class feature with inheritance allows programmers to factor commonalities evident in classes, whether they are data members or methods.

From this analysis, one might ask, “What would be the ideal CR diagram in a programming language?” We propose a conceptual CR diagram in Figure 71. It is a chain of features with FC relationships between them. Then, each of these features would have another feature that allows the programmer to organize the usage of those features. This ideal CR diagram maximizes the benefit of factorization with the sequence of features that are related by FC and allows programmers to organize the code at each factorizational step.

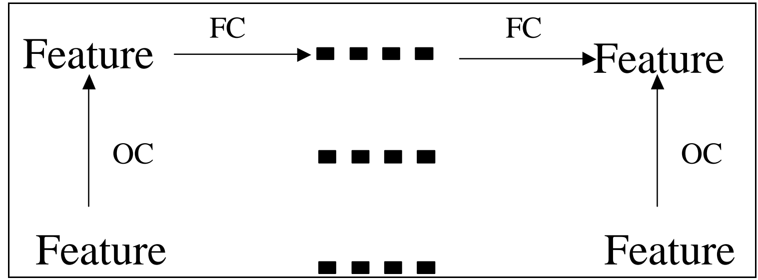

Figure 71: Proposed Ideal CR Diagram

However, we may factor the organizational feature by using only one feature that would organize all of the factorizational features as shown in Figure 72. This new CR diagram would possible be more ideal. This is only a proposal and we cannot know that for sure until there is more research is conducted.

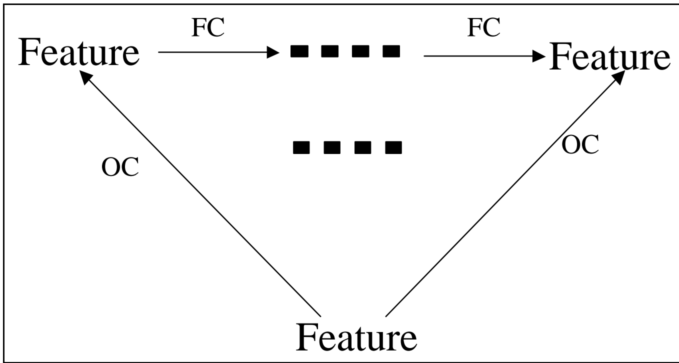

Figure 72: Refined Ideal CR Diagram

AspectJ Analysis

AspectJ is an aspect oriented programming (AOP) tool that was introduced by the AspectJ group at Xerox PARC [9 Kiczales2001] [10 Kiczales et. al]. AspectJ extends the Java language to allow programmers to group crosscutting concerns, i.e. aspects. We will limit our analysis to specific usage of aspects, introductions, join points, pointcuts and advices in AspectJ. We will not discuss the relationship between AspectJ and features in Java.

The introductions in AspectJ could be thought of as adding nodes and edges into graphs. Introduction of methods corresponds to adding nodes in the static call graph, while introducing data members and inheritances are adding nodes and edges in the class graph. In Figure 73, two different types of introductions are shown. First AspectJ statement introduces the data member variables `f1` and `f2` to the class `basket` and their type is `Weight`. This introduction adds a has-a edge in the class graph. The second introduction introduces the method `t1()` to the class Basket. This method create a node in the static call graph with conditional edges to nodes `t1_crossing_f()` and `t1_crossing_f2()`.

```
Weight Basket.f1, Basket.f2;

 public void Basket.t1(){
   if (f != null) t1_crossing_f();
   if (f2 != null) t1_crossing_f2();
 }
```

Figure 73: AspectJ Introductions for Introducing Data Members and Methods for the Basket Example

There are other means of adding methods, data members and inheritance relationships using pure Java, but this method allows programmers to group crosscutting concerns in one aspect. From this description of the introduction feature in AspectJ, we can conclude that introduction is an organizational concern feature. It allows programmers to organize the code in different ways that makes more sense to them.

Join points are points where the edges are joined with nodes in the four graphs of 4GMP. This does not fit into either OC feature or FC feature, because by itself, it does not allow for organization or factorization of the program nor its four graphs. We call this type of feature a specification concern (SC) feature. In general, a join point allows programmer to specify a specific point within the call graph. In other programming paradigms such as OOP or Functional Programming, programmers could not specify these points. They could only modify them by modifying code in those programming paradigms. Thus, SC features allow programmers to specify elements in the tool.

There are many different types of join points in AspectJ. Not only can join points be in the class graph, but it may also be in the class graph and object graphs as well. Programmers can

specify join points for specific classes that are encountered during the program's execution. A join point may even specify specific data values the object of a certain type should have as well. Therefore, the join point has SP relationship with all four of the graphs in the 4GMP.

Pointcuts are a set of join points. It allows programmers to specify the join points that the user is concerned with. Therefore, pointcut is an organizational concern feature. It allows programmers to organize a set of join points. However, the programmers may use the wildcard "*" in a pointcut. This has the effect of factoring similar properties of different join points within a pointcut. Thus, a pointcut with the wildcard "*" is a factorizational concern feature since it allows programmers to specify the set of join points with patterns which is much more compact.

```
call(void  Weight.t1())

call(void  Weight.t1()) && source(Fruit)

call(* *.t1*())
```

Figure 74: AspectJ Pointcut Examples

The first pointcut in Figure 74 describes a set of join points that results from a method call to method `t1()` in `Weight`. The second pointcut describes another set of join points and it holds true if the method `t1()` in `Weight` is called from a method in `Fruit`. These two pointcuts are examples of how programmers can organize different join points together to specify a certain point in the static call graph. The third pointcut describes all join points in the program have has the string "t1" in the method name with any return type and no arguments. This is an example of factorizational feature of pointcuts using the wildcard "*". Thus, the wildcard feature of pointcuts has a factorizational concern relationship with pointcuts.

The advice in AspectJ is used to incrementally add behavior to an existing program. There are three different advice types, before, after and around. Each of these advices is associated with a pointcut. These advices does not factorize nor organize the pointcuts, instead the pointcut allows the programmers to specify the different points in which the advice bodies should be executed. The example AspectJ before advice shown in Figure 75 shows a simple advice with a

pointcut. The semantics of the pointcut specify the point in which the advice body should be executed.

```
before () : call(void  Weight.t1())
        && target(Weight) {
…..
}
```

Figure 75: AspectJ Before Advice

Lastly, we will analyze the AspectJ aspect feature. The aspects contain introductions, pointcuts, and advices. Thus, it is a feature that allows the programmers to organize these other features. By the definition of the OC feature, AspectJ aspect is an OC feature. As we can see in Figure 76, the aspect **FactExample** contains introductions, pointcuts, and an around advices. All of these statements could be defined in any aspect as long as the pointcut used by the advices are defined within the scope of the aspect.

```
aspect FactExample {
    static int FactClass.fact(int x) {
        return 1;
    }
    pointcut factBase(int x):
        args(x) && call(static int FactClass.fact(int))
        && if(x==1);
    int around(int x) : factBase(x) {
        return 1;
    }
    pointcut factRecursion(int x):
        args(x) && call(static int FactClass.fact(int))
        && if (x > 1);

    int around(int x) : factRecursion(x) {
        return x * FactClass.fact(x-1);
    }
}
```

Figure 76: Factorial Example Implemented in AspectJ

From this analysis and the resultant relationships that we have deduced, we may create a CR diagram of AspectJ. This CR diagram is not complete, as it does not account for other AspectJ features such as, inheritance of aspects, abstract aspects, etc. It also does not take into account

all of the features available from the pure Java code that is allowable in AspectJ. The CR diagram of the AspectJ features discussed is illustrated in Figure 77.

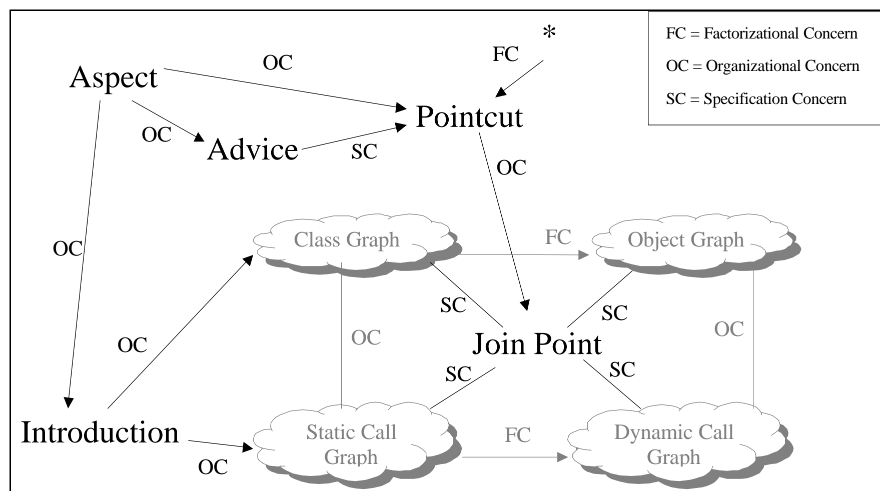

Figure 77: AspectJ Concern Relationship Diagram

One of the things that is apparent in the CR diagram of AspectJ features discussed shown in Figure 77 is that there are many more OC relationships than any other relationships. Thus, much of the features in AspectJ are designed to organize the code in a program. Also, the join point has SC relationship with the four graphs of the 4GMP. This implies that join point may specify anything in those four graphs and this fact make the join point feature very powerful compared to the other features. Not only that, but it is also a bridge that connects the other features aspect, advice and pointcut to the four graphs.

From this CR diagram, we can see that AspectJ features are imbalanced towards join points. Because of this imbalance, it would be much more cost effective to add factorizational features for the other features such as introduction and advice. This would balance out the features and their utility.

DemeterJ Analysis

DemeterJ is a tool introduced by the research group at Northeastern University [11 Lieberherr et. al] [14 DemeterJ]. In contrast to AspectJ that extends Java, DemeterJ creates a new software development process and a new language that is built on top of Java. We will analyze the class dictionary, strategy graph, traversal graph, visitor and advice concepts in DemeterJ.

Class dictionary allows programmers to specify the class graph in a more concise form than allowed by Java or C++. The class dictionary is in a modified BNF format that allows an intuitive way of expressing the class graph. The is-a relationships correspond to alternations for a type, while has-a relationships corresponds to links between different classes. In an essence, a class dictionary factors out the syntax for class declaration and the inheritance relationships. It also does double duty as a parser generator using JavaCC [17 Java CC]. The class dictionary specifying a language that parses XML specification of the Basket Example is shown in Figure 78.

```
Basket = “<basket>” <p> Pencil <f> Fruit <f2> Fruit “</basket>”.
Fruit : <w> Weight.
Pencil = “<pencil>” “</pencil>”.
Weight = “<weight>” <i> int “</weight>”.
Orange = “<orange>” <c> Color “</orange>”.
Color = “<color>” <s> String “</color>”.
```

Figure 78: Class Dictionary of Basket Example that Generates a XML Parser for the Schema Specified by the Class Dictionary

These characteristics of class dictionary, concise definition of the class graph using BNF and automatic generation of a parser make it a factorizational concern feature. It factors out the syntax needed for specifying the class graph in Java. Therefore, this feature of class dictionary has a FC relationship with the class graph. The parser generation feature of class dictionary on the other hand generates methods for parsing the language defined by the class dictionary. Thus, it adds nodes and edges in the static call graph by generating Java code. In effect, DemeterJ has factored out the regularities in the parser code. Therefore, the parser generation feature of class dictionary has a FC Relationship between class dictionary and static call graph.

A strategy is a type of graph that specifies how a traversal should be created given a class graph. A strategy is applied on a class graph and yields a traversal graph as shown in Figure 79. An example strategy, "from Company to Salary" is a strategy that has two nodes Company and Salary. The source of the graph is Company and the destination of the intended traversal would be Salary. In a way, strategy graph specifies all possible traversal graphs, which is similar to the way the class graph specifies all possible instances of the object graph. Thus, a strategy

graph is the result of factorization of the traversal graph and there is a FC relationship between the two concepts.

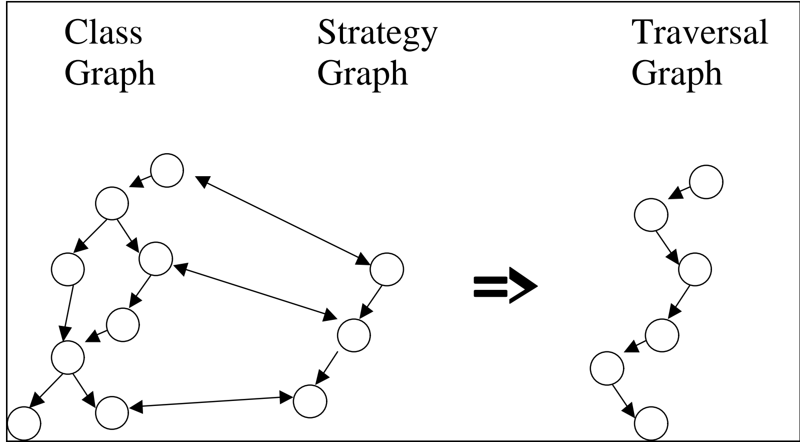

Figure 79: Visualization of the Relationship between Class Graph, Strategy Graph and Traversal Graph

The traversal graph, result of applying a strategy graph to a class graph, is actually a subgraph pattern that is evident in the class graph. Thus, the class graph specifies all the possible traversal graphs, analogous to the class graph and object graph relationship. Therefore, the class graph has a FC relationship with the traversal graph. Strategy graph also has a FC relationship with the traversal graph as well, because of the similar relationship between strategy graph and the traversal graph.

Another feature of the traversal graph in DemeterJ is that DemeterJ uses the traversal graph to generate Java code that executes the traversal. In effect, it factors out the regularity of the traversal methods and it is described in a more compact form by the traversal graph. Thus, traversal graph also has a FC relationship to the static call graph.

As illustrated in Figure 35, visitors in DemeterJ can be thought of as entities that traverse through an object graph using the traversal graph as a map. As the visitor is traversing, the object graph, it invokes advices on particular object types. Similar to the way advices are invoked in AspectJ. The body of the advice specifies the work that the visitor will accomplish during the traversal. Thus, visitor has a collection of advice. This implies that visitors allow the

programmers to organize the advices within DemeterJ. Therefore, there is an OC relationship between visitor and advice.

The differentiation between the traversal graph and the visitor is analogous to the data structure/algorithm differentiation in the four graphs of the 4GMP. The traversal graph is the particular data structure in which the programmer wants to traverse and the visitor is the algorithm that the programmer wants to execute during the traversal. Therefore, there is an OC relationship between traversal graph and the visitor.

The CR diagram of DemeterJ from the DemeterJ features discussed is shown in Figure 80. It does not take into account all of the features of DemeterJ, only the basic features that were discusses in this section.

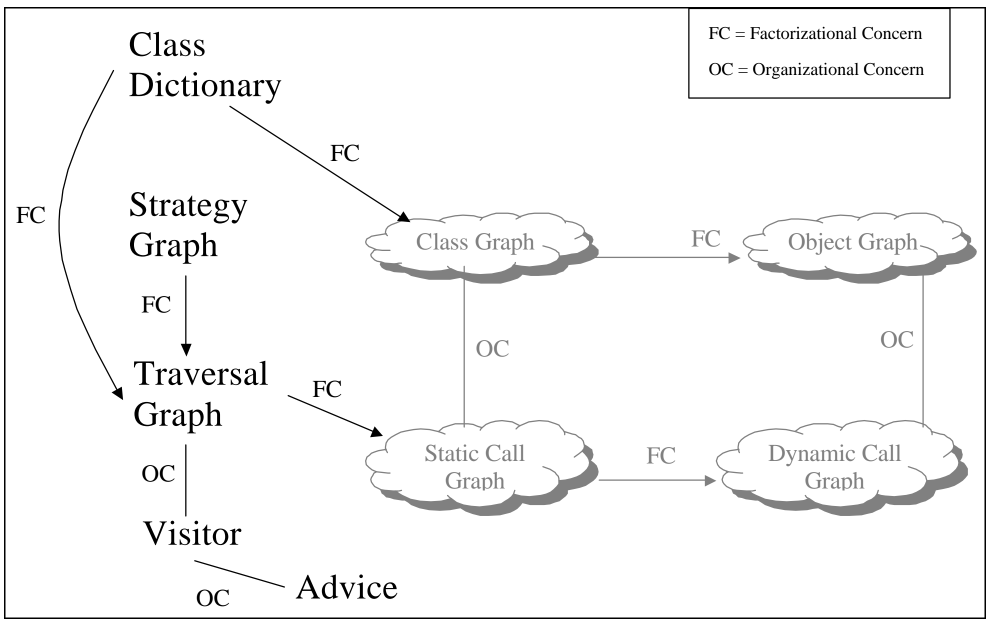

Figure 80: CR Diagram of DemeterJ

From the CR diagram of DemeterJ feature shown in Figure 80, we can see that there are many FC relationships. This seems like an idea situation for a programming language, until one examines the scoping of the features. The source of the FC relationships is the class dictionary and DemeterJ only generates traversals using the class dictionary. There is no DemeterJ feature that allows programmers to factor code that is not related to the class graph. Therefore, adding such feature would fill a gap in the DemeterJ's set of features.

Conclusion

We have combined the graphs from AspectJ and Demeter to create the Four Graph Model of Programs (4GMP) in this chapter.. By analyzing these four graphs class graph, object graph, static call graph and dynamic call graphs, we have formulated two types of concerns factorizational concern (FC) and organizational concern (OC). The organizational concern features allow programmers to organize the code in different ways. The way Demeter allows programmers to separate the data structure description from the algorithm is an example of the organizational concern feature. Meanwhile, the factorizational concern allows programmers to factor commonalities from the program. Features that allow factorization tend to allow programmers to specify a specification and an input to create large regular output. The relationship between the class graph and the object graph is an example of a factorizational concern relationship.

We also presented a third concern, the Specification Concern (SC). This is a concern that allows programmers to specify different things within the tool. Things such as values, method names, join points, function pointers, etc. are all things that maybe specified within programming languages and are SC features.

With these three concerns, FC, OC and SC we may create a concern relationship diagram. These diagrams allow AOP tool developer to visualize how different features address the three concerns. From this visualization, analysis of the available features is possible. This analysis in turn can be used for feature inclusion decisions. This proposed feature analysis methodology shows much promise from its application to the small set of AspectJ and DemeterJ features. However, there are much research need to be conducted in this area to make this methodology useful.

SUMMARY AND FUTURE DIRECTION

In this thesis, we have presented many different experiments, concepts, analysis and results. First, we included the thesis proposal to illustrate the original purpose of this thesis. Then, we introduced the AOP tools Fred, AspectJ, DemeterJ, and DJ, which will be used in experiments through out this thesis. After that, we presented out first experiment in which we translated examples written in Fred to a graphical notation and AspectJ. The ease in which we were able to translate Fred examples to AspectJ led us to the possibility that code written in DemeterJ maybe translated to AspectJ and vice versa. We found that DemeterJ code was translatable to AspectJ. However, we needed to add a new feature called guide and other language features to DemeterJ in order to translate AspectJ code to DemeterJ. From there, we started to analyze the JPM and the Traversal Model of programs to find out more about how these two views of programs are related. The analysis concluded that the two metaphors may describe each other and that the two metaphors could be applied at the same time. In fact, the programs always had the two metaphors present, but the features and terminology used in AspectJ and Demeter tools emphasized one metaphor over the other.

This realization is the theoretical basis for DAJ that integrates Demeter and AspectJ concepts. The implementation of this conceptual integration in DAJ was described and the traversal specification language of DAJ was discussed in detail. Then, we did some performance analysis to compare the relative performance between DemeterJ, DJ and DAJ traversal implementations. We found that the performance is comparable to DemeterJ and much better than DJ. Therefore, DAJ is a successful tool that combines AOP concepts from Demeter and AspectJ and implements traversals that performs almost as well as DemeterJ.

From all of these experiments, from DAJ to various translations, we proposed the Four Graph Model of Programs (4GMP). The 4GMP came about by decomposing a program into the four graphs class graph, object graph, static call graph and dynamic call graphs. From analysis of the relationship between these four graphs, we proposed the two main Software Design Concerns, factorizational concern and organizational concern. We postulated that we can use these concern relationships and the 4GMP to create a concern relationship (CR) diagram that would allow us to visualize the relationship between different features within an AOP tool. CR

diagram was presented for a small set of AspectJ and DemeterJ feature usage. From the AspectJ CR diagram creation process, we discovered the third kind of relationship, specification concern (SC) relationship.

The three relationships, FC, OC, and SC maybe used in conjunction with the 4GMP to create CR diagrams for features analysis. This proposed methodology allows designers of AOP tools to analyze the different features. However, much research is still needed in order to make this feature analysis methodology useful.

This thesis has touched upon many different areas, tools, and concepts. Nonetheless, all of the work presented thus far has led to the proposal of a new feature analysis methodology for aspect oriented programming tools. Because of its relatively young field is gaining momentum, the need for analysis methodologies that will allow designers of the tools to create useful tools is quite important. In order for any analysis methodology to be successful, we believe that the designers must address the organizational, factorizational and specification concerns with the features that they include in their tool.

Furthermore, more research into the different software development concerns such as the three concerns discussed in this thesis is needed. Possible new concerns, such as interface concern that deal with how programs from different organizations interface with each other needs to be analyzed and discovered. Therefore, there is still much research needed in this direction.

BIBLIOGRAPHY

[1 Orleans02] Doug Orleans, "Incremental Programming with Extensible Decisions," First International Conference on Aspect Oriented Software Development, 2002

[2 Hugunin2001] Jim Hugunin, "The Next Steps for Aspect Oriented Languages," Workshop on New Visions for Software Design and Productivity: Research and Applications, 2001, http://www.isis.vanderbilt.edu/sdp/Papers/Papers.htm

[3 Aldrich] Jonathan Aldrich, "Challenge Problems for Separation of Concerns," OOPSLA, 2000

[4 Chu-Carroll] Mark C. Chu-Carroll, "Separation of Concerns: An Organizational Approach," OOPSLA, 2000

[5 Black et. al.] Andrew P. Black and Mark P. Jones, "Perspectives on Software," OOPSLA, 2000

[6 Sutton et. al.] Stanly M. Sutton Jr and Isabelle Rouvellou, "Applicability of Categorization Theory to Multidimensional Separation of Concerns," OOPSLA, 2000

[7 Chaves et. al.] Christina von Flach G. Chaves and Carlos J.P. de Lucena ,"Design-level Support for Aspect-Oriented Software Development," OOPSLA, 2001

[8 Tucker 1997] Allen B. Tucker, "The Computer Science and Engineering Handbook," CRC Press, Inc. 1997, pg 2296, Figure 106.2

[9 Kiczales2001] Gregor Kiczales et al., "Getting Started with Aspectj," Communications of the ACM, October 2001, Vol. 44, No. 10

[10 Kiczales et. al] Gregor Kiczales, et al., "An Overview of AspectJ," Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP). Spring 2001

[11 Lieberherr et. al] Karl J. Lieberherr and Doug Orleans, "Preventive Program Maintenance in {Demeter/Java} (Research Demonstration)," International Conference on Software Engineering, ACM Press, 1997

[12 Kiczales] Gregor Kiczales, Ontology of Aspect Oriented Programming, Private Communication

[13 MzScheme] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[14 DemeterJ] DemeterJ, http://www.ccs.neu.edu/research/demeter.

[15 DJ] DJ, http://www.ccs.neu.edu/research/demeter/DJ.

[16 APLib] AP Library Java Docs, http://www.ccs.neu.edu/research/demeter/DemeterJava/docs/api/.

[17 Java CC] JavaCC, http://www.webgain.com/products/java_cc/

[18 Lakoff & Johnson] Georgy Lakoff, Mark Johnson, "Philosophy in the Flesh," Basic Books, 1999, pp31-34,51-53,60-70

[19 Lieberherr et al.] Karl Lieberherr, Boaz Patt-Shamir, "Traversals of Object Structures: Specification and Efficient Implementation," Northeastern University Technical Report, 1997, http://www.ccs.neu.edu/research/demeter/biblio/strategies.html

[20 HowStuffWorks] How Stuff Works, http://www.howstuffworks.com/

[21 KoffeeKorner] Koffee Korner, http://www.koffeekorner.com/koffeehealth.html

[22 CACM] Communications of the ACM, "Aspect-Oriented Programming," October 2001.

[23 Lieberherr] Karl Lieberherr, "Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns," PWS Publishing Company, Boston, 1996

[24 Orleans] Doug Orleans, Karl Lieberherr, "DJ: Dynamic Adaptive Programming in Java," The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, 2001