

ASPECTUAL CONCEPTS

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

Northeastern University

2002

Supervisor: _____
Karl Lieberherr

Date

Reader: _____
Mitch Wand

Date

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. The purpose of this research is 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, DJ, and Branch Oriented Programming. 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. Branch Oriented Programming attempts to define the program's control flow graph using the predicated execution and around concepts. DJ uses traversals and advice for data structures. 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 tool to another. The simple implementation of factorial function written in Fred and AspectJ with Branches shown below illustrate how one can apply the concepts of branches using AspectJ.

Factorial example in Fred:

```
(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)))))
```

Factorial Example in AspectJ with Branches:

```
class FactClass {}
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);
}
}

```

In this implementation, the FactClass is empty and the factorial method is defined within the FactExample aspect. The two pointcuts and the corresponding around advices are the two possible branches from the method fact().

The close correspondence between the Fred implementation and the branches with AspectJ implementation of the factorial function is very clear in this example. There's almost one to one correspondence between the two implementations. This illustrates the possibility of implementing the Branch Oriented Programming concept using AspectJ and this process of was relatively trivial.

In order to find more of these relationships, a consistent view that one can take to make all of the AOP approaches more clear would be useful. One way to accomplish this is to view these ideas as abstracting the different parts of a program into graphs. The nodes and edges are different, but the basic theme is the same. This allows us to think of all of the different things that one can do with graphs, traversals, mapping, grouping of nodes, sub-graphs, 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.

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 the previous example, we are

looking at the case 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 the pointcut to specify the specific join point that you want to expose to the join point encountered later in the traversal of the dynamic call graph. In the example, we want to expose the Caller *c* to all of the Workers *w*.

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

With traversal strategies specified in [19 Lieberherr et al.] we would specify a strategy and a NodesOf operator:

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

and we would intersect this set of classes with the set of classes in set Worker:

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

When the NodesOf operator is applied to a graph, it returns all the nodes in the graph. It would be useful to extend AspectJ with strategies and NodesOf 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 of the object graph. In all of these cases, the program can expose parts of the graph that was traversed earlier during the current traversal. This is a fundamental 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 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 are to find out the relationship between AspectJ and other tools. Examples of these experiments were presented above.

The focus of these experiments, integrating AspectJ and AP Library, is 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 only 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 previously. 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 has to be better ways of expressing what a user would want the computer to accomplish on behalf of the user. A quick survey of latest Aspect Oriented Programming Languages were taken. The languages surveyed were [DemeterJ], [DJ], FRED, and [AspectJ]. DemeterJ and DJ were developed at Northeastern University and was a natural selection as a graduate student there. FRED was developed by Doug Orleans as a part of his Ph.D. Dissertation and its close ties with AspectJ made it prime candidate. AspectJ was selected for its popularity among the AOP community. These four AOP Tools were determined to be the tools that we will be used in our experimentation and analysis.

FRED

FRED [Orleans02], developed by Doug Orleans, is a language that integrates Aspect Oriented Programming and Predicate Dispatching, implemented in [MzScheme]. The main components in FRED are Message, Branch, and Decision Point. In order to simplify these terminologies, we will map these components to components from functional programming. 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 just use this analogy for simplicity sake. 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 predicates.

In order to understand how this works, we will present a simple factorial example. In FRED, a programmer defines a Message by calling the function `define-msg`. In Figure 1, the first statement defines a Message called `fact`. Next, we define some Branches for this Message, `fact`. First, we will handle the base case where the first argument is equal to one. This is expressed in FRED in Figure 1 by the second MzScheme statement. We used 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 1. 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 the multiplication.

```
(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

There is some complex situation regarding the predicates for the two Branches. 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 interpreted, FRED invokes the first Branch instead of the second Branch. The rationalization for this decision is that first predicate is more specific than the second predicate. Thus, FRED always selects the more specific predicated Branch. For all other cases for the call to the function `fact`, the second Branch is invoked. This gives us the behavior that we want from FRED in implementing 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 an execution of a program. A Strategy is a direction on how to traverse the Object Graph. Strategy such as "from Company to Employee" and "from Top to Bottom via Middle" combined with a Class Graph yields a Traversal Graph. It is a sub-graph of the Class Graph that includes all possible paths defined by the given Strategy. The Visitor 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. This Demeter Process is shown in Figure 2.

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 Dictionary as well, since it is implemented as a Java Class. The Strategy and the definition of the Visitor are specified in Behavioral Files.

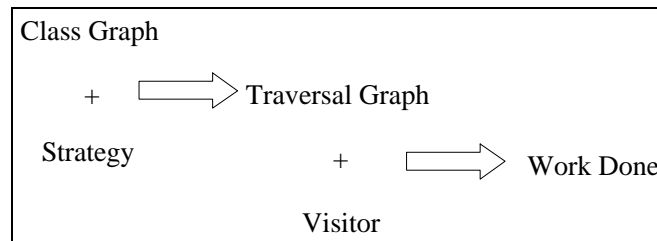


Figure 2: Demeter Process Overview

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 3. The Basket Class may contain three objects, one Pencil and two Fruit. A Fruit may also be an Orange. Every Fruit has Weight, which is an integer and an Orange has Color which is represented by a String. In this example, we will count the integer Weights within a Basket.

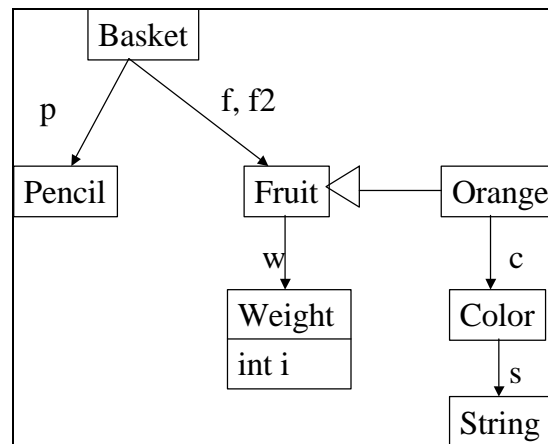


Figure 3: Class Graph of the Basket Example

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 4. The identifier on the left is the class being defined. The "=" can be interpreted as "has-a" in Object Oriented Programming terminology. The identifier within "<" and ">" are labels for these "has-a" edges within the Class Graph. The "is-a" relationship is defined by the ":" symbol. The "*common*" indicates that the Class Fruit has a "has-a" relationship 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 4: Class Dictionary of the Basket Example

Because of the way DemeterJ implements alternation class as abstract class, we need to change the way inheritance is expressed between Fruit and Orange. If want the Fruit Class to be concrete instead of abstract, we need to use "extends" instead of the Alternation with ":". The modified Class Dictionary is shown in Figure 5. The ":" for Fruit has changed to "=" and the string "extends Fruit" was added to the definition of Orange. This also signifies that Orange inherits from Fruit.

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

Figure 5: 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 count all of the Weight object within the Basket, we need to traverse from the Basket Class 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 6.

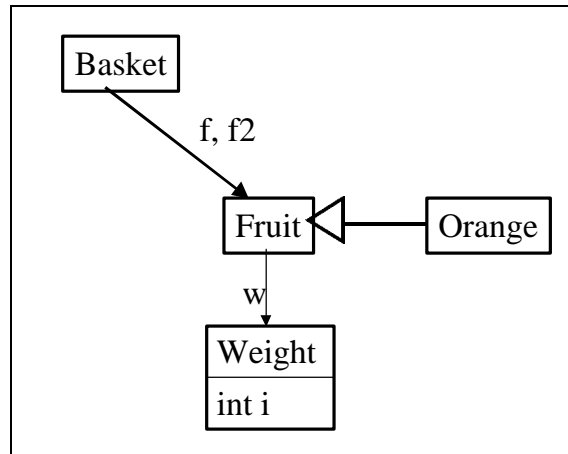


Figure 6: Traversal Graph as a result of applying the Strategy "from Basket to Weight" to the Basket Example Class Graph

Now that we have a Traversal Graph, we need define the traversal and Visitor to traverse the Class Graph and count the integers stored within each Weight object. We accomplish this by defining the traversal and the Visitor within the Behavioral file. However, we have not actually declared the Visitor in the Class Dictionary. We may accomplish this by adding the line shown in Figure 7 to the Class Dictionary.

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

Figure 7: Declaration of the CountWeightVisitor in the Class Dictionary

Now, were ready to define the traversal and the Visitor in the Behavioral 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 "code" 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 with parenthesis around it. 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, 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 8.

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

The definition of the CountWeightVisitor is similar. We start with the scoping 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 state() 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. This is used to signify pure Java code to the DemeterJ weaver.

The Advice specified within CountWeightvisitor is a Before Advice. It is executed before Visiting all the "children" of the node Weight. Within the Advice, there's a keyword "host", which is similar to "this" in Java. However, it refers to the object of type Weight we are currently visiting in this case. The accessor method get_i() in the Advice is generated automatically for us by DemeterJ. This Advice is executed every time we encounter an object of type Weight and add the integer within Weight to the 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 Visitor. 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 9 below.

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

    {{ int total = 0; }}

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

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

Figure 9: 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 7. 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.

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 4. 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 10.

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

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.

DJ

The next implementation of the Demeter Concepts is DJ. This is a set of Java libraries that allow programmers to specify Class Graph, Strategy, and Visitor through an API. This is useful for programmers that do not want to learn a new language and use an experimental tool. They may include the library in their program and obtain the full capabilities of these concepts.

As in DemeterJ, there has to be a way of specifying the Class Graph in DJ. While DemeterJ uses Class Dictionary to specify the Class Graph, DJ uses Java's Reflection to construct the Class Graph. DJ also provides an API to access this functionality. The class ClassGraph in the AP Library [APLib], allow 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 11: Definition of the Basket Example Class Graph in Figure 3

The actual specification of the Class Graph is pure Java code as shown in Figure 11. The class `ClassGraph` only allows the programmers to access the Class Graph and the relevant methods. Thus, the definition of the Class Graph is accomplished by the usual method of specifying classes and their definitions in Java. The DJ's `ClassGraph` class provides the reference to the Class Graph defined by the programmer. The Java code to specify the Basket Example Class Graph in Figure 3 is shown in Figure 11. It uses the usual method of data member variables for "has-a" relationships and uses the "extends" keyword for the "is-a" relationships. This Java code would have been generated from the Class Dictionary in DemeterJ.

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

Figure 12: 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 call the constructor to the class `ClassGraph` as shown in Figure 12. DJ uses Java Reflection to construct the graph representation

from the current running program. This allows programmers to adapt their existing applications to use DJ.

```
class CountWeightVisitor extends Visitor {
    int total;

    public void start() {
        total = 0;
    }

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

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

Figure 13: Definition of the CountWeightVisitor for DJ

Next, we define a Visitor to count the weights within the Basket as shown in Figure 13. The actual Java code is very similar to the definition for DemeterJ in Figure 8, except for minor syntactic differences. Thus, we can conclude that DemeterJ's syntax for the Behaviorals files are very close to the Java equivalents.

Lastly, we need to invoke the traversal with the Object Graph, Strategy and Visitor. This Object Graph is created by calling constructors for the classes in the Class Graph. 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 14.

```

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 14: 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, however they are powerful in their concept and application.

AspectJ

The last AOP Tool that we will be discussing is AspectJ. 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 to be organized in a different manor than in Java. Join Point allows programmers to specify a point within the execution of an application. Pointcut specifies a set of Join Points. Finally, the Advice is executed when the Pointcut for it holds true. This description of programs using Join Point is called a Join Point Model.

We will show these concepts in action with the Basket Example. As with other examples, we need to present how the data structures, 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 15: Defining Classes for the Basket Example

First, we define empty classes as shown in Figure 15. Then, we use the AspectJ Introductions to introduce the data members and the inheritance relationship as shown in Figure 16. 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 Aspects.

```
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 16: Defining Relationships for the Basket Example Class Graph

Next, we define constructor and accessor methods for the classes within the Basket Example. In Figure 17, 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 we need to type more, because we need to specify the class names for each method introduced.

```

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 17: Definition of Constructors and Methods for the Basket Example

Now, we specify the code to count the integer values within `Weight` objects in Figure 18. 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 count the `Weights`. We define the method `getTotal()` for `Basket`, `Fruit`, and `Weight`. These methods are used to access the `Weight` objects.

In order to add 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 data passing from 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.

This `Aspect` has the same semantics as the implicit `Visitor` in Figure 9. The methods `getTotal()` defined for `Basket` and `Fruit` are generated by `DemeterJ`. While in `DJ`, the call to `traverse()` takes care of this function.

```

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();
    }
}

```

Figure 18: CountWeight Aspect of Basket Example

Lastly, 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 19. 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 19: 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 20. 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.

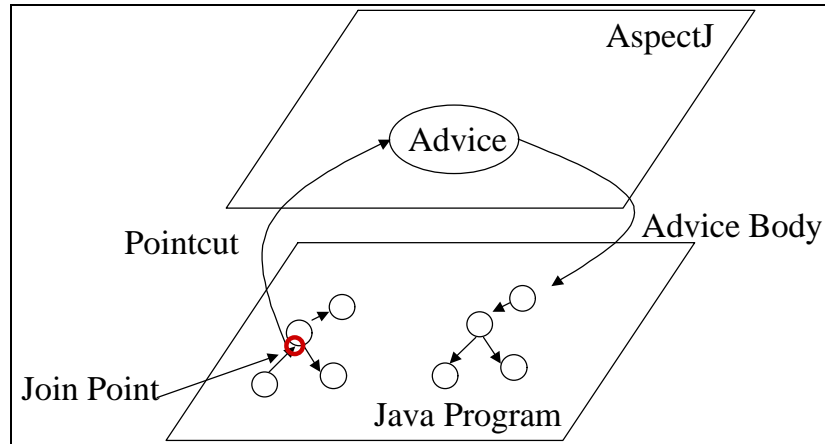


Figure 20: A Conceptual Model of AspectJ

We have introduced the AOP Tools FRED, DemeterJ, DJ and AspectJ. 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 the subsequent chapters, we will translate some examples of FRED to better understand the semantics 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. In order to find the performance of DAJ, DemeterJ and AspectJ+DJ, we present an implementation of a simple HTTP server for each these tools. From the discussions and experiments presented, we apply the Graph Theoretical View to develop an analysis methodology for Aspect Oriented Programming Tools with a

Four Graph Model of Programs. 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	Semantics
	Parameter
flat-cord? l	Branch Predicate
	Message, Function, or Basic Block
	Branch
	Return Value as Parameter
	Branch on a return trip from a branch
	Termination Branch

Figure 21: Primitives for Graphical Expression of FRED

In order to describe FRED, we have broken down what actually happens within each function in Figure 21. We have a circle with the name of the function within it. This is a node within our Graphical Representation. Next, we need a way to connect these nodes together and the

arrow signifies 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 signify 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 so a Termination Branch was added.

As an example, we have translated a simple factorial example in Figure 22 using the primitives that are listed in Figure 2. The Scheme expressions are shown on the left and the 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, the Branch goes to the function fact back to itself with `(- x 1)` as the parameter. The Return Branch is also specified in the last Scheme expression. The Return Branch goes to the plus function with the parameters x and the value returned from the original branch. This is how we translate FRED code to the Graphical Notation.

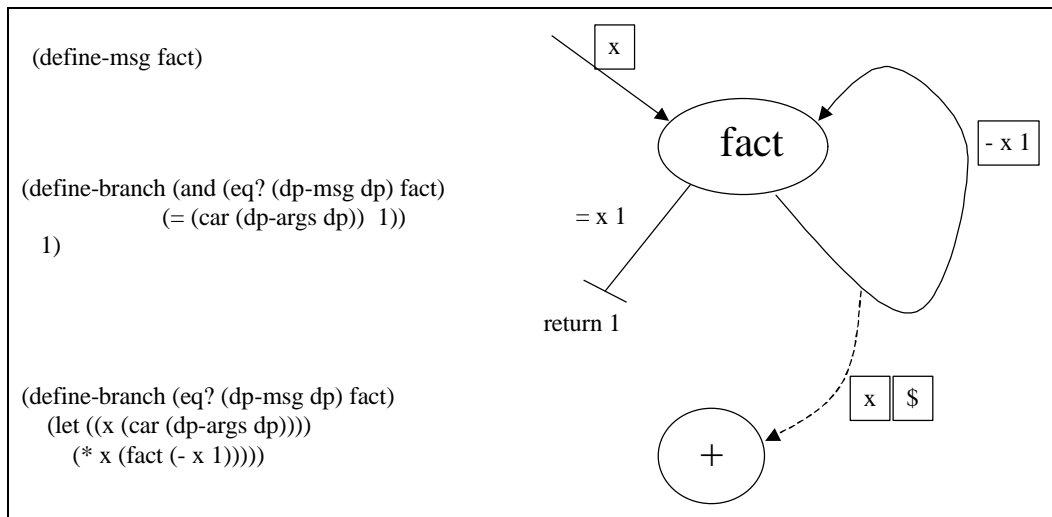


Figure 22: Graphical Translation of Factorial Example

From this translation process, we can make several observations. First, 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 programming languages such as C++. Thus, it is upto the programmer to organize these Branches and Messages in anyway that is optimal for a particular application.

The second observation to make is that the mechanism to describe the return value is quite clumsy in the Graphical Notation. It is not very intuitive and it is hard to understand what is happening. This is because 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 Scheme is Forward Centric, meaning that we describe explicitly what is happening during the one-way trip and the things that are happening during the return trip is implicit. Many of the programming languages are forward centric, i.e. explicit syntax for invocation of functions and implicit syntax for handling values that are returned. This makes it difficult to describe the events during the return trip.

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 manor, 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 fundamental assumptions evident in programming languages and we can explore further, the semantics of and the trade-offs made for programming languages.

We will take a look at a more complex example of the Graphical Notation for FRED to explore these observations further. 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 `n`th 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 these three functions.

First, we will present and analyze the function `concat` that concatenate two strings. In this example, the method `define-method` is used. This is collapsing the two methods `define-node` and `define-branch`. Therefore, it has three parameters: name of the method, predicate for the Branch, and a Scheme expression 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 3.

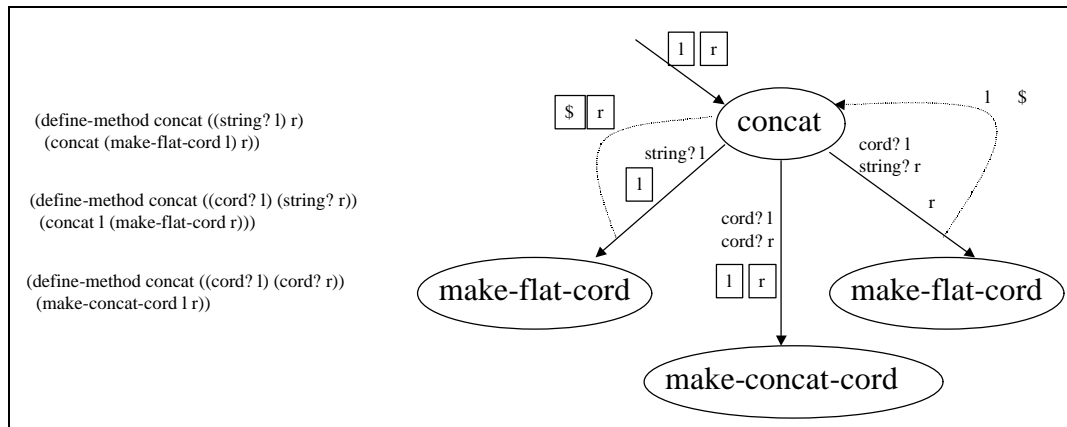


Figure 23: Graphical Notation for `concat`

In Figure 3, 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 the 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 and this is represented by the middle Branch in Figure 23. 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 the 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. 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 24 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.

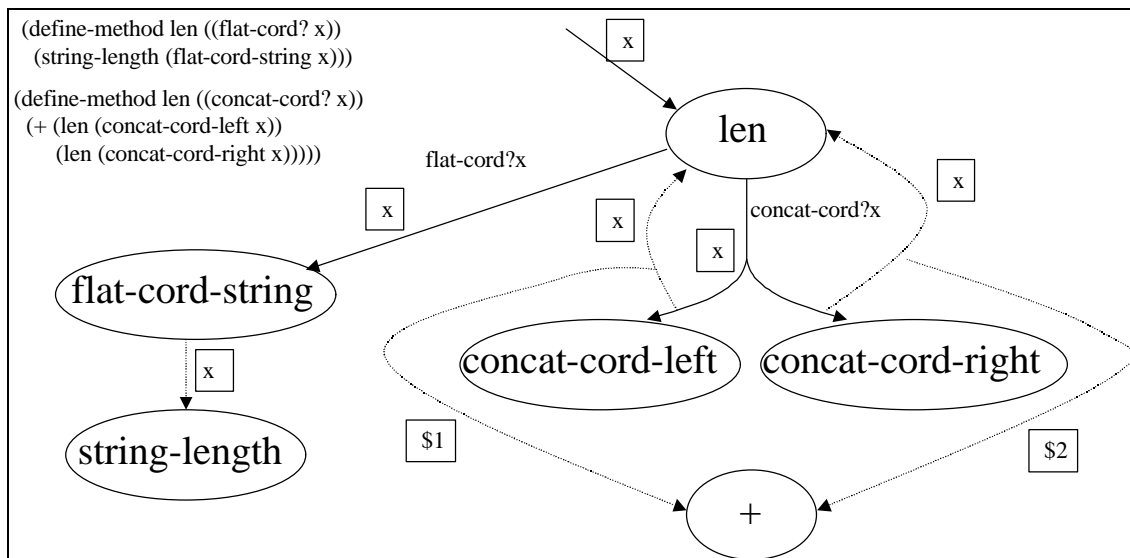


Figure 24: Graphical Notation for len

More complicated branch on the right is representing the second Scheme statement in Figure 4. 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 originally split merge at this point, since the values from the recursive call to `len` are needed for the call to `+`. The resultant value from this `+` function 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 i th 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.

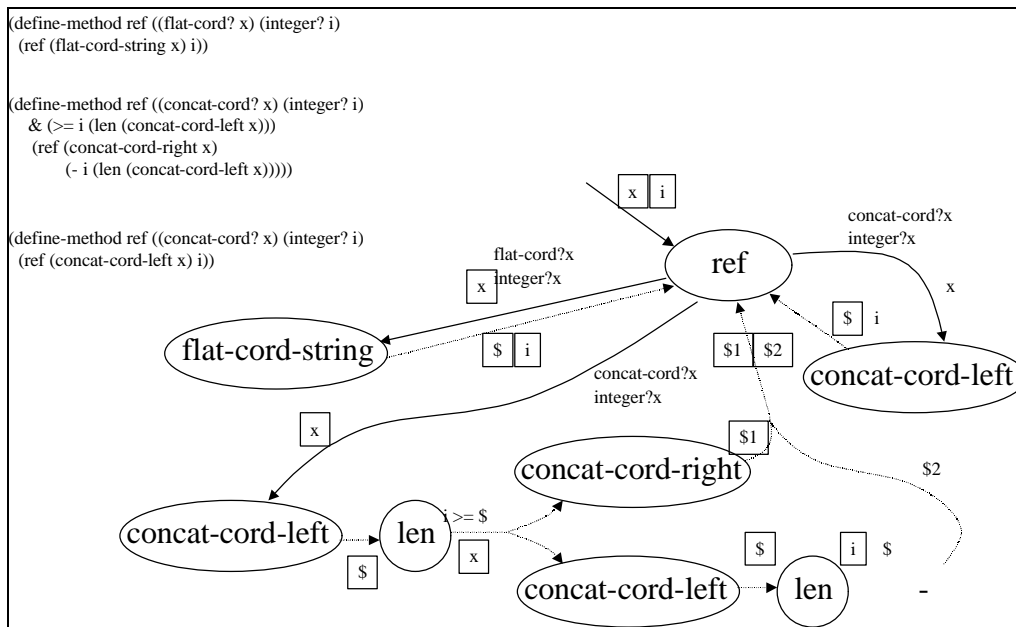


Figure 25: Graphical Notation for `ref`

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 25 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 " $i \geq S$ ", i.e. the integer argument is greater than or equal to the result from len, then call 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 is more evident. These observations have given us insights into some fundamental nature of programming languages. The grouping observation points to the scoping of different programming primitives. The awkwardness of Return Branches point to the fact that we tend to program thinking about making progress forward and not thinking too much about what happens when things return. 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. How some of the observations that we have made in translating FRED to a Graphical notation is manifested in this process? These are reasons for translating FRED to AspectJ.

First, we will attempt to translate the factorial example. The code in Figure 26 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.

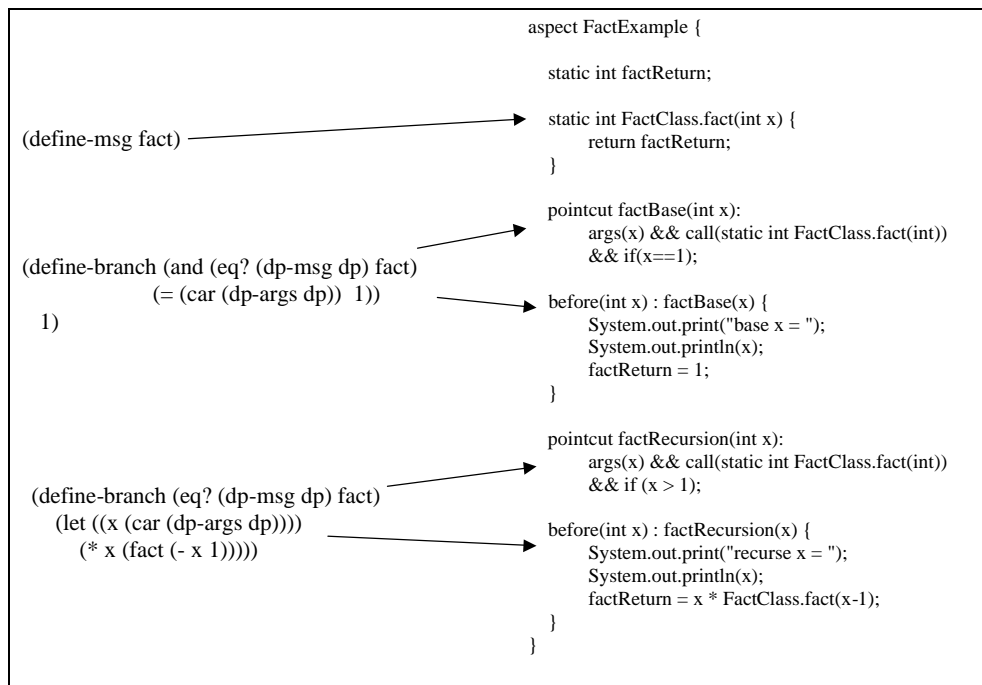


Figure 26: Factorial Example Translated from FRED to AspectJ

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 completes the execution. 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 compiler does not handle the case where an execution of an Advice is the result of a method call from an 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 27. The process is the same as the factorial example 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`.

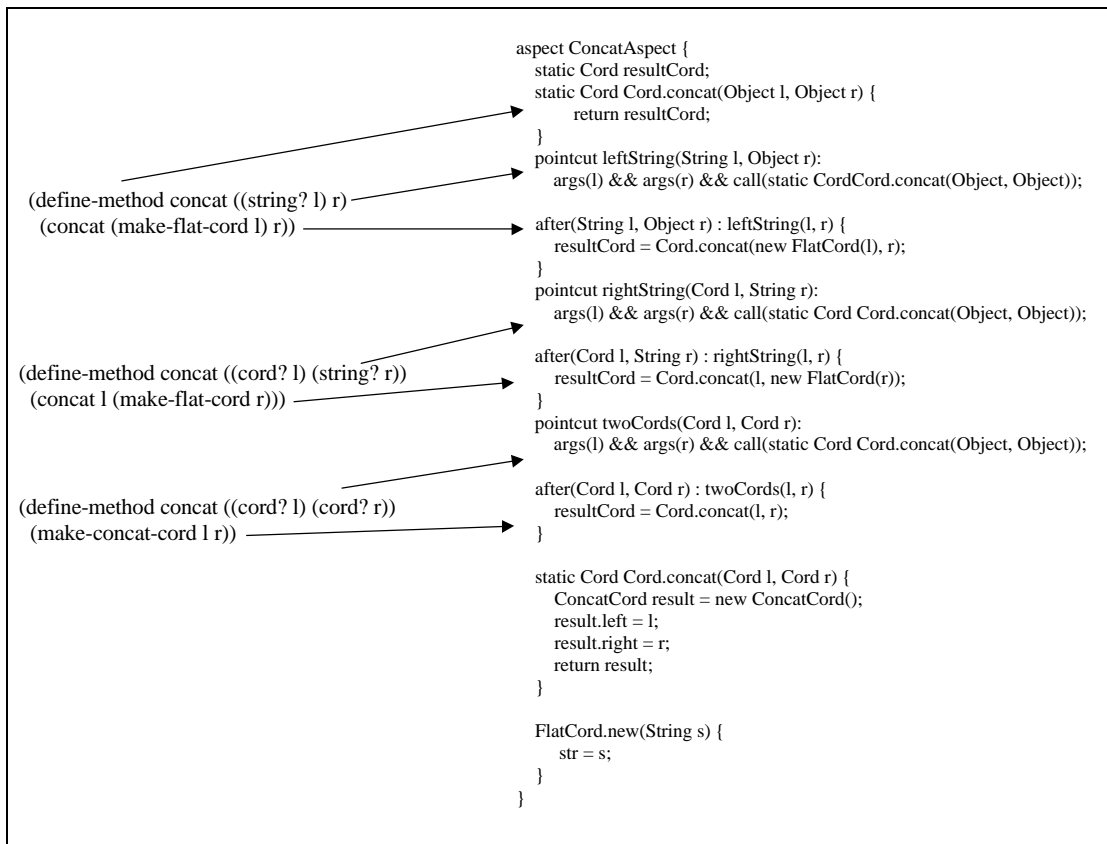


Figure 27: Translation of `concat` from FRED to AspectJ

These two translations of FRED to AspectJ have been surprisingly trivial. It shows that much of the observations that we made earlier may be applied to AspectJ as well. Notice the scoping of the Pointcuts and methods, the implicit or explicitness of the return values and the schema relationship. Thus, these three observations will be central for this thesis.

TRANSLATIONS 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, Join Point Model 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, JPM is method centric, i.e. the join points must arise out of method calls. Thus, the main concern of a JPM is the Call Graph within a program.

In contrast, "Program as a Journey" metaphor 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, he specifies the strategy in which a visitor should follow during the traversal of the Class Graph. The actual description of what to do is specified within a Visitor. 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.

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, 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 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. The Traversal Graph that we will translate is shown in Figure 6. In this translation process, we will use Introductions to

introduce method calls that will traverse the Traversal Graph correctly. 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 other way to distinguish the label via other types of Join Points. We also need to take of the inheritance relationship between Orange and Fruit by calling `super.t1()` in `Orange.t1()`. This will correctly handle the case where the Fruit reference is referencing an Orange object. The implementation of this traversal and the Pointcuts that describe the relevant Join Points are shown in Figure 28.

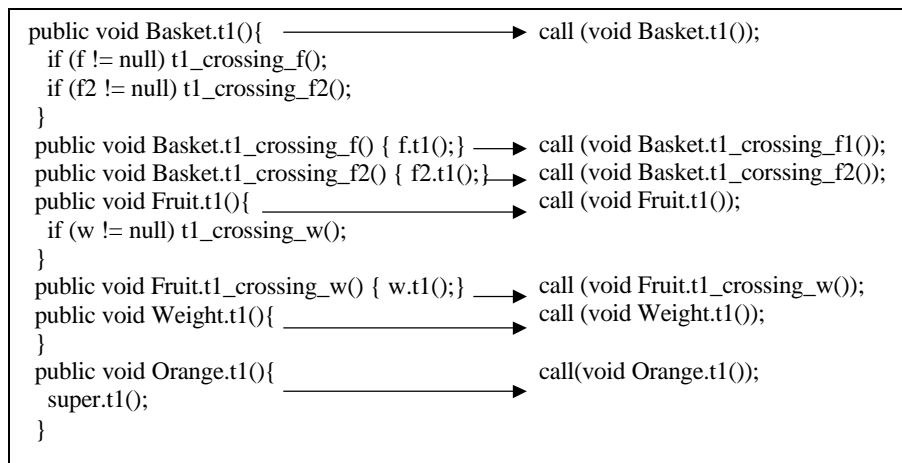


Figure 28: Implementation of the Basket Example Traversal in AspectJ and the Pointcuts for the Relevant Join Points within the Traversal

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 29 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` 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 29: 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 straight forward process. Therefore, we should be able to integrate Demeter concepts with AspectJ, which would be very useful. This is exactly what we did in DAJ.

We verified that we may implement Demeter concepts using AspectJ, but can we implement AspectJ concepts, i.e. using DemeterJ? We believe that this can be done. All we have to do is to find the correct conceptual mapping of the Demeter concepts Class Graph, Strategy and Visitor. First, we need some kind of graph that we need to traverse. In AspectJ, the join points are mostly based on the Static and Dynamic Call Graphs. Thus, it would be natural for us to treat the Call Graphs as the Class 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(). The Visitor then just has to have Advice to execute when it reaches a certain node, i.e. a method.

In this manner, we may translate AspectJ code in terms of Demeter. 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 Advice. The OS reads the program and start to execute the program. During the execution, the CPU/OS creates and destroys 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.

From this, 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.

```
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 30: AspectJ Basket Example Translated to Modified DemeterJ

In Figure 30, we attempt to translate AspectJ Basket Example to a modified DemeterJ. First, we create the Call Graph by using the modified BNF. This just creates the connections, i.e. method calls between these different methods. 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 should

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 last statement is the case where any method of the name getTotal() should have predicated method invocation if its date member is not equal to null. As you can see, it has some semblance to the AspectJ syntax. Lastly, we define the traversal with the Visitor, Guide, and a Strategy.

This modified DemeterJ has predicated execution by using the Guide, which is consistent with the metaphor that Demeter is using. This incorporates the predicate execution from FRED and the Pointcut syntax from AspectJ. Thus, merging of these tools into the modified DemeterJ would be a new direction in 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 issues from Human-Computer Interaction and Software Engineering.

JOIN POINT MODEL VS. TRAVERSAL MODEL OF PROGRAMMING

In this chapter, we will discuss and analyze the metaphors, Join Point Model and "Program as a Journey" being used in AspectJ and Demeter 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.

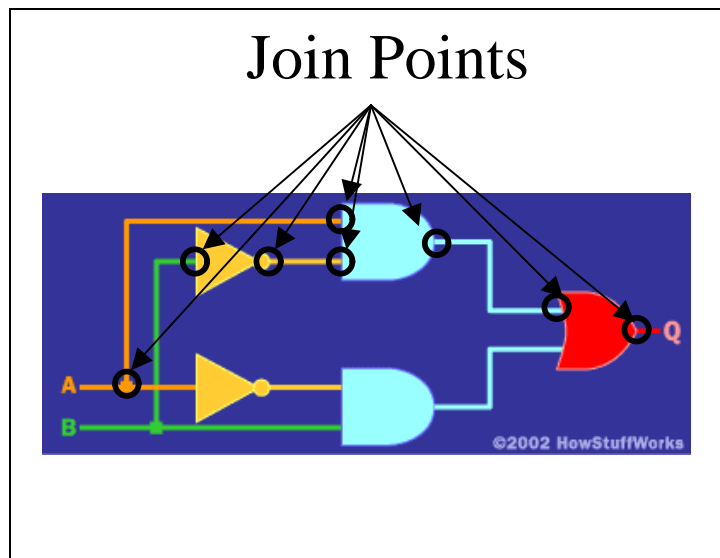


Figure 31: Schematic Diagram of an XOR from [20 HowStuffWorks]

We would call the Join Point Model a "building" metaphor. Many things in science and engineering use this type of metaphor in which things are built by their building blocks. Things such as furniture, molecules, and buildings have points in which building blocks join. 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 31. The Join Points for a caffeine molecule is shown in Figure 32.

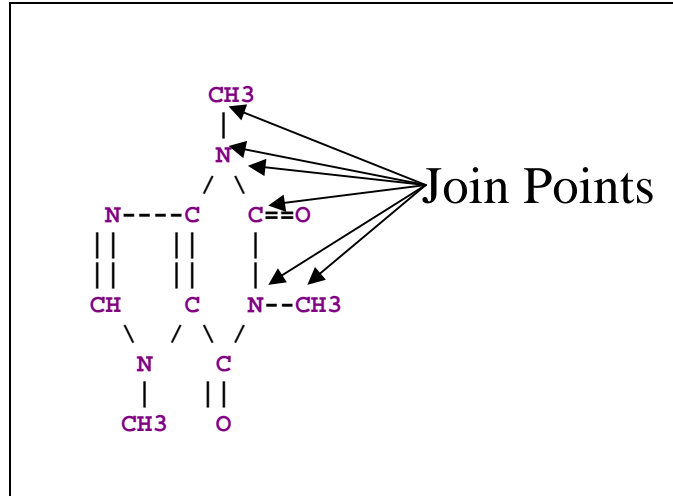


Figure 32: Join Points for the Molecular Model of Caffeine from [21 KoffeeKorner]

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 33.

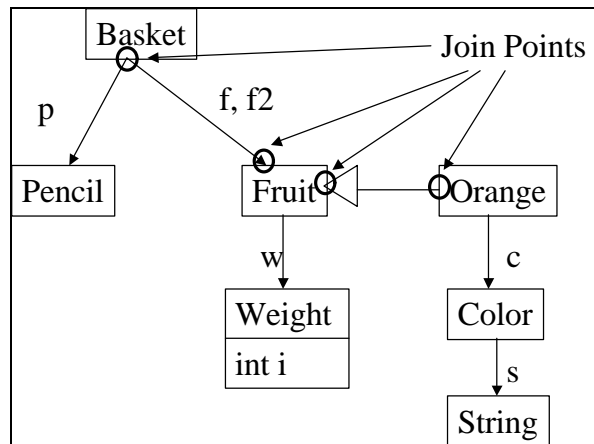


Figure 33: Join Points for the Basket Example Class Graph

However, we need to add something to the metaphor in order to add Advices. We need to add traversals. In AspectJ, the Join Points are points within the execution of a program. In this model, something is already happening, so adding Advices makes sense. It does not make

sense for the Class Graph, because there is nothing happening in the graph itself, until we add the traversal 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. 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, because all of the computer related fields and most of the sciences use this type of "building" metaphor.

The "Program as a Journey" metaphor used in Demeter is an instance of the Journey metaphors. 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 think of everyday things as a Journey.

In Demeter, we may map each of the major components 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 actual world or place in which he is traveling. The Strategy is the directions that the person is following in order to get to the destination. The Visitor is the person on the journey. The Advice in the Visitor is the actions the Visitor will perform at each location. The Guides, as introduced in "Translating Between AspectJ and Demeter," is the local guide that will help make localized decisions during the journey. Because of this perfect mapping between Demeter and a Journey, Demeter is an example of the Journey metaphor at work.

At first glance, the possibility that the "Program as a Journey" metaphor could be used to describe the Join Point Model does not seem to be high. 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 31, 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 networks of gates. Thus, we may apply the JPM and the Journey metaphors to the XOR schematic diagram in Figure 31.

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 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, because it is used everyday 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 Johns calls 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.

Even with these strengths for the metaphors used in AspectJ and Demeter, the ability to gain feed back about what the program is doing 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 were no effort in including in visibility features into the design. 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 Mode and "Program as a Journey" can be used to describe each other. These 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 included in the metaphor and the programming language for the AOP Tool to be successful.

DAJ: DEMETER INTEGRATED WITH ASPECTJ

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 "Implementing Traversals in AspectJ." DAJ integrates DJ and AspectJ to implement a system that allows programmers to specify traversals for their AspectJ programs.

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 design decision that we made. We also wanted the new language to have similar syntax as AspectJ to increase usability for programmers. Therefore, we decided to create a tool that would use AspectJ and DJ to generate the traversals in AspectJ and use AspectJ's compiler as the backend weaver.

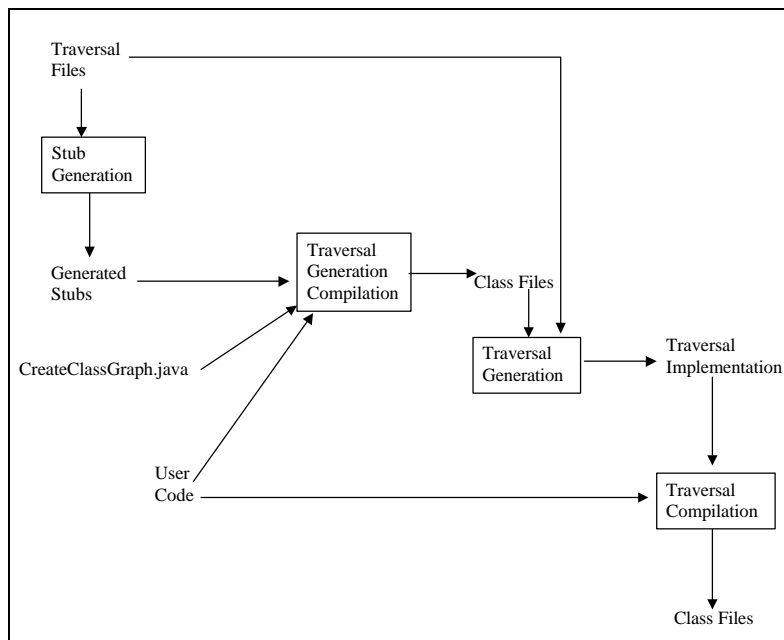


Figure 34: Four Compilation Phases of DAJ

There are four phases in DAJ as shown in Figure 34: stub generation, traversal generation compilation, traversal generation, and traversal/user code 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 are not generated yet, will cause a compilation error. Thus, for every traversal, DAJ will generate a stub traversal method for the source of the Strategy.

In the following phase, the stubs, CreateClassGraph.java and the user code is compiled. The CreateClassGraph.java 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. The AspectJ Compiler is used to generate the .class files to execute in the next phase.

In the traversal generation phase, DAJ executes the code that has been compiled in the previous phase. The CreateClassGraph.java includes AspectJ code to intercept the call to the main method. It then generates a ClassGraph object using DJ and starts the traversal generation process. DAJ parses the traversal files and generates the AspectJ traversal implementations as outlined previously.

Lastly, 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.

Traversal Specification Following DJ and AspectJ Syntax

The traversal file has three major components, the ClassGraph declarations, traversal declarations and aspect definition that enclose the ClassGraph and traversal declarations. The ClassGraph declaration may have two different types of Class Graphs: a Default or a Class Graph Slice. The default declaration is the ClassGraph object that is obtained from CreateClassGraph.java using DJ.

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

Figure 35: Default Class Graph Declaration Syntax and Examples

As shown in Figure 35, the default Class Graph declarations has 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. This similarity allows 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 36. 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, and a call to the constructor for the class ClassGraph. The constructor has ClassGraph variable and a Strategy in string format as the two arguments.

```
ClassGraph variable = 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 36: 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 37. Its syntax with the "declare" and "traversal" keywords are consistent with the AspectJ declare statements. These keywords are followed by a variable identifier, colon, Strategy in string form, and a semi-colon. Again, DAJ uses the default Class Graph from DJ to generate the traversal implementation.

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

Figure 37: 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 that there is a

Class Graph identifier within parenthesis between the traversal variable identifier and the colon as shown in Figure 38.

```
declare traversal variable(CG_var): "strategy";  
declare traversal t2(CG2): "from Here to There via Points";  
declare traversal myTrav(default): "from A via X to B";
```

Figure 38: Syntax and Examples of Traversal Declaration with Class Graph Argument

The last language feature is the aspect declaration. It contains ClassGraph declarations and Traversal Declarations as shown in Figure 39. The syntax is designed to be similar to AspectJ. However, programmers may not add AspectJ code with in the traversal files.

```
aspect MyTraversal {  
    ClassGraph defaultCG;  
    ClassGraph cg1 = new ClassGraph(defaultCG,  
        "from * bypassing {java.lang.String} to *");  
    declare traversal t1: "from CompoundFile to SimpleFile";  
    declare traversal t2(CG1): "from CompoundFile to *";  
}
```

Figure 39: 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 40 will generate to traversal code in Figure 28. Thus, we may replace the traversal code in Figure 28 with the DAJ traversal file in Figure 40 below. The other code may stay the same.

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

Figure 40: Traversal File for the Basket Example

DAJ is a tool that allows AspectJ programmers to use Demeter concepts in their programs. It will generate traversals from Traversal Files. It allows smooth integration of AspectJ and Demeter concepts.

A SIMPLE HTTP SERVER IMPLEMENTATION

Introduction, why this choice.

AspectJ + DJ implementation

DemeterJ implementation

DAJ implementation

Performance of traversal measure

Analysis of the 3 different methods & performance of programs

Conclusion

ASPECT ORIENTED PROGRAMMING TOOLS ANALYSIS USING 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 [1], Alderich attempts to provide different types of concerns and illustrate the types of problems that AOP Tools must solve. While in [2], Chu-Carroll attempts to organize the program in a novel program organization. Very abstract view of program organization is shown in [3] by Black and Jones. Sutton and Rouvellou take a novel approach and attempts to apply Categorizational Theory in [4]. All of these approaches attempt to come up with a method for thinking about concerns and how to deal with them.

In this paper, we introduce a Four Graph Model of Programs (4GMP) by applying the Graph-Theoretical View. The four graphs in 4GMP can be analyzed to formulate fundamental relationships about the graphs within 4GMP and the different features within an AOP Tool.

By using the model and the mentioned relationships, we can compare different features for overlap, equivalence and orthogonality. We can use these types of comparisons for relative measure of feature utility. In addition, they are necessary for feature inclusion decisions for AOP Tools.

The application of the 4GMP on DemeterJ [9,10] and AspectJ [7, 8], will illustrate the usefulness of 4GMP in AOP Tools analysis. DemeterJ is a tool that allows users to abstract programs into graph traversals and are early adaptors of AOP. AspectJ is a tool that extends the Java language to allow programs to specify aspects and has large support within the AOP community.

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.

Graphic Decomposition of Programs

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. 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.

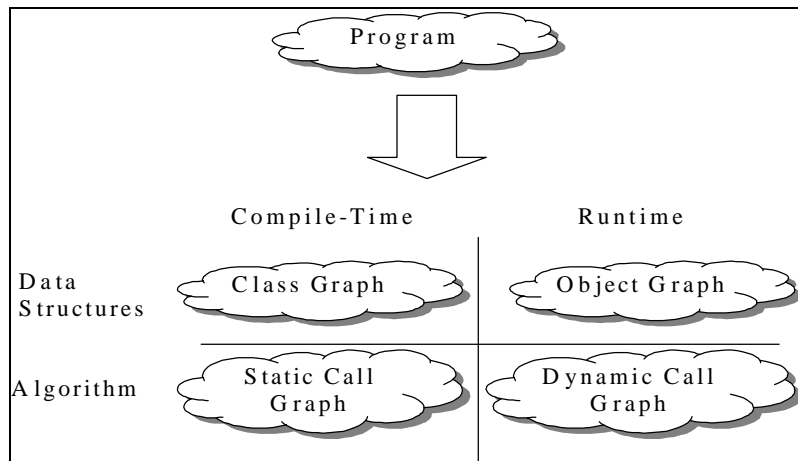


Figure 41: Graphic Decomposition of Programs

To understand how the different graphs interact, you just need 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.

You can make similar relationships between Static Call Graph and the Dynamic Call Graph. The Static Call 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 one view of Class Graphs and Object Graphs 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, the input, specification, and output are graphs. Processor is the processing logic that processes the specification and the input to generate the output. Processor is recursively defined.

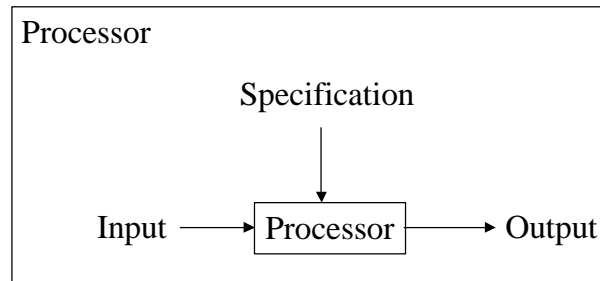


Figure 42: Graph Transformation Model (GTM) in 4GMP

This model can be applied to the current compiler and execution process. The figure below illustrates how specification and input combine to generate output. 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 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. The Computer System generates the Dynamic Call Graphs and Object Graphs using the program as the specification and the user's input as the input into the computer system.

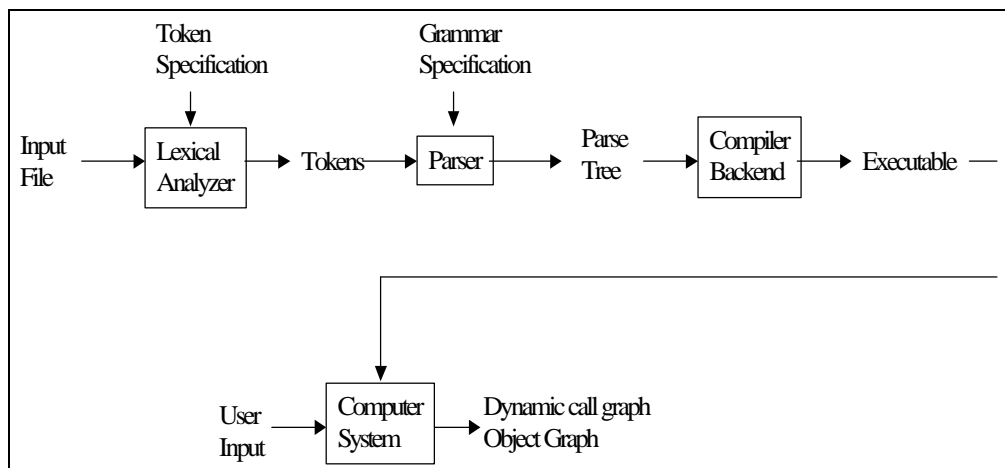


Figure 43: Application of GTM for Compiled Programs

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 cost of the execution of the implementation of the Algorithm or vice versa. This implies that you can some how reduce one graph by increasing the other. There is still very little understanding of the relation between these two within the 4GMP and further investigation is needed.

Organizational Concerns

When a graph gets to be very large, it get to be very hard to find things to modify or understand it. Sorting or organizing is a way to solve this problem. We can organize the graph 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 the 4GMP.

The Organizational Concerns (OC) are important in that it allows us to create some order, which is necessary for us to divide the program into manageable chunks that we can create and manipulate. 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?

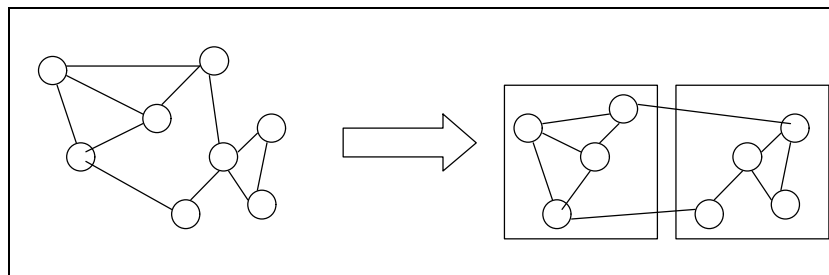


Figure 44: Organizational Concerns in 4GMP

For this problem, we can borrow the concept of cohesion and coupling from software engineering. 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 some collaboration, whether the collaboration is a data centric view or algorithm centric view

of the collaboration. This allows use to create some metric to measure the effectiveness of the graph subdivision.

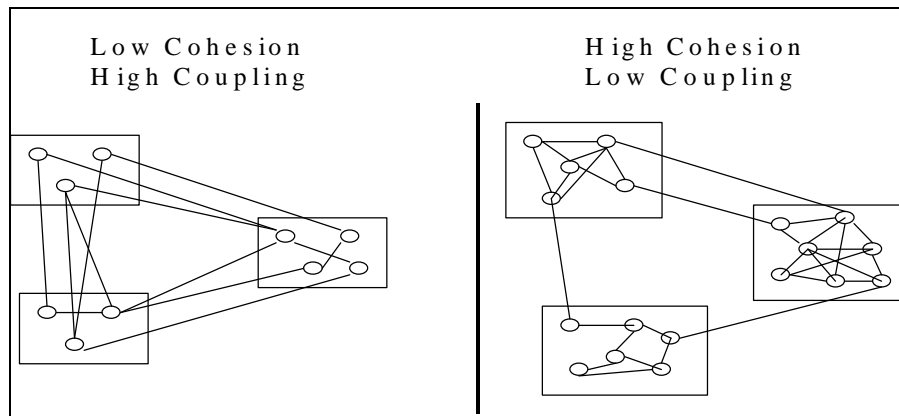


Figure 45 Cohesion and Coupling from [6]

The figure from [6] shown above, illustrates the desirable High Cohesion and Low Coupling verses the undesirable Low Cohesion and High Coupling cases for Organizational Concerns.

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 graph. We call the concerns of this factorization process the Factorizational Concerns. These factorizations occur at many different levels.

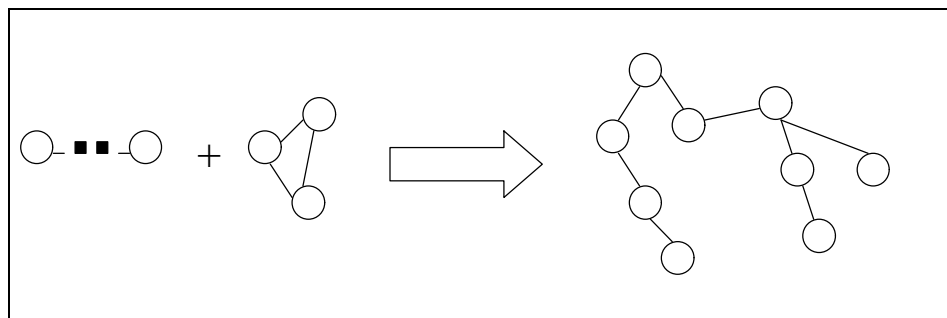


Figure 46: Factorizational Concerns

This picture should look very familiar. It is the Graph Transformation Model of 4GMP. 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.

From this, we can conclude that there is a Factorizational Concern relationship between the Class Graph and the Object Graph and between the Static Call Graph and the Dynamic Call Graph. From this we can create a Concern Relationship (CR) Diagram of the four graphs in 4GMP. The CR Diagram of the four graphs in 4GMP is shown below.

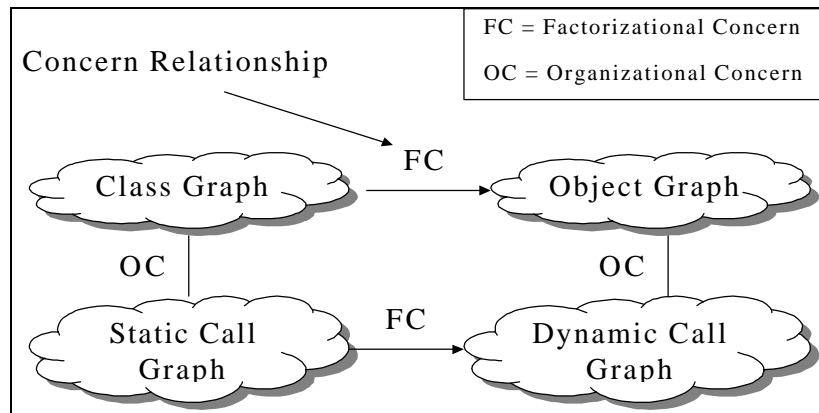


Figure 47: Concern Relationship Diagram of 4GMP

From the CR diagram above, the Factorizational Concerns and Organizational Concerns seems to be orthogonal, i.e. they do not interfere with each other. As an example, let's take a look at the Object Oriented features, objects and inheritance. Objects are grouping of data and the associated operations on the data. Thus, this is an Organizational Concern Feature, a programming feature that assists programmers in organizing the program. The inheritance feature allows you to factor out common data and operations to the base class. Thus, this is a Factorizational Concern Feature, a programming feature that allows programmers to factor out commonalities within a program. Therefore, in Object Oriented Programming, the objects have a OC relationship, and the inheritance hierarchy has a FC relationship with the program.

Evaluation of Features

We now know a little more about the relationships, OC and FC, however do still do not know how to evaluate the effectiveness of these features. First, we will attempt to evaluate the effectiveness of OC Features and FC Features in general. Then, we will discuss the evaluation of features that addresses the same concern.

The Organizational Concern Features allows us to deal with graphs that are simply too large for us to deal with. These features allow programmers to apply the current programming system to larger software systems, modules did it in the 80's and objects did it in the 90's.

Even though the OC Features allow programmers to create large software systems, it does not reduce the amount of code that the programmer has to write. Matter of fact, it actually increases it. The programmer has to add extra information about how the program is grouped.

The Factorial Concern Features allows programmers to factor out the regularities within a program and reduces the amount of code the programmers has to write. From this, we can conclude that FC Features allow programmers to reduce the amount of work in either programming or in maintaining the program.

From this analysis, the orthogonality of OC and FC Features become intuitive. The FC Features attempt to reduce the amount of code being written, while OC Features deal with the eventuality of the software system becoming large. Theoretically, if FC features were able to factor out enough code such that the amount of code does not grow too large, then we don't need OC Features. Even though this type of system is very hard to achieve, we can say that FC Feature are more cost effective than OC Features in general for this reason.

Then, how do we quantify the effectiveness of features that have the same Concern Relationship? If you analyze the different FC Features, inheritance hierarchy, functions, macros, etc., you can see that the difference between these features is the scope in which these features are applicable. You can factor out similar code with functions, while you can factor out similar functions and similar data members with inheritance hierarchy. We can loosely conclude that larger the scope in which a feature can be applied more powerful the feature becomes.

The largest scope a FC Feature can have is the scope of all valid instance of the language in which the feature is being applied. One instance of this large scope is the FC Relationship between C and Assembly. C allows programmers to express their programs in a much more

compact form than they can in Assembly. Thus, different statements in C factor out the repeated patterns that are evident in most programs written in Assembly.

AspectJ Analysis

AspectJ is an Aspect Oriented Programming (AOP) Tool that was introduced by the AspectJ group at Xerox PARC [7, 8]. AspectJ extends the java language to allow programmers to group crosscutting concerns, i.e. Aspects. We will limit our analysis to Introductions, Join Points, Pointcuts and Advice 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. There are other means of adding methods, data members and inheritance relationships in Java, but this method allows programmers to group crosscutting concerns in one aspect. From this description, 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 neither OC Feature nor FC Feature, because by itself, it does not allow for organization or factorization of the program. We call this type of feature a Specification Feature.

Pointcuts are a set of join points. It allows programmers to specify the join points in a concise manor. This is a Factorizational Concern since it allows programmers to specify the set of join points with patterns which are much more compact. Therefore, Pointcuts allow programmers to factor out commonalities within a set of Join Points.

When the condition specified by a Pointcut holds true, we need to specify the code that needs to be executed. We specify the code within an Advice. The Advice is usually executed before, after or around (instead of) the Join Point. This mechanism allows the programmer to modify the behavior of the code by addition instead of modification (i.e. incremental programming).

The concern relationship for Advice is complex. It uses a Pointcut to determine its point of execution and it has a Specification Concern relationship with Pointcuts. However, it allows programmers to factor out the method body from all of the Join Points specified by the Pointcut. Thus, it has a FC relationship with the Static Call Graph.

This relationship is different from the ones mentioned so far, because this relationship is implied through the FC relationship between the Pointcut and the Join Point. From this observation, we can conclude that there are some rules about how one may make relations between features or graphs that are not directly related. This is discussed more in-depth later in this section.

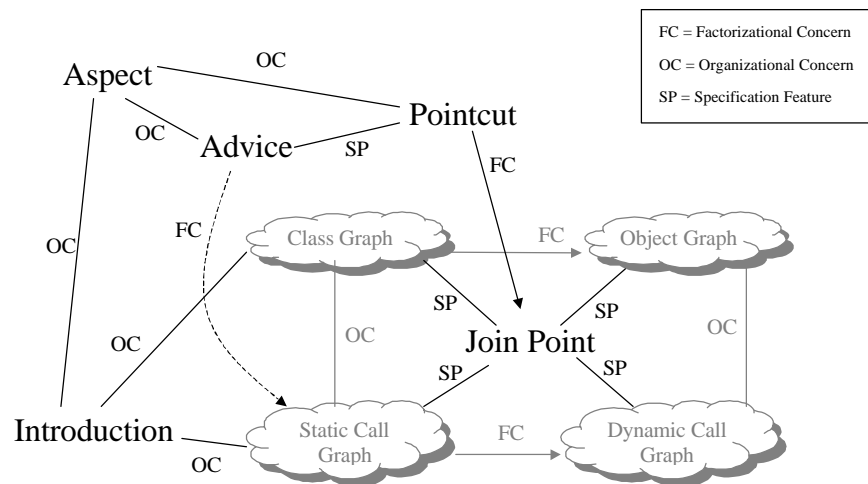


Figure 48: AspectJ Concern Relationship Diagram

Finally, AspectJ Aspects allow programmers to group Introductions, Pointcuts and Advices. Thus, this is an Organizational Concern in relation to these features. The fact that AspectJ has a different syntactic container for these AspectJ features separate from the Java class syntactic container, one can divide all of AspectJ program into either purely Java or AspectJ Introductions or AspectJ's Pointcuts and Advices.

As mentioned above, we can create some rules about how relationships between elements within the CR Diagram that are not directly related. We observed that Advice has a FC relationship with the Static Call Graph through the FC relationship between Pointcut and Join Point. Let us analyze this a little further.

The interesting part about the CR Diagram of AspectJ in Figure 8 is that Join Point has SP Relationship between all four of the graphs in 4GMP. So, depending on which Join Points you are specifying within in a Pointcut, Pointcut can have FC Relationship with the different graphs.

First, consider the relationship between the Class Graph and Object Graph with the Join Point. If a Join Point specifies a Join Point in the Class Graph, that Join Point has a FC Relationship with the Object Graph. This can be implied by the FC Relationship between the Class Graph and the Object Graph.

What happens if Join Point specifies Join Points in the Object Graph? Well, there isn't any implications that we can conclude. This is because the OC Relationship between Object Graph and Dynamic Call Graph does not give us any insights. However, we can attempt to convert this Join Point from specifying a Join Point in the Object Graph to specifying a Join Point in the Class Graph. This processes factors out Join Points in the Object Graph into the Class Graph.

Similar relationships can be extracted from the SP Relationships between Join Point, Static Call Graph and Dynamic Call Graph. The difference is that the Advice itself is a method and thus it allows for a FC Relationship between Advice and Static Call Graph. Same does not hold true for the Class Graph and Object Graph since Advice does not allow to specify nodes in these graphs. However, adding such a feature in the future could be a possibility.

From the CR Diagram we can see that there many more features that addresses OC instead of FC. We can call this type tool that address mostly the Organizational Concerns as OC Centric Tool.

DemeterJ Analysis

DemeterJ is a tool introduced by the research group at Northeastern University [9, 10]. In contrast to AspectJ, it creates a new software development process that is built on top of Java. We will analyze Class Dictionary, Strategy Graph, Traversal Graph, Visitor and Advice 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 is-a relationships corresponds to actual 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 [11].

These characteristics of Class Dictionary make it a Factorizational Concern Feature. It factors out the syntax needed for specifying the Class Graph. This feature of Class Dictionary is a FC Relationship between Class Dictionary and Class Graph. This is assuming that Class Graph is specified using Java. The generated parser code is just nodes and edges in the Static Call Graph. Thus, there is 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. An example of a 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, thus a Strategy Graph is the result of factorization of the Traversal Graph. This is a FC Feature and has a FC Relationship with the 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. From this, we can conclude that Class Graph has a FC Relationship with the Traversal Graph.

But how are Strategy Graph and Class Dictionary related? They do not impose any constraints on each other at all. Matter of fact, nodes in the Strategy Graph and the Class Graph may be disjoint and the resultant Traversal Graph can be null. Thus, we can say that they are orthogonal, but we can not say if they have a any particular type of relationship in the 4GMP.

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, it as Advice, similar to the Advice in AspectJ, on a particular object type. In the body of the Advice the Visitor does some work. So,

Visitor has a collection of Advice and this implies that there is an OC Relationship between Visitor and Advice.

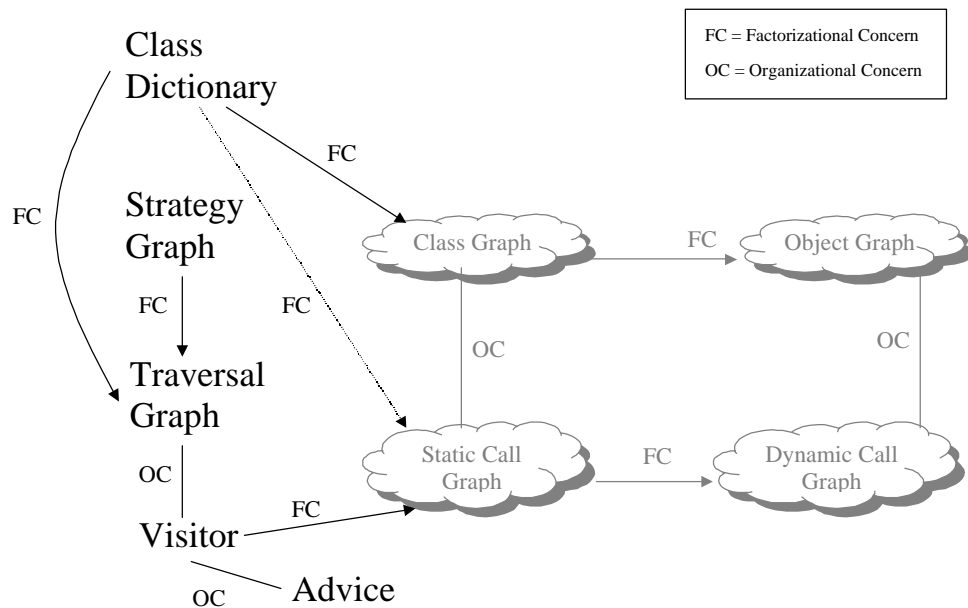


Figure 49: CR Diagram of DemeterJ

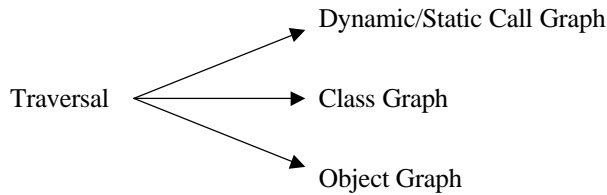
From the CR Diagram of DemeterJ, we can see that DemeterJ has more FC Relationships than OC Relationships. Thus, we can say that DemeterJ is FC Centric Tool. Its features allow the programmer to factor out different parts of a program into smaller components. This is contrary to AspectJ, an OC centric tool.

This leads us to believe that integration of features within AspectJ and DemeterJ would be synergetic. OC Features are orthogonal to FC Features since their concerns in which they are addressing are different. Thus, integrating DemeterJ and AspectJ would make sense.

Mapping of Traversal Concept

How the model predicts that the concept of traversals can be factored out from the 4GMP.

mapping of traversal concept to dynamic call graph, class graph, and object graphs.



Conclusion and Future Work

In this paper we have used the Graph-Theoretical View to develop the Four Graph Model of Programs (4GMP). From this model, we were able to find two different relationships,

Organizational Concern Relationship and Factorizational Concern Relationship.

Organizational Concern Relationship is how different features or abstract ideas that are addressing concerns of organizational nature. It attempts to sort the program in a manor that allows programmers to handle large software systems. Programming tools that have many OC Relationships are called OC Centric and AspectJ is an example of an OC Centric tool.

Factorizational Concern Relationship allows programmers to address the issues involved in factoring out commonalities within a program. Factorizational Concern Features reduces the amount of code and increase productivity. When FC Relationships dominate within a CR Diagram of a tool, then we call it FC Centric. One such tool is DemeterJ.

We have also analyzed some features of AspectJ and DemeterJ and created Concern Relationship diagrams to understand how these features related to each other. This gave us a better understanding of how the tools worked and to identify the main concern in which the tool was attempting to fix.

By using the 4GMP and the relationships that comes from it, we were able to identify the purpose of those features and the types of problems that those features were concerned with. This methodology of comparing different tools allow tools developers and programmers to understand what features are useful for what purpose.

However, there is still much work that needs to be done with the model and its uses. We need to better understand not only the Aspect Oriented Programming Tools, but the tools that are utilized by these AOP Tools, such as Java and Byte Code. This understanding is crucial for making the correct feature inclusion decisions.

The relationship between software engineering and the 4GMP needs further study. How different software development processes and productivity are related to the different features within a tool. There needs to be better understand of the applicability scope of the different features needs to be studied and quantified.

This type of analysis also needs to be applied to other emerging AOP Tools as well to understand the types of problems that they are attempting to solve. It is important for the developers of AOP Tools, but important also for adaptors of AOP Tools. It allows the users to understand how to use the tools effectively.

Other ramification of the 4GMP is the existence of one language that can specify the four graphs in 4GMP and operations on those graphs. This implies that another level of global factorization exists that will allow programmers to specify the four graphs in a more concise manor. For example, one can create on syntax for describing the traversals within the four graphs of 4GMP that may be interpreted within the context of each graph.

CONCLUSIONS

Goal of this thesis, to come up with a way of analyzing AOP Tools.

Translation of FRED, traversals, aspectj to demeter traversals, brought out the semantics of these different approaches. Analyzed the metaphors used in AspectJ's JPM and Demeter's "Program as a Journey" metaphors.

Using these experiments and analysis, we presented 4GMP and it's role in analyzing AOP Languages. Benefits of this type of analysis for future development.

BIBLIOGRAPHY

- [1 Orleans02] "Incremental Programming with Extensible Decisions," Doug Orleans, AOSD, 2002
- [2 Hugunin2001] "The Next Steps for Aspect Oriented Languages," Jim Hugunin, Workshop on New Visions for Software Design and Productivity: Research and Applications, 2001, <http://www.isis.vanderbilt.edu/sdp/Papers/Papers.htm>
- [3 Aldrich] "Challenge Problems for Separation of Concerns," Jonathan Aldrich
- [4 Chu-Carroll] "Separation of Concerns: An Organizational Approach," Mark C. Chu-Carroll
- [5 Black et. al.] "Perspectives on Software," Andrew P. Black and Mark P. Jones
- [6 Sutton et. al.] "Applicability of Categorization Theory to Multidimensional Separation of Concerns," Stanly M. Sutton Jr and Isabelle Rouvellou
- [7 Chaves et. al.] "Design-level Support for Aspect-Oriented Software Development," Christina von Flach G. Chaves and Carlos J.P. de Lucena
- [8 Tucker 1997] "The Computer Science and Engineering Handbook," Allen B. Tucker, CRC Press, Inc. 1997, pg 2296, Figure 106.2
- [9 Kiczales2001] "Getting Started with Aspectj," Gregor Kiczales et al., Communications of the ACM, October 2001, Vol. 44, No. 10
- [10 Kiczales et. al] "An Overview of AspectJ," Gregor Kiczales, et al., Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP). Spring 2001
- [11 Lieberherr et. al] "Preventive Program Maintenance in {Demeter/Java} (Research Demonstration)," Karl J. Lieberherr and Doug Orleans, International Conference on Software Engineering, ACM Press, 1997
- [12 Kiczales] Gregor's ontology presentation/paper
- [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] "Philosophy in the Flesh," Georgy Lakoff, Mark Johnson, 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>