

DAJ User's Guide

by John J. Sung

Introduction

The integration between AspectJ and Demeter concepts came about as my Master's Thesis topic. We wanted to gain a better understanding of how the concepts applied in AspectJ and Demeter would interact. Importantly, we wanted to understand the fundamental concepts applied and their interactions.

These concepts and their interactions are important in analyzing the tools based on these concepts and in planning a directed development of these tools. In this analysis, we should be able to determine the source of the expressive power of the tools, types of problems it solves and potential complications.

Installation

Requirements

DAJ was developed with and compatible with java, aspectj and demeterj. The specific versions used are listed below.

- java sdk 1.3.1 or later from www.java.sun.com
- aspectj 1.0 or later from www.aspectj.org
- rt.jar of demeterj 0.8.4 or later from www.ccs.neu.edu/research/demeter/DemeterJava

Downloading and unpacking DAJ

Download the Latest version of DAJ from <http://www.ccs.neu.edu/research/demeter/DAJ>. You should use your browser to download the jar file. Then, use the "jar xf daj[version].jar" to unpack the DAJ in the desired location. It will create a directory structure that looks like this:

```
daj[version]
  daj.jar
  basket
    BasketMain.java
    BasketMainCount.java
    BasketTraversal.trv
  ccg
    CreateClassGraph.java
```

For unix workstations, you might need to unpack the daj.jar file, because ajc might not be able to load the .jar file. At least, I wasn't able to get it to work without unpacking.

```
jar xf daj.jar
```

Setting the Environment

You need to add these paths to your class path:

[fullpath]/daj.jar or if unpacked: [fullpath]/daj[version]

Here's an example classpath variable on windows:

```
.;d:\development\daj\daj1.0Alpha\daj.jar;d:\development\javacc\javacc1.1\JavaCC.zip;  
d:\development\Demeter\demeterj-0.8.4\demeterj.jar;d:\development\Demeter\demeterj-  
0.8.4\rt.jar;c:\jdk1.3\lib\dt.jar;c:\jdk1.3\lib\tools.jar;  
d:\development\aspectj\aspectj1.0\lib\aspectjrt.jar;d:\development\aspectj\aspectj1.0\lib\aspec  
tjtools.jar;
```

You also need to make sure that your environment is setup for java, aspectj, and demeterj's runtime library. You do not actually need the demeterj.jar in your path to run DAJ, but I used demeterj to implement DAJ. Thus, it is in my classpath.

If you are a frequent user of these tools, it might be a good idea to setup scripts to setup the environment variables given a full path to these tools. This will facilitate upgrading of tools and other software development activities. I have a tree of script calls that will setup the environments that I tend to work frequently in and this strategy has allowed me to manage my environment variables effectively.

Running DAJ

To run DAJ you need to pass DAJ as the class to execute the main method. DAJ requires 2 options, -ccg and -main. -ccg is the option to specify the location of the CreateClassGraph.java file in the ccg directory. This file is compiled with the user code to generate the implementation of the traversals specified within the trv files. The -main option specifies the class that contains the main method for the user code. This class is passed to the java vm in the traversal generation process.

The DAJ command line specification:

```
java edu.neu.ccs.demeter.daj.DAJ -ccg CreateClassGraph.java -main Main [options]
[java files] [trv files]
```

java files - files with .java extension that contains AspectJ code or pure java code.

trv files - files with .trv extension that contains the traversal specification. See Traversal Files Format section for syntax and semantics of the traversal specification.

Required

-ccg [java location] - Location of the CreateClassGraph.java file

-main [class name] - The class name of the class that contains the static main method

Optional

-gpa - This option generates printing advice for each of the traversals that outputs a trace of the calls to the traversal methods.

-dtrv [directory path] - Outputs the generated traversal files to the specified directory.

-ajc "ajc compiler" - The string containing the actual command line for the ajc compiler. This is used for using other than "ajc" for the compilation.

-ajtdebug - Printout the debug information for the traversal generation phase. This output can be very large, however it does print out the actual traversal graph and the code generated for each traversal. This can be useful in debugging your strategy.

To run the Basket Example, you would type this:

```
java edu.neu.ccs.demeter.daj.DAJ -gpa -ajc "ajc.bat" -ccg
d:\development\daj\dev\ccg\CreateClassGraph.java -main
BasketMain -dtrv trav BasketMain.java BasketMainCount.java
BasketTraversal.trv
```

This line is also in basket.bat.

The Basket Example

Compiling and Running the Basket Example on Windows

This example run of the basket example is for windows. There is a basket.bat batch script file that will run the example.

```
>basket.bat
```

```
Compiling Basket Example
```

```
%I - Generating Stubs
```

```
prefix for stub: trav
```

```
Generating Stub file: trav/BasketTraversal.java
```

```
%I - traversal generation compilation
```

```
ajc.bat d:\development\daj\dev\ccg\CreateClassGraph.java
```

```
BasketMain.java BasketMainCount.java trav/BasketTraversal.java
```

```
%I - traversal generation
```

```
java BasketMain -gpa -d trav BasketTraversal.trv
```

```
%I - traversal compilation
```

```
ajc.bat BasketMain.java BasketMainCount.java
```

```
trav/BasketTraversal.java
```

```
Running the Basket Example
```

```
call(void Basket.t1())
```

```
call(void Basket.t1_crossing_f())
```

```
call(void Fruit.t1())
```

```
call(void Fruit.t1_crossing_w())
```

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

```
call(void Basket.t1_crossing_f2())
```

```
call(void Fruit.t1())
```

```
call(void Fruit.t1_crossing_w())
```

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

```
Total weight of basket = 15
```

```
call(void Basket.t2())
```

```
call(void Basket.t2_crossing_f())
```

```
call(void Fruit.t2())
```

```
call(void Fruit.t2_copy1_crossing_w())
```

```
call(void Weight.t2_copy1())
```

```
call(void Basket.t2_crossing_f2())
```

```
call(void Fruit.t2())
```

```
Total weight2 of basket = 5
```

Compiling and Running the Basket Example on Unix

This example was ran on a Solaris machine, but it should work for other Unix machines.

```
> java edu.neu.ccs.demeter.daj.DAJ -gpa -ccg
../ccg/CreateClassGraph.java -main BasketMain -dtrv trav
BasketMain.java BasketMainCount.java BasketTraversal.trv
```

```
%I - Generating Stubs
prefix for stub: trav
Generating Stub file: trav/BasketTraversal.java
```

```
%I - traversal generation compilation
ajc ../ccg/CreateClassGraph.java BasketMain.java
BasketMainCount.java trav/BasketTraversal.java
```

```
%I - traversal generation
java BasketMain -gpa -d trav BasketTraversal.trv
```

```
%I - traversal compilation
ajc BasketMain.java BasketMainCount.java
trav/BasketTraversal.java
```

```
> java BasketMain
call(void Basket.t1())
call(void Basket.t1_crossing_f())
call(void Orange.t1())
call(void Fruit.t1_crossing_w())
call(void Weight.t1())
call(void Basket.t1_crossing_f2())
call(void Fruit.t1())
call(void Fruit.t1_crossing_w())
call(void Weight.t1())
Total weight of basket = 15
call(void Basket.t2())
call(void Basket.t2_crossing_f())
call(void Orange.t2())
call(void Fruit.t2_copy1_crossing_w())
call(void Weight.t2_copy1())
call(void Basket.t2_crossing_f2())
call(void Fruit.t2())
Total weight2 of basket = 5
```

The Explanation of the Basket Example

The Basket Example has been written to illustrate a typical use of DAJ. There are three files to this example, `BasketMain.java`, `BasketTraversal.trv`, and `BasketMainCount.java`.

`BasketMain.java` - contains code for the class graph shown in Figure 1.

`BasketTraversal.trv` - contains the traversal specifications

`BasketMainCount.java` - contains an aspectj interpretation of a visitor that counts the weights of the fruit within a `Basket`.

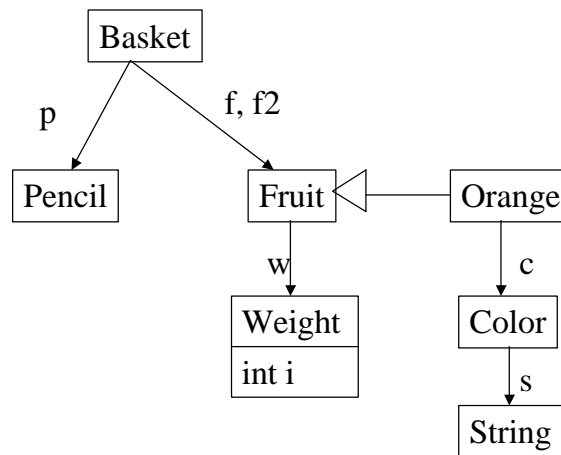


Figure 1 : Class Graph (UML Diagram) of the Basket Example

The class definitions in `BasketMain.java`, shown below in File 1, defines the "schema" of the data that we'll be working with. The actual instance is the object graph instantiated in the `main()` method. The methods `totalWeight1()` and `totalWeight2()` are introduced within `BasketMain.Count.java`. They start the traversals defined within `BasketTraversal.trv` and count the numbers stored within the `Weight` class.

File 1: `BasketMain.java`

```
// The "Basket" example to illustrate an simple usage of
// DAJ.
// The class graph:
// has-a with a label: -label->
// is-a          : -<:-
// Basket -p->Pencil
//          -f->Fruit -w->Weight -i->int
//          \-<:- Orange -c->Color -s->String
//
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; }
    }

    class BasketMain {
        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 totalWeight = b.totalWeight1();
            System.out.println("Total weight of basket = " +
            totalWeight);

            totalWeight = b.totalWeight2();
            System.out.println("Total weight2 of basket = " +
            totalWeight);
        }
    }
}

```

The `BasketTraversal.trv`, File 2, contains the definitions for traversal methods `t1()` and `t2()`. The first class graph declaration declares a default class graph. This is the class graph defined by the classes within the "current running program." This means that anything that you have defined without packages will be in the class graph. The second class graph statement defines a class graph slice, or a subgraph, using the strategy passed to it. The strategy defines a traversal that does not pass through and edge with `java.lang.String` as the destination. It means that the traversal will stop when it encounters `java.lang.String` class.

File 2: BasketTraversal.trv

```

// traversals for basket
aspect BasketTraversal {
    ClassGraph default;
    ClassGraph myClassGraph = new ClassGraph(default, "from
Basket to * bypassing {->*,*,java.lang.String}");
    declare traversal t1(myClassGraph) : "from Basket to
Weight";
    declare traversal t2(myClassGraph) : "from Basket via
Orange to Weight";
}

```

The third and fourth statements define traversals t1 and t2 using class graph slice that we have defined. The traversal t1 defines a traversal that starts from Basket to all reachable Weight objects, while traversal t2 defines a traversal that traverses to Weight only through the Orange Object.

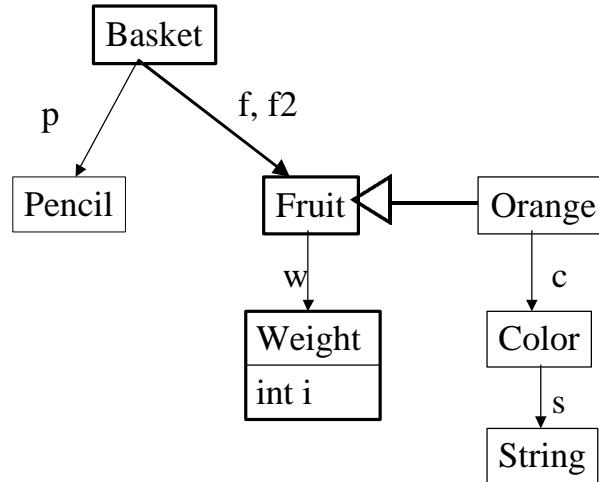


Figure 2: Traversal Graph for t1 with strategy "from Basket to Weight"

The traversal graph for t1 is highlighted in Figure 2. As you can see, the traversal graph is a subgraph of the original Class Graph. The traversal generation algorithm knows how to handle the inheritance edges such that it behaves logically, i.e. if there's an Orange object, it also treats it as a Fruit object.

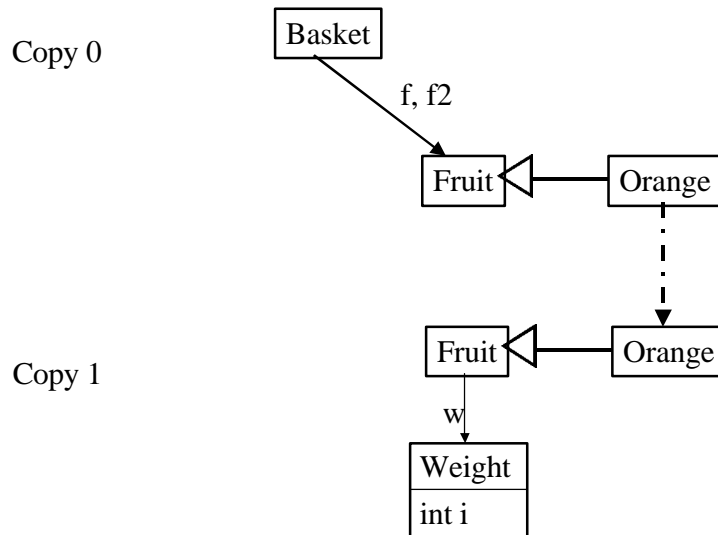


Figure 3: Traversal Graph for t2 with strategy "from Basket via Orange to Weight"

For the traversal t2 with strategy, "from Basket via Orange to Weight" illustrated in Figure 3 is more complicated. It is because of the via statement. During the traversal, we must ensure that we have traversed to an instance of Orange before getting to Weight. To do this, Demeter creates two copies of the graph, copy 0 for the traversal that has not encountered an instance of Orange and copy 1 that has encountered an instance of Orange.

When the traversal encounters a Fruit object, the traversal is stopped. When the traversal encounters the Orange object, itself a Fruit object, it "jumps" to copy 1 and then continues the traversal to Fruit then to Weight. This is how Demeter handles this case and DAJ generates code that follows this algorithm.

In the BasketMainCount.java file, the methods totalWeight1() and totalWeight2() are introduced to the BasketMain class. It's defines an aspect that is used to define the behavior for counting the weights stored within a Basket object. The introduced methods initialize the value to return to zero, call the traversal method and then returns the returnVal. The actual work of addition occurs within the before advice. The pointcut for the before advice is used for both t1 and t2 traversals.

File 3: BasketMainCount.java

```
// the aspect for counting the total weight of the basket
aspect BasketMainCount {
    static int returnVal;

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

    int Basket.totalWeight2() {
        returnVal = 0;
        t2();
        return returnVal;
    }

    pointcut t2WeightPC(Weight weight) : (call(* *t2*()) ||
call(* *t1*())) && target(weight);
    before(Weight weight) : t2WeightPC(weight) {
        returnVal += weight.get_i();
    }
}
```

In Demeter terminology, BasketMainCount.java is a visitor that traverses traversals t1 and t2. The visitor counts the integers stored within the Weight objects that it encounters during the traversal. Notice how the advice can be used for both traversal without modification. The pointcuts would have to change if we only wanted it applicable for only one traversal instead of two.

Traversal File Format

The traversal file format is designed such that it is syntactically similar to AspectJ. The idea is that this extension to AspectJ could be integrated into the compiler in the future. Thus, it uses aspect declaration, declare key words, etc. within it. In the traversal File, one can declare class graph slices and traversals. DAJ takes these and generates AspectJ code that implements the traversal.

The ClassGraph declaration can have two different forms, default or a class graph slice. The default declaration is the ClassGraph object that is obtained from CreateClassGraph using DJ. It has the form:

```
ClassGraph cg1;  
ClassGraph defaultCG;  
ClassGraph oneMoreDefaultCG1;
```

The Class Graph Slice form of the Class Graph declaration looks like a DJ Class Graph constructor with a Class Graph and a strategy.

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

Notice that the Class Graph Slice form of the Class Graph declaration needs to match the previous declaration of the ClassGraph with the first argument of the ClassGraph constructor. This is designed to make this look very much like DJ code.

The Traversal Declaration has two forms as well, a default traversal and a traversal with class graph as an argument. The default traversal basically implements the traversal with the default class graph and has the form:

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

The traversal with class graph specification takes in a class graph and produces the traversal with it. It has the form:

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

The last language feature is the aspect declaration. This is designed to look like AspectJ code. If interested in AspectJ, check out www.aspectj.org. The aspect declaration should contain a list of ClassGraph declarations and traversal declarations. Here is an example of an aspect declaration.

```
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 *";  
}
```

A file should contain one and only aspect declaration and the name of the aspect should match the file name before the .trv extension. DAJ will give an error and exit if this is not the case.

Warning! Strategies must have a valid class name for the start of the traversal specified by the string after the keyword "from".

Compilation Process

The compilation process that DAJ executes has four steps listed below.

- Generation of Stubs based on the traversal
- Compilation of CreateClassGraph.java, AspectJ files and Traversal Files
- Execution of the compiled files with the class specified by the -main option. Traversal implementation in AspectJ code is generated using DJ.
- Compilation of AspectJ Files with generated traversal implementation.

During the generation of stub files, the traversal files with extension .trv is processed. A file with the same name, but with .java extension is created and a method for the traversal is introduced for the class that is the starting point for the traversal specified. This is needed for the first compilation of the AspectJ code for traversal generation.

Introduction to Demeter Concepts

This section is for developers that are not familiar with Demeter. We will introduce the basic concepts behind Demeter and explain how Demeter tools DemeterJ and DJ apply these concepts.

Demeter deals with the data and algorithm crossing cutting concerns. Because data and algorithm are used everywhere, the scope in which these concepts may be applied is very large. The data is the Class Graph and the algorithm is further separated into Strategy and Visitor. The way in which these are defined within DemeterJ, DJ, and DAJ differ but the concept and the metaphor used are the same.

The Class Graph can be viewed as a schema that defines all possible Object Graphs, i.e. an instance of the Class Graph. It has enough information to allow us to traverse the Object Graph. Under the journey metaphor, Class Graph is in effect a map that we can follow to carry out the Strategy.

The Strategy is nothing more than instructions on how to traverse a subgraph of the Class Graph. Strategies like, "from Basket to Weight" and "from Company to Salary" are instructions in how the traversal should be conducted. Minimally, a Strategy defines a start node and an end node. However, more complex Strategies can include or exclude nodes or specific edges. For example, the Strategy "from Basket via Orange to Weight" instructs the visitor to traverse from Basket going through Orange to Weight. Thus, this strategy requires the node Orange to be traversed before reaching Weight. Here's an example that excludes a specific edge, "from Company to Salary bypassing ->Subsidiary,Europe,Division." It says to traverse from Company to Salary but not going through the edge that connects Subsidiary and Division together by label Europe. The nodes are Classes and the label is the name of the data member in Demeter.

Now that we have defined the map that we can follow and the Strategy that we can follow to find the right path in our journey, we need to define what happens during that journey. All we have done so far is to define the route in which we'll be traversing. This is where Visitors come into play. Visitors define advices on what to do before, after or around nodes during the traversal. The bodies of these advices are like ordinary methods, except that it's passed the node that it's currently visiting. Thus, you may do some operation on these nodes.

The operations defined by the advices are executed before we pass through the node, after we have visited the node (on our return trip), or instead of (around) continuing the trip. You can think of all before advices being executing during the trip going from the starting node to destination node and the after advices being executed during the return trip from destination to the starting node. This is a more general form of head recursive functions and tail recursive functions. It's more general, because we can recurs through many different type of nodes instead of just one type, i.e. the function itself. The around method is like a detour in the journey, it allows some amount of control over the traversal during runtime. For example, it might be desirable to stop the current traversal, continue the traversal or only traverse down a subset of the paths available.

Class Graph

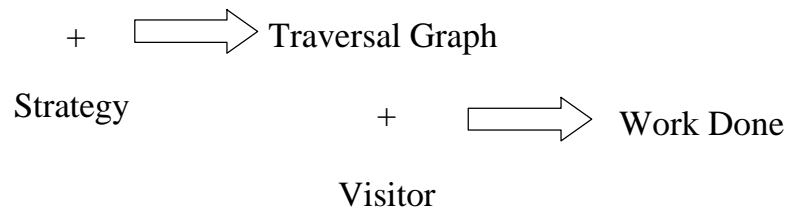


Figure 4: DemeterJ Process

The DemeterJ Process in Figure gives an overview of how the Class Graph, Strategy and Visitor are used together to get work done. The Traversal Graph shown is a subgraph of the Class Graph defined by applying the Strategy on that Class Graph. The interesting thing to note is that the definition of these components is ambiguous. The set of nodes and edges defined by the Class Graph, Strategy and Visitor can be disjoint, thus these components are independent of each other. The Strategy might yield an empty Traversal Graph or the Visitor defines advices for none of the nodes within the Traversal Graph, yielding no work.

References for DAJ Developers

DAJ - <http://www.ccs.neu.edu/research/demeter/DAJ>

- The DAJ User Manual
- The PowerPoint Presentation Tutorial
- Master's Thesis Proposal

DemeterJ - <http://www.ccs.neu.edu/research/demeter>

- DJ Fact Sheet & Resources
- DemeterJ Fact Sheet & Resources
- AP Library Fact Sheet & Resources
-
- "Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns", Karl Lieberherr, PWS Publishing Company, Boston, 1996.

AspectJ - <http://www.aspectj.org>

- Documentation > index > AspectJ Tutorial
- Documentation > prog guide
- Documentation > dev guide

Java - <http://www.java.sun.com>

- Docs & Training > Java2 Standard Edition > J2SE 1.3 API English