Master’s Thesis Defense: Aspectual Concepts

John J. Sung
Outline

- Motivation
- JPM and Demeter Concepts
- Demeter integrated with AspectJ (AJ)
- AJ Semantics/Syntax
- AJ System Architecture
- Conclusion
Motivation

- Aspect Oriented Software Development Gaining Momentum
  - 1st International Conference on AOSD
  - AspectJ, DemeterJ, ConcernJ, HyperJ, etc.

- AOSD is Relatively Young
  - Demeter C++ ~1989
  - DemeterJ ~ 1996
  - AspectJ ~1997
Questions

• What are the fundamental concepts behind AOSD?
• How are these concepts related?
• How can these concepts be combined?
• What are the fundamental problems that software language features attempt to solve?
Join Point Model (JPM)

• Fundamental Concepts
  – Join Point, Pointcut, Advice, Aspect, Introduction
  – Concepts applied to call graph in AspectJ
  – Introduction also applied to class graph
  – Construction Metaphor
Construction Metaphor

• Concepts in JPM create an image of structures constructed from building blocks
AspectJ Abstraction of Programs

Program

Call Graph + Pointcuts + Advices
AspectJ Abstraction of Programs

Class C1
b1.foo();

within(Bar)

Class Bar
int x;

Aspect A
before() : within(Bar) {
    foo()
    x = 0;
    if (x < 0)
        throw KException
}
Another View

AspectJ

Advice

pointcut

advice body

join point

Java Program
Demeter

• Fundamental Concepts
  – Class Graph, Strategy, Visitor, Advice
  – Concepts applied to data structures, i.e. class graph
  – Journey Metaphor
Journey Metaphor

- Concepts in Demeter create an image of a person on a journey
Demeter Abstraction of Programs

Program

Class Graph + Strategy Graph + Visitor Advices
Demeter Abstraction of Programs

Class Graph  Strategy Graph  Traversal Graph  Visitor Advices

before(..)  after(..)  before(..)
Combining Concepts

• Application of JPM to Demeter
  – Class graph is built of classes and edges
  – Join points become the points along traversals
  – Advices are executed at these join points
Combining Concepts

- Application of Demeter to JPM
  - CPU executes the traversal through the dynamic call graph
  - CPU manages the dynamic call graph
  - Advices are for the one visitor: CPU
Demeter AspectJ:  

- Extends AspectJ with traversal capability
- Defines a traversal specification language
- Uses DJ to generate Class Graph, Traversal Graph
- Generates the AspectJ implementation of the traversal using AspectJ introductions
- Implemented using DemeterJ, DJ, AspectJ
- Blending of JPM and Demeter Concepts
Design of DJA

- **First Design**
  - Add syntax to AspectJ language
  - Modify the open source ajc
  - Use DJ and AspectJ to modify the behavior

- **Second Design**
  - Use ajc as the back end
  - Create a small language to describe traversals
  - Use DJ, AspectJ, DemeterJ
  - Less coupling with the AspectJ compiler
Traversal Specification Language

• Designed to have similar syntax to DJ and AspectJ
• Allows users to specify Class Graph, Traversal, and Visitor
• Generates AspectJ implementation of traversals using introductions
• Visitor advices are called by using AspectJ advices
Class Graph Specification

• Default Class Graph
  – ClassGraph cgvar;

• Class Graph Slice
  – ClassGraph cgvar = new ClassGraph(cg, "strategy");
Visitor Specification

- Uses Java Reflection to obtain method signatures
- Recognized Methods
  - around, before, after, start, finish, returnValue
- Visitor Declaration
  - Visitor visitorClass;
Traversal Specification

- Default Traversal Specification
  - declare traversal tvar : "strategy";

- Traversal with Class Graph
  - declare traversal tvar(cgvar) : "strategy";

- Traversal with Visitor
  - declare traversal tvar(cgvar, visitorvar) : "strategy";
aspect aspectName {
    class graph declarations;
    traversal declarations;
    visitor declarations;
}
What DAJ Generates

• For each default and class graph slice traversal
  – method void tvar() for the source node of the traversal strategy
• For each traversal with visitor
  – method void tvar() for the source node of the traversal strategy
  – method void tvar(visitorClass) for the source node of the traversal strategy
  – Appropriate AspectJ advices for each advice method in visitorClass
A Simple Basket Example

```java
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; }
}
```
A Simple Basket Example

Basket
  p
  f, f2
  Pencil
  f

Fruit
  w
  Weight
  int i

Orange
  c
  Color
  s
  String
class BasketVisitor {
    int total;

    public void start() {
        total = 0;
    }

    public int returnValue() {
        return total;
    }

    void before(Weight w) {
        total += w.get_i();
    }
}
aspect BasketTraversal {
    ClassGraph default;
    ClassGraph myClassGraph =
        new ClassGraph(default, "from Basket to *");
    Visitor BasketVisitor;
    declare traversal t1(myClassGraph, BasketVisitor) :
        "from Basket to Weight";
    declare traversal t2(myClassGraph, BasketVisitor) :
        "from Basket via Orange to Weight";
}
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());

        BasketVisitor bv = new BasketVisitor();
        b.t1(bv);
        int totalWeight = bv.returnValue();
        System.out.println("Total weight of basket = " + totalWeight);

        b.t2(bv);
        totalWeight = bv.returnValue();
        System.out.println("Total weight2 of basket = " + totalWeight);
    }
}
Generated Code for Visitor

```java
static BasketVisitor t1_visitor;
public void Basket.t1(BasketVisitor v) {
    t1_visitor=v;
    t1_visitor.start();
    t1();
}
before(Weight host) :
    call(public void t1*()) && target(host) {
    t1_visitor.before(host);
}
void Basket.t1() {
    t1_copy0();
}
```
// traversal t1 : {source: Basket -> target: Weight} with { }
public void Basket.t1_copy0(){
  if (f != null) t1_copy0_crossing_f();
  if (f2 != null) t1_copy0_crossing_f2();
}
public void Basket.t1_copy0_crossing_f() { f.t1_copy0();}
public void Basket.t1_copy0_crossing_f2() { f2.t1_copy0();}
public void Fruit.t1_copy0(){
  if (w != null) t1_copy0_crossing_w();
}
public void Fruit.t1_copy0_crossing_w() { w.t1_copy0();}
public void Weight.t1_copy0(){
}
public void Orange.t1_copy0(){
  super.t1_copy0();
}
pointcut pointcut_t1() : call(public void t1*());
before () : pointcut_t1 () {
  System.out.println(thisJoinPoint);
}
System Architecture

- **DAJ Main**
  - Parses command line arguments
  - Manages the DAJ code generation phases

- **Stub Generation**
  - Generates stubs for traversal methods

- **Traversal Generation Compilation**
  - Compiles using ajc the stubs, user code, and CreateClassGraph.java
System Architecture

- Traversal Generation
  - Uses DJ, Java Reflection, and DemeterJ to generate AspectJ traversal code
- Traversal Compilation
  - Compiles the generated traversal code with user code
System Architecture

- DAJ Main
  - Stub Generation
    - Method Generation
  - Traversal Generation
    - Shell
  - Traversal Compilation
    - Shell
  - Traversal Compilation
DAJ Process

Traversals Files

Stub Generation

Generated Stubs

Traversal Generation

Compilation

Traversal Implementation

User Code

Code Provided by DAJ

Intermediate Files

Final Application

CreateClassGraph.java

Class Files

Class Files

Final Application
Traversals Generation

- CreateClassGraph.java
  - intercepts the call to main
  - instantiates a ClassGraph using DJ
- Arguments and ClassGraph
- Uses DJ to generate Traversal Graph
- Translates the Traversal Graph to AspectJ introductions
- Visitor code generation
  - uses Java Reflection to obtain method signatures
  - generates AspectJ advices for each visitor advice
Example Run of DAJ

[denali: ~/demeter/daj/dev/basket2] > java
edu.neu.ccs.demeter.daj.DAJ -main BasketMain -dtrv trav -ccg
../ccg/CreateClassGraph.java BasketMain.java BasketTraversal.trv
BasketVisitor.java

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

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

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

%I - traversal compilation
ajc BasketMain.java BasketVisitor.java trav/BasketTraversal.java
Measuring Performance

- Traversals implemented in DemeterJ, DJ, DAJ
- Vary the size of the Object Graph
- Measure the elapsed time between start and finish
- SunBlade 100 running SunOS 5.8
Performance Comparison

![Graph showing performance comparison between DemeterJ, DJ, and DAJ. The x-axis represents Object Graph Size ranging from 5000 to 20000, and the y-axis represents Elapse Time (ms) ranging from 0 to 20000.]
DemeterJ and DAJ Performance

![Graph showing the performance comparison between DemeterJ and DAJ]
Performance Comparison

<table>
<thead>
<tr>
<th>OG Size</th>
<th>DemeterJ</th>
<th>DJ</th>
<th>DAJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>23</td>
<td>3577</td>
<td>32</td>
</tr>
<tr>
<td>10000</td>
<td>40</td>
<td>7908</td>
<td>48</td>
</tr>
<tr>
<td>15000</td>
<td>55</td>
<td>11863</td>
<td>62</td>
</tr>
<tr>
<td>20000</td>
<td>60</td>
<td>14905</td>
<td>70</td>
</tr>
</tbody>
</table>
Future DAJ Improvements

- [http://www.ccs.neu.edu/research/demeter/DAJ](http://www.ccs.neu.edu/research/demeter/DAJ)
- Two syntax supported
  - DJ/Java like style
  - AspectJ’s declare syntax
- Handle Collections
  - Traversal through Java Collections
- Handle interfaces
  - Generates Traversals correctly for interfaces
Questions

- What are the fundamental concepts behind AOSD in Demeter and AspectJ?
- How are these concepts related?
- How can these concepts be combined?
- What are the fundamental problems that software language features attempt to solve?
Software Design Concerns

- Organizational Concern
  - How do I organize my code?

- Factorizational Concern
  - How can I factor similar code?

- Specification Concern
  - What can I specify within the program?

- Interface Concern
  - How can I interface with someone else’s code?
Conclusions

- Metaphors used in Demeter and JPM
  - Construction and Journey Metaphors
  - Usage of one or the other depends on the users and application
  - Can describe each other
  - Combinable
Conclusions

- **DAJ**
  - Mixing Demeter with AspectJ concepts
  - Uses DJ, DemeterJ and AspectJ
  - Extends AspectJ incrementally
  - Faster traversals than DJ
  - [www.ccs.neu.edu/research/demeter/DAJ](http://www.ccs.neu.edu/research/demeter/DAJ)
Conclusions

- Software Design Concerns
  - Organizational, Factorizational, Specification, and Interface
  - How do programming features address SDC?
  - How do SDCs related to AOP?
  - Further research needed
Future Direction

• Analyze the metaphors used in other tools
  – ComposeJ, HyperJ, etc.
• Try combining these concepts
• Use the Software Design Concerns to analyze AOP tools
AspectJ from PARC

(Demeter AspectJ)
References

- www.ccs.neu.edu/research/demeter/DAJ
- www.ccs.neu.edu/research/demeter/Demeter Java
- www.ccs.neu.edu/research/demeter/DJ
- www.aspectj.org
- www.aosd.net
Static Scattering and Tangling

- aspect$_i$ is scattered across many classes (i = 1,2,3)
- class X tangles aspects 1, 2 and 3

Class diagram:

- class A consisting of three aspects
- Adding to classes
Dynamic Scattering and Tangling

- Each aspect (colors) is scattered across many classes (shapes).
- Class \( \bigcirc \) tangles all three aspects.

At those calls the aspect enhances the behavior.
**Demeter**

**subcomputation** = join points related to traversing through the objects guided by traversal specification and class graph.

<table>
<thead>
<tr>
<th></th>
<th>AspectJ</th>
<th>DJ</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kind</strong></td>
<td>Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td><strong>On What</strong></td>
<td>Dynamic call graph of base program</td>
<td>Dynamic call graph of a <strong>subcomputation</strong> of base program</td>
</tr>
<tr>
<td><strong>When</strong></td>
<td>Pointcuts</td>
<td>Signatures of visitor methods</td>
</tr>
<tr>
<td><strong>What</strong></td>
<td>Before / around / after advice</td>
<td>Before / around / after visitor method body</td>
</tr>
</tbody>
</table>
EMETER

Pointcut
- set of execution points of any method, ...
- rich set of primitive pointcuts: this, target, call, ... + set operations
- where to enhance

Advice
- how to enhance

Visitor method sig.
- set of execution points of traversals
- specialized for traversals (nodes, edges)
- where to enhance

Visitor method bodies
- how to enhance

From DJ to AspectJ